Set System Oracles for Subtrajectory Clustering

Franken, Thomas T.P.

Award date:
2021

Set System Oracles for Subtrajectory Clustering

Version 1.2

Master's Thesis, Eindhoven University of Technology

Thomas T.P. Franken
August 17, 2021

Supervisors:
Kevin Buchin, Eindhoven University of Technology
Anne Driemel, University of Bonn

Abstract

We study the problem of Subtrajectory Clustering, which seeks to summarize a selection of trajectories by partitioning them between a set of clusters, where each cluster has a center curve and the trajectories in a cluster are similar to the center curve of the cluster. The goal is to use as little center curves as possible for the partitioning. More specifically, we study the variant which does not seek to partition but to cover the selection of trajectories, for which a reduction to Set Cover is possible. To facilitate this reduction, a Set System Oracle is required, which contains the information on which center curves cover what part of the selection of trajectories. We take a Set System and use it to create new approaches for the creation of a Set System Oracle. We analyse the limitations of the approaches, as well as how to use them for more general Set Systems than the one we consider. As a subproblem, we study the problem of Reachability in Free Space Grids and conduct a literature study to solve that subproblem. Lastly, we also implement a proof of concept implementation and test its boundaries.

Contents

1 Introduction

In essence, trajectories are recorded movement, like the movement of cars on roads, migratory birds, random walks, the movement of football players on a field, and more. To do this, trajectories have not only spatial coordinates but also a temporal coordinate to define the direction of the movement.

It is often useful to get patterns out of movements; while maps can be made from the roads themselves, movement data can be helpful in distinguishing the common roads from less-used roads, leading to a map with the less important information filtered away. Similarly, to consider general flight patterns of birds one has to summarize them, as it is the broad strokes which are interesting, not the individual lines. Neither is there always meaning in individual steps of a football player; what's interesting are the overall movements of the football player. To get these patterns, one has to somehow summarize the trajectories representing the movements.

The problem of subtrajectory clustering deals with summarizing trajectories. It tries to find *center curves* which summarize the entire curve in less detail, but completely. To decide whether a curve is summarized by a candidate center curve, a distance measure is needed, for which the Fréchet distance is used, which deals with the distance between trajectories during traversal. See the preliminaries for more on the Fréchet Distance.

In a paper by Akitaya, Brüning, Chambers and Driemel [4], the authors take a different approach: instead of dividing the subtrajectories by center curves, they instead want to *cover* the subtrajectories by the center curves. To this end, they used Greedy Set Cover and transformed the problem to a set system for which they computed minimum-sized Set Covers, or approximations of minimum-sized Set Covers (as the problem is NP-hard). See the Related Work section for more information.

In this thesis, we have investigated the creation of Set System Oracles for the specific case where the trajectories are polygonal, have vertices and the Discrete Fréchet Distance is used with these vertices. We have made a naïve Set System Oracle creation method and some Set System Oracle creation methods which utilizes Free Space Grids, during the creation of which we have investigated the boundaries of using similar methods.

1.1 Short Overview of Related Work

Other than the related work discussed in section 3 (which includes a literature study of two papers on reachability oracles and the paper by Akitaya, Brüning, Chambers and Driemel [4] mentioned above), another approach that uses Greedy Set Cover and the Discrete Fréchet Distance to solve the Subtrajectory Clustering problem is the paper "Subtrajectory Clustering: Models and Algorithms", by Agarwal, Fox, Munagala, Nath, Pan and Taylor [1], where the authors introduce the concept of *pathlets*, which is a sequence of points derived from trajectories which is not necessarily a subtrajectory of any one trajectory. The authors then make a new model for subtrajectory clustering with these pathlets, where a subtrajectory clustering is a set of pathlets along with an assignment of subtrajectories to pathlets, and defines Pathlet Cover, which also relies on a reduction to Set Cover. They also give an $O(\log m)$-approximation algorithm for pathlet-cover in a time depending on the amount of candidate pathlets b and the total amount of vertices m.

A third approach using Greedy Set Cover can be found in the paper "Group Diagrams for Representing Trajectories", by M. Buchin, Kilgus, Kölzsch [8]. The approach creates a single graph out of all input trajectories, with the edges being represented by subtrajectories of the input and vertices of the original trajectories being combined in vertices if the vertices are within a distance measure. From this graph, the method is to find edges to be cluster representatives (all edges correspond to subtrajectories), with which Set Cover is run to find the minimal set of these clusters which covers the entire input.

Some further approaches for subtrajectory clustering without a focus on Set Cover we studied during our literature study were:

- In the paper "Detecting Commuting Patterns By Clustering Subtrajectories", by K. Buchin, M. Buchin, Gudmundsson, Löffler and Luo [7], the authors give an approximation algorithm for finding the longest subtrajectory cluster with the continuous Fréchet distance, for one or multiple trajectories, also using Free Space Graphs and Free Space Grids. This is then used in the paper "Clustering Trajectories for Map Construction", by K. Buchin, M. Buchin, Duran, Fasy, Jacobs, Sacristán, Silveira, Staals and Wenk [5], where the authors use the Fréchet Distance to create maps from travel data. Their algorithm first uses subtrajectories to construct bundles, which are dependent on the amount of subtrajectories, the maximum length of the subtrajectories in the bundle and the pairwise distance (Fréchet distance) between the subtrajectories. Then, the bundles are applied to the map greedily. In the paper "Improved Map Construction using Subtrajectory Clustering", by K. Buchin, M. Buchin, Gudmundsson, Hendriks, Sereshgi, Sacristán, Silveira, Sleijster, Staals and Wenk [6], the authors follow up on the last paper and improve that method.
- In the paper "Trajectory Clustering: A Partition-and-Group Framework", by Lee, Han and Whang [12], the authors first partition all trajectories into line segments and then group those segments. This grouping is done based on density clusters based on a distance function based on perpendicular distance, parallel distance and angle distance between two line segments.

Additionally, we also considered the paper "Jaywalking Your Dog: Computing the Fréchet Distance with Shortcuts", by Driemel and Har-Peled [10], which makes use of Free Space Diagrams (see Definition 6). In the paper, the authors compute the Fréchet Distance (see Definition 4) in cases where it is allowed to take a shortcut between vertices of one of the trajectories.

1.2 Acknowledgements

During the development of the implementation, we have made use of the Chicago Map Trajectory Data from the paper of Ahmed, Karagiorgou, Pfoser and Wenk [2, 3]. Our thanks for the use of that dataset, without which the implementation would never have been what it is now.

1.3 Overview

The goal is to investigate how to compute a Set System Oracle for a Set System to be used for Greedy Set Cover to use while solving Subtrajectory Covering.

In this thesis the focus is on the Discrete Fréchet Distance and polygonal trajectories with vertices with two spatial coordinates. We will define the problems in more general terms, but assume the use of the Discrete Fréchet Distance and polygonal trajectories with two spatial coordinates per vertex when using those problems. Additionally, we focused on solving the problem for a Set System which takes all possible subtrajectories of the trajectories as center curves.

The rest of this thesis will first have a section on preliminaries and a summary of three papers we have studied in our literature study, after which the goals will be further discussed in Section 4. For the main goal, a naïve method to create a Set System Oracle is created using dynamic programming in Section 5. After this section, we will discuss the results we have constructed with the use of Free Space Grids, after which we will discuss the implementation and present some test data.

2 Preliminaries

2.1 Trajectories

We consider trajectories to be defined as follows:

Definition 1 (Trajectories In General)**.** *A* Trajectory *P is a parameterized curve, so for every time coordinate $t \in [0, 1]$, $P(t)$ is a point in space. The time coordinate can also be referred to as its direction or orientation.*

This thesis limits itself to 2-dimensional polygonal trajectories:

Definition 2 (2-dimensional Polygonal Trajectories)**.** *A 2-dimensional polygonal trajectory P has n vertices with indices $p_1, ...p_n$, with each vertex with index p_i denoted by the coordinates $(x_{p_i}, y_{p_i}, t_{p_i})$, where $x_{p_i} \in \mathbb{R}$ and $y_{p_i} \in \mathbb{R}$ are the spatial coordinates and $t_{p_i} \in \mathbb{R}$ is the temporal coordinate. As a shorthand, we denote the vertex with index p_i as "vertex p_i" or "$P[p_i]$".*

From this point on, any mention of trajectories refers to trajectories conforming to definition 1.

Trajectories can also have subtrajectories:

Definition 3. *A* subtrajectory *R of P is a trajectory consisting of vertices with indices $p_a, .., p_b$ of P, with $a \leq b$. An alternative notation is "$P[p_a, p_b]$", where p_a and p_b are the indices of vertices in P.*

2.2 The Fréchet Distance

The Frèchet Distance is a distance measure defined as follows:

Definition 4 (The Fréchet Distance, [10])**.** *Fréchet distance $\boldsymbol{d}_{\mathcal{F}}(X, Y)$, where X and Y are general trajectories, is defined as:*

$$\boldsymbol{d}_{\mathcal{F}}(X, Y) = \inf_{f:[0,1]\to[0,1]} \max_{t\in[0,1]} \|X(f(t)) - Y(t)\| \tag{1}$$

where f is an orientation preserving (so $f(0) = 0$ and $f(1) = 1$) continuous surjective function over time.

In words, we have a certain distance (see next sentence) defined over all mappings, and we choose the mapping for which that distance is the smallest. The distance itself is then the distance between the vertices of X and Y that are paired in the mapping of the time vertices.

The classic explanation is as follows:

If you have a dog on a leash walking over curve Y, and you walk over curve X, then the Frèchet distance is the smallest possible length of the leash you need to have when traversing X and Y if you and your dog are able to change walking speed or stop, but not go backward.

The Frèchet distance complies with the triangle inequality, so $\mathbf{d}_{\mathcal{F}}(X, Z) \leq \mathbf{d}_{\mathcal{F}}(X, Y) + \mathbf{d}_{\mathcal{F}}(Y, Z)$.

In a special case of the Fréchet distance, the Discrete Fréchet Distance, only the distances between the vertices of the curve are considered.

The Discrete Fréchet Distance is defined using couplings. A coupling C of the vertices of F, S is a sequence of pairs of vertices $C = \langle C_1, ..., C_k \rangle$ with $C_r = (f_i, s_j)$ for all $r = 1, ..., k$ and some $i \in \{1, ..., n\}, j \in \{1, ..., m\}$, fulfilling

- $C_1 = (f_1, s_1)$ and $C_k = (f_n, s_m)$
- $C_r = (f_i, s_j) \Rightarrow C_{r+1} \in \{(f_{i+1}, s_j), (f_i, s_{j+1}), (f_{i+1}, s_{j+1})\}$ for $r = 1, ..., k-1$

Definition 5 (Discrete Fréchet Distance, [7])**.** *Let F, S be two polygonal curves in $\mathbb{R}^c$. Then the discrete Fréchet distance $\boldsymbol{d}_{d\mathcal{F}}(F, S)$ is defined as*

$$\boldsymbol{d}_{d\mathcal{F}}(F, S) = \min_{coupling\ C} \max_{(f_i, s_j)\in C} \|f_i - s_j\|,$$

where C ranges over all couplings of the vertices of F and S.

The Fréchet Distance between two polygonal curves can be expressed using a *free space diagram* [7]:

Definition 6 (Free Space Diagram[7])**.** *The Free Space Diagram of two polygonal curves F and S, with n and m vertices, respectively, is the set*

$$F_\Delta(F, S) = \{(i, j) \in [1, n] \times [1, m] : \boldsymbol{d}_{Euclid}(\Phi_F(i), \Phi_S(j)) \leq \Delta\}$$

where Φ_F denotes a mapping from $[1, n] \to \mathbb{R}^c$ (for any dimension c) that maps $i \in \{1, ..., n\}$ to $f_i \in F$ and maps all other points $i \in [1, n]$ to points on the edges between vertex $f_{\lfloor i \rfloor}$ and $f_{\lceil i \rceil}$. Additionally $\boldsymbol{d}_{Euclid}(x, y)$ denotes the Euclidian distance between x and y.

It was proven by Alt & Godau that the Fréchet Distance between F and S is less than d if and only if there is a monotone path in the Free Space Diagram $F_d(F, S)$ from the location in the Free Space Diagram denoted by (f_1, s_1) (the bottom left) to the location in the Free Space Diagram denoted by (f_n, s_n) (the top right) (adapted from the paper by K. Buchin, M. Buchin, Gudmundsson, Löffler and Luo[7]; we were not able to access the paper by Alt & Godau). A monotone path in the Free Space Diagram is a continuous path through the Free Space Diagram that does not decrease its coordinates: Let a and b be any two points on the path and let a come earlier than b. Then $a.x \leq b.x$ and $a.y \leq b.y$.

Free Space Grids are the Discrete Fréchet Distance equivalent of Free Space Diagrams as used with the Continuous Fréchet Distance. They consists of the mn gridpoints, with a gridpoint $(f_i, s_j) \in F \times S$ existing if $\mathbf{d}_{Euclid}(f_i, s_j) \leq \Delta$. Every gridpoint (f_i, s_j) has at most three outgoing transitions, depending on the existence of its neighbours: one to (f_{i+1}, s_j), one to (f_i, s_{i+1}) and one to (f_{i+1}, s_{i+1}). In Free Space Grids, similar to the property of Free Space Diagrams, the Discrete Fréchet Distance between 2-dimensional polygonal curves F and S is less than Δ if and only if there is a monotone path in the Free Space Grid $FSG_\Delta(F, S)$ from gridpoint (f_1, s_1) to (f_n, s_n). In this context, a monotone path in the Free Space Grid is a path where any gridpoint (f_x, s_y) in the path is followed by either gridpoint (f_{x+1}, s_y), $(f_x, s + y + 1)$ or (f_{x+1}, s_{y+1}).

2.3 Subtrajectory Clustering

The problem of Subtrajectory Covering is related to the problem of Subtrajectory Clustering. To define both problems, it is useful to define what a Subtrajectory Cluster is:

Definition 7 (Subtrajectory Cluster [7])**.** *Given a trajectory Q with n vertices, a* subtrajectory cluster $C_Q(m, \ell, \Delta)$ *for Q of length ℓ consists of at least m distinct subtrajectories $Q_1, ..., Q_m$ such that the time intervals for two subtrajectories overlap in at most a vertex, the distance between the subtrajectories and Q is at most Δ and at least one subtrajectory has length ℓ.*

A trajectory can also be called a curve.

The definition of a Subtrajectory Cluster leads to the problem of Subtrajectory Clustering:

Problem (Subtrajectory Clustering)**.** *Given a trajectory T with n vertices and potential requirements on m (the amount of subtrajectories per cluster), ℓ (the length of subtrajectories in the cluster), Δ (the distance between the curves in the cluster) or a combination of the three, what is the smallest size of a set of subtrajectory clusters C needed such that the entire trajectory is divided between subtrajectory clusters?*

This problem can be also be asked as a decision problem ("does there exist a subtrajectory clustering with [requirements on m, ℓ or Δ]?"). Additionally, shortcuts and multiple trajectories can also be taken into account in versions of this problem.

2.4 Set Cover

The idea is to use Set Cover to solve Subtrajectory Covering, for which we define the notion of a Set System and Set Cover here. We also give the Greedy Approximation Algorithm we use.

2.4.1 Problem Definition

Definition 8 (Set System, [4])**.** *Let X be a set. We call a set $\mathcal{R}$, where any $r \in \mathcal{R}$ is of the form $r \subseteq X$ a set system with ground set X.*

Problem (Set Cover Problem, [4])**.** *Let $\mathcal{R}$ be a set system with ground set X. A set cover of $\mathcal{R}$ is a subset $S \subset \mathcal{R}$ such that the ground set is covered by the union of the sets in S. The set cover problem asks to find a set cover for a given $\mathcal{R}$ of minimum size.*

The decision version of this problem (does a covering exist with size at most k) is NP-complete. [9]

2.4.2 The Greedy Approximation Algorithm [9]

The Greedy Approximation Algorithm works by picking at each iteration the set in the set system that covers the greatest number of uncovered remaining elements. It runs in polynomial time and has an approximation ratio of $H(\max|S| : S \in \mathcal{R}$, where $H(d)$ is the dth harmonic number, or $\sum_{i=1}^{d} 1/i$.

Algorithm 1 Greedy Set Cover Algorithm [9]

```
1: procedure GreedySet(X, R)
2:     U = X
3:     C = ∅
4:     while U ≠ ∅ do
5:         select an S ∈ R that maximizes |S ∩ U|
6:         U = U − S
7:         C = C ∪ {S}
8:     end while
9:     return C
10: end procedure
```

3 Related Work - Literature Study

In this section, we will summarize our literature study of three specific papers more in-depth. The first paper we will discuss is the paper "Covering a Curve with Subtrajectories", by Akitaya, Brüning, Chambers and Driemel [4], as this thesis is directly related to the paper.

After that, we shall discuss two papers we found on the subject of Reachability in depth, as those are the best answers we found for Problem 1 in Section 4 when trying to solve that problem via Free Space Grids. While we did not implement either of the two approaches discussed in the paper, we take that the method from the paper "Compact Oracles for Reachability and Approximate Distances in Planar Digraphs" by Thorup [14] has been implemented as base for our theoretical analysis.

3.1 Covering a Curve with Subtrajectories [4]

In this paper, by Akitaya, Brüning, Chambers and Driemel [4], the authors consider the problem of covering an input curve with a small set of representative curves, where every portion of the input curve is similar to some representative under the Fréchet distance. To this end, they split the curve P along m breakpoints with real values $t_1, ..., t_m$ into $m-1$ subtrajectories. They also use a parameter ℓ, which stands for the amount of vertices a representative curve Q can have, and Δ, which stands for the Fréchet distance limit. They then want to cover the entire curve. For the set-system $\mathcal{R}$ the ground set X is defined as the line segments between consecutive vertices. Each set $r_Q \in \mathcal{R}$ is defined by a polygonal curve Q and contains all break points z which are the start of a line segment *covered* by Q. A subtrajectory is *covered* by Q if the Fréchet distance is less or equal than Δ.

The goal is to compute minimum-size set covers. The paper then studies how to find a set C of size k of representative cluster centers with ℓ vertices. The cost function for such a set is the maximum Fréchet distance of any interval covered by its representative curve for the smallest possible set of intervals.

Unfortunately this problem is NP-Hard. However, a 3-approximation is achievable by using the triangle inequality on the Fréchet distances between curves and simplifications of curves. The set system used for this 3-approximation is denoted by $\tilde{\mathcal{R}}_0$. This set system can then be used with the Greedy Set Cover algorithm.

The paper also adapted a set cover algorithm by Brönniman and Goodrich, where they define a set system $\tilde{\mathcal{R}}_2$. In this set system, every break point z has two points $x_z \leq z \leq y_z$ where x_z is the break point with the smallest index such that $d_F(\tau_{x_z,z}, P[t_{x_z}, t_z]) \leq 4\Delta$, where $\tau_{x_z,z}$ is the straight line between x_z and z, and y_z is the breakpoint with the greatest index such that the Fréchet distance between the straight line between z and y_z and the subtrajectory between z and y_z is less than 4Δ.

A set $r_{i,j} \in \tilde{\mathcal{R}}_2$ is then defined as all points z for which there exists a subtrajectory starting at $x \in [x_z, z]$ and ending at $y \in [z, y_z]$ for which the simplification gained by connecting the vertices is within Fréchet distance 2Δ from the straight line between vertices i and j.

For this set system, the paper makes an oracle with constant query time. To do that, first all points z get their x_z and y_z computed, after which an $m \times m$ matrix is computed to tell us for a pair of vertices i and z whether there is an x with $x_z \leq x \leq z$ such that $\|P(t_x) - P(t_i)\| \leq 2\Delta$ in constant time. This can then be used to find whether point x exists by checking whether $M(i, x_z) \leq z$ and whether point y exists by checking whether $M(j, z+1) \leq y_z$. To find whether $z \in r_{i,j}$, the following queries need to be answered:

1. $M(i, x_z) \leq z$
2. $M(j, z+1) \leq y_z$
3. $\|s - P(t_z)\| \leq 2\Delta$ with s being the bisector between the points $P(t_z)$, $P(t_{z+1})$ and the line segment $\tau_{i,j}$.

If any of these is "no", the answer is "no".

3.2 Reachability

3.2.1 Compact Oracles for Reachability and Approximate Distances in Planar Digraphs [14]

For this paper, by Thorup [14], we focused on studying the reachability part of the paper. In this part, the author creates an oracle for reachability queries that, for a graph of n vertices takes $O(n \log n)$ time and space to create and can answer queries in constant time. Below, the method used will be summarized, following the original sections of the paper.

Part 1 - Getting 2-layered graphs from the original graph The first part of the method consists of decomposing the original graph G into 2-layered digraphs $G_0, ..., G_{k-1}$. To that end, they introduce the following definitions and a lemma:

Definition 9 (t-layered Digraph [14])**.** *A digraph H is* t-layered *iff it has a t-layered spanning tree T*

Definition 10 (t-layered Spanning Tree [14])**.** *A* t-layered Spanning Tree *T is a disoriented rooted spanning tree such that all paths from the root in T consists of at most t dipaths in H, where a dipath is a directed path.*

Lemma 1 (Decomposing to Layers [14])**.** *A digraph G can, in linear time, be transformed into digraphs $G_0, ..., G_{k-1}$ such that:*

1. *The total number of edges and vertices in $G_0, ..., G_{k-1}$ is linear to the number of edges and vertices in G.*
2. *Each vertex v has an index $\iota(v)$ such that a vertex w is reachable from v in G iff it is reachable from v in $G_{\iota(v)-1}$ or in $G_{\iota(v)}$.*
3. *Each G_i is a 2-layered digraph.*
4. *G_i is a minor of G, so G_i is obtained from G by deletion of edges and vertices and contraction of edges. If G is planar, so is G_i*

They then prove the above lemma by defining a method to decompose the graph G for which the resulting digraphs satisfy the lemmas. They take the parent graph G and then create layers, starting with a single source vertex as L_0, as follows:

1. Every odd layer consists of the previous layer and all vertices that can reach the previous layers.
2. Every even layer consists of the previous layer and all vertices that it can reach.

Then, for every two consecutive layers L_i and L_{i+1}, a subgraph G_i is created from the induced graph of L_{i+1}, with all preceding layers $L_{<i}$ together compressed to a single source vertex. Afterwards, a 2-layered spanning tree T_i can be created for each subgraph G_i by having all edges of L_i facing away from the source and all edges from L_{i+1} facing towards the source if i is odd and the other way around if i is even. This method satisfies all conditions of Lemma 1 [14]. The method is illustrated on a Free Space Grid in Figure 1, with different sources: Figure 1a has the lower left vertex as source, Figure 1b has some vertex in the Free Space Grid as source and Figure 1c has the top right vertex as source.

Part 2 - Undirected Planar Spanning Tree Separation Next, the method throws away all direction for a given 2-layered digraph H with 2-layered spanning tree T, but only for this part. The idea of this part of the method is to find separators to decompose the graph in a tree-like fashion, which the paper does as follows:

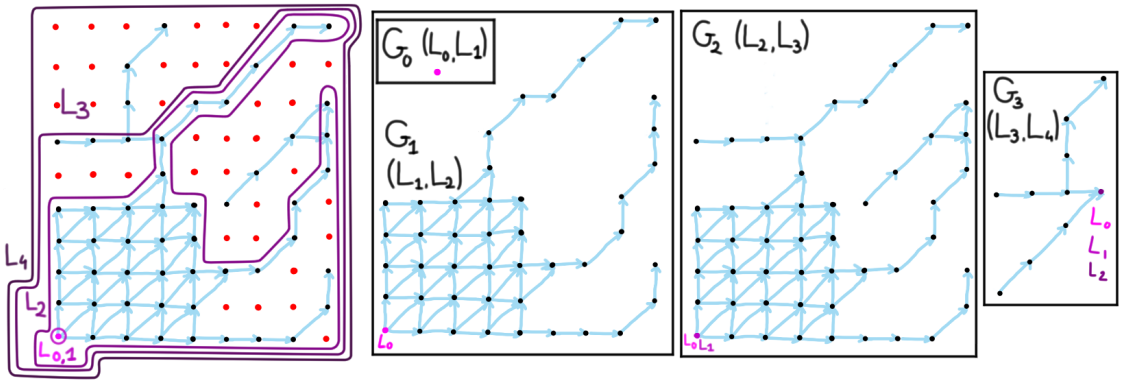
1. Triangulate the graph H to get H^{Δ} and put it on a sphere.
2. Choose a triangle Δ. This is now the outer face of the sphere.
3. Using the spanning tree T, create cycles with the edges of Δ. These are *fundamental cycles*. The side of the cycle that does not contain Δ is considered the inside of the cycle. If one of the edges of Δ is contained in T, the inside of its fundamental cycle is considered empty.
4. If one of the three fundamental cycles of Δ has more than half of the edges of H strictly inside, the triangle is flipped along the edge of Δ which created that fundamental cycle.
5. If this is not the case, the directed paths making up the parts of the fundamental cycles contained in T are used as separator dipaths in the recursion of Part 4.

The author proves that this terminates, for more information, see the paper [14].

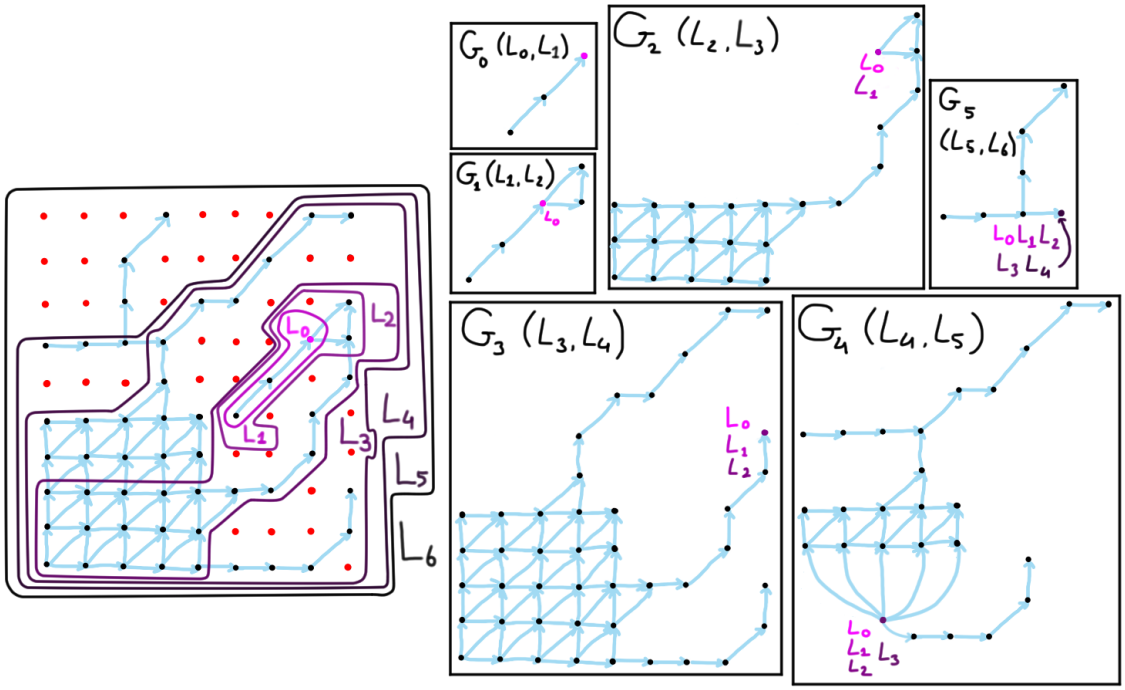
Part 3 - Reachability via dipath. This part of the method states that there is a dipath from vertex u to vertex w intersecting dipath Q iff u connects to a vertex a in Q and w connects from a vertex b in Q where $a \leq b$ in Q. It is also established that all connections between vertices in H and dipath Q can be established in linear time.

Part 4 - The Basic Recursion. At this point parts 1-3 are combined into a recursive method, starting from a graph G:

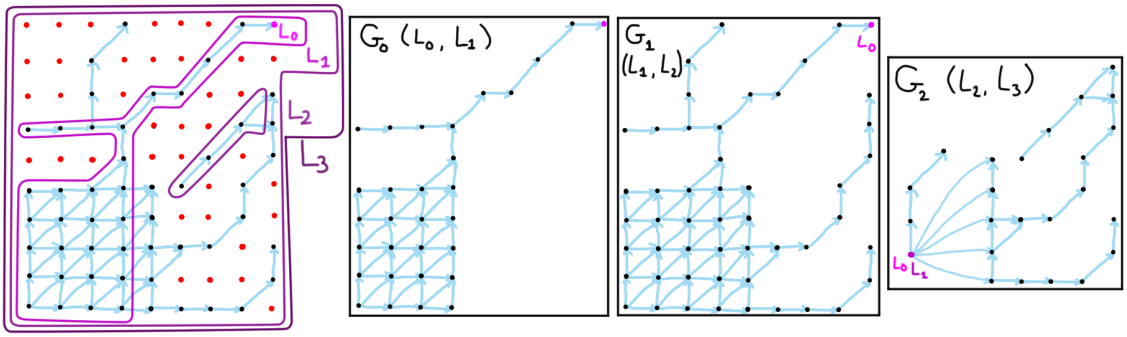
1. Reduce the graph to a set of 2-layered digraphs as in part 1. Then, for every 2-layered digraph H and spanning tree T:
   (a) Use part 2 to get a separator S, which contains 6 dipaths from H. For every dipath $Q \in S$:
      i. Get all connections between vertices of H and Q. For each vertex $v \in H$, create two arrays: $\texttt{to}_v$ and $\texttt{from}_v$, which are indexed by the separator dipaths so that $\texttt{to}_v[Q]$ and $\texttt{from}_v[Q]$ contain the numbers in Q of the vertices to which v connects and from which v connects, respectively. If v does not connect to Q, then $\texttt{to}_v[Q] = \infty$ and if v does not connect from Q, then $\texttt{from}_v[Q] = -1$.



(a) The decomposition into layers and the resulting subgraphs if the lower left vertex is taken as source.



(b) The decomposition into layers and the resulting subgraphs if some random vertex is taken as source.



(c) The decomposition into layers and the resulting subgraphs if the top right vertex is taken as source.

Figure 1: The decomposition into layers and the resulting subgraphs illustrated on a Free Space Grid with multiple sources. The figure is based on methods from the paper by Thorup [14].

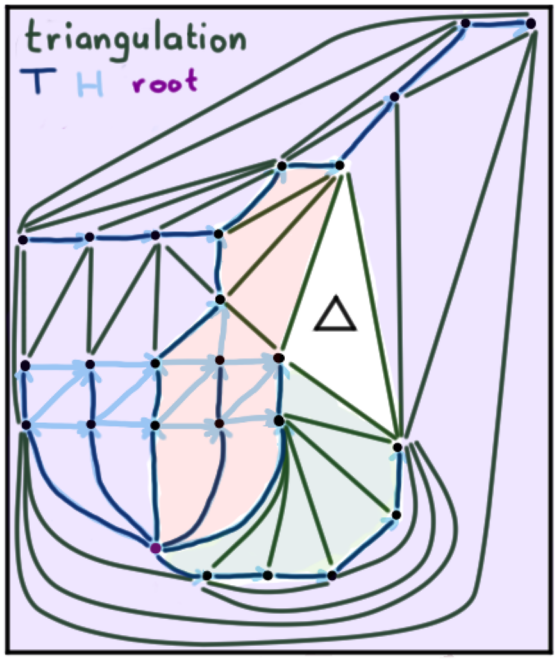


Figure 2: The Separation illustrated on a 2-layered digraph from Figure 1. Note that the green edges and the edges denoting the borders between the colours do not count for step 3 of the method. (The triangulation is not complete to keep the picture clear). The figure is based on methods from the paper by Thorup [14].

ii. Contract S to the new root r^S and give it weight 0. Recurse to step 1a with all components of $H \setminus V(S)$. Recursion depth is $O(\log n)$.

