UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BarcelonaTech
FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

On the design of an implementation of kinetic minimum spanning trees

THESIS FOR THE ACHIEVEMENT OF A BACHELOR'S DEGREE IN COMPUTER ENGINEERING

Director:	Amalia Duch Brown
Department:	Computer Science
Specialisation:	Computing
Date:	January 2020

Erik-Augund Kvam Garí

Abstracts

Resum

En aquest projecte dissenyem una possible implementació dels arbres cinètics d'expansió mínims proposats teòricament per Agarwal et al. a 'Parametric and kinetic minimum spanning trees' [1]. El problema que es vol resoldre amb aquesta proposta és el de mantenir un arbre d'expansió mínim d'un graf amb pesos que canvia al llarg del temps. Els possibles canvis del graf són causats per canvis als pesos de les arestes –els quals són funció d'un paràmetre t que representa el temps—- a més de permetre addicions i supressions d'arestes i actualitzacions de la funció de càlcul del pes de les arestes.

Resumen

En este proyecto diseñamos una posible implementación de los árboles cinéticos de expansión mínimos propuestos teóricamente por Agarwal y col. en 'Parametric and kinetic minimum spanning trees' [1]. El problema que se pretende resolver con esta propuesta es de mantener un árbol de expansión mínimo de un grafo con pesos que cambia a lo largo del tiempo. Los posibles cambios del grafo vienen dados por cambios en los pesos de las aristas –que son función de un parámetro t que representa el tiempo—- además de permitir adiciones y supresiones de aristas y actualizaciones de la función de cálculo del peso de las aristas.

Abstract

In this project, we design a possible implementation of kinetic minimum spanning trees proposed theoretically by Agarwal et al. in 'Parametric and kinetic minimum spanning trees' [1]. The problem we try to solve with this proposal is to maintain a minimum spanning tree of an edge-weighted graph that changes through time. The possible changes in the graph come from changes in the edge weights –which are functions of a parameter t which represents time– in addition to permitting additions and deletions of vertices and edges and updates to the edge weight calculation function.

Contents

Chapter 1

Introduction

Computers, from their inception, have been conceptualised as discrete mathematical objects. Discrete mathematical objects, like the integer arrays that are the foundation of modern computation, can assume only distinct, separated values [17].

This means that computers dealing with the continuity of the real world need specialised processes. Regarding time, the naïve approach is to discretize the continuity. This is done by advancing time in discrete time steps and deleting and reinserting each item at their new position. But how do you choose which time step difference to advance to? On the one hand, if it's too small, we might waste a lot of resources calculating small, insignificant changes. On the other, if it's too big, we may miss discrete events, like two points moving towards each other, which collide at exactly one point in time. Even if we found the ideal time step for each concrete case, in many fields events tend to occur in irregular patterns, which means that no value will avoid both problems simultaneously for all time.

To deal with these kinds of problems, kinetic data structures (KDS) were introduced in 1997 by Basch, Guibas and Hershberger [2]. Theoretical work on them has been done since their inception, but not many library implementations have been created. The main resource in this field is an area of the Computational Geometry Algorithms Library (CGAL), a software project that aims to provide easy access to efficient and reliable geometric algorithms in the form of a C++ library [21]. The area, developed by Russel [18], provides a general framework, but using it requires knowledge of KDS, apart from the problem that one is trying to solve itself. To avoid reinventing the wheel, the owners have made available implementations for a sorted list, a 2D Delaunay triangulation and 3D Delaunay and regular triangulations.

Another use for KDS that has been theoretically put forward is for maintaining a minimum spanning tree (MST) of a graph with continuously changing edge weights [3]. An MST is a subgraph of a weighted graph that contains just the edges necessary to keep the graph connected while minimising the total weight. There are quite a few use cases for minimum spanning trees. One example, explained in [6], would be a telecommunications company trying to lay cable in a new neighbourhood. If it is constrained to bury the cable only along certain paths (e.g. roads), then there would be a graph containing the points (e.g. houses) connected by those paths. Some of the paths might be more

expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. Currency is an acceptable unit for edge weight – there is no requirement for edge lengths to obey normal rules of geometry such as the triangle inequality. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects every house; there might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost, representing the least expensive path for laying the cable. This can also be used as a stepping stone to other problems that can be reduced to it [3].

The main goal of this project is to provide an implementation of a kinetic data structure that stores the minimum spanning tree of a graph, where the edge weights are changing along time. To do this thoroughly, we intended to:

1. Analyse, experimentally and theoretically, the algorithms proposed in the literature to choose the one which performs best,
2. Implement the chosen algorithm using the framework provided by the CGAL KDS [18] library,
3. Study the efficiency of the final implementation experimentally and finally (if possible)
4. Submit the implementation to the CGAL editorial board.

Unfortunately, the project could not be completed as initially planned. Once the planning phase was done, we started researching the details of the CGAL, especially the submission process. It became fast apparent that the section on KDS that we intended to expand had not had much development in some time and had been left only in a semi-usable state, so much so that the latest versions (from 4.12 onward) of the CGAL had dropped it. No submissions were accepted for deprecated sections, only a full re-build, so the initial idea could not be undertaken.

Given this obstacle, we decided to try to design our library, independent from CGAL, which solved the problem we were trying to solve. The intention was to generate a design that could either be adapted to the framework of the KDS section of the CGAL, were it ever to be recuperated, be used by another library interested in the problem or implemented in a standalone fashion.

Agarwal et al. [1] proposed a solution for the problem in general terms, which is the one we will try to design a program for. We chose this paper because of two main reasons. First, it provides one of the best solutions found to date and is cited widely. Second, it is well-known and educational resources exist around it, so someone new to the field will find it easier to understand the solution.

This document is structured as follows. Chapter 1 introduces the general idea for the project. Chapter 2 explains what's required to understand the problem. Chapter 3 details the solution. Chapter 4 is the proposal itself. Chapters 5 and 6 describe the rest of the details of the project development. Chapter 7 presents the extracted conclusions.

Chapter 2

Preliminaries

To aid in understanding the project, we will describe some of the preliminary concepts. The explanations are merely to introduce the concepts to the reader, and are not meant to be formal definitions.

2.1 Graphs

Let V be a set, whose elements we call *vertices*. Let E be a set of subsets of V with two different vertices. We call the members of E edges. The tuple $G = (V, E)$ is a *graph*. For example, we could have $V = \{a, b, c, d, e, f, g\}$ and

$$\begin{aligned} E = \{&\{a, b\}, \{a, d\}, \\ &\{b, c\}, \{b, d\}, \{b, e\}, \\ &\{c, e\}, \\ &\{d, e\}, \{d, f\}, \\ &\{e, f\}, \{e, g\}, \\ &\{f, g\}\}. \end{aligned}$$

This would mean there are seven vertices and eleven edges. A graphical representation can be seen in Figure 2.1.

