The Spatial and Metric Template library: Final Report

Joaquim Casals
Director: Conrado Martinez
Co-Director: Amalia Duch Brown
Specialization: Computing

June 21, 2015
Abstract(Català)

Aquest projecte està basat en el PFC (Projecte final de carrera) "the Spatial and Metric Template Library (SMTL)" realitzat per Guillem Marpons sota la supervisió de Conrado Martinez. La SMTL és una llibreria genèrica i de codi obert d'algorismes i estructures de dades espacials i mètriques. Aquest tipus de dades són presents en una gran multitud de problemes computacionals ja que molts problemes admeten una interpretació geomètrica de les seves dades. Per tant, disposar de bones eines que poden treballar amb aquest tipus de dades serà útil en molts contextos. Per exemple, es pot utilitzar en problemes de reconeixement de patrons, o en un camp com l’aprenentatge automàtic.

L’objectiu principal d’aquest projecte és desenvolupar les funcionalitats necessàries per tenir una llibreria funcional. La idea clau darrere la SMTL és desenvolupar totes aquestes eines de manera que el programador les pugui utilitzar de forma transparent, és a dir, sense conèixer els detalls de la implementació, però poden gaudir dels coneixements més recents en aquest camp de recerca.

Per altra banda, que sigui escalable i fàcil d’ampliar és un objectiu important. Aquest projecte vol tenir continuïtat i que altra gent pugui contribuir, millorant la llibreria, readaptant-la a les seves necessitats, etc...
Abstract (Castellano)

Este proyecto se basa en el PFC (Projecte final de carrera) de Guillem Marpons llamado "the Spatial and Metric Template Library (SMTL)" dirigido por Conrado Martinez. La SMTL es una librería genérica y de código abierto de algoritmos y estructuras de datos espaciales y métricas. Este tipo de datos son presentes en gran multitud de problemas computacionales ya que muchos problemas admiten una interpretación geométrica de sus datos. Es por eso que disponer de buenas herramientas que puedan trabajar con este tipo de datos será útil en muchos contextos. Por ejemplo, se puede utilizar en problemas de reconocimiento de patrones o en el campo del aprendizaje automático.

El objetivo principal de este proyecto es desarrollar las funcionalidades necesarias para tener una librería funcional. La idea clave detrás la SMTL es desarrollar todas estas herramientas de manera que el programador las pueda utilizar de forma transparente, o sea, sin conocer las particularidades de la implementación, pero teniendo a su alcance todos los conocimientos más recientes en este campo de investigación.

Por otro lado, que sea escalable y fácil de ampliar es un objetivo importante. Este proyecto quiere tener continuidad y que otra gente pueda contribuir a la librería, mejorándola, adaptándola a sus necesidades, etc...
Abstract (English)

This project is based on the PFC (Projecte final de carrera) of Guillem Marpons called ”the Spatial and Metric Template Library (SMTL)” directed by Conrado Martinez. The SMTL is an open-source generic library of algorithms and spatial and metric data structures. This type of data is present in a variety of computational problems since most problems admit a geometric interpretation of its data. Therefore, having good tools that can deal with this type of data will find lots of uses in a myriad of contexts. For instance, it can be used in pattern recognitions problems, or in a field like machine learning.

The main objective for this project is developing the necessary features to have a functional library. The key idea behind this library is to develop a set of tools that programmers can use in a transparent way, i.e. without knowing the detail of how it works, but using the state of the art knowledge in spatial and metric data structures.

Also scalability and future development is an important objective. This project wants to have continuity, that is, that other people might contribute and continue improving the library, or maybe re-adapt it to its needs, etc...
Chapter 1

Introduction

Spatial and metric data can be found in a lot of problems. For instance, in a multimedia database, it is often the case where the objects (pictures, video, etc...) are too complex, and we deal with some extracted features that allow us to work with lighter objects. Another instance where we find this kind of data is in rendering software, for instance we need a way to structure of a scene in order to efficiently perform a ray-tracing algorithm. In all this cases, it is also the case that there is no universal worst-case best solution, and we have to be content with average cases. Therefore, it makes sense to have a somewhat big set of tools to deal with this data and so we don’t have to develop ad-hoc solutions to structure our problem’s data. This is the main motivation of the SMTL, to provide general and parametrizable data structures that can be adapted to the problem at hand.

It seems only obvious to take as a referent the Standard Template Library (STL). The STL is a C++ library that relies on templatization and iterators to provide general data structures and algorithms for various problems. Like the STL provides templatized algorithms and data structures, the SMTL aims to deliver a similar set of tools but centered around the spatial and metric data structures and the algorithms related, such as nearest neighbour search or range search. There are two key ingredients to the STL that will also be central in the SMTL

- Templatization
- Iterators

This two tools are basic to give the SMTL enough generality and flexibility to suite a range variety of needs. In the following chapters we will discuss more lengthy this two topics.

1.1 Scope

From the preceding section we can conclude that the SMTL is somewhat of a neverending task. We could always implement one new thing, or tweak one part. That is why we have set some initial goals. Should this goals be fulfilled in due time, we may consider adding extra functionality.

Although the design part of the library has already been finished, most of the tasks will be related to implementing new features rather than redesigning, although the latter is not totally discarded. First of all we will implement the following data structures:

Spatial data structures

Final report
1.2. METHODOLOGY

- Standard Kd-tree
- Relaxed Kd-tree
- Squarish Kd-tree

Metric data structures
- VP tree
- Generalized Hyperplane Trees (GHT)

Once this is considered done (following some criteria that we’ll define in the Methodology section) then the algorithms will be implemented. As of right now, the library already contains two searching algorithms, which are Depth first search (DFS) and Hierarchical Ranking Search (HRS). The idea would be to extend this list.

Along the algorithms implementations we will design and implement a system to efficiently retrieve the results of the search performed by the algorithms. Because, obviously, any algorithm has an output. In this case, our algorithms are search algorithms, which retrieve elements that fulfill some criteria defined by what we call a ”query”. In conjunction with the algorithms we must provide a good way to retrieve this answer, which is the ultimate purpose of our library.

Finally we want to also design and implement a system to generate and retrieve performance data so as to assess how our library behaves, and the quality of our work.

1.2 Methodology

First we will implement the data structures and validate them. Once this is done, we’ll work in parallel in the design and implementation of the algorithms and the ”search result” retrieving system. Once this is validated, first on its own and after used with the data structures previously defined, we will proceed to assess the overall performance of the system.

1.2.1 Tools

This library is developed in C++, so we will use this language. We have decided to stay with the 98 C++ standard, because right now it’s the most supported C++ version. In a foreseeable future this library could be readapted using the full capacity of the latest C++ standards, but this falls out of the scope of this project. Regardless of the standard, C++ gives us good generality, through its template system, while being ”low level” enough to be efficient. Therefore, we can build somewhat abstract code while retaining the efficiency that the C++ compilers gives us.

1.3 Difficulties

The data structure implementation part should be pretty straightforward and should not present much trouble. Implement the algorithms might be a little bit trickier. Firstly, because we must adapt the algorithms to our particular implementations, although this should not cause much trouble, it can be more problematic than the data structure implementation.

But the real challenge is designing the search result retrieval system. We must find a way, that using the already existing library’s architecture allows for such a system, and on top of that, not only it must be feasible but also it must be efficient.
Another difficult part is assessing the quality of our library. It is not straightforward to find a metric that allows us to conclude which data structure performs the best and under what conditions, so these will be another important part. Granted, it is not crucial, since our main goal is to develop a large variety of tools, so it is not so much about which is the better, but to have a set of well-implemented data structures, alongside well-implemented algorithms that can work on these data structures.

1.4 Contextualization

In this project there are two main areas of interest, those are:

- Spatial and metric data structures
- C++ template system

Spatial and metric data structures  What do we mean when we say spatial and metric data? For spatial data we mean points of dimension N (that is, with N components or "values") that can be embedded in a geometric space. In other words, we want points for which it makes sense to ask "where they are" so for example, we can ask which points fall in a rectangle defined by point $X$ and $Y$. When we refer to metric data, what we essentially ask is that for any two points, we have a way (a function) that tells us how far away they are, or in other words, how similar they are. It’s important to note that we can have spatial AND metric data, meaning that both conditions can be satisfied at the same time if we define the points properly.

C++ template system  Templates are a system that allows us to parametrize our code. As it is explained in "Part I: The basics" from the Josuttis’ book "C++ Templates: The complete guide" [VJ02]. This system allows us to reuse program code in an elegant and efficient way. For example, a binary search tree behaves the same whether it stores ints or bools, therefore it makes sense that we implement it once and not one tree for each type.

1.4.1 State of the art

In this section, we’ll give a brief overview of the state of the art in the fields related to the project.

Spatial and metric libraries

From the time the SMTL was first developed (2002) to now, there has been some project similar to the SMTL. It is important to remark similar, because we haven’t found any project that tries to solve the same problem exactly as the SMTL, which is to give a robust, efficient and general data structures that support queries of different types. In the following lines we’ll comment two of this projects:
ANN  Approximated nearest neighbour (ANN) \cite{MA10} is a library that comes from the academia (Maryland University) and is opensource as the SMTL. The ANN implements different data structures that support nearest neighbor queries (both approximate and exact queries). This library comes quite close to the SMTL goal but there are two main differences. First, the SMTL aims at nearest neighbour queries, but also other types of queries. Also, the ANN is not designed in a STL-way (Standard Template library). That means, that the ANN doesn’t offer the customisation possibilities that the templates allow.

