ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2018-ENST-047

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « INFORMATIQUE et RESEAUX »

présentée et soutenue publiquement par

Thibault DEBATTY

le 5 octobre 2018

Design and analysis of distributed k-nearest neighbors graph algorithms

Directeur de thèse : **Pietro MICHIARDI**
Co-directeur de thèse : **Wim MEES**

Jury

M. Giovanni NEGLIA, Chargé de Recherche HDR, INRIA	Rapporteur
M. Jean-Michel DRICOT, Professeur associé, Université Libre de Bruxelles	Rapporteur
M. Marc DACIER, Professeur, Eurecom	Examinateur
Mme Elena BARALIS, Professeur, Politecnico di Torino	Examinatrice

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

THÈSE

Acknowledgements

This thesis was possible thanks to the help and support of many.

First of all, I would like to thank my two advisors, Pietro Michiardi and Wim Mees, for their valuable advice, coaching, support and patience.

I naturally thank the members of the thesis committee for taking the time to read this thesis: Giovanni Neglia, Jean-Michel Dricot, Marc Dacier and Elena Baralis.

I would like to specially thank Olivier Thonnard, without whom this thesis would definitively not have been possible.

A lot of thanks go to my friends and colleagues from the Royal Military Academy, from Eurecom, and from the Symantec office at Eurecom.

And last but not least, I would like to thank my wife Axelle for supporting me, particularly the days and nights preceding each paper submission deadline.

Abstract

A k-nn graph is a special kind of graph where each node has an edge (a link) to the k most similar other nodes in the graph according to some appropriate measure of similarity. A k-nn graph can actually be used as an index, with multiple advantages: 1) it requires little memory, 2) it can be used with any measure of similarity, even non-metric and 3) although building a k-nn graph is usually a CPU intensive and time consuming operation, processing a k-nn graph is very fast.

In this thesis we study the usage of k-nn graphs to analyze large datasets. We make the assumption that the dataset is so large that it cannot be processed using a single computer. Hence we propose algorithms designs relying on the bulk synchronous parallel (BSP) model, then we implement them and evaluate their performance using either the Apache Hadoop MapReduce framework or the Apache Spark framework.

We study two different aspects of k-nn graphs: 1) how to build and store them and 2) how to use them efficiently to analyze data. In the first part of this thesis we propose an efficient way to build a k-nn graph from large text datasets, we show how to update a k-nn graph when data has to be inserted in or removed from the graph, and how to partition the nodes of a large k-nn graph between multiple compute nodes in a way that optimizes the analysis of the graph. In the second part we show that k-nn graphs can successfully be used to perform clustering of large text datasets and to detect compromised computers in a network.

Résumé

Un graphe des k plus proches voisin (graphe k-nn, de l'anglais k-nearest neighbors) est un type de graphe spécifique où chaque point (nœud) possède un lien vers les k autres points les plus semblables du graphe selon une mesure appropriée de similarité. Pour l'analyse de grands volumes de données, un graphe k-nn peut être utilisé comme index, et offre de multiples avantages : 1) il nécessite peu de mémoire, 2) il peut être utilisé avec n'importe quelle mesure de similarité, même non métrique et 3) bien que la construction d'un graphe k-nn soit généralement une opération qui nécessite beaucoup de calcul et de temps, le traitement d'un graphe k-nn est très rapide.

Dans cette thèse, nous étudions l'utilisation de graphe k-nn pour analyser de grands volumes de données. Nous supposons d'ailleurs que le jeu de données est si important qu'il ne peut être traité à l'aide d'un seul ordinateur. Nous proposons donc des algorithmes s'appuyant sur le modèle BSP (bulk synchronous parallel), puis nous les implémentons et évaluons leur performance en utilisant soit le framework Apache Hadoop MapReduce, soit le framework Apache Spark.

Nous étudions deux aspects différents des graphes k-nn : 1) comment les construire et les stocker et 2) comment les utiliser efficacement pour analyser les données. Dans la première partie de cette thèse, nous proposons une méthode efficiente pour construire un graphe k-nn à partir de grands ensembles de données textuelles, nous montrons comment mettre à jour un graphe k-nn lorsque des données doivent être insérées ou supprimées du graphe, et comment partitionner les points d'un grand graphe k-nn entre