Graphs come in many varieties. If a graph's edges are pairs, instead of sets, the order of the vertices in a set matters. This can be represented as

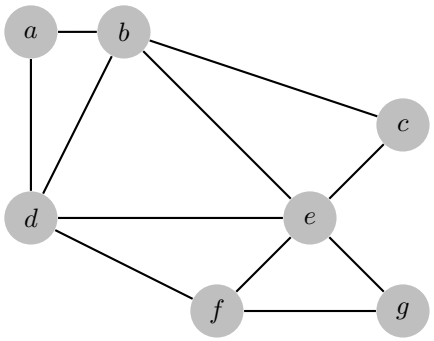

Figure 2.1: A simple graph

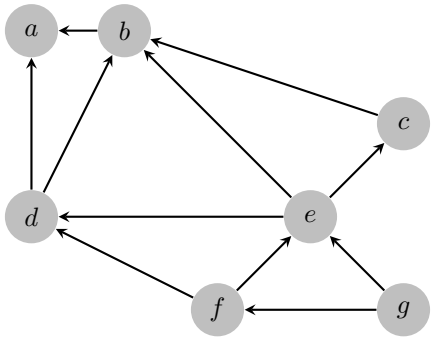

Figure 2.2: A directed graph

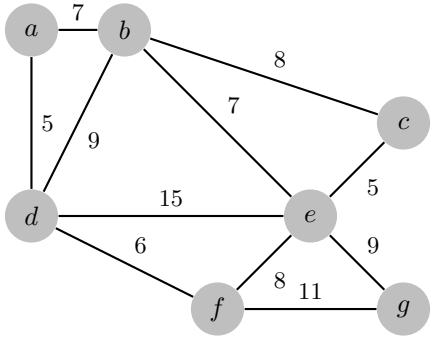

Figure 2.3: A weighted graph

edges having a direction, like the ones in Figure 2.2 and the graph is called a *directed graph* or a *digraph*. To disambiguate, we call a graph that is not directed *undirected.*

Let l be a function that assigns to each vertex or edge an element of a set L called a *label*. We call the triple $G = (V, E, l)$ a *labeled* graph. If l assigns to elements of V, we call the graph *vertex-labeled*. If l assigns to elements of E, we call the graph *edge-labeled*. If $l : E \to \mathbb{L}$ and $\mathbb{L}$ is a set of numbers, such as $\mathbb{R}$ or $\mathbb{Z}$, we call the labels *weights*. An example of a edge-weighted graph is the one in Figure 2.3.

A graph is a *subgraph* of another graph if its vertex and edge sets are subsets of the vertex and edge sets of the original graph. The labels of a labeled subgraph are the labels of the original graph where its vertex or edge are in the subgraph. For example, if $G = (V, E, w), w : E \to \mathbb{L}$ is an undirected, weighted graph, the graph $G' = (V', E', w')$ is a subgraph of G if $V' \subseteq V, E' \subseteq E$ and $w' : E' \to \mathbb{L}, \forall e \in E', w'(e) = w(e)$. As an example, we could take the vertices a, b, c and d only and some of the edges that join them. This defines the graph in Figure 2.4, which is a subgraph of the weighted graph in Figure 2.3.

We call a sequence of edges joining a sequence of vertices without repeating any edge or vertex a *path*. The marked edges of Figure 2.5 form a path between a and g. The weight of a path is the sum of the weights of the edges in the path. In our example, the weight of the path is 29.

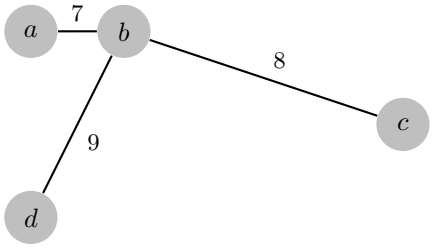

Figure 2.4: A subgraph of the graph in Figure 2.3

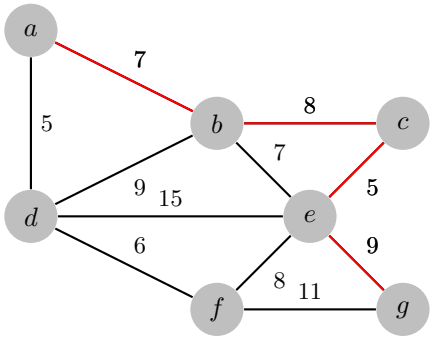

Figure 2.5: A path in a weighted graph

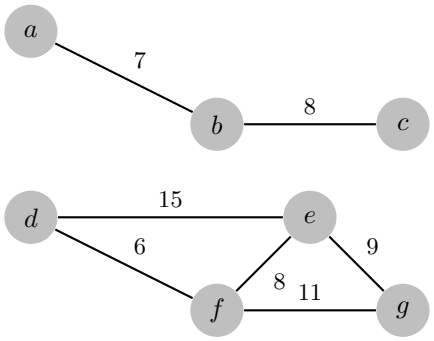

Figure 2.6: An unconnected graph with two connected components

If for every pair of vertices $u, v \in V$ we can find a path between them, we call the graph *connected*. If not, we call it *unconnected*. In an unconnected graph, every subgraph $G_i = (V_i, E_i, w_i)$ that is connected and has no outside connections, that is $\forall v \in V_i, \nexists e \in E, e = \{v, u\}, u \in V, u \notin V_i$, is a connected component of G. The graph in Figure 2.6 is unconnected and has two connected components, one with vertices $\{a, b, c\}$ and the other with vertices $\{d, e, f, g\}$.

2.2 Minimum Spanning Trees

Let $G = (V, E, w)$ be an undirected, weighted graph, where w is the weighing function $E \to \mathbb{R}$.

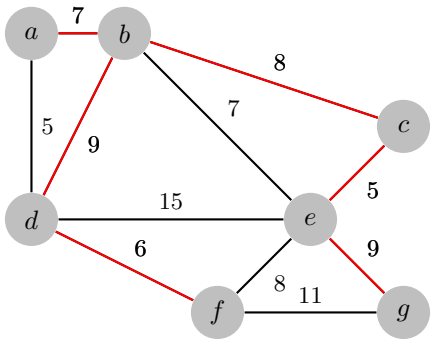

Figure 2.7: A graph with a spanning tree marked in red

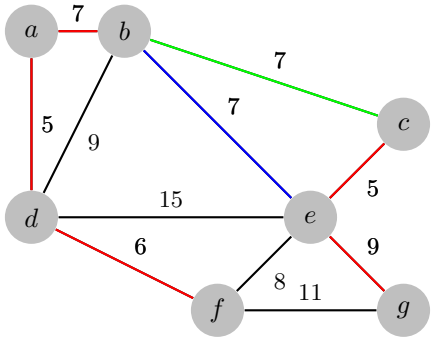

Figure 2.8: A graph with two minimum spanning trees marked, one in red and blue and one in red and green

Let $G' = (V, E', w')$ be a subgraph of G. If $\forall u, v \in V$ there is exactly one path between u and v, we call G' a *spanning tree* of G. The red edges in Figure 2.7 form a spanning tree.

A spanning tree of G where $\sum_{e \in E'} w'(e)$ is minimum is called the *minimum spanning tree* of G. The notation $\mathrm{MST}(G)$ refers to such a graph. There can be several spanning trees that are minimum, depending on the edge weights. For example, in Figure 2.8, the red edges together with either the blue or green edge form a minimum spanning tree of the underlying graph.

When G is unconnected, since there's at least one pair of vertices of V with no path between them, we can't have a spanning tree of G. If we treat every connected component $G_i = (V_i, E_i, w_i)$ of G as its own graph and find each of their respective spanning trees, the subgraph $F = (V, E', w), E' = \bigcup_{\forall i} E_i$ is called a *spanning forest* of G. Analogously to the minimum spanning tree, we can define the *minimum spanning forest* of G as the spanning forest of G with the minimum total weight sum. Figure 2.9 shows an example.

2.3 Kinetic Data Structures

Kinetic data structures were developed by Basch, Guibas and Hershberger [2] to manage data that changes continuously over time. Specifically, data that

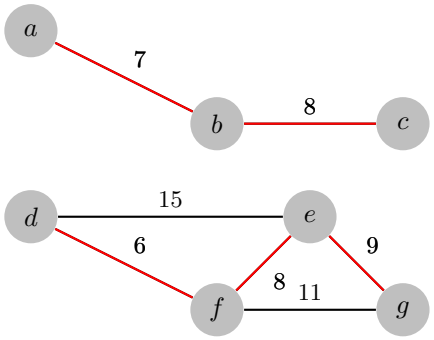

Figure 2.9: An unconnected graph with its minimum spanning forest marked in red

contains a set of mostly independent objects moving through an n-dimensional field.

An example, extracted from Demaine [7], is trying to maintain the previous element of each point in a set, where the points are moving back and forth in a single dimension. It can be visualised as cars on a raceway. We would like to know, for example, the distance between a car and the previous, or who is expected to be in the lead at a certain time in the future.

Each object in a kinetic data structure has a value which is a defined function of time that represents its movement. In our example, that would be the current speeds and positions of the cars.