Querying these arrays for reachability information comes down to a constant time check (does vertex u reach Q earlier than the vertex that connects to w from Q?) if you have the index of the right dipath Q. However, finding this dipath Q is harder. Therefore, every vertex v enumerates, in a fixed order, all dipaths in all ancestor separators until v itself is chosen for a separator, which is called the *final* call of v. To answer a reachability query between u and w, the goal is then to find the separator S separating u and w, which can be done by finding the nearest common ancestor in the enumerations of u and w.

Finding this nearest common ancestor takes $O(\log n)$ time naively. Using some preprocessing from the paper "Fast Algorithms for Finding Nearest Common Ancestors" [11], the nearest common ancestor can be found in constant time. The method works by creating, for a nearest common ancestor operation on a tree T, a balanced binary tree B existing of the vertices T and some other vertices and a compressed tree C. The nearest common ancestor in B is computed in constant time, converted to a nearest common ancestor in C in common time, which is then converted to the nearest common ancestor in T. For more information see the paper [11].

Once S is found, if there is a dipath Q in S or all $O(\log n)$ ancestor separators of S, for which $\texttt{to}_u[Q] \leq \texttt{from}_v[Q]$, u reaches w. It is necessary to check all $O(\log n)$ separators, as the separators are contracted. Therefore, if the path from u to w goes through the root we need to check an ancestor separator, as we do not know if the path from u reaches the path to w in the root.

Part 5 - Frames to reduce the query time In this part, the author reduces the query time from $O(\log n)$ to constant. This is done by using frames.

Instead of contracting the separators in the recursion, a set F of root paths to leaves of T separating H from the rest of G_i will be passed down, a so-called *frame* of H. The amount of

frames during the recursion is kept constant size by using *subgraph reducing* recursions and *frame reducing* recursions.

Both recursions work like the recursion described before: find a good separator and use that to divide $H + F$ into smaller parts. The subgraph reducing recursion assigns the edges of F a weight of 0, and the separator divides $H + F$ into parts with at most $\frac{H}{2}$ vertices. Frame reducing recursion then tries to find a separator S such that all components of $(H + F) \setminus V(S)$ have at most half the leaves of F. Using these two recursions, the amount of frame paths is kept constant. As every subgraph reducing recursion still halves the weight of the subgraph H, the logarithmic depth is still recursive.

To use these frames to answer reachability queries in G_i, all connections over G_i between H and dipaths in the frame F and separator S are needed. For a dipath Q, the connections between H and Q over G_i means all the vertices in H that connect to or from the first or last vertex in Q that v can reach in G_i.

The connections in G_i between H and the frame are passed down in the recursion. $H + F$ including these connections as edges is denoted as $H \star F$. The number of the extra edges is linear to the number of vertices in H. To further reduce $H \star F$ such that its size is linear in the number of edges in G_i incident to edges in H, the paper introduces *topological* vertices, which are endpoints and branching points of dipaths in F, and *selected* vertices, which are vertices in F that neighbour a vertex in H in $H \star F$. The reduction is to skip all vertices which are not topological or selected. Constructing the reduced frame is linear in $H \star F$.

To query the resulting structure and arrays to see whether a vertex u reaches a vertex w, again, the nearest common ancestor of the final call of u and w, C, needs to be found, which can be done in constant time, after which the indices to be checked lie within the separation number of C and the separation number of the parent of C, which is a constant amount, making the total query time constant.

Part 6 - A Pure Label-Based Implementation This part asserts that the whole oracle can be incorporated in a labeling scheme with $O(\log n)$-size labels.

3.2.2 An Efficient Data Structure for Lattice Operations [13]

In this second paper we focused on for Reachability, by Talamo and Vocca [13], the authors present a method applicable for Directed Acyclic Graphs satisfying the lattice property. The lattice property [13, Proposition 2.1] states that if and only if there exist four vertices, u, v, w, z with four paths with as endpoints $(u, w), (u, z), (v, w), (v, z)$, then there also exists an x and the four paths $(u, x), (v, x), (x, w), (x, z)$. Partial Lattices always satisfy this property. As a Free Space Grid is a DAG which satisfies the lattice property, the method of this paper is also applicable in our case.

The method works as follows:

Part 1 - Clusters A vertex c can have two clusters, $Clus^+(c)$ (which contains c and all its predecessors) and $Clus^-(c)$ (which contains c and all its successors. If there are two vertices c and v, then if the $+$-clusters have overlap I, then the least upper bound of I is contained in I. The same holds for the $-$-clusters and the greatest upper bound of I. Removing a cluster $Clus(c)$ from a DAG satisfying the lattice property results in a DAG satisfying the lattice property.

Part 2 - Double Trees With a cluster $Clus(c)$, the vertices in $V - Clus(c)$ are called the *external* vertices and the vertices in $Clus(c)$ are called the *internal* vertices. The set $Ext(Clus(c))$ contains all external vertices connected to at least one internal vertex.

To maintain connectivity information, each internal vertex u gets their own tree, a spanning tree of the graph induced by the set $Pred(u) \cap Clus^+(c)$ or $Succ(u) \cap Clus^-(c)$, depending which cluster $Clus(c)$ is. This internal tree for vertex u is denoted as $IntTree(u, c)$.

Every external vertex v has an internal representative, defined as $LUB(Clus^+(c) \cap Clus^+(v))$ or $GLB(Clux^-(c) \cap Clus^-(v))$, depending on which cluster $Clus(c)$ is. This leads to the insight

that $Ext(Clus(c)) = \bigcup_{u \in Clus(c)} Ext(u)$. Therefore, every internal vertex u has an external tree induced by u, $ExtTree(u, c)$, which is a spanning tree of the subgraph induced by $Ext(u)$, rooted at u. The external trees of $Clus(c)$ are pairwise disjoint.

To combine this in one handy structure, the *Double Tree* of internal vertex u in $Clus(c)$ is defined as $DT(u, c) = IntTree(u, c) \cup ExtTree(u, c)$. Each of these Double Trees is associated with a partial order, which means that x can only reach y if the coordinates from x are smaller than the coordinates from y.

Part 3 - Basic Decomposition Strategy This part shows an algorithm with the basic decomposition strategy. The strategy can be summarized as two methods:

1. a recursive Cluster Builder, which `chooses` the vertex to make a cluster from and then removes the cluster from the original graph.

2. a Double Tree Builder, which takes a cluster and builds the Double Trees of that cluster.

How they choose the vertex for the cluster is not yet defined.

The output is then a collection $\mathcal{C}$ of clusters and a collection $\mathcal{T}$ of all double trees. With $x \in V$, these have the following invariants:

1. x belongs only to one cluster $Clus(c_i)$.
2. given a cluster $Clus(c_i)$ with $x \notin Clus(c)$, there is only one $DT(u_{i,j}, c_i)$ which includes x as an external vertex, with some $u_{i,j} \in Clus(c)$.
3. given a cluster $Clus(c_i)$, each element of the collection $\{DT(u_{i,j}, c_i)\}$ is, by definition, univocally identified by the element $u_{i,j} \in Clus(c_i)$.

With this, the crux of the matter is reduced to finding the right double tree. To this end, the paper presents a data structure made up of three lookup structures:

A is a table indexed on the vertices of V and contains their clusters.

B is a set of tables indexed on the vertices of V and contains lookup tables which are indexed on the double trees and contain x's location in the double tree if $x \in V$ belongs to that tree or null otherwise.

C is a set of tables indexed on the vertices of V and contains lookup tables which are indexed on the clusters: if x is an external vertex of a cluster, it contains the double tree x belongs to and its location in it.

The query goes as follows, with $x \in Clus(c_i)$ and $y \in Clus(c_j)$:

1. if $Clus(c_i) = Clus(c_j)$, use **B** to get the double trees rooted at x and y. x can reach y if its coordinates are smaller than that of y in one of the double trees.

2. Else, look up the double tree in $Clus(c_i)$ which contains y in **C**, and look up x's position in this double tree in **B**. Also look up the variant where y and x's places are swapped. If in one of these double trees x has smaller coordinates than y, x can reach y.

This data structure answers queries in constant time. However, while `choose` is an undefined strategy, the space occupancy is $O(n^2)$. To reduce this to $O(n\sqrt{n})$, a better strategy needs to be applied.

Part 4 - A Better Strategy To improve the choice of cluster vertex, the normal vertex is split into three: fat, thin and good. Good vertices are vertices which have one cluster with a size between $\frac{\sqrt{n}}{4}$ and $\frac{\sqrt{n}}{2}$, fat if its $Clus^+$ size is bigger than $\frac{\sqrt{n}}{2}$ but all vertices in it have a $Clus^+$ size smaller than $\frac{\sqrt{n}}{4}$, or the same condition with $Clus^-$, and thin if it is neither fat nor good and has a cluster of size less than $\frac{\sqrt{n}}{4}$. First, the strategy looks for good vertices and adds its clusters to the result.

If there are no good vertices, it looks for fat vertices. Let x be this fat vertex, and let X be the first level of its $Clus^+$ spanning tree (w.l.o.g.). None of the vertices in X have a $Clus^+$

of size bigger than $\frac{\sqrt{n}}{4}$. Therefore, the strategy combines clusters from vertices in X until the combination is a good-sized cluster. Let $C(X)$ be the vertices in X in the new good-sized cluster. The strategy then introduces a dummy vertex d_C, after which is redirects the paths connecting all vertices in $C(X)$ to go through v.

If there are no fat vertices, the strategy takes thin vertices and makes a *cluster forest* out of their clusters, which are considered together. Thin clusters are added to the forest F until $|F| \geq \frac{\sqrt{n}}{4}$ or until $mk \geq n$, where $m = |Ext(F)|$, k is the amount of clusters in the forest and n is still the amount of vertices in the original graph.

The resulting clusters and cluster forests and double trees again have some invariants:

1. x belongs to only one cluster and to only one forest (if the cluster is not the forest).
2. In clusters to which x does not belong, x is present in at most one double tree.
3. given a cluster $Clus(c_{i,j})$, each double tree rooted in $u \in Clus(c_{i,j})$ of that cluster is univocally identified by u.

To use these new structures, the data structure from the last part needs to be augmented:

A Does the same thing as before, but also stores the identifier for the forest to which a vertex belongs, if the cluster is a part of a forest.

B is unchanged.

C is a new structure, a set of lookup tables, one for each vertex in V. For each forest F (note that a forest can also be a single cluster), if $x \in Ext(F)$, the data structure contains a pointer to structure **D**. Otherwise it contains a null value.

D is a new structure, a set of lookup tables associated with a vertex x first and then with forests for which a table exists if $x \in Ext(F)$. $\mathbf{D}(F_i)$ stores stores the identifier of the unique double tree to which x belongs and its position in it.

This data structure still has a constant query time, but the total time and space complexity is now $O(n\sqrt{n})$.

4 Goal and Problem Statement

The goal is to investigate means to compute a Set System Oracle to be used for Greedy Set Cover to use while solving Subtrajectory Covering, as said in the introduction, with a focus on the Discrete Fréchet Distance and polygonal trajectories. This can be expressed as the following question:

Main Question. *As seen in Section 3.1, the paper "Covering a Curve with Subtrajectories" by Akitaya, Brüning, Chambers and Driemel [4] seeks to cover a curve by creating a set system $\tilde{\mathcal{R}}_2$ out of the curve and its subtrajectories, working from a curve and a point within the Fréchet distance of that curve to get a path covered by the curve which includes the point. It then uses that set system and Greedy Set Cover to solve Subtrajectory Covering.*

Can we create a method for creating an efficient data structure to function as Set System Oracle for Greedy Set Cover?

Researching this question is the main goal of this thesis. With our focus on the Discrete Fréchet Distance and two-dimensional polygonal trajectories, the main problem of Subtrajectory Covering is defined as follows:

Main Problem (Subtrajectory Covering)**.** *Given a selection of m polygonal trajectories $\mathcal{S}$ with $O(n)$ vertices and a threshold Δ, what is the smallest size of a set of subtrajectory clusters C from a Set System $\mathcal{R}$ needed such that all m trajectories are covered by these subtrajectory clusters, where each subtrajectory cluster $C \in \mathcal{R}$ is defined by a center curve D from a set of center curves $\mathcal{D}$?*

We have chosen to focus on the Set System where the center curves are all possible subtrajectories from every trajectory. The Set System we try to make an Oracle for and the Oracle itself are then defined as follows:

Definition 11 (Set System and Set System Oracle)**.** *The Set System $\mathcal{R}$ to solve Subtrajectory Covering for the Discrete Fréchet Distance with Greedy Set Cover on m polygonal trajectories $\mathcal{S}$ with threshold Δ has a ground set X made up of all $O(mn)$ points in $\mathcal{S}$. A set $r \in \mathcal{R}$ is defined by a subtrajectory P from some trajectory in the selection $\mathcal{S}$ as follows:*

$$r_P = \{S[s_{i'}] \in S \mid S \in \mathcal{S}, \exists S[s_i], S[s_j] \in S, i \leq i' \leq j \text{ with } \boldsymbol{d}_{d\mathcal{F}}(P, S[s_i, s_j]) \leq \Delta\}$$

In which $S[s_{i'}]$ is the vertex with index $s_{i'}$ in S and $S[s_i, s_j]$ is the subtrajectory of S from index s_i to index s_j. (Note that as X is made up of all points in any trajectory in $\mathcal{S}$, $S[s_x] \in (S \in \mathcal{S})$ implies that $S[s_x] \in X$.)

In words, the set $r_P \in \mathcal{R}$ consists of all vertices with index $s_{i'}$ in any trajectory S in the selection for which there exist two other vertices in S with indexes s_i and s_j where $i \leq i' \leq j$ where P and $S[s_i, s_j]$ have a Discrete Fréchet Distance of less than Δ.

We call a vertex, subtrajectory or trajectory covered *by subtrajectory P if r_P includes the vertex or all vertices of the subtrajectory or trajectory.*

A Set System Oracle *for this Set System takes a subtrajectory P as query and returns the set r_P.*

For our methods, we will assume that the center curves P are polygonal, as we are still working with the Discrete Fréchet Distance. For development of the methods, the Set System above is what we use, also to analyse time and space constraints. Every method has a section that analyses possible expansion to more general Set Systems.

We discuss a Naïve Dynamic Programming Method to construct an oracle in Section 5.

As this is still a fairly broad question and problem, we decided to split the problem up into intermediate questions, of which the first one was about Reachability:

Question 1. *As an intermediate problem, can we find an efficient data structure for the following problem?*

Problem 1. *The data structure takes two trajectories F and S as input, which have to be preprocessed to answer queries which specify one subtrajectory P from trajectory F and one subtrajectory Q from trajectory S from each trajectory and wants to know*

a. whether the Fréchet distance between P and Q is less than some defined threshold Δ.

b. what the Fréchet distance between P and Q is.

or both.

As we focused on Free Space Grids, we investigated the problem of Reachability in Free Space Grids, of which results can be found in Section 6.2. In short, this problem is solved by the paper by Thorup [14] summarized in Section 3.2.

After that, we considered the next two intermediate questions in the context of Free Space Grids:

Question 2. *Can we use the insights or the results from the last problem to create an efficient data structure for the following problem?*

Problem 2. *The data structure takes two trajectories F and S as input, which have to be preprocessed to answer queries which specify one subtrajectory P from F and a vertex s_a from the S and asks for either*

a. some subtrajectory Q

b. the longest subtrajectory Q

where Q is a subtrajectory with start vertex s_a from the second trajectory for which the Fréchet distance to P is smaller or equal to a threshold Δ.

Question 3. *Can we use the insights or the results from the last problem to create an efficient data structure for the following problem?*

Problem 3. *The data structure takes two trajectories F and S as input, which have to be preprocessed to answer queries which specify one subtrajectory P from F and a vertex s_p from the second trajectory and asks for a subtrajectory Q which includes the vertex s_p from the second trajectory for which the Fréchet distance to P is smaller or equal to a threshold Δ.*

The results for these two problems can be found in Sections 6.3, 6.4 and 6.6.1.

After solving these problems the goal is to use their results to solve the original problem for our Set System and create a Set System Oracle out of them:

Question 4. *How can the data structure resulting from the last question be transformed into a Set System Oracle as described in Definition 11 to be used by Greedy Set Cover?*

The results for this question can be found in Sections 6.5 and 6.6.2. Afterwards, we considered the following question:

Question 5. *How can our methods be applied to more general Set Systems with a set $\mathcal{D}$ of d center curves and m trajectories to be covered?*

Our insights into this question can be found in Sections 5.4 and 6.7.

5 The Naïve Dynamic Programming Method

In this section, we describe a method to tackle the goal question straightforwardly. The naïve method also implicitly tackles problem 2, by skipping the second research question and bruteforcing the third research question in the process of solving the first research question. The method can be found in the file `ExtremelyNaiveQuerier.java` of the implementation.

5.1 The Method

5.1.1 Part 1

The first part of the method is to compute for all possible subtrajectories which other subtrajectories are within Fréchet distance of them. The resulting Data Structure solves Problem 1a. With some alterations, it can also be modified to save the Fréchet distance between every trajectory (if that Fréchet distance is not bigger than the threshold), which is problem 1b, but that is not implemented or included in the given algorithms.

In a bit more detail, the construction of this structure uses dynamic programming: Let the first trajectory be denoted as F with vertices indexed as $f_1, ..., f_n$ and let the second trajectory be denoted as S with vertices indexed as $s_1, ..., s_n$. Let the first subtrajectory from vertex f_a to vertex f_b with $f_a \leq f_b$ be denoted as $P = F[f_a, f_b]$. Let the second subtrajectory from vertex s_a to vertex s_b with $s_a \leq s_b$ be denoted as $Q = S[s_a, s_b]$.

The base cases are entries where either $f_a = f_b$ or $s_a = s_b$. Assuming w.l.o.g. that $f_a = f_b$ is the case, then the entry (f_a, f_a, s_a, s_b) is set to *true* if all vertices in the subtrajectory $S[s_a, s_b]$ are within Fréchet distance Δ from f_a.

The recursive cases are computed as follows: For every combination of subtrajectories $F[f_a, f_b]$ and $S[s_a, s_b]$, the entry (f_a, f_b, s_a, s_b) in the Data Structure is set to *true* if for any $f_a \leq f_i \leq f_b$ and $s_a \leq s_j \leq s_b$ the entries (f_a, f_i, s_a, s_j) and $(f_{i+1}, f_b, s_{j+1}, s_b)$ are both *true*. For this, it is important that the entries are filled in according to subtrajectory sizes.

Solving Problem 1b To also know what the Discrete Fréchet Distance between subtrajectories is, as asked by Problem 1b, the algorithm can be augmented as follows:

1. In the base cases, save instead of a value *true* or $false$ either the biggest distance between f_a and a vertex in $S[s_a, s_b]$ or -1.

2. In the recursive step, first set the entry for the current subtrajectories $F[f_a, f_b]$ and $S[s_a, s_b]$ to -1. Then, check for all combinations (f_i, s_j) with $f_a \leq f_i \leq f_b$ and $s_a \leq s_j \leq s_b$ whether the distance saved in the entries (f_a, f_i, s_a, s_j) and $(f_{i+1}, f_b, s_{j+1}, s_b)$ are both bigger than zero. If that is the case and the biggest entry is bigger than the current entry for $F[f_a, f_b]$ and $S[s_a, s_b]$, update the current entry with this biggest entry.

That way, instead of returning just whether a subtrajectory covers another subtrajectory, the resulting data structure returns the Discrete Fréchet Distance between the subtrajectories, or -1 if the Discrete Fréchet Distance between the subtrajectories is bigger than the threshold. This solves Problem 1b.

To also solve Problem 1b if the threshold is undefined, see section 6.2.4.

Note that these augmentations are not in the algorithm presented below, as they have not been implemented.

5.1.2 Part 2

To do Greedy Set Cover with this information, it is important to know how many vertices are covered by every subtrajectory as per problem 3, as that is the decision factor for the Greedy Set Cover. Therefore, for every subtrajectory, check all the vertices and save whether they are covered by the subtrajectory.

As Greedy Set Cover considers all possible sets in the set system, we have chosen to make this data structure preprocess-heavy instead of query-heavy, as every subtrajectory is queried regardless.

5.2 The Algorithm

In the Algorithm, $\mathcal{S}$ is the selection of trajectories. For Problem 1, that selection has 2 trajectories; the implementation does not have that assumption so neither shall the algorithm. See Algorithms 2, 3 and 4. Note that Algorithm 3 does not include the augmentations for querying the precise Discrete Fréchet Distance to reflect the implementation and that for Algorithm 4 the implementation uses some implementation-specific structures and that the version presented here is the generalised version (though they function the same).

Algorithm 2 Naive Dynamic Programming Method

```
1: procedure NaiveProb1Sol(S, Δ)
2:     Data Structure FréchetDists ← DoPartOne(S, Δ)
3:     Set System SetSystem ← DoPartTwo(S, Δ, FréchetDists)
4:     return SetSystem
5: end procedure
```

Algorithm 3 Naive Dynamic Programming Method Part 1

```
1: procedure DoPartOne(S, Δ)
2:     Initialize 6-dimensional matrix FréchetDists with every entry set to false.
3:     for {Trajectory F | F ∈ S} do
4:         for {Trajectory S | S ∈ S} do

                                        ▷ Do the base cases where the first trajectory is a single vertex
5:             for all vertices f_a in F in increasing order do
6:                 for all vertices s_a in S in increasing order do
7:                     for all vertices s_b in S in increasing order and with s_a ≤ s_b do
8:                         if (
9:                             (s_{b−1} < s_a or FréchetDists[F][S][f_a][f_a][s_a][s_{b−1}] = true)
10:                            and dist(f_a, s_b) ≤ Δ
11:                        ) {
12:                            FréchetDists[F][S][f_a][f_a][s_a][s_b] ← true
13:                        }
14:                    end for
15:                end for
16:            end for

                                        ▷ Do the base cases where the second trajectory is a single vertex
17:            for all vertices f_a in F in increasing order do
18:                for all vertices f_b in F in increasing order and with f_a ≤ f_b do
19:                    for all vertices s_a in S in increasing order do
20:                        if (
21:                            (f_{b−1} < f_a or FréchetDists[F][S][f_a][f_{b−1}][s_a][s_a] = true)
22:                            and dist(f_b, s_a) ≤ Δ
23:                        ) {
24:                            FréchetDists[F][S][f_a][f_b][s_a][s_a] ← true
25:                        }
26:                    end for
27:                end for
28:            end for

                                                    ▷ Do the other cases
29:            for all sizes ℓ_1 > 0 of subtrajectories of F in increasing order do
30:                for all starting vertices f_a of subtrajectories of size ℓ_1 of F do
31:                    for all sizes ℓ_2 > 0 of subtrajectories of S in increasing order do
32:                        for all starting vertices s_a of subtrajectories of size ℓ_2 of S do
33:                            for all (f_i, s_j) with f_a ≤ f_i ≤ f_{a+ℓ_1} and s_a ≤ s_j ≤ s_{a+ℓ_2} do
34:                                if (
35:                                    FréchetDists[F][S][f_a][f_i][s_a][s_j] = true and
36:                                    FréchetDists[F][S][f_{i+1}][f_{a+ℓ_1}][s_{j+1}][s_{a+ℓ_2}] = true
37:                                ) {
38:                                    FréchetDists[F][S][f_a][f_{a+ℓ_1}][s_a][s_{a+ℓ_2}] ← true
39:                                }
40:                            end for
41:                        end for
42:                    end for
43:                end for
44:            end for
45:        end for
46:    end for
47:    return FréchetDists
48: end procedure
```

Algorithm 4 Naive Dynamic Programming Method Part 2

```
1: procedure DoPartTwo(S, Δ, FréchetDists)
2:     Initialize 5-dimensional matrix SetSystem with every entry set to false.
3:     for {Trajectory F | F ∈ S} do
4:         for all possible starting vertices f_a ∈ F do
5:             for all possible ending vertices f_b ∈ F, with f_a ≤ f_b do
6:                 for {Trajectory S | S ∈ S} do
7:                     for all vertices p ∈ S do
8:                         for all subtrajectories S[s_a, s_b] ∈ S with s_a ≤ s_p ≤ s_b do
9:                             if (
10:                                FréchetDists[F][S][f_a][f_b][s_a][s_b] = true
11:                            ) {
12:                                SetSystem[F][f_a][f_b][S][p] ← true
13:                                break
14:                            }
15:                        end for
16:                    end for
17:                end for
18:            end for
19:        end for
20:    end for
21:    return SetSystem
22: end procedure
```

5.3 Discussion of method

As can be seen by the many for-loops in these algorithms, this method is rather slow. If we take the amount of trajectories as m and the largest trajectory size in the selection as n, then the running time of Part 1 is $O(m^2 \cdot n^2 \cdot n^2 \cdot (f_b - f_a) \cdot (s_b - s_a)) = O(m^2 n^6)$, as $(f_b - f_a)$ and $(s_b - s_a)$ are both $O(n)$ and for every size of subtrajectory ℓ there are $n - \ell$ possible starting vertices of that subtrajectory. The running time of part 2 is $O(m \cdot n^2 \cdot m \cdot n \cdot n^2) = O(m^2 n^5)$. It therefore solves problem 1 (with 2 trajectories) in $O(n^6)$ time, giving a data structure with constant query time. A Set System Oracle for Greedy Set Cover is also made in $O(m^2 n^6)$ time in total. The space requirement is $O(m^2 n^4)$.

The recursive case of Part 1 cannot be done faster than in $O(n^2)$ per case, as whether subtrajectory 1 is within Fréchet Distance of subtrajectory 2 only needs one vertex combination (f_i, s_j) along which the subtrajectories can be split (such that 1a is within Fréchet Distance of 2a and 1b is within Fréchet Distance of 2b) to be assigned *true*, but if subtrajectory 1 is not within Fréchet Distance of subtrajectory 2, all vertex combinations (f_i, s_j) need to be checked.

Similarly, assigning *true* to s_p may go faster than $O(n^2)$, but assigning $false$ to s_p requires checking all $O(n^2)$ options.

Correctness of Algorithm 2 To prove Algorithm 2 correct, we will prove Algorithms 3 and 4 correct and then use that to conclude that Algorithm 2 creates a valid Set System Oracle for solving Subtrajectory Covering using Greedy Set Cover.

Lemma 2 (Naïve Dynamic Programming Method Part 1 Correctness)**.** *Let the Data Structure created in Part 1 be* `FréchetDists`*.* `FréchetDists` *has value true for subtrajectories $F[f_a, f_b]$ of any size and $S[s_a, s_b]$ of any size iff subtrajectory $F[f_a, f_b]$ covers subtrajectory $S[s_a, s_b]$.*

Proof. We prove the lemma by strong induction.

IH `FréchetDists` has value *true* for subtrajectories $F[f_a, f_b]$ of size x and $S[s_a, s_b]$ of size y iff the subtrajectory $F[f_a, f_b]$ covers subtrajectory $S[s_a, s_b]$.

Base If `FréchetDists` has value *true* for the subtrajectories $F[f_a, f_a]$ and $S[s_a, s_b]$, that means that for all vertices in subtrajectory $S[s_a, s_b]$, all vertices before that had a distance to vertex f_a of less than the threshold and the distance between vertex f_a and the current vertex is also less than the threshold. By definition, this means that $F[f_a, f_a]$ is within Discrete Fréchet distance of $S[s_a, s_b]$. This also holds for subtrajectories of the form $F[f_a, f_b]$, $S[s_a, s_a]$. If $F[f_a, f_a]$ covers $S[s_a, s_b]$, then all vertices in $S[s_a, s_b]$ have a distance of at most the threshold to vertex f_a. In this case, the algorithm will assign the corresponding cell in the matrix *true*.

Step Assume, by strong induction, that `FréchetDists` has correct values for all subtrajectory combinations where the first trajectory has size $< x$ and the second trajectory has size $< y$, per the induction hypothesis.

Assume `FréchetDists` has the value true for $F[f_a, f_b]$ and $S[s_a, s_b]$, while in reality the first trajectory does not cover the second trajectory. According to the algorithm, this means that there is a vertex f_i in $F[f_a, f_b]$ and a vertex s_j in $S[s_a, s_b]$ such that $F[f_a, f_i]$ covers $S[s_a, s_j]$ and $F[f_{i+1,f_b}]$ covers $S[s_{j+1}, s_b]$. Due to the Induction Hypothesis, we can assume this is indeed the case. But then there is a way to "walk" through $F[f_a, f_b]$ and $S[s_a, s_b]$ such that the distance is always less than the threshold: first you walk through the trajectories $F[f_a, f_i]$ and $S[s_a, s_j]$ satisfying the threshold. Then the walk goes simultaneously to vertices $f_{i+1} \in F$ and $s_{j+1} \in S$ and then you walk through $F[f_{i+1}, f_a]$ and $S[s_{j+1}, s_b]$ satisfying the threshold. Therefore $F[f_a, f_b]$ and $S[s_a, s_b]$ has a Fréchet distance of less than the threshold, which means the first trajectory must cover the second trajectory.

If the trajectory $F[f_a, f_b]$ covers the second trajectory $S[s_a, s_b]$, that means that there is a way to walk through the two lines satisfying the threshold requirement put on the walk. Specifically, there is a vertex f_i on the first trajectory and a vertex s_j within threshold distance of vertex f_i on the second trajectory such that $F[f_a, f_i]$ covers $S[s_a, s_j]$ and $F[f_{i+1}, f_b]$ covers $S[s_{j+1}, s_b]$. If not, then $F[f_a, f_b]$ does not cover $S[s_a, s_b]$, which contradicts the assumption. As these two subtrajectories exist and through the induction hypothesis we know that the algorithm has set the cells in the matrix for the two subcases to *true*, the algorithm will also set the cell in the matrix for subtrajectories $F[f_a, f_b]$ and $S[s_a, s_b]$ to *true*.

The Base Case and the Step Case hold, so the lemma holds and Algorithm 3 is correct. □

This can be used to prove algorithm 4 correct:

Lemma 3 (Naïve Dynamic Programming Method Part 2 Correctness)**.** *Let the Data Structure created in Part 2 be* `SetSystem`*.* `SetSystem` *has value true for query* $(F \in \mathcal{S}, f_a, f_b, S \in \mathcal{S}, s_p)$ *if and only if there exists a subtrajectory* $S[s_a, s_b]$ *of* $S \in \mathcal{S}$ *with* $s_a \leq s_p \leq s_b$ *which is within the Discrete Fréchet Distance of subtrajectory* $F[f_a, f_b]$ *of* $F \in \mathcal{S}$*.*

Proof. Let `FréchetDists` be the result of applying Algorithm 3 with the same selection of trajectories $\mathcal{S}$ and threshold Δ. As Algorithm 3 is correct, `FréchetDists` contains the value *true* for a query $(F \in \mathcal{S}, S \in \mathcal{S}, f_a, f_b, s_a, s_b)$ iff $F[f_a, f_b]$ is within Discrete Fréchet Distance of $S[s_a, s_b]$. To prove the lemma, we shall prove both sides of the bi-implication:

- Assume `SetSystem` has value *true* for query (F, f_a, f_b, S, s_p). Then by the if-statement on line 9, `FréchetDists` also has value *true* for some query $(F, S, f_a, f_b, s_a, s_b)$ with $s_a \leq s_p \leq s_b$. From that it follows that $F[f_a, f_b]$ is within Discrete Fréchet Distance of $S[s_a, s_b]$ with $s_a \leq s_p \leq s_b$.
- Assume that there exists a subtrajectory $S[s_a, s_b]$ of $S \in \mathcal{S}$ with $s_a \leq s_p \leq s_b$ which is within the Discrete Fréchet Distance of subtrajectory $F[f_a, f_b]$ of $F \in \mathcal{S}$. Then, as all subtrajectories $S[a, b] \in S$ with $a \leq s_p \leq b$ are checked in the for loop on line 8 of the algorithm, eventually subtrajectory $S[s_a, s_b]$ will be found and query $(F \in \mathcal{S}, f_a, f_b, S \in \mathcal{S}, s_p)$ in `SetSystem` will be assigned value *true*.

