Bruno Filipe Fernandes Simões Salgueiro Faustino

Licenciado em Engenharia Informática

Implementation for Spatial Data of the Shared Nearest Neighbour with Metric Data Structures

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Doutor João Carlos Gomes Moura Pires,
Professor Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente: Professora Doutora Ana Maria Dinis Moreira

Arguente: Professora Doutora Maribel Yasmina Campos Alves Santos

Vogal: Professor Doutor João Carlos Gomes Moura Pires

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2012

Implementation for Spatial Data of the Shared Nearest Neighbour with Metric Data Structures

In loving memory of José Jerónimo Guerreiro Faustino and
Rogélia Ermelinda Salgueiro Faustino.

Acknowledgements

First and foremost, I would like to express my deep gratitude to Assistant Professor João Moura Pires, for relying on me with this thesis and for all his challenging and valuable questions, useful reviews, patient guidance and encouragement, not only regarding this thesis, but also about helping me grow as an individual. I would like to thank the Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa for providing its students with such a grand campus, not only for its size and for the variety of services available on it, but also for the significant number and size of green spaces. I am particularly grateful to the Computer Science Department for giving me a research grant in the System for Academic Information Analysis (SAIA) project, envisioned and guided also by Assistant Professor João Moura Pires. I wish to acknowledge and thank several other professors: Assistant Professor Maribel Yasmina Santos, for providing her knowledge about the clustering algorithm used in this thesis; Assistant Professor Margarida Mamede, for sharing her knowledge in metric spaces; Assistant Professor João Costa Seco, who answered some questions regarding the interoperability of programming languages; and Assistant Professor Artur Miguel Dias, for explaining how the Java Native Interface works. I would also like to thank all my colleagues that presented me with memorable moments during the elaboration of this thesis, namely the following: André Fidalgo, for all his support and contributions in brainstorms; Sérgio Silva, for his support; Márcia Silva, for all the fun and unwind moments; and Manuel Santos, for helping prepare the server for the experimental results and sharing his knowledge in emerging technologies.

Thanks to Complexo Educativo Cata-Ventos da Paz, Escola Básica D. António da Costa and Escola Secundária Cacilhas-Tejo, for the given educational background.

Thanks to the Clube Português de Caravanismo, where part of this document was wrote, for providing its campers with such sound conditions and environment.

Last, but not least, my special thanks to my parents, for the given education and for the given support during the elaboration of this thesis. I would also like to express my special thanks to all the remaining family members and friends, that had an important role during this important moment in my life.

Abstract

Clustering algorithms are unsupervised knowledge retrieval techniques that organize data into groups. However, these algorithms require a meaningful runtime and several runs to achieve good results.

The Shared Nearest Neighbour (SNN) is a data clustering algorithm that identifies noise in data and finds clusters with different densities, shapes and sizes. These features make the SNN a good candidate to deal with spatial data. It happens that spatial data is becoming more available, due to an increasing rate of spatial data collection. However, the SNN time complexity can be a bottleneck in spatial data clustering, since it has a time complexity in the worst case evaluated in $O(n^2)$, which compromises its scalability.

In this thesis, it is proposed to use metric data structures in primary or secondary storage, to index spatial data and support the SNN in querying for the *k*-nearest neighbours, since this query is the source of the quadratic time complexity of the SNN.

For low dimensional data, namely when dealing with spatial data, the time complexity in the average case of the SNN, using a metric data structure in primary storage (*k*d-Tree), is improved to at most $O(n \times \log n)$. Furthermore, using a strategy to reuse the *k*-nearest neighbours between consecutive runs, it is possible to obtain a time complexity in the worst case of $O(n)$.

The experimental results were done using the *k*d-Tree and the DF-Tree, which work in primary and secondary storage, respectively. These results showed good performance gains in primary storage, when compared with the performance of a referenced implementation of the SNN. In secondary storage, with 128000 objects, the performance became comparable with the performance of the referenced implementation of the SNN.

Keywords: df-tree, kd-tree, shared nearest neighbour, snn, spatial data

Resumo

Os algoritmos de agrupamento são técnicas não supervisionadas de aprendizagem automática, que organizam os dados em grupos. Contudo, estes algoritmos requerem um tempo de execução significativo e várias corridas para alcançar bons resultados.

O *Shared Nearest Neighbour* (*SNN*) é um algoritmo de agrupamento que identifica o ruído nos dados e encontra grupos com densidades, formas e tamanhos distintos. Estas características fazem do *SNN* um bom candidato para lidar com os dados espaciais. Acontece que os dados espaciais estão cada vez mais disponíveis, por estarem a ser colectados a um ritmo impressionante. Contudo, a complexidade temporal do *SNN* pode ser um obstáculo à aplicação do algoritmo sobre dados espaciais, visto que a sua complexidade temporal no pior caso é $O(n^2)$, o que compromete a sua escalabilidade.

Nesta dissertação, é proposto o uso de estruturas de dados métricas em memória primária ou secundária, para indexar os dados e dar suporte ao *SNN* à consulta pelos *k*-vizinhos mais próximos, já que esta é a fonte da complexidade quadrática do *SNN*.

Para dados de baixa dimensionalidade, nomeadamente com dados espaciais, a complexidade temporal no caso esperado do *SNN*, com o recurso a uma estrutura de dados métrica em memória primária (*kd-Tree*), foi melhorada para quanto muito $O(n \times \log n)$. Se for usada uma estratégia de reaproveitamento dos *k*-vizinhos mais próximos entre corridas consecutivas, é possível obter uma complexidade no pior caso avaliada em $O(n)$.

Foram usadas experimentalmente a *kd-Tree* e a *DF-Tree*, que funcionam em memória primária e secundária, respectivamente. Os resultados demonstraram ganhos de desempenho significativos em memória primária, em relação a uma implementação de referência do *SNN*. Em memória secundária, com 128000 objectos, o desempenho tornou-se comparável ao desempenho da implementação de referência do *SNN*.

Palavras-chave: *df-tree*, *kd-tree*, *shared nearest neighbour*, *snn*, dados espaciais

Contents

List of Figures

1

Introduction

This introductory chapter describes the context and motivation that lead to the main problem of this dissertation, defining some important concepts that need to be known *a priori*. Following, there is a brief description about the approach taken to address the problem and the resulting contributions of this approach . By the end of this chapter, it is described the outline of this document.

1.1 Context and Motivation

Spatial data consists generally of low dimensional data defined by spatial attributes, for instance, geographical location (latitude and longitude), and other attributes, for example, heading, speed or time [SKN07]. Nowadays, there is an increasing rate of spatial data collection [SKN07], from which useful knowledge can be extracted by using machine learning techniques [dCFLH04, Fer07, GXZ+11].

One widely used technique to extract knowledge from data is data clustering, because it does not require *a priori* information. Data clustering consists in an unsupervised knowledge retrieval technique, which forms natural groupings in a data set [DHS00]. The goal is to organize a data set into groups where objects that end up in the same group have to be similar to each other and different from those in other groups [MG09]. This way, it is possible to extract knowledge from data without *a priori* information, making data clustering a must have technique already used in many well-known fields: pattern recognition, information retrieval, etc.

In order to achieve the data clustering, there are numerous data clustering algorithms that require distance functions to measure the similarity between objects. These functions might not be specific to a data set, but are at least specific to the domain or the type of the

data set.

Clustering algorithms are broadly classified, according to the type of the clustering results [MH09]:

- Hierarchical - data objects are organized into a hierarchy with a sequence of nested partitions or groupings, using a bottom-up approach of merging clusters from smaller to bigger ones (dendrogram).
- Partition - data objects are divided into a number of non-hierarchical and non-overlapping partitions of groupings.

In addition, clustering algorithms can also be broken down to the following types, depending on their clustering process [GMW07]: center-based, where clusters are represented by centroids that may not be real objects in a data set; search-based, where the solution space is searched to find a globally optimal clustering that fits a data set; graph-based, where the clustering is accomplished by clustering a graph built from a data set; grid-based, where the clustering is done per cell of a grid created over a data set; and density-based, where clusters are defined by dense regions of objects, separated by low-density regions.

Nevertheless, the main issues in data clustering algorithms become clear, depending on the application [ESK03]:

- How to choose the number of clusters in partition algorithms or when to stop the merging operation in hierarchical algorithms? Because we can not foresee the number of clusters that fit a certain data set;
- How to identify the most appropriate similarity measure? Because the similarity between objects in the same cluster must be maximal and minimal when comparing objects in different clusters;
- How to deal with noise in data? Because usually all data sets have noise (objects that are sparse from most objects) and they may compromise the clustering;
- How to find clusters with different shapes, sizes and densities? Because not all clusters have the same shape, size and density, and they need to be clearly identified;
- How to deal with the scalability? Because, as stated before, there is a tendency towards a large amount of spatial data and this has an impact on the runtime of the clustering algorithms.

Density-based clustering algorithms are a good option to help solve some of the above issues. In density-based clustering algorithms, the clusters can be found with arbitrary shapes, having higher or lower density. This way, some of the objects in lower density regions can be defined as noise, since they are more sparse from the rest. An example of a density-based clustering algorithm is the Shared Nearest Neighbour (SNN) [ESK03].

The SNN is a partition density-based clustering algorithm that uses the *k*-nearest neighbours of each object in a data set in its similarity measure. The SNN requires three user-defined parameters:

- *k* - the number of neighbours;
- ε (*Eps*) - the similarity threshold that tries to keep connections between uniform regions of objects and break connections between transitional regions, enabling the SNN to handle clusters with different densities, shapes and sizes;
- *MinPts* - the density threshold that helps the SNN to deal with noise, because when it is met, the objects are representative. Otherwise, they are likely to be be noise in a data set.

Unlike some clustering algorithms (*e.g.* *K*-Means [Mac67], CURE [GRS98]), the SNN does not need a predetermined number of clusters, as it discovers the number by itself, giving it freedom to find the natural number of clusters, according to a data set and the user-defined parameters. Aside from this advantage, the SNN is also a good clustering algorithm because, it identifies noise in a data set and deals with clusters with different densities, shapes and sizes, which is good, because it is not restricted to certain types of clusters.

Some found applications using the SNN, support it: "*In this experiment, we observe that there is correlation between the level of noise and the learning performance of multifocal learning.*" ([GXZ+11]); "*This algorithm particularly fits our needs as it automatically determines the number of clusters – in our case the number of topics of a text – and does not take into account the elements that are not representative of the clusters it builds.*" ([Fer07]). More recently, there was also an application of the SNN over a spatial data set, regarding maritime transportation trajectories, discovering distinct paths taken by ships [SSMPW12].

However, the SNN has a bottleneck that compromises its scalability: its time complexity, which is essentially due to the *k*-nearest neighbours query executed for each object in a data set. In the worst case, the query needs to travel through all the objects for each object. Therefore, the time complexity in the worst case scenario is, in the Big O Notation, $O(n^2)$, where n represents the number of objects in a data set.

The main goal of this thesis is to improve the scalability of the SNN when dealing with spatial data, by improving its time complexity.

1.2 Problem

Regardless of the type of the clustering results or of the clustering process, some clustering algorithms require the *k*-nearest neighbours of each object in a data set for its clustering process. For instance, the Chameleon algorithm (hierarchical and graph-based) [KHK99] and the Shared Nearest Neighbour (SNN) (partition and density-based) [ESK03]. The scalability problem of these clustering algorithms that need to query for the *k*-nearest

neighbours of each object has to be addressed in order to improve their efficiency. However, the following issues should also be borne in mind, since they also have a significant impact on the performance of the clustering algorithms:

- Storage - where to store a data set to be clustered: primary or secondary storage. Choosing the type of storage is important, not only because sometimes the data sets are to huge to be simply fit in primary storage, but also due to the time constraints associated with each of these types of storage. Primary storage operations (read and write) are faster than secondary storage operations by a factor of 10^5 or more (with the current technology) [Bur10]. Since clustering algorithms need to access a data set, it is important to choose where to store it, because if it resorts to secondary storage, the secondary storage access speed is meaningful and must be accounted in the runtime of the clustering algorithm.

- Fine-tuning of Parameters - the clustering algorithms require user-defined parameters that have significant impact in the clustering results, as it will be seen in chapter 2. These user-defined parameters need to be changed and adapted according to a data set being clustered, through out a sequence of runs. This is the only way to be certain and to find the right user-defined parameters that give output to the best possible clustering results for a given data set. This can be a time consuming operation, since the algorithms need significant time to compute and to cluster all the objects in a data set on every single run, reinforcing the importance that the time complexity has on the scalability of the clustering algorithms.

In view of the SNN, to achieve an improvement in its scalability, besides dealing with the scalability problem of the *k*-nearest neighbours queries, the above points are also going to be considered, in order to improve the performance of the SNN through out a sequence of runs needed to achieve the best possible clustering results over a data set.

There is already one variant of the SNN that tries to improve its scalability. This variant is called, the Shared Nearest Neighbour Algorithm with Enclosures (SNNAE) [BJ09]. It creates enclosures in a data set to outline the range of the *k*-nearest neighbours query for each object, so it does not need to go through all the objects, only going through the ones in the same enclosure. However, the time complexity in the worst case of the SNNAE is evaluated in $O(n \times e + s^2 \times e + n \times s)$, which is still $O(n^2)$, as $s = \frac{n}{e}$, where e and s represent the number of enclosures and the average number of objects in each enclosure, respectively.

1.3 Approach

The approach taken to achieve an improvement relies on metric spaces. A metric space is represented by a data set and a distance function, which provides support on answering similarity queries: range queries or *k*-nearest neighbours queries.

Data in metric spaces is indexed in appropriate metric data structures. These structures can be stored in primary or secondary storage and, depending on the type and dimensionality of a data set, can be generic or not.

Building a metric data structure, *i.e.* indexing a data set, takes a significant runtime, but once built, they can be reused and some can be updated, avoiding the runtime required to build the structure with a data set all over again.

The goal is to improve the scalability of the SNN over spatial data, by combining the SNN with metric data structures, taking advantage of their capability of answering the *k*-nearest neighbours queries of the SNN, and by reusing the metric data structure and the queries results among consecutive runs.

Some clustering algorithms already use metric data structures, for example: the authors of Chameleon [KHK99] state that its time complexity can be reduced from $O(n^2)$ using metric data structures; CURE [GRS98] has a worst case time complexity of $O(n^2 \times \log n)$ and uses a metric data structure to store representative objects for every cluster; and DBSCAN [EKSX96] takes full advantage of a metric data structure, in order to reduce its time complexity from $O(n^2)$ to $O(n \times \log n)$.

1.4 Contributions

The contribution of this thesis is an implementation of the SNN that:

- uses metric data structures in primary or secondary storage that give support to the *k*-nearest neighbours queries of the SNN;
- uses generic distance functions or specific distance functions more appropriate for spatial data;
- has its time complexity evaluated, when working in primary storage;
- has the quality of its clustering results evaluated by comparing its results with results referenced in the literature;
- has its scalability evaluated by using a spatial dataset with a meaningful size and by comparing its performance with a referenced implementation of the original SNN.

1.5 Outline of the Document

The rest of the document is organized in six chapters. The related work is divided in two chapters, due to the importance of each presented subject: in chapter 2 it is explained the basic concepts about clustering algorithms and some clustering algorithms are detailed; in chapter 3 important properties about metric spaces and metric data structures are explained and some metric data structures are also detailed. On chapter 4, the approach taken regarding the use of metric data structures in primary or secondary storage with

the SNN is explained. The experimental results are disclosed in chapter 5, where the evaluation of the quality and validity of the clustering results is made, and the performance of this approach is evaluated. Finally, in chapter 6 the conclusions are taken and the future work is unveiled.

2

Clustering Algorithms

This chapter details some concepts of clustering algorithms and explains the main clustering algorithm in this thesis, the SNN, and other clustering algorithms: SNNAE, Chameleon, CURE and DBSCAN.

2.1 Introduction

A data set $D = \{x_1, \ldots, x_n\}$ is a collection of n objects. The object $x_i = (x_{i,1}, \ldots, x_{i,dim})$ is described by dim attributes, where $x_{i,j}$ denotes the value for the attribute j of x_i [GMW07].

Clustering requires the evaluation of the similarity between objects in a data set. The most obvious similarity measure is the distance between two objects. If the distance is a good measure of similarity, it is expected that the distance between objects in the same cluster is smaller and it is bigger between objects in different clusters [DHS00]. Given a data set, it is a hard task to choose which distance function to use to evaluate the similarity between objects .

A Minkowski distance is an example of a widely used family of generic distance functions [SB11]. Given x and y that represent two points in a dim-space and a $p \geq 1$ (in order to hold the properties of a metric), a Minkowski distance is defined by:

$$L_p(x, y) = \left(\sum_{i=1}^{dim} |x_i - y_i|^p \right)^{1/p}. \tag{2.1}$$

When $p = 1$, we get a well-known distance function, the Manhattan distance:

$$L_1(x, y) = \sum_{i=1}^{dim} |x_i - y_i|. \tag{2.2}$$

And when $p = 2$, we obtain the most common and generic distance function [ESK03], the Euclidean distance, where x and y represent two points in a euclidean dim-space:

$$L_2(x, y) = \sqrt{\sum_{i=1}^{dim} (x_i - y_i)^2} = \sqrt{(x_1 - y_1)^2 + \ldots + (x_{dim} - y_{dim})^2}. \tag{2.3}$$

However, the Euclidean distance function is not always the best, as it does not work well when a data set that is being clustered has high dimensionality, because same objects in the data set are sparser and their differences might be very small to distinguish with. This fact is even more noticeable with some types of data, for instance: binary types [ESK03]. Therefore, the chosen distance function must be suited for a data set and this is why choosing a distance function is not a simple task.

Another task is choosing the right user-defined parameters that the algorithms need before they start clustering a data set. These user-defined parameters have impact on the clustering results, so they need to be chosen with advisement, *i.e.* density, shape and size of clusters identified depends not only in the algorithm itself, but also on the user-defined parameters [RPN+08].

Afterwards, as clustering algorithms are an unsupervised learning technique and there might not be predefined classes or examples that can help validate the clustering results, it becomes necessary an evaluation from an expert user in the application domain or the use of cluster validity measures that measure if the found clustering results are acceptable or not. These cluster validity measures may be generic (explained in more detail on section 2.5) or specific to the application domain.

2.2 Shared Nearest Neighbour

The Shared Nearest Neighbour (SNN), introduced by Jarvis and Patrick [JP73], is a partition density-based clustering algorithm that uses the number of shared nearest neighbours between objects as a similarity measure.

Later, the SNN was improved with a density measure, so the algorithm could handle noise and clusters with different densities, shapes and sizes [ESK03].

2.2.1 Parameters

The initialization of the algorithm requires the following user-defined parameters:

- *k* - the number of neighbours;

- ε (*Eps*) - the minimum similarity between two objects. It tries to keep connections between uniform regions of objects and break connections between transitional regions;
- *MinPts* - the minimum density of an object. It is used to determine if an object is representative or is likely to be noise in a data set.

Besides these user-defined parameters, the algorithm needs a distance function to be used in the *k*-nearest neighbours queries for each object.

2.2.2 Algorithm

The SNN algorithm starts by computing a $n \times n$ matrix with the distances between all pair of objects in a data set (Figure 2.1). In order to measure these distances, the SNN uses a generic or a specific distance function defined by the user. Afterwards, by using this matrix and the k value defined also by the user, the SNN is able to obtain the *k*-nearest neighbours of each object, by keeping the k objects in each row with the smallest distance values.

	1	2	...	n
1	$d(1,1)$	$d(1,2)$	...	$d(1,n)$
2	$d(2,1)$	$d(2,2)$	...	$d(2,n)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n	$d(n,1)$	$d(n,2)$	...	$d(n,n)$

Figure 2.1: SNN - Distances Matrix

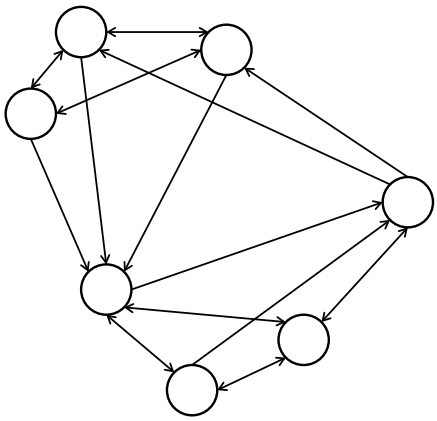


Figure 2.2: SNN Example - Graph with the 3-nearest neighbours of the 7 Objects

After obtaining the *k*-nearest neighbours of each object, a weighted graph is created with all the objects in the data set and with each of them connected to all their respective *k*-nearest neighbours, as seen in the example in figure 2.2. The example in figure 2.2 is represented by a data set with 7 objects and their respective 3-nearest neighbours. Each

node represents one of the 7 objects and each arrow points to one of the 3 neighbours of an object. When the arrow is a left right arrow, it means that both objects have each other in their sets of k-nearest neighbours, *i.e.* they are mutual neighbours. If it only points in one direction, it means that the source object of the arrow has the object pointed by the arrow as a neighbour.