A KDS must support 3 types of operations:

- `modify(x,f(t))`: The previous position function describing point x is replaced by $f(t)$. This allows us to refit the structure to more up-to-date information, which is essential if the function used to approximate the position of a point is imperfect.
- `advance(t)`: Advances the current time in the data structure to t. The current time must be less than t.
- `query`: Queries the data structure relative to the current time. The specific queries are dependent on the data structure in question.

The approach is to store a set of boolean conditions that uniquely represent the current state of the structure called *certificates*. These allow us to know when the structure we are trying to maintain is no longer valid and needs changes, instead of trying to calculate it at every time or arbitrary times. We call the moments when a certificate fails *events*. For the predecessor example, we could store the points in a binary search tree and a possible set of certificates would be $\forall i, x_i \leq x_{i+1}$, where x_i is the i-th element of the list.

The choice of certificates is the biggest factor in the design of a kinetic data structure. Advancing time consists, basically, of processing events one at a time, as they come. This means a good implementation strikes a balance between the number of events and the number of certificates that have to be checked when an event happens. Our previous example might be too simple to be able to show the difference, but for more complex cases it is trivial to see

that a certificate set can use a larger number of more specific certificates or a smaller number of more general certificates.

Guibas [12] proposes four metrics to for evaluating kinetic data structures. They are usually measured in relation to the number of individual objects in the structure, n.

Responsiveness A KDS is good if the time to process an event is small. The responsiveness of a KDS is the worst-case amount of time needed to update the certificate set after an event. For our example, when certificate $x_i \leq xi + 1$ fails, we will have to swap x_i and x_{i+1} in three certificates: the one that relates x_i to its predecessor, the failing certificate itself and the one that relates x_{i+1} and x_{i+2}. Calculating their failure time is just a matter of finding when the position-velocity equations of each pair cross, so it is $O(1)$. Finally, we have to re-add the certificates to the priority queue, which takes $O(\log n)$ time for each change. The final responsiveness comes out to $O(1) \times O(1) \times O(\log n) = O(\log n)$.

Efficiency As we said before, having certificates that are too specific, depending on the case, might mean we need too many certificates. The efficiency of a KDS is the worst-case number of events processed. In our case, at most each car will pass all other cars at some point; that means we will have $O(n^2)$ events.

Locality The locality of a KDS is the maximum number of certificates in which any one object can ever appear. In our case, each x_i appears in only two certificates at any time: one represents what its predecessor is, and one represents whose predecessor it is.

Compactness The size of a KDS is the maximum number of certificates in the set. We call a KDS compact if its size is of order similar to $O(n)$. In our case, we have $n - 1 = O(n)$ certificates at any point.

2.4 The Kinetic spanning tree problem

The kinetic spanning tree problem is in the middle of two problems: finding the minimum spanning tree of a graph (explained in Section 2.2) and dealing with data that changes continuously over time (explained in Section 2.3).

The problem, then, is to maintain the MST(G) for a graph G that is changing over time.

The definition of *changing over time* can have several interpretations, so depending on whether we use a parametric or kinetic graph, we will have one version or another of the problem.

A parametric graph is a graph $G = (V, E, w_t)$ such that the weighting function w_t changes depending on a time variable t. The vertex and edge sets are constant, but each edge weight is a function of the current time, like the one in Figure 2.10.

Similarly, in a kinetic graph $G = (V, E, l)$, we allow for insertions and deletions of vertices and edges as well as having a weighing function that is a function of a time variable t. We also allow for updates to the labelling function. These changes are assigned a time in which they happen, so we often

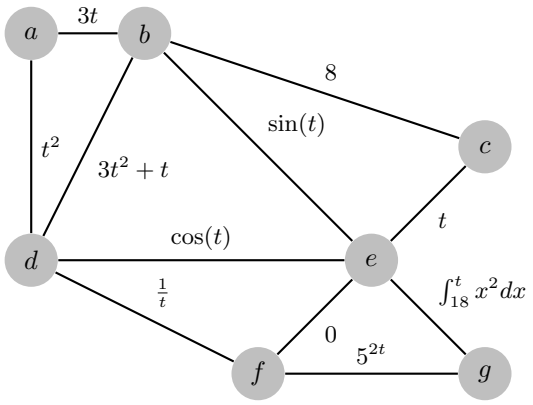

Figure 2.10: A parametric graph

refer to the graph $G_t = (V_t, E_t, l_t)$ as the instance of a kinetic graph for a specific time t.

Insertions and deletions of edges can be simulated using a small construction, so we differentiate *structurally kinetic* graphs, those where the insertions and deletions are stored separately, from *functionally kinetic* graphs, where these changes are simulated in a slightly different graph.

Chapter 3

Theoretical solution by Agarwal et al. [1]

The solution we have chosen comes from a paper from Agarwal et al. [1]. It proves the existence of an algorithm for the kinetic and parametric minimum spanning tree problems in $O(n^{2/3}\log^{4/3} n)$ per change in the MST. Introducing randomisation, the cost is reduced to $O(n^{2/3}\log n)$.

As we explained before, a kinetic data structure is characterised by three things: the type of its independent objects, how it performs each operation and what certificates it uses.

By definition, all vertices should be in the MST. In the calculation of an MST, we are mostly just interested in which edges are in the MST and which aren't, so that is what we will use as our objects. So as not to complicate the calculation of the edge weights at each point in time, the authors choose to limit edges to having an *affine* function as a weight. This means each edge weight function has the form $w(e,t) = a_e t + b_e$, where a_e, b_e are constants. All weight functions are, then, linear in the (t,w) plane.

The three operations our KDS should be able to support are `modify(x,f(t))`, `advance(t)` and `query`. The `query` function is not specified in the paper, as it depends mostly on the interest of the user. Since the kinetic problem allows for events to be insertions and deletions, we can simplify the `modify(x,f(t))` function to simply create and evaluate a deletion and an addition of an edge at the current time. This is probably why it is also not mentioned in the paper. The `advance(t)` function is the main length of the paper. It finds the next event and evaluates it until the current time reaches t. The challenge is mainly to establish how to find the next event and how to process each event.

3.1 Swaps

A swap is an easy enough concept: remove an edge from a spanning tree and add a different edge so that the tree remains a spanning tree. Formally, if T is a spanning tree, $e \in T, f \notin T$ form a swap if the cycle created when adding f to T includes e. The weight of the swap is $w(e,f) = -w(e) + w(f)$, which is by how much the total weight of T would change if we perform the swap.

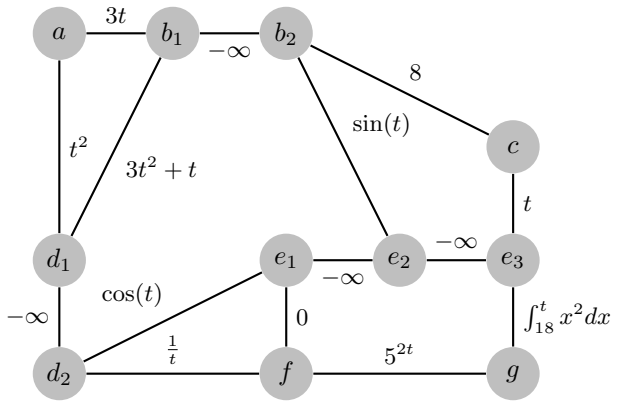

Figure 3.1: A transformation of the graph in figure 2.10 so that all vertices have degree at most three

3.2 Clusters

To avoid recalculating the whole data structure every time anything changes, we partition the graph in a way that helps us maintain the calculations in a local area. Specifically, we need to partition V in sets we will call *clusters* in such a way that:

1. Each cluster contains at most z vertices.
2. Each cluster induces a connected subtree of the MST. That is, if we take the vertices in a cluster and connect them as they are connected in the MST, the resulting graph is a tree.
3. For cluster C, if C contains more than one vertex, then there are at most two edges in the MST having one endpoint in C.
4. No two clusters can be combined and still satisfy the other conditions.

Doing this is possible in linear time, as explained by Frederickson [9].