As both sides of the bi-implication are proven, the lemma holds and Algorithm 4 is correct. □

From the above two lemmas we can prove the correctness of the overall algorithm:

Theorem 1 (Naïve Dynamic Programming Method Correctness)**.** *Algorithm 2 returns, given a selection of trajectories $\mathcal{S}$ and a threshold Δ, a valid Set System Oracle for solving Subtrajectory Covering using Greedy Set Cover for the selection and the threshold.*

Proof. As we know both Algorithms 3 and 4 are correct, Algorithm 2 creates the structure `SetSystem` from $\mathcal{S}$ and Δ, which can, in $O(mn)$ time return all vertices in any trajectory of the selection which are covered by a subtrajectory $F[f_a, f_b]$ from a trajectory F. (The amount of vertices covered by $F[f_a, f_b] \in F$ can also be returned at the same time.) This is, by Definition 11, a Set System Oracle for our Set System. □

Conclusion This method creates a Greedy Set Cover Oracle in $O(m^2n^6)$ time and $O(m^2n^4)$ space, which can be queried in $O(mn)$ time per query. The query time is always $O(mn)$, as that is the size of the set r_P belonging to a subtrajectory P which covers every vertex of every trajectory.

5.4 Beyond Our Set System

As per Question 5, in this section we will describe how to adapt our method to more general Set Systems. This method is applicable to more general Set Systems with a few tweaks:

Let $\mathcal{D}$ be the set of d polygonal center curves. By replacing the $\mathcal{S}$ on line 3 of Algorithm 3 and on line 3 of Algorithm 4 for $\mathcal{D}$, these algorithms works for any set system with a set $\mathcal{D}$ of center curves and a set $\mathcal{S}$ of curves to be covered, which means that the overall algorithm also works for any other set system. Additionally, Algorithm 4 can be simplified to just iterate over all center curves, as we are not interested in what subtrajectories of center curves cover.

The overall running time of this method is then $O(dmn_d^3n_m^3)$, with n_m being the length of the largest trajectory and n_d being the length of the largest center curve. The space requirement is $O(dmn_d^2n_m^2)$.

6 Free Space Grids

In this section, we investigate the use of *Discrete Free Space Grids* to solve our problems. First, we will convert the problems to Free Space Grid terms, after which we will discuss the solutions for the problems we have found which involve Free Space Grids and what investigations have turned up nothing. These discussions will include the methods we have used.

Note: In this section, we consider n to be the amount of vertices in the largest trajectory considered when creating a Free Space Grid out of two trajectories. Therefore, the amount of gridpoints, which we denote as N, is $O(n^2)$.

6.1 The Problems in Free Space Grid terms

Now we know that we are using Free Space Grids, we can transform most of the problems to Free Space Grid variants. Problem 4 and Problem 5 are not transformed, as they do not have much to do with the application of Free Space Grids.

6.1.1 Problem 1a

The first subproblem of Problem 1 can be adapted to get the following problem

Problem 4 (Problem 1a FSG)**.** *The Data structure takes two trajectories F and S as input, which have to be preprocessed to answer queries which specifies a subtrajectory $P = F[f_a, f_b]$ from F and a subtrajectory $Q = S[s_a, s_b]$ from S and wants to know whether gridpoint (f_a, s_a) can reach gridpoint (f_b, s_b) in the Free Space Grid with a Discrete Fréchet Distance threshold of Δ*

If this problem is solved, we know that if a query answers *true*, $F[f_a, f_b]$ and $S[s_a, s_b]$ have a Discrete Fréchet Distance of at most Δ, due to the properties of Free Space Grids, which solves Problem 1a.

6.1.2 Problem 1b

Problem 1b is the odd one out in this thesis, as instead of having the threshold and creating a Free Space Grid using this threshold to find out if pairs of subtrajectories have a lower Discrete Fréchet Distance than this threshold, it takes the subtrajectories as they are and wants to find out the threshold.

The problem can be adapted as follows:

Problem 5 (Problem 1b FSG)**.** *The Data structure takes two trajectories F and S as input, which have to be preprocessed to answer queries which specifies subtrajectory $P = F[f_a, f_b]$ from F and subtrajectory $Q = S[s_a, s_b]$ from S, and wants to know what the maximum Fréchet Distance is on the path from gridpoint (f_a, s_a) to gridpoint (f_b, s_b) in the Free Distance Grid with an infinite threshold.*

Solving this problem gives us the threshold needed to have subtrajectory $F[f_a, f_b]$ reach $S[s_a, s_b]$ in a Free Space Grid, which means that this is the Discrete Fréchet Distance between the two subtrajectories, due to the properties of Free Space Grids.

6.1.3 Problem 2

To adapt Problem 2 for Free Space Grids, it is important to note what a solution to the problem looks like in the Free Space Grid.

As seen in Problem 2, what we get is a subtrajectory P from the first trajectory F and we are looking for a subtrajectory Q with a startpoint s_a from the second trajectory S for which the Discrete Fréchet Distance to P is less than a threshold Δ.

In a Free Space Grid made according to threshold Δ with coordinate format $(row, column)$, where the row coordinate reflects the first trajectory F and the column coordinate reflects the second trajectory S, s_a is expressed as the column $\{(f_x, s_a) \in F \times S \mid f_x \in F\}$. P can be expressed as a slab of the Free Space Grid defined by the rows $\mathcal{A} = \{(f_a, s_x) \in F \times S \mid s_x \in S\}$ and $\mathcal{B} = \{(f_b, s_x) \in F \times S \mid s_x \in S\}$, where vertex f_a is the first vertex of P and vertex f_b is the last vertex of P.

To see this explanation pictured, see Figure 3.

We can then transform Problem 2a to the following problem:

Problem 6 (Problem 2a FSG)**.** *The Data Structure takes two trajectories F and S as input, which have to be preprocessed to answer queries which specify one subtrajectory $P = F[f_a, f_b]$ from F and a startpoint s_a from S and asks for a coordinate s_b such that (s_a, f_a) reaches (s_b, f_b) in the Free Space Grid of F and S with given threshold Δ.*

Similarly, we can transform Problem 2b to the following problem:

Problem 7 (Problem 2b FSG)**.** *The Data Structure takes two trajectories F and S as input, which have to be preprocessed to answer queries which specify one subtrajectory $P = F[f_a, f_b]$ from F and a startpoint s_a from S and asks for* the biggest *coordinate s_b such that (s_a, f_a) reaches (s_b, f_b) in the Free Space Grid of F and S with given threshold Δ.*

In Figure 3, the pink point would be a solution for both problems, but the bright green point would only be a solution for Problem 6.

6.1.4 Problem 3

Problem 3 works almost the same in Free Space Grids as Problem 2, with the important distinction that instead of s_a, the vertex s_p given is a random vertex that has to be somewhere in subtrajectory

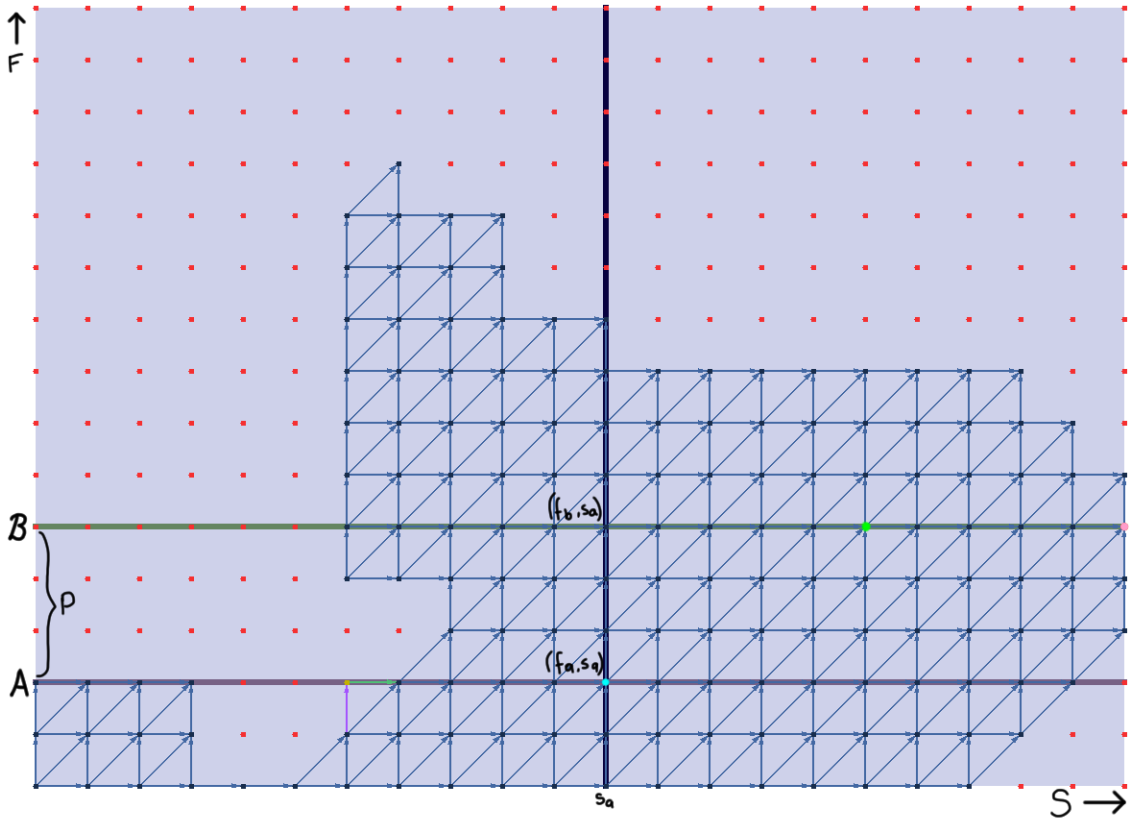


Figure 3: The context of Problem 2 in Free Space Grids visualised.

Q. Because of that, it is useful to split rows $\mathcal{A}$ and $\mathcal{B}$ along the column s_p, to get $\mathcal{A}_\ell = \{(f_a, s_x) \in F \times S \mid s_x \in S \wedge s_x \leq p\}$, $\mathcal{A}_r = \{(f_a, s_x) \in F \times S \mid s_x \in S \wedge s_x \geq p\}$, $\mathcal{B}_\ell = \{(f_b, s_x) \in F \times S \mid s_x \in S \wedge s_x \leq s_p\}$ and $\mathcal{B}_r = \{(f_b, s_x) \in F \times S \mid s_x \in S \wedge s_x \geq s_p\}$. The visualisation can be found in Figure 4.

Problem 3 can be transformed to:

Problem 8 (Problem 3 FSG)**.** *The data structure takes two trajectories F and S as input, which have to be preprocessed to answer queries which specify one subtrajectory $P = F[f_a, f_b]$ from F and a vertex s_p from the second trajectory and asks for a subtrajectory $Q = S[s_a, s_b]$ with $s_a \leq s_p$ and $s_b \geq s_p$ for which gridpoint (f_a, s_a) reaches gridpoint (f_b, s_b) in the Free Space Grid of F and S with given threshold Δ.*

In the visualisation, a solution to problem 3 would be the gold point as s_a and the pink point as s_b.

6.2 Solving Problem 1 using Free Space Grids

(In the implementation, methods described in this subsection are implemented in the package `Reachability`.)

To solve Problem 1, we need to solve Problems 4 and 5, as discussed in the last section. To do this, we started with naïve solutions an investigation of the structure of Free Space Grids, after which we did a literature study.

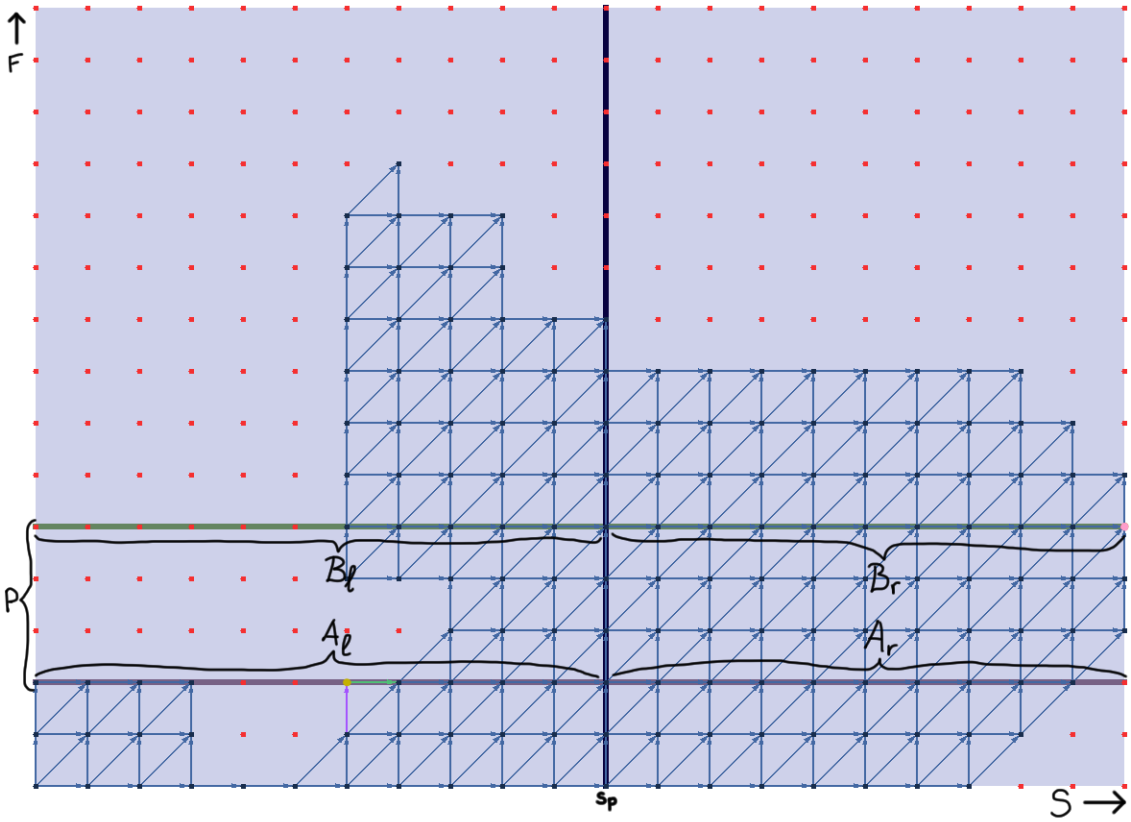


Figure 4: The context of Problem 3 in Free Space Grids visualised.

6.2.1 Naïve Solutions

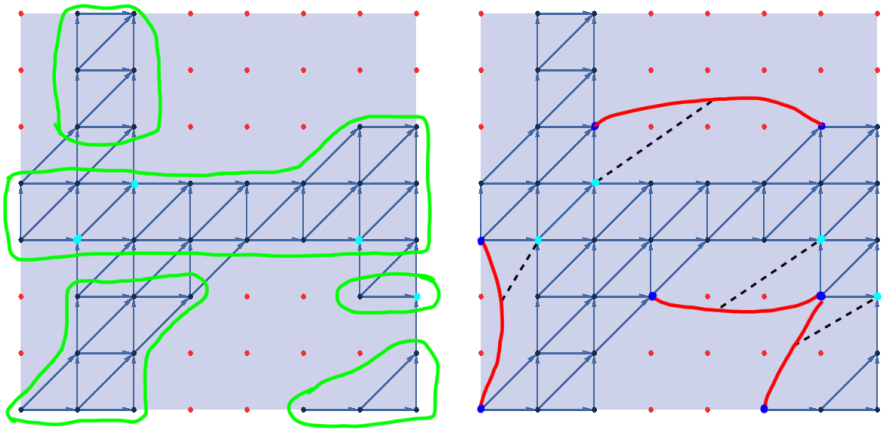
There are two kinds of naïve solutions for this problem, precomputing everything and precomputing nothing:

- Precomputing everything involves computing the full transitive closure of G. This can be done by going through the graph from the end to the start, and updating a matrix with entries for every vertex pair every time you go through a vertex. This preparation takes time $O(N^2) = O(n^4)$ to complete and also takes up $O(N^2)$ space. The worst case scenario for this method is when the biggest amount of vertices have to be checked, which is when everything is within Fréchet distance δ.
- Precomputing nothing amounts to doing a single pair shortest path computation every time the data structure is called. One way to do this is to apply BFS with as starting vertex the origin vertex given in the query. This takes $O(V + E)$ time, which, as the amount of edges is constant per vertex, comes down to a query time of $O(N) = O(n^2)$. Another way to do this is by doing a sweep, which would also take $O(N)$ time. The space requirement of this solution is $O(N)$, which is the space needed to save the Free Space Grid.

6.2.2 The Structure of Free Space Grids

We have tried to find ways to simplify the reachability problem in the structure of Free Space Grids. We did not find any, but we did find the following:

- The amount of edges a gridpoint (r, c) can have is constant: at most 3 ingoing edges, from gridpoints $(r-1, c)$, $(r, c-1)$ and $(r-1, c-1)$, and at most 3 outgoing edges, to gridpoints $(r+1, c)$, $(r, c+1)$ and $(r+1, c+1)$, by the definition of Free Space Grids.



(a) A Block Decomposition. (b) Defining gridpoints illustrated.

Figure 5: Blocks Illustrated.

- If two gridpoints neighbour either in row or in column or in both, they have an edge between them, by the definition of Free Space Grids. If you have three gridpoints (r, c), $(r+1, c)$ and $(r+1, c+1)$, then edge $((r, c), (r+1, c+1))$ does not add any information and can possibly be omitted. Same if you have three gridpoints (r, c), $(r, c+1)$ and $(r+1, c+1)$. This is not the case if (r, c) and $(r+1, c+1)$ exist, but not $(r+1, c)$ and $(r, c+1)$.
- Any path in the Free Space Grid only goes up and to the right. Additionally, paths can only cross on vertices.

Other than that, we investigated the use of *blocks* to simplify the Free Space Grid:

Blocks in a Free Space Grid. A Free Space Grid contains completely filled structures for which the answer to the reachability question from gridpoint (r_1, c_1) to gridpoint (r_2, c_2) can be simplified to "$r_1 \leq r_2$ and $c_1 \leq c_2$?". These structures (or blocks) can be defined most easily by what is not in it:

Definition 12 (Blocks in a FSG)**.** *A* block *is a cluster of existing gridpoints in the Free Space Grid such that:*

- *If a gridpoint* (r, c) *exists such that gridpoints* $(r+1, c)$ *and* $(r, c+1)$ *exist but* $(r+1, c+1)$ *does not exist, then* $(r+1, c)$ *is in a different block than* $(r, c+1)$*. These gridpoints are also called* split*-gridpoints.*
- *If a gridpoint* (r, c) *exists such that gridpoints* $(r-1, c)$ *and* $(r, c-1)$ *exist but* $(r-1, c-1)$ *does not exist, then* $(r-1, c)$ *is in a different block than* $(r, c-1)$*. These gridpoints are also called* combine*-gridpoints.*

The combination of split and combine gridpoints we call defining gridpoints.

As can be seen from the definition, there can be many block decompositions, of which Figure 5a is an example. The definition works by preventing reachability questions that cannot be answered by the question "$r_1 \leq r_2$ and $c_1 \leq c_2$?" from being put into the same block, which is illustrated in Figure 5b, with the defining gridpoints in blue and an example that cannot be solved if this gridpoint was not taken as defined gridpoint indicated in black, blue and red. To simplify the graphs using these defining gridpoints, two more defining gridpoints are needed:

1. The leftmost gridpoint without ingoing edges from the same block for every block (*begin*-gridpoints). Together with the combine-gridpoints, these form the *incoming*-gridpoints.

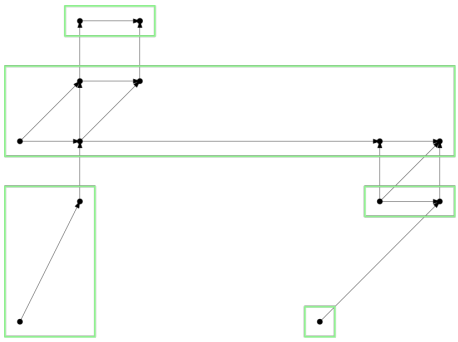
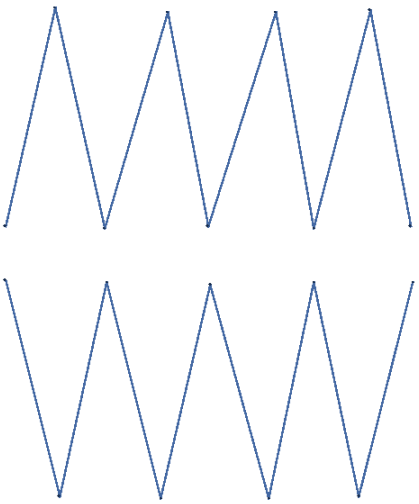


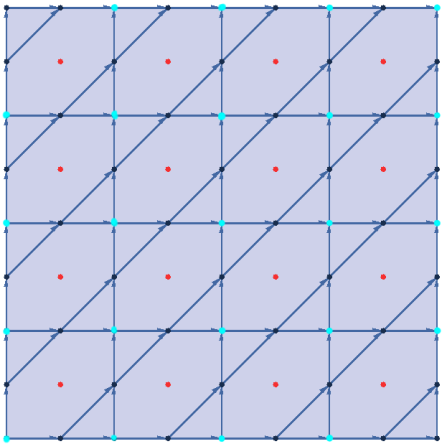
Figure 6: Simplification by Blocks

2. All gridpoints that neighbour to a defining gridpoint (begin, split or combine) of another block.

The actual simplification then consists of removing all gridpoints vertices and simplifying the paths between gridpoints left behind, as seen in Figure 6. While we did not get too into the actual specifics of this process, we believe this can be done while keeping the amount of edges per vertex constant. Querying in the same block would then be two constant time checks, but querying between blocks depends on how many vertices there are in this reduction. As in the worst case the amount of blocks is equal to the amount of defining gridpoints, The total amount of vertices is in the worst case $O(4m)$, where m is the amount of defining gridpoints in the Free Space Grid. Sadly, as can be seen in Figure 7, the amount of defining gridpoints in a Free Space Grid is $O(N)$. This means that while this simplification would simplify reachability querying in the same block, it would not simplify reachability querying between blocks significantly.



(a) The Trajectories



(b) The resulting Free Space Grid, defining gridpoints in light blue

Figure 7: A case where roughly a quarter of the existing gridpoints are defining gridpoints

The conclusion of our investigation was therefore that the block approach would not lead to a theoretical reduction in the time needed to make a Reachability Oracle for a Free Space Grid. Due to our time constraints and the fact that we found a suitable and fast method to construct a Reachability Oracle in our literature study, we decided to leave it at that.

6.2.3 Solving Problem 4

To solve this problem, other than investigating the structure of Free Space Grids, we have done a literature study as can be seen in Section 3.2, from which we have concluded that the paper discussed in Section 3.2.1 would be the best one to use. The method in this paper would enable us to make an oracle with $O(N \log N) = O(n^2 \log n)$ preprocessing time and space, which can be queried in constant time. In the rest of the sections, we will assume that this has been done in theory.

In practice, the time constraints and the complexity of the method meant that we have not implemented it in the implementation. Instead, we have implemented a naïve reachability oracle along the lines of precomputing everything, which sweeps through the Free Space Grid for every one of the N vertices in it and checks for every other vertex if it is reachable from the start vertex:

Algorithm 5 Naïve Reachability Oracle Construction

```
1: procedure PreprocessReachability(Trajectory F, Trajectory S, Free Space Grid FSG)
2:     Initialize 4-dimensional matrix Reachability.
3:     for all vertices f_a ∈ F with f_0 ≤ f_a ≤ f_n do
4:         for all vertices s_a ∈ S with s_0 ≤ s_a ≤ s_n do
5:             for all vertices f_b ∈ F, with f_a ≤ f_b do
6:                 for all vertices s_b ∈ S, with s_a ≤ s_b in the order of S do
7:                     if gridpoint (f_a, s_a) or gridpoint (f_b, s_b) does not exist in FSG then
8:                         Reachability[f_a][s_a][f_b][s_b] = false
9:                     else if (f_a, s_a) = (f_b, s_b) then
10:                        Reachability[f_a][s_a][f_b][s_b] = true
11:                    else
12:                        for All incoming edges of (f_b, s_b) in FSG do
13:                            Let gridpoint (r, c) be the source of the edge.
14:                            if Reachability[f_a][s_a][r][c] = true then
15:                                Reachability[f_a][s_a][f_b][s_b] = true
16:                            end if
17:                        end for
18:                    end if
19:                end for
20:            end for
21:        end for
22:    end for
23:    return Reachability
24: end procedure
```

This method takes $O(N^2) = O(n^4)$ time and the same amount of space. The amount of incoming edges is at most constant, so it will not impact the running time. It is correct: A gridpoint can reach itself, and any gridpoint a that can reach a gridpoint b with an outgoing edge to gridpoint c, can also reach gridpoint c.

6.2.4 Solving Problem 5

For this problem, one can again create naïve solutions following the precomputing everything or precomputing nothing approach. The first will result in an oracle with a preprocessing time of $O(N^2)$ and a constant query time, with a space requirement of $O(N^2)$. The second will result in a preprocess time of $O(N)$ and a query time of $O(N)$, with a space requirement of $O(N)$. Like the naïve implementation of last section, both can be done with a sweep for every start gridpoint, which would work as follows:

- Take the smallest Discrete Fréchet Distance saved from the at most three originators of the incoming edges.
- Take the biggest Discrete Fréchet Distance of the smallest one from the predecessors and the Fréchet Distance between the pair that makes up the current gridpoint.

Unlike in Naïve Solution 1, this works for all possible Fréchet Distances and not just those that are below the threshold. When precomputing everything, the results of all sweeps have to be saved, which causes the space requirement of $O(N^2)$.

This problem was not the priority in this thesis, so we have not found any methods beyond this naïve method for it. The most interesting applications of the problem lie in expansions for the oracle, like making the threshold a variable, which we have not focused on.

6.3 Solving Problem 2 using Free Space Grids

(In the implementation, methods described in this subsection are implemented in the package `FSGMethods`.)

To solve Problem 2, we first considered naïve ways to solve it. Afterwards, we considered the problem and formulated a solution in between the precompute everything and the precompute nothing approach.

6.3.1 Naïve Solutions

1. Like with Problem 1, we could take the transitive closure and then check for all gridpoints (f_b, s_x) with $s_x \geq s_a$ whether the gridpoint (f_a, s_a) has an edge to that gridpoint as query. For Problem 2a, we would be done if we found such an edge, which would take us $O(n)$ time, as the amount of gridpoints on row $\mathcal{B}$ is $O(n)$. For Problem 2b it would take the same time, but as an optimization step querying could start at the end of the row. The space requirement is $O(N^2)$ for the transitive closure.
2. Like with Problem 1 we could also do Single Pair Shortest Paths queries for every vertex in row $\mathcal{B}$. This would take us $O(n \cdot N) = O(n^3)$ time and would have a space requirement of $O(N)$.
3. We could also use the Reachability Oracle gained through solving Problem 1 and query that from gridpoint (f_a, s_a) to every gridpoint in row $\mathcal{B}$, giving us a query time of $O(n)$ and a space requirement of $O(N \log N)$.
4. Taking a leaf from the book of the transitive closure, we could compute everything in advance by computing for every gridpoint in the Free Space Grid (which is then taken as gridpoint (f_a, s_a)) and every row in the Free Space Grid (which is then taken as row $\mathcal{B}$) what the furthest gridpoint (f_a, s_a) can reach is on row $\mathcal{B}$. This takes $O(n^2 \cdot N) = O(n^4)$ time: There are $O(N)$ gridpoints in the Free Space Grid, $O(n)$ rows in the Free Space Grid and $O(n)$ gridpoints to check on each of the rows. As all of the answers have to be saved for $O(1)$ query time, the space requirement is $O(n^3)$, as one coordinate s_b has to be saved per gridpoint s_a per row $\mathcal{B}$.

From these naïve solutions we concluded that the range of useful non-naïve solutions lied between the following parameters:

1. $O(N \log N) = O(n^2 \log n)$ preparation time and space and $O(n)$ query time, which is the solution which makes use of the Reachability Oracle as created per our literature study and uses the naïve query method as stated above in point 3.
2. $O(n^4)$ preparation time, $O(n^3)$ space and $O(1)$ query time, which is the time needed for naïve solution 4 in the options above.

6.3.2 Solving Problems 6 and 7

To start with, it is easy to answer whether there exists a gridpoint with column coordinate s_b that solves Problems 6 and 7. It takes a single sweep, in which the highest reachable gridpoint that can be reached is saved into every gridpoint. Whether an s_b exists to satisfy Problems 6 and 7 can then be answered by the question "is row of the highest reachable vertex of gridpoint (f_a, s_a) higher than row $\mathcal{B}$?". If this is the case, then due to the fact that all paths in the Free Space Grids go up and to the right, the fact that (f_a, s_a) can reach a row higher than $\mathcal{B}$ means that it will have to go through a gridpoint (f_b, s_x) on row $\mathcal{B}$ with $s_x \geq s_a$. Taking s_x as s_b would solve at least Problem 6, and the fact that an s_x exists means that there is also a *biggest* coordinate s_b to solve Problem 7.

To actually find gridpoint (f_b, s_b) to which gridpoint (f_a, s_a) connects, additional work needs to be done. We decided to search for a method that was logarithmic in query time but took as little preprocess time as possible. The reason for this was that we already had a naïve solution with a linear query time and a naïve solution with a constant query time. We did not think trying to improve the Reachability Oracle was the main priority, but due to some observations in following paragraphs we did think there could be a logarithmic query time Free Space Grid method, of which we wanted to investigate how much preprocessing time was needed. In the end, we found that this logarithmic query time Free Space Grid method needs $O(n^3)$ preprocessing time and $O(n^3)$ space, making it a middle solution to both naïve solutions.

To find a logarithmic query time Free Space Grid method, the goal was to find a way to remove a significant amount of gridpoints from the query based upon information gotten from one gridpoint. To do this, we first decided to make the solution with Problem 7 in mind, as the extra constraint in that problem gives us more information to work with from a single gridpoint. The gridpoints to query are the gridpoints on row $\mathcal{B}$, and we distinguished three different types of gridpoints on row $\mathcal{B}$, as seen in Figure 8.

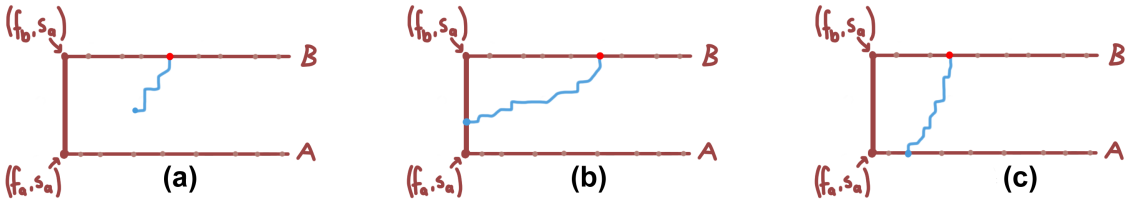


Figure 8: The different kinds of gridpoints in row $\mathcal{B}$. The gridpoints in (a) cannot be reached from column s_a or row $\mathcal{A}$. The gridpoints in (b) can be reached from column s_a. The gridpoints in (c) can be reached from row $\mathcal{A}$. It is possible for gridpoints to belong to both (b) and (c).

It takes only constant time to find out whether a gridpoint is of type (a) by saving the lowest row every gridpoint can be reached from. However, when we encounter a gridpoint a of type (a) on row $\mathcal{B}$ during query time, we need to find out where the next gridpoint relevant for the method can be found, about which a single gridpoint gives no information. As we cannot discount anything except gridpoint a, the best way to find a new relevant gridpoint is to sequentially go through the neighbours of a, and then through their neighbours, and so forth. This, however, is linear at the worst case, so it has to be done during preprocessing. Preprocessing this for all pairs of rows takes $O(n^3)$, which means that a sublinear query time for this problem (with our Set System) in Free Space Grids requires at least $O(n^3)$ preprocessing time.