After creating the weighted graph with its contents, two objects stay connected, if and only if, they have each other in their sets of *k*-nearest neighbours. If this is the case, the weight of the connection is the value of their similarity, *i.e.* the number of shared neighbours between the two connected objects (Figure 2.3).

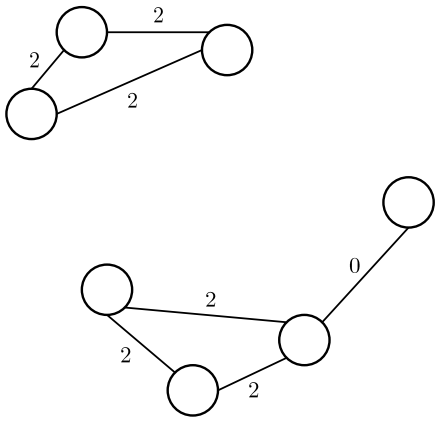


Figure 2.3: SNN Example - Graph with the Similarity Values (*Eps* = 2)

The number of connections that each object has, defines the density of the object. However, all connections that do not surpass the user-defined similarity threshold (*Eps*) are eliminated and are not considered in the density of the object. If an object has a density value bigger than the user-defined density threshold (*MinPts*), then the object is a representative object. Otherwise, it is likely to be noise in a data set. See figure 2.4, where the grey objects are the representative objects.

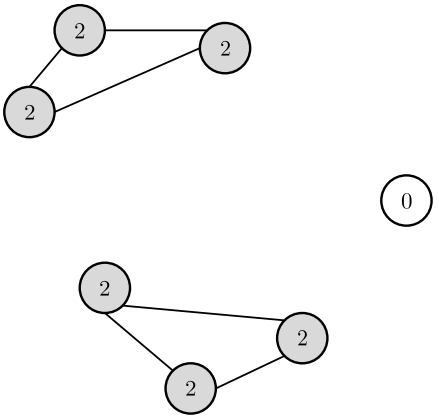


Figure 2.4: SNN Example - Graph with the Density Values (*MinPts* = 2)

The clusters are formed by the established connections, where non-noise objects in a

cluster are representative or are connected directly or indirectly to a representative object. The objects that do not verify these properties are finally declared as noise (Figure 2.5, where each color represents a distinct cluster).

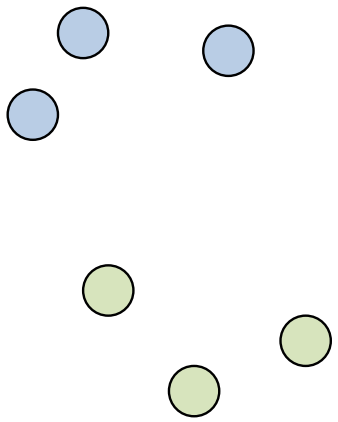


Figure 2.5: SNN Example - Graph with the Clustering Results

This approach will be referenced in this document as the Original SNN.

2.2.3 Other Variants

In [MSC05] the SNN and the DBSCAN, as representatives of density-based clustering algorithms, are analysed. The authors show some proposed variants of the SNN, namely a representative object-based approach and another one based on graphs. The authors also show that the results obtained with these variants are very similar to each other and very similar to the Original SNN.

The representative objects-based approach differs in how the clustering is done. After identifying the representative objects, it starts by clustering them, checking the properties needed to connect two objects. Then, all non representative objects that are not in a radius of *Eps* of a representative object, are defined as noise. Finally, non representative objects that were not defined as noise, are clustered in the cluster of the representative object with which they share the biggest similarity.

The graph-based approach follows the essence of the Original SNN, only diverging in how the noise is identified. After identifying the representative objects, an object is defined as noise, if it is not in a radius of *Eps* of a representative object. Then, the clustering is done in the same way as the Original SNN. Objects in a cluster are not noise, and are representative or are directly or indirectly connected to a representative object, checking the necessary properties to establish a connection between two objects.

Regardless of the approach, the Original SNN or one of these found variants, the time complexity in the worst case of the SNN is $O(n^2)$, where n represents the number of objects in a data set. This time complexity is associated with the need to compute a distances matrix to obtain the *k*-nearest neighbours.

With the purpose of improving this limitation, comes another variant of the SNN,

the Shared Nearest Neighbours Algorithm with Enclosures (SNNAE) [BJ09] (explained in more detail on section 2.3.1).

2.2.4 Synopsis

The SNN is a partition density-based clustering algorithm that:

- Does not need the number of clusters as an user-defined parameter, discovering by itself the natural number of clusters, according to the contents of a data set and the user-defined parameters;
- Deals with noise (Figure 2.6, based on [ESK03], where the red dots and the blue dots, point out the noise and the clustered objects, respectively);

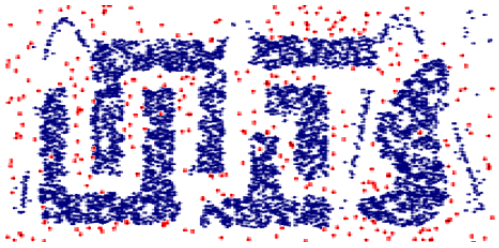


Figure 2.6: Clusters and Noise Identified by the SNN

- Finds clusters with different densities, shapes and sizes.

However, the SNN has a:

- Space complexity of $O(n^2)$, due to the space occupied by the matrix with the distances between objects;
- Time complexity in the worst case of $O(n^2)$, due to the operation that fills the matrix with all the distances between all pairs of objects, in order to get the k-nearest neighbours of each object.

2.3 Other Clustering Algorithms

On this theoretical research, several algorithms were studied: SNNAE [BJ09], Chameleon [KHK99], CURE [GRS98] and DBSCAN [EKSX96]. These algorithms were chosen, because they are briefly introduced in [ESK03] and some are also tested against the Original SNN.

2.3.1 SNNAE

The SNNAE is a partition density-based clustering algorithm and is a variant of the Original SNN. This approach tries to make the k-nearest neighbours queries more efficient by creating enclosures over a data set, since these queries are the source of the quadratic time

complexity of the Original SNN. The purpose of these enclosures is to limit the range of search for the *k*-nearest neighbours of an object, requiring less distance calculations and a smaller runtime, since less objects are queried about their distances to another object.

In order to create the enclosures, the algorithm requires a less expensive distance function to calculate the division of data in overlapping enclosures. Then, the SNNAE uses a more expensive distance function to find the *k*-nearest neighbours, only inside the enclosure of the queried object.

The enclosures are created over a data set. First of all, all the objects are stored in a single cluster. Then, it is calculated the radius R that covers all the objects in the cluster, where $\vec{x_i}$ denotes the ith object from the n objects in a data set and $\vec{x_0}$ is the center of all objects in a data set:

$$\vec{x_0} = \sum_{i=1}^{n} \frac{\vec{x_i}}{n} \tag{2.4}$$

$$R = \left(\sum_{i=1}^{n} \frac{(\vec{x_i} - \vec{x_0})^2}{n} \right)^{\frac{1}{2}} \tag{2.5}$$

From this radius R, the circular area that covers a data set, where dim represents the dimensionality of a data set, is:

$$Circular\ Area = 3.14 \times R^{dim} \tag{2.6}$$

And the rectangular area, where L_i is the difference between the maximum and minimum value for the attribute i, is:

$$Rectangular\ Area = \prod_{i=1}^{dim} L_i \tag{2.7}$$

Then, using the $ratio$ between the two areas (circular and rectangular), a radius r of the overlapped enclosures is calculated:

$$r = dim \times ratio + \frac{ratio}{2} \tag{2.8}$$

Thereafter, the first object o in a data set is considered the center of the first enclosure. From this step forward, one of three things can happen to the remaining objects (see Figure 2.7 based on [FSTR06]):

- Nearest Adjacent Object - All the objects that have a distance from o smaller than or equal to r are gathered in the list of nearest adjacent objects of that enclosure;
- Nearest Far Adjacent Object - All the objects with a distance from o greater than r and less than or equal to $r \times 1.5$ are gathered in the list of nearest far adjacent objects of that enclosure;

- Center of the Next Enclosure - All the objects whose distance from o is greater than $r \times 1.5$ and less than $r \times 2$ are considered to be as center of the next enclosure.

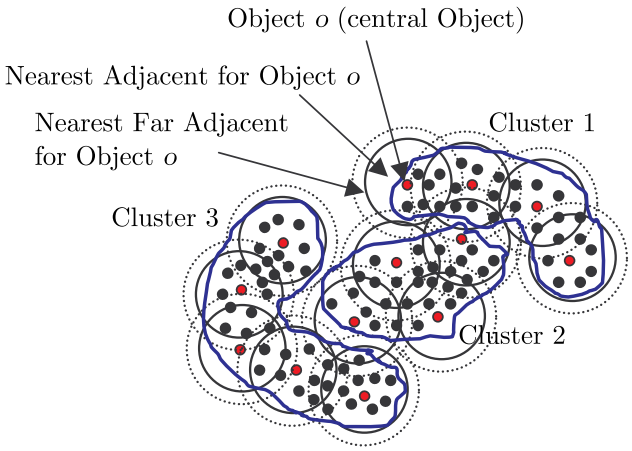


Figure 2.7: Enclosures in the SNNAE

Following the creation of the enclosures, the next step is to find the optimal value for ε (*Eps*). In the SNNAE, the *Eps* is calculated dynamically, instead of being a user-defined parameter like in the Original SNN. *Eps* is calculated by evaluating the distances between every pair of objects in one enclosure, in order to find the maximum distance in one enclosure. This process is repeated for all enclosures. Then, *Eps* becomes the average of the maximum distances in the enclosures.

In the final step, the Original SNN algorithm is applied, but this time, the algorithm only searches for the *k*-nearest neighbours over the enclosure of the queried object.

The user-defined parameters of the SNNAE are:

- *k* - the number of neighbours;
- *MinPts* - the minimum density of an object. It is used to determine if an object is representative or is likely to be noise in a data set.

Besides these user-defined parameters, the SNNAE also requires a less expensive distance function for the creation of the enclosures and a more expensive distance function to obtain the *k*-nearest neighbours.

According to the authors, the SNNAE is more scalable, more efficient and obtains better clustering results than the Original SNN. However, it is hard to verify these assertions, since the experimental results in [BJ09] were carried out with a very small data set even when the purpose is to validate the scalability improvement obtained with the SNNAE. Some possible conclusions are drawn from figure 2.8, taken from [BJ09], where it can be seen that the SNNAE takes less half of the time finding the *k*-nearest neighbours, only taking a little more time calculating the initial clusters and composing the final clusters, when using a data set with 209 objects, each with 7 integer attributes.

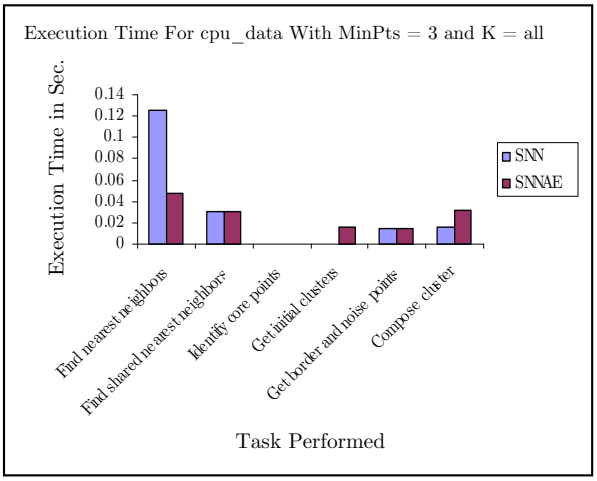


Figure 2.8: Runtime of the SNNAE and the Original SNN

Regarding the quality of the clustering results, these are not shown and are not compared with the ones obtained with the Original SNN, making it hard to check if they are better or not.

The time complexity of the SNNAE in the worst case is evaluated in $O(n \times e + s^2 \times e + n \times s)$, where e and s represent the number of enclosures and the average number of objects in each enclosure, respectively. However, since s is the average number of objects per enclosure, *i.e.* $s = \frac{n}{e}$, we can deduce that the time complexity of the SNNAE is also quadratic, $O(n^2)$.

2.3.2 Chameleon

The Chameleon is a two-phase hierarchical clustering algorithm (Figure 2.9, taken from [KHK99]).

It starts by creating a list of the *k*-nearest neighbours of each object in a data set. Then, using the just filled list, a graph is constructed, where each node represents an object and connects to another, if and only if, both nodes have each other in their sets of *k*-nearest neighbours.

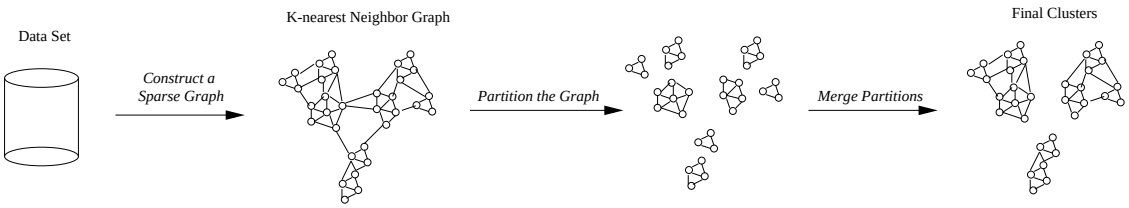


Figure 2.9: Overview of the Chameleon

In the next step, it uses a partition algorithm that splits the graph in smaller clusters. Then, it applies a hierarchical algorithm that finds the genuine clusters by merging

smaller ones together. In order to do this, the algorithm uses the interconnectivity between clusters and their nearness, measured by thresholds and conditions imposed at the initial parametrization of the Chameleon. This hierarchical phase stops when a scheme of conditions can no longer be verified. There are two types of schemes implemented in the Chameleon:

- The first one only merges two clusters that have their interconnectivity and nearness above some user-defined thresholds: $T_{Interconnectivity}$ and $T_{Nearness}$;
- The second scheme uses a function (2.9) that combines the value of the interconnectivity and nearness of a pair of clusters, merging the pair of clusters that maximizes the function.

$$Interconnectivity(Cluster_i, Cluster_j) \times Nearness(Cluster_i, Cluster_j)^{\alpha}. \quad (2.9)$$

  This function needs another user-defined parameter named α. If α is bigger than 1, then the Chameleon gives more importance to the nearness of clusters. Otherwise, it gives more importance to their interconnectivity.

The user-defined parameters of the Chameleon are:

- *k* - the number of neighbours;
- *MinSize* - the minimum granularity of the smaller clusters acquired after applying the initial partition algorithm;
- α - used to control the interconnectivity and nearness of clusters on the second scheme.

Beyond these user-defined parameters, the algorithm needs a distance function to find the *k*-nearest neighbours and, as already pointed out, a scheme of conditions to be used in the hierarchical phase of the Chameleon.

2.3.3 CURE

The CURE is a hierarchical clustering algorithm.

The CURE starts by creating a cluster for each single object in a data set. From this step forward, it merges the clusters that are nearer. To do so, it needs to calculate the distance from these clusters by using their representative objects.

In the CURE a representative object is a central object of the cluster. To find the set of representative objects of a cluster, it starts by identifying well scattered objects that form an area (outline) of the cluster. Then, by closing in on the area by a fraction of α, it gets down to the central objects of the cluster.

Now, it calculates the distance from a representative object from the set of representative objects of a cluster and one from the set of another cluster. This step is repeated for

all pairs of representative objects in the sets of a pair of clusters, until it finds the minimum distance between two clusters in evaluation. After finding the minimum distance between all the clusters, it merges the ones that are closer to each other.

All the previous operations are repeated until it cuts down the number of clusters to a user-defined number.

The user-defined parameters of the CURE are:

- α - the fraction used to close in on the area;
- *K* - the number of clusters.

Besides these user-defined parameters, the CURE also requires a distance function to evaluate the distance between representative objects.

2.3.4 DBSCAN

The DBSCAN is a partition density-based clustering algorithm. For each single object in a cluster, its *Eps*-neighbourhood must have at least a user-defined number of objects (*MinPts*). The shape of its *Eps*-neighbourhood is determined by the distance function used.

To reach the notion of a cluster, in the context of the DBSCAN, the following definitions must be met:

- *Eps*-neighbourhood - the objects inside a radius $\leq$ *Eps* of an object o:

$$N_{Eps}(o) = p \in D \mid d(o,p) \leq Eps \tag{2.10}$$

  , where D represents a data set, p another object and d the distance function chosen.

- Directly Density-Reachable - an object o is directly density-reachable from another object p, if:

$$o \in N_{Eps}(p) \wedge |N_{Eps}(p)| \geq MinPts. \tag{2.11}$$

- Density-Reachable - an object o is density-reachable from another object p if a chain of objects $o_1, o_2, \ldots, o_n, o_1 = p, o_n = o$ exists, such that o_{i+1} is directly density-reachable from o_i;

- Density-Connected - an object o is density-connected to a another object p if an object q exists in a such a way that both o and p are density-reachable from q;

- Cluster - finally, in order to obtain a cluster, the following properties must be verified:

$$\forall o, p : \text{if } o \in C \text{ and } p \text{ is density-reachable from } o, \text{ then } p \in C \tag{2.12}$$

$$\forall o, p \in C : o \text{ is density-connected to } p \tag{2.13}$$

, where C represents a cluster.

- Noise - objects that did not fit into any cluster are defined as noise:

$$\forall o \in D \mid \forall i : o \notin C_i \tag{2.14}$$

, where i represents the number of clusters found.

The DBSCAN starts with an arbitrary object that has not yet been visited and obtains all its density-reachable objects. If *MinPts* is met, then a cluster is created. Otherwise, a cluster is not created and the object is marked as visited. The algorithm stops when there are no more objects that can be fit into clusters.

The user-defined parameters of the DBSCAN are:

- *Eps* - the radius of the neighbourhood;
- *MinPts* - the minimum number of objects required to form a cluster.

Besides these user-defined parameters, the algorithm needs a distance function to get the objects inside the *Eps*-neighbourhood of an object.

2.4 Evaluation of the Clustering Algorithms

In this section, all of the presented clustering algorithms in 2.3 are compared with the Original SNN.

2.4.1 SNN and Chameleon

One downside of the Chameleon, where compared to the SNN, is that it does not work well with high dimensionality data (*e.g* documents) [ESK03], because distances or similarities between objects become more uniform, making the clustering more difficult. On the other hand, the SNN promises a way to deal with this issue by, not only using a similarity measure that uses the number of shared neighbours between objects, but also using a density measure that helps to handle the problems with similarity that can arise when dealing with high dimensionality problems.

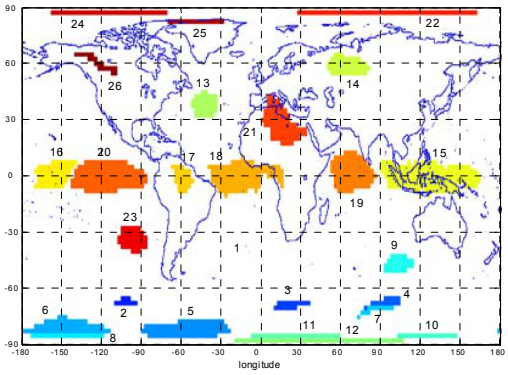
2.4.2 SNN and CURE

When comparing the CURE with the SNN, there are three drawbacks that the CURE has and the SNN has not. First, the user needs to define the number of clusters, as the CURE does not naturally discover the number, according to the remaining user-defined parameters. Discovering the number of clusters naturally can be good in some applications (*e.g.* [Fer07]). Second, the CURE has a limited set of parameters which, despite of making easier the fine-tuning, restricts the control over the clustering algorithm and, consequently,

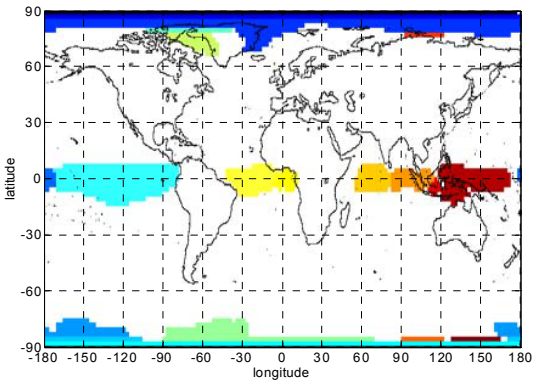
its clustering results. Third, the CURE handles many types of non-globular clusters, but it cannot handle many types of globular shaped clusters [ESK03]. This is a consequence of finding the objects along the outline of the cluster and then shrinking the area of the outline towards the center. Whereas, the SNN is able to identify clusters with different densities, shapes and sizes, without compromising any of them.