To take advantage of this, we need to previously transform the graph G into a graph G' with degree at most 3. To do this, we will turn all vertices with degree $\Delta > 3$ to $\Delta - 2$ vertices connected in a line. The two ends take two edges of the original vertex each, and each vertex in the middle takes one. All edges in the path will have as small a cost as possible so that they always are in the MST. Figure 3.1 shows the graph in Figure 2.10 after such a transformation.

Let two edges $e \in$ MST and $f \notin$ MST form a swap. If both endpoints of f are in the same cluster, e is in the same cluster and we call it an *intra-cluster swap.* If the endpoints of f are in different clusters and e is in one of them, we call it a *dual-cluster swap.* If, instead, e is in none of the clusters of the endpoints of f, it's an *inter-cluster swap.*

3.3 Dynamic convex hull

An edge e with weight $w(e) = a_e t + b_e$ can be represented as a line in the (t, w) plane. It can also be represented as a point in the (x, y) plane with coordinates

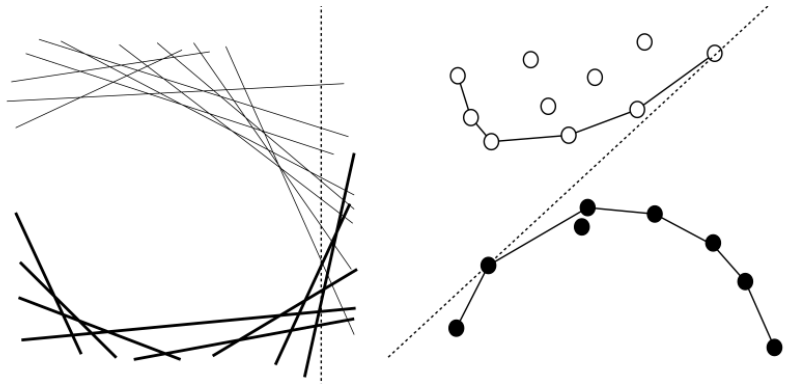

Figure 3.2: From Agarwal et al. [1]: *First non-positive swap: in line arrangement (left), rightmost point above lines from tree edges and below lines from non-tree edges; in dual point arrangement (right), line with highest slope above points from tree edges and below points from non-tree edges.*

$(-a, b)$. This is best seen graphically, as in Figure 3.2, which appears in the same paper [1]. On the left (t, w) plane, each edge is an line. On the right (x, y) plane, each edge is a point.

The dark lines and edges represent the edges that are in the MST, and the clear ones, the ones that aren't. The next swap is the point in the (t, w) plane with t greater than the current time at which a line in the MST crosses a line not in the MST; that is, the value of an edge in the MST becomes greater that one of the edges outside the MST. Analogously, it's the line in the (x, y) plane with greatest slope tangent to the convex hulls of the sets of edges in and out of the MST.

This swap can only be done, though, if the edges crossing can form a swap. Connecting f might not let us release e. This is why we can't use this as our only calculation method, only when we know all edges of the MST will form swaps with edges not in the MST.

At this point, the paper asserts finding the next swap can be done in $O(\log n)$ time, without giving a reference. We used a paper by Cole [4] as the reference.

3.4 Megiddo's parametric search

As we will see in Sections 3.5 and 3.5, often it will be easier for us to find the best swap at a certain time t than to find the value t^* of the first non-positive swap. Parametric search is a technique invented by Megiddo [14] for transforming a decision algorithm into an optimization algorithm. The basic idea is to simulate a basic algorithm that takes as input a numerical parameter X as if it were being run with the (unknown) optimal solution value X^* as its input. For this reason, we call it the *simulated algorithm*. This test algorithm is assumed to behave discontinuously when $X = X^*$, and to operate on its parameter X only by simple comparisons of X with other computed values, or by testing the sign of low-degree polynomial functions of X. To simulate the algorithm, each of these comparisons or tests need to be simulated, even though

the X of the simulated algorithm is unknown. To simulate each comparison, the parametric search applies a second algorithm, a *decision oracle*, that takes as input another numerical parameter Y, and that determines whether Y is above, below, or equal to the optimal solution value X^*.

3.5 Finding the next swap

The next swap can be one of the three types of swaps we explained in Section 3.2: inter-, dual- or intra-cluster. If we can find the next swap of each type, we will be able to find the overall next swap.

Finding the next intra-cluster swap

At any given t we can see if a spanning tree is still minimum with an MST verification algorithm. This can be done in $O(n)$ time, as shown by Dixon, Rauch and Tarjan [8] and King [13].

If we sort a list of the edges by their weight, we'll have a discontinuity when one surpasses the other. According to Cole [5], this can be done in $O(n \log n)$ time, for a graph of n vertices.

We can then calculate the next swap for each cluster by using the first function as the decision oracle and the second as the simulated algorithm for Meggido's parametric search. Since each cluster has $O(z)$ vertices, it will take $O(z \log z)$ time.

Finding the next dual-cluster swap

Let $f = \{f_1, f_2\}$ be an edge not in the MST with one endpoint in a cluster C and another not in C. We call such an edge an *external* edge of C. Notice that f can only form swaps with edges in the path between its endpoints, or the MST would be disconnected.

The MST has at most two edges with one endpoint v, w in a cluster C. Define $\mu_v(f)$ as the edge e on the path in the MST $\{f_1, \ldots, v\}$ that forms a swap with f and where $w_f(t) - w_e(t) = 0$ for the smallest t. Refer to Figure 3.3 for a graphical interpretation.

If we don't have any of the $\mu_v(f)$ edges of a cluster C, we can traverse the tree from each vertex v with one endpoint in C and keep adding them to a convex hull data structure like the one in Section 3.3. Every time we find an external edge, $\mu_v(f)$ is the line of highest slope tangent to the point representation of f and the convex hull of the points added. When we backtrack, we undo the last addition. This can be done in time $O(\log z)$. Since there are $O(z)$ vertices, we can find all of them in $O(z \log z)$ time.

Storing this information lets us maintain a data structure that, for each cluster, contains all of the $\mu_v(f)$ edges. We change $O(1)$ clusters each update, so we can keep it updated in $O(z \log z)$ time per update. Define $\mu_v(C_i), v \notin C_i$ as the pair $(\mu_v(f), f)$ for which $w_f - w_{\mu_v(f)} = 0$ earliest, where $f = \{f_1, f_2\}, f_2 \in C_i$ and f is an external edge of the cluster v is in. There are $O(m/z)$ such C_i per cluster. We can modify this information in time $O(z)$ per modification of a cluster, cycling through $O(z)$ vertices, finding their $\mu_v(f)$ and taking the earliest. We also store a lowest-common-ancestor data structure for the tree formed by contracting each cluster of the partition; this takes time $O(m/z)$

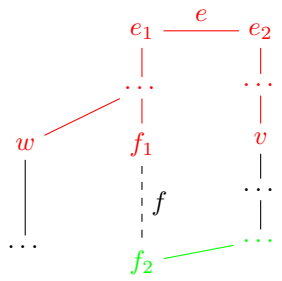

Figure 3.3: A graph with a dual-cluster swap (e-f), with the two clusters marked in red (C) and green (C') and non-tree edges marked as dashed lines

per update to maintain. Thus, we can maintain both data structures in time $O(z \log z + m/z)$ per update.

To find the next dual swap of $f = \{f_1, f_2\}, f_1 \in C_1, f_2 \in C_2$, first we use the lowest-common-ancestor data structure to find the vertices v_1, v_2 that connect C_1 and C_2 in the MST in $O(1)$ time. Then, $\mu_{v_1}(C_2)$ and $\mu_{v_2}(C_1)$ form swaps with f and they can be found also in $O(1)$ time. The best swap of the two can thus be found also in $O(1)$ time.

Finding the next inter-cluster swap

To find the next intra-cluster swap, we can do it by generating a graph G' from the original G and computing the earliest t^* such that $\text{MST}(G(t^*)) = \text{MST}(G'(t^*))$ using Megiddo's parametric search (Section 3.4). The point at which the equality no longer holds lets us determine which swap will happen at that time.