For both gridpoint types (b) and (c), it holds that if (f_a, s_a) can reach this gridpoint, it is a solution for Problem 6 and you can remove all gridpoints to the left of it out of consideration for Problem 7. Furthermore, for gridpoint type (c) it holds that if (f_a, s_a) cannot reach this gridpoint, all gridpoints to the right can be taken out of consideration for both problems:

Assume (f_a, s_a) cannot reach a gridpoint (f_b, s_x) of type (b), but can reach a gridpoint (f_b, s_y) with $x < y$. Then, as (f_a, s_a) can reach (f_b, s_y) and all the vertices on that path have coordinates

with the row coordinate between f_a and f_b and column coordinates larger or equal to s_a, the path to (f_b, s_y) must cross the path from row $\mathcal{A}$ to (f_b, s_x). This is a contradiction.

Similarly, for gridpoint type (b) it holds that if (f_a, s_a) cannot reach this gridpoint, all gridpoints to the left can be taken out of consideration for both problems.

The only problem is to distinguish type (b) and type (c) gridpoints. Without making assumptions on the paths and existing vertices between the rows, we found that single gridpoints give too little information about their path to be able to distinguish the kind of gridpoint it is from the gridpoint and reachability information on its own. We cannot go through the path, as it has no guarantee that going through the path will be faster than a $O(n)$ sweep. Using the gridpoints that reach row $\mathcal{B}$ from row $\mathcal{A}$ to find out whether a gridpoint b is of type (b) or (c) does not help either:

If a gridpoint a of row $\mathcal{A}$ with column coordinate $s_a < a_c \leq b_c$ cannot reach b, then either a gridpoint a_1 with column coordinate $a_c < a_{1c} \leq b_c$ or a gridpoint a_2 with column coordinate $s_a < a_{2c} \leq a_c$ could still reach b, and to find out which one it is we would first have to match a to the gridpoint on row $\mathcal{B}$ that it connects to, and as that is the problem we wanted to solve by considering row $\mathcal{A}$ in the first place, we did not look too much into this.

As involving row $\mathcal{A}$ would not help us, no constant amount of rows would either. Therefore, to keep the query time sub linear, more preprocessed information would be needed (as involving a possible $O(n)$ amount of rows could not be done in the query if the time has to be kept sub linear). Furthermore, as this information was dependent on two specific rows and one gridpoint at the very least, creating a structure for this would also take $O(n^3)$ time.

By saving the farthest gridpoint any gridpoint on any row $\mathcal{B}$ can be reached from on any row $\mathcal{A}$, it is possible to distinguish between type (b) and (c) vertices in constant time. Using that, the preprocessing methods and the query algorithm are as follows:

The Preprocessing Methods

1. A method to compute for all gridpoints, in a sweep from the lower left gridpoint to the upper right gridpoint, the lowest gridpoints they can be reached from using information from the incoming neighbours. This can be done in $O(N)$ time and space.

2. A method to compute for all gridpoints, in a sweep from the upper right gridpoint to the lower left gridpoint, the highest gridpoint they can reach using information from the outgoing neighbours. This can be done in $O(N)$ time and space.

3. Optionally, a method that computes for every gridpoint on a row what the leftmost and rightmost gridpoint on that same row are that can reach and can be reached from the gridpoint. This is useful to shrink the search space. This can be done in $O(N)$ time and space.

4. A method to compute, for all gridpoints on all rows $\mathcal{B}$ what the furthest gridpoint on any row $\mathcal{A}$ is that can reach this gridpoint, using Algorithm 6. This algorithm takes $O(n^3)$ time (n^2 combinations of two rows and $O(n)$ vertices on every row $\mathcal{B}$) and the same amount of space (to save the answers).

5. A method to prune all gridpoints on row $\mathcal{B}$ that could not be reached from row $\mathcal{A}$ during the preprocessing (for all pairs of rows $\mathcal{A}$ and $\mathcal{B}$). This takes $O(n^3)$ time (n^2 combinations of two rows and $O(n)$ gridpoints on every row $\mathcal{B}$) and the same amount of space: The result is a data structure which holds all *gridpoints of interest* to the query algorithm for each row combination. Gridpoints of Interest for a certain row combination (f_a, f_b) will be denoted as $POI_{f_a}^{f_b}$

   Optionally, you can also prune this further, by removing gridpoints that are connected to gridpoints to the right of them on the same row: If a gridpoint b_1 is connected to a gridpoint b_2 on row $\mathcal{B}$ with $b_1.column < b_2.column$, then b_2 is an equally suitable candidate for Problem 6 and a better suitable candidate for Problem 7, as everything that can reach b_1 can also reach b_2 and b_2 is further on row $\mathcal{B}$ than b_1

Algorithm 6 Furthest To Reach Gridpoint Preprocessing

```
1: procedure FurthestToReach(Free Space Grid FSG, Reachability Oracle Reach)
2:     Initialize 3-dimensional matrix FtR as an empty matrix.
3:     for all rows f_a do
4:         for all rows f_b with b ≥ a do
5:             Column s_a ← FSG.width        ▷ FSG.width denotes the last gridpoint on the row
6:             Column s_b ← FSG.width        ▷ 1 is the index of the first gridpoint on the row
7:             while s_b > 0 do
8:                 while (f_b, s_b) does not exist in FSG or (f_b, s_b) is not reached from row f_a do
9:                     Decrement s_b
10:                    if s_b <= 0 then
11:                        Go to the next iteration of the loop at step 4.
12:                    end if
13:                end while
14:                if s_a > s_b then
15:                    s_a ← s_b
16:                end if
17:                while (f_a, s_a) does not exist in FSG or (f_a, s_a) does not reach row f_b do
18:                    Decrement s_a
19:                    if s_a <= 0 then
20:                        Go to the next iteration of the loop at step 4.
21:                    end if
22:                end while
23:                if Reach.query((f_a, s_a), (f_b, s_b)) = true then
24:                    FtR[f_a][f_b][s_b] = (f_a, s_a)
25:                    Decrement s_b
26:                else
27:                    Decrement s_a
28:                end if
29:            end while
30:        end for
31:    end for
32:    return FtR
33: end procedure
```

The Query Algorithm For the query, we implemented Algorithm 7. The intuition behind this algorithm is as follows:

1. Take the middle gridpoint of the remaining gridpoints.
2. Check whether this gridpoint can be reached from (f_a, s_a). If this is the case, recurse to the right, as we are looking for the furthest reachable gridpoint. (If you wanted to solve only Problem 6, this right recursion would not be necessary.)
3. If not, check whether this gridpoint is of type (b) or of type (c). If the gridpoint is of type (b), no gridpoints to the left of this gridpoint can be reached by (f_a, s_a), so recurse to the right. If it is of type (c), no gridpoint to the right of this gridpoint can be reached by (f_a, s_a), so recurse to the left.
4. return the last remaining gridpoint.

For this algorithm's running time, note that $POI_{f_a}^{f_b}$ has $O(n)$ vertices in it, and that every iteration of the recursive query, half of the remaining $POI_{f_a}^{f_b}$ is disregarded. This makes the

Algorithm 7 Problem 7 Query Algorithm

1: **procedure** QUERYPROBLEM7(Gridpoints of Interest $POI_{f_a}^{f_b}$, Reachability Oracle $Reach$, gridpoint (f_a, s_a), Data Structure FtR)
2: **if** (f_a, s_a) cannot reach row f_b **then** ▷ f_b is the row of any gridpoints in POI
3: **return** NO.
4: **end if**
5: Gridpoint $(f_b, s_b) \leftarrow$ RECURSIVEQUERY$(POI_{f_a}^{f_b}, Reach, (f_a, s_a), FtR, 1, POI_{f_a}^{f_b}.length + 1)$ ▷ $POI_{f_a}^{f_b}$ ranges from 1 to $POI_{f_a}^{f_b}.length$
6: **return** (f_b, s_b).
7: **end procedure**
8: **procedure** RECURSIVEQUERY($POI_{f_a}^{f_b}, Reach, (f_a, s_a), FtR$, Integer $left$, Integer $right$)
9: **if** $left = right$ **then**
10: **return** $POI_{f_a}^{f_b}[left - 1]$
11: **end if**
12: $currentIndex \leftarrow \lfloor left + \frac{right - left}{2} \rfloor$
13: $currentPoint \leftarrow POI_{f_a}^{f_b}[currentIndex]$. ▷ Note that it has row f_b by definition of $POI_{f_a}^{f_b}$
14: **if** $reach.query((f_a, s_a), currentPoint) = true$ **then**
15: **return** RECURSIVEQUERY$(POI_{f_a}^{f_b}, Reach, (f_a, s_a), FtR, currentIndex + 1, right)$
16: **else if** $FtR[f_a][f_b][currentPoint.column].column < s_a$ **then**
17: **return** RECURSIVEQUERY$(POI_{f_a}^{f_b}, Reach, (f_a, s_a), FtR, currentIndex + 1, right)$
18: **else**
19: **return** RECURSIVEQUERY$(POI_{f_a}^{f_b}, Reach, (f_a, s_a), FtR, left, currentIndex)$
20: **end if**
21: **end procedure**

running time $O(\log n)$, provided that querying $POI_{f_a}^{f_b}, Reach$ and FtR can be done in constant time.

Correctness of Algorithms 6 and 7 To be sure we can use the above method to solve Problems 6 and 7 with $O(n^3)$ preprocessing time and space and a query time of $O(\log n)$, we will prove the algorithms we have given correct. As the rest of the preprocessing is simple, we will skip them. As a result for Problem 7 is also a result for Problem 6, proving the algorithms correct for Problem 7 will be sufficient.

We shall first prove the correctness of Algorithm 6, as that is required for the correctness of Algorithm 7.

Lemma 4 (Furthest To Reach Gridpoint Preprocessing Correctness)**.** *The data structure created by Algorithm 6 returns gridpoint (f_a, s_a) when queried with rows f_a, f_b and coordinate s_b if and only if gridpoint b can be reached from row f_a and gridpoint (f_a, s_a) is the furthest gridpoint on row f_a that can reach gridpoint (f_b, s_b).*

Proof. Assume that the Reachability Oracle works as it is supposed to. The sweep in the algorithm works on the following principle:

If you have a gridpoint b_1 and b_2, both on row $\mathcal{B}$ and with $b_2.column > b_1.column$, and b_2 has gridpoint a_2 as the gridpoint that is the furthest on a row $\mathcal{A}$ that can reach it, the gridpoint a_1 furthest on row $\mathcal{A}$ that reaches gridpoint b_1 will be on a column coordinate smaller than a_2: if $a_2.column < a_1.column$, then the paths from a_1 to b_1 and from a_2 and b_2 cross, which would mean that a_1 would actually be the furthest gridpoint on row $\mathcal{A}$ that reaches b_2, which contradicts the assumption that a_2 was the furthest gridpoint on row $\mathcal{A}$ that reaches b_2.

By this principle, we know that if a gridpoint (f_b, s_b) can be reached from row f_a, the furthest gridpoint it can be reached from (f_a, s_a) will be considered in the sweep. As the if-statement on line 23 is the only place other than the while loop on line 17 (which the gridpoint (f_a, s_a) does not

trigger as it exists and can reach row f_b), (f_a, s_a) will trigger the if-statement on line 23, which will set (f_a, s_a) as the answer to query (f_a, f_b, s_b). The if-statement also ensures that no gridpoint (f_a, a) with $s_a > a$ will get a chance to overwrite the answer to the query, as s_b is decremented. Therefore the data structure can only return (f_a, s_a) for query (f_a, f_b, s_b). Thir proves the lemma from right to left.

If the data structure in the algorithm returns a gridpoint (f_a, s_a) for query (f_a, f_b, s_b), we know that (f_b, s_b) can be reached from row f_a, as otherwise its entry in the data structure would have been empty courtesy of the while loop on line 8. Assume that there is a gridpoint (f_a, a) with $s_a < a$ that can also reach (f_b, s_b). Then gridpoint (f_a, a) would have triggered the if-statement on line 23, which would mean that s_b would be decremented and s_a would never have been considered for (f_a, f_b, s_b), which would make it impossible for the data structure to return (f_a, s_a). This proves the lemma from left to right.

As the lemma is proven from left to right and from right to left, Algorithm 6 is correct. □

As algorithm 6 is correct, we will use that to prove Algorithm 7 correct:

Theorem 2 (Problem 7 Query Algorithm Correctness)**.** *Algorithm 7 returns the gridpoint (f_b, s_b) if and only if the coordinate s_b satisfies Problem 7.*

Proof. To prove the theorem, assume that all preprocessing works as expected (for most preprocessing this is trivial, see Lemma 4 for Preprocessing Method 4). We will prove both sides of the bi-implication in the theorem:

1. Assume that the algorithm returns the gridpoint (f_b, s_x). Then assume the gridpoint does not satisfy Problem 7. This means that there is a gridpoint on row $\mathcal{B}$ with column coordinate $s_b > s_x$ that (f_a, s_a) can reach. However, for (f_b, s_x) to be returned all recursions after (f_b, s_x) must have been recursions to the right, otherwise $left$ would become larger than s_{x+1} and (f_b, s_x) could not have been returned. That means that to the right of s_x there could only have been gridpoints of type (c) being considered, including the gridpoint immediately to the left of (f_b, s_x). As s_{x+1} must also be smaller than s_b (or s_b would have been returned) and (f_b, s_{x+1}) is of type (c), the path from row $\mathcal{A}$ to s_{x+1} and from (f_a, s_a) to (f_b, s_b) crosses, which means that (f_b, s_{x+1}) is reachable from (f_a, s_a), contradicting that (f_b, s_{x+1}) is of type (c). Therefore, s_x satisfies Problem 7.

2. Assume a vertex $s_b \in S$ satisfies Problem 7 for the rows $\mathcal{A}$ and $\mathcal{B}$ and vertex s_a. Then assume that with those rows and s_a, the algorithm does not return gridpoint (f_b, s_b). The algorithm cannot return No, as we assumed the preprocessing works as expected and s_a can reach row $\mathcal{B}$, so it must return a gridpoint (f_b, s_x) with $s_x \neq s_b$. Depending on what kind of gridpoint (f_b, s_x) is, there are different contradictions to be derived:

   - if $s_x < s_b$ and (f_a, s_a) can reach (f_b, s_x), or (f_b, s_x) is of type (b), then the contradiction works the same as that in point 1 of this proof, where a contradiction is derived from the fact that (f_b, s_{x+1}) is both not reachable and reachable from (f_a, s_a).
   - if $s_x < s_b$ and (f_a, s_a) cannot reach (f_b, s_x) and (f_b, s_x) is of type (c), then the path from (f_a, s_a) to (f_b, s_b) must cross the path from row $\mathcal{A}$ to (f_b, s_x), which means that (f_b, s_x) can be reached from (f_a, s_a), which is a contradiction.
   - if $s_x > s_b$ and (f_a, s_a) can reach (f_b, s_x), that causes a contradiction with (f_b, s_b) being defined as the furthest gridpoint (f_a, s_a) can reach on row $\mathcal{B}$.
   - if $s_x \geq s_b$, (f_a, s_a) cannot reach (f_b, s_x) and (f_b, s_x) is of type (b), then there is a path from some (f_c, s_a) with $f_a < f_c \leq f_b$ to (f_b, s_x). There is also a path from (f_a, s_a) to (f_b, s_b). As $s_b \leq s_x$, these paths must cross. Therefore, (f_a, s_a) can reach (f_b, s_x), which is a contradiction.
   - if $s_x \geq s_b$, (f_a, s_a) cannot reach (f_b, s_x) and (f_b, s_x) is of type (c), then it's recursion disregards index s_{x+1}, which means that (f_b, s_x) cannot be returned by the virtue that the gridpoints of type (c) recurse to the wrong direction.

As (f_b, s_x) can fall in none of the above categories, it cannot exist, which means that our assumption must be wrong, $s_x = s_b$ and the algorithm returns (f_b, s_b).

As we have proven that the algorithm returns the gridpoint (f_b, s_b) if and only if the coordinate s_b satisfies Problem 7, the algorithm is correct. □

Conclusion With the above preprocessing and query algorithm, we are able to solve Problems 6 and 7 with preprocessing time and space requirement of $O(n^3)$ and a query time of $O(\log n)$ per query.

6.4 Solving Problem 3 Using Free Space Grids

(In the implementation, methods described in this subsection are implemented in the package `FSGMethods`.)

To solve Problem 3, we again considered naïve ways to solve it. Afterwards, we used the logarithmic approach from Problem 7 to create a logarithmic approach for this problem.

6.4.1 Naïve Solutions

1. With the transitive closure, check for all vertices in $\mathcal{A}_\ell$ whether they are adjacent to a vertex in $\mathcal{B}_r$, stopping when you have found one. ($\mathcal{A}_\ell$ and $\mathcal{B}_r$ defined as in figure 4). Creating the transitive closure takes $O(n^4)$ time and space, and querying this data structure takes $O(n^2)$, as when s_p is in the middle of S, you can check at most $O(\frac{n}{2} \cdot \frac{n}{2})$ combinations.

2. Whenever you query for an answer to this problem, shortest path algorithms or a sweep would also work. Single Pair Shortest Path would need to be applied for at most all combinations between gridpoints on $\mathcal{A}_\ell$ and $\mathcal{B}_r$, which would cause it to have a running time of $O(n^4)$. A sweep would need to be done for all vertices on row $\mathcal{A}_\ell$, which would cause the full query to have a running time of $O(n^3)$. In both cases, the preprocessing time and space is $O(n^2)$.

3. **The Naïve Reachability FSG Method**: With a constant reachability oracle, the query could be reduced to a constant time operation for potentially every combination between gridpoints on $\mathcal{A}_\ell$ and $\mathcal{B}_r$. This would take $O(n^2)$ time per query and would require a preprocessing time and space of $O(n^2 \log n)$.

4. **The Naïve Shortcut Method**: With the solution to Problem 7, the query would amount to a single operation per gridpoint on $\mathcal{A}_\ell$. Depending on which solution you pick for Problem 7, this would take $O(n^3)$ preprocessing time and space and $O(n \log n)$ query time when using Algorithm 7 or $O(n^4)$ preprocessing time, $O(n^3)$ space and $O(n)$ query time when using naïve solution 4 of Section 6.3.1. If using the latter, all queries can be done in $O(n^4)$ time during preprocessing, which allows for constant query time.

6.4.2 Solving Problem 8

To solve Problem 8, we use the logarithmic query time method for problem 7. For clarity's sake, querying this algorithm will be written as $FR.query()$. Other than the preprocessing needed to use that data structure (as can be found in the last section), we also need to preprocess the gridpoints on Row $\mathcal{A}$ much like we preprocessed the gridpoints on Row $\mathcal{B}$ for the solution in last section, for the same reasons:

All gridpoints on Row $\mathcal{A}$ that cannot reach Row $\mathcal{B}$ must be found so they can be ignored, which can be done via a sweep. Optionally, every gridpoints a for which there is another gridpoints earlier on the same row that can reach a can also be ignored, for everything that a reaches on Row $\mathcal{B}$ can also be reached by this earlier gridpoints. Computing this for all pairs of rows takes $O(n^3)$ time and space.

For clarity's sake, while these can also be called points of interest, we shall call the set of remaining gridpoints *points to query* or PTQ, and the points to query for a certain row combination (f_a, f_b) shall be denoted as $PTQ_{f_a}^{f_b}$.

The Query Algorithm For the query part of this problem, we implemented Algorithm 8. The intuition goes as follows:

- For the current gridpoint (f_a, s_a), if $s_a > s_p$, neither this gridpoint nor all gridpoints to the right of it will ever be a solution, so recurse to the left.
- Let gridpoint (f_b, s_b) be the furthest gridpoint on row $\mathcal{B}$ that (f_a, s_a) can reach. If $s_b < s_p$, (f_a, s_b) is not the solution and neither is any gridpoint to the left of it so recurse to the right.
- If there has not been recursed by now, the pair (s_a, s_b) is a solution to Problem 8.
- If there are no gridpoints left, there is no solution for this case of Problem 8.

Algorithm 8 Problem 8 Query Algorithm

procedure QUERYPROBLEM8(Gridpoints to Query $PTQ_{f_a}^{f_b}$, Gridpoints of Interest $POI_{f_a}^{f_b}$, Reachability Oracle *Reach*, Problem 7 Oracle $Furthest$, Data Structure FtR, Integer s_p, Trajectory S)
    Subtrajectory indices $(s_a, s_b) \leftarrow$ RECURSIVEQUERY($PTQ_{f_a}^{f_b}, POI_{f_a}^{f_b}, Reach, Furthest, FtR, p, 1, PTQ_{f_a}^{f_b}.length + 1$)
    return (s_a, s_b).
end procedure
procedure RECURSIVEQUERY($PTQ_{f_a}^{f_b}, POI_{f_a}^{f_b}, Reach, Furthest, FtR, s_p$, Integer $left$, Integer $right$)
    if $left = right$ **then**
        return NO
    end if
    $currentIndex \leftarrow \lfloor left + \frac{right - left}{2} \rfloor$
    Gridpoint $(f_a, s_a) \leftarrow PTQ_{f_a}^{f_b}[currentIndex]$.
    if $s_a > s_p$ **then**
        return RECURSIVEQUERY($PTQ_{f_a}^{f_b}, POI_{f_a}^{f_b}, Reach, Furthest, FtR, p, left, currentIndex$)
    end if
    Gridpoint $(f_b, s_b) \leftarrow Furthest.query(POI_{f_a}^{f_b}, Reach, (f_a, s_a), FtR)$
    if $s_b < s_p$ **then**
        return RECURSIVEQUERY($PTQ_{f_a}^{f_b}, POI_{f_a}^{f_b}, Reach, Furthest, FtR, p, currentIndex + 1, right$)
    else
        return (s_a, s_b)
    end if
end procedure

Correctness of Algorithm 8. To prove Algorithm 8 correct, we shall prove the following theorem:

Theorem 3 (Problem 8 Query Algorithm Correctness)**.** *Algorithm 8 is correct:*

1. *If Algorithm 8 returns a subtrajectory $S[s_a, s_b]$, $S[s_a, s_b]$ satisfies Problem 8.*
2. *If a subtrajectory satisfying Problem 8 exists, Algorithm 8 returns a subtrajectory $S[s_a, s_b]$.*

Proof. To begin with, assume that the Reachability data structure and the data structures from Section 6.3.2 (and their preprocessing) work as intended, as established in Section 6.3.2 and the paper by Thorup [14]. Additionally, assume the preprocessing is properly implemented. We will prove the both implications in the theorem.

- If Algorithm 8 returns a subtrajectory $S[s_a, s_b]$, $S[s_a, s_b]$ satisfies Problem 8:

  Assume it is not. Then either $s_a > s_p$, in which case the if-statement on line 11 would have recursed and s_a could not have been returned or $s_b < s_p$, in which case the if-statement on line 15 would have recursed and s_b could not have been returned. As both lead to a contradiction and the substructures are assumed to work (so gridpoint (f_a, s_a) does indeed reach (f_b, s_b)), the assumption must be false.

- If there exists a subtrajectory which satisfies the problem, the algorithm returns a subtrajectory (s_a, s_b).

  Assume it does not. Then at some point during the algorithm, $left = right$. This can only happen if for any considered subtrajectory (s_a, s_b), $s_a > s_p$ or $s_b < s_p$. This is not the case for any subtrajectories which satisfy the problem $S[s_a, s_b]$, so the gridpoint (f_a, s_a) must have never been considered. That means one of two things:

  - They could have been disregarded during a recursion to the left. This can only happen when they are to the right of a gridpoint (f_a, s_x) that triggers a recursion to the left. This recursion is only triggered if $s_x > s_p$. But if $s_x > s_p$, then, as (f_a, s_a) is to the right of (f_a, s_x), also $s_a > s_p$, which contradicts that s_a, s_b satisfies the problem.
  - They could have been disregarded during a recursion to the right by gridpoint (f_a, s_x) and the furthest reachable gridpoint (f_b, s_y), which can only happen if gridpoint (f_a, s_a) is to the left of gridpoint (f_a, s_x). Gridpoint (f_a, s_a) can reach a gridpoint (f_b, s_b) with $s_b > s_p$, as it satisfies the problem. However, as $s_y < s_p$, the paths from (f_a, s_a) to (f_b, s_b) and from (f_a, s_x) to (f_b, s_y) must cross, which means that (f_a, s_x) can reach (f_b, s_b), which means that the $Furthest$ datastructure gave the wrong gridpoint. As $Furthest$ is assumed to work as intended, this is a contradiction.

  As neither of the two options are possible, (f_a, s_a) is considered, which contradicts our earlier assumption.

As both implications in the theorem have been proven, the theorem holds and Algorithm 8 is correct. □

Conclusion. With the above algorithm, we are able to solve Problem 8 with preprocessing time and space of $O(n^3)$ and a query time of $O((\log n)^2)$ ($O(\log n)$ iterations of $O(\log n)$ time). We call this method the **Free Space Grid Method**.

6.5 Considering Question 4

(In the implementation, methods described in this subsection are implemented in the package `Querier`.)

Now we have the Free Space Grid Methods to find out whether one vertex is covered by a subtrajectory, we still need to make that into a Set System Oracle that can return all vertices covered by a subtrajectory. (Note that we can also choose to do this when Greedy Set Cover is called, but by the nature of what Greedy Set Cover does, we need to know all covered vertices for all possible subtrajectories regardless, so we need to query everything.)

A naïve way to do that, as implemented in the `Querier/EasyQuerier.java` file in the implementation, is by simply querying all vertices for every subtrajectory using Algorithm 8. To query for all subtrajectories of a trajectory F what they cover of subtrajectory S takes $O(n^3)$ preprocessing time and space and then $O(n^3(\log n)^2)$ query time. For two trajectories, this then

takes $O(n^3(\log n)^2)$ time to get all information we need for the Oracle. For m trajectories, this takes $O(m^2n^3(\log n)^2)$ time.

A smarter way to use the results of our Free Space Grid Methods to make a Set System Oracle makes use of the fact that for every subtrajectory $P = F[f_a, f_b]$ of a trajectory F, *all* vertices s_p in any trajectory S need to be queried. With this insight, it is possible to directly use Algorithms 6 and 7 instead of Algorithm 8, as seen in Algorithm 9.

Algorithm 9 Long Jump Query Algorithm

```
1: procedure LongJump(Vertex f_a ∈ F, vertex f_b ∈ F, Points of Interest POI^{f_b}_{f_a}, Trajectory
   S, Reachability Oracle Reach, Problem 7 Oracle Furthest, Data Structure FtR)
2:     Initialize an array covered with false for all vertices in S
3:     Let s_a ← 1.
4:     while s_a <= S.length do                ▷ The vertices on S range from 0 to S.length
5:         GridPoint (f_b, s_b) ← Furthest.query(POI^{f_b}_{f_a}, Reach, (f_a, s_a), FtR)
6:         if GridPoint (f_b, s_b) does not exist then
7:             s_a ← s_a + 1
8:         else
9:             for All vertices of S with index s_a ≤ i ≤ s_b do
10:                covered[s_j] = true
11:            end for
12:            GridPoint (f_a, s_x) ← FtR[f_a][f_b][s_b]
13:            if s_a = s_x then
14:                s_a ← s_a + 1
15:            else
16:                s_a ← s_x
17:            end if
18:        end if
19:    end while
20:    return covered
21: end procedure
```

This algorithm takes $O(n \log n)$ time, which makes the total query time for all subtrajectories for a pair of two trajectories $O(n^3 \log n)$, which makes the total query time $O(m^2n^3 \log n)$. No significant space requirement factor is added to the existing $O(n^3)$ space requirement.

Correctness of the Long Jump Query Method To prove Algorithm 9 correct, we will prove Theorem 4.

Theorem 4 (Long Jump Query Method Correctness)**.** *The* `covered` *Array constructed by Algorithm 9 for subtrajectory $(f_a, f_b) \in F$ and trajectory S returns that a certain vertex $s_p \in S$ is covered by subtrajectory (f_a, f_b) if and only if s_p is covered by (f_a, f_b).*

Proof. Assume $Furthest$ and FtR work correctly (see Section 6.3 for that). To prove the theorem, we shall prove it both ways:

- Assume that for some vertex $s_p \in S$ and some subtrajectory $(f_a, f_b) \in F$ the Array constructed by Algorithm 9 returns *true*. This can only happen if for some s_a with $s_a \leq s_p$ (f_a, s_a) can reach some (f_b, s_b) with $s_p \leq s_b$. This means that subtrajectory (s_a, s_b) is within Discrete Fréchet Distance of (f_a, f_b), and that subtrajectory (s_a, s_b) includes s_p. By definition 11, this means that s_p is covered.
- Assume that there is a vertex $s_p \in S$ which is covered by (f_a, f_b), but that the Array constructed by Algorithm 9 returns $false$. As the Array returns $false$, either there is no $s_a \leq s_p \leq s_b$ where (f_a, f_b) covers (s_a, s_b), which is not the case by assumption, or somehow

the subtrajectory (s_a, s_b) that includes s_p and is covered by (f_a, f_b) is never found. The only possible place this could happen is on line 16, where s_a can be skipped by a jump.

In this case, let $s_{x_1} < s_a < s_{x_2}$ be the vertices in S where the jump from s_{x_1} to s_{x_2} is what skipped s_a. Let (f_b, s_{y_1}) be the furthest reachable gridpoint from (f_a, s_{x_1}). We know that (f_a, s_{x_2}) is the furthest reachable gridpoint on row $\mathcal{A}$ that reaches (f_b, s_{y_1}), so $s_{x_2} \leq s_{y_1}$ (if not that contradicts with how paths in a Free Space Grid work). This means that $s_{x_1} < s_a < s_{x_2} \leq s_{y_1}$, which means that everything between s_a and s_{x_2} gets assigned *true* in the loop on line 9. Therefore, $s_p > s_{x_2}$. But if that is the case, $s_{y_1} < s_b$, which means that the path from (f_a, s_a) to (f_b, s_b) crosses the path from (f_a, s_{x_2}) to (f_b, s_{y_1}). Therefore, the furthest gridpoint on row $\mathcal{B}$ that (f_a, s_{y_2}) can reach is at least (f_b, s_b), so s_p will still be assigned *true* in the loop on line 9.

As in all cases, s_p gets assigned *true*, our assumption that the Array returns $false$ for s_p cannot be correct.

This proves the bi-implication both ways, so the theorem holds and Algorithm 9 is correct. □

Conclusion. Using the Long Jump Query Algorithm with Algorithm 7, we can construct a Set System Oracle for out Set System as defined in Definition 11 in $O(m^2n^3 \log n)$ time, with a space requirement of $O(m^2n^3)$. We call this method the **Free Space Grid Long Jump Method**.

6.6 Final Steps: Improving Our Results by Using Recursive Sweeps

Until now, our solution of Problem 7 makes use of a Reachability Oracle and a logarithmic Query Time Algorithm 7. However, we also found a better method, which led us to a method with $O(n^3)$ preprocess time, $O(1)$ query time and $O(n^3)$ space required. Due to the time constraints, we were not able to implement or test this method, but we present and prove it in this section.

6.6.1 A Better Solution for Problem 7

The main insight that led to this method is that we can use a kind of recursive sweep dependent on the amount of edges to get the information we need to solve Problem 7. Originally this was thought up for the context of general Set Systems, but it works for our Set System as well. This information is accessible in constant time from each vertex if we assume pointers. The method is displayed in Algorithm 10 and works as follows:

- At the beginning of the recursive sweep for one row $\mathcal{B}$ in the Free Space Grid, the rightmost point b_1 of $\mathcal{B}$ is taken, after which all the predecessors of b get a pointer to b in their array `Furthest`, with index $\mathcal{B}$. Every point is only processed once: if the point is already processed the value `Furthest`$[\mathcal{B}]$ will not be -1 and the point will not be added to the list, which means it will not be processed again.
- After this first sweep, all predecessors of $\mathcal{B}$ are processed. Then, take the rightmost point b_2 of $\mathcal{B}$ that is not a predecessor of $\mathcal{B}$ and sweep again along its predecessors. All points that are already predecessors of b_1 will not be processed again as their value for `Furthest`$[\mathcal{B}]$ is not -1, but b_1's column value, which means that any gridpoint a will get the column value of the furthest gridpoint they can reach.
- Continue this through row $\mathcal{B}$ until there are no unprocessed points on $\mathcal{B}$ left.