2.4.3 SNN and DBSCAN

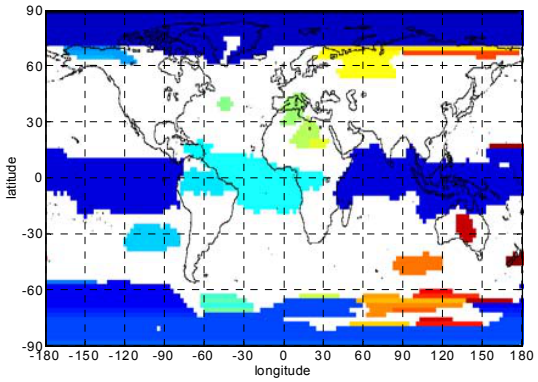
In some experimental results [ESK03, MSC05] that compare the SNN and the DBSCAN, the quality of the clustering results of the DBSCAN was worse than the quality of the clustering results of the SNN. Figure 2.10, taken from [ESK03], shows the clusters obtained with a NASA Earth science time series data, where the densities of the clusters is important for explaining how the ocean influences the land. According to the authors, the SNN has able to identify clusters of different densities. However, the DBSCAN could not find the clusters with a higher density without compromising the clusters with a lower density.



(a) SNN



(b) DBSCAN (*Eps* = 0.985, *MinPts* = 10)



(c) DBSCAN (*Eps* = 0.98, *MinPts* = 4)

Figure 2.10: Clustering Results from the SNN and the DBSCAN

For example, in figure 2.10a, the SNN identified the cluster number 9 and the cluster number 13, while the DBSCAN could not identify them in the clustering presented in

figure 2.10b. In an attempt to identify these clusters, the parameters of the DBSCAN were adjusted (Figure 2.10c). With this adjustment, the DBSCAN has able to identify these clusters, but compromised the quality of the other clusters previously identified, which had a higher density.

The DBSCAN showed that it could not find the low-density clusters without compromising the quality of the high-density clusters, which implies that the DBSCAN cannot handle data containing clusters of differing densities. On the other hand, the SNN has able to simultaneously find clusters of different densities.

2.5 Validation of Clustering Results

After obtaining the clustering results, they must be evaluated but, as there might not be *a priori* information, it is difficult to be certain that the obtained clustering is acceptable or not. In general terms, there are three types of criteria that define the validity measures used to evaluate the clustering results [HBV02a, HBV02b]:

- External - the clustering results are evaluated based on a pre-specified structure, which is imposed on a data set and reflects our intuition about the clusters of a data set. Given a set of clustering results $C = \{C_1, C_2, \ldots, C_m\}$ and a defined by intuition partition of a data set $P = \{P_1, P_2, \ldots, P_s\}$, there are two different approaches: comparing the clusters with a partition of data, where several indices can be used to measure the degree of similarity between C and P (*e.g.* Jaccard Coefficient) or comparing a proximity matrix Q to a partition P;

- Internal - the results are evaluated in terms of quantities and features inherited from a data set. These criteria depends on the type of the clustering results. If it is a hierarchical clustering algorithm, then a hierarchy of clustering schemes must be applied. Otherwise, if it is a partition clustering algorithm, then a single clustering scheme has to be used;

- Relative - done by comparing the clustering results to other results obtained using the same clustering algorithm, but with different parameters. That is, using a set of possible values for each parameter of the clustering algorithm, the validity measure depends on the existence or not of the number of clusters as a parameter. If the number of clusters is not a parameter, then it uses a procedure that calculates a set of validity indexes in a number of runs of the clustering algorithm, and then picks the best value of the validity indexes obtained in each run.

These are some validity measures that evaluate the clustering results. However, the basic idea behind this metrics depends on the distance between representative objects, on the number of objects in each cluster and on average values. But, for example, if we are using an algorithm that gives output to arbitrary shaped clusters that do not have a specific representative object as they have arbitrary shapes, then the validity measures

cannot make the validation properly. The alternative is to use validity measures that work better with some types of clustering algorithms or that do not use representative objects, for instance: one that is based on the variance of the nearest neighbour distance in a cluster [KI06]; one that is formulated based on the distance homogeneity between neighbourhood distances of a cluster and density separateness between different clusters [YKI08]; or one based on the separation and/or overlap measures [Ža11].

2.6 Summary

The following figure 2.11 summarizes some important properties, regarding all the previous studied algorithms (Sections 2.2 and 2.3). This table was built using [MTJ06, ZWW07, EKSX96, GRS98, KHK99, BJ09, ESK03], where:

- n - the number of objects in a data set;
- Chameleon: k - the number of neighbours; *MinSize* - controls the minimum granularity of the smaller clusters acquired after applying the initial partition algorithm; α - used to control the interconnectivity and nearness of clusters on the second scheme;
- CURE: α - the fraction used to close in on the area; K - the number of clusters;
- DBSCAN: m - portion of the original data set; *Eps* - the radius of the neighbourhood; *MinPts* - the minimum number of objects required to form a cluster;
- SNN: k - the number of neighbours; ε (*Eps*) - the minimum similarity between two objects; *MinPts* - the minimum density of an object;
- SNNAE: e - the number of enclosures; s - the average number of objects in each cluster; k - the number of neighbours; *MinPts* - the minimum density of an object.

Clustering Algorithm	Time Complexity		Handles			Clusters Shape	User-defined Parameters
	Average Case	Worst Case	High Dimensionality	Large Datasets	Noise		
Chameleon	-	$O(n^2)$	No.	Yes.	Yes.	All.	$K, MinSize, \alpha$
CURE	$O(n^2)$	$O(n^2 \times \log n)$	No.	Yes.	Yes.	Non-globular	α, K
DBSCAN	$O(n \times \log n)$	$O(n^2)$	No.	Yes.	Yes.	All.	$\varepsilon, MinPts$
SNN	-	$O(n^2)$	Yes.	Yes.	Yes.	All.	$K, \varepsilon, MinPts$
SNNAE	-	$O(n \times e + s^2 \times e + n \times s)$	Yes.	Yes.	Yes.	All.	$K, MinPts$

Figure 2.11: Studied Clustering Algorithms

3

Metric Spaces

This chapter starts by introducing the definition and the properties of metric spaces. Afterwards, there is a description about and some existing types of metric data structures, which index data in metric spaces. In the end, some chosen metric data structures are explained with a specific focus on their algorithm for the *k*-nearest neighbours query.

3.1 Introduction

A metric space is a pair, $MS = (\mathbb{U}, d)$, where $\mathbb{U}$ represents the universe of valid objects and d a distance function that measures the distance between objects in $\mathbb{U}$ [CNBYM01].

The distance function of the metric space must obey to the following properties:

$$d : \mathbb{U} \times \mathbb{U} \to \mathbb{R}_0^+ \tag{3.1}$$

$$\forall x, y \in \mathbb{U} \qquad d(x, y) \geq 0 \tag{3.2}$$

$$\forall x, y \in \mathbb{U} \qquad d(x, y) > 0 \iff x \neq y \tag{3.3}$$

$$\forall x, y \in \mathbb{U} \qquad d(x, y) = 0 \iff x = y \tag{3.4}$$

$$\forall x, y \in \mathbb{U} \qquad d(x, y) = d(y, x) \tag{3.5}$$

$$\forall x, y, z \in \mathbb{U} \qquad d(x, z) \leq d(x, y) + d(y, z) \tag{3.6}$$

If the strict positiveness (Property 3.3) is not verified, then the space is called a pseudometric space. However, the most important property is the triangle inequality (Property 3.6), since it helps to prune the similarity queries over the metric space.

There are essentially three basic types of similarity queries over a metric space:

- Range - Given a data set $\mathbb{X} \subseteq \mathbb{U}$, a query radius $r \in \mathbb{R}^+$ and an object $q \in \mathbb{X}$:

$$\begin{aligned} \text{Range}(\mathbb{X}, q, r) = \mathbb{Y} \text{ where, } \mathbb{Y} \subseteq \mathbb{X} \\ \forall y \in \mathbb{Y} \qquad d(q, y) \leq r. \end{aligned} \tag{3.7}$$

  Returns all the objects in the metric space, that are inside a specified query radius r of an object q.

- k-Nearest Neighbours - Given a data set $\mathbb{X} \subseteq \mathbb{U}$, a natural number $k \in \mathbb{N}$ and an object $q \in \mathbb{X}$:

$$\begin{aligned} k\text{-}NN(\mathbb{X}, q, k) = \mathbb{Y} \text{ where, } \mathbb{Y} \subseteq \mathbb{X} \text{ and } |\mathbb{Y}| = k \\ \forall y \in \mathbb{Y}, \quad \forall x \in \mathbb{X} - \mathbb{Y} \qquad d(q, y) \leq d(q, x). \end{aligned} \tag{3.8}$$

  Returns the k objects in the metric space that are nearer a queried object q.

- Nearest Neighbour - Does the same as the k-Nearest Neighbours query (3.8), but with k having the value of 1, returning only the nearest neighbour.

Data in metric spaces is indexed in appropriate metric data structures. These metric data structures provide support for the above similarity queries, giving the possibility of making these queries more efficient by minimizing the number of distance evaluations.

3.2 Metric Data Structures

The classification of metric data structures can be summarized into the following taxonomy (see Figure 3.1 based on [CNBYM01]):

- Voronoi Type - also known as *clustering the space*. The space is divided into areas, where each area is defined by a center and some objects from a data set. This allows the search to be pruned by discarding an area and the objects inside it, when comparing the queried object with the center of an area [NUP11]. An area can be outlined by a:
  - Hyperplane - hyperplane between the center of two areas;
  - Covering Radius - a radius around the center of an area.

Covering Radius is best suited for indexed data sets with a higher dimensionality metric space or a larger query radius, but it is generally impossible to avoid covering radius overlaps in space and, therefore, even exact queries have to go through several branches of the metric data structure. Nevertheless, these overlaps help with the insertion of new objects.

On the other hand, this problem does not occur when using hyperplane-based structures, because there are no overlaps. However, this method is more rigid and complicates update operations [NUP11].

- Pivot-Based - uses the notion of pivots (a subset of objects from a data set). The data structure calculates and stores the distances between each pivot and the remaining objects. Afterwards, when a query is executed, the query evaluates the distance between the queried object and the pivots. Given an object $x \in \mathbb{X}$, a set of pivots p_i, a queried object q and a distance r, x can be discarded if [PB07]:

$$|d(p_i, x) - d(p_i, q)| > r \text{ for any pivot from the set } p_i \tag{3.9}$$

The above calculation uses the calculated distances in a preprocessing stage and the triangle inequality (Property 3.6) to directly discard objects. With this, there is a speed up of the search by avoiding some distance evaluations between objects that will not belong in the query result [SMO11].

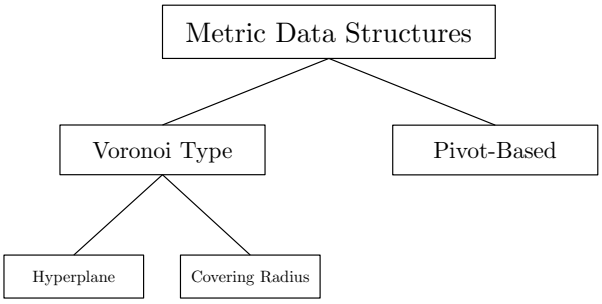


Figure 3.1: Metric Data Structures Taxonomy

Besides these classifications, all the data structures can be stored in primary or secondary storage and can also be defined as [Sar10]:

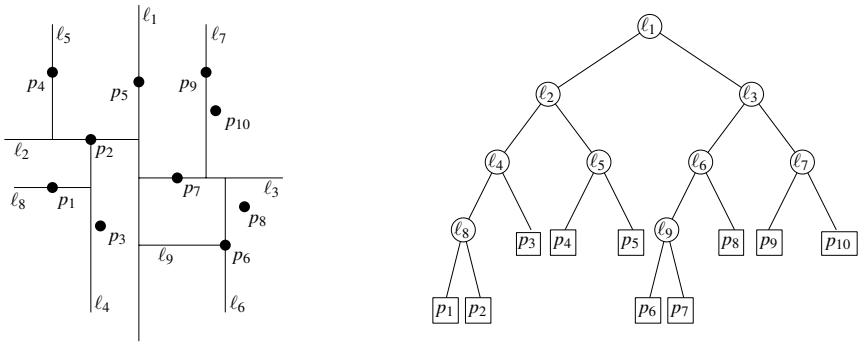
- Dynamic - if they have update (insert or remove) operations. If they do not, they are defined as static;

- Generic - if they accept every data type and distance functions. If they work with specific data types or specific distance functions, then they are defined as non-generic (specific).

Metric data structures based on *clustering the space* are more suited for high-dimensional metric spaces or larger query radius, that is, when the problem is more difficult to solve [NUP11]. Due to this fact, the ramification about Pivot-based metric data structures was not explored in this thesis and the chosen metric data structures are based on *clustering the space*: the *k*d-Tree [Ben75], because it is one of the most popular in the literature [CNBYM01], is written for primary storage in various programming languages, has its time complexity evaluated in the Big O Notation and is suited for low dimensionality problems (*e.g.* spatial data) [Mar06]; and the DF-Tree [TTFF02], because there are few options of secondary storage metric data structures and because it shows better experimental results [TTFF02]. The *M*-Tree [CPZ97] is also explained, because the DF-Tree is a descendent of the *M*-Tree.

3.2.1 *k*d-Tree

The *k*d-tree is a dynamic, non-generic and primary storage metric data structure for indexing a finite *k*-dimensional metric space that recursively splits a data set into partitions of points.



(a) Partitions (b) Tree

Figure 3.2: *K*d-Tree Example

Each non-leaf node in the tree represents a split made in a data set and is a binary search tree. The root of the tree represents the partition that contains the whole data set.

The canonical idea behind the *k*d-Tree is [FBF77]:

- The dimension used to split a data set into partitions changes for every single level of the tree, cycling through all the *k* dimensions;
- The split value of the root of every tree can be chosen using a median point over the current dimension value of the points in the current partition (leads into a balanced *k*d-tree);
- All the points of the current partition with a dimension value less than or equal to the split value of the root, go into the left tree, representing the left partition.

Otherwise, the points end up in the right tree, representing the right partition.

- Leaf nodes store the actual points in a data set.

Figure 3.2 taken from [BCKO10] illustrates the partitions made to a data set and the respective *k*d-tree.

The algorithm of the *k*-nearest neighbours query [FBF77] of the *k*d-tree starts by finding the last partition that includes the query point, that is, goes left or right in the tree in the same way that it was built. It explores the left side of the tree, if the split dimension value of the query point is less than or equal to the respective dimension value of the root. Otherwise, it explores the right side of the tree.

When it reaches a leaf node, the points are examined and the queue with the nearest points is updated. If the points found are nearer than the further points in the queue. Then, it backtracks and goes up in the tree.

When it reaches a non-leaf node, it checks if the queue already has a user-defined number of nearest neighbours:

- If it has, then it checks if the geometric boundaries delimiting the points in the sub-tree intersect the further point in the queue. If they do intersect, than it explores the sub-tree recursively for nearer points. If they do not intersect, the search is pruned, the sub-tree is not explored and it goes up.
- If it has not, then the sub-tree is explored.

The algorithm finishes when the above evaluation is made to the root node of the *k*d-tree.

According to [Ben75, FBF77] the complexities of the *k*d-tree are, where d is a constant of proportionality that represents the dimensionality of a data set:

	Average Case	Worst Case
Space	$O(n)$	$O(n)$
Build	$O(d \times n \times \log n)$	$O(d \times n^2)$
Insert	$O(\log n)$	$O(n)$
Remove	$O(\log n)$	$O(n)$
Search	$O(\log n)$	$O(n)$

Figure 3.3: *K*d-tree Time and Space Complexities

3.2.2 *M*-Tree

The *M*-Tree is a paged, balanced, generic and dynamic secondary storage metric data structure for indexing a finite metric space. It partitions the objects in the metric space according to their distances and stores them in fixed *M* sized nodes. The distance between objects is evaluated by a user-defined distance function d. The tree is constituted by:

- Routing Objects (Representatives) - each routing object has the following format: <O_r,$ptr(T(O_r))$,$r(O_r)$,$d(O_r, P(O_r))$>, where: O_r is the routing object; $ptr(T(O_r))$ points to the root of a sub-tree, called the covering tree of O_r; $r(O_r)$ defines the radius from O_r to all the objects in the covering tree, *i.e.* the covering radius of O_r, requiring $r(O_r) > 0$; and $d(O_r, P(O_r))$ defines the distance from the routing object O_r to its parent $P(O_r)$, if it has one;

- Leaf Nodes - each leaf node has the following format: <O_j,$oid(O_j)$,$d(O_j, P(O_j))$>, where: O_j of the tree has a pointer to the actual object $oid(O_j)$, which is used to access the whole object that can possibly be in another data file; and $d(O_j, P(O_j))$ defines the distance from the leaf node O_j to its parent $P(O_j)$.

The *M*-Tree recursively organizes the objects in overlapping regions defined by a covering radius. The covering radius and the distance between one object and its parent are used to help prune the similarity queries in the *M*-Tree.

The following Figure 3.4 taken from [Pat99] samples a *M*-Tree on the $[0, 1]^2$ domain with the Euclidean Distance.

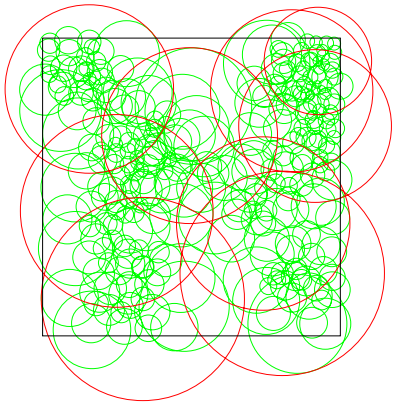


Figure 3.4: *M*-Tree Example

The algorithm of the *k*-nearest neighbours query of the *M*-tree retrieves the k-nearest neighbours of a query object q, assuming that there are at least k objects in the indexed metric space. It uses two global structures: a priority queue of active sub-trees that infer the order of evaluation and an array of k elements to store the result.

Each entry in the priority queue is a pair. The first field of the pair is a pointer to the root of a tree $T(O_r)$ and the second field is a lower bound d_{min} on the distance between q and any object in $T(O_r)$, that is:

$$d_{min}(T(O_r)) = max(d(O_r, q) - r(O_r), 0) \tag{3.10}$$

Since no object in $T(O_r)$ is at a distance from q less than $d(O_r, q) - r(O_r)$, then the priority queue returns the tree that has the minimum d_{min} value, because the goal is to find the objects that are closer to q, improving the efficiency of the algorithm.

Each entry of the nearest neighbours array has a pair where the first field denotes the nearer object and the second field represents the distance from the object to q. The algorithm starts by filling the array with null objects with a infinite distance from q and the idea is to compute for every sub-tree, an upper bound:

$$d_{max}(T(O_r)) = d(O_r, q) + r(O_r) \tag{3.11}$$

If between two sub-trees $T(O_{r1})$ and $T(O_{r2})$, the d_{max} of $T(O_{r1})$ is smaller than the d_{min} of $T(O_{r2})$, then the sub-tree $T(O_{r2})$ can be pruned from the search and the d_{max} values are inserted in the appropriate positions of the array.

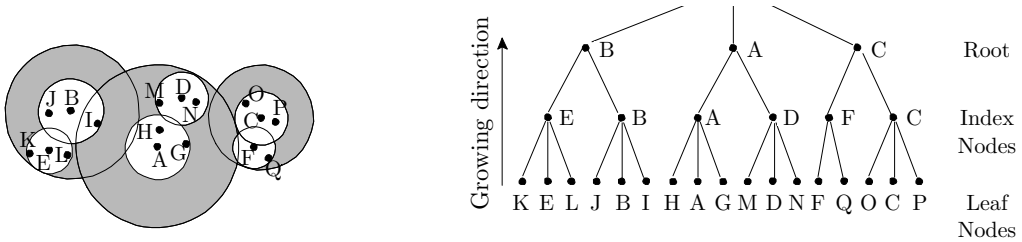
When it reaches a leaf node O_j, it checks if the node fits in the array. If it does, it uses the distance $d(O_j, q)$ to discard active sub-trees in the queue that have a d_{min} bigger than $d(O_j, q)$.