The graph we have to generate, then, needs to reflect the connections between clusters. We turn clusters with one edge in the MST with one endpoint in the cluster into a vertex and clusters with two such edges into an edge between the two endpoints, with its weight equal to that of the heaviest edge in the path between the endpoints. For every two clusters C_1, C_2 connected by a path that exits the clusters through v_1, v_2 respectively, we contract all edges not in the MST between C_1 and C_2 into an edge $\{v_1, v_2\}$ with weight equal to that of the lightest.

3.6 `advance(t)`

The `advance(t)` function, then, will run a loop of finding the next event (insertion, deletion, modification or swap), evaluating it and advancing until the current time reaches the target `t`.

If we reach an insertion, we will update all our structures and check if there is any non-positive swap we need to do.

If we reach a deletion, we need to update our structures and, if the edge was in the MST, replace it with the best option. This is calculated by querying the convex hull data structure of Section 3.3 to find the best swap with the edge we are trying to delete.

If we find the next swap, we perform it: update the structures and the MST and continue.

When we reach the target time t, we're done. We update our structures and exit the loop.

Chapter 4

Implementation

To aid in the understanding of our design, it will be explained in a model. We will try to specify it loose enough so the changes to fit it in either an object-oriented paradigm or a functional style are minimal. These two paradigms include almost all the main programming languages used today [15]. Since the original plan was to use this for a library written in C++, the nomenclature will follow C++ object-oriented standards.

To implement the solution, we will need three small classes (`Graph`, `Event` and `Certificate`) and a main one to tie it all together and do the calculations we will call `Kinetic Minimum Spanning Tree`.

The three simple classes will need a parametric constructor that creates the structure from its components. We have omitted this from the description since they all assign the parameters to the components of the class and take $O(1)$ time.

Additionally, the internal components of each class will be available for consultation by the other classes, either directly or by calling getter functions or some other simple queries that should not take more than $O(1)$ time.

4.1 The `Graph` class

Objects of this class will represent undirected, edge-weighted graphs.

Data structures

We will define a vertex to be an integer which will be its ID. We will maintain a vertex set `V`.

Edges will be represented in an adjacency list `E`. Since it is an undirected graph, we will need to make sure $\forall u, v \in V, u \in \texttt{E[}v\texttt{]} \iff v \in \texttt{E[}u\texttt{]}$.

Finally, we will define `w` to be a function which, for every edge, returns a function of t that is the weight of the edge at that point in time. Since we are using only edge weights of the form $a_e t + b_e$, we can return the pair (a_e, b_e) instead.

Functions

This data structure will support additions and deletions to `E` and changes to `w`.

`add(u,v,a,b)` Adds an edge $\{u, v\}$ to the edge set, which means adding `v` to `E[u]` and `u` to `E[v]`. The weight of the edge is set to $w_{uv}(t) = at + b$. Takes $O(1)$ time.

`delete(u,v)` Deletes edge $\{u, v\}$ from the graph, that is, `v` from `E[u]` and `u` from `E[v]`. Takes $O(1)$ time.

`modify(u,v,a,b)` Changes $w_{u,v}$ to equal $at+b$. Can be simplified to deleting and re-adding the edge with the new weight. Takes $O(1)$ time.

4.2 The `Event` class

An `Event` is a change to the structure either of the graph or of the MST.

Data structures

An event will be of one of three types: an `Addition`, a `Deletion` or a `Certificate failure`. Both `Addition` and `Deletion` refer to edges, not to vertices.

Notice there are no edge weight updates. This is because an updates can be parsed simply as a `Deletion` followed by an `Addition`.

In all three cases, we will store the failure time and make it available for consultation.

We will also store the data necessary for the change. For `Addition` and `Deletion` events, the vertex IDs `u` and `v` and, only for `Addition`, the new function $w_{uv}(t) = at+b$ or its components `a` and `b`. For `Certificate failure` events, we will store the certificate that fails at time t.

Functions

This data structure will support only creations and consultations of its elements.

4.3 The `Certificate` class

A `Certificate` is a boolean assertion that a possible swap between an edge e in the MST and an edge f not in the MST is positive, which means its weight $w = w_f - w_e$ is greater than 0.

Data structures

We need to store the two edges. We can either store a reference to them or the vertex endpoints, depending on what is easiest.

Functions

We will only need a simple parametric creator and queries. Both can be done in $O(1)$ time.

4.4 The `Parametric Minimum Spanning Tree` class

This class will be the workhorse of the design and the actual Kinetic Data Structure. It will contain both the structure we are interested in, as well as the necessary structures to calculate efficiently the next event.

Data structures

Firstly, we will need to store the current time, the current graph G and the current MST. For both G and its MST we will use instances of the `Graph` class. Additionally, we will need to store two data structure to aid in finding the next swap: one for inter-cluster swaps and one for dual-cluster swaps.

Inter-cluster swaps (convex hulls) For each cluster with two terminal clusters, we will store the convex hull of the points in the path between the two terminal vertices of the cluster (see Section 3.3). For each pair of clusters, we will store the convex hull of the points that represent the edges connecting them. To do both of these, we will use the CGAL [21] and its sections on points and convex hulls.

Dual-cluster swaps (lowest common ancestor) For an arbitrary root vertex r in G, we will define the ancestor of any vertex v as any vertex in the path from v to r, including v and r. The lowest common ancestor of two vertices u and v is, then, the ancestor of both u and v that has all of the other ancestors of both u and v as its ancestors. We will store the lowest common ancestor of each pair of vertices.

Dual-cluster swaps (convex hulls) For each cluster and each of its external vertices, we will store a convex hull of the external vertex f and the edges in the path between f and the terminal vertex in the path between the endpoints of f.

Functions

As a KDS, we need to support three functions: `modify`, `advance` and `query`.

query The `query` function is highly dependent on the interest of the user. Some examples include knowing the current MST, its weight or its edges. In any case, they should all be queries on the structure at the current time, not at any future or past time.

modify We will treat modifications as if they were events happening at the current time. We will offer three functions `add`, `delete` and `modify` which will evaluate events at the current time. Algorithm 1 shows a sample `delete` function. Choosing a cluster size of $z = m^2/3$ minimises the update time, creating an event takes $O(1)$ time, and evaluating it takes $O(z \log z + m/z + m^2/z^2 \log^2 z) = O(m^{2/3} \log^{4/3} m)$ time, so any modification will run in $O(m^{2/3} \log^{4/3} m)$ time as well.

Algorithm 1 `delete(u,v)`: delete the u, v edge

```
a ← Certificate.createAddition(u,v)
evaluate(a)
return
```

advance(t*) Advance time until it's equal or greater to `t*`. We will need to calculate the next event and then evaluate it until the next one happens after `t*`. An example is shown in Algorithm 2.

There are two main time sinks: finding the next swap and updating the structures. The next inter-cluster swap can be found in $O(z \log z)$, the next intra-cluster swap in $O(m \log z)$ and the next dual-cluster swap in $O(m^2/z^2)$. Updating the structures is done in $O(m/z + z \log z)$ for the dual-cluster structure and $O(z \log z)$ for the inter-cluster structure for each update. This means our total time goes to $O(p(z \log z + m/z + m^2/z^2 \log^2 z))$, with p being the number of updates. Since the choice of z is ours, we can minimise the time per update by setting $z = m^{2/3}$ and we will get a total time of $O(pm^{2/3} \log^{4/3} m)$.

Algorithm 2 `advance(t*)`: Advance time t until a limit t^*

```
s_e ← nextInter()
s_d ← nextDual()
s_a ← nextIntra()
s ← next(s_e, s_d, s_a)
EventQueue.push(s)
x ← EventQueue.top()
while x.failureTime < t* do
  x ← EventQueue.pop()
  t ← x.failureTime
  updateStructures()
  evaluate(x)
end while
```

We have referenced some functions in the algorithms which we will need to define.

evaluate(e) Evaluate the effects of event `e`. If `e` is an `Addition`, we will need to update G and the MST and any other internal structures. If `e` is a `Deletion`, we will need to check if the edge is in the MST or not. If it is, we'll need to find the best option to replace it.

next(a,b,c) Pick the swap between a, b and c that happens first.

nextInter Find the next inter-cluster swap. Use Megiddo's parametric search, simulating the verification of $\mathrm{MST}(G(t^*)) = \mathrm{MST}(G'(t^*))$, where G' is the contraction explained in Section 3.5. Instead of programming the verification from scratch, we will use a library to do so. It depends on the language, but as an example, the Boost Graph Library [19] could be useful if we are using C++.