Metric Spaces library  The Metric Spaces library \cite{ENC07} is an open-source library that implements different metric data structures. Clearly, this library focuses on the metric data and doesn’t provide solutions for spatial data. It is implemented in C and Java, and like the ANN, the Metric Spaces library doesn’t follow the STL philosophy.

These two projects are the ones we have found that come closer to what the SMTL is/expects to be. We think that the SMTL can fill a gap because there is no general, templatized library aimed at dealing with different types of spatial and metric data, and designed to be easy to use but fully customizable. Also, we must keep in mind that these project doesn’t start from step 0, because SMTL version 1.0 has already some functionality and has a solid design upon which we’ll work and expand the functionality of the library. These two projects are just examples of the fact that the idea of having a general, customizable set of tools to deal with spatial and metric data has not been fully explored yet.

Main datastructures

This section lists some of the most important data structures devised to deal with spatial and metric data.

**Kd-tree**  The k-d tree is a binary tree in which every node is a dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k-dimensions, with the hyperplane perpendicular to that dimension’s axis It is noteworthy the fact that for the Kd-tree there are multiple variants (such as the squarish Kd-trees \cite{DJZC06}) which I will not comment in this section. That goes to show that KD-trees are a flexible and potent tool for storing spatial and metric data.

**Vantage point tree**  A vantage-point tree, or VP tree is a BSP tree that segregates data in a metric space by choosing a position in the space (the “vantage point”) and dividing the data points into two partitions: those that are nearer to the vantage point than a threshold, and those that are not. By repeatedly applying this procedure to partition the data into smaller and smaller sets, a tree data structure is created where neighbors in the tree are likely to be neighbors in the space. This tree-structure is specifically designed to deal with metric spaces, but has the ”problem” that it isn’t dynamic, meaning that once the vp-tree is constructed, we cannot insert or eliminate points, while preserving the structure.
Generalized Hyperplane tree The Generalized Hyperplane tree (GHT) \cite{Uh91} are tree-like structures that partition the space using two reference points at each step. Given this two points (A and B), it recursively divides the point set in points closer or more similar to point ”A” and points closer or more similar to point ”B”. This structures only requires for the points to have a metric defined.

General bibliography overview

As a final note, I want to comment a couple of books that will be useful to develop this project. This books are Foundations of Multidimensional and Metric Data Structures \cite{Sam06} and Computational Geometry: Algorithms and Applications \cite{BCKO08}. The first book, as the title implies, deals exclusively about spatial (multidimensional) and metric data structures. The second book deals with computational geometry in general, but it has some chapters devoted to comment on data structures, and space partitioning. Both books are very instructional and designed in a way that combines both theory and practice in the form of exercises/worked examples. Obviously, this will not be the only source of information, as seen in the previous sections.
Chapter 2

Spatial & Metric data structures

In this chapter we will review the current data structures implemented with their variations. For this part I will be using, among others, the article "Incremental Similarity Search in Multimedia Databases" (Hjaltason and Samet) [HS00] which is in turn part of the more extended [Sam06].

As the title of this chapter implies, this structures can be divided in two big groups

- Spatial data structures which are based in coordinates. These structures make use of the fact that some objects can be represented by means of a multidimensional vector. Obviously this type of structures are specially indicated for objects that belong to multidimensional spaces.

- Metric data structures which are based in a function $f : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ that accepts two objects of the domain and returns a number indicating the similarity between those objects. This data structure are devised to deal with objects for which we don’t have clear features that we could use in a multidimensional space, and we can only work with the similarity or distance value.

![Figure 2.1: Kd-tree example](image)

**Standard Kd-tree** The Kd-tree (or k-dimensional tree) is a data structure that divides the space with axis aligned (k-1)-dimensional hyperplanes. If $\mathcal{S}$ is the collection of elements of the tree, at each level we have a point of $\mathcal{S}$ that defines a hyperplane going through it. When we insert a new element $x$ we just traverse the tree in the following way.
• If \( x \) is to the left of the hyperplane, proceed through the left subtree
• If \( x \) is to the right of the hyperplane, proceed through the right subtree
• If \( x \) belong to the hyperplane, then we must define first to which subtree we will assign this cases. We can choose left or right.

This hyperplanes are the result of discriminating with respect to one of the \( k \) coordinates of the elements’ key. Therefore, each node of the tree has a pair \(<\text{Key, Element}>\) and a discriminant \( 0 \leq d \leq k \). For the standard variant, this discriminant is chosen in a cyclic way as we traverse the tree downwards. That is, if the ancestor node has discriminant \( c \) then the node has discriminant \((c + 1) \mod k\).

This data structure is dynamic, in the sense that allows insertion and deletion of elements, although the deletion might be inefficient. One solution is to just reinsert the elements of the sub-tree rooted at the eliminated node. This approach is general for any Kd-tree variation, but for example for the standard Kd-tree we could just substitute the root for a suitable value, or rebuild the sub-tree and then append it again.

![Figure 2.2: Squarish Kd-tree example](image)

![Figure 2.3: Picking the hyperplane that cuts the longest edge](image)

**Squarish Kd-tree** One variant of the Kd-tree is the squarish Kd-tree introduced in [DJZC06]. This version differs in the way the discriminant is calculated. Instead of picking it in a cyclic way, what we will do is pick the discriminant for which the hyperplane induces...
a more "squared" region. In order words, what we do is to split by the coordinate whose interval is the greatest in the region bounded by the ancestor’s hyperplanes. An example will help clarify the concept. Say we have the squarish Kd-tree of the figure 2.3 where \( k = 2 \). As we can see, we can “theoretically” pick \( d \) hyperplanes (in this case 2) but in the squarish we pick hyperplane which splits with respect to the longest edge. As one might think, we can have multiple hyperplanes that split along the longest edge, in which case we can pick whichever we want, e.g we could pick one randomly.

**Relaxed Kd-tree** Under "relaxed" I have put two different variants that make use of randomization. The first one, which we shall call "relaxed kd-tree", relies on the discriminant selection. Essentially what we will do is pick the discriminant in a uniformly distributed way. The other variant, introduced in [DECM98], consists in having Kd-trees where given \( n \) elements, any of the \( K^n \cdot n! \) configurations resulting from their respective insertion sequences is equally probable. To that effect we require two operations called join and split, that allows to insert elements into a Relaxed randomized Kd-tree while keeping it randomized [DM09].

**Quadtree** Another spatial data structure is the Quadtree. It receives this name because each node of the tree has 4 children. This trees can be seen as a particular case of a more general tree, similar to the Kd-trees in the sense that we split the space with axis-aligned hyperplanes but in the case of this general tree, at each level we split with \( d \) hyperplanes, instead of 1. Therefore, at each level we divide the space into \( 2^d \) regions instead of the 2 of the Kd-tree. For the Quadtree we have two main variants

- **Point**
- **Region**

![Figure 2.4: Point Quadtree example](image)

![Figure 2.5: Region Quadtree example](image)

Any Quadtree, regardless of its kind, has 4 children usually called North West (NW), South West (SW), South East (SE) and North East (NE). The main difference between
the two types is that the point Quadtree splits the space with respect to the data that it stores, and the region tree splits the space without taking into account the data it contains. A point Quadtree stores a point in each node, and then defines the hyperplanes with respect to this point, while the region variant at each level splits the space in 4 equal sized regions. This recursive subdivisions continue until each region holds a maximum number of elements, which is arbitrary. It follows then, that all elements will be found at the leaves of the region Quadtree.

**General costs of operations on multidimensional data** All this structures have expected depth $O(\log n)$. For a general analysis of the costs of different operations such as the expected cost of a partial match or a range query these are performed, for example, in \[Cre10\] and \[Bro04\].

**Vp-tree** The Vp-tree (independently found by \[Yia93, Uhl91\]) belong to the ball partitioning methods. It partitions the data by picking a pivot $p$ (called the vantage point) and a distance $r$. With this, the rest of elements are divided according to their distance to $p$. For any element $x \neq p$ we define two sets $A$ and $B$.

- If $d(p, x) < r$ then $x \in A$
- If $d(p, x) > r$ then $x \in B$
- If $d(p, x) = r$ then we can pick either, but this choice must be consistent (always the same, within the tree)

![Figure 2.6: The Vp-tree performs a ball decomposition of the space](image)

This structure is static. We can build a Vp-tree from a collection of elements, but once build it doesn’t support insertion nor deletion. This data structure was originally devised to perform nearest neighbor queries. This variant, at each level only divides the data in two, but there are other variants where we pick more than one pivot

**Vp$^{sb}$-tree** This variant, introduced in \[Yia93\], stores at each node the distance from the object at that node to all its ancestors. This change allows us to define two bounds for each node

Given a query object $o$

$$d_{1o}(q, o) = \max_i |d(q, p_i) - d(p_i, o)|$$

Final report CHAPTER 2. SPATIAL & METRIC DATA STRUCTURES
\[ d_{hi}(q, o) = \min_i |d(q, p_i) + d(p_i, o)| \]

Where the \( p_i \) are the ancestral pivots of \( o \). With this boundaries, for example, when performing a ball query with radius \( r \), we can discard when \( d_{lo}(q, o) > r \), or just report it when \( d_{hi}(q, o) \leq r \). Also when performing nearest neighbour queries, with some tweaking of the algorithm, we can get a better performance using this bounds.