Until now, there are no publications about the remove operation [NUP11] in the *M*-Tree. So, regarding update operations, it only supports insertion.

The authors do not give theoretical results regarding the time complexities in the Big O Notation of the *M*-Tree, but experimental results with 10-nearest neighbours show search costs growing logarithmic with the number of objects [CPZ97].

3.2.3 DF-Tree

The DF-Tree (Distance Field Tree) is a paged, dynamic and generic secondary storage metric data structure for indexing a finite metric space. It is based on the Slim-Tree [TTFS02] and improves it by tackling the compromise of having a representative of a node selected from the representatives of its children, allowing more representatives to be used. On the other hand, the Slim-Tree is based on the *M*-Tree (3.2.2) and improves it by tackling the compromise where the representatives restrict where a new object can be stored, minimizing the intersection of nodes. The tree has the following structure (Figure 3.5) [TTFF02]:



(a) Regions (b) Tree

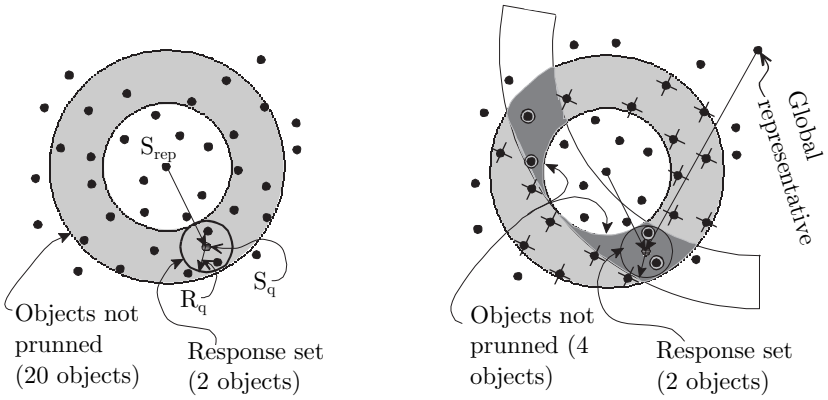
Figure 3.5: DF-Tree Example

- Internal Node - each node entry has the following format: <S_i, R_i, $d(S_i, S_{Ri})$, Df_i, $Ptr(T_{Si})$, $NEntries(Ptr(T_Si))$>, where: S_i is the representative object of the respective sub-tree pointed by $Ptr(T_{Si})$; R_i is the covering radius of the sub-tree;

$d(S_i, S_{Ri})$ is the distance between S_i and the representative of this node SRi; Df_i is the distance fields, *i.e.* the distance between the representative object and the global representative i; and $NEntries(Ptr(T_{Si}))$ is the current number of entries in the sub-tree;

- Leaf Node - each leaf node has the following format: <S_i, Oid_i, Df_i, $d(S_i, S_{Ri})$>, where: S_i is the actual object and its respective object identifier Oid_i; Df_i is the distance fields between the object and the global representative i; and $d(S_i, S_{Ri})$ is the distance between S_i and the representative of this node SRi.

The basic idea behind the DF-Tree is to build the tree combining the use of one representative per node (*e.g.* in the Slim-Tree) with the distance fields generated by each global representative. A global representative is chosen from the objects in the tree and is independent from the other representatives, including node representatives, and is applied to each object in the tree. The goal of using global representatives is to prune even more the number of distance evaluations. The number of distance evaluations that can be pruned depends on the areas defined by each representative object (Figure 3.6a, taken from [TTFF02]). Using the global representatives, this number can be pruned even more (Figure 3.6b).



(a) Without (b) With

Figure 3.6: DF-Tree Pruning using Global Representative

The number and placement of global representatives, unlike node representatives that are defined during the tree construction and, therefore, cannot be changed, can be calculated and modified when the first similarity query is answered or after the tree building. Given a dimensionality dim, the number of global representatives can be only $\lceil dim \rceil$. The DF-Tree has a method that evaluates if it currently needs to update its set of global representatives, so the set is not inadequate and does not compromise the pruning ability of the tree. If the tree needs to update its global representatives, there is also a method for this purpose.

As the DF-Tree is a descendent of the *M*-Tree, its algorithm for the *k*-nearest neighbours query is similar to the algorithm of the *M*-Tree.

4

Approach

This chapter explains how metric data structures help solving the problem and presents the architecture of the solution, justifying some choices made. Then, the primary storage solution is detailed and has its time complexity evaluated. By the end of this chapter, the secondary memory solution is also explained.

4.1 Introduction

4.1.1 Implementing the SNN using Metric Data Structures

Implementing the SNN with metric data structures, helps to solve the issues stated in 1.2:

- Storage - If we are handling with a big spatial data set that does not fit into primary storage, then it is necessary that the metric data structure chosen is stored in secondary storage. This is an important matter because, if the metric data structure is stored in secondary storage, the storage access times must be accounted to the runtime of the SNN. Therefore, the SNN should deal with both of these situations, handling a metric data structure in primary or secondary storage;

- Fine-tuning of Parameters - The attempts carried out to find the right parameters that give output to the best possible clustering results for a given data set, lead to multiple runs of the SNN.

  As stated before, the time complexity of the SNN in the worst case is $O(n^2)$. This quadratic behaviour is observed experimentally in chapter 5 with a referenced implementation of the Original SNN. The key factor for this time complexity is the

k-nearest neighbours queries for each object in a data set, since one single query is evaluated in the worst case with a time complexity of $O(n)$.

With these requirements, running the Original SNN multiple times can be time consuming. So, in [ESK03], metric data structures are briefly suggested to reduce the time complexity of the SNN. With such an improvement, running the Original SNN multiple times would become faster and less time consuming.

With the integration of metric data structures with the Original SNN, it is convenient to consider three types of runs (Figure 4.1): Tree, Neighbouring and Clustering (TNC) - the initial run, where it is necessary to build the metric data structure, query and store the *k*-nearest neighbours of each object, before carrying out the clustering steps of the SNN; Neighbouring and Clustering (-NC) - the run where the metric data structure is already built, but it still needs to query and store the *k*-nearest neighbours, before the clustering steps of the SNN; and Clustering (–C) - the run where the metric data structure is already built and the *k*-nearest neighbours are already known, for example, the *k* of the new run is less than or equal to the *k* of the previous run. As the *k* is less than or equal to the *k* previously calculated, the SNN can compute its clustering steps without querying the metric data structure again, only using part or all of the previously acquired *k*-nearest neighbours, requiring the *k*-nearest neighbours to be ordered ascendantly by their distances from the queried object.

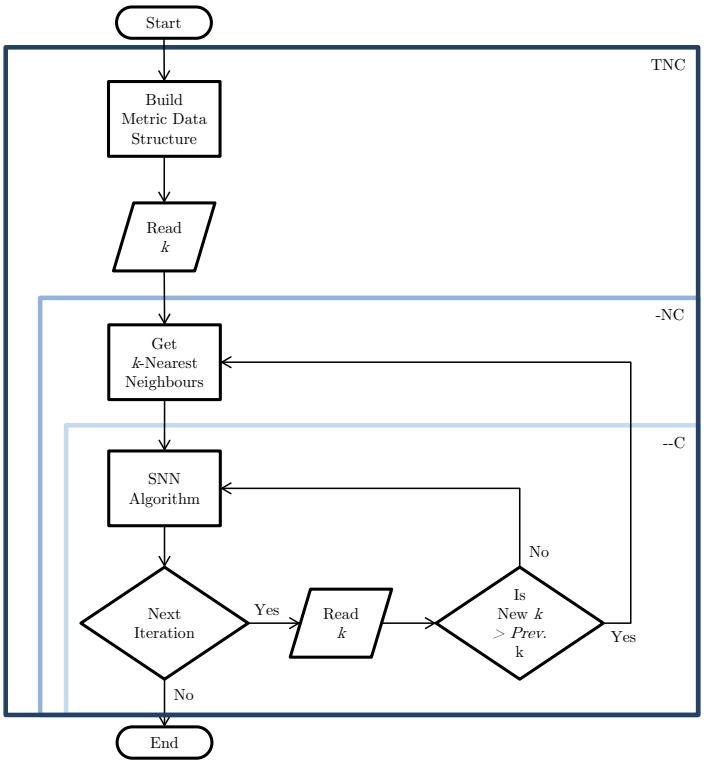


Figure 4.1: Runs Taxonomy

4.1.2 Architecture

The first step taken towards the architecture consisted on choosing which metric data structures to use with the SNN. As mentioned on chapter 3, the chosen metric data structures, for each type of storage, were: the *k*d-Tree and the DF-Tree. The *k*d-Tree for being popular in the literature [CNBYM01], for being written for primary storage in various programming languages, for having its time complexity evaluated in the Big O Notation and for being suited for low dimensionality problems (*e.g.* spatial data) [Mar06]. The DF-Tree because, there are few alternatives of secondary storage metric data structures and it delivers better results [TTFF02].

The next choice was in which programming language write the SNN. Since the *k*d-Tree is written in a wide range of programming languages, this metric data structure was not a bottleneck in the final verdict. On the other hand, as the DF-Tree is only written in C++ (part of the GBDI Arboretum library in [Dat06]), at first this was the primary choice. Nevertheless, because of Weka [HFH+09] being written in Java, because of having more knowledge and practice in Java and because of having found a *k*d-Tree in Java [Sav12], well-structured and well-documented, the SNN was written in Java.

Writing the SNN in Java, brought up an issue regarding how the SNN would get access to the *k*-nearest neighbours given by the DF-Tree, since the DF-Tree is only written in C++. The Java Native Interface (JNI) [Lia99] and the Protocol Buffers [Goo12] were considered. However, as JNI should only be used when there are no other alternative solutions that are more appropriate, since applications that use the JNI have inherent disadvantages [Lia99] and as Protocol Buffers seemed more risky at the time, representing a bumpier road, these two options were discarded. Therefore, the found solution was to use a DataBase Management System (DBMS) as a mean of connection between the SNN in Java and the DF-Tree in C++, without neglecting the access times that are demanded by the use of a DBMS. In the following experimental results, the MySQL DBMS was chosen with no particular reason, as every DMBS can be used.

In order to do this, when using the DF-Tree, it is deployed a Structured Query Language (SQL) file with all the attributes that define each object in a data set and their respective *k*-nearest neighbours, using identifiers for each object (explained in more detail in section 4.3.1). With this, the SNN implementation only works with identifiers during the clustering and does not need to store in primary storage the whole objects, since they are stored in secondary storage, requiring less primary storage space. When using the *k*d-Tree, the objects stay in the primary storage metric data structure, while the SNN continues to cluster with support to their identifiers.

Considering all the above decisions, the solution was designed according to the architecture in figure 4.2. There are two different modules:

- Primary Storage Module - represented by the left side of the figure. Everything works in primary storage and is written in Java. While using the *Reader*, it starts by reading each object from the data set, preprocessing it and giving to each object

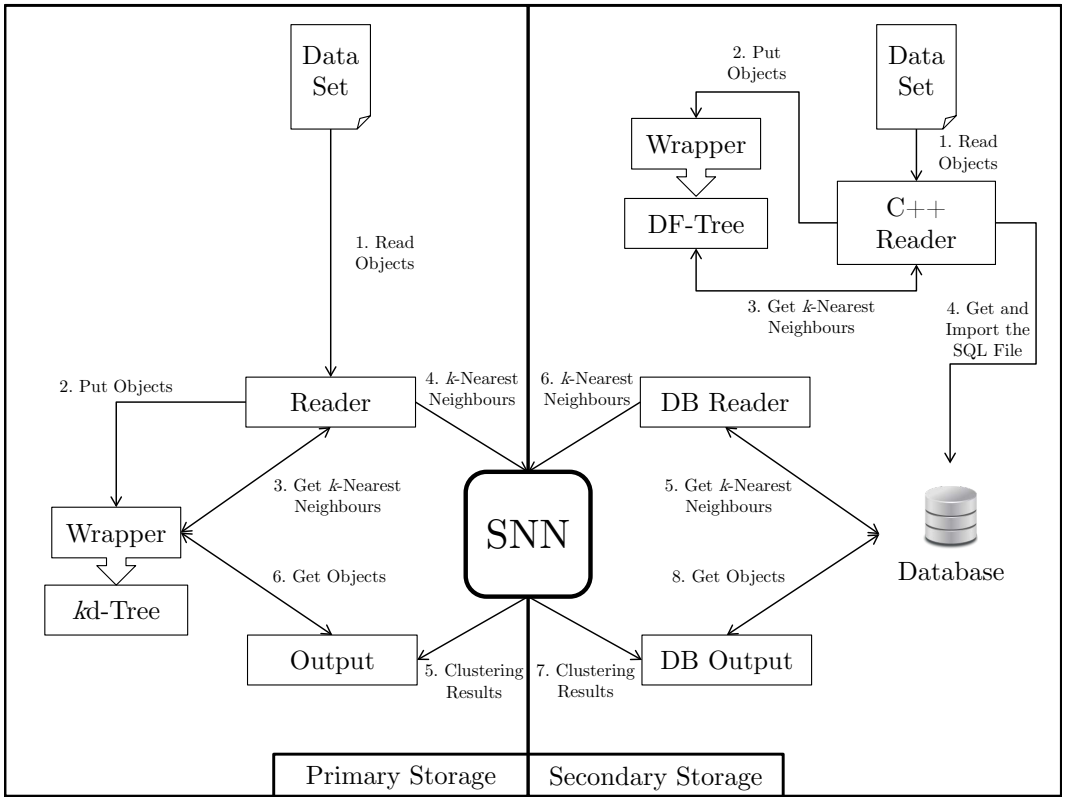


Figure 4.2: Architecture

an unique identifier, in order to store each object in the *k*d-Tree, using its respective wrapper. All the remaining components work with identifiers. Afterwards, the *Reader* gets the *k*-nearest neighbours of each object and gives them to the *SNN*, triggering the clustering process. When the *SNN* ends the clustering, the *Output* uses the identifiers of the objects to get their attributes from the *k*d-Tree, using also its respective wrapper, in order to generate a readable output file with the clustering results sent by the *SNN*.

- Secondary Storage Module - represented by the right side of the figure. The *C++ Reader* and the *DF-Tree* (the actual metric data structure and its wrapper) are written in C++. All other components are written in Java, working in primary storage. As the primary storage module, the secondary storage module begins by using the *C++ Reader* to read each object from the data set, preprocessing it and giving to each object an unique identifier, storing each object in the DF-Tree, using its respective wrapper. Then, the *C++ Reader* obtains the *k*-nearest neighbours from the DF-Tree and creates an SQL file to be imported to a DBMS with all objects and respective *k*-nearest neighbours. From this point forward, the *DB Reader* accesses the DBMS, in order to obtain the *k*-nearest neighbours and gives them to the *SNN* to start the clustering process. When the clustering is achieved, the *DB Output* uses the identifiers of each object to query the DBMS for the attributes of each of the objects, creating also a readable output file with the clustering results sent by the *SNN*.

Both modules use wrappers in the metric data structures to create a level of transparency to the *Readers* or the *Outputs*, so they do not need to be changed, when used with a different metric data structures. Each of these two modules are explained in more detail in the following sections.

4.2 Using the *k*d-Tree with the SNN

On this section, this approach is explained and the improvement that can be obtained by using the *k*d-Tree with the SNN is evaluated, regarding its time complexity, by making the *k*-nearest neighbours queries more efficient. The main procedure written for this approach is summed up in Algorithm 1. This approach will be known in this document by *k*d-SNN.

Algorithm 1 Primary Storage Procedure

```
1:  function kD-SNN
2:      while making consecutive runs do
3:          if first run then
4:              use the Reader to build the kd-Tree with the objects in a data set
5:          end if
6:          if first run or new k > previous k then
7:              use the Reader to obtain the k-nearest neighbours
8:          end if
9:          SNN(k-nearest neighbours)
10:         if output is requested then
11:             use the Output
12:         end if
13:     end while
14: end function
```

4.2.1 Reader

As disclosed in the architecture, the *Reader* is used to read a data set and build the *k*d-Tree with all objects, in order to obtain the *k*-nearest neighbours of each object. Each *Reader* is written for a specific format of a data set and, when a new *Reader* needs to be written, it has to follow an appropriate interface with the required methods to read the data set, build the tree and query for the *k*-nearest neighbours of each object.

On the first run of the algorithm (TNC), the *k*d-Tree needs to be built with a data set. The time complexity in the average case of building the *k*d-Tree with low dimensionality objects is $O(d \times n \times \log n)$ [FBF77], where d is the dimensionality of a data set. This low dimensionality scenario is often witnessed when dealing with spatial data, even when the distance function used is not simply the geographical distance between objects. For instance, in [SSMPW12], it is used a distance function that, besides using the geographic location, also uses the bearing (heading) of a given motion vector, to measure the distance

between objects. Under these circumstances, it can be asserted that the time complexity in the average case of building the *k*d-Tree is proportional to $O(n \times \log n)$.

Once the *k*d-Tree is built, the next step is to query it for the *k*-nearest neighbours of each object. The resulting *k*-nearest neighbours and their respective distances are stored in a Hash Map that requires the sets of *k*-nearest neighbours to be ordered ascendantly by the distance between objects. The time complexity in the average case of one single *k*-nearest neighbours query is proportional to $O(\log n)$, where the constant of proportionality depends on the distance function, on the dimensionality of a data set and on the *k* value. Therefore, the time complexity in the average case of obtaining the *k*-nearest neighbours of all objects in a data set is also proportional to $O(n \times \log n)$.

After obtaining the *k*-nearest neighbours of each object, they are provided to the SNN, triggering the beginning of the clustering.

4.2.2 SNN

The SNN implemented was based on the representative objects-based approach explained in section 2.2.3 and has several clustering steps as in Algorithm 2.

Algorithm 2 Procedure of the SNN

Require: A Hash Map with the *k*-nearest Neighbours of each Object
1: **function** SNN
2: MEASURING THE SIMILARITIES AND THE DENSITIES
3: NOISE DETECTION
4: CLUSTERING THE REPRESENTATIVE OBJECTS
5: CLUSTERING THE REMAINING OBJECTS
6: CLUSTERING THE NOT YET CLUSTERED OBJECTS
7: **end function**

As soon as the SNN obtains the *k*-nearest neighbours of each object, it initiates the clustering process, starting by measuring the similarity between objects and their respective densities (Algorithm 3). The most complex step is to compute the similarity between a pair of objects, since the SNN intersects their sets of *k*-nearest neighbours to determine the number of shared nearest neighbours. This intersection has a time complexity in the worst case evaluated in $O(k^2)$, corresponding to going over one of the sets of *k*-nearest neighbours for every nearest neighbour in another set. The calculated similarities between mutual neighbours should be stored in a Hash Map, so they can be reused in the following steps of the SNN, avoiding the time complexity of recalculating the similarities. Since, the SNN needs to iterate over all objects, intersecting a set of *k*-nearest neighbours of an object with the set of *k*-nearest neighbours of each of its neighbours, which as asserted before has a time complexity of $O(k^2)$, algorithm 3 is evaluated with a time complexity in the worst case of $O(n \times k \times k^2) \iff O(n \times k^3)$.

Now that the density of each object is known, the noise must be detected (Algorithm 4). This algorithm initially requires all objects to be defined as noise. It starts by iterating

Algorithm 3 Measuring the Similarities and the Densities

```
1: function MEASURING THE SIMILARITIES AND THE DENSITIES
2:     for all objects do
3:         density of the object ← 0
4:         get the k-nearest neighbours of the object from the Hash Map
5:         for all k-nearest neighbours of the object do
6:             if the similarity between the object and its neighbour is not calculated then
7:                 if the object and its neighbour are mutual neighbours then
8:                     measure the similarity between the object and its neighbour
9:                 else
10:                    similarity between the object and its neighbour ← 0
11:                end if
12:            end if
13:            if similarity between the object and its neighbour ≥ Eps then
14:                density of the object ← density of the object + 1
15:            end if
16:        end for
17:    end for
18: end function
```

over all objects, to identify the ones that are representative. When an object is representative, it goes through all of its neighbours, to define the ones that are noise or not. If a non representative neighbour is in a radius of *Eps* of the current representative object, it is declared as non noise. Besides this, wherever an object is representative, it is added to a representative objects graph, represented by a Hash Map, used to cluster the representative objects. These operations lead algorithm 4 to a time complexity in the worst case of $O(n \times k)$.