Running Time. In this recursive sweep, all edges are only considered once, as all gridpoints can only be processed once: if a gridpoint is processed, it will no longer have a value of -1, which means that it cannot be processed again. Due to the definition of a Free Space Grid, there are only $O(3N)$ edges in a Free Space Grid, which means that one iteration of the loop in line 3 of Algorithm 10 takes $O(N)$ time. As there are n rows $\mathcal{B}$, this makes for a total running time of

Algorithm 10 Better Problem 7 solution

```
1: procedure FindFurthest(Free Space Grid FSG)
2:     For every gridpoint a, initialize an O(n) array Furthest with −1 in every cell, which saves
       the rightmost point b a can reach on any row B. Assuming constant time pointer access, the
       rightmost point from a specific row B can be accessed in O(1) time. The total space taken to
       save the Free Space Grid will then be O(n^3).
3:     for every row B do
4:         create a variable i which is the column index of the point left of the leftmost processed
       point on row B. Initialize it to the rightmost point of B
5:         while i ≥ 0 do                          ▷ 0 being the index of the leftmost point of B
6:             create a list L which keeps track of the gridpoints to process.
7:             Take point b with index i from B and put it in L. Decrement i.
8:             while L is not empty do
9:                 Take the first gridpoint p of L and remove it from L
10:                for all predecessors q of gridpoint p (at most 3) do
11:                    if q.Furthest[B]= −1 then
12:                        q.Furthest[B]= b.column
13:                        put q in L.
14:                        if p is on row B then
15:                            i = min(i, p.column − 1)
16:                        end if
17:                    end if
18:                end for
19:            end while
20:        end while
21:    end for
22: end procedure
```

$O(n^3)$ and a total query time of $O(1)$. The space is also $O(n^3)$, as you save $O(N)$ gridpoints with an array of $O(n)$ size.

Correctness of Algorithm 10. We prove the correctness of our algorithm with the following theorem:

Theorem 5 (Algorithm 10 Correctness)**.** *Algorithm 10 is correct:*

a.`Furthest`[$\mathcal{B}$] of gridpoint a contains the (column coordinate) of point b on row $\mathcal{B}$ if and only if b is the Furthest gridpoint a can reach on row $\mathcal{B}$.

Proof. To prove the theorem we prove both sides of the bi-implication:

- Assume that a.`Furthest`[$\mathcal{B}$]$= b_1.column$. Then assume there is another gridpoint b_2 with $b_2.column > b_1.column$ which is actually the furthest gridpoint on $\mathcal{B}$ that a can reach. First, note the following:

  b_2 cannot be skipped by the updating of i, as that leads to the following contradiction: Assume b_2 was skipped by updating i. This can only during the sweep of a gridpoint b_3 with $b_3.column > b_2.column$ which is reached by a gridpoint b_4, with $b_4.column < b_2.column$. Both b_3 and b_4 must be on row $\mathcal{B}$, as otherwise b_3 cannot have its own sweep and b_4 would not update i. We know that b_4 reaches b_3 and as that path can only be on row $\mathcal{B}$ due by definition of Free Space Grids, b_2 must also reach b_3, as it is in between b_4 and b_3. This, however, means that a can also reach b_3, which contradicts our assumption that b_2 is the furthest gridpoint on row $\mathcal{B}$ that a can reach.

  As b_2 cannot be skipped by i, this means b_2 would have its own sweep in the algorithm. In this sweep, $b_2.column$ will be saved in all gridpoints that cannot already reach a gridpoint right of b_2. This includes a. This means that when b_1 has its sweep, when gridpoint a is considered, it will not be added to the process list as the value in a.`Furthest`[$\mathcal{B}$] will be $b_2.column$. Therefore, a.`Furthest`[$\mathcal{B}$]$\neq b_1.column$, which contradicts our assumptions. This means that a.`Furthest`[$\mathcal{B}$]$= b_1.column$ implies that b is the furthest gridpoint that a can reach on row $\mathcal{B}$.

- Assume that a can reach a gridpoint b_2 as the furthest gridpoint on row $\mathcal{B}$. Then assume that a.`Furthest`[$\mathcal{B}$]$= b_1.column$, where $b_2 \neq b_1$. If $b_1.column < b_2.column$, then follow the proof of the first part of this proof, which leads to a contradiction. If $b_1.column > b_2.column$, then by the implication of the first part of this proof, there is a contradiction with the first assumption of this case. In both cases, a contradiction is reached. This proves that if b is the furthest gridpoint that a can reach on row $\mathcal{B}$, a.`Furthest`[$\mathcal{B}$]$= b_1.column$.

As both parts of the bi-implication are proven, the theorem holds and Algorithm 10 is correct. □

Conclusion. With the above algorithm, we are able to solve Problems 6 and 7 with a preprocessing time and space requirement of $O(n^3)$ and a query time of $O(1)$. This solution to Problem 7 will allow us to speed up Algorithm 8 without increasing preparation space, giving it a new query time of $O(\log n)$ instead of $O((\log n)^2)$. So far, we have not found a way to solve Problem 8 with a lower running time. We call the method using Algorithm 10 and 8 to solve Problem 8 the **Recursive Sweep Method**.

6.6.2 Bypassing the Reachability Oracle Completely

The method as described above is not compatible with the Free Space Grid Long Jump Method, as that one also depends on Algorithm 6. Luckily, we can bypass Algorithm 6 as well using recursive sweeps, leading to an alternative algorithm. This algorithm does not improve the preprocessing time, space or the query time when compared to Algorithm 6, but does not use the Reachability Oracle. It can then be used together with Algorithm 10 to speed up the Free Space Grid Long Jump Method. The algorithm is displayed in Algorithm 11, and works as follows:

- Like Algorithm 10, this algorithm does a recursive sweep from right to left, however this sweep goes upwards from a row $\mathcal{A}$. Every gridpoint b in the Free Space Grid gets their own array `FtR`, which saves the rightmost gridpoint per row $\mathcal{A}$ which reaches b.
- As this sweep goes upwards, the sweep deals with successors instead of predecessors. This keeps the running time $O(3N)$. A single sweep in this recursive sweep probably takes less time than a sweep in Algorithm 10, but there will be more of them, as there is no skipping vertices.

Algorithm 11 Alternative to Algorithm 6

```
1: procedure FurthestToReach2(Free Space Grid FSG)
2:     For every gridpoint b, initialize an O(n) array FtR with −1 in every cell, which saves the
   rightmost point a that can reach b on any row A. Assuming constant time pointer access, the
   rightmost point from a specific row A can be accessed in O(1) time. The total space taken to
   save the Free Space Grid will then be O(n^3).
3:     for every row A do
4:         create a variable i which is the column index of the point left of the leftmost processed
   point on row A. Initialize it to the rightmost point of A
5:         while i ≥ 0 do                          ▷ 0 being the index of the leftmost point of A
6:             create a list L which keeps track of the gridpoints to process.
7:             Take point a with index i from A and put it in L. Decrement i.
8:             while L is not empty do
9:                 Take the first gridpoint p of L and remove it from L
10:                for all successors q of gridpoint p (at most 3) do
11:                    if q.FtR[A]= −1 then
12:                        q.Furthest[A]= a.column
13:                        put q in L.
14:                    end if
15:                end for
16:            end while
17:        end while
18:    end for
19: end procedure
```

Running Time. The running time of this algorithm is again dependent on the amount of edges as all vertices are processed once. This gives it a total preprocessing time of $O(n^3)$ for our set system and a query time of $O(1)$. It uses $O(n^3)$ space.

Correctness of Algorithm 11 The correctness of Algorithm 11 is similar to the correctness of Algorithm 10:

Theorem 6 (Algorithm 11 Correctness)**.** *Algorithm 11 is correct:*

b.FtR[$\mathcal{A}$] of gridpoint b contains the (column coordinate) of point a on row $\mathcal{A}$ if and only if a is the furthest gridpoint on row $\mathcal{A}$ that can reach b.

Proof. Similarly to Algorithm 11, we prove both sides of the bi-implication.

- Assume for some gridpoint a_1 on row $\mathcal{A}$ and some gridpoint b that b.FtR[$\mathcal{A}$]$= a_2.column$, but there is another point a_2 on $\mathcal{A}$ with $a_1.column < a_2.column$. Like with Algorithm 10, this would mean $a_2.column$ would be saved in b.FtR[$\mathcal{A}$], which would prevent $a_1.column$ being saved there. This contradicts our first assumption. This proves that b.FtR[$\mathcal{A}$]$= a_2.column$ implies that a is the furthest gridpoint on row $\mathcal{A}$ that can reach b.

- Assume that the rightmost gridpoint on row $\mathcal{A}$ that can reach b is a_2. Then assume that $b.\texttt{FtR}[\mathcal{A}] = a_2.column$ for some other gridpoint a_2 on row $\mathcal{A}$, and that $a_1 \neq a_2$. Then if $a_1.column > a_2.column$, a contradiction can be derived like in the first case and if $a_2.column < a_1.column$, a contradiction can be derived by using the implication proven by the first case. This proves that if a is the furthest gridpoint on row $\mathcal{A}$ that can reach b, $b.\texttt{FtR}[\mathcal{A}] = a.column$.

As both parts of the bi-implication are proven, the theorem holds and Algorithm 11 is correct. □

Conclusion. With Algorithms 10 and 11, we can remove the use of a Reachability Oracle from the Free Space Grid Long Jump Method as well, which results in an improved Free Space Grid Long Jump Method, with a preprocess time of $O(n^3)$ per combination of trajectories, $O(n^2)$ total queries per trajectory combination and $O(n)$ query time per trajectory combination, which allows us to create a Set System in $O(m^2n^3)$ preprocessing time, with a query time of $O(mn)$ and a space requirement of $O(m^2n^3)$. We call the method using the two algorithms of this section with Algorithm 9 to create a Set System Oracle for our Set System the **Recursive Sweep Long Jump Method**.

6.7 Beyond Our Set System

The final question that remains is Question 5, as the methods described in this section have been made with our Set System in mind. It can still be applied on general Set Systems, however.

When considering general Set Systems with a set of d Candidate Center Curves $\mathcal{D}$, with n denoting the longest length of any candidate curve P or trajectory S and g denoting the longest length of any trajectory S, the following would change for the Free Space Grid Long Jump Method:

1. Instead of applying the method on all pairs of trajectories, the method would be applied on all combinations of center curves and trajectories to be covered, which would be $O(dm)$ combinations. The rest of the methods would be applied to the FSG grid created between one center curve and a trajectory to be covered.

2. The only two rows of note for the querying algorithm are the bottom and top rows of one of the FSG grids created out of a combination of a center curve F and a subtrajectory S. Therefore, while the $O(N)$ sweeps in the preprocessing methods in Sections 6.3 and 6.4 are still required for the logarithmic query time methods to work, the $O(n^3)$ preprocessing methods (like Algorithm 6) can be shortened to one sweep of $O(n)$ time, making the overall processing time of Free Space Grid Long Jump Method $O(n^2 \log n)$ for every Free Space Grid.

3. The Query Algorithms still work the same way.

For the Recursive Sweep Long Jump Method, both recursive sweeps will only have to be applied on one row, the top row for Algorithm 10 and the bottom row for Algorithm 11. This makes the total running time of both algorithms $O(n^2)$.

This would make finding whether a single vertex of a trajectory S is covered by a center curve P possible in $O((\log n)^2)$ query time and $O(n^2 \log n)$ preprocessing time and space with the Free Space Grid Method and $O(\log n)$ query time and $O(n^2)$ preprocessing time and space with the Recursive Sweep Method.

Finding all vertices of S covered by P possible in $O(n \log n)$ time and $O(n^2 \log n)$ preprocessing time and space with the Free Space Grid Long Jump Method and in $O(n)$ time and $O(n^2)$ preprocessing time and space with the Recursive Sweep Long Jump Method.

Lastly, finding all vertices on all trajectories covered by all center curves possible in $O(dm(n^2 \log n + n \log n) = O(dmn^2 \log n)$ time with the Free Space Grid Long Jump Method and in $O(dm(n^2 + n)) = O(dmn^2)$ time with the Recursive Sweep Long Jump Method.

Making a Set System Oracle of one combination of center curve and trajectory would take $O(n^2 \log n)$ amount of space for the Free Space Grid Long Jump Method, but as only the resulting

coverage information needs to be saved, which is a single bit of information per vertex on the trajectory, the space needed in total is $O(dmg + n^2 \log n)$ for that method. Similarly, the space for the Recursive Sweep Long Jump Method is $O(dmg + n^2)$.

7 Implementation

The implementation can be found at `https://github.com/No311/CurveClustering`.

The implementation makes use of the Chicago Map Trajectory Data from the paper of Ahmed, Karagiorgou, Pfoser and Wenk [2, 3]. Additionally, it makes use of the file `WrapLayout.java` *from* `https://tips4java.wordpress.com/2008/11/06/wrap-layout/`*, which is an adaptation of the standard FlowLayout used in java swing to makes it resizable.*

During the development of the methods, we have also created an implementation to test and visualise methods and trajectories. In this section, we will describe what you can do with it and how. First, we will give an overview of the main components of the application and what functions they have, after which we will go more in depth for every component where applicable.

We have put the pictures of the implementation in Appendix A

7.1 Overview of Main Components

There are 5 main components of the application:

1. The **Trajectory Tab**, which is pictured in Figure 11 and is the main tab of the application. On this tab, we can manipulate trajectories, from opening to editing.
2. **Free Space Grid Tabs**, of which an example is pictured in Figure 13. Free Space Grid Tabs are tabs where we visualise Free Space Grids and can query Reachability information and our Free Space Grid data structures as presented in Section 6.
3. **Set System Oracle Tabs**, of which an example is pictured in Figure 15. Set System Oracle Tabs contain coverage information, which we can use to execute Greedy Set Cover or which we can query interactively.
4. The **infoText** panel, which can be seen on the right of any tab and contains a log of activity.
5. The **menu**, from which Trajectories can be opened, created, saved and closed, tabs can be created and Greedy Set Cover can be executed.

7.2 The Trajectory Tab

On the Trajectory Tab, other than the menu and infoText field, which are visible on all tabs, there are two main components: the Trajectory Panel, where Trajectories are visualised and manipulated, and the Bottom Panel, where certain functions can be applied to Trajectories and some variables can be viewed or changed.

To manipulate trajectories, the Trajectory Tab has multiple features. Some of these features are called through the menu, but will be described here as the result will be found in the Trajectory Tab:

- Menu Functions:
  - From the menu, you can **Open Trajectories** through a File Dialog, which allows you to browse your files and multiselect the ones you want to open.

    As of now, the application does not have extended file support. The only file it supports is a text file with a format where every Trajectory vertex has it's own line, with an x-coordinate, y-coordinate and time coordinate separated by spaces.

The application does not support trajectories with more than two space dimensions as of now.

When a Trajectory is opened, it will automatically be added to the selection and visualised on the Trajectory Tab on the Trajectory Panel.

Note that trajectories can have wildly different coordinates. To mitigate this, the Trajectory Panel will always take the first vertex of the first Trajectory in the selection as the origin.

- From the menu, you can also **Save Trajectories**, which will output a file in the format described above.
- Additionally, you can **Close Trajectories** from the menu.
- The last menu function of importance is the function to create a **New Trajectory**. This will make a new empty Trajectory and add it to your selection.

- The Trajectory Panel is draggable and zoomable.
- On the Bottom Panel, there is a list of **Selected Trajectories**. You can multiselect trajectories on this list to visualise them on the Trajectory Panel. The number in brackets behind the trajectories is how many vertices they have.
- The Trajectory selected in the **Editable Trajectory** panel is the one selected to be edited when you have the **Editing** checkbox enabled. Editing controls are as follows:
  - By left clicking on the mouse or touchpad, a vertex is added at the end of the current editable trajectory.
  - By holding Shift, the Trajectory Panel will behave as it does when the Editing checkbox is not enabled, allowing you to quickly switch between dragging and adding vertices. (Note that you can also drag while editing but it will still create a vertex.)
  - By holding Alt, left-clicking the mouse will remove the last vertex from the current editable trajectory.
- The **Draw Size Slider** draws trajectories thicker or thinner depending on the value.
- **Show Grid** enables or disables the Grid in the Trajectory Panel. The Grid is drawn in dark and light grey. This difference in color is purely for counting purposes and clarity; whenever a grid block is referenced, it will mean the smaller grid blocks and not the larger grid blocks outlined with dark grey.
- **Units per Gridblock** shows the amount of units per small grid block. What unit that is depends on how the trajectories are made.
- The **Simplify** button applies a simplification function as implemented in `Algorithms/Simplification.java` to all selected trajectories. This simplification function needs a μ-value, which is a threshold for how far the vertices in the simplification can be from each other, in units.
- The **Sample** button applies a sampling function as implemented in `Algorithms/Sampling.java` to all selected trajectories. `Sampling.java` contains two sampling functions, one which samples relative to time (so every line segment has a similar amount of vertices sampled) and one that samples relative to space (so all sampled vertices are a similar distance away from each other when placed on the original trajectory). The space variant is called when the Sample button is pressed. Both variants need as input how many vertices the new trajectory will have.

The Trajectory Tab is mainly implemented in `Interface/GUIMain.java`. The Trajectory Panel is implemented in both `VisualPanels/VisualPanel.java` and `VisualPanels/TrajectoryPanel.java`. The load and save functions are implemented in `Interface/TrajectoryInitializer.java` and the sampling and simplification algorithms are implemented in the `Algorithms` folder. Trajectories and Vertices and Arrows are implemented in the `Objects` folder.

7.3 Creating a Grid Tab

To create a Grid Tab, the option **Create Free Space Grid** needs to be chosen in the menu. Note that to do this, at least one trajectory must be opened. This will open a wizard pictured in Figure 12. In the two lists, you choose the first trajectory F and the second trajectory S. The subtrajectories used as defining subtrajectories come from F. The vertices to be covered are the vertices of S. You can choose the same trajectory for both.

Choosing two trajectories enables you to enter the thresholds. You can enter multiple, and a Free Space Grid shall be created for each threshold, disregarding time and space limitations.

For a Free Space Grid, choosing a Reachability structure and a FSG structure is optional, although obviously the functionality of the two will not be present in the Free Space Grid tab created if none are chosen.

When done, click the "Create Free Space Grid" button, which will create a new Free Space Grid tab.

The Naive Reachability structure is implemented as described in Algorithm 5, in the file `Reachability/ReachabilityNaive.java`.

For the FSG structures, which are implemented in the map `FSGMethods`:

- The Naive Prep FSG structure is created through the **Naïve Shortcut Method Preprocessed**, which is the same method as the **The Naïve Shortcut Method** from Section 6.4.1, but all queries have been done during preprocessing, giving it a constant query time. It is implemented in the file `FSGMethodsPrepNaive.java`.
- The Naive No Prep FSG structure is created through the **Naïve Reachability FSG Method** from Section 6.4.1. It is implemented in the file `FSGMethodsNoPrepNaive.java`.
- The Naive Query structure is created through the **Naïve Shortcut Method** from Section 6.4.1, with query time $O(n)$. It is implemented in the file `FSGMethodQueryNaive.java`.
- The Log Query structure is created through the **Free Space Grid Method Optimized**, as described in depth Section 6. It makes use of Algorithms 6, 7 and 8.
- The Log Query (No Opt) is also created through the **Free Space Grid Method** but does not have the optimizations described in Section 6.3.

Note that the reason for the different names is that the names in the implementation are short but a bit less descriptive, which makes the wizard's size more manageable and the wizard's layout clearer. Also note that due to time constraints the **Recursive Sweep Method** from Section 6.6.1 is not implemented.

The Grid Wizard is implemented in the file `Wizards/FSGridWizard.java`

7.4 The Free Space Grid Tab

On the Free Space Grid Tab, there are again two main components, other than the infoText field and the menu: the Grid Visual Panel and the Bottom Panel. Again, the Grid Visual Panel is for the visualisation, this time a Free Space Grid. On the Bottom Panel, you can again change some variables and turn the Grid on and off. On the right side of the Bottom Panel, the trajectories chosen as first and second trajectories are shown, as well as the threshold. The first trajectory

is the row coordinate of the coordinates, the second trajectory the column coordinate of the coordinates.

The functions of this tab are as follows:

- In the Grid Visual Panel, red gridpoints do not exist, blue gridpoints exist, green gridpoints can be reached from the selected gridpoint and purple gridpoints can reach the selection. In a Query to the FSG structure, the furthest gridpoint the selected gridpoint on row $\mathcal{A}$ can reach on row $\mathcal{B}$ is colored a dark orange.
- The Grid Visual Panel can be dragged and zoomed, like all visual panels. Additionally, clicking on a gridpoint will display the reachability information by coloring all the gridpoints and edges that can reach the selected gridpoint and all the gridpoints and edges that can be reached from the selected gridpoint. This of course only works if the Reachability structure is prepared. The selected gridpoint is colored gold.
- If the Reachability structure is prepared, the button **Query Reachability Oracle** will initiate a Reachability Query, where you first select gridpoint a and then a gridpoint b in the Grid Visual Panel by clicking them, after which the infoText will display whether they can reach each other, as well as the Reachability information of a. Starting some other functions (like another kind of query) will cancel this query. A right mouse click will also cancel this query.
- If a Free Space Grid Method is prepared, clicking the **Query Free Space Grid Method Oracle** will initiate a query to the FSG structure prepared (if there is any). In the query, you will first have to click a gridpoint from row $\mathcal{A}$, then a gridpoint from row $\mathcal{B}$ and then a gridpoint from column s_p. The result will be printed in the infoText field and also visualised in the Grid Visual Panel. Starting some other functions (like another kind of query) will cancel this query. A right mouse click will also cancel this query.

The Free Space Grid Tab is implemented in `Tabs/GridTab.java`. The method to initialize the right Reachability Structure and the FSG structure is implemented in `Methods/SetSystemMethods.java` The Grid Visual Panel is implemented in `VisualPanels/VisualPanel.java` and `VisualPanels/FSGridPanel.java`. Gridpoints, grid edges and arrows are implemented in `Objects`, as is the matrix representation of the Free Space Grid in `FSGrid.java`. The Reachability structure and FSG structure are implemented in the files described in last section.

7.5 Creating a Set System Oracle Tab

To create a Set System Oracle Tab, the option **Initialize Set System** needs to be chosen in the menu. To do this, again, at least one trajectory must be opened. Choosing the option will open the wizard pictured in Figure 14. In the trajectory list, you can choose the trajectories you wish the tab to be initialized for. Then, you can choose to initialize the Set System Oracle with the Naive Method or the Free Space Grid Methods.

This wizard has some functionality to only make certain options available once its prerequisites are chosen. For the "Initialize Set System Oracle" button to become available, trajectories have to be chosen, a threshold needs to be specified and the methods used to create the Oracle also have to be specified (the Naive Method or the Free Space Grid Method, including Reachability, FSG Structure and Query Strategy).

The Naive Method structure is created through the **Naïve Dynamic Programming Method** as described in Section 5, which is implemented in the file `Querier/ExtremelyNaiveQuerier.java`.

The Reachability and FSG structures are created through the same methods and implemented in the same files as described in Section 7.3.

The Naive Query Strategy works as described in Section **??** and is implemented in the file `Querier/EasyQuerier.java`. The Long Jump Query Strategy also works as described in **??** and Algorithm 9, and is implemented in the file `Querier/LongJumpQuerier.java`. Note that using the Long Jump Query Strategy with the Log Query Free Space Grid Method results in a

structure implemented through the **The Free Space Grid Long Jump Method**, which makes use of Algorithms 6, 7 and 9, but not Algorithm 8. Also note that due to time constraints, the **Recursive Sweep Long Jump Method** from Section 6.6.2 has not been implemented or used. All the algorithms are described in Section 6.

7.6 The Set System Oracle Tab

The main components on the Set System Oracle Tab, pictured in Figure 15, other than the menu and the infoText field, are the Set System Oracle Visual Panel and the Bottom Panel. The Set System Visual Panel works much like the Trajectory Panel of the Trajectory Tab, except that is has some added functionality and cannot be used for editing or selecting which trajectories to view. The Bottom Panel can again be used to manipulate the draw size and show or hide the grid, with the added functionality of facilitating the Subtrajectory Covering query on the left side. The right side of the Bottom Panel again shows the shown trajectories and the threshold.

The added functionality of this tab is as follows:

- By selecting vertices on the Set System Oracle Visual Panel, you can query the Set System Oracle. Left clicking on a vertex will select that vertex as the subtrajectory you query. Selecting another vertex will select that vertex instead of the current vertex.

  Having Shift pressed while selecting a vertex x from the trajectory you have just selected subtrajectory (a, b) from extends the subtrajectory with a as the focus: if $x > b$ or $a \leq x \leq b$, b is set to x. If $x < a$, a is set to x. Having Alt pressed does the same, but with b as the focus. If $a \leq x \leq b$, a is set to x instead of b.

  Having Control pressed will query a new subtrajectory on top of those which have already been queried. Note that only the last created subtrajectory can be customized with Alt and Shift.

  Selected vertices are colored red, like the begin and end vertices of selected subtrajectories. The rest of the selected subtrajectories are yellow in color (vertices darker yellow, edges yellow). Covered vertices and edges are visualised in green. Unrelated vertices and edges are blue.

- Clicking the button **Subtrajectory Covering** will start the execution of Greedy Set Cover with the current Set System Oracle. Limits on the lengths of considered subtrajectories can be filled in in the block "ℓ-values used for Subtrajectory Covering". If the value for the minimum ℓ is left empty, it is set to 0, which is the length of subtrajectories of single vertices, and if the value for the maximum ℓ is left empty, it will be set to the length of the longest subtrajectory possible. The result of the execution of Greedy Set Cover is visualised and printed in infoText.

The Set System Oracle Tab is implemented in the file `Tabs/OracleTab.java`. The methods to initialize the Oracle are implemented in `Methods/SetSystemMethods.java`. The Set System Oracle Visual Panel is implemented in `VisualPanels/VisualPanel.java` and `VisualPanels/SetTrajectoryPanel.java`. Greedy Set Cover is implemented in `Algorithms/GreedySetCover.java`. The structures are implemented in the files described in last sections.

7.7 Doing Greedy Set Cover Without A Tab

To do Greedy Set Cover without initializing a tab, you can choose the option **Do Greedy Set Cover** in the menu, after which you will see the wizard pictured in Figure 16 The way this wizard works is identical to the Set System Oracle Wizard for the elements the two wizards share, and the ℓ-values work the same as on the Set System Oracle Tab.

Once you have pressed the "Do Greedy Set Cover" button at the bottom of the wizard, the program will start computing feverishly, after which it will output the results in the infoText field. Currently, the result cannot be saved or imported from a file.

The Greedy Set Cover Wizard is implemented `VisualPanels/GreedySetWizard.java` and `VisualPanels/OracleWizard.java`.

8 Experimentation

8.1 Tested Methods

In this section, we test the running time of the methods with several trajectories simplified or combined to see how the methods cope with an increasing amount of vertices. Amongst these trajectories is one combination of trajectories especially made to be a difficult case for the Free Space Grid Method, and one combination of vertices especially made to be a difficult case for the Free Space Grid Method with optimizations.

The solutions tested are as follows:

- **The Free Space Grid Method**, as described in depth Section 6. It makes use of Algorithms 6, 7 and 8. This variant does not use optimizations.
- **The Free Space Grid Method Optimized**. Like the last method, this method makes use of Algorithms 6, 7 and 8. The Optimizations are described in Section 6, but to summarize: for every gridpoint x, the gridpoint x_s on the same row with the smallest column coordinate that can reach x and the gridpoint x_ℓ on the same row with the largest column coordinate that x can reach are saved. These are then used to shrink the points of interest POI and points to query PTQ even further.
- **The Free Space Grid Long Jump Method**. This method makes use of Algorithms 6, 7 and 9, which are all described in Section 6. It also includes the optimizations described above.
- **The Naïve Dynamic Programming Method** from Section 5.
- **The Naïve Reachability FSG Method** from Section 6.4.1.
- **The Naïve Shortcut Method** from Section 6.4.1, with query time $O(n)$.
- **The Naïve Shortcut Method Preprocessed**, which is the same method as the **The Naïve Shortcut Method**, but all queries have been done during preprocessing, giving it a constant query time.

8.2 Test Comments

We deemed testing the method we theoretically use for Reachability [14] not the focus of this thesis and we have not implemented that method in this project (also because of time constraints). Therefore, we use a placeholder method. Sadly, the preprocessing time and space requirement of this placeholder method is $O(n^4)$ and it is used in all methods but the Naïve Dynamic Programming Method.

The Space data given is not precise but an indication. This has to do with the placeholder method, which we suspect skews the space results somewhat. Due to the way Java's garbage collector works, we have not yet been able to study that more in depth.

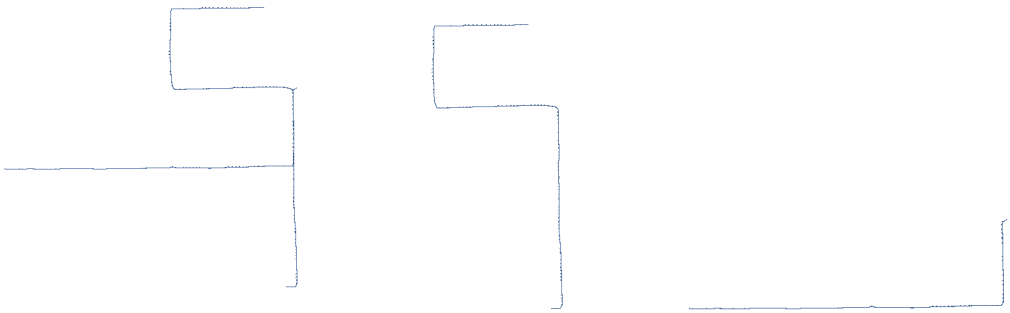
We believe that to properly test the implementation, at the very least the theoretically used Reachability Method [14] should be implemented, but we were unable to do that in this project. Nevertheless, we hope the tests give an insight in the already implemented methods. Note that the methods described in Section 6.6 have not been tested due to time constraints.

8.3 Test Environment

The tests have been run on a Lenovo Thinkpad W541, with Windows 10 64 bit, with an Intel Core i7-4710MQ processor and 8 GB RAM. The program is written and run in IntelliJ 2020.3, with Java SE Development Kit 15 and a heap space limit of 5012 MB.

8.4 Test Results

8.4.1 Comparing the Naive Dynamic Programming Method to the Free Space Grid Long Jump Method



(a) Both Trajectories (b) Trajectory 1, `trip_0` [2, 3] (c) Trajectory 2, `trip_1` [2, 3]

Figure 9: The Trajectories used for testing the Naïve Dynamic Programming Method against the Free Space Grid Long Jump Method, from the Chicago Data Set [2, 3].

To test the scalability of the Naïve Method and get an indication of how it performed against our best Free Space Grid Method, we sampled the trajectories shown in Figure 9 (`trip_0` and `trip_1` from the Chicago data set [2, 3]) with increasing amounts of vertices. We then tested the Naïve Dynamic Programming Method and the Free Space Grid Long Jump Method by doing Greedy Set Cover on the sampled trajectories.

Procedure. For every method, we launched the program, after which we opened the trajectories and sampled them with intervals of 25 vertices (25, 50, 75, etc.). Then, we applied Greedy Set Cover via the menu option `Do Greedy Set Cover` on them without specifying ℓ-values, as that means that all possible subtrajectories need to be considered.

Results. The results are shown in Table 1. The time and space used for the initialization of the Oracle and the execution of Greedy Set Cover are added together in the results. The threshold for all rows is 250, and the ℓ-values are undefined.

	Naïve Dynamic Programming Method			Free Space Grid Long Jump Method	
Vertices	Running Time (s)	Initial Used Memory (MB)	Post-Execution Used Memory (MB)	Running Time (s)	Matching Results
25	0.197	11	15	0.327	Yes
50	5.253	11	52	0.587	Yes
75	42.581	11	182	1.366	Yes
100	242.008	11	520	3.096	Yes
125	942.454	11	1160	6.186	Yes
150	Heap Space Error			11.696	-

Table 1: Comparing the Naïve Dynamic Programming Method to the Free Space Grid Long Jump Method (with threshold 250 and no ℓ-values.