**Gh-tree** The Gh-tree [Uh91] is a metric data structure that partitions the data with respect to two points at each level. Given a collection of elements and a metric function \( d \), we pick two elements (pivots) \( p_1, p_2 \). Then we divide the collection in two groups

- Points closer to \( p_1 \)
- Points closer to \( p_2 \)

![Gh-tree example](image)

Figure 2.7: Gh-tree example

Again, if we have any equidistant point we can assign it to whichever group, but in a consistent way. This structure does accept insertion, although it is primarily devised to help perform nearest neighbour searches. What this structure doesn’t support is deletion, although we could always rebuild the tree.

**GNAT** the Geometric Near-Neighbour Access Tree (GNAT) is a variant of the Gh-tree. If instead of at each level picking 2 pivots, we pick \( M \) pivots then, at each level, each object is assigned to the sub-tree rooted at the closest pivot. This kind of decomposition is related to the Voronoi partition of a space.

**AESA** As a final example of metric data structure we have the Approximating and Eliminating Search Algorithm (AESA). In this case, we don’t have a hierarchic data structure, but instead a matrix of distances. This matrix has any distance between two given objects of the set. Then, at the price of a costly pre-process (in both time and space) it has been experimentally showed that we can solve queries in time \( O(1) \).

Obviously there are many other data structures and variations that I won’t mention. This section is just a taste and to see that there is a huge variety. There are data structures tuned to perform Nearest Neighbor search (for example AESA), there are more general data structures that try to exploit some property to make working with data easier (Kd-tree, for example), we can also find data structures fine tuned to work with secondary memory, (that is, taking into account that the information will be stored in a disk or some external device and limit the size of the nodes accordingly).
Chapter 3

A learning guide to SMTL

In this chapter I will give an overview to the functionalities of the SMTL.

3.1 C++ in the context of the SMTL

I would like to review some features of C++ that are used in the SMTL. The first thing to note is that the SMTL has been programmed using the C98 C++ standard and not any of the new C++ standards, that add type inference, anonymous functions etc... This decision was made in order to offer the maximum compatibility with the different C++ compilers, and also because we felt that the potential benefits of using things like type inference, weren’t important enough in the context of the SMTL to warrant their use. One feature of C++ that the SMTL makes heavy use, and in a sense is the spirit of the library, is templatization.

3.1.1 Template vs Heritance

In C++ we have, essentially, two ways to make our code general. Use classes hierarchy and inheritance or use templates. Obviously both aren’t equal and have quite some differences. For instance, one property of class inheritance that gives its flexibility is that whenever you call a method from a class, this call is solved at runtime. Templatization doesn’t allow (per se) this dynamic behaviour, since all code and calls are resolved at compile time. Let me put an example in pseudo-C++ code to exemplify the basic difference between the two approaches.

Say we want to implement a search binary tree. Obviously, the tree structure and functionality doesn’t change whether it contains bools, doubles or strings. We only need for our elements to be Comparable (i.e have the "<" or ">" operator defined).

Class inheritance approach

This approach presents some performance issues and also design issues. First of all, the most important issue, is that our tree would have as type some abstract class, for example:

```c++
class BaseClass{
    virtual bool operator<(BaseClass b) = 0;
};
```

Say we define two classes, that inherit:
• MyDouble

```cpp
class MyDouble : public BaseClass {
    double value;
    ...  
    bool operator<(MyDouble d){
        ...
    }
}
```

• MyBool

```cpp
class MyBool : public BaseClass {
    bool value;
    ...  
    bool operator<(MyBool d){
        ...
    }
}
```

First problem is clear: we have to define new classes for types that are native to C++. Another issue, is that in order to compare first we would have to cast a BaseClass object into the appropriate class, and then, since the operator $<$ is overloaded, the runtime would have to navigate the class hierarchy in order to find the appropriate method to execute. But the most important problem is a conceptual problem. With this approach, we could have trees that store (theoretically, i.e. the compiler would not complain) at the same time MyDoubles and MyBools, which is certainly not intended. What we would like is for the tree to be defined with a parameter, and at compile time the compiler just replaces this name for the appropriate value, and that’s exactly what templatization offers.

**Templates approach**

Now what we will do is define our data structure as a template.

```cpp
template<typename Value>
class BST{
    class BSTElement{
        Value value;
        ...
    }
    ...  
}
```

What this allows is that now if we want to have a tree of doubles, we just need to do:

```cpp
BST<double> t = BST<double>();
```

And the compiler will generate code for the BST, substituting each appearance of Value for the template argument (in this case double). This suits perfectly our requirements because we won’t be mixing types, we don’t need special dynamic behaviour and we have an elegant way to make our BST general.
3.1.2 Overloaded operators

In C++, we can redefine the function of operators like "+", ",", "()" (function call) etc... In the SMTL we have used this functionality to make the code more "elegant" in a way. For example, for the objects that can be used as a Key in the Kd-trees, we use the operator "<" in order to do the comparisons. We could have opted to define a custom "regular" operation, such as object1.less(object2). This could have the advantage that the objects could have two orders defined, namely that of "<" and the order induced by less(), but more often than not, the ordering induced by "<" is the same that we would like to use in the Kd-tree comparisons, so we have decided to keep it together, and use the "<" operator for comparisons,

3.2 The SMTL and the iterator interface

One of the distinctive treats of the SMTL is the use of iterators. We define an iterator as a class that gives us an interface to navigate a given collection of objects. We can navigate a Kd-tree or we can navigate the result of a nearest neighbour iterator. Essentially, an iterator is an object that allows us to abstract the details of the implementation, and see a collection of elements as a list.

In the figure 3.1 we see an iterator for a breadth first traversal applied to a tree. As we can see, from the perspective of \( \text{it} \), we see the element \( e_3 \) and all the other elements as a list, bounded by and abstraction called \( \text{end} \) that represents the end of the collection. When we ask for the following element, we just ask the iterator for the next element of the list, and the iterator has the necessary knowledge as to know how to traverse the tree (in this case) to give us the next element.

If the user has any familiarity with the iterator structure of the STL, the SMTL works the same. For any "iterable" collection, we have a begin() and end() method. The iterators comply some rules:

- begin() will return an iterator pointing to the first member of the collection.
- end() is an iterator pointing past the end.
- if the collection is empty begin() == end()
- the iterators can be dereferenced via the operator *. Dereferencing an end() iterator is undefined.
- the operator "++" applied to an iterator will make the iterator point the next object in the collection.
3.3. HOW TO USE THE DATA STRUCTURES

Let’s see an example. Say we want to iterate a a Kd-tree. We only need to call begin() and iterate until the end()

```cpp
KdTree<2,BasicKey<int>, double, StdSelector>::iterator it3 = t.begin();
while(it3 != t.end()) {
    cout << (*it3).second << endl;
    ++it3;
}
```

This code will print all the members of the tree t. In this case, because we iterate a Kd-tree, when we dereference the iterator we get a pair<Key,Value>. If we were dereferencing a VP-tree or GH-tree, that don’t have elements with key, we would get the value directly.

The iterators are heavily used in the STL. It allows to define algorithms based on a ”conceptual” sequence, without having to know/deal with the details of the implementation. Also an iterator is defined by its behaviour. Anything can be considered an iterator, as long as it follows the rules defined above. Iterators are further discussed in section 4.9.

In the SMTL all iterators are what is called ”ForwardIterators”, which mean that we can only invoke go forward in the sequence (by invoking the ++ operator). It is also important to note that any iterator will become invalid upon invocation of any insertion or deletion method.

3.3 How to use the data structures

In this section I will briefly show the main functionalities of each data structure. There are some common features shared among the different structures:

- **begin() and end()** : All data structures define this two functions which return an iterator to the beginning of the collection and to the end(). Since all data structures are tree-like, the iterator traverses the collections in an inorder manner.

- **query()** : This function allows the index to be queried. It is templated by the type SearchAlgorithm and the type of query although it has some specialized functions. For more information please refer to section 3.5.

- All indices make a copy of the values they hold. This means that if we insert an item and modify afterwards, the copy stored in the data structure won’t be affected.

3.3.1 Kd-tree

In order to use the SMTL Kd-tree, we need to add the following include:

```cpp
#include "SMTL/KdTree.hpp"
```

Now we will define the Kd-tree, we will use a typedef, because the templated declaration can be very cumbersome.

```cpp
typedef KdTree<2,BasicKey<int>, string, StdSelector> kdtree;
kdtree t = kdtree();
```

The constructor creates a Kd-tree without elements. As we can see, the Kd-tree takes 3 parameters:

- An integer representing the dimension of the keys
3.3. HOW TO USE THE DATA STRUCTURES

- The **Key** type. In this case BasicKey is a class offered by the SMTL that essentially represents the keys as a vector of whatever type we want, in this case int.

- The **Value** type. This is the type of the objects that are being mapped.