Algorithm 4 Noise Detection

```
Require: All Objects Defined as Noise
1: function NOISE DETECTION
2:     for all objects do
3:         if the object is representative then
4:             define the object as non noise
5:             add the object to the representative objects graph
6:             get the k-nearest neighbours of the object from the Hash Map
7:             for all k-nearest neighbours of the object do
8:                 if the neighbour is non representative then
9:                     if the distance between the object and its neighbour is ≤ Eps then
10:                        define the object as non noise
11:                    end if
12:                end if
13:            end for
14:        end if
15:    end for
16: end function
```

Afterwards, the clustering is initiated, starting with the representative objects, then the remaining objects (non representative and non noise) and ending with the not yet clustered objects (Algorithm 2).

Algorithm 5 Clustering the Representative Objects

```
1:  function CLUSTERING THE REPRESENTATIVE OBJECTS
2:      for all objects do
3:          if the object is representative then
4:              get the k-nearest neighbours of the object from the Hash Map
5:              for all k-nearest neighbours of the object do
6:                  if the neighbour is representative then
7:                      if the similarity between the object and its neighbour is ≥ Eps then
8:                          connect the object and its neighbour in the graph
9:                      end if
10:                 end if
11:             end for
12:         end if
13:     end for
14:     identify the clusters obtained by finding the connected sub-graphs in the graph
15: end function
```

The most important algorithm is the clustering of the representative objects, since it defines the backbone of the clusters (Algorithm 5). For that, the SNN iterates over all objects, connecting in the graph all representative objects and respective representative *k*-nearest neighbours that have a similarity greater than *Eps*. Connecting the objects takes the time complexity of checking if the connection already exists. Since, each object as at most *k* connections, connecting a pair requires a $O(k)$ time complexity in the worst case, leading this step to a time complexity in the worst case of $O\Big(n \times k \times O(k)\Big) \iff O(n \times k^2)$.

After iterating all and connecting the possible objects, the SNN finds all maximally connected components [Lov04] of the graph, that represent the clusters, using the breadth first search. The breadth first search has a time complexity in the worst case of $O(V + A)$ [WCC09], where V represents the number of vertices in the graph and A the number of arcs (connections) in the graph. Since, in this case, the number of vertices is at most the number of objects and the number of connections is at most the number of objects times the number of *k*-nearest neighbours, finding all maximally connected components of the graph takes in the worst case $O\Big(n + (n \times k)\Big) \iff O(n \times k)$.

So, algorithm 5 has a time complexity in the worst case of $O(n \times k^2)$.

After obtaining the backbone of the clusters, all remaining objects are clustered to the cluster of the representative object with which they are more similar (Algorithm 6).

In order to find the most similar representative object, the *Eps* threshold is not used to break connections between mutual neighbours. The idea is to cluster the the non representative and non noise objects to a cluster of a representative neighbour with which it is a mutual neighbour and shares the highest similarity possible among its representative neighbours, even if the highest similarity is zero and they are just connected, because

Algorithm 6 Clustering the Remaining Objects

```
1: function CLUSTERING THE REMAINING OBJECTS
2:     for all objects do
3:         if the object is non representative and non noise then
4:             get the k-nearest neighbours of the object from the Hash Map
5:             for all k-nearest neighbours of the object do
6:                 if the neighbour is representative then
7:                     check if this is the most similar representative neighbour
8:                 end if
9:             end for
10:            if found a most similar representative neighbour then
11:                the object joins the cluster of the neighbour
12:            else
13:                find the predominant cluster between its neighbours
14:                if found a predominant cluster between its neighbours then
15:                    the object joins the predominant cluster
16:                else
17:                    define the object as not yet clustered
18:                end if
19:            end if
20:        end if
21:    end for
22: end function
```

both objects are mutual neighbours.

In case they do not have a representative object as a neighbour, their neighbours are evaluated, with the goal of finding a predominant cluster between them. The predominant cluster step was introduced in this implementation with the purpose of giving the remaining objects, that do not have a representative object as their neighbour, a cluster. In order to do this, all the neighbours of a non representative object are evaluated and the most common cluster between them is chosen as the predominant cluster. However, as the clustering of the remaining objects might not yet be completed, while the neighbours are being checked for a predominant cluster, some might not have a predominant cluster among its neighbours, because some of these neighbours might be remaining objects too and were not processed yet. Objects in this situation are defined as not yet clustered and will be clustered in the next step of the SNN.

Finding the predominant cluster among the neighbours has a time complexity in the worst case that depends on two operations: the clustering scoring, in order to identify the most predominant cluster, which has a time complexity in the worst case of $O(k)$; and the looking over all the scored clusters, for the one with the most occurrences, which has a time complexity of $O(k)$ in the worst case. Therefore, finding the predominant cluster has a time complexity in the worst case of $O\Big(O(k) + O(k)\Big) \iff O(k)$.

This way, for each remaining object, either a representative object is found among its neighbours, or the predominant cluster among its neighbours needs to be found, bringing

the time complexity in the worst case of algorithm 6 to $O\Big(n \times (k + O(k))\Big) \iff O(n \times k)$.

The last algorithm gives all not yet clustered objects, one last chance of being clustered by being clustered to the predominant cluster among its *k*-nearest neighbours (Algorithm 7).

Algorithm 7 Clustering the Not Yet Clustered Objects

```
1:  function CLUSTERING THE NOT YET CLUSTERED OBJECTS
2:      for all not yet clustered objects do
3:          find the predominant cluster between its neighbours
4:          if found a predominant cluster between its neighbours then
5:              the object joins the predominant cluster
6:          else
7:              define the object as noise
8:          end if
9:      end for
10: end function
```

As pointed out in the previous algorithm (Algorithm 6), the time complexity in the worst case of finding the predominant cluster among the *k*-nearest neighbours of an object is $O(k)$. Since, all not yet clustered objects need to be iterated and for each one finding its predominant cluster, algorithm 7 has a time complexity in the worst case of $O(m \times k)$, where m denotes the number of not yet clustered objects, that were not clustered in the previous algorithms of the SNN.

Algorithms	Worst Case
Measuring the Similarities and the Densities	$O(n \times k^3)$
Noise Detection	$O(n \times k)$
Clustering the Representative Objects	$O(n \times k^2)$
Clustering the Remaining Objects	$O(n \times k)$
Clustering the Not Yet Clustered Objects	$O(m \times k)$

Figure 4.3: Time Complexities of the Algorithms of the SNN

Considering all the time complexities evaluated (Figure 4.3), it follows that the time complexity in the worst case of the clustering steps of the SNN is $O(n \times k^3)$, which is proportional to $O(n)$.

4.2.3 Output

The *Output* is optional and can be done in various formats, they just need to be written according to an available interface, not restricting the user to some formats. Nevertheless, two types of outputs were implemented, since Weka was used as a plot tool: the format of an Attribute-Relation File Format (ARFF) file supported by Weka or plotted on a chart also with support to Weka.

An ARFF file is a standard way of representing data sets that consist of independent, unordered instances of objects and does not involve relationships between instances [WF05].

```
%Example of an ARFF file
@relation arff_example

@attribute X numeric
@attribute Y numeric
@attribute cluster {1, 2, 3}

@data
-8.667331, 41.149643, 1
-8.66603, 41.153057, 1
-8.663754, 41.146879, 1
-8.579538, 41.201668, 2
-8.659527, 41.150455, 1
-8.57466, 41.192401, 2
-8.575636, 41.198254, 2
-8.582302, 41.182484, 2
-8.630425, 41.146716, 3
-8.627662, 41.146554, 3
-8.579863, 41.191588, 2
-8.585716, 41.185735, 2
```

Figure 4.4: Example of an ARFF file

Figure 4.4 is an example of an ARFF file, where % represent the comments. Following the comments, there is the name of the data set (*@relation*) and, the attributes used in the data set are defined (*@attribute*), in this case, X, Y and *cluster*. Nominal attributes are defined also by the set of values that they can take, for example, *cluster*. Numeric values are followed by the keyword *numeric*. After the attributes definitions, *i.e.* following the *@data* line, the objects are finally represented, one by each line. The values of each attribute of the object are separated by commas. Although the *cluster* attribute is the predicted value, in an ARFF file there is no reference to what attribute is the class attribute.

Since the SNN only uses identifiers, the clustering results are also presented with identifiers. So, in order to give an appropriate output and getting all the information needed for the output format, it is required to get the attributes of the objects. In order to do this, the *Reader* must provide access to an iterator to the *k*d-Tree, giving access to all the attributes of the objects.

4.2.4 Time Complexity Evaluation

Bearing in mind that besides the above clustering steps required by the SNN it is also needed to index the metric data structure and get the *k*-nearest neighbours of each object in the data set, it follows that the time complexity of the *k*d-SNN depends solely on the time complexity of building the *k*d-Tree, of the *k*-nearest neighbours queries done to the *k*d-Tree and of the clustering steps of the SNN. Since the *k*d-Tree is built in a $O(n \times \log n)$ time complexity in the average case and is queried for the *k*-nearest neighbours of each

object also in a $O(n \times \log n)$ time complexity in the average case [Ben75, FBF77], then the time complexity of the TNC run and the -NC run is of at most $O(n \times \log n)$, since it is bigger than the $O(n)$ time complexity in the worst case of the clustering steps of the SNN.

As the time complexity in the worst case of the clustering steps of the SNN is $O(n)$, it represents the time complexity in the worst case of the –C run. Consequently, the time complexity of the *k*d-SNN depends on each run (Figure 4.5).

Run	Average Case	Worst Case
TNC	$O(n \times \log n)$	-
-NC	$O(n \times \log n)$	-
–C	-	$O(n)$

Figure 4.5: Time Complexities of the Runs of the *k*d-SNN

4.3 Using the DF-Tree with the SNN

On this section, it is explained the approach taken when using the DF-Tree with the SNN. As disclosed in the related work, the DF-Tree is a descendant of the *M*-Tree, which has experimental results where 10-nearest neighbours were obtained and it showed a query cost growing logarithmic with the number of objects. However, as the DF-Tree does not have its own time complexities documented, the integration of the DF-Tree with the SNN will not have its time complexity evaluated.

Algorithm 8 Secondary Storage Procedure

```
1:  function SECONDARY STORAGE
2:      while making consecutive runs do
3:          if first run or new k > number of neighbours in the DBMS then
4:              use the C++ Reader to obtain the k-nearest neighbours
5:              import the deployed SQL file to the DBMS
6:          end if
7:          if first run or new k > previous k then
8:              use the DB Reader to get the k-nearest neighbours from the DBMS
9:          end if
10:         SNN(k-nearest neighbours)
11:         if output is requested then
12:             use the DB Output
13:         end if
14:     end while
15: end function
```

As asserted in the architecture, the SNN gets access to the *k*-nearest neighbours given by the DF-Tree through a DBMS. So, when using the DF-Tree, the procedure goes according to the algorithm 8. This approach will be known in this document by DF-SNN.

4.3.1 C++ Reader

The *C++ Reader* is in charge of building the DF-Tree with a data set, of querying it for the *k*-nearest neighbours and of deploying a SQL file with all the attributes that define the objects and their respective *k*-nearest neighbours.

Although it is possible to reuse several methods written, for instance, the one that gives the output in the format of a SQL file, for each format of a data set, a *C++ Reader* has to be written, as in the primary storage approach with the *k*d-Tree.

Since the DF-Tree works in secondary storage, the DF-Tree is stored in a DAT file with a specific format created by the authors of the DF-Tree, resorting to serialize and deserialize methods written specifically for the type of objects in a data set.

After the DF-Tree is built, it becomes capable of answering the *k*-nearest neighbours queries. The results are then given in the format of a SQL file that creates two tables with the following format:

- Objects Table - each row consists of the identifier (ID) of an object and all the attributes that define the respective object in a data set;

ID	...
0	...
⋮	⋮
$n-1$	...

Figure 4.6: SQL File - Objects Table

- *k*-Nearest Neighbours Table - each row represents the identifier of an object, the identifier of one of its *k*-nearest neighbours and the distance between them. This row is repeated *k* times for each object.

ID	*k*-Nearest Neighbour ID	Distance
0	...	...
⋮	⋱	⋮
$n-1$	...	...

Figure 4.7: SQL File - *k*-Nearest Neighbours Table

4.3.2 DB Reader

After acquiring the SQL file from the *C++ Reader*, it is imported to a DBMS, so the SNN can gain access to the *k*-nearest neighbours.

The *DB Reader* was created for this purpose and is generic for any format of a data set, since its goal is just to read the standard *k*-nearest neighbours table stored in a DBMS, as

it is the same for all data sets. In order to do this, it requires a DBMS connection interface written in Java for the DBMS.

When the *k*-nearest neighbours are all in primary storage, the SNN can start its clustering progress. Bear in mind that, since the *k* value is limited to the number of neighbours stored in the DBMS, if there is a need to use more *k*-nearest neighbours than the ones stored in the DBMS, then the *C++ Reader* has to be run again, to get the SQL file with all the new sets of *k*-nearest neighbours.

Due to technical difficulties with the use of the DF-Tree, every time the C++ *Reader* is run again, the tree needs to be built and cannot be reused between runs. So, in the context of the *C++ Reader*, every time it is run, it matches the TNC run, since the -NC run does not exit. However, in the primary storage component, the -NC run exists, since we can query the DBMS for more neighbours than we initially had requested, as long as the number of neighbours does not surpass the number of neighbours stored in the DBMS. If we do not need to query for more neighbours, either the *C++ Reader* or the DBMS, the run –C is executed.

The SNN used in the DF-SNN is the same presented in section 4.2.2, because there is a care to read the *k*-nearest neighbours in the format required by the SNN written, fostering the portability of the SNN.

4.3.3 DB Output

As the SNN ends its clustering and produces the respective clustering results, the output may be given via the *DB Output*. The difference between the *DB Output* and the *Output*, is the source of information. As the *Output* in the *k*d-SNN uses the identifiers to make the match with the objects given by the iterator, the DF-SNN uses the identifiers to get the information directly from the DBMS. Everything else is exactly the same, the *DB Reader* is written to give output in an ARFF file or plotted on a chart with support to Weka. Other types of output are also supported, they just need to be written according to an interface available.

The *DB Output* requires, as the *DB Reader*, a connection to the DBMS used by the DF-SNN.

5

Experimental Results

This chapters discloses the experimental results, where three synthetic data sets and one real data set, entitled Marin Data, were used, in order to evaluate the quality of the clustering results obtained with the *k*d-SNN and with the DF-SNN. Each of these results is compared with the ones obtained with another implementation of a variant of the SNN. Another real data set, extracted from Twitter, was used to evaluate the scalability of the *k*d-SNN and of the DF-SNN. Their scalability is also compared with the scalability of a referenced implementation of the Original SNN.

5.1 Data Sets

5.1.1 Synthetic

Three synthetic data sets were used, each of them has objects defined by two attributes that represent the objects position in a two-dimensional space (Figure 5.1). The distance between the objects is measured with the generic euclidean distance over a two-dimensional space.

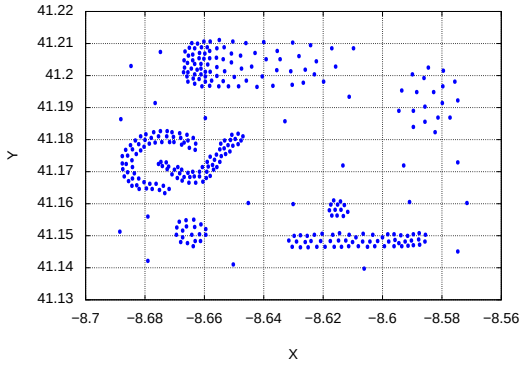
Attribute	Type
X	Decimal Number
Y	Decimal Number

Figure 5.1: Attributes of the Synthetic Data Sets

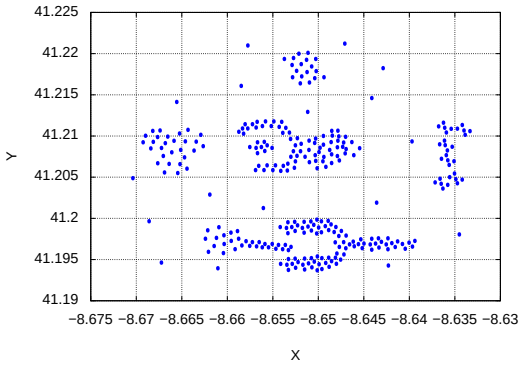
These three datasets were chosen due to their usage in the literature and because two of them were used in experimental results to validate the representative objects-based

approach of the SNN [MSC05], serving as benchmark.

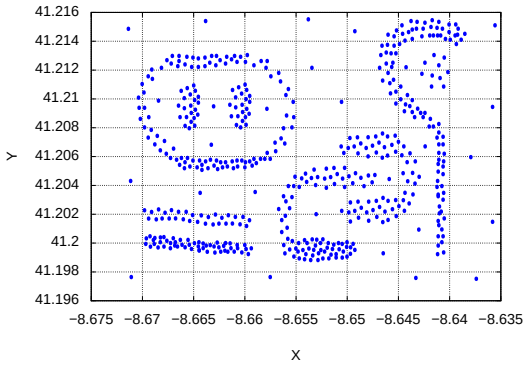
As can be seen in figure 5.2, each data set has a unique and specific clustering tendency, where the main clusters can be easily perceived by visually analysing the distribution of the objects in the data set.



(a) Synthetic 1



(b) Synthetic 2



(c) Synthetic 3

Figure 5.2: Distribution of the Synthetic Data Sets

The first synthetic data set (Figure 5.2a) has 322 objects, the second (Figure 5.2b) has 292 objects and the third and last (Figure 5.2c) has 523 objects. Some of these objects clearly represent noise, as can be seen in the data sets respective figures.

5.1.2 Marin Data

This data set was picked up due to the work already done in the area of clustering spatial data with the SNN [SSMPW12]. It is defined by 315794 objects that represent the position of a ship at a certain time. These objects were obtained with a time interval of 60 seconds. Each object has 29 attributes (a sub set of its attributes is shown in Figure 5.3), where only the geographical location (latitude and longitude) and the bearing (heading) of the ship were used to measure the distance between objects.

Attribute	Type
Ship Type Identifier	Integer
Latitude	Decimal Number
Longitude	Decimal Number
Bearing	Integer

Figure 5.3: Subset of Attributes of the Marin Data Set

The distance function used is specific to this data set, being more appropriate to this type of spatial data, since it needs to deal with the bearing of the motion vectors that represent the traffic routes:

$$Function(p_1, p_2) = w \times \left(\frac{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}{mDist} \right) + (1 - w) \times \left(\frac{\Phi(|b_1 - b_2|)}{mBearing} \right) \quad (5.1)$$

, with

$$\Phi(|b_1 - b_2|) = \begin{cases} |b_1 - b_2| & , |b_1 - b_2| \leq 180^{\text{o}} \\ 360^{\text{o}} - |b_1 - b_2| & , |b_1 - b_2| > 180^{\text{o}} \end{cases} \quad (5.2)$$

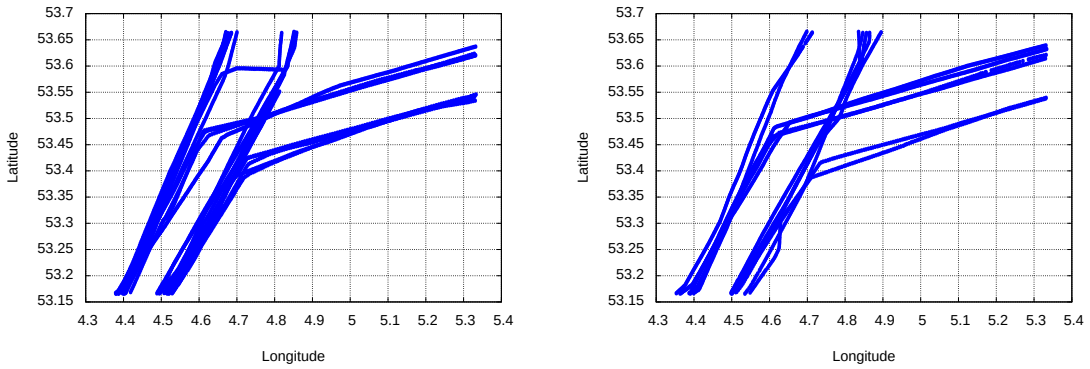
, where w is the weight assigned to the location, $(1-w)$ is the weight assigned to the bearing, $mDist$ is the maximum distance between a pair of objects in the data set, $mBearing$ is a constant value of 180^{o} and p_1 and p_2 represent two objects, with each object having x representing its latitude, y representing its longitude and b representing its bearing.