Discussion of Results. As you can see, the Naïve Method takes longer and takes up more space than the Free Space Long Jump Method (as that method can actually handle 150 vertices).

8.4.2 Comparing the Scalability of the Free Space Grid Methods

To test the scalability of the Free Space Grid Methods, we have tested the amount of trajectories and the amount of vertices that the Free Space Grid methods can handle, for all Free Space Grid Methods.

For the Trajectories Scalability Tests, made a trajectory consisting of a single line from $(0, 0)$ to $(0, 10000)$, sampled to contain 100 vertices, of which we opened an increasing amount of copies. Greedy Set Cover is applied with 2 different thresholds, 1 and 10000, to get the sparsest Free Space Grid possible (that is not empty and does have a vertex on every row) and the fullest Free Space Grid possible, to see the impact of the amount of gridpoints in the Free Space Grid has. Of course, more gridpoints will probably mean a longer running time.

For the Vertices Scalability Tests, we used the trajectory consisting of a single line from $(0, 0)$ to $(0, 10000)$ and sampled it for an increasing amount of vertices.

Procedure. First we sampled the trajectory for the vertices scalability tests and saved copies of the trajectory for the trajectories scalability tests. For every method with either threshold, we launched the program, after which we performed the trajectory scalability tests one by one by applying Greedy Set Cover via the menu option `Do Greedy Set Cover` (no specified ℓ-values). In the same way, for every method with either threshold, we launched the program, opened the vertices scalability tests trajectories and performed the vertices scalability tests one by one by applying Greedy Set Cover on the trajectory with the right amount of vertices (and only that one) via the menu option `Do Greedy Set Cover` (no specified ℓ-values).

Results. The results of both the scalability with regards to trajectories and to vertices are shown in the tables in Appendix B, which have also been plotted and visualised in Appendix B. For the trajectories scalability tests, the space information has not been included. This is because the space information we recorded at the time turned out to be not very useful, there were no Heap Space Errors and there were time constraints to be considered.

Heap Space Error did occur in the vertices scalabilities tests, so we added the maximum used MBs of any one creation of a Reachability, FSG or Query structure. Note that the java garbage collector might have collected during the creation of the structures, so the amount used is only an indication. We did not induce the amount of MBs used during the execution of Greedy Set Cover as it was consistently 1 MB or less during the vertices scalability tests.

Additionally, as it was hard to get the information we wanted from the output at the time we did the trajectories scalability tests by hand, we wrote a program to extract it for us, which is found in the implementation under `Processable/GetRunningTimes.java`. We have since corrected this problem so the program is not needed for future tests (including the vertices scalabilities tests).

The Total Running Time in the tables is the total time it takes to initialize the Set System Oracle added to the time it takes to do Greedy Set Cover.

Discussion of Results. For the Trajectories Scalability Tests, it can be concluded that the amount of vertices in the Free Space Grid does indeed have an impact on the running times, as we thought.

Additionally, it proves that the Reachability structure takes the longest to prepare, which is not surprising as we use a placeholder method, with one exception: the Naïve Reachability FSG Method, where the Querying Time is far larger than the Reachability Time for filled Free Space Grids. This makes sense when you consider that it uses the Naïve Query method, where every single vertex on S has to be queried for every subtrajectory of F, which takes $O(n^2 \cdot n \cdot n^2) = O(n^5)$ time in total.

Surprising is that about half of the running time of the program is not accounted for in preparation times of the structures and the application of Greedy Set Cover. This might be because we call the garbage collector outside of structure initialization times. Other than that, we mostly prepare new iterations for new combinations of trajectories outside of structure preprocessing times. Also surprising is that the Query Time of the Free Space Grid Method Optimized took only 0.633 seconds with 13 trajectories, where it took 2.005 seconds with 12. We have no real explanation for that and assume it to be an outlier. It can be seen in Figure 19, along with another similar outlier. We have double checked them both and as far as we know no mistake has been made in inputting the information in the table. The possibility for human error is still there, however. It could also have been caused by the computer we ran things on, as sudden changes in speed have been seen in other cases, as can be seen in the graphs.

Lastly, the Long Jump Method is not significantly faster than the Free Space Grid Method Optimized in the results for threshold 10000, which is interesting. The structure preprocessing times do show that the Query Time is significantly reduced between the Free Space Grid Method Optimized and the Long Jump Method, however, which makes sense: in a completely filled Free Space Grid, the Long Jump Method would only have to do one query per subtrajectory, where the Free Space Grid Method optimized would still need to do $O(n)$. The other structures' preprocessing times are around the same in both methods.

Overall, the results do show that the Free Space Grid Methods are a better choice than the naïve methods when considering trajectory scalability.

For the Vertices Scalability Tests, it becomes clear that the methods all use a lot of memory. We do not think the amount of MBs reported for 200 vertices is too reliable; we expect some garbage collection to already happen during the preprocessing, as the space used by the Query Method and the Free Space Grid Method suddenly jumps to an equal level as the Reachability Space used, where before there used to be a gap.

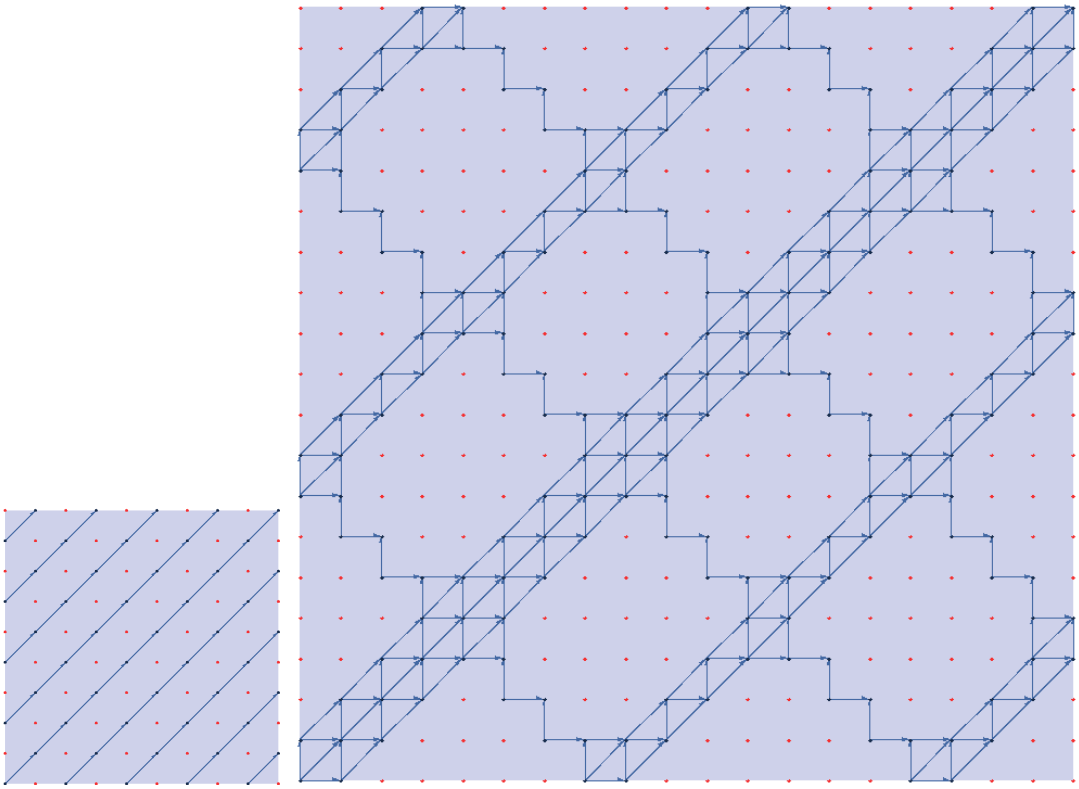
We have already done some memory optimization, but it seems more is needed. We think that changing the methods to work with a set of center curves and a set of to be covered trajectories might be better, because while we need to use the Free Space Grid Method more the amount of memory used per iteration of the Free Space Grid Method will decrease significantly. This also depends on the amount of vertices of a center curve compared to the trajectories to be covered for the Reachability structure, but for now we assume that the center curves are in general meant to be shorter than the trajectories. For the Free Space Grid structure and the Query structure, this decrease in memory used is almost guaranteed; as stated before, only one subtrajectory needs to be considered, which is the entire center curve.

The running times are not too surprising: preparing the Reachability Structure still takes the longest and overall the optimizations and the Free Space Method seem to be faster than the non-optimized methods and the Naïve Methods.

8.4.3 Optimalization Test: Diagonals Only FSG

To highlight the differences between the Free Space Grid Method and the Free Space Grid Method optimized, we have devised a trajectory which works as follows:

1. Vertex 1 is at $(0, 0)$
2. Vertex 2 is at $(0, 10000)$
3. Vertex 3 is at $(0, 0)$
4. Vertex 4 is at $(0, 10000)$
5. and so on.



(a) Initial FSG Grid Pattern

(b) Sampled FSG Grid Pattern

Figure 10: Diagonally focused Free Space Grids

With the above trajectory with 10 vertices, this will result in the Free Space Grid as pictured in Figure 10a. For this Free Space Grid, the optimizations of the Free Space Grid Method Optimized should not matter: every vertex can only reach or be reached by itself on the same row. Therefore, the Free Space Grid Method Optimized should not be significantly faster than the Free Space Grid Method in this case. However, if we sample the trajectory and set the threshold to something that would allow some vertices to be within Discrete Fréchet Distance of their neighbours, like 1000, this would mean that on the sampled trajectory, the Optimized Method should be faster than the original method, while there is no change when we take the above method and create a trajectory of 100 vertices. An example of a sampled Free Space Grid (6 original vertices, 20 sampled, threshold 3000) is shown in figure

Procedure. To test this, we prepared trajectories of 10, 50, 100, 150 and 200 vertices as explained before, which we call the extended trajectories. Additionally, we made sampled variants of the trajectories with 50, 100, 125, 150, 175 and 200 vertices from the trajectory with 10 vertices. Then, for both sets of trajectories and both methods (optimized and non-optimized, both with a naive query method) We then applied Greedy Set Cover with threshold 1000 (which we think sufficiently big for an effect, but not big enough to fill the entire FSG immediately) via the menu

option `Do Greedy Set Cover` (no specified ℓ-values). We did not test larger trajectories due to heap space errors.

Results. The results are shown in Table 2.

	Extended Trajectories			
	Free Space Grid Method		Free Space Grid Method Optimized	
Vertices	Query Time (s)	Total Running Time (s)	Query Time (s)	Total Running Time (s)
10	0.001	0.12	0.002	0.119
50	0.016	0.28	0.015	0.282
100	0.071	1.717	0.067	1.721
125	0.15	3.539	0.153	3.567
150	0.269	7.046	0.242	7.028
175	0.451	14.504	0.4	12.628
200	0.673	25.861	0.631	26.637
	Sampled Trajectories			
	Free Space Grid Method		Free Space Grid Method Optimized	
Vertices	Query Time (s)	Total Running Time (s)	Query Time (s)	Total Running Time (s)
10	0.001	0.132	0.001	0.12
50	0.008	0.246	0.008	0.254
100	0.043	1.67	0.029	1.573
125	0.082	3.485	0.072	3.636
150	0.136	6.545	0.093	6.656
175	0.237	11.548	0.137	11.341
200	0.369	25.031	0.204	23.872

Table 2: Comparing the Free Space Grid Method and the Free Space Grid Method Optimized on Sampled and Extended Free Space Grids with Diagonals.

Discussion of Results. As can be seen in the table, the Query running times for the extended trajectories are more comparable between the Free Space Grid Method and the Free Space Grid Method Optimized than the Query running times for the sampled trajectories, which begin to diverge somewhat substantially for 150 vertices or more. Still, this should be tested again when more trajectories more than 200 vertices can be considered, as while this seems to support the hypothesis of the tests, this evidence alone is not exactly conclusive (the difference in running times can also be caused by the computer it is run on).

9 Conclusion & Discussion

9.1 Conclusion

We have investigated the use of Free Space Grids for creating a Set System Oracle for solving Subtrajectory Covering with Greedy Set Cover. To do this, we have defined a Set System which takes all subtrajectories of all trajectories as potential center curves, created a Naïve Set System Oracle creation method using Dynamic Programming, and used Free Space Grids to create a method which can be used to create the Set System Oracle in $O(m^2n^3)$ time and $O(m^2n^3)$ preprocessing space. We have described how to extend our methods beyond our Set System to the more general

case of having a Set System with a set of center curves and a set of trajectories to be covered, where the fastest method will take $O(dmn^2)$ time and $O(dmg + n^2)$ space.

Additionally, we have implemented an application which implements most of the methods described and used in theory, of which we have tested the running times and to a lesser extent the space requirements.

We think the results do good work towards the goal of finding efficient methods for the creation of Set System Oracles, especially in the context of Free Space Grids and the Discrete Fréchet Distance. There is room for further investigation, especially with general Set Systems, but the new methods we already found work and are faster than the naïve solutions we found in the same context (Free Space Grids and polygonal curves).

While we have not looked into it too much, we are unsure if there are methods for our Set System which use Free Space Grids with a faster computational complexity than presented in this thesis, as in our Set System the total amount of candidate curves per combination of trajectories F and S is $O(n^2)$ and the amount of points to be covered is $O(n)$, which means that the amount of queries that has to be computed per combination of trajectories for the creation of the Set System Oracle is $O(n^3)$, which also has to be saved. For faster computational complexities you would have to derive some of the answers to queries without actually computing them.

We are nevertheless happy we used our Set System and the results we found for it, as focusing on this Set System made the topic a bit less abstract, which was very useful especially during the beginning of the project. We are also happy with the other results, as creating those results are often what led to further insights, and the application, as implementing results helped us to work out the details.

9.2 Future Work

The project has the following next steps:

Firstly, the method as defined in Section 10. We were not able to do this due to time constraints.

Another next step for this project would be the implementation of the method described in Section 3.2.1 and in the paper by Thorup [14]. This has not been done as of yet as the application was meant as a proof of concept for the methods created in the project, and the method described in the paper by Thorup [14] is extensive and not created during this project, which meant it was not a priority. Implementing this, however, would allow us to test the implementation better and more meaningfully, for example against the method detailed in Section 10.

A step after that would be space optimization. One way of doing this is by converting the entire application to work with the more general Set System which has a set of center curves and a set of to be covered trajectories, which we think will decrease the space requirement but can impact the running time negatively.

Another step for the application would be the implementation of wider file support and editing options, as well as other convenient functions. Being able to create a Free Space Grid instead of having to generate one would make it easier to test the Free Space Grid methods. Additionally, we could add Set System generation methods, which would generate a set of center curves to be used with the Free Space Grid methods.

We could also implement the automatic use of simplification and sampling for cases that exceed the available space.

Theoretically, a next step would be looking into how to make sets of center curves for Set Systems that are as small as possible. With general set systems, the amount of center curves is an important factor for the complexity. With our Set System we do not think we will be able to improve the running time and space much.

Other steps could be to try to expand the theoretical method to continuous Fréchet Distance or to allow the threshold as a variable.

A last step would be investigating how to minimize Free Space Grids further for the general Set System variants of our method, as there is only one row combination for those Free Space Grids, which means that compressing the rest can lead to faster methods.

References

[1] Pankaj K. Agarwal, Kyle Fox, Kamesh Munagala, Abhinandan Nath, Jiangwei Pan, and Erin Taylor. Subtrajectory clustering: Models and algorithms. In Jan Van den Bussche and Marcelo Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 75–87. ACM, 2018. doi:10.1145/3196959.3196972.

[2] Mahmuda Ahmed, Sophia Karagiorgou, Dieter Pfoser, and Carola Wenk. A comparison and evaluation of map construction algorithms. *CoRR*, abs/1402.5138, 2014. URL: http://arxiv.org/abs/1402.5138, arXiv:1402.5138.

[3] Mahmuda Ahmed, Sophia Karagiorgou, Dieter Pfoser, and Carola Wenk. A comparison and evaluation of map construction algorithms using vehicle tracking data. *GeoInformatica*, 19(3):601–632, 2015. doi:10.1007/s10707-014-0222-6.

[4] Hugo A. Akitaya, Frederik Brüning, Erin Chambers, and Anne Driemel. Covering a curve with a set of subtrajectories. *EuroCG*, 2021.

[5] Kevin Buchin, Maike Buchin, David Duran, Brittany Terese Fasy, Roel Jacobs, Vera Sacristán, Rodrigo I. Silveira, Frank Staals, and Carola Wenk. Clustering trajectories for map construction. In Erik G. Hoel, Shawn D. Newsam, Siva Ravada, Roberto Tamassia, and Goce Trajcevski, editors, *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2017, Redondo Beach, CA, USA, November 7-10, 2017*, pages 14:1–14:10. ACM, 2017. doi:10.1145/3139958.3139964.

[6] Kevin Buchin, Maike Buchin, Joachim Gudmundsson, Jorren Hendriks, Erfan Hosseini Sereshgi, Vera Sacristán, Rodrigo I. Silveira, Jorrick Sleijster, Frank Staals, and Carola Wenk. Improved map construction using subtrajectory clustering. In Panagiotis Bouros, Tamraparni Dasu, Yaron Kanza, Matthias Renz, and Dimitris Sacharidis, editors, *LocalRec'20: Proceedings of the 4th ACM SIGSPATIAL Workshop on Location-Based Recommendations, Geosocial Networks, and Geoadvertising, LocalRec@SIGSPATIAL 2020, November 3, 2020, Seattle, WA, USA*, pages 5:1–5:4. ACM, 2020. doi:10.1145/3423334.3431451.

[7] Kevin Buchin, Maike Buchin, Joachim Gudmundsson, Maarten Löffler, and Jun Luo. Detecting commuting patterns by clustering subtrajectories. *Int. J. Comput. Geom. Appl.*, 21(3):253–282, 2011. doi:10.1142/S0218195911003652.

[8] Maike Buchin, Bernhard Kilgus, and Andrea Kölzsch. Group diagrams for representing trajectories. *Int. J. Geogr. Inf. Sci.*, 34(12):2401–2433, 2020. doi:10.1080/13658816.2019.1684498.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[10] Anne Driemel and Sariel Har-Peled. Jaywalking your dog: Computing the fréchet distance with shortcuts. *SIAM J. Comput.*, 42(5):1830–1866, 2013. doi:10.1137/120865112.

[11] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

[12] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. Trajectory clustering: a partition-and-group framework. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 593–604. ACM, 2007. doi:10.1145/1247480.1247546.

[13] Maurizio Talamo and Paola Vocca. An efficient data structure for lattice operations. *SIAM J. Comput.*, 28(5):1783–1805, 1999. doi:10.1137/S0097539794274404.

[14] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, 2004. doi:10.1145/1039488.1039493.