- Finally, the third parameter is the Selector. I will talk a little bit more about the Kd-tree selector in section 4.3.2, but essentially this object tells us how to insert and delete items. For example, the difference between a standard Kd-tree and a squarish Kd-tree is the way in which we choose the discriminant. This object, for example, allows us to specify how the discriminant selected. In this case we use the StdSelector, which chooses the discriminant in a cyclic way.

Now we will insert an element, to do that we only need to provide the Kd-tree with the key and value we want to insert. We can also pass an iterator pointing to a suitable item.

```cpp
... vector<int> v1 = vector<int>();// vector representing the key
v1.push_back(24);
v1.push_back(33);
BasicKey<int> k1 = BasicKey<int>(v1); // BasicKey receives a vector
// representing the key
kdTree::insert(k1, "Barcelona");
```

In most cases the Kd-tree doesn’t accept duplicates. There is one exception, which is if we use the RelaxedRandomized selector, in which case there’s no guarantee that the values of the Kd-tree will be unique after insertion. For more information refer section 4.3.2. The Kd-tree provides the type of its elements through typedef as follows:

```cpp
typedef Key key_type;
typedef Value mapped_type;
typedef node::Element element_type;
```

- KdTree::key_type the type of the key
- KdTree::mapped_type the type of the values stored in the KdTree
- KdTree::element_type a synonym for std::pair<key_type, mapped_type>

If we want we can modify the an element:

```cpp
kdTree::iterator it = t.find(k1);
if(it != t.end()){
    (*it).second = "Stockholm";
} else{
    cout << "No trobat" << endl;
}
```

As we can see find() returns an iterator pointing to the element, or pointing to end() if the element is not present in the Kd-tree. In order to modify the value we only need to access the .second element (since the Kd-tree stores pairs) and change the value to the desired one.

We can also delete an element in the following way:

```cpp
... t.remove(k1); // k1 is of the type KdTree::key_type
...
3.3.2 VP-tree

In order to use the Vp-tree we need to add the following include:

```cpp
#include "SMTL/VpTree.hpp"
```

We can define a VpTree as follows:

```cpp
vector<Point> elems ....
typedef VpTree<Point, Distance, MedianSelector> VpTree;

VpTree tree = VpTree(elems);
```

As you can see now the constructor takes a parameter, which is a vector. This is the only constructor defined. The Vp-tree is defined (at the moment) as a static structure. Once it is build from a set of points it cannot be modified. In the design section I discuss this decision (section 4.5). We can still ask for a begin() iterator and we can perform nearest neighbour queries. As we can see, we have 3 template parameters:

- The first parameter (Point) is the type of the objects stored in the tree.
- The second parameter (Distance) is the metric function that will define the distance between objects. It goes without saying that it is the programmer responsibility that the function must be compatible with the first parameter type.
- The third parameter is the selector for the Vp-tree, it is discussed in section 4.5.1. This selector is used in the construction of the tree, and is responsible for the selection of the pivots.

3.3.3 Gh-tree

In order to use the Gh-tree we need to add the following include:

```cpp
#include "SMTL/GHTree.hpp"
```

We can define a GHTree as follows:

```cpp
vector<Point> elems ...
typedef GHTree<Point, Distance> ghtree;
ghtree tree = ghtree();
typedef std::vector<Point>::iterator itT;
for (itT it = elems.begin(); it != elems.end(); ++it){
    tree.insert(*it);
}
```

The Gh-tree has an insert method, as we can see. It takes an iterator pointing to a suitable element, or a suitable element directly. Although it has insert, it doesn’t have as of now delete.

3.4 Auxiliary objects

In this section we’ll get some familiarity with two important pieces of the SMTL. The objects that we will use as keys in the Kd-tree and the definition of new metrics. The SMTL doesn’t impose a given way to implement this two elements. Instead, all we do is require for some operations/definitions to be made. Any object with those properties will be considered a Key/Metric. This is the philosophy throughout the library. Not so much defining what an object must look like, but how an object must behave like.
3.4.1 How to use and define Key types

The key types used by the Kd-tree are defined as multi-component keys, therefore our keys have two parts:

- the key component: this is the elements that we will retrieve when we invoke the [] operator on the key type
- the key type (which is a multi-component key): this defines the key and makes use of the key components

The key component must have the following properties:

- Has the operator < defined. Moreover, for the correct functioning this operator should induce a totally ordering in the set formed by this type.
- Has the operator == defined. We need to be able to test for equality between keys. It is recommended that the != operator is defined as well, although we can always get inequality as not( == ), having it already defined on its own allows for cleaner code in the library.

The key type must have the following properties:

- Has the operator [] defined. We must have a way to retrieve the different key components. In the SMTL it is assumed that the key components are indexed by ints. In theory this is not required, but I think that most of the time this would be the way to do it, and it isn’t a too intrusive convention that allows for the SMTL code to be simpler and probably more familiar.
- Has the operator == defined. Then again, we could traverse the key and test equality of the different key components, but it is simpler to have it already defined in the key.

As an example, the SMTL implements the BasicKey.hpp class, which is a very simple class that stores in a vector the key components. A more extended discussion about this topic can be found in the section 4.7. The BasicKey looks like this:

```cpp
template<typename KeyComp, int Dim = 0>
class BasicKey{
private:
    int dimension;
public:
    typedef KeyComp key_comp;
    vector<KeyComp> key;
    ....
    KeyComp& operator[](int i){
        return key[i];
    }
    const KeyComp& operator[](int i) const{
        return key[i];
    }
    bool operator==(const BasicKey<KeyComp>& _hk) const{
        if(_hk.getDimension() != dimension) return false;
        typename vector<KeyComp>::const_iterator it = key.begin();
        ....
    }
};
```

Final report
typename vector<KeyComp>::const_iterator itp = _hk.key.begin();
while (it != key.end() and itp != _hk.key.end()) {
    if (*it != *itp) return false;
    ++it;
    ++itp;
} return true;
}
bool operator!=(BasicKey<KeyComp>& _hk) const{
    return !(*this == _hk);
}
int getDimension() const{
    return dimension;
}

As we can see, it is a simple class that has two templates. One which is the key component type and the dimension of the key. In this case the key components are stored in a vector, but obviously this is just an option as is not required in any case.

### 3.4.2 How to use and define Metric types

Any class that we want to use as a Metric function only needs two properties:

- having the type image_type defined, which corresponds to the type that represents the distance or similarity between two objects
- having the operator "()" defined as a function that takes two arguments of the same type and returns a value of type image_type

As an example the SMTL implements de Minkowski norm. An example of custom made Metric object for the euclidean distance could be the following:

```cpp
class Distance{
public:
    typedef double image_type;
    double operator()(const Point& x, const Point& y){
        double c1 = x.x - y.x;
        double c2 = x.y - y.y;
        double c3 = x.z - y.z;
        return sqrt(c1*c1 + c2*c2 + c3*c3);
    }
};
```

### 3.5 Querying a data structure

Any SMTL data structure has a index.query(Query q) function. This function receives a Query object as a parameter and returns an object which is responsible for searching in the index the objects that fulfill the query. To query a data structure, we will always follow the same procedure:

1. Define the query q object with the appropriate query information
2. Invoke the query(q) method of the data structure with the query q as an argument

3. The return type of query() is an object responsible for performing the search in the data structure. We can invoke begin() and end() and iterate the collection of objects fulfilling the query.

Since the Kd-tree is the one with more query variety, I will put an example with the 3 queries defined for this data structure.

**Nearest Neighbour query**  First we define the NN query object

```cpp
typedef NNSearcher<
    Minkowski<kdtree::Key_type, 2>
  > searcher;

kdtree::NNquery q = kdtree::NNquery(k8, kmin, kmax);

kdtree::searcher s = t.query<
    Minkowski<kdtree::Key_type, 2>
  >(q);

kdtree::searcher::qiterator qit = s.begin();

while (qit != s.end()) {
  cout << (*qit).second << endl;
  ++qit;
}
```

We need three parameters for the NNquery. The first parameter is always required, and the other two are only required for the Kd-tree.

- The first parameter (k8) is the object (in this case a Key) for which we want to calculate the nearest neighbour.
- The second and third parameter define a bounding box that contains all elements of the Kd-tree. For the Vp-tree and Gh-tree this are not necessary. For more information refer section 4.10.1.

In the case of the Kd-tree, the query is templated by the metric function, in this case the Euclidean distance (Minkowski, 2). For the VP and GH trees, this is not the case, since the trees already have a metric. The query call returns a NN searcher object, which is responsible for performing the search. The only thing left is to ask for the begin() iterator and iterate the results of the query. In this case the nearest neighbour query is an incremental query, therefore if we iterate until the end, we will get all elements ordered by proximity to the query object (k8 in this case).

**Range query**  The range query is Kd-tree exclusive and as we will see, using the different queries is rather similar and intuitive. The range query retrieves all elements that fall in the interval defined by the user. For the range query we must also first define the query and then just iterate the results calculated by the Searching object.

```cpp
kdtree::Rangequery qr = kdtree::Rangequery(min, max);

kdtree::RangeSearcher sr = t.query(qr);

kdtree::RangeSearcher::qiterator qrit = sr.begin();

while (qrit != sr.end()) {
  cout << (*qrit).second << endl;
  ++qrit;
}
```

Final report
Here the query receives two parameters. The first one is the lower bound of the search and the second one is the upper bound of the search.

**Partial Matching query**  The partial match query is Kd-tree exclusive, like the Range query, and it retrieves all keys which match a subset of the key, defined by the user.