Since this data set is defined by several types of ships (Figure 5.4), representing different paths taken by the ships, only two of these types were used, working as two different data sets: LPG and Oil (Figure 5.4). There are 4168 objects of the ship type LPG and 2640 having a Oil ship type.

Ship Type Description	Ship Type Identifier	$mDist$ Value
Oil	$13, 14, 15, 16, 17, 18$	83467
LPG	$20, 21, 22, 23$	82039

Figure 5.4: Subset of Ship Types in the Marin Data Set

The LPG ship type was chosen, because it is the one used in [SSMPW12], that will serve as a benchmark. On the other hand, the Oil ship type was chosen, since it is one of the types used in the full experimental results carried out and provided in a document by the same authors. Figure 5.5 shows the spatial distribution of the objects for each ship type data set. Regarding the MARIN data set clustering results, both ship types had $w = 0.90$. The clustering tendency is the distinct traffic routes taken by the ships.



(a) LPG Ship Type (b) Oil Ship Type

Figure 5.5: Spatial Distribution of the Marin Data Set

5.1.3 Twitter

The Twitter data set was extracted from Twitter, using the available Application Programming Interface (API) [Twi12]. The extraction was made on several days of December 2011, of January 2012 and of February 2012.

Attribute	Type
Latitude	Decimal Number
Longitude	Decimal Number
Number of Days	Decimal Number

Figure 5.6: Attributes of the Twitter Data Set

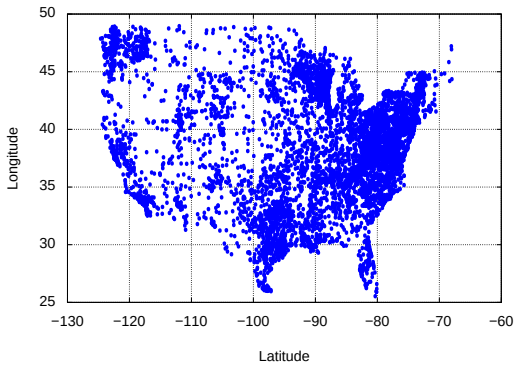


Figure 5.7: Spatial Distribution of the Twitter Data Set

It has 512000 objects defined by three attributes (Figure 5.6), representing the geographical location of a tweet and, taking into consideration the date on which a tweet was performed, the number of days that have passed since 1 December 2011. There are

8001 different geographical locations (Figure 5.7). The distance function used was also the generic euclidean distance over the three attributes.

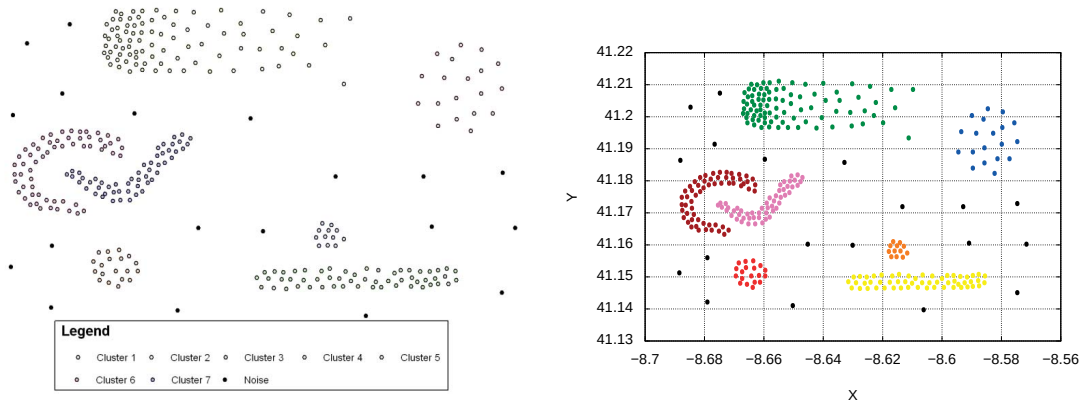
This spatio-temporal data set was chosen in order to evaluate the scalability gain of the *k*d-SNN and of the DF-SNN, since it has a meaningful number of objects to be clustered.

5.2 *k*d-SNN

5.2.1 Evaluation of the Quality of the Clustering Results

This evaluation starts with the clustering results obtained when applying the *k*d-SNN over the synthetic data sets. Two of these results, Synthetic 1 (Figure 5.2a) and Synthetic 2 (Figure 5.2b), are compared with the results in [MSC05] obtained with a variant of the SNN. The Synthetic 3 (Figure 5.2c) has no benchmarks, since no clustering results obtained with the SNN were found in the literature. However, as the clustering tendency can be visually perceived, the Synthetic 3 was also analysed.

All the presented clustering results have their clusters identified by a unique color and their noise represented by the black color, except for the Marin data set clustering results, where the noise is not shown.

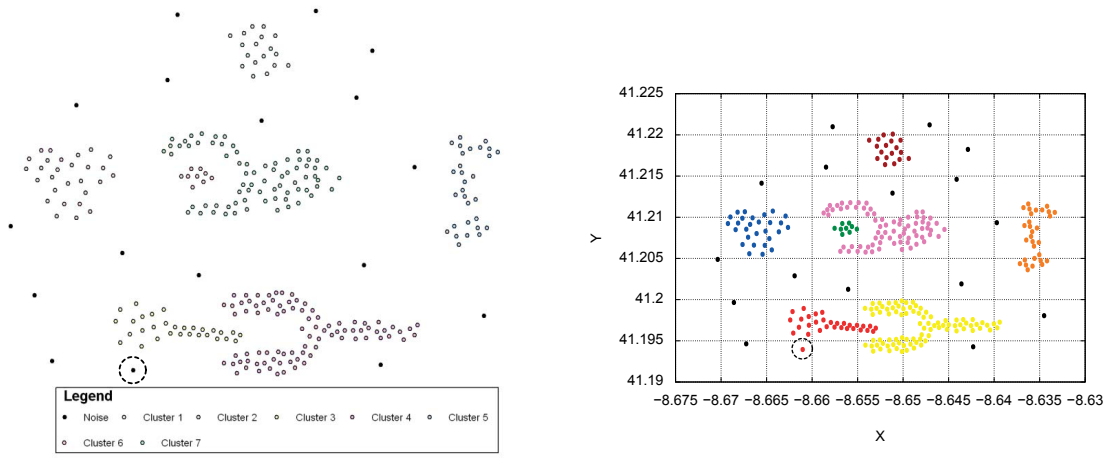


(a) Benchmark [MSC05] (b) *k*d-SNN

Figure 5.8: Clustering Results of the Synthetic 1 Data Set (Evaluation of the *k*d-SNN)

The clustering results for Synthetic 1 are presented in figure 5.8. On figure 5.8a is shown the benchmark with $k = 7$, $Eps = 0.3 \times k$, $MinPts = 0.7 \times k$ and on figure 5.8b is shown the clustering results obtained with the *k*d-SNN with $k = 7$, $Eps = 2$ and $MinPts = 5$. As can be seen, both implementations returned the same results for the Synthetic 1 data set.

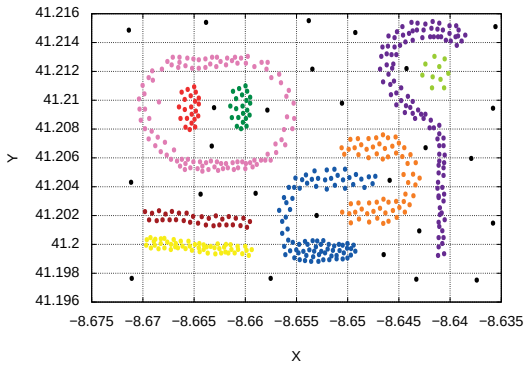
On the other hand, when comparing the benchmark of the Synthetic 2 (Figure 5.9a), having $k = 7$, $Eps = 0.3 \times k$, $MinPts = 0.7 \times k$, with the clustering results obtained with the *k*d-SNN (Figure 5.9b) having $k = 7$, $Eps = 2$ and $MinPts = 5$, there is one slight difference.



(a) Benchmark [MSC05] (b) *k*d-SNN

Figure 5.9: Clustering Results of the Synthetic 2 Data Set (Evaluation of the *k*d-SNN)

There is one object that was not defined as noise and was clustered, while on its respective benchmark it was defined as noise (object inside the dotted circle). It seems that the object had a density value below the density threshold and it was not identified as noise (remaining object). However, it was clustered, because it had a clustered representative neighbour with which it shared a higher similarity.



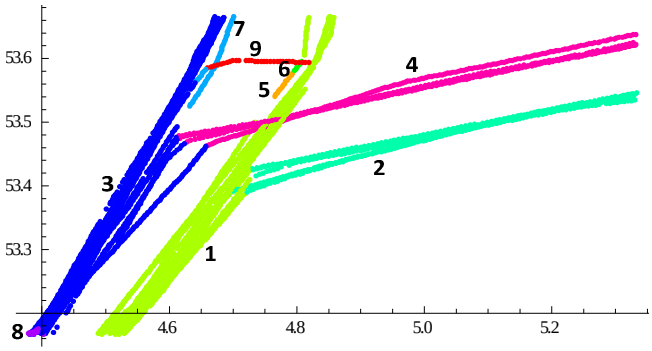
(a) *k*d-SNN

Figure 5.10: Clustering Results of the Synthetic 3 Data Set (Evaluation of the *k*d-SNN)

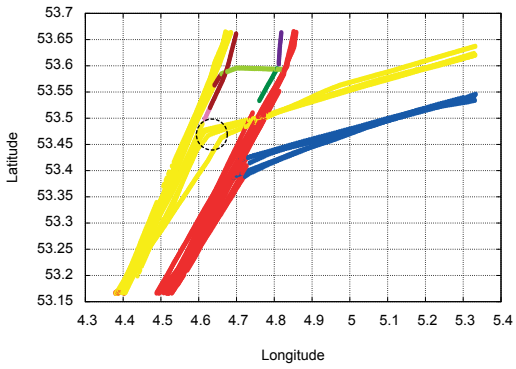
Differences between some clustering results and their respective benchmarks, may be due to some minor differences between the implemented variants of the SNN, for instance: what to do when there is no representative objects among its neighbours? In this approach, the SNN seeks to find a predominant cluster for the object without a representative neighbour. What if there are two or more most similar representative objects among its neighbours and only one can be chosen? In this approach, the first one found

is the one chosen. It can also be due to different *k*-nearest neighbours results, for example, two or more neighbours may share the same distance from the queried object and not all fit in the final set of *k*-nearest neighbours, and on each implementation a different neighbour makes it to the final set of *k*-nearest neighbours, which in this approach would depend on the tiebreaker used by the metric data structure.

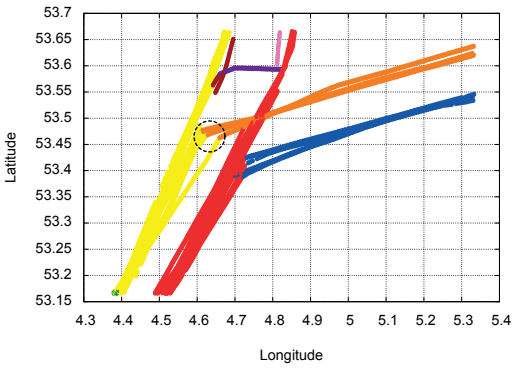
When dealing with the Synthetic 3, the clustering results (Figure 5.10a) obtained with $k = 7$, $Eps = 2$ and $MinPts = 5$ are the expected, as they match the clustering tendency of the data set. These clustering results were not compared with a benchmark, since no results for this data set were presented in [MSC05] and were not found in the literature using the Original SNN or one of its variants.



(a) Benchmark [SSMPW12]



(b) *k*d-SNN

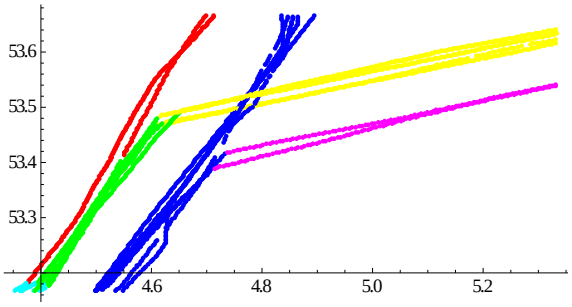


(c) *k*d-SNN

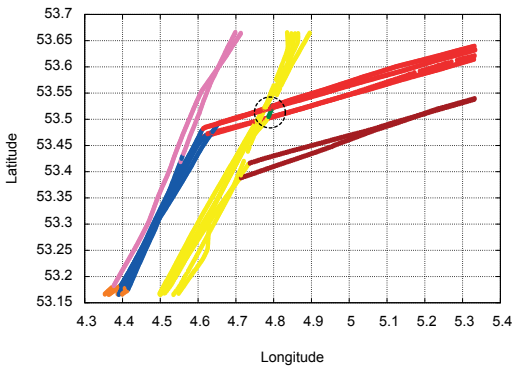
Figure 5.11: Clustering Results of the LPG Ship Type (Evaluation of the *k*d-SNN)

Following, the clustering results of the Marin data set are presented, starting with the LPG ship type and, then, the Oil ship type. Despite having the same clustering structure, the clustering results are not exactly the same, for both ship types. Not just some objects ended up in different clusters, but the user-defined parameters were also slightly adjusted, trying to obtain more similar results to the respective benchmark. The reasons behind these differences may be the ones pointed out before, for the synthetic data sets, since the reasons are tied to the written implementation of the variant of the SNN and not to the data sets themselves.

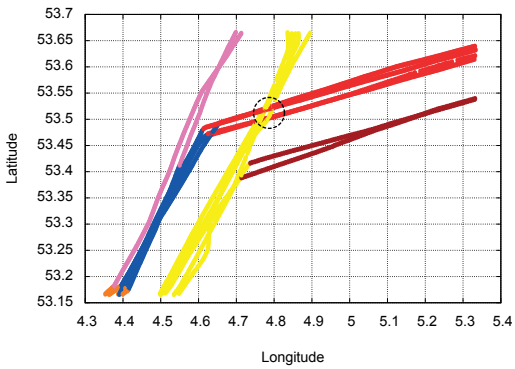
When using the same values for the user-defined parameters of the SNN as in the benchmark (Figure 5.11a, having k = 12, *Eps* = 3 and *MinPts* = 7), the clustering results had one main difference. See figure 5.11a for the benchmark results and figure 5.11b for the *k*d-SNN results with the same values as the benchmark. Route number 3 and 4 in the benchmark were just one cluster (yellow cluster) in the *k*d-SNN results. The cluster should have been split, since the heading of the ships changed. When the *MinPts* was changed to 8 (Figure 5.11c), the route was split (yellow and orange clusters), leading to more similar results. They still have minor differences, for instance, noise points that are not defined as noise in the benchmark, but the main routes were identified.



(a) Benchmark



(b) *k*d-SNN



(c) *k*d-SNN

Figure 5.12: Clustering Results of the Oil Ship Type (Evaluation of the *k*d-SNN)

With the Oil ship type, the clustering structure of the clustering results obtained with the *k*d-SNN was practically equal to the benchmark. See figure 5.12a for the benchmark, taken from the document containing the clustering results provided by the authors, and see figure 5.12b for the clustering results of the *k*d-SNN, having k = 12, *Eps* = 3 and *MinPts* = 7, the same values as in the benchmark.

There was one difference, as one of the routes was broken (yellow and green clusters in Figure 5.12b), when crossing over another route with a different heading (red cluster). This did not happen in the benchmark with the same user-defined parameters. When the number of k-nearest neighbours was increased by one (Figure 5.12c), trying to get the route entirely, by accounting one more neighbour ($k = 13$) that could move the objects

from the green cluster to the yellow cluster, the cluster was not broken, obtaining the same routes identified in the benchmark (6 clusters).

5.2.2 Performance Evaluation

To evaluate the scalability of the *k*d-SNN, the Twitter data set was used, but was broken down to several subsets of different sizes: 500, 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, 256000 and 512000, in order to evaluate the performance with these different sizes.

The SNNAE is not used as a benchmark, because the source code is not publicly available and the experimental results presented in [BJ09] are done with a data set with 209 objects. These limitations do not allow a conclusive comparison with the SNNAE.

Since this thesis is focused on evaluating the improvement in the scalability of the SNN and not in finding quality and meaning clusters on the data sets used, and given the relevance of the number of neighbours in the SNN, the experimental results were done with the $k = 8$ ($Eps = 2$ and $MinPts = 6$), $k = 10$ ($Eps = 3$ and $MinPts = 7$) and $k = 12$ ($Eps = 4$ and $MinPts = 8$). The values given to the other user-defined parameters followed the procedure used in [MSC05], where the *Eps* and the *MinPts* are equal to 30% and 70% of the *k* values used, respectively.

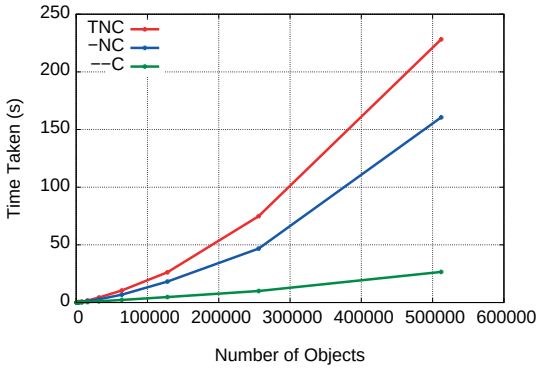


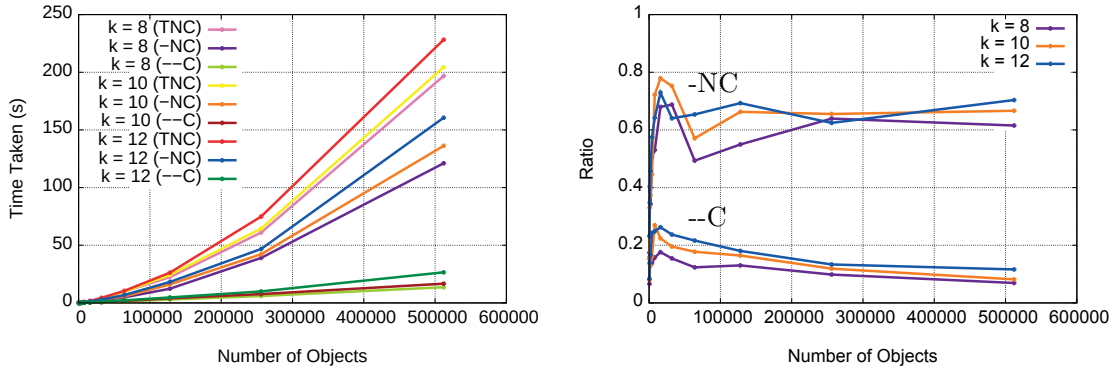
Figure 5.13: Runtime of the *k*d-SNN with $k = 12$

As it can be seen in figure 5.13, the runtime is decomposed in the three runs pointed out in figure 4.1: Tree, Neighbouring and Clustering (TNC) - the initial run, where it is necessary to build the metric data structure, query and store the *k*-nearest neighbours of each object, before executing the clustering steps of the SNN; Neighbouring and Clustering (-NC) - the run where the metric data structure is already built, but it still needs to query and store the *k*-nearest neighbours, before the clustering steps of the SNN; and Clustering (–C) - the run where the metric data structure is already built and the *k*-nearest neighbours are already known and only the clustering steps of the SNN are carried out.

In figure 5.13, the runtime taken by the *k*d-SNN to cluster the several subsets with 12-nearest neighbours ($k = 12$) is represented. It follows that the time spent to build

the *k*d-Tree has a significant impact, by comparing the runtime of the TNC run with the required by the -NC run. This impact is not only related with the time taken to build the metric data structure, but it is also due to the time spent in preprocessing the Twitter data set before each object is indexed in the *k*d-Tree. Regarding the -NC run and the –C run, its noticeable the effective impact that acquiring the *k*-nearest neighbours has on the runtime of the *k*d-SNN.