Appendices

A Implementation Pictures

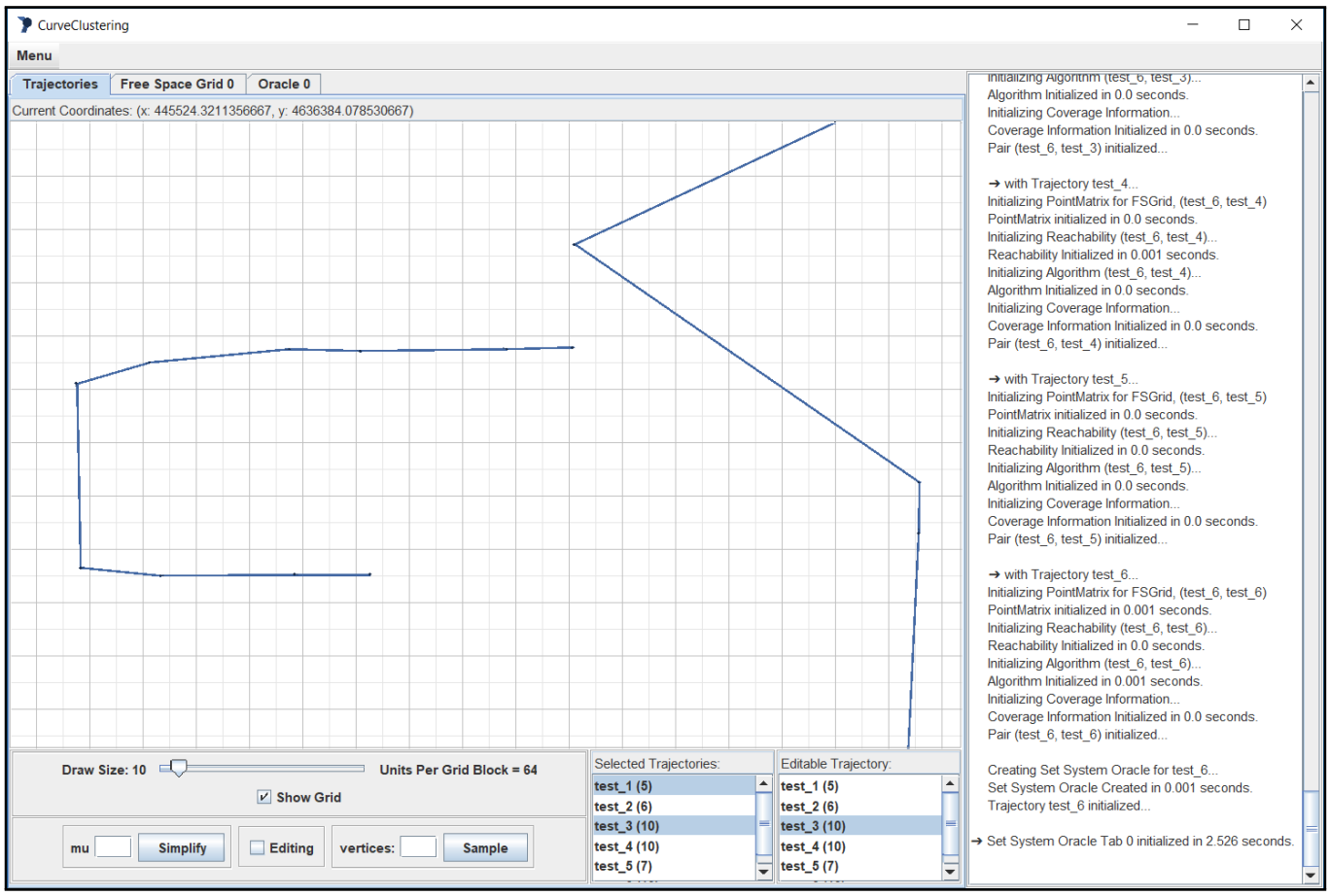


Figure 11: The Main Tab of the Application

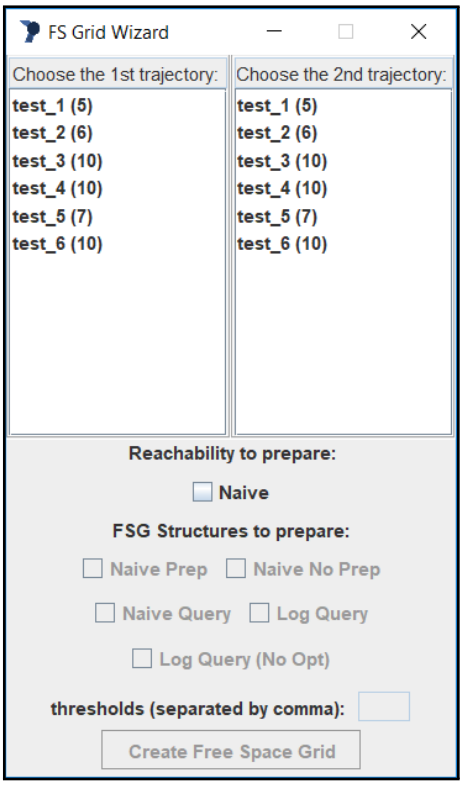


Figure 12: The Free Space Grid Wizard

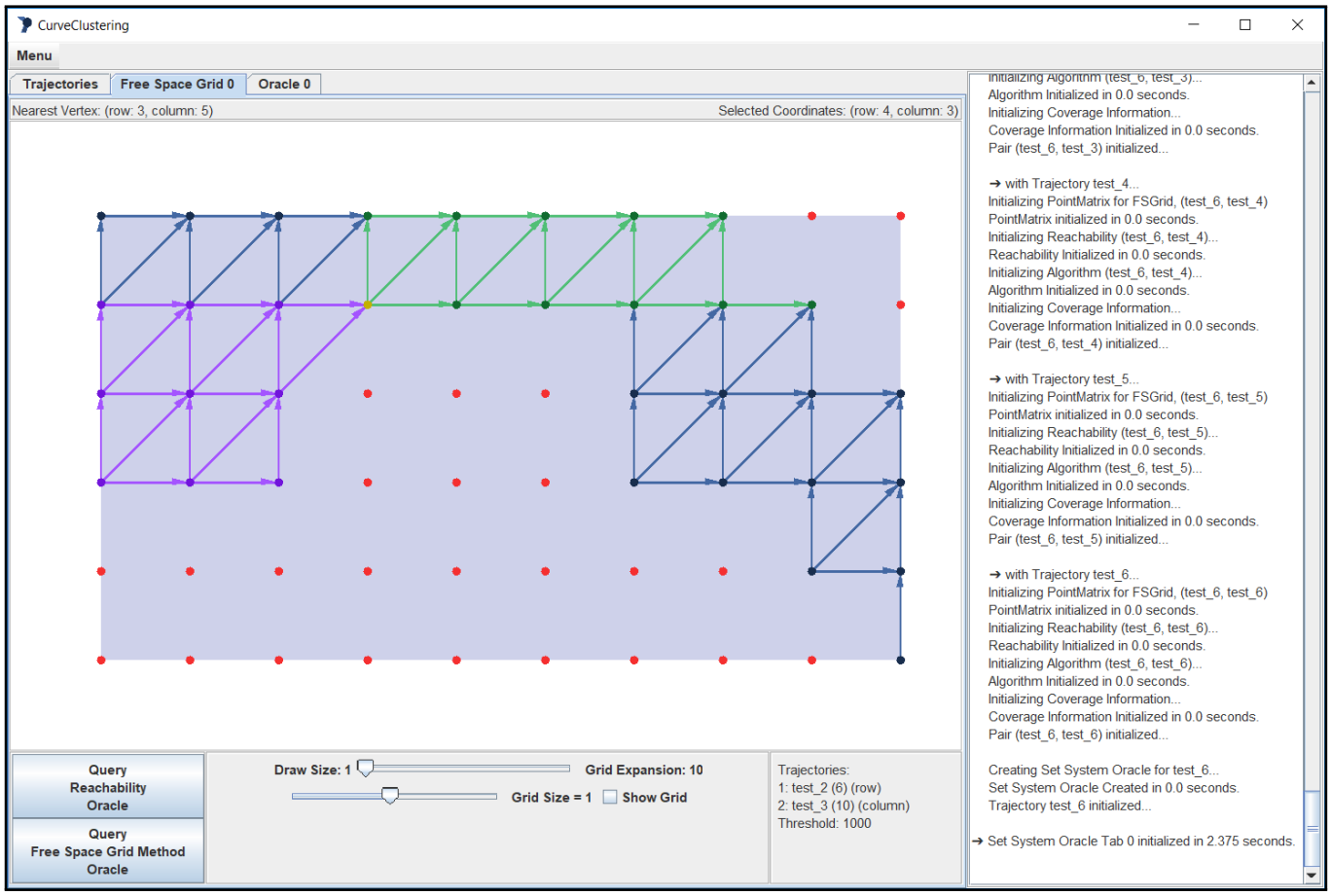


Figure 13: A Free Space Grid Tab of the Application

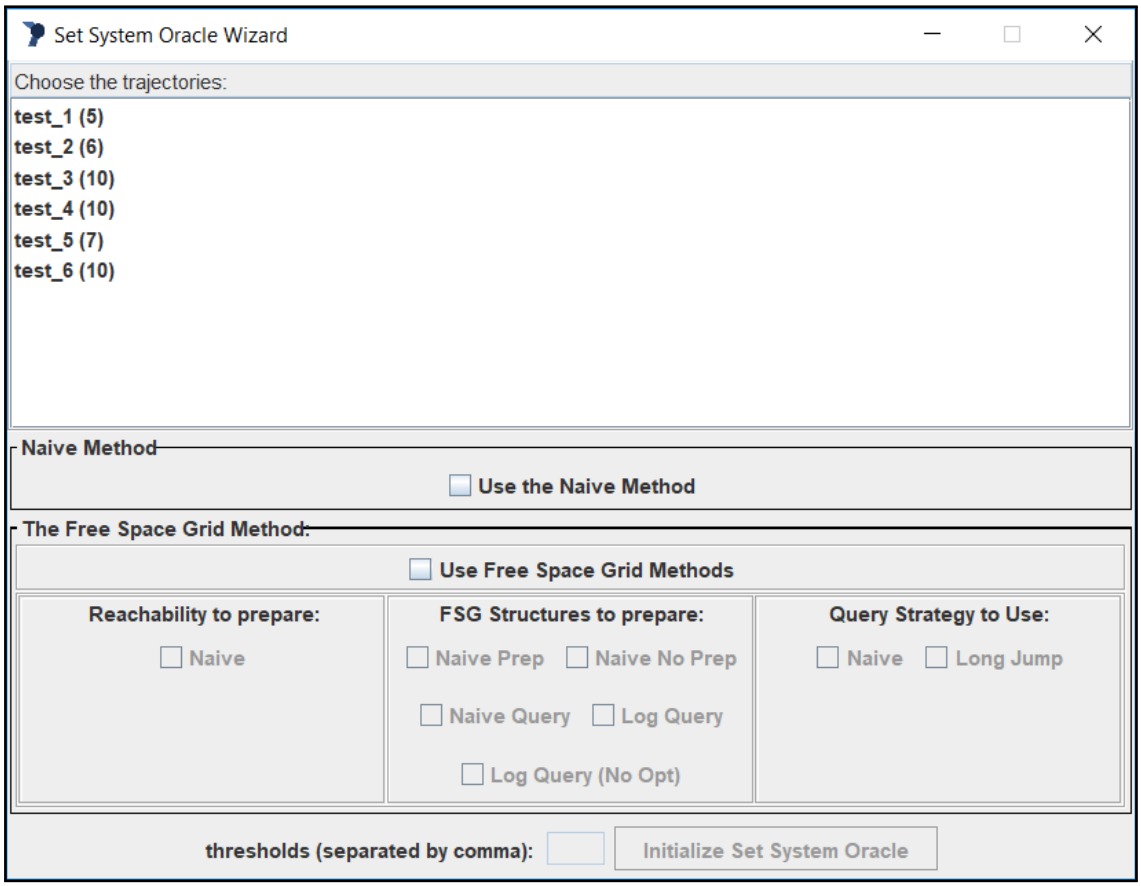


Figure 14: The Set System Oracle Wizard

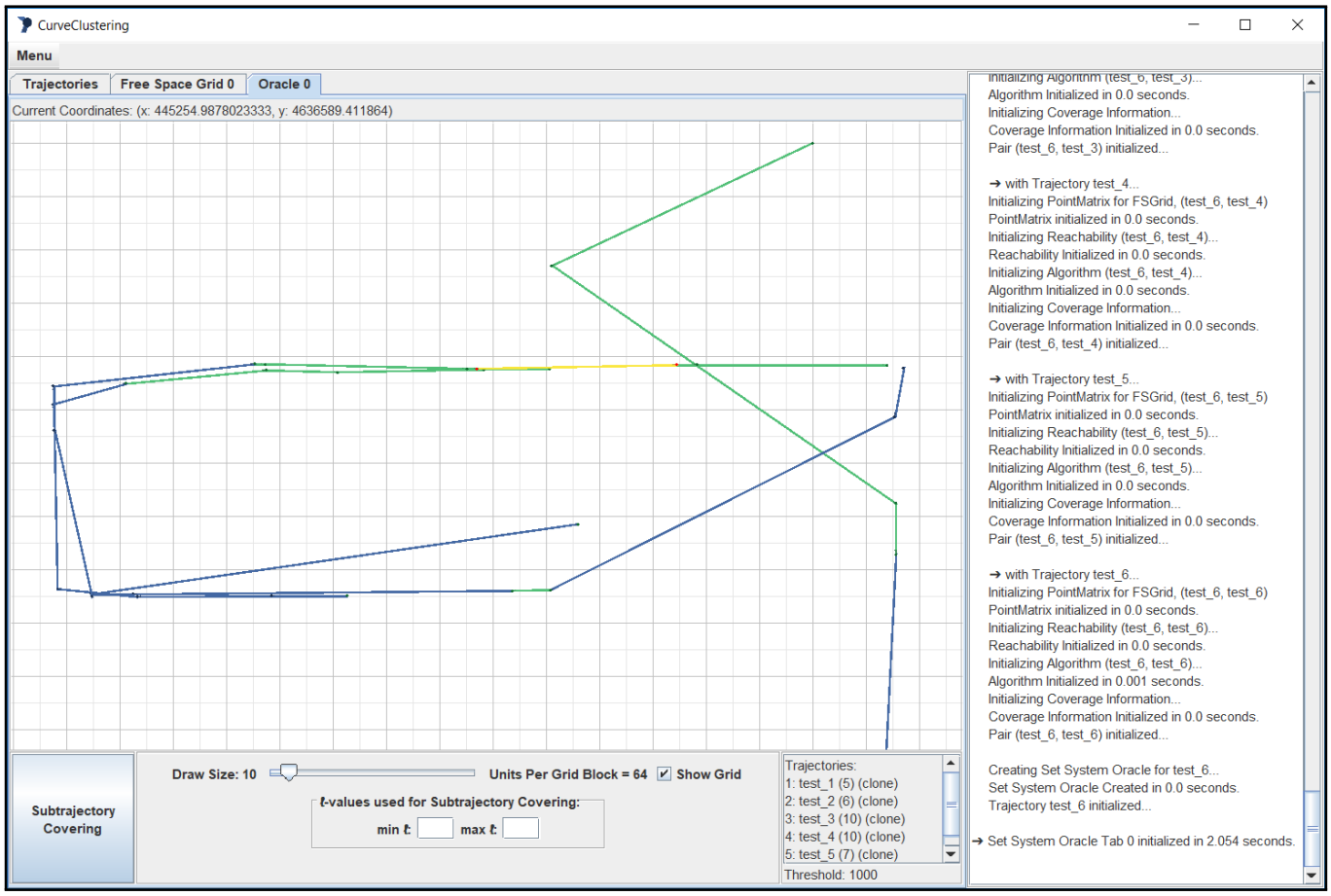


Figure 15: An Oracle Tab of the Application

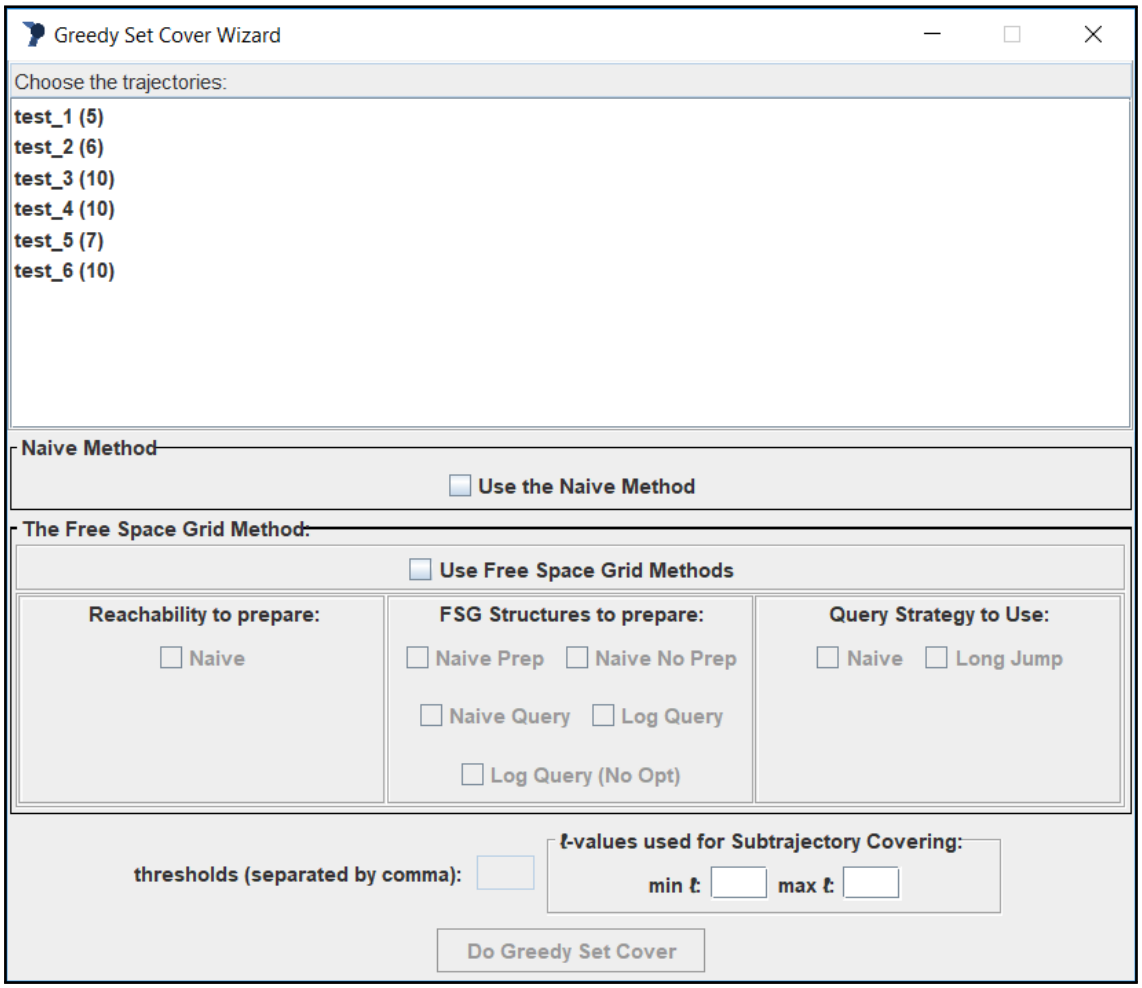


Figure 16: The Greedy Set Cover Wizard

B Scalability Results

B.1 Trajectories

Note that all trajectories in this subappendix are the trajectory from $(0, 0)$ to $(0, 10000)$, sampled to have 100 vertices.

B.1.1 Tables

	Naïve Reachability FSG Method, threshold 1				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	0.952	0.002	0.166	0.083	1.546
2	3.564	0.013	0.17	0.161	5.26
3	4.175	0.024	0.388	0.371	7.925
4	6.56	0.034	0.667	0.617	13.243
5	10.087	0.052	1.032	0.97	20.748
6	14.729	0.081	1.48	1.491	30.481
7	19.946	0.116	2.021	2.272	42.247
8	25.739	0.16	2.651	2.936	55.553
9	34.523	0.19	3.554	3.712	73.742
10	39.383	0.216	4.142	4.644	88.423
11	49.5	0.241	5.152	5.549	111.025
12	59.498	0.272	6.279	6.603	136.236
13	68.091	0.326	7.286	8.425	161.308
14	79.157	0.343	8.45	9.59	191.184
15	89.836	0.38	9.678	10.339	222.831
16	99.682	0.467	10.979	12.207	256.891
17	111.785	0.54	12.425	13.396	295.064
18	125.035	0.575	14.156	14.762	341.369
19	133.518	0.639	15.671	17.229	384.413
20	159.374	0.815	19.246	18.67	457.6
	Naïve Reachability FSG Method, threshold 10000				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	1.609	0.014	1.686	0.071	3.788
2	6.32	0.03	6.608	0.189	14.751
3	10.911	0.062	15.446	0.384	30.567
4	13.524	0.086	27.102	0.645	47.718
5	23.586	0.177	42.276	1.016	77.562
6	34.557	0.276	62.115	1.518	114.221
7	43.009	0.309	80.504	2.249	146.142
8	56.299	0.394	105.961	2.854	192.438
9	76.323	0.539	139.547	3.793	256.607
10	91.11	0.642	169.136	4.172	310.869
11	107.855	0.688	203.176	5.589	372.802
12	125.704	0.799	237.895	6.603	438.485
13	155.645	1.073	287.973	8.031	539.385
14	187.439	1.257	341.32	8.861	644.994
15	193.749	1.216	374.892	10.286	700.440
16	231.349	1.498	437.458	11.943	829.788
17	261.964	1.705	502.44	14.037	957.649
18	285.494	1.864	555.39	15.237	1062.321
19	300.377	1.847	606.019	17.142	1155.309
20	338.457	2.202	683.668	19.204	1313.857

Table 3: Testing the Trajectories Scalability of the Naïve Reachability FSG Method

	Naïve Shortcut Method, threshold 1				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	0.914	0.03	0.04	0.069	1.381
2	3.512	0.062	0.074	0.161	5.15
3	4.172	0.105	0.165	0.368	7.844
4	6.289	0.18	0.282	0.628	12.787
5	9.925	0.289	0.46	0.977	20.417
6	14.492	0.39	0.631	1.456	29.767
7	19.421	0.585	0.86	2.273	41.226
8	27.082	0.949	1.207	3.017	58.156
9	32.219	0.996	1.519	3.77	71.004
10	40.806	1.233	1.893	4.823	90.999
11	50.861	1.403	2.376	5.806	113.914
12	56.884	1.592	2.652	6.723	132.011
13	63.986	1.732	3.015	7.721	152.891
14	73.791	1.94	3.459	9.011	180.664
15	85.868	2.19	4	10.726	214.394
16	96.033	2.527	4.583	11.739	247.18
17	106.983	2.93	5.346	14.177	286.395
18	121.724	3.307	6.114	15.372	334.721
19	128.098	3.508	6.575	16.896	369.898
20	137.588	3.848	7.32	18.371	416.211
	Naïve Shortcut Method, threshold 10000				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	1.627	0.478	0.055	0.067	2.647
2	6.265	1.266	0.116	0.16	9.46
3	9.843	2.478	0.258	0.354	16.608
4	13.914	4.545	0.462	0.625	26.256
5	22.22	6.904	0.71	1.001	41.216
6	32.687	10.184	1.019	1.496	60.865
7	43.45	13.316	1.391	2.251	81.523
8	59.153	17.817	1.87	2.935	110.125
9	71.964	21.867	2.285	3.713	136.135
10	88.054	27.138	2.835	4.579	168.23
11	106.407	32.351	3.422	5.534	204.394
12	128.119	38.992	4.096	6.55	248.274
13	147.685	44.96	4.842	7.597	291.361
14	170.629	52.434	5.547	8.98	341.449
15	210.073	63.293	6.639	10.909	421.348
16	251.606	74.986	7.588	12.413	507.367
17	277.616	83.213	8.61	13.853	569.387
18	312.333	93.965	9.712	15.604	649.434
19	338.734	104.157	10.72	16.639	722.738
20	362.583	114.277	11.754	19.118	799.108

Table 4: Testing the Trajectories Scalability of the Naïve Shortcut Method

	Naïve Shortcut Method Preprocessed, threshold 1				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	0.935	0.23	0.018	0.068	1.606
2	3.559	0.195	0.017	0.164	5.265
3	4.168	0.362	0.041	0.351	7.976
4	6.431	0.647	0.061	0.622	13.254
5	9.802	1.005	0.101	0.971	20.558
6	14.451	1.431	0.132	1.463	30.357
7	19.693	2.023	0.186	2.248	42.194
8	25.073	2.601	0.232	3.024	55.334
9	31.43	3.298	0.299	3.862	70.681
10	37.949	3.955	0.368	4.781	87.006
11	46.005	4.73	0.448	5.555	106.858
12	55.358	5.833	0.546	6.769	131.509
13	66.711	6.737	0.633	8.064	160.24
14	78.652	7.971	0.752	9.375	192.234
15	87.894	9.02	0.86	10.602	222.528
16	98.102	10.233	0.977	11.992	256.53
17	108.824	11.655	1.11	13.993	294.764
18	122.154	13.348	1.268	15.828	342.359
19	127.14	14.255	1.365	16.657	375.622
20	140.101	16.16	1.542	19.24	429.354
	Naïve Shortcut Method Preprocessed, threshold 10000				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	1.653	0.588	0.016	0.068	2.959
2	6.237	1.53	0.011	0.187	9.606
3	10.411	3.148	0.026	0.366	17.864
4	14.28	5.466	0.03	0.629	27.264
5	23.169	8.913	0.052	0.968	44.303
6	33.46	12.783	0.062	1.459	64.123
7	43.294	16.207	0.08	2.289	83.288
8	57.291	21.147	0.108	2.871	109.678
9	71.84	26.768	0.141	3.714	138.887
10	97.241	34.941	0.188	4.767	184.912
11	109.146	40.095	0.223	5.728	213.913
12	125.318	46.412	0.232	6.674	246.313
13	156.416	57.585	0.316	7.716	312.393
14	176.577	65.093	0.353	9.331	358.333
15	208.542	76.435	0.397	10.978	426.629
16	231.867	84.812	0.428	11.944	481.02
17	253.083	94.21	0.496	14.525	537.571
18	281.283	105.83	0.553	15.053	608.177
19	312.351	117.605	0.593	16.855	686.323
20	339.588	130.244	0.664	18.484	764.007

Table 5: Testing the Trajectories Scalability of the Naïve Shortcut Method Preprocessed

	Free Space Grid Method, threshold 1				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	0.902	0.025	0.03	0.067	1.368
2	3.529	0.075	0.056	0.15	5.128
3	4.127	0.108	0.116	0.362	7.72
4	6.419	0.169	0.193	0.627	12.792
5	9.788	0.258	0.306	0.968	19.797
6	14.604	0.409	0.441	1.416	29.559
7	19.733	0.539	0.597	2.252	41.042
8	25.385	0.743	0.774	2.866	53.751
9	32.757	0.875	1.005	3.722	70.006
10	38.209	1	1.228	4.598	84.694
11	46.574	1.146	1.482	5.575	104.701
12	54.82	1.346	1.788	6.664	126.524
13	65.309	1.592	2.08	7.779	153.155
14	75.21	1.76	2.417	9.045	180.837
15	91.4	2.136	2.949	10.59	223.709
16	98.18	2.298	3.236	12.459	251.096
17	116.4	2.969	3.805	14.277	304.959
18	123.045	3.158	4.196	15.016	333.715
19	127.647	3.185	4.526	17.41	368.157
20	148.4	3.786	5.471	19.087	438.381
	Free Space Grid Method, threshold 10000				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	1.605	0.173	0.049	0.07	2.326
2	6.261	0.382	0.158	0.159	8.616
3	9.87	0.765	0.363	0.386	15.12
4	14.136	1.42	0.641	0.625	23.594
5	22.353	2.376	1.007	0.974	37.4
6	32.2	3.415	1.447	1.451	54.161
7	44.898	4.528	2.01	2.244	75.46
8	57.707	5.725	2.587	2.93	97.759
9	74.925	7.494	3.308	3.726	127.162
10	88.493	8.988	4.04	4.585	152.76
11	106.318	10.529	4.925	5.55	185.631
12	125.975	12.25	5.866	6.604	222.246
13	148.572	14.143	6.928	7.808	264.823
14	173.956	16.298	8.019	9.097	313.071
15	199.915	18.354	9.242	10.302	365.006
16	224.906	20.868	10.51	12.06	418.309
17	252.353	23.899	11.791	12.96	477.31
18	280.414	26.333	13.199	14.834	540.886
19	303.911	29.399	14.675	16.634	602.574
20	341.634	32.922	16.405	18.488	687.186

Table 6: Testing the Trajectories Scalability of the Free Space Grid Method

	Free Space Grid Method Optimized, threshold 1				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	0.984	0.026	0.031	0.068	1.334
2	3.588	0.12	0.054	0.154	5.252
3	4.057	0.094	0.118	0.363	7.615
4	6.2	0.164	0.197	0.64	12.544
5	9.865	0.272	0.309	0.983	19.929
6	15.312	0.461	0.484	1.514	31.217
7	19.591	0.566	0.614	2.372	41.086
8	26.159	0.774	0.85	2.87	55.038
9	31.393	0.854	1.036	3.732	68.306
10	39.695	1.052	1.278	4.676	87.116
11	49.94	1.252	1.645	6.642	111.671
12	63.045	1.565	2.005	7.127	142.187
13	72.545	1.765	0.633	8.458	166.704
14	86.739	2.105	2.888	9.24	202.044
15	98.841	2.359	3.317	10.546	237.219
16	112.289	2.595	3.833	12.496	276.934
17	123.18	3.028	4.038	13.774	314.953
18	138.414	3.433	4.578	15.389	365.141
19	147.784	3.629	5.097	18.016	410.169
20	148.273	3.758	5.332	19.227	435.773
	Free Space Grid Method Optimized, threshold 10000				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	1.686	0.099	0.035	0.078	2.333
2	6.128	0.239	0.049	0.174	8.208
3	9.886	0.425	0.11	0.375	14.473
4	14.077	0.831	0.191	0.625	22.436
5	21.958	1.255	0.293	0.968	35.037
6	31.935	1.824	0.421	1.448	50.928
7	43.106	2.496	0.576	2.192	69.347
8	58.677	3.197	0.757	2.95	94.251
9	73.261	4.216	0.967	3.694	119.125
10	88.058	4.83	1.177	4.57	144.447
11	105.441	5.821	1.419	5.526	175.373
12	128.059	7.041	1.717	6.994	215.571
13	155.646	9.032	2.062	8.188	264.911
14	179.055	10.024	2.392	9.336	308.892
15	209.819	11.705	2.781	10.593	366.969
16	220.621	11.977	3.039	12.13	395.463
17	252.291	13.513	3.436	13.894	459.708
18	311.391	18.097	4.329	16.019	567.916
19	303.514	16.46	4.266	17.133	576.017
20	334.785	17.96	4.752	18.846	647.885

Table 7: Testing the Trajectories Scalability of the Free Space Grid Method Optimized

	Free Space Grid Long Jump Method, threshold 1				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	0.893	0.026	0.012	0.071	1.338
2	3.514	0.106	0.028	0.159	5.116
3	4.167	0.103	0.038	0.367	7.749
4	6.183	0.16	0.058	0.617	12.409
5	9.957	0.27	0.082	0.969	19.884
6	14.623	0.4	0.125	1.457	29.474
7	19.685	0.553	0.171	2.316	40.757
8	25.473	0.738	0.232	2.942	53.749
9	36.042	1.048	0.341	3.894	75.238
10	42.547	1.144	0.375	4.619	91.286
11	52.053	1.348	0.466	5.847	114.215
12	61.072	1.551	0.54	6.748	136.774
13	70.812	1.775	0.617	9.163	164.097
14	85.068	2.094	0.742	9.168	197.235
15	96.287	2.31	0.852	11.033	230.533
16	109.523	2.614	0.954	12.153	267.714
17	107.894	2.73	1.029	13.3	282.873
18	114.812	3.012	4.223	15.249	319.285
19	127.589	3.279	1.257	16.545	362.709
20	136.894	3.481	1.405	19.052	408.983
	Free Space Grid Long Jump Method, threshold 10000				
Trajectories	Reachability Time (s)	FSG Method Time (s)	Querying Time (s)	Greedy Set Cover Time (s)	Total Running Time (s)
1	1.564	0.092	0.005	0.075	2.144
2	6.2	0.231	0.014	0.157	8.228
3	10.469	0.567	0.026	0.383	15.499
4	15.08	0.888	0.032	0.645	23.649
5	24.171	1.512	0.056	1.078	37.972
6	33.854	1.93	0.077	1.512	53.083
7	45.982	2.646	0.101	2.31	73.295
8	60.832	3.412	0.133	2.946	97.119
9	81.543	4.563	0.189	3.762	129.298
10	91.432	5.027	0.197	4.59	148.42
11	112.833	6.153	0.251	5.655	184.547
12	137.785	7.961	0.291	6.729	227.527
13	148.262	7.972	0.339	7.862	251.468
14	193.925	10.799	0.43	9.484	326.818
15	199.071	10.632	0.456	10.318	346.541
16	224.73	11.762	0.504	11.699	396.769
17	247.303	13.255	0.576	14.14	448.58
18	276.137	14.671	0.631	15.062	508.734
19	309.903	16.829	0.721	16.795	583.166
20	338.805	18.266	0.797	19.348	653.19

Table 8: Testing the Trajectories Scalability of the Free Space Grid Long Jump

B.1.2 Graphs

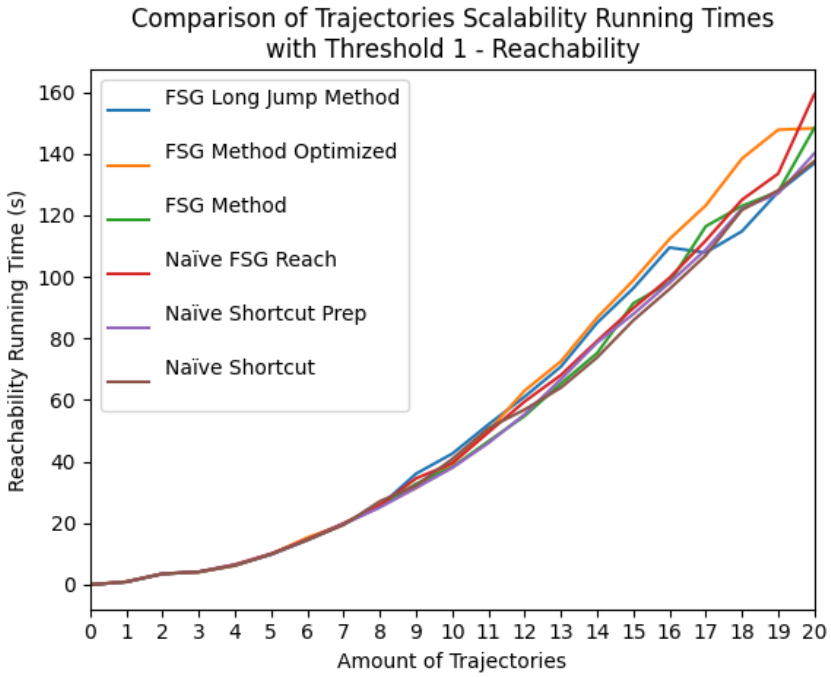


Figure 17: The Reachability Running Times for Trajectories Scalability with threshold 1 of all Methods visualised.

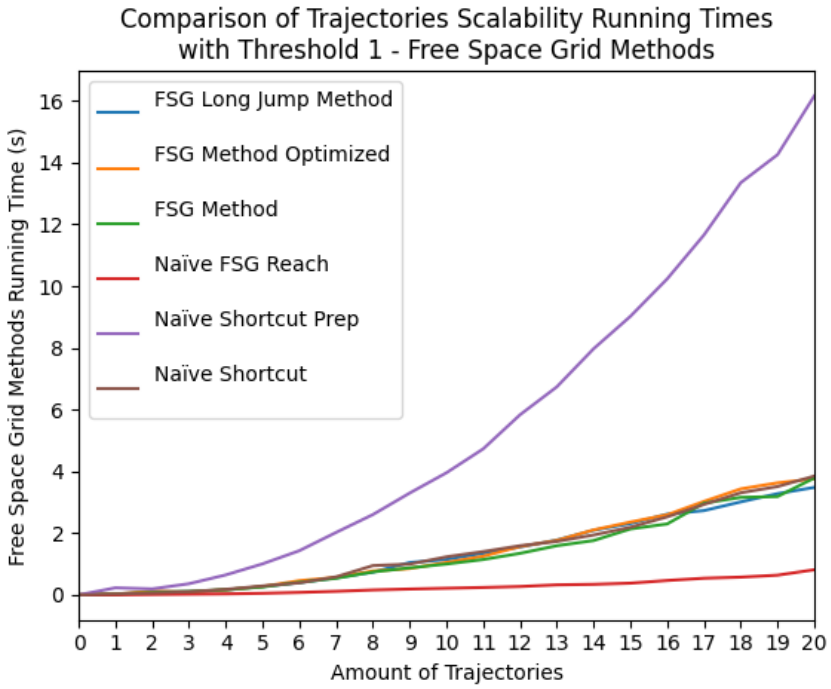


Figure 18: The Free Space Grid Method Running Times for Trajectories Scalability with threshold 1 of all Methods visualised.

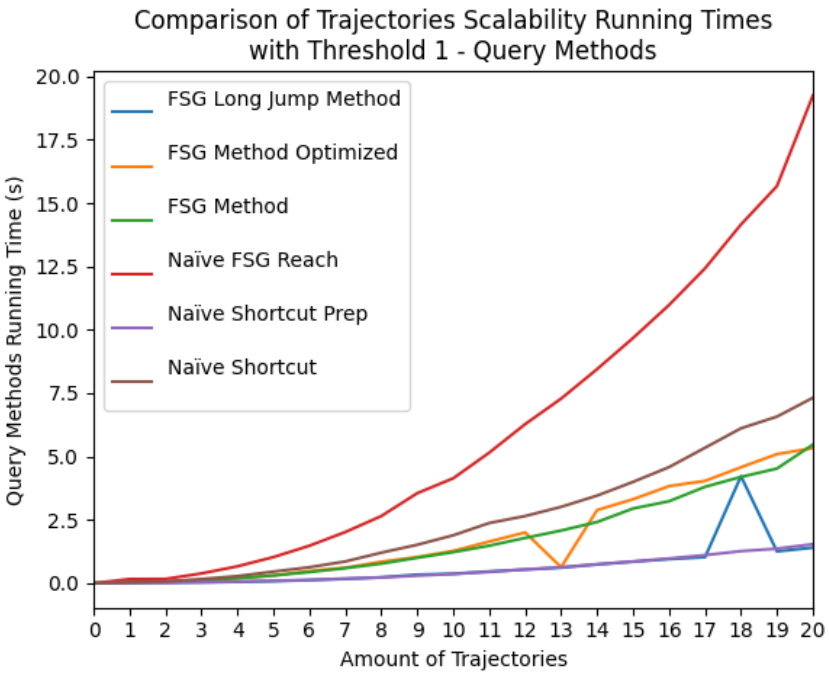


Figure 19: The Query Methods Running Times for Trajectories Scalability with threshold 1 of all Methods visualised.

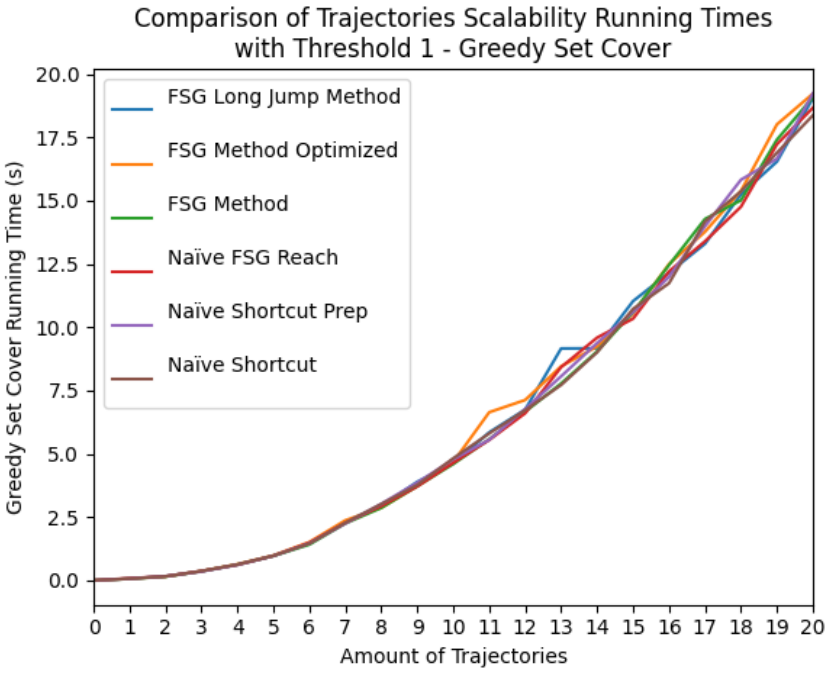


Figure 20: The Greedy Set Cover Running Times for Trajectories Scalability with threshold 1 of all Methods visualised.

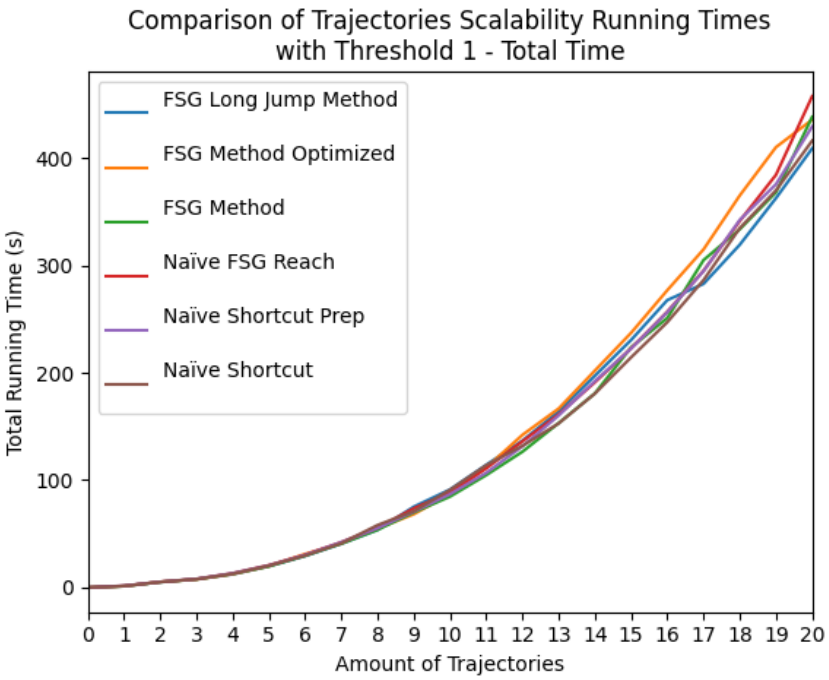


Figure 21: The Total Running Times for Trajectories Scalability with threshold 1 of all Methods visualised.

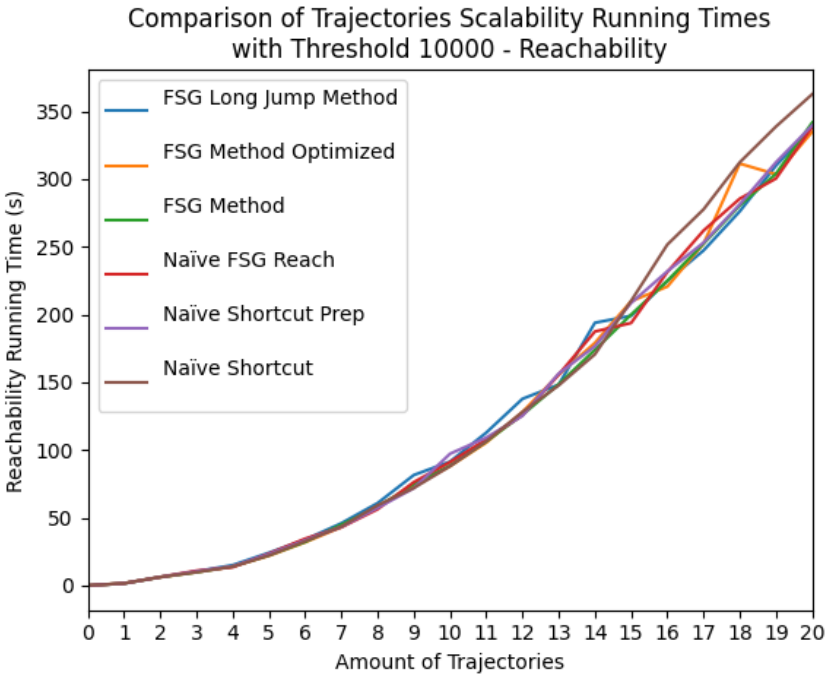


Figure 22: The Reachability Running Times for Trajectories Scalability with threshold 10000 of all Methods visualised.