```cpp
... 
BasicKey<int> q = {0,33}
vector<bool> b = {false, true}

kdtree::PMatchSearcher qpm = kdtree::PMatchSearcher(q, b);
kdtree::PMatchSearcher sr = t.query(qpm);
kdtree::PMatchSearcher::qiterator qrit = sr.begin();
while (qrit != sr.end()) {
    cout << (*qrit).second << endl;
    ++qrit;
}
```

First we define the partial key we want to match. The only values that will matter are those in positions set to true by the second parameter of the query. This second parameter is just a vector of booleans that indicates which key components should be taken into account for the query. Once the query is defined, the rest of the process is exactly the same as for the other queries.
Chapter 4

SMTL Design

In this chapter we will take a closer look to the internal workings of the SMTL. This chapter should be used to obtain a deeper understanding of the rationale behind some decisions made in the SMTL and to be able to extend the library.

4.1 SMTL components

The SMTL is thought as a set of components that can work together in solving different problems and also as an incomplete set of tools, with scalability as a goal to reach. In the SMTL we can identify a few different types of components that make the core of the library:

- Data Structures: In general a data structure is a construct that allows for efficient and easy information retrieval. Different data have different properties, so the data structures are characterized by the type of information they contain. In the SMTL the data structures aim at dealing with spatial and metric data.

- Queries: Although being able to store and retrieve data is an interesting functionality, we also want to be able to ask for certain subsets of data that fulfill some properties, such as being in a given range or being in a given radius from another. The SMTL offers some already defined queries and also offers the mechanisms required to add whatever queries are deemed necessary.

- Searching algorithms: In order to execute the queries we need algorithms able to explore the data structure and report the correct elements. In the SMTL there are a couple of these implemented, that can be reused for other queries.

- Iterators: One of the distinctive features of the SMTL is the iterator based interfacing, inspired by the STL. Iterators allows us to abstract the nitty-gritty details of the implementation and just focus in the use of the elements, in a completely transparent way for the programmer.

4.2 General data structure traits

All the data structures implemented so far in the SMTL are trees. We have a root node which has some sub-trees as children and we proceed recursively. This is by no means an imposed rule, that is, the SMTL doesn’t enforce its data structures to be hierarchical (trees). But because this is a fact as of now, I will dedicate this section to comment on some common features of the three data structures implemented.
4.3. KD-TREE

4.2.1 Size of the tree

When redesigning the SMTL I was faced with a decision. Where to put the size of the data structure. One option is to have the size as a global attribute, that is, we only have the number of all elements in the collection and if we want to know the size of any sub-tree we must traverse it in order to come up with this number. The other option is for any (sub)tree to know its size. Then the size of any given tree (including the one which corresponds to the root and hence to the whole data structure) is

$$size(T) = 1 + \sum \text{size}(T_i)$$

This implies some things

1. The data structure takes a $O(n)$ extra space
2. Now we can retrieve the size of a given sub-tree in $O(1)$ time
3. This implies that if the data structure is dynamic (like the Kd-tree) the process of adding and removing elements can be a little bit more complicated

This decision was made because I feel that knowing the size of each sub-tree in constant time overcomes the disadvantages. Queries such as given a key component $k_i$ knowing how many keys have $k_i < k'_i$ can be useful. Obviously this is a design decision, and if through the use of the SMTL and empirical evidence we would find that it turns out to not be that useful it could and should be modified. Also, this feature allows us to implement efficiently the Relaxed randomized Kd-trees.

4.3 Kd-tree

We define a Kd-tree $T$ following the formal definition of the data structure:

- $T$ can be empty, in which case in the SMTL it is represented by a null pointer.
- $T$ is represented by a node that has a pair (key_type k, value_type v) a discriminant $i$ and two pointers $T^-$ and $T^+$. $T^-$ is defined as the Kd-tree containing the points $(k', v')$ such that $k'[i] < k[i]$. $T^+$ is defined as the Kd-tree containing the points $(k', v')$ such that $k'[i] \geq k[i]$.

The main difference between the different Kd-tree variations is the discriminant selection. There is an exception, which is the Relaxed randomized Kd-tree. This Kd-tree also differs in the way the elements are inserted and deleted.

4.3.1 Deleting elements from a Kd-tree

When we delete an item from a Kd-tree as a general rule, what happens is that all the children are reinserted. This approach is not the most efficient, but is required for Kd-trees like the Squarish Kd-tree. Because of this, I have decided to keep it more or less homogeneous and not add a lot of variations for each Kd-tree. The exception, obviously, is the Relaxed randomized Kd-tree that required special methods to do the deletion (and insertion). As we have seen in previous sections (section 3.3.1) the definition of a Kd-tree requires of 3 parameters
4.3. KD-TREE

- An integer representing the dimension of the keys
- The Key type
- The Value type
- The Selector

The selector is an object that is responsible to encapsulate the differences in the Kd-trees variants. The first idea to encode the different variants was to program template specializations. Template specializations are a way to define specific code as a function of the parameters. For example

```cpp
template<typename Value>
class Foo{
    void print() {
        cout << "non specialized" << endl;
    }
}

class Foo<Double> {
    void print() {
        cout << "specialized double" << endl;
    }
}
```

As we can see, we can redefine the whole class for each possible template argument. The director’s project suggested to encapsulate the different behaviours in an external object, and I really think is a better approach since

1. We don’t duplicate code. Right now we have 4 specializations, we would have to redefine the whole Kd-tree 4 times in order to modify the insert and deletion
2. As a direct consequence, it is easier to extend the SMTL, because to add a new Kd-tree variation we only need define a new selector
3. We avoid redundancy, which is error prone

4.3.2 Selector object

The selector objects can be found in the file "SMTL/Selectors.hpp". It is parametrized by Key and Dimension, because the selector must be usable in any possible specialization of the Kd-tree. It has the following methods:

- `NextOp next(const Key& key, const Key& nodeKey, int size, int i)` where key is the Key to be inserted, nodeKey is the Key of the current node we are exploring, size is the size of the current tree and i is the current discriminant
- `int calculateDiscriminant()` : this functions returns the discriminant that should be used when generating a new node, according to the state of the Selector object
- `bool deleteRoot()` : Return true if when deleting we should delete at the root or reinsert the children. Return false for all selectors except the Relaxed randomized selector.
The function next returns a NextOp which is a value in the set \{ \text{LEFT}, \text{RIGHT}, \text{DUPLICATE}, \text{INSERT} \}.

- \text{LEFT} : Is used to indicate that we should proceed to the left sub-tree
- \text{RIGHT} : Is used to indicate that we should proceed to the right sub-tree
- \text{DUPLICATE} : This is a special case, that indicates that the key is equal to nodeKey and therefore we should ignore it
- \text{INSERT} : This is used to indicate that the new value should be inserted at the current root. As of right now this is only used for the \text{Relaxed randomized selector}.

### 4.4 Common features of the VP tree and GH tree

We say a data structure is dynamic when we can modify its elements, that is, it implements the following functionality

- \text{insert} : allows for elements to be added to the collection after construction
- \text{delete} : allows for elements to be eliminated from the collection

The Kd-tree, therefore is dynamic. Both the VP tree and the GH tree will be considered static (although the GH tree implements the insert functionality) because neither of them allows for deletions. As of the current SMTL version both metric data structures implemented (VP & GH trees) don’t support find or deletion. This is a design decision and not an inherent property of this data structures. The main reason is that both VP and GH trees don’t have a ”good” solution to the deletion problem. There were some methods proposed in discussions with the project director, such as marking the deleted nodes and then ignore them. Then we would define a ratio $\alpha = \frac{\text{#elementsmarked}}{\text{#totalelements}}$ and rebuild the whole tree with the unmarked objects whenever this ratio went higher than a user-defined limit. This is a worthy route to be explored, but as of the first release this isn’t implemented yet. Another common feature is that the find functionality is not supported. The reason for this is because as of right now, both metric structures store elements, not pairs \(<\text{Key}, \text{Value}>\). Therefore, in order to look for an item, one would already need to have the object. Granted, the find functionality could still be interesting if we want to test if an item belong to the collection or not, but this can also be achieved using a nearest neighbour query and seeing if the closest object is at distance 0 or really close to 0. Then again, this should be reconsidered if the SMTL users find it interesting to have a find functionality for this structures.

### 4.5 Vp-tree

We define a Vp-tree following the formal definition, let $d(x,y)$ be the metric function used in the Vp-tree, then:

- If $T$ is empty it is represented by null pointer
- a $T$ is represented by a node. This node has an element $p$ and a radius $r$. It has two children $T^-$ and $T^+$. All elements $x$ in $T^-$ are such that $d(p,x) \leq r$ and all elements $x'$ in $T^+$ are such that $d(p,x') > r$

The class constructor receives a vector with the elements and proceeds into the construction of the tree. This data structure allows for nearest neighbour queries.
4.5.1 Selector object

When building the Vp-tree, we need a method to select the pivoting element. Once the pivoting element is chosen the procedure is the same always.

- Pick a pivot \( p \)
- Calculate for all elements \( x \), \( d(p, x) \)
- Pick a distance \( r \) and build recursively the children with the appropriate elements

The selector object is responsible for picking both the pivot \( p \) and the radius \( r \). A selector must implement the following functionality:

- \texttt{process(vector Elements)} : This function is responsible to update the information in the selector object to make sure the following functions return a correct value
- \texttt{Element getPivot()} : returns the pivot to be used after the elements have been processed
- \texttt{Metric::image_type getRadius()} : returns the radius to be used after the elements have been processed
- \texttt{vector Elements getSmaller()} : returns the elements that will go into \( T^- \)
- \texttt{vector Elements getGreater()} : returns the elements that will go into \( T^+ \)

Right now there is a median selector implemented which can be found in the ”SMTL/VpS-electors.hpp” file. This selector at each step picks a random pivot and then defines the radius so as to be the median value of all \( d(p, x) \).

4.6 Gh-tree

The Gh-tree implementation follows the formal definition but with some changes. A node of the Gh-tree has (besides an element of the collection) three pointers:

- Pointers to \( T^- , T^+ \)
- Pointer to another node called \textit{siblings}

Because each node requires two elements, I have decided that instead of having nodes holding two variables, having nodes holding just one element and having an extra pointer. This allows for performing operations in the Gh-tree in a more cleaner way because we don’t need to define a default value for the elements or deal with uninitialized elements.

- An empty node \( T \) is represented with a null pointer
- A node \( T \) can present 2 cases
  1. Having only one pivot defined: Then the node \( T \) must be a leaf node. It has an element and null pointers to \( T^- , T^+ \) and \textit{siblings}
  2. Having the two pivots defined: Then the node \( T \) has an element and a pointer to a GHNode in \textit{siblings}. Pointers to \( T^- , T^+ \) may or may not be null.
4.7 Key’s design

In the context of the SMTL when we talk about a key we refer to a multidimensional vector whose components have a total order defined. That is if $k_i \in X$ where $X$ is the domain of the key component $k_i$, the following properties hold:

- $\forall x \in X \leq x$
- $\forall x, y \in X \leq y \land y \leq x \Rightarrow x = y$
- $\forall x, y, z \in X \leq y \land y \leq z \Rightarrow x \leq z$
- $\forall x, y \in X \leq y \lor y \leq x$

In the SMTL there is no base class defined from which any key must inherit. Instead, we take the template approach and just assume that the operators < and [] are defined, and that the previous properties hold. The main rationale behind this season is that having a base class, would impose too heavy a restriction in my opinion. It would force people to inherit from that class in order to be able to make use of those two operators mentioned before. Besides the incremented cost in running time (due to the runtime search through the class hierarchy when calling a method) we also force an unnecessary layer of complexity to the class design.

4.8 Metric’s design

The SMTL does not impose any rule in what a metric should be. It is only asked to abide the following rules:

- Have the operator () defined as a function $d : C \times C \rightarrow I$
- $d(x, y) > 0$
- $d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq d(x, y) + d(y, z)$ (triangular inequality)
Right now there is the Minkowski norm implemented in the SMTL. It can be found in the file "SMTL/minkowski.hpp". It is an adaptation of the same class found in [Cre10]. The Minkowski norm $L_k$ is defined as follows:

Given a vector $v$ of dimension $d$ and an integer $k \geq 1$

$$L_k = \|v\|_k = \left( \sum_{i=1}^{d} |v_i|^k \right)^{\frac{1}{k}}$$

### 4.9 Iterators

In the SMTL all iterators can be defined as forward iterators. This means that if $it$ is an iterator we can

- $\text{++it}$ : will advance the iterator one position
- $\text{*it}$ : allows access to the object being pointed
- $\text{it == it'}$ : we can compare two iterators to determine if they point the same object

Equality of iterators is defined as pointing to the same node.

The SMTL implements forward iterators (as defined by the STL). This means that they implement the operations defined before. Also, all SMTL iterators inherit from the std::iterator class, specialized as a forward iterator.

There are two types of forward iterators, mutable and non-mutable. Essentially, if the iterator is mutable if we do something like