Regarding the impact of the number of neighbours, see figure 5.14a for the runtime values and figure 5.14b for the ratio $\frac{\text{Runtime of the -NC Run}}{\text{Runtime of the TNC Run}}$ and the ratio $\frac{\text{Runtime of the –C Run}}{\text{Runtime of the TNC Run}}$, for each of the number of neighbours. Taking into consideration that the TNC run and the -NC run have a time complexity in the average case of $O(n \times \log n)$ and that the –C run has a time complexity in the worst case of $O(n)$, then the proportions of the -NC ratio and the –C ratio should not be the same and should not be constant as the number of objects grows, which is verified in these experimental results. The -NC ratio was considerably the same along the different number of objects, as it ranged around 60%. As for the –C ratio, its values are less significant and it supports the $O(n \times \log n)$ time complexity in the average case of the TNC run, since as the number of objects is increased, the –C ratio values start to tend towards the same values.



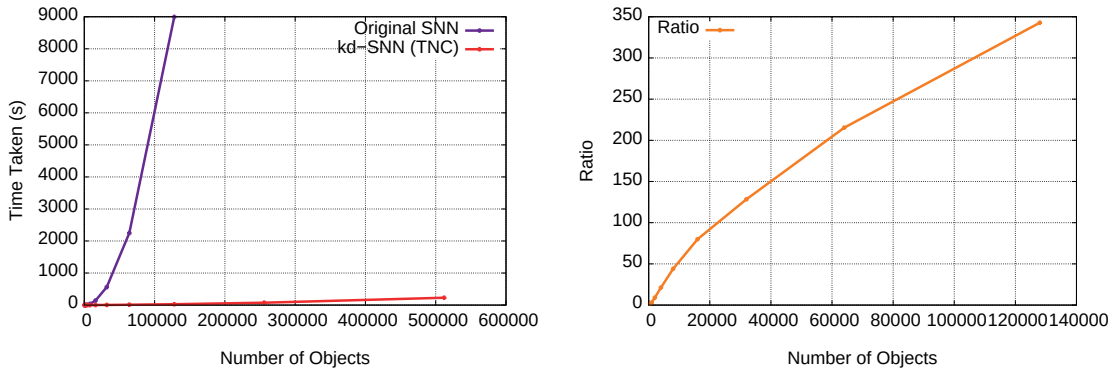
(a) Runtime

(b) Ratio

Figure 5.14: Runtime of the *k*d-SNN with $k = 8$, $k = 10$ and $k = 12$

Now, the TNC run of the *k*d-SNN is compared with the runtime of the referenced implementation of the Original SNN [ESK03]. In the benchmarking tests the implementation of the Original SNN was run with $k = 12$, but with the rest of the parameters involved in its implementation as default. In figure 5.15a, the runtime of the Original SNN and of the *k*d-SNN are presented, showing the quadratic behaviour of the Original SNN. Since the Original SNN takes a significant runtime, it was not evaluated for a data set above 128000 objects. However, this did not compromise the benchmarking test, as it is still perceived the improvement obtained with the *k*d-SNN.

On figure 5.15b, the ratio, $\frac{\text{Runtime of the Original SNN}}{\text{Runtime of the TNC Run of the } k\text{d-SNN}}$, reveals that for a spatial data set with 128000 objects, the runtime of the Original SNN was around 300 times higher



(a) Runtime (b) Ratio

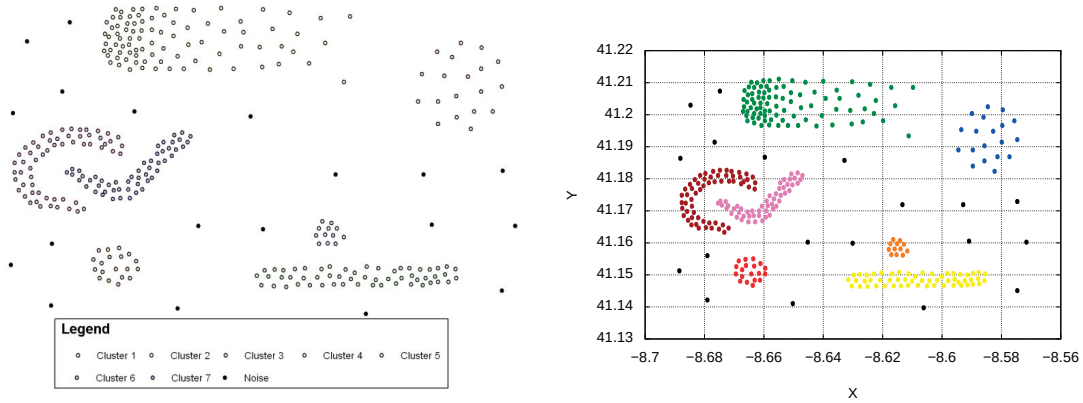
Figure 5.15: Runtime of the Original SNN and of the TNC run of the *k*d-SNN with $k = 12$

than the runtime of the *k*d-SNN, granting a distinct improvement in the scalability of the SNN algorithm, when using the *k*d-SNN to cluster this data set.

5.3 DF-SNN

5.3.1 Evaluation of the Quality of the Clustering Results

Bearing in mind the procedure of evaluation taken with the *k*d-SNN, the same procedure is carried out in the evaluation of the DF-SNN. First, the clustering results of the synthetic data sets are evaluated and, then, the two Marin data sets that represent the LPG and Oil ship types.

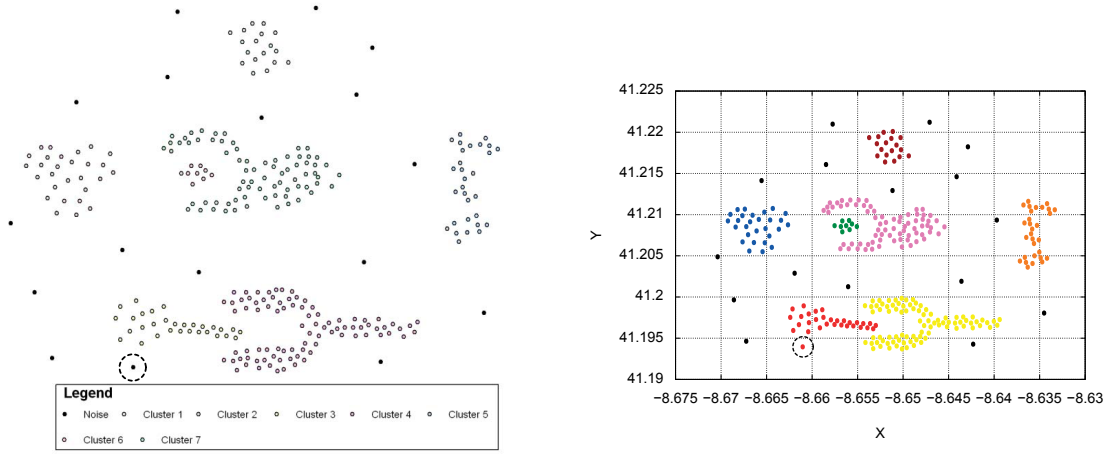


(a) Benchmark [MSC05] (b) DF-SNN

Figure 5.16: Clustering Results of the Synthetic 1 Data Set (Evaluation of the DF-SNN)

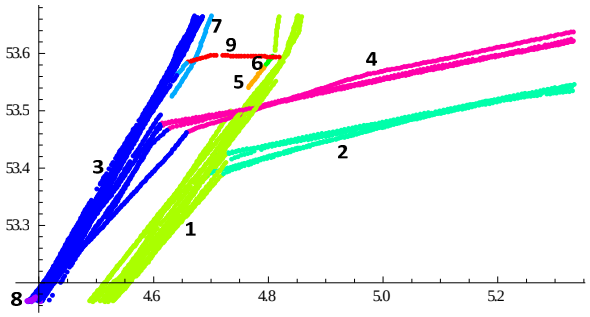
The Synthetic 1 still got the same clustering results as with the *k*d-SNN (Figure 5.8b). On figure 5.16a the benchmark is shown with k = 7, $Eps = 0.3 \times k$, $MinPts = 0.7 \times k$ and

on figure 5.16b are shown the clustering results obtained with the DF-SNN, having $k = 7$, $Eps = 2$ and $MinPts = 5$, that agree with its respective benchmark.

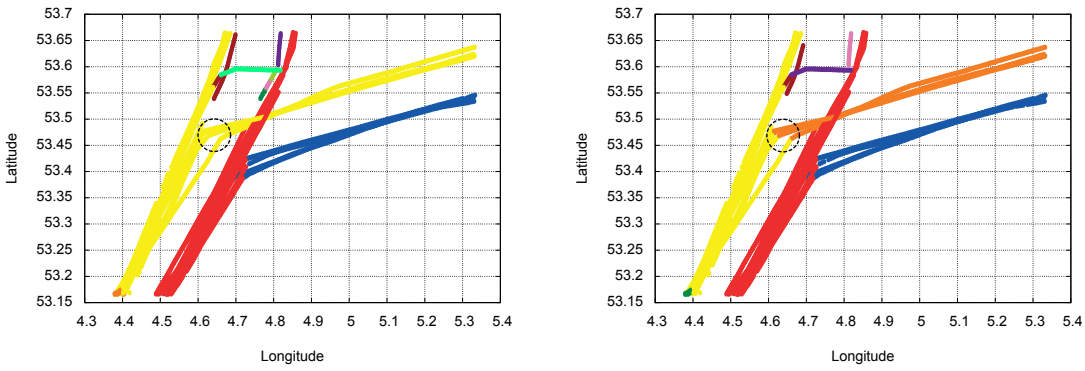


(a) Benchmark [MSC05] (b) DF-SNN

Figure 5.17: Clustering Results of the Synthetic 2 Data Set (Evaluation of the DF-SNN)



(a) Benchmark [SSMPW12]



(b) DF-SNN (c) DF-SNN

Figure 5.18: Clustering Results of the LPG Ship Type (Evaluation of the DF-SNN)

The Synthetic 2 clustering results are similar to the benchmkark, but it continues to cluster the object that was defined as noise in its benchmark (the object inside the dotted

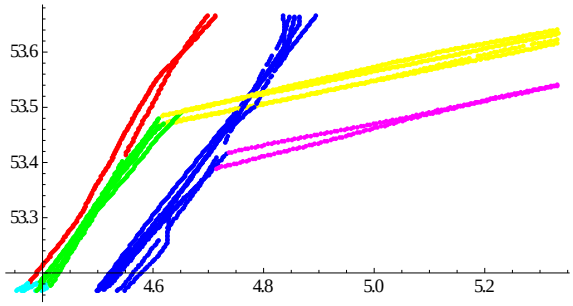
circle). See figure 5.17a for the benchmark with $k = 7$, $Eps = 0.3 \times k$, $MinPts = 0.7 \times k$ and see figure 5.17b for the clustering results obtained with the DF-SNN having $k = 7$, $Eps = 2$ and $MinPts = 5$.

The Synthetic 3 had the exact same results as with the *k*d-SNN, using $k = 7$, $Eps = 2$ and $MinPts = 5$.

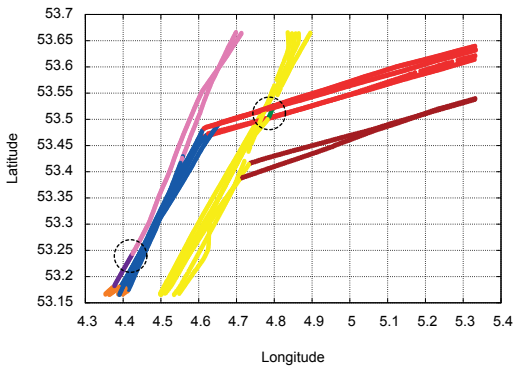
As for the clustering results of the Marin data sets, some results are slightly different from the clustering results obtained with the *k*d-SNN and the benchmark.

Figure 5.18b shows the clustering results of the LPG ship type, using the same user-defined parameters used in the benchmark, having $k = 12$, $Eps = 3$ and $MinPts = 7$ (Figure 5.18a). The same routes were identified and the one route was still not broken when the heading changed, as well as in the *k*d-SNN clustering results (Figure 5.11b). Since the same implementation of the SNN was used in the *k*d-SNN and in the DF-SNN, and the route was broken again, this may have happened due to differences between the implementation of the SNN used in the benchmark and the implementation used in the *k*d-SNN and in the DF-SNN.

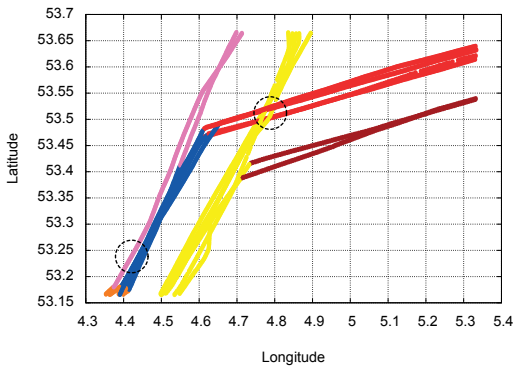
Adjusting the *MinPts* to 8, as in the *k*d-SNN, lead to the identification of similar routes (Figure 5.18c), agreeing with the *k*d-SNN clustering results (Figure 5.11c) and their respective benchmark (Figure 5.18a).



(a) Benchmark



(b) DF-SNN



(c) DF-SNN

Figure 5.19: Clustering Results of the Oil Ship Type (Evaluation of the DF-SNN)

Regarding the clustering results of the Oil ship type, when using the same values for

the user-defined parameters used in its benchmark with k = 12, *Eps* = 3 and *MinPts* = 7 (Figure 5.19a), in the clustering results (Figure 5.19b) the route was still broken in two (yellow and green clusters) when it crossed the route with a different heading (red cluster), as in the clustering results of the *k*d-SNN (Figure 5.12b). In addition, these clustering results also broke another route in two (violet and pink clusters), which did not happen in the *k*d-SNN clustering results. Since the same SNN implementation was used, this is due to differences in the *k*-nearest neighbours given by each of the metric data structures.

When the number of neighbours was increased by one ($k = 13$), the clustering results (Figure 5.19c) were similar to the *k*d-SNN clustering results (Figure 5.12c) and similar to the routes identified in the benchmark.

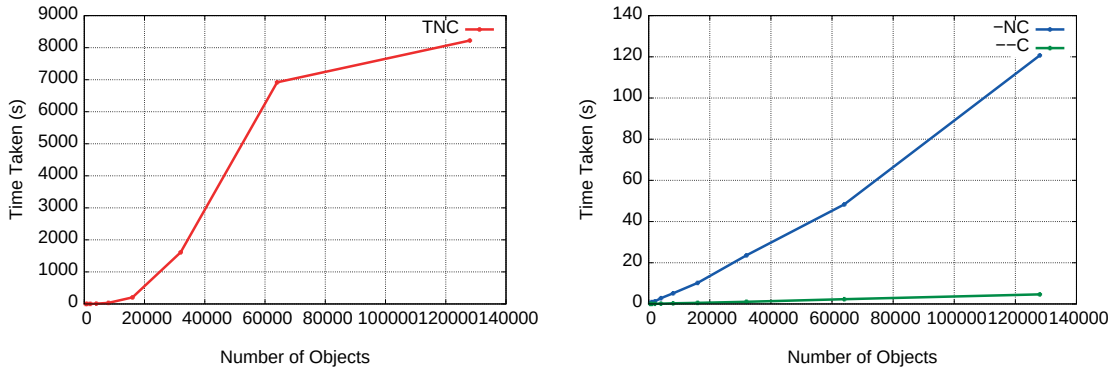
5.3.2 Performance Evaluation

The evaluation procedure carried out with the *k*d-SNN was the same that was taken with the DF-SNN. However, since the runtime of the TNC run of the DF-SNN is greater than its remaining runs, as it will be seen, the TNC run will be plotted in a different chart, in order not to compromise the interpretation and evaluation of the runtime of the -NC run and of the –C run.

The runtime of the DF-SNN is decomposed in the three runs pointed out in figure 4.1, however in the DF-SNN, besides the runtime of building the metric data structure, querying it and doing the clustering, it also accounts the runtime of deploying the SQL file, importing it to the DBMS and obtaining the *k*-nearest neighbours from the DBMS.

The Tree, Neighbouring and Clustering (TNC) run is the initial run, where it is necessary to build the metric data structure, query it for the *k*-nearest neighbours, deploy and import the SQL file to the DBMS. These operations are all part of the secondary storage component. Then, the primary storage component accesses the DBMS to get the *k*-nearest neighbours and does the clustering. The Neighbouring and Clustering (-NC) run accounts the time of getting the *k*-nearest neighbours, deploying and importing the SQL file to the DBMS, getting the *k*-nearest neighbours and, for last, clustering the data set. At last, the Clustering (–C) represents just the runtime of clustering the data set, since it does not require an access to the DBMS, since the *k*-nearest neighbours are already in primary storage.

In figure 5.20a it is shown the runtime of the TNC run of the DF-SNN, with 12-nearest neighbours. By comparing the runtime taken by the TNC run and the remaining two runs (Figure 5.20b), it is perceived that the runtime of getting the *k*-nearest neighbours from the DF-Tree, deploying and importing the SQL file, getting the *k*-nearest neighbours from the DBMS and clustering the data set, represented a very small part of the total runtime of the TNC run of the DF-SNN, leading the secondary storage component to the primary source of the runtime of the DF-SNN. Regarding the runtime required for the importing of the SQL file with the objects and their respective *k*-nearest neighbours (Figure 5.21) to the DBMS with 128000 objects, the importing took 0.7% of the runtime of the TNC run,



(a) DF-SNN (TNC)

(b) DF-SNN (-NC and –C)

Figure 5.20: Runtime of the DF-SNN with $k = 12$

representing a very small part of the runtime. The weight of importing became smaller as the data set size was increased, since the weight of building the tree became bigger with the number of objects in the data set.

As for the -NC run, it started to have an increase in its runtime as the size of the data set increased, not only because of the growing runtime required to get the *k*-nearest neighbours, but also due to the increasing number of rows that needed to be inserted in the DBMS. For example, with 128000 objects, it needed to input 1536000 rows with the *k*-nearest neighbours of each object, plus the 128000 rows with the objects attributes. Since the -NC run took less time than the TNC run, the weight of importing the SQL file to the DBMS had a bigger role in the runtime of the -NC run (Figure 5.21), for instance, with 128000 objects, the weight was approximately 50% and for the remaining data set sizes, the weight was always around the same range of values.

Data Set Size	Runtime of Importing on the TNC Run	-NC Run
500	26.5%	37.1%
1000	18.4%	30.6%
2000	19.1%	46.7%
4000	18.0%	54.1%
8000	7.9%	51.6%
16000	2.5%	50.4%
32000	0.8%	53.9%
64000	0.3%	46.9%
128000	0.7%	46.3%

Figure 5.21: Weight of Importing the SQL file to the DBMS

It is concluded that, the runtime of the DF-SNN was not as promising as the runtime obtained with the *k*d-SNN, namely the TNC run. This was somehow expected, due to the

distinct nature of each of the metric data structures, as the *k*d-Tree works in primary storage and the DF-Tree works in secondary storage, where the last demands higher access times.

As it is presented in figure 5.22, the number of accesses made by the DF-Tree to the secondary storage, became more and more significant as the data set size increased. This is due to the outgrowth of the tree size, leading to a much more deeper and larger tree. Not only this had an impact in the tree building, but it also complicated the whole answering process of the *k*-nearest neighbours queries, since more sub-trees needed to be explored and they were in different pages [1] that constantly need to be acquired from the secondary storage. This lead to an increase of the runtime, namely of the TNC run, when the data set had a magnitude higher than 16000 objects, becoming proportional to the number of secondary storage accesses.

Data Set Size	No. of Secondary Storage Accesses	
	Building	Querying
500	13633	5828
1000	45112	15554
2000	169668	40791
4000	674122	128139
8000	2644085	320973
16000	10305032	910158
32000	40709616	2664980
64000	160820199	7323828

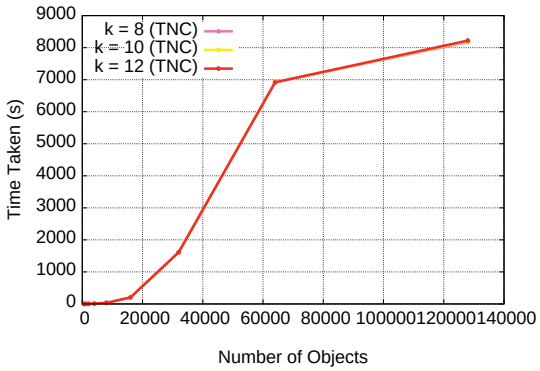
Figure 5.22: No. of Secondary Storage Accesses by the DF-Tree with $k = 12$

As the number of neighbours changed between the range of 8-nearest neighbours ($k = 8$), 10-nearest neighbours ($k = 10$) and 12-nearest neighbours ($k = 12$), the runtime of the DF-SNN was essentially the same (Figure 5.23), despite of the number of *k*-nearest neighbours used.