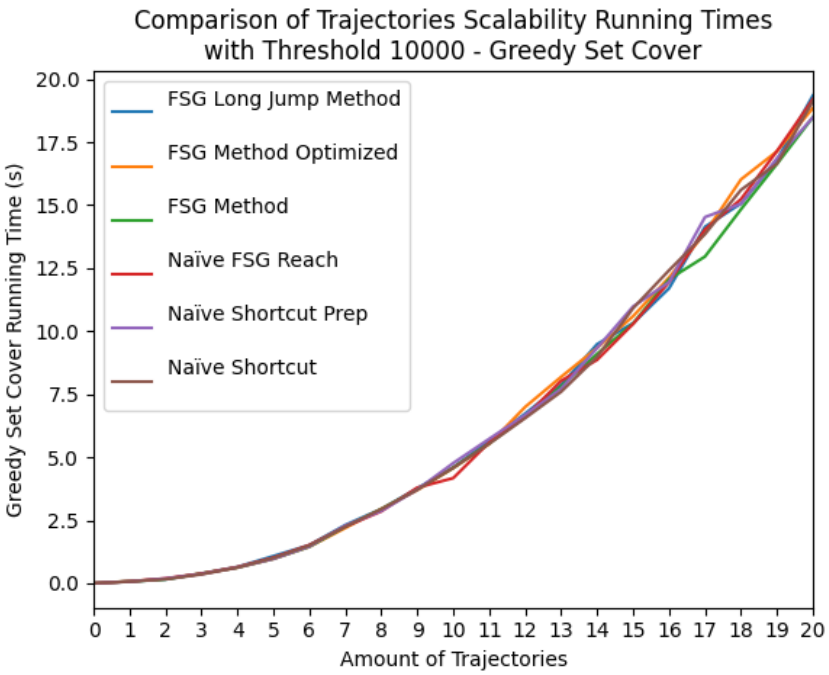


Figure 23: The Free Space Grid Method Running Times for Trajectories Scalability with threshold 10000 of all Methods visualised.

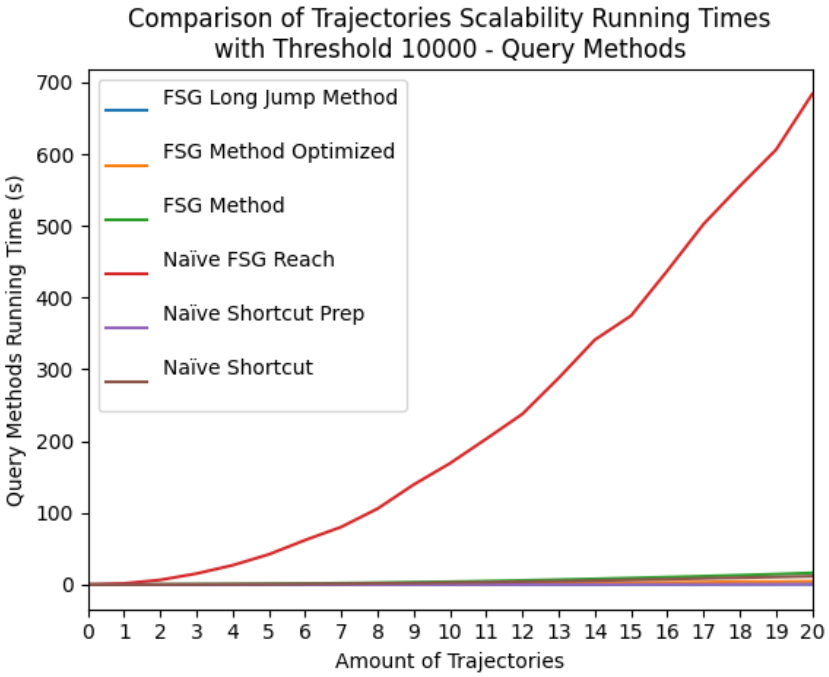


Figure 24: The Query Methods Running Times for Trajectories Scalability with threshold 10000 of all Methods visualised.

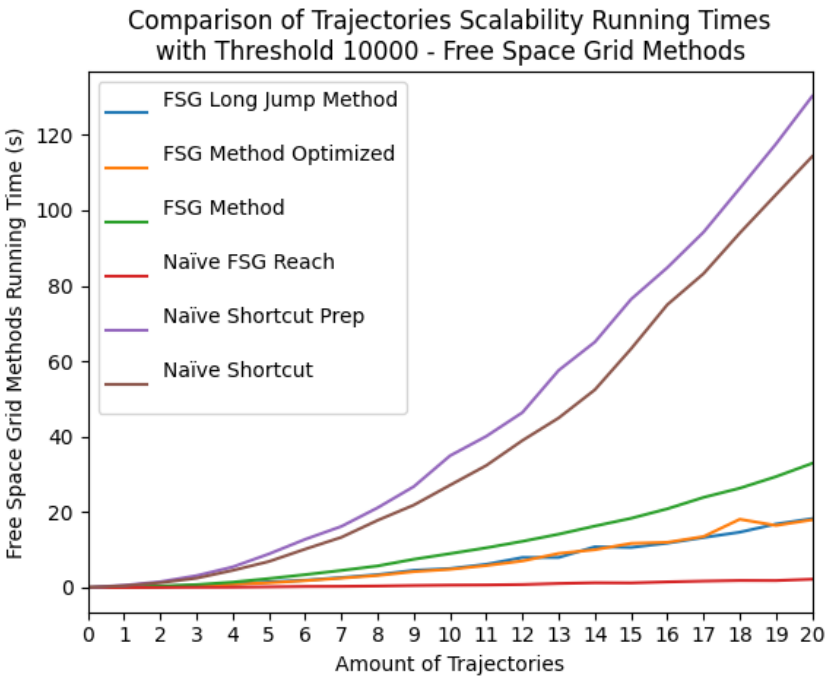


Figure 25: The Greedy Set Cover Running Times for Trajectories Scalability with threshold 10000 of all Methods visualised.

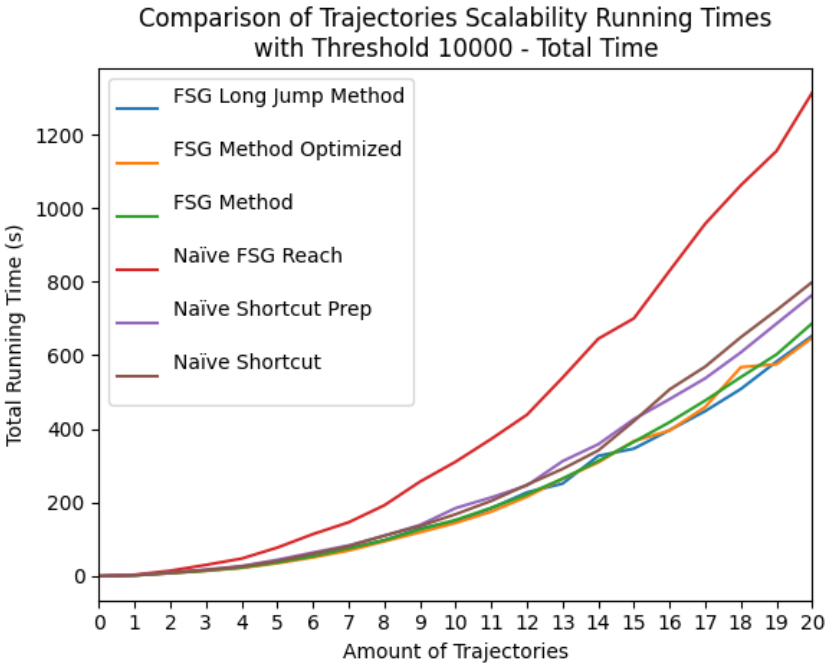


Figure 26: The Total Running Times for Trajectories Scalability with threshold 10000 of all Methods visualised.

B.2 Vertices

Note that the trajectories used are all the trajectory from $(0, 0)$ to $(0, 10000)$, sampled with an increasing amount of vertices. Every execution of Greedy Set Cover was on only one trajectory.

B.2.1 Tables

	Naïve Reachability FSG Method, threshold 1							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.106	30	0.001	20	0.031	21	0.014	0.286
100	0.951	337	0.004	124	0.043	140	0.049	1.424
150	3.458	932	0.005	579	0.196	597	0.198	5.079
200	11.715	1803	0.582	1721	0.582	1765	0.708	15.699
250	Heap Space Error							
	Naïve Reachability FSG Method, threshold 10000							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.148	30	0.003	24	0.073	26	0.012	0.413
100	1.611	335	0.01	150	1.636	151	0.048	3.762
150	8.146	953	0.015	615	11.546	621	0.208	21.215
200	51.935	1829	0.033	1785	47.482	1798	0.655	102.667
250	Heap Space Error							

Table 9: Testing the Vertices Scalability of the Naïve Reachability FSG Method

	Naïve Shortcut Method, threshold 1							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.136	27	0.009	21	0.012	22	0.02	0.323
100	0.907	380	0.031	142	0.016	147	0.062	1.332
150	1.706	1063	0.025	606	0.071	624	0.191	3.174
200	11.081	1865	0.048	1786	0.208	1828	0.691	14.607
250	Heap Space Error							
	Naïve Shortcut Method, threshold 10000							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.147	32	0.053	26	0.01	27	0.014	0.394
100	1.68	268	0.372	165	0.026	159	0.088	2.614
150	7.934	1002	1.034	656	0.12	647	0.189	10.585
200	40.951	1894	3.033	1848	0.358	1860	0.639	47.805
250	Heap Space Error							

Table 10: Testing the Vertices Scalability of the Naïve Shortcut Method

	Naïve Shortcut Method Preprocessed, threshold 1							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.097	31	0.025	21	0.004	22	0.012	0.276
100	0.867	303	0.071	142	0.004	147	0.048	1.348
150	1.807	952	0.274	607	0.014	624	0.188	3.468
200	11.26	1840	0.401	1785	0.038	1828	0.665	14.849
250	Heap Space Error							
	Naïve Shortcut Method Preprocessed, threshold 10000							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.144	31	0.067	26	0.005	27	0.015	0.399
100	1.597	335	0.548	164	0.003	159	0.05	2.666
150	8.29	850	1.264	656	0.008	647	0.189	11.018
200	39.008	1889	3.94	1894	0.015	1860	0.643	46.188
250	Heap Space Error							

Table 11: Testing the Vertices Scalability of the Naïve Shortcut Method Preprocessed

	Free Space Grid Method, threshold 1							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.12	26	0.008	22	0.008	21	0.014	0.295
100	0.901	260	0.024	143	0.016	142	0.051	1.36
150	1.661	980	0.065	609	0.045	605	0.195	3.159
200	11.15	1772	0.052	1789	0.122	1782	0.666	14.482
250	Heap Space Error							
	Free Space Grid Method, threshold 10000							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.152	29	0.035	29	0.013	28	0.015	0.388
100	1.697	320	0.146	183	0.041	171	0.048	2.401
150	7.809	936	0.267	720	0.147	686	0.189	9.738
200	41.934	1841	0.954	1919	0.345	1861	0.63	46.645
250	Heap Space Error							

Table 12: Testing the Vertices Scalability of the Free Space Grid Method

	Free Space Grid Method Optimized, threshold 1							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.099	31	0.008	22	0.007	22	0.013	0.267
100	0.959	238	0.028	143	0.014	142	0.052	1.445
150	1.715	1057	0.058	609	0.046	605	0.259	3.276
200	11.363	1847	0.049	1789	0.116	1782	0.671	14.632
250	Heap Space Error							
	Free Space Grid Method Optimized, threshold 10000							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.154	28	0.022	26	0.007	26	0.016	0.362
100	1.625	335	0.095	159	0.012	153	0.049	2.244
150	8.053	923	0.213	645	0.039	628	0.192	9.782
200	38.976	1851	0.437	1853	0.093	1815	0.649	42.731
250	Heap Space Error							

Table 13: Testing the Vertices Scalability of the Free Space Grid Method Optimized

	Free Space Grid Long Jump Method, threshold 1							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.099	29	0.008	22	0.003	21	0.015	0.264
100	0.896	250	0.025	143	0.017	138	0.064	1.379
150	1.714	976	0.057	609	0.013	592	0.193	3.131
200	11.153	1815	0.049	1789	0.027	1751	0.689	14.37
250	Heap Space Error							
	Free Space Grid Long Jump Method, threshold 10000							
Vertices	Reachability Time (s)	Reachability Space (MB)	FSG Method Time (s)	FSG Method Space (MB)	Querying Time (s)	Querying Space (MB)	GSC Time (s)	Total Running Time (s)
50	0.145	32	0.022	26	0.002	25	0.01	0.347
100	1.63	288	0.107	159	0.003	153	0.049	2.21
150	8.288	941	0.17	644	0.01	629	0.226	9.953
200	38.518	1800	0.38	1853	0.013	1815	0.645	42.191
250	Heap Space Error							

Table 14: Testing the Vertices Scalability of the Free Space Grid Long Jump Method

B.2.2 Graphs

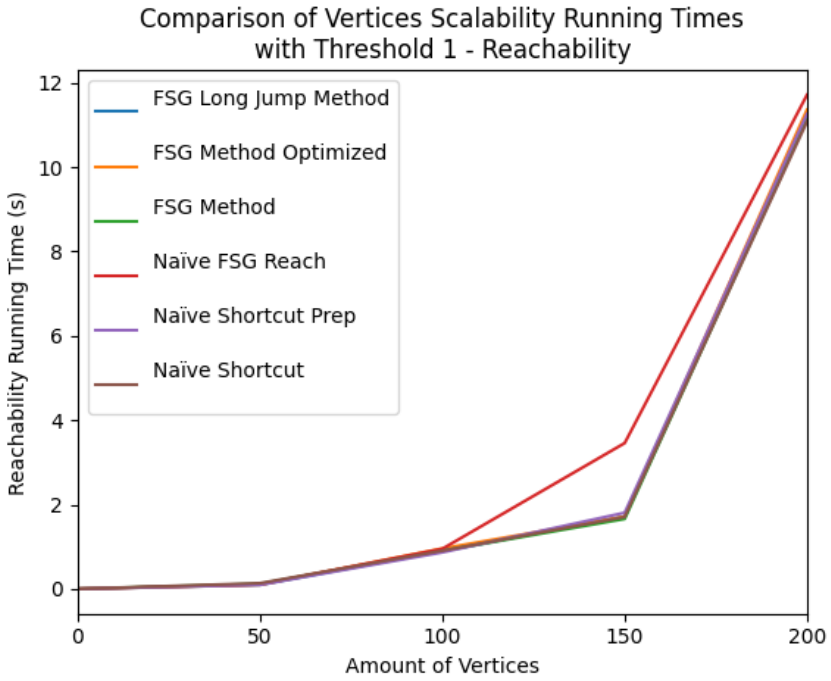


Figure 27: The Reachability Running Times for Vertices Scalability with threshold 1 of all Methods visualised.

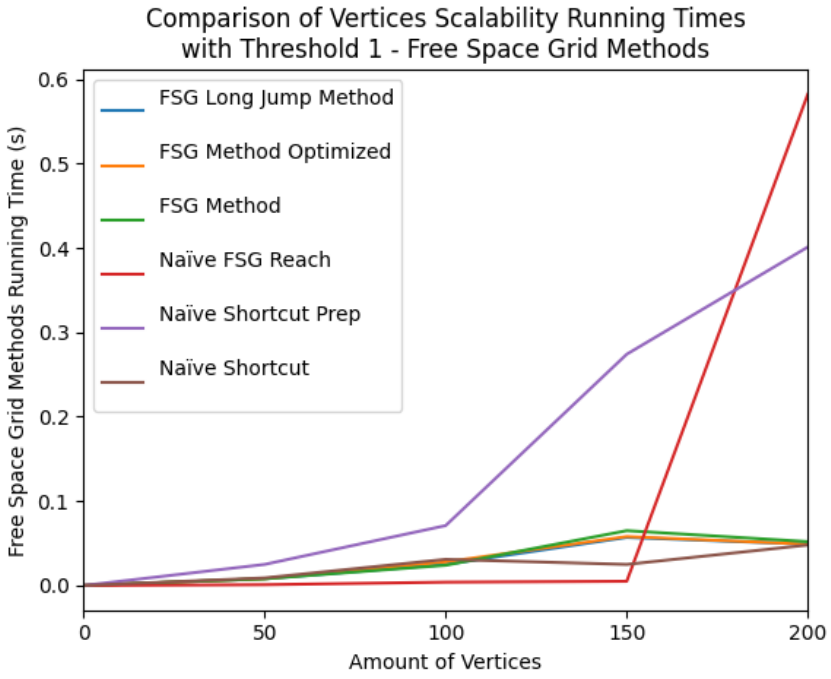


Figure 28: The Free Space Grid Method Running Times for Vertices Scalability with threshold 1 of all Methods visualised.

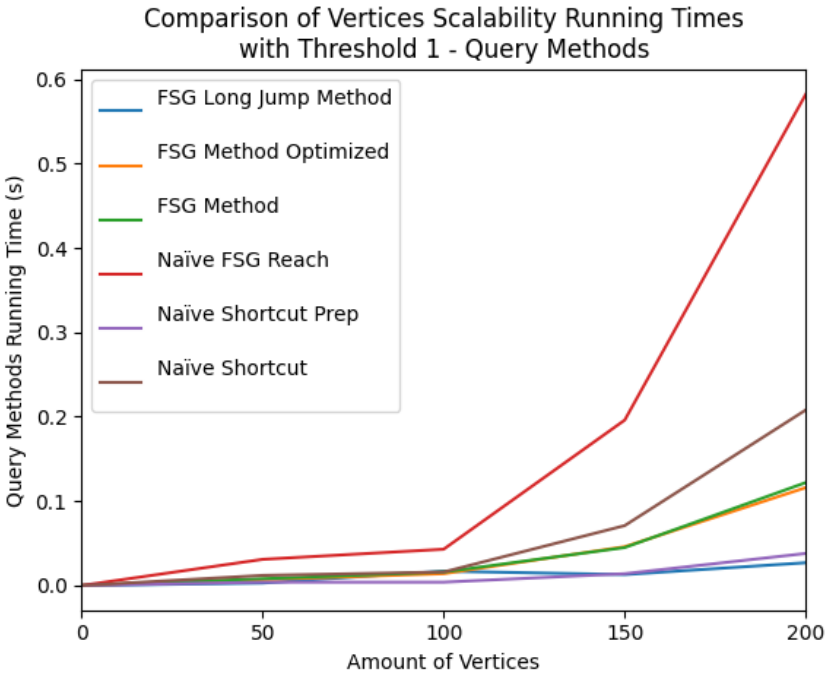


Figure 29: The Query Methods Running Times for Vertices Scalability with threshold 1 of all Methods visualised.

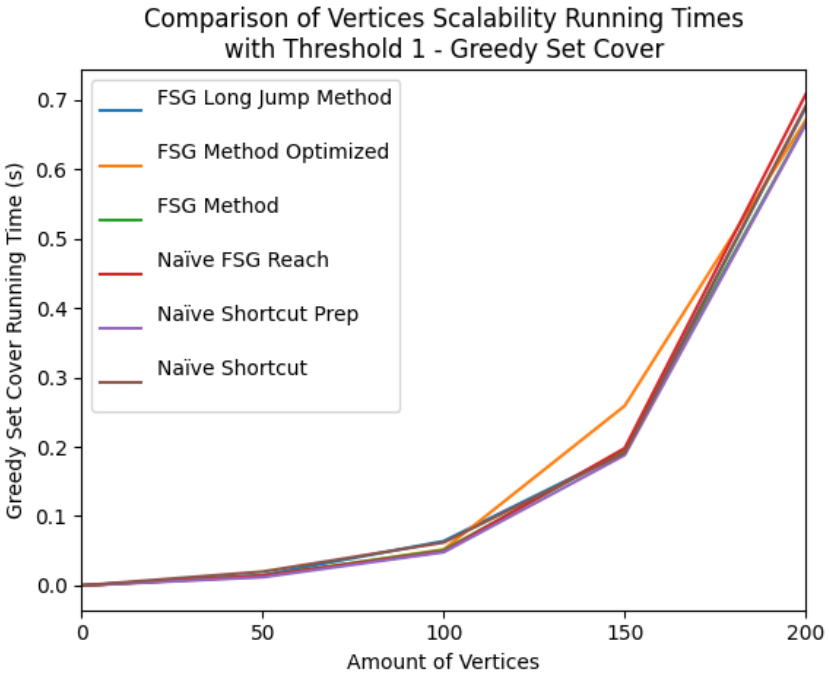


Figure 30: The Greedy Set Cover Running Times for Vertices Scalability with threshold 1 of all Methods visualised.

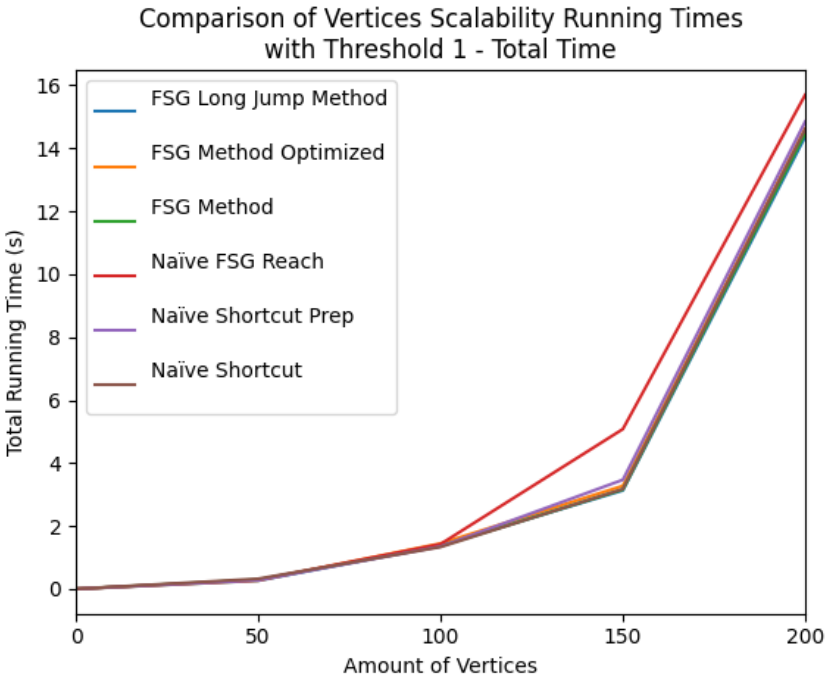


Figure 31: The Total Running Times for Vertices Scalability with threshold 1 of all Methods visualised.

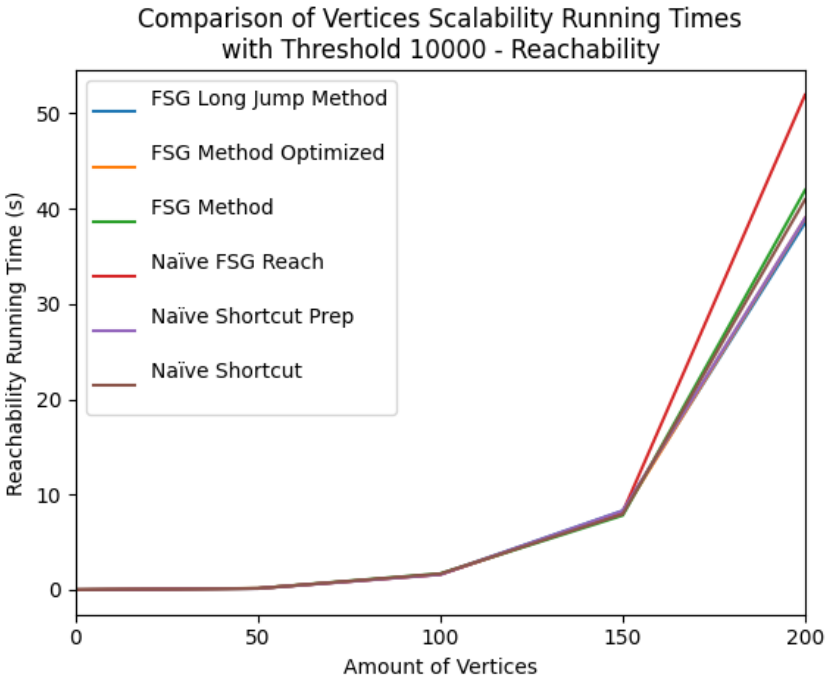


Figure 32: The Reachability Running Times for Vertices Scalability with threshold 10000 of all Methods visualised.

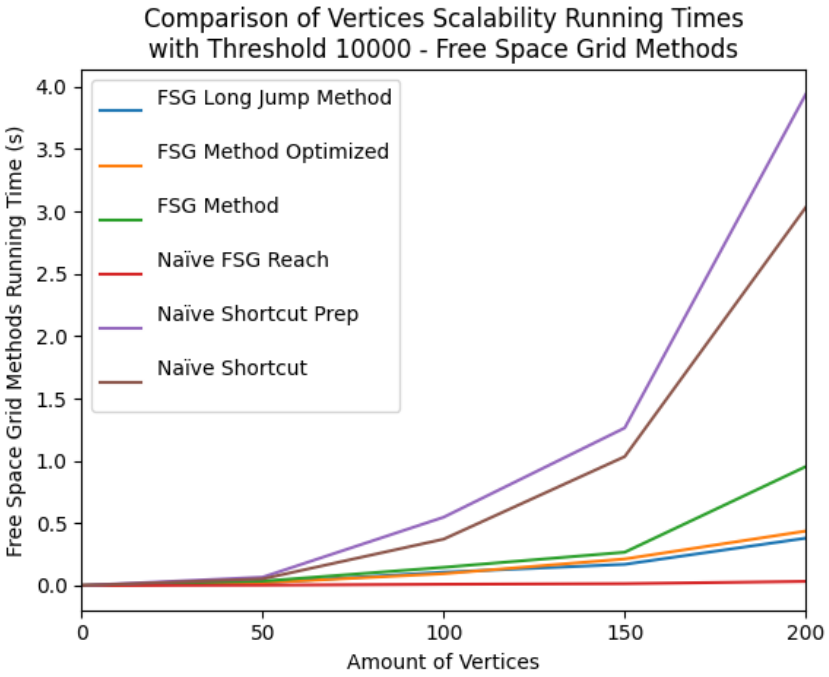


Figure 33: The Free Space Grid Method Running Times for Vertices Scalability with threshold 10000 of all Methods visualised.

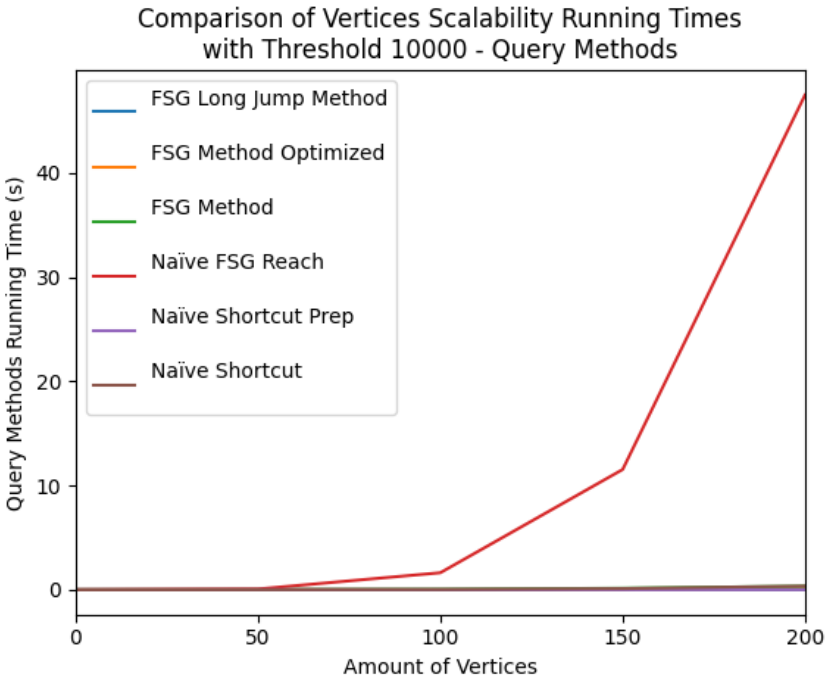


Figure 34: The Query Methods Running Times for Vertices Scalability with threshold 10000 of all Methods visualised.

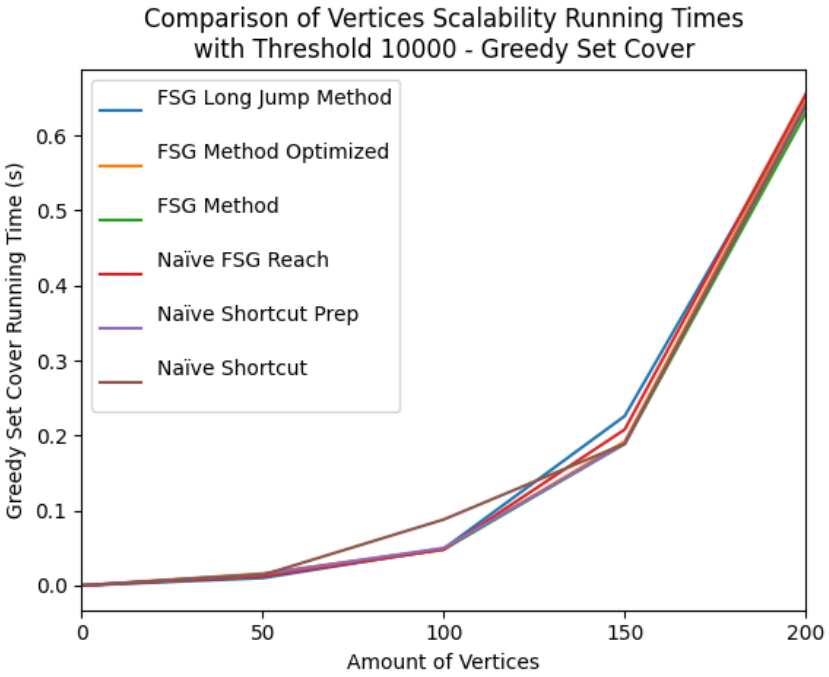


Figure 35: The Greedy Set Cover Running Times for Vertices Scalability with threshold 10000 of all Methods visualised.

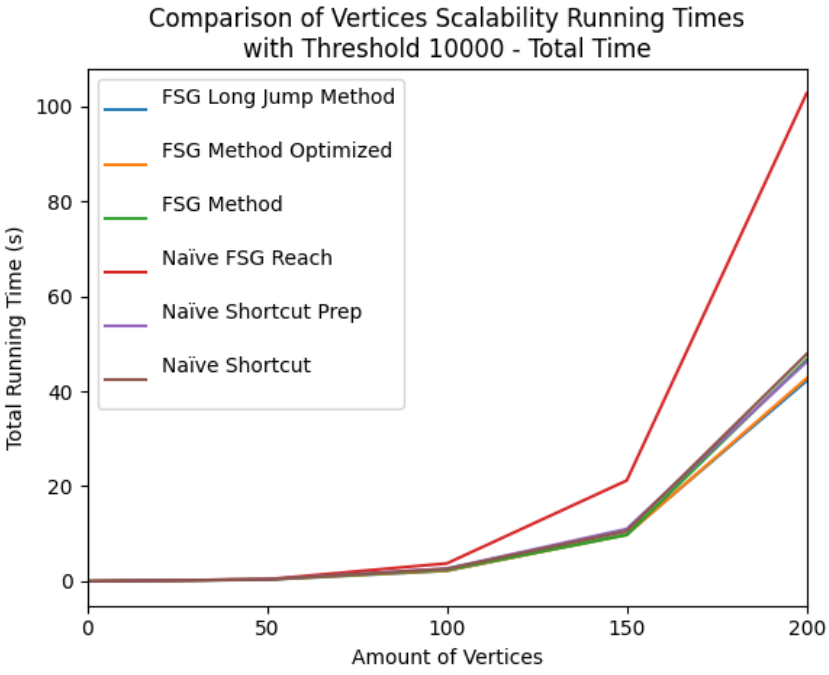


Figure 36: The Total Running Times for Vertices Scalability with threshold 10000 of all Methods visualised.