```cpp
... *it = <value>
...```

We will modify the object to which *it points. If is non-mutable it will have no effect and the object pointed will remain unmodified.

In the SMTL all iterators are mutable, except for the QueryIterators.

### 4.10 Queries

In order to execute a query we have two essential parts

- The Query parameters, e.g in the range query we have to define in which range we want to find items
- The Query "engine", the piece of software responsible for exploring the data structure and report the appropriate elements

In the SMTL there is a class for each part. In the file "SMTL/Queries.hpp" there is the definition for the 3 types of queries currently supported. This classes hold the required information regarding the parameters of the query and may have additional functionality. For example, we can ask the range query object if a given Key is within its range or not.

The query "engine" is essentially the searching algorithm that computes the different queries. It consists of a data structure that stores the search nodes (a priority queue in the NN search, a stack in the other queries). Essentially it defines two methods:
• initFirst : this method initializes the search, doing the necessary processing. As of now it is only defined because most times the very first item pushed into the data structure requires special treatment

• processNext : this method receives a search element, processes it and adds the corresponding elements to the data structure

As explained in the previous section of the design (section 4.9) the query iterator is a templatized class used by all the searcher objects. Therefore, the search objects are subject to some extra conditions, namely they must implement the following methods:

• getLast : This method returns the last query compliant element of the search

• processNext : this method calculate the next query compliant element

As of right now each class define their own searchers, although they share some common features. This is discussed further in the section 4.11

4.10.1 Nearest neighbour query

This query stores 3 values

• The point to which we want to find the nearest neighbours

• The smallest possible key, and the greatest possible key (this is only used in the Kd-tree Nearest neighbour query)

In this case the query object is only used as a container for the query parameters. Regarding the second point, the smallest and greatest key, this is done so we can define a bounding box that includes all elements of the Kd-tree at the start of the search. A more sensible approach would be to have the query ”engine” define this on their own, but I have decided not to, because of the way the standard library defines this maximums. If our key is formed by, say, integers, then the theoretically the smallest and greatest keys are

\[ k_{\text{min}} = (-\infty, \ldots, -\infty) \text{ and } k_{\text{max}} = (\infty, \ldots, \infty) \]

This could be represented with a boolean vector that indicates whether the position has a "real" value or represents \( \pm \infty \). Since a computer ends up discretizing its values there is a maximum integer value and a minimum integer value. I have choosen this approach because I feel it is less cumbersome and we are only sacrificing two values (the top value and the bottom value). The problem is that C++ doesn’t have a unified method for retrieving this value. For example for the integer type we can retrieve the minimum and maximum values with `std::numeric_limits<int>::min()` and `std::numeric_limits<int>::max()`. As we can see, is not a property, or structure lying within the own type integer. This means that if we wanted to use a custom type we would have to add a specialization to the limits template. I have opted for a more pragmatic approach and ask the user to submit this types into the query ”engine” via the query object. In the future it would be interesting to find a more elegant and transparent way to deal with this.

4.10.2 Range query

This query have two main functionalities

• Storing the lower and upper bound of the range

• Providing a matching procedure that can tell us if a given object is within the query range
4.10.3 Partial match query

This query have three main functionalities

- Storing the partial match object
- Indicating which key components of the key are relevant for the query
- Providing a matching procedure that can tell us if a given object is a partial match of the query value

4.11 Searching algorithms

This algorithms are the responsibles for performing the search in the data structures. First I will give a general overview of the searching system, and then I will comment on the specifics of the searching algorithms used in the SMTL.

4.11.1 Searching system

The searching system can be broken down into 4 classes

- Driver: this class is responsible to "drive" the search. It will retrieve an element from the container, process it, obtain the successor elements and push them into the container.

- Container: just a class that stores the search elements, e.g. in the nearest neighbour query it is a priority queue.

- Element: the class of the elements that will be pushed into the container, often will be a wrapper for a pointer to a node of the data structure, together with some extra information required for the search.

- Brancher: this class is responsible to generate the elements that will go into the container. It is called brancher since we are dealing with hierarchical structures (for the moment) and so if we decide to ignore a certain element all the sub-tree rooted at that element will be ignored.

![Diagram of search system classes](image)

Figure 4.2: Classes involved in performing searches on the data structures

In the current implementation of the SMTL, the driver class and the brancher class are merged into one. We can identify the driver class as the "processNext" method of
the searcher objects and the brancher class as the "processElement" method. Because this searches aren’t quite complex, I have decided to merge them, but should we want to implement more complex searches it might be worth to reconsider this decision.

4.11.2 Hierarchical Ranking Search (HRS)

This algorithm can be found in [Sam06]. It was already implemented in the SMTL 1.0 and it can also be found in [Cre10]. This algorithm is based on two things

- A data structure that keeps objects ordered according to some score (e.g. a priority queue)
- Having a way to retrieve an estimate of the best score that can be found in a tree

This algorithm is primarily thought to be used to compute the nearest neighbour query but it could evaluate some other ranking, if we can find a suitable function for the second point. The algorithm works in the following way

Let \( e_0 \) be a type associated to nodes of the explored hierarchy and \( e_1 \) types that represent subtrees or subhierarchies.