Algorithm 3 `evaluate(e)`: Evaluate the effects of event `e`

```
G' ← transform(G)
if x is an insertion then
  Update the structures
  if there is a non-positive swap s then
    perform(s)
  end if
end if
if x is a deletion of an edge e ∈ MST then
  for each group of edges connecting a pair of clusters E_{i,j} do
    find the best replacement edge e_{i,j} for e in E_{i,j}
  end for
  Replace the best e_{i,j}
end if
if x is a deletion of an edge e ∉ MST then
  Update the structures
end if
if x is s then
  Update the structures
end if
```

This process is done in time $O(z \log z)$, using $O(z)$ calls to the decision oracle, which takes $O(\log z)$ time. The simulated algorithm itself takes only $O(1)$ time.

nextDual Find the next dual-cluster swap. For each pair of clusters, query the lowest common ancestor data structure to find the terminal vertices on the path between them and take the lowest of the two. Use the convex hull data structureto find the best swap of the two clusters. Finally, return the earliest swap of all the pairs of clusters.

There are $O(m/z)$ clusters and we need to do an $O(1)$ operation on each pair, so this operation will take $O(m^2/z^2)$ time.

nextIntra Find the next intra-cluster swap. For each cluster, simulate a sort of the edges between vertices in the cluster by their weight and use a check if the MST of the cluster is still minimum to see if we have surpassed the next intra-cluster swap. Instead of programming the verification from scratch, we will use a library to do so. It depends on the language, but as an example, the Boost Graph Library [19] could be useful if we are using C++.

MST verification is shown to be $O(n \log n)$ [8] and the each cluster has $O(z)$ vertices, so this operation will take $O(z \log z)$ time per cluster. Since there are $O(m/z)$ clusters, the function should take time $O(m/z \cdot z \log z) = O(m \log z)$.

Chapter 5

Planning and economic analysis

5.1 Time plan

As explained in Section 1, the scope of the project and with it, the plan, changed during its execution. To help legibility, the original plan is shown in appendix D and this section contains only a summary. The comparison of the three plans (initial, follow-up and estimated final) is shown in table 5.1.

The original plan had seven main task groups. *Project planning* included defining the project and its execution. A report was written for each section to get the specification exactly. This was also used to write up the initial report, with some hours dedicated to the layout of the final report, represented in the *Initial report* task group. *Submission* collated the hours dedicated to the submission process. The design, implementation, testing and improvement of the library itself was included in the *Project execution* group. It was planned as a cascade to design a minimum viable product, then an agile cycle for improvements until the submission process to CGAL was initiated, and finally another agile cycle to close the submission process. The *Final report* and *Follow-up report* groups included the work needed to adapt and finish the required evaluation reports, and the *Presentation* includes the preparation of the defence.

At the time of the follow-up report, the new scope had not yet been set, but a proposal was written up that intended to maintain the scope by cramping as much work into the remaining time as possible. A timing was devised, which we included in table 5.1, but it is mostly not applied.

The *Project planning* and *Initial report* did go as planned, so their estimations reflect that.

In a similar situation is the *Follow-up report*. There were weekly meetings with the project director, so the follow-up report and meeting were planned at a semi-arbitrary point in the development. When trying to re-frame the project and scope, then, it was used as a tool for the plan proposal to be more in-depth. This served as an initial point in the debate.

The hours dedicated to the *Submission* are reduced ten-fold. It didn't take much to find out the latest library didn't support KDSs, but making sure that was the case and trying to find a way around it did, which is why there are three hours in this task even though most of it wasn't done.

The time dedicated in the *Project execution* and *Final report* groups are

Task group	Initial	Follow-up	Final (estimation)
Project planning	105	105	105
Submission	30	0	3
Initial report	30	30	30
Project execution	150	150	102
Final report	140	140	130
Follow-up report	16	16	16
Presentation	30	30	30
Total	501	471	416

Table 5.1: Comparison of time planned or executed in each of the stages of the project in hours

harder to calculate. There were two different periods for work days. Before the 24th of December, the hours dedicated per day were inconsistent depending on the time available, so it is very hard to estimate how much time was used. On the other hand, between Christmas Eve and the delivery deadline of the 21st of January, excluding holidays, the whole day was dedicated exclusively to the project. To make the calculation simple, we suppose the work of the first period compensates the work not done during the second period if we suppose an 8-hour work day. The total comes up to 29 days $\times$ $8\frac{\text{h}}{\text{day}}$ = 232 h, which are divided between the two task groups in as well an estimation as we could manage.

Finally, the *Presentation* hours haven't yet been done, so the estimate remains the same.

5.2 Economic Analysis

Budget

The total budget of this project was of 12290.93 €, most of which is staff salaries. If we suppose the workers were paid, the cost would have been of 10918.38 €.

Income

There is no predicted income for this project.

Expenses

The main costs of the production of this project are the salaries for the workers and the material cost derived from the use of the computer. Table 5.2 shows a summary of the expenses.

Salaries

The salaries for the project lead and developer are taken from [10] and [16], respectively. The total yearly salary is divided by the same ratio in both cases to obtain a consistent hourly rate. The budget separated the tasks between the

Concept	Budgeted cost	Final cost
Project lead salary	10186.55€	10836.80€
Developer salary	2022.80€	0.00€
Electricity usage	10.19€	8.13€
Depreciation	71.39€	71.39€
Total	12290.93€	10918.38€

Table 5.2: Summary of expenses

lead and developer. Since the change in scope (see Section 1 for the details), there is no need for a developer and the whole work is done by the project lead.

Material cost

The only material to be used is a computer. The model used is a MacBook Pro (13-inch, Late 2016). The exact consumption could not be found, but when in use and charging it still charges the battery, which means the maximum wattage is that of the charger, 85W. This is just an upper bound, but since the cost of electricity is so small compared to the salaries, the total budget is just barely skewed.

The cost in Spain of electricity is around 0.23€/kWh [11]. All tasks require the use of the same computer, so for a total of 416h, the total electrical bill will be of 85W × 416h × 0.23€/kWh = 8.13€.

The estimated cost due to amortization is derived from the initial cost, salvage value, time used and useful lifespan [20] as shown in equation 5.1.

$$\begin{aligned} &(\text{Initial cost} - \text{Salvage value}) \times \frac{\text{Time used}}{\text{Useful lifespan}} = \\ &= (2099€ - 100€) \times \frac{4 \text{ months}}{7 \text{ years}} = \\ &= 71.39€/\text{quarter}. \end{aligned} \tag{5.1}$$

Reflection and commentary

The cost of the project is big for a school project, but it is mostly due to the salary being taken as if the staff were paid averagely for a person with their work. A junior programmer or manager would probably get paid less to compensate for inexperience and possible errors. Since it is mostly an academic endeavor that can be done on a single computer, there is not as big of a budget as with other computer engineering projects, especially ones which have to be continually running.

Although the aim of the project is clear, there is no evident application which quickly needs a solution to be viable. This is also the situation for many mathematically-focused studies, that uncover facts that may not be of use until years later. The intention is for the project to contribute to the general availability of KDSs and allow future applications to be more easily found.

Chapter 6

Sustainability

6.1 Environmental sustainability

For the production, the project uses only a computer and a very small amount of electricity, so the environmental footprint is very small. If the computer was to be bought new, a different model may have been more efficient, but the production of a new computer would be much worse for the environment than the use of a slightly less efficient one.

For the rest of the product life, at most it might cause some calculations by someone using our design to be more efficient, and thus, use less power. The environmental effect, though, is highly dependent on the exact applications. Worst-case it will have no effect, so any other situation improves in an environmental sense.