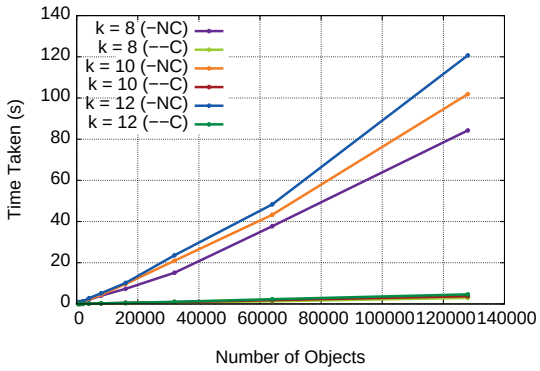
It seems that the runtime of the TNC run was not that affected with the change in the number of neighbours, at least with the values used (Figure 5.23a). On the other hand, despite not representing a significant difference in the runtime, since it is smaller than the runtime of the TNC run, the -NC run and the –C run were more sensitive to the variance of the number of neighbours. The -NC run was sensitive to the number of neighbours, because more rows needed to be inserted in the DBMS and then retrieved from the DBMS by the primary storage component. For instance, with 64000 objects and $k = 12$, there were 768000 rows that needed to be read, but with the same number of objects and $k = 8$, there were only 512000 rows, which is $\frac{2}{3}$ of the number of rows with $k = 12$. The –C run was also sensitive, since the number of neighbours has impact on the SNN algorithm.

On figure 5.23c, the ratio $\frac{\text{Runtime of the -NC Run}}{\text{Runtime of the TNC Run}}$ and the ratio $\frac{\text{Runtime of the –C Run}}{\text{Runtime of the TNC Run}}$ are

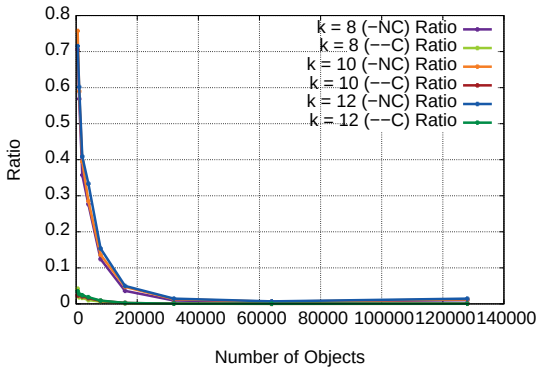
[1] Buffers of secondary storage-backed pages temporarily kept in primary storage by the operating system for quicker access and analysis.



(a) DF-SNN (TNC)



(b) DF-SNN (-NC and –C)



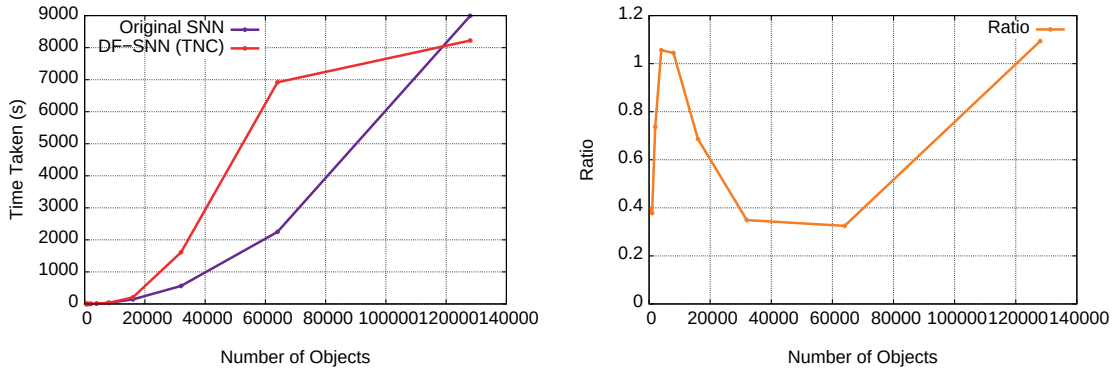
(c) Ratio

Figure 5.23: Runtime of the DF-SNN with $k = 8$, $k = 10$ and $k = 12$

shown, for the several k values. As already pointed out, it is concluded, by looking at the figure, that the runtime of the primary storage component was not significant in the total runtime of the DF-SNN, leading the total runtime of the DF-SNN to be dependent of the runtime of the secondary storage component, even with the variance in the number of neighbours.

In figure 5.24, the runtime of the DF-SNN is compared with the runtime of the Original SNN, showing a not so good improvement in the scalability of the Original SNN (Figure 5.24a).

Initially, with a data set with less than 4000 objects, the DF-SNN was not able to achieve an improvement over the Original SNN, as the ratio, $\frac{\text{Runtime of the Original SNN}}{\text{Runtime of the DF-SNN}}$, was below 100%. With a data set with a magnitude between 4000 and 8000 objects, the DF-SNN had an improvement of approximately 5% over the Original SNN. However, when the number of objects was above 8000 objects, there was a breaking point of this pattern, where the improvement started to decrease to none and the DF-SNN became, again, worse than the Original SNN. When it reached 128000 objects, the DF-SNN recovered



(a) Runtime

(b) Ratio

Figure 5.24: Runtime of the Original SNN and of the TNC run of the DF-SNN with $k = 12$

and was again able to achieve an improvement over the Original SNN of approximately 10%, becoming comparable with the performance of the referenced implementation of the Original SNN. These facts, left the DF-SNN as a not so good improvement in the scalability of the SNN, in contrast with the *k*d-SNN, when using the Twitter subset with 128000 objects.

6

Conclusions

This thesis proposes an implementation of the SNN clustering algorithm to deal with spatial data. It uses metric data structures to support the *k*-nearest neighbours queries of the SNN and, consequently, improve the scalability of the algorithm, since it currently has a $O(n^2)$ time complexity in the worst case. Bear in mind that this approach is applicable to all clustering algorithms that require the *k*-nearest neighbours of each object for its clustering process.

The scalability of clustering algorithms is important, because these type of algorithms require multiple runs to do the fine-tuning of the user-defined parameters, in order to achieve the best possible clustering results.

Two distinct implementations were made, taking advantage of different metric data structures: one implementation using the *k*d-Tree, which works in primary storage, and another implementation using the DF-Tree, which, on the other hand, works in secondary storage. Each of these metric data structures index the data sets and reply with the results of the *k*-nearest neighbours queries of each object in the indexed data set.

For low dimensional data, namely when dealing with spatial data, the proposed primary storage implementation improved the time complexity in the average case of the SNN to at most $O(n \times \log n)$: on an initial run, where it is necessary to build the *k*d-Tree, query it for the *k*-nearest neighbours and run the SNN; or on a run following the initial, where the *k*d-Tree is already built and it needs to be queried again for more *k*-nearest neighbours and run the SNN. When there is no need to query for more *k*-nearest neighbours and the SNN is the only step executed, reusing the previously calculated *k*-nearest neighbours, the time complexity in the worst case can be improved to $O(n)$.

The experimental results of the primary storage approach evaluated the quality and validity of the clustering results and the improvement in its scalability. In order to make

the evaluation of the clustering results, three synthetic and two real data sets and their respective clustering results obtained with another implementation of the SNN, were used. To measure the improvement in the scalability of the SNN, the runtime of the primary storage implementation was compared with the runtime of a referenced implementation in the literature of the SNN.

The primary storage implementation returned similar clustering results for the data sets used, when compared to their respective benchmarks.

The performance results of the primary storage implementation showed a good improvement in the scalability of the SNN, when compared with the referenced implementation. The initial run improved the scalability of the SNN, substantiating the improvement in its scalability, as it clustered a three-dimensional data set with 128000 objects using 12-nearest neighbours, 300 times faster than the referenced implementation. However, in the following runs, using a strategy over the user-defined parameters, starting with a higher number of *k*-nearest neighbours, the primary storage implementation was even more faster. If the algorithm starts with a higher number of *k*-nearest neighbours, and then, in the following runs, the number is adjusted downward, the runtime to query the metric data structure for more *k*-nearest neighbours can be avoided, only executing the SNN algorithm, which is significantly faster than the initial run, as it only takes 10% of the runtime of the initial run. However, the improvement in these performance results may start to decrease as the dimensionality of a data set grows, since the theoretical results of the time complexity of the *k*d-Tree involve the dimensionality of a data set, which when not in a low dimensionality scenario, may start to affect the scalability of the *k*d-Tree, compromising its performance and, consequently, the performance of the SNN.

Regarding these primary storage implementation results, a paper [FMPS12] with these results was submitted and accepted on the INForum 2012, a Portuguese annual symposium on computer science, applied informatics and information technology.

On the other hand, the secondary storage implementation did not obtain such promising results as the primary storage implementation. It still got similar clustering results for the used data sets, when compared to their respective benchmarks. However, only with a three-dimensional data set with 128000 objects using 12-nearest neighbours, the secondary storage approach was able to obtain a runtime of the initial run comparable with the referenced implementation, improving the runtime in approximately 10%. With fewer objects, only with 4000 and 8000 objects, the secondary storage approach got a better runtime, with an improvement of approximately 5%. The referred strategy over the user-defined parameters also applies to this secondary storage implementation. In the experimental results, the following runs obtained a greater time than the initial run. With 128000 objects and using 12-nearest neighbours, the following runs took less than 0.05% of the time of the initial run, where only the SNN algorithm is executed. It follows that, this approach should also start with a higher number of neighbours, and then, the SNN can have a bigger leeway regarding the number of neighbours that can be used and that are stored in the database.

Despite the fact that the DF-Tree does not have their time complexity evaluated, some experimental results taken showed that the *k*d-Tree requires a lot more distance calculations than the DF-Tree. This leads to believe that the DF-Tree may handle more the growth of the dimensionality of a data set, when compared with the *k*d-Tree. These assertions could be validated by making several evaluations with data sets with a different dimensionality.

So, to reinforce the results of this thesis, several evaluations could be done:

- evaluate the influence of the dimensionality of the data sets over the runtime. In this experimental results, the runtime was evaluated with a three-dimensional data set with numeric attributes. However, some experimental results could be obtained, using data sets with more attributes, numerical or alphanumerical, to obtain a better perception of how the growth of the dimensionality reduces the gains of the implementations of the SNN using the primary and secondary storage approaches;

- evaluate the effect of the domain size of the data set attributes on the runtime. Experimental results could be obtained, using data sets that have attributes with a larger domain size, in order to perceive the impact that it has over the runtime of the implementations of the SNN using the primary and secondary storage approaches;

- evaluate the impact of using different data sets over the runtime. Every data set has a clustering tendency and, consequently, an impact over the runtime of the SNN, since every data set has a specific set of values for the user-defined parameters of the SNN, that lead to specific clustering results. The goal would be to evaluate how a specific data set with a specific set of values for the user-defined parameters of the SNN has impact on the runtime of the implementations of the SNN using the primary and secondary storage approaches, while evaluating the influence of the variation of the values of the Eps and the $MinPts$, the remaining user-defined parameters of the SNN.

Besides the above evaluations, since the scope of this thesis was later focused on spatial data, some experimental results with spatial data metric data structures could have be done, in order to evaluate the performance obtained with these type of metric data structures, for instance, the R-Tree [Gut84]. The future work can involve evaluating the performance with these type of metric data structures and, due to the results obtained with the runtime of the primary storage implementation, some future work can also be done towards the tuning of the user-defined parameters:

- auto-tuning - if there are enough resources to establish an evaluation criteria of the clustering results, the SNN can be adjusted to make an auto-tuning of the user-defined parameters, as it produces the clustering results, in order to find the best possible clustering results in a timely manner;

- interactive-tuning - this solution fits in the research area of Geovisual Analytics [AAJ+07], which seeks to find ways to engage the user to provide computer support to solve spatial-related decision problems, by enhancing human capabilities to reason and analyse the output that is given by the computer. The runtime obtained, strengthens the idea of an interactive solution, where the interaction between the user input of the values of the user-defined parameters of the SNN and the clustering results, can be done in a timely manner, going towards the idea of Geovisual Analytics.

Bibliography

[Ža11] Borut Žalik. Validity index for clusters of different sizes and densities. *Pattern Recognition Letters*, 32(2):221–234, 2011.

[AAJ+07] G. Andrienko, N. Andrienko, P. Jankowski, D. Keim, M. J. Kraak, A. MacEachren, and S. Wrobel. Geovisual analytics for spatial decision support: Setting the research agenda. *Int. J. Geogr. Inf. Sci.*, 21(8):839–857, January 2007.

[BCKO10] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, softcover reprint of hardcover 3rd ed. 2008 edition, November 2010.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[BJ09] H. B Bhavsar and A. G Jivani. The shared nearest neighbor algorithm with enclosures (SNNAE). In *2009 WRI World Congress on Computer Science and Information Engineering*, volume 4, pages 436–442. IEEE, April 2009.

[Bur10] Stephen D. Burd. *Systems Architecture*. Cengage Learning, August 2010.

[CNBYM01] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, September 2001.

[CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. *Processing*, Athens, Gr:426–435, 1997.

[Dat06] Databases and Images Group. Downloads | databases and images group. http://www.gbdi.icmc.usp.br/en/download, August 2006.

[dCFLH04] Luis M de Campos, Juan M Fernández-Luna, and Juan F Huete. Bayesian networks and information retrieval: an introduction to the special issue. *Information Processing & Management*, 40(5):727–733, September 2004.

[DHS00] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley-Interscience, 2 edition, October 2000.

[EKSX96] Martin Ester, Hans-peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *Computer*, 1996(6):226–231, 1996.

[ESK03] Levent Ertöz, Michael Steinbach, and Vipin Kumar. Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data. *In Proceedings of Second SIAM International Conference on Data Mining*, 2003.

[FBF77] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, September 1977.

[Fer07] Olivier Ferret. Finding document topics for improving topic segmentation. 2007.

[FMPS12] Bruno Filipe Faustino, João Moura-Pires, and Maribel Yasmina Santos. Implementação eficiente do shared nearest neighbour em dados espaciais. In Antónia Lopes and José Orlando Pereira, editors, *Proceedings of INForum 2012*, pages 476–479, Monte da Caparica, Portugal, September 2012. Universidade Nova de Lisboa.

[FSTR06] A. M Fahim, A. M Salem, F. A Torkey, and M. A Ramadan. Density clustering algorithm based on radius of data (DCBRD). *Computer Sciences and Telecommunications*, (4):32–43, 2006.

[GMW07] Guojun Gan, Chaoqun Ma, and Jianhong Wu. *Data Clustering: Theory, Algorithms, and Applications*. SIAM, Society for Industrial and Applied Mathematics, May 2007.

[Goo12] Google Developers. Protocol buffers. `https://developers.google.com/protocol-buffers/`, April 2012.

[GRS98] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: an efficient clustering algorithm for large databases. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, page 73–84, New York, NY, USA, 1998. ACM.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[GXZ+11] Yong Ge, Hui Xiong, Wenjun Zhou, Siming Li, and Ramendra Sahoo. Multifocal learning for customer problem analysis. *ACM Trans. Intell. Syst. Technol.*, 2(3):24:1–24:22, May 2011.

[HBV02a] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. Cluster validity methods: part i. *SIGMOD Rec.*, 31(2):40–45, June 2002.

[HBV02b] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. Clustering validity checking methods: part II. *SIGMOD Rec.*, 31(3):19–27, September 2002.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.

[JP73] R. A Jarvis and E. A Patrick. Clustering using a similarity measure based on shared near neighbors. *IEEE Transactions on Computers*, C-22(11):1025–1034, November 1973.

[KHK99] G. Karypis, Eui-Hong Han, and V. Kumar. Chameleon: hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, August 1999.

[KI06] Ferenc Kovács and Renáta Iváncsy. Cluster validity measurement for arbitrary shaped clusters. In *Proceedings of the 5th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases*, page 372–377, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS).

[Lia99] Sheng Liang. *Java™ Native Interface: Programmer's Guide and Specification*. Prentice Hall, 1 edition, June 1999.

[Lov04] Alen Lovrencic. maximally connected component. `http://xlinux.nist.gov/dads/HTML/maximallyConnectedComponent.html`, December 2004.

[Mac67] JB Macqueen. Some methods of classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[Mar06] David Marshall. Nearest neighbour searching in high dimensional metric space. Technical report, 2006.

[MG09] Jeremy Mennis and Diansheng Guo. Spatial data mining and geographic knowledge discovery—An introduction. *Computers Environment and Urban Systems*, 33(6):403–408, 2009.

[MH09] Harvey J. Miller and Jiawei Han. *Geographic Data Mining and Knowledge Discovery, Second Edition*. CRC Press, 2 edition, May 2009.

[MSC05] Adriano Moreira, Maribel Yasmina Santos, and Sofia Carneiro. Density-based clustering algorithms – DBSCAN and SNN. http://get.dsi.uminho.pt/local/download/SNN&DBSCAN.pdf, July 2005.

[MTJ06] Wannes Meert, Remko Tronçon, and Gerda Janssens. Clustering maps. 2006.

[NUP11] Gonzalo Navarro and Roberto Uribe-Paredes. Fully dynamic metric access methods based on hyperplane partitioning. *Inf. Syst.*, 36(4):734–747, June 2011.

[Pat99] Marco Patella. *Similarity Search in Multimedia Databases*. PhD thesis, Dipartimento di Elettronica Informatica e Sistemistica, Università degli Studi di Bologna, Bologna, Italy, February 1999.

[PB07] Oscar Pedreira and Nieves R. Brisaboa. Spatial selection of sparse pivots for similarity search in metric spaces. In *Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '07, page 434–445, Berlin, Heidelberg, 2007. Springer-Verlag.

[RPN+08] Salvatore Rinzivillo, Dino Pedreschi, Mirco Nanni, Fosca Giannotti, Natalia Andrienko, and Gennady Andrienko. Visually driven analysis of movement data by progressive clustering. *Information Visualization*, 7(3):225–239, June 2008.

[Sar10] Ângelo Miguel Loureiro Sarmento. *Estruturas de dados métricas genéricas em memória secundária*. MSc thesis, Departamento de Informática, Universidade Nova de Lisboa, 2010.

[Sav12] Savarese Software Research Corporation. libssrckdtree-j generic k-d tree java library. http://www.savarese.com/software/libssrckdtree-j/, 2012.

[SB11] Tomáš Skopal and Benjamin Bustos. On nonmetric similarity search problems in complex domains. *ACM Comput. Surv.*, 43(4):34:1–34:50, October 2011.

[SKN07] Kalyankumar Shencottah K. N. *Finding Clusters in Spatial Data*. MSc thesis, University of Cincinnati, 2007.

[SMO11] Raisa Socorro, Luisa Micó, and Jose Oncina. A fast pivot-based indexing algorithm for metric spaces. *Pattern Recognition Letters*, 32(11):1511–1516, August 2011.

[SSMPW12] Maribel Yasmina Santos, Joaquim P. Silva, João Moura-Pires, and Monica Wachowicz. Automated traffic route identification through the shared nearest neighbour algorithm. In Jérôme Gensel, Didier Josselin, and Danny Vandenbroucke, editors, *Bridging the Geographic Information Sciences*, pages 231–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[TTFF02] Caetano Traina,Jr., Agma Traina, Roberto Santos Filho, and Christos Faloutsos. How to improve the pruning ability of dynamic metric access methods. In *Proceedings of the eleventh international conference on Information and knowledge management*, CIKM '02, page 219–226, New York, NY, USA, 2002. ACM.

[TTFS02] C. Traina, A. Traina, C. Faloutsos, and B. Seeger. Fast indexing and visualization of metric data sets using slim-trees. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):244–260, April 2002.

[Twi12] Twitter. REST API resources | twitter developers. https://dev.twitter.com/docs/api, 2012.

[WCC09] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Cheng. *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann, February 2009.

[WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann, 2 edition, June 2005.

[YKI08] N. A Yousri, M. S Kamel, and M. A Ismail. A novel validity measure for clusters of arbitrary shapes and densities. In *19th International Conference on Pattern Recognition, 2008. ICPR 2008*, pages 1–4. IEEE, December 2008.

[ZWW07] Bing Zhou, He-xing Wang, and Cui-rong Wang. A hierarchical clustering algorithm based on GiST. In De-Shuang Huang, Laurent Heutte, and Marco Loog, editors, *Advanced Intelligent Computing Theories and Applications. With Aspects of Contemporary Intelligent Computing Techniques*, volume 2, pages 125–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.