![Figure 4.3: Hierarchy search induced by the algorithm](image)

**Algorithm 1: HRS algorithm extracted from [HS00]**

```plaintext
1 Function HRS(q, S, T) is
2     Q := PriorityQueue()
3     e := root of the hierarchy induced by q, S and T
4     push(Q, e, 0)
5     while Q ≠ do
6         e_t := top(Q)
7         pop(Q)
8         if t = 0 then /* e_t is an element */
9             report e_t as the next element
10            else
11                push(Q, e_0, d(p, e_0))
12                for ∀c ∈ children(e_t) do
13                    push(Q, c, d(p, c))
14            end
```

Final report

CHAPTER 4. SMTL DESIGN
It can be shown [HS00] that this algorithm visits the same elements of the hierarchy as would a depth first approach. Therefore, while we may pay a bit extra overhead due to the priority queue and defining the dealing with the two types of elements, we get a lot of flexibility, because we can calculate nearest neighbours at demand, without having to know at the start of the search how many we want to report.

Variants

The very interesting thing of this algorithm, besides the incremental approach, is the fact that with one algorithm we cover many possible queries. For instance:

- Ball query: Given a radius $r$ and the center of the ball $p$, we just need to perform nearest neighbour search with respect to $p$ and keep extracting neighbours up until we find one at distance greater than $r$. We can also do the complementary, reporting all objects at distance greater than $r$, we just iterate and ignore all elements, until we have find one at distance $\geq r$. Then we know that if we keep iterating we will find query compliant elements.

- K-nearest neighbor: This is a common query and it’s quite straight forward as to how to do it. The most straightforward way to do it, would be to just report elements up until we have found $k$ elements. Another way, that involves tweaking the implementation, is to have a second priority queue limited to $k$ real elements, ordered in decreasing order of distance to the query point. Whenever we want to push the $k + 1$ element into the primary queue we do it only if the distance to the query point is smaller than the distance of the element at the top of the second priority queue. This allows us to ignore sub-trees of the hierarchy and thus pruning the search.

- Farthest element: We can also compute the farthest element if we have a suitable distance function that gives us an upper bound on the distance to the query point. Then we only need to change the ordering of the queue or change the sign of the distance.

4.11.3 Depth/Breadth First Search (DFS/BFS)

The range and partial match queries of the Kd-tree are implemented with a Depth First search. The algorithm goes as follows

1. Pick an element $e$ from the stack/queue. If $e$ complies the query report it

2. For each children of $e$ decide whether it is possible to find further elements in that subhierarchy

This is a common algorithm and I won’t comment much more. As of right now is implemented with a stack, therefore yielding a Depth First Search (DFS). Only swapping the stack for a queue we would get a Breadth First Search (BFS). It is direct, that these type of algorithm gives support for a wide variety of pruning search based queries, for example, we could implement more complicate range queries, where the query object is not axis-aligned, it is say a convex polygon in 2D. Then we could test if it intersects the hyperrectangle at each subtree and prune based off that.
Chapter 5

Project planning

5.1 Tasks & Temporal planning

5.1.1 Familiarization with the SMTL 1.0

The first task set was to get familiar with the SMTL 1.0. I did so by reading the final report, and fiddling about with the code. The initial plan was to reuse de SMTL 1.0 code and expand it. Unfortunately, I couldn’t reuse the code due to technical problems and I had to reimplement the SMTL 1.0. This caused a deviation in the initial planning. For this task having the original SMTL design was very helpful because it allowed me to read the rationale behind some decisions made in the code, and allowed me to get a better understanding of the library.

5.1.2 Data Structures

Initially it was planned to first implement the data structures. The idea was that essentially what makes the library are the data structures and the operations we can perform on them, but obviously in order to do that we first need to have the data structures implemented. As it turned out, the whole development was made at the same time. What I mean is that while I was implementing and designing the data structures, I was already designing and implementing part of the algorithms and the query result retrieval system. Regarding the variety of data structures, I could fulfill the planification and all data structures that were planned were implemented. The list is the following:

Spatial data structures
- Standard Kd-tree
- Relaxed Kd-tree
- Squarish Kd-tree

Metric data structures
- VP tree
- Generalized Hyperplane Trees (GHT)

Actually, the director suggested a variation on the relaxed kd-trees, that has also been implemented.
5.1.3 Algoritms & query result retrieval system

In the initial planification I didn’t define any concrete algorithms to be implemented. At the end I reimplemented those on the SMTL 1.0 because I couldn’t reuse the code from the SMTL 1.0. Therefore, the final algorithms supported by the STML 2.0 are the following:

- Find (Kd-tree exclusive) : Given a key, find and report the value associated with the given key
- Range query (Kd-tree exclusive) : Given a range of values for each key component, finding all keys and their associate value that comply.
- Partial match (Kd-tree exclusive) : A little variation of the ”normal” search. We will look for a match of a subset of the key components.
- Nearest neighbour : Given an element, report its nearest neighbour(s) ordered by distance of proximity
- Ball query : Given a key k, a radius r and a metric function f, find all points x s.t. \( f(x, k) \leq r \)

5.1.4 Performance evaluation

As a final task, we wanted to retrieve performance data to assess the quality of our work. We wanted to answer questions such as ”Given input of type X, which data structure works better”, or ”which algorithm works better in average”, etc... Unfortunately I haven’t been able to carry out this task due to lack of time.

5.1.5 Documentation

The code has been documented while developing, although it has needed some refinement at the end. The final report has been partially done with the GEP reports.
5.2 Time scheduling

This was the initial scheduling. This was, and is, a rough estimate since I haven’t kept track of the hours spent developing the SMTL 2.0.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Time in hours</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project planning</strong></td>
<td>77</td>
</tr>
<tr>
<td>Scope of the project</td>
<td>15</td>
</tr>
<tr>
<td>Temporal planning</td>
<td>12</td>
</tr>
<tr>
<td>Economic management and sustainability</td>
<td>15</td>
</tr>
<tr>
<td>First presentation</td>
<td>10</td>
</tr>
<tr>
<td>Contextualization and bibliography</td>
<td>25</td>
</tr>
<tr>
<td><strong>Project development</strong></td>
<td>337</td>
</tr>
<tr>
<td>Implement data structures</td>
<td>20</td>
</tr>
<tr>
<td>Validate data structures</td>
<td>20</td>
</tr>
<tr>
<td>Research &amp; discuss algorithms</td>
<td>12</td>
</tr>
<tr>
<td>Implement algorithms</td>
<td>60</td>
</tr>
<tr>
<td>Validate algorithms</td>
<td>35</td>
</tr>
<tr>
<td>Design query result system</td>
<td>25</td>
</tr>
<tr>
<td>Implement query result system</td>
<td>40</td>
</tr>
<tr>
<td>Validate query result system</td>
<td>40</td>
</tr>
<tr>
<td>Design performance tests</td>
<td>15</td>
</tr>
<tr>
<td>Implement performance tests</td>
<td>30</td>
</tr>
<tr>
<td>Assess library performance</td>
<td>40</td>
</tr>
<tr>
<td><strong>Final tasks</strong></td>
<td>40</td>
</tr>
<tr>
<td>Documentation</td>
<td>40</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>454</strong></td>
</tr>
</tbody>
</table>

Table 5.1: Timetable detailing the tasks and their expected time requirement in hours as in the initial planning
5.3. RESOURCES

The resources don’t differ from the ones planned.

5.3.1 Hardware

Personal Computer, Specifications:

- Intel core i5-4210U
- 6GB Ram
- Nvidia Geforce 820M

5.3.2 Software

- Linux: We’ll use Linux as our OS in which we will develop the SMTL 2.0
- GCC: Our C++ compiler will be the gnu compiler GCC, using the C98 standard
- Rstudio: In order to manipulate the performance data and generate performance report (finally not used)
- \LaTeX: For all the documentation
5.4 Budget

The budget is divided in three sections:

- **Hardware resources**
- **Software resources**
- **Human resources**

As it is explained in the following paragraphs, in this project, the bulk of the budget has been put to human resources, since we don’t need much hardware and all of our software is free.

### 5.4.1 Hardware resources

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Useful life</th>
<th>Amortisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personal computer</td>
<td>700 €</td>
<td>4</td>
<td>87.5 €</td>
</tr>
</tbody>
</table>

Table 5.2: Hardware resources budget

### 5.4.2 Software resources

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Useful life</th>
<th>Amortisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Mint</td>
<td>0 €</td>
<td>3</td>
<td>0 €</td>
</tr>
<tr>
<td>GCC compiler</td>
<td>0 €</td>
<td>3</td>
<td>0 €</td>
</tr>
<tr>
<td>Rstudio</td>
<td>0 €</td>
<td>3</td>
<td>0 €</td>
</tr>
<tr>
<td>LATEX</td>
<td>0 €</td>
<td>3</td>
<td>0 €</td>
</tr>
</tbody>
</table>

Table 5.3: Software resources budget

### 5.4.3 Human resources

In the following table we can find the budget allocated for human resources. Due to the changes of the previsions, this budget was modified in the follow-up report. We can identify two clear separate parts of our project. First, we have the planning stage, which has been carried out entirely by the Project manager. The software designer, programmer and tester have carried out the other phases stated in the planning. The designer and

<table>
<thead>
<tr>
<th>Role</th>
<th>Price per hour</th>
<th>Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project manager</td>
<td>50 €</td>
<td>75</td>
<td>3750 €</td>
</tr>
<tr>
<td>Software designer</td>
<td>35 €</td>
<td>115</td>
<td>4025 €</td>
</tr>
<tr>
<td>Software programmer</td>
<td>25 €</td>
<td>160</td>
<td>4000 €</td>
</tr>
<tr>
<td>Software tester</td>
<td>20 €</td>
<td>104</td>
<td>2080 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>13855 €</strong></td>
</tr>
</tbody>
</table>

Table 5.4: Human resources budget as initially planned
the programmer are responsible for designing and implementing the data structures and algorithms planned and the software tester has performed the required tests. As seen in the following table, we added more hours to the software programmers.

<table>
<thead>
<tr>
<th>Role</th>
<th>Price per hour</th>
<th>Hours</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project manager</td>
<td>50 €</td>
<td>75</td>
<td>3750 €</td>
</tr>
<tr>
<td>Software designer</td>
<td>35 €</td>
<td>115</td>
<td>4025 €</td>
</tr>
<tr>
<td>Software programmer</td>
<td>25 €</td>
<td>190 (160+30)</td>
<td>4750 €</td>
</tr>
<tr>
<td>Software tester</td>
<td>20 €</td>
<td>104</td>
<td>2080 €</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>484</td>
<td>14605 €</td>
</tr>
<tr>
<td>Budget difference</td>
<td></td>
<td>+30</td>
<td>+750 €</td>
</tr>
</tbody>
</table>

Table 5.5: New budget for the human resources budget

5.4.4 Total budget cost

Putting together the information from previous sections, we get the following cost for our project:

<table>
<thead>
<tr>
<th>Concept</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware resources</td>
<td>87.5 €</td>
</tr>
<tr>
<td>Software resources</td>
<td>0 €</td>
</tr>
<tr>
<td>Human resources</td>
<td>14605 €</td>
</tr>
<tr>
<td>Total</td>
<td>14692.5 €</td>
</tr>
</tbody>
</table>

Table 5.6: Total budget

5.5 Budget control

In this project, the only budget problem we have found has been in the human resources department. As the planning stated in the section Temporal planning wasn’t accurate, we required more hours, and so we needed more money. We already identified that the parts that required more hours than planned have been those related to the Software designer and programmer, since the design and implementation of the library are crucial tasks. Testing and assessing the quality of the library is important, but we obviously prioritized having a working and functional library and cut some hours of quality assessment by the software tester.

5.6 Project Sustainability

5.6.1 Economic Sustainability

Planification

This project doesn’t require maintenance, per se. Although this project is intended to be expanded in the future, we don’t think that should be considered maintenance, but just an extension of what we have.

It is hard to say whether this project could be accomplished with a lower cost. Because since it is not a closed project (in the sense that it is intended to be expanded in the future)
it is tricky (as we have seen in the scope section) to set the scope of the project. What is definitely true is that we could trade features for time and money, and the other way around. Also the budget is almost exclusively spent in human resources, so it seems rather difficult to do the same with less money.

This project could be competitive money wise. It could be offered as a one time paying license, but because this library is somewhat general, and not very specific highly optimized, most likely we would find opensource competitors developed by the community. Therefore, should we want this project to be economically viable, we would have to either target a very specific market or spend a lot of time in optimizing it and making sure that it’s the best.

Because the idea is to be an opensource project, this library is intended to be shared with the community and in exchange of free use get the community’s input to improve the library.

Most of the project’s time will be devoted to the implementaion of data structures and algorithms.

This project is going to be awarded an 8 in the economical sustainability because the cost is as close as it can get to raw human power, and we don’t have any extra expenses.

Results

As seen in the results, we had a deviation in the budget. This budget deviation represents about a 5.4%, and in absolute terms 750 €.

The economic sustainability of the results in this project are going to be awarded a 7.

Risks

I consider that these section should be graded a 0, since although we had trouble, we had already planned for schedule deviations.

5.6.2 Social Sustainability

Planification

Obviously this project is directed to countries that are relatively advanced in technology regards. The library is intended to be used by industry and academia alike. This project aims to improve how people deal with metric and spatial data. This type of data arises in many fields, such as medicine, engineering, physics, etc... therefore this project will help to solve problems arising in those fields. To put an example, it may speed up the retrieval and the storage of medical profiles, therefore translating into a better medical assistance for the people. We don’t think this project will affect negatively any sector.

This project is going to be awarded a 5 in Social sustainability. Although it has potential to improve the quality of life, it will not do it by itself, and could only help other projects to do so.

Results

There social dimension hasn’t changed in the course of the project, therefore this section is going to be awarded a 10.
Risks
We haven’t encountered any risks (expected or not), regarding the social sustainability, therefore I am going to award a 0.

5.6.3 Environmental Sustainability

Planification
In this project, the main resource we will be using is the electricity needed to power the computer. Since the project is estimated to take 454 hours, and assuming a 300W average potency needed to run our computer, we need approximately 136 kWh, which translates roughly to 52.36 kg of CO$_2$. This energy is totally necessary and cannot be reduced with the hardware that is planned to be used.

Since this project is a continuation of an existing one, we will in fact be reusing code, and making extensive use of the developments already done in this field.

This project is going to be awarded a 9 in environmental sustainability, since its environmental impact is only the energy consumed by the computer, which is almost negligible, compared to what the average person consumes daily.

Results
Although we couldn’t directly reuse the code, we reused the design, so I am going to award a 9 to the results section.

Risks
I don’t think that we have encountered any environmental risk, therefore I will award a 0 to the risks section.

5.6.4 Sustainability Matrix

<table>
<thead>
<tr>
<th></th>
<th>Economical</th>
<th>Social</th>
<th>Environmental</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planification</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>Results</td>
<td>7</td>
<td>10</td>
<td>9</td>
<td>26</td>
</tr>
<tr>
<td>Risks</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>15</td>
<td>15</td>
<td>18</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 5.7: Sustainability matrix
Chapter 6

Conclusions

6.1 Final result

We have for the most part completed the objectives set in the planification. We have a functional library, which is general and allows to use data structures holding any type of element as long as it is suitable for that structure (e.g having a metric defined for the Vp-tree). All data structures we planned have been implemented, and although we only reimplemented the algorithms featured in the SMTL 1.0 (we didn’t add new algorithms), we do provide the design for a framework that allow general queries in hierarchical objects to be done in an incremental way. Unfortunately, we couldn’t run performance tests to see how our library behaves with different structures and data sets. This project is a good approach to the design and implementation of data structures. I’ve had the opportunity to learn the theoretical designs and translate it into a functional implementation.

Contrary to what I expected, the most difficult part has not been so much the intricacies of the data structure, but the design of the implementation and giving it a certain sense of coherence between the different pieces, as well as devising mechanisms to have a library as general as possible.

6.2 Future work

This library would be just but an initial stepping stone for the SMTL. Besides being just a library we would like it to be a pack of tools to use, implement, design, benchmark, etc... spatial and metric data structures. Some of the next goals we could set

- Benchmarking the current implementation. Since we couldn’t really run this benchmarks in the context of this TFG, it would be interesting to see how the SMTL behaves.

- Expand functionality. For example, we could look into having different types of data structures based on the functionality. This way we could have a "pure" Vp-tree that is static, and a "pseudo-dynamic" VP-tree which has delete functionality (although it would rebuild the entire tree) etc...

- Design a visualization tool for the data structure. This would give us two main benefits. Firstly, we could get an idea of how some data structures divide data or how algorithms on data structures work, for instance, we could visualize a nearest neighbour query execution and get an intuition of how it behaves. The other benefit,
which is a consequence of the first is that then the SMTL would become a didactic tool, used to teach and learn about spatial and metric data structures.

- It could be interesting to also develop a set of tools to aid the design and implementation of the data structures, for example, having programs that allow the easy generation of invariants and conditions on the data structure, to be later feed to some of the already existing verification tools to help in making sure that the data structures comply with their theoretical design.
Appendices
Appendix A

Expanding the SMTL

When one wants to expand the functionality of the SMTL some guidelines as to how to proceed might be appreciated. In any case, there are no set rules that define what or what not can be an SMTL component, but it is interesting to keep somewhat of an internal cohesion between all the SMTL elements, the ones already implemented and the ones that might be implemented in the future. Any data structure should have the following components

- begin and end iterators (at least forward iterators) that allow traversal of the structure
- templatized by the type of elements that will hold
- A method query() overloaded by the type of query

When defining new queries it would be interesting to follow the separation explained in section 4.11. Even if we collapse one or two classes, as is done right now, the parts defined should be identifiable. As a final remark, the already implemented elements can be used as a guideline to define new elements. The key ideas to follow is use iterators as interfaces between collections to abstract the implementation details and templatization to provide the required generality.
Appendix B

Installation & Documentation

B.1 Installation

In the delivered file there is a folder named "SMTL". Inside there are all files that compose the SMTL. In order to use the SMTL in our applications, we only need to copy this folder wherever we want and include it in the compilation path. It will be required a C++ compiler compatible with the C++98 standard. The SMTL has been developed with the GCC 4.8.2 version provided under the ubuntu environment.

B.2 Documentation

Besides this document there is a reference manual in HTML format under the Documentation/html folder. It was generated by doxygen and allows for quick consultation of the methods and attributes of the SMTL classes.
Bibliography