6.2 Social sustainability

This is the last part of my degree, which will give me certification of the things I have learned in the past few years. Additionally, I feel I will be slightly contributing to the general knowledge of humanity, which I think is a noble goal.

It may be that the project is done and then no one ever uses the implementation because the potential applications have better solutions in other frameworks, but it is equally likely that suddenly KDSs become pivotal in some new algorithmic strategy for an important problem. The exact repercussions are hard to predict, but it seems the lower bound is that there is no effect, so anything that deviates from that is positive.

6.3 Self-assessment

Completing the survey has made me see that, although I believe myself pretty involved in environmental and social issues, I have still much to learn in this field. I noticed the phrases I agreed most with were when the intention was the focus, rather than the knowledge or ability.

I believe this to be the result of two things. On the one hand, there is a lack of content in the degree about it. On the other, though, sustainability is treated by both students and faculty as this unimportant, added-on part

of subjects. In my opinion, we could improve this if this kind of evaluations influenced the final score of the subjects.

I knew already, from my studies, that economic matters tend to be very unappealing to me, and though I can do the analyses, it requires from me a great deal of effort. The projects I have done in school have required little or no infrastructure, so an economic study has never felt like an important evaluation. I understand, though, that in the capitalism we live in money is always an issue to address, especially in computer science projects, in which size can quickly get out of hand.

Chapter 7

Conclusions and future work

We have explained the details of the paper by Agarwal et al. [1] and designed a library to provide for its implementation by using a myriad of different techniques, including geometrical structures like convex hulls, combinatorial algorithms like Megiddo's parametric search and graph theoretic algorithms.

Implementation of the library has been an objective from the start, but different factors around the project have not allowed for a full implementation to be finished in time. This is a clear next step, which would greatly improve both the design and efficiency by allowing for testing to be done, so any flaws in reasoning or wrong assumptions could be exposed.

A second avenue for furthering this project would be to include additional features from the solution which did not make it into the design but can improve slightly the lower bound. The two main venues for attack are allowing for randomisation in some of the algorithms and applying sparsification.

Additionally, the paper offers even better solutions if the graphs studied are planar or from another minor-closed family of graphs. This could be studied and added to improve efficiency in these particular cases.

Bibliography

[1] Pankaj K Agarwal et al. 'Parametric and kinetic minimum spanning trees'. In: *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*. IEEE. 1998, pp. 596–605.

[2] Julien Basch, Leonidas J Guibas and John Hershberger. 'Data structures for mobile data'. In: *Journal of Algorithms* 31.1 (1999), pp. 1–28.

[3] Julien Basch, Leonidas J Guibas, Li Zhang et al. 'Proximity Problems on Moving Points'. In: *Symposium on Computational Geometry*. 1997, pp. 344–351.

[4] Richard Cole. 'Searching and storing similar lists'. In: *Journal of Algorithms* 7.2 (1986), pp. 202–220.

[5] Richard Cole. 'Slowing down sorting networks to obtain faster sorting algorithms'. In: *Journal of the ACM (JACM)* 34.1 (1987), pp. 200–208.

[6] Wikipedia contributors. *Minimum spanning tree — Wikipedia, The Free Encyclopedia*. 2019. URL: `https://en.wikipedia.org/w/index.php?title=Minimum_spanning_tree&oldid=916913357` (visited on 21/11/2019).

[7] Erik D. Demaine. *Lecture 4, Course 6.851: Advanced Data Structures*. 2012. URL: `http://courses.csail.mit.edu/6.851/spring12/lectures/L04.html?notes=4`.

[8] Brandon Dixon, Monika Rauch and Robert E Tarjan. 'Verification and sensitivity analysis of minimum spanning trees in linear time'. In: *SIAM Journal on Computing* 21.6 (1992), pp. 1184–1192.

[9] Greg N Frederickson. 'Ambivalent Data Structures for Dynamic 2-edge-connectivity and k smallest spanning trees'. In: *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*. IEEE. 1991, pp. 632–641.

[10] Glassdoor.es. *Sueldo: Project Leader*. 2019. URL: `https://www.glassdoor.es/Sueldos/project-leader-sueldo-SRCH_KO0,14.htm?countryRedirect=true` (visited on 07/10/2019).

[11] GlobalPetrolPrices.com. *Spain electricity prices*. Mar. 2019. URL: `https://www.globalpetrolprices.com/Spain/electricity_prices/#hl223` (visited on 07/10/2019).

[12] Leonidas J Guibas. 'Kinetic data structures–a state of the art report'. In: (1998).

[13] Valerie King. 'A simpler minimum spanning tree verification algorithm'. In: *Workshop on Algorithms and Data Structures*. Springer. 1995, pp. 440–448.

[14] Nimrod Megiddo. 'Applying parallel computation algorithms in the design of serial algorithms'. In: *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*. IEEE. 1981, pp. 399–408.

[15] Stack Overflow. *Developer survey 2019*. 2019. URL: `https://insights.stackoverflow.com/survey/2019#technology` (visited on 19/01/2020).

[16] PayScale. *Software developer salary in Spain*. 2019. URL: `https://www.payscale.com/research/ES/Job=Software_Developer/Salary` (visited on 07/10/2019).

[17] John Renze and Eric W. Weisstein. *"Discrete Mathematics." From MathWorld–A Wolfram Web Resource*. URL: `http://mathworld.wolfram.com/DiscreteMathematics.html` (visited on 24/09/2019).

[18] Daniel Russel. 'Kinetic Data Structures'. In: *CGAL User and Reference Manual*. 4.11.3. CGAL Editorial Board, 2018. URL: `http://doc.cgal.org/4.11.3/Manual/packages.html#PkgKdsSummary`.

[19] Jeremy Siek, Lie-Quan Lee and Andrew Lumsdaine. *The Boost Graph Library (BGL)*. 2001. URL: `https://www.boost.org/doc/libs/1_72_0/libs/graph/doc/index.html`.

[20] Apple Support. *Vintage and obsolete products*. 2019. URL: `https://support.apple.com/en-ca/HT201624` (visited on 07/10/2019).

[21] The CGAL Project. *CGAL User and Reference Manual*. 4.11. CGAL Editorial Board, 2017. URL: `http://doc.cgal.org/4.11/Manual/packages.html`.

Appendix A

List of Figures

Appendix B

List of Tables

List of Algorithms

Appendix D

Original timeplan

D.1 Description of tasks

In this chapter, we define a list of tasks to be carried out, in detail. All tasks are done by the developer and require only a computer to develop on, so this information is omitted from the task descriptions. Table D.1 shows a summary of the task list.

Project planning

Define the scope and context

Code PP1
Description Define the scope of the project and context in which it will be developed. Create a report summarising the information.
Estimated workload 20 hours
Depends on -
Dependent on this task IR1 Correct and finish initial assessment report

Define the time plan

Code PP2
Description Define the time plan for the entirety of the project. Create a Gantt chart and a report summarising the information.
Estimated workload 20 hours
Depends on -
Dependent on this task IR1 Correct and finish initial assessment report

Make the economic and sustainability reports

Code PP3
Description Prepare an economic report on the project. Prepare a document showing the social, environmental and economic sustainability of the project.
Estimated workload 20 hours
Depends on -
Dependent on this task IR1 Correct and finish initial assessment report

Code	Time (h)	Depends on	Dependent
PP1	20	-	IR1
PP2	20	-	IR1
PP3	20	-	IR1
PP4	30	-	PE1
PP5	15	-	SU1
PP	105		
SU1	10	PP5	SU2
SU2	20	SU1	-
SU	30		
IR1	30	PP1-PP3	-
IR	30		
PE1	50	PP4	PE2, FR3
PE2	40	PE1	PE3, PE4, FR4
PE3	30	PE2	FR5
PE4	50	PE2	-
PE	150		
FR1	10	-	FR7
FR2	15	-	FR7
FR3	25	PE2	FR7
FR4	15	PE3	FR7
FR5	25	PE3	FR7
FR6	30	-	FR7
FR7	20	FR1-FR6	-
FR	140		
FU1	15	-	FU2
FU2	1	FU1	-
FU	16		
PR1	30	-	-
PR	30		
Total	501		

Table D.1: Summary of tasks to be fulfilled

Read literature on minimum spanning tree kinetic data structures

Code PP4
Description Do a bibliographic search on kinetic data structures that hold a minimum spanning tree with continuously changing edge weights and similar constructs to both the MST and KDS topics.
Estimated workload 30 hours
Depends on -
Dependent on this task PE1 Design

Read library protocol and submission process

Code PP5
Description Read and review the protocol and process to add new features to the kinetic data structure section of the Computational Geometry Algorithms Library [18]. Make sure it fits as expected into the time plan or adapt accordingly. Observe the derived requirements for the implementation of the library.
Estimated workload 15 hours
Depends on -
Dependent on this task SU1 Prepare formal submission

Submission

Prepare formal submission

Code SU1
Description Prepare the necessary information and documentation to submit the work to the Computational Geometry Algorithms Library [18]. Send the submission according to their process.
Estimated workload 10 hours
Depends on PP5 Read library protocol and submission process
Dependent on this task SU2 Respond to corrections and commentary from reviewer

Respond to corrections and commentary from reviewer

Code SU2
Description Once the submission process starts, a reviewer assigned by the CGAL editorial board will send back commentary and corrections to be done to have the changes included to the library. These changes will have to be implemented.
Estimated workload 20 hours
Depends on SU1 Prepare formal submission
Dependent on this task -

Initial report

Correct and finish initial assessment report

Code IR1
Description Once all sections of the initial assessment report are done, the

corrections and commentary received from the teacher will be taken into account to make the initial assessment report. Some time will also be taken to put together the document from its parts.
Estimated workload 30 hours
Depends on PP1 Define the scope and context, PP2 Define the time plan, PP3 Make the economic and sustainability reports
Dependent on this task -

Project execution

Design

Code PE1
Description Define what classes are going to be implemented and how they will communicate with each other. Create a high-level version of the algorithms found in the literature and analyse which one should perform best. Define the high-level inner workings of the chosen algorithm.
Estimated workload 50 hours
Depends on PP4 Read literature on MST KDSs
Dependent on this task PE2 Implement, FR3 Write final report section on design

Implement

Code PE2
Description Create a working example of the design created in PE1.
Estimated workload 40 hours
Depends on PE1 Design
Dependent on this task PE3 Test, PE4 Make changes and additional features, FR4 Write final report section on implementation

Test

Code PE3
Description Test the implementation experimentally to make sure it works as expected and complies with all requirements.
Estimated workload 30 hours
Depends on PE2 Implement
Dependent on this task FR5 Write final report section on testing

Make changes and additional features

Code PE4
Description If the results of the tests are positive, design, implement and test additional features that can be useful. If not, make the needed changes to make sure everything works as it should.
Estimated workload 50 hours
Depends on PE2 Implement
Dependent on this task -

Final report

Write final report section on minimum spanning tree kinetic data structures

Code FR1
Description Write the section of the final report concerning minimum spanning trees, kinetic data structures and their combination.
Estimated workload 10 hours
Depends on Start of PP4 Read literature on MST KDSs
Dependent on this task FR7 Finalise the report

Write final report section on library protocol

Code FR2
Description Write the section of the final report concerning the submission process to the Computing Geometry Algorithms Library [21] and the requirements for submitted code.
Estimated workload 15 hours
Depends on Start of PP5 Read library protocol and submission process
Dependent on this task FR7 Finalise the report

Write final report section on design

Code FR3
Description Write the section of the final report that explains the design process and results.
Estimated workload 25 hours
Depends on PE2 Design
Dependent on this task FR7 Finalise the report

Write final report section on implementation

Code FR4
Description Write the section of the final report concerning implementation details and limitations.
Estimated workload 15 hours
Depends on PE3 Implement
Dependent on this task FR7 Finalise the report

Write final report section on testing

Code FR5
Description Write the section of the final report that explains the testing process, results and conclusions.
Estimated workload 25 hours
Depends on PE3 Test
Dependent on this task FR7 Finalise the report

Write final report sections on project development

Code FR6
Description Write the sections of the final report that explain how the project has developed in relation to the planning and what adaptations have been made.
Estimated workload 30 hours
Depends on -
Dependent on this task FR7 Finalise the report

Finalise the report

Code FR7
Description Compile all written sections of the report and structure the document correctly. Make sure it contains no grammatical or typographical errors. Ensure all explanations are complete. Format consistently and correctly.
Estimated workload 20 hours
Depends on FR1 Write final report section on MST KDSs, FR2 Write final report section on library protocol, FR3 Write final report section on design, FR4 Write final report section on implementation, FR5 Write final report section on testing, FR6 Write final report sections on project development
Dependent on this task -

Follow-up report

Prepare the follow-up report

Code FU1
Description Use written sections of the final report and other material to prepare the follow-up report.
Estimated workload 15 hours
Depends on -
Dependent on this task FU2 Follow-up meeting

Follow-up meeting

Code FU2
Description Hold a meeting with the director presenting the follow-up report. Discuss project direction and schedule.
Estimated workload 1 hours
Depends on FU1 Prepare the follow-up report
Dependent on this task -

Presentation

Prepare the oral defence

Code PR1
Description Prepare an oral presentation explaining the project, including the plan, development and results.
Estimated workload 30 hours

Depends on -
Dependent on this task -

D.2 Gantt chart

Figure D.1 shows the resulting Gantt chart of the tasks described above. To show information in as good a layout as can be managed, the figure appears in landscape mode.

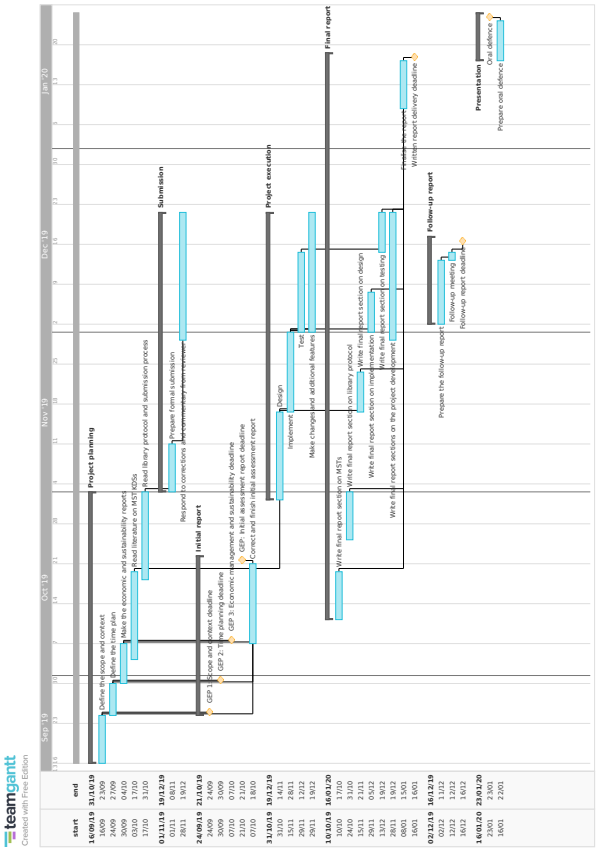

Figure D.1: Gantt chart for the project

D.3 Alternative plans

There are two main risks in the carrying out of this project: that there may not be an appropriate solution to the problem we are trying to solve or that the time plan is not able to be followed accurately.

If there is no appropriate solution

Tasks SU1 and SU2, which concern the formal submission to the CGAL, will not be carried out. Instead, a study of the problems encountered will be done and added as a section of the final report.

If the time plan is not able to be followed accurately

Using the dependencies defined in the time plan, tasks will be reorganised and retimed if a big deviation from the plan is detected. It depends on the stage the project is at, but tasks concerning the initial, follow-up and final assessments will be prioritised. The rest may be shortened to arrive at the destination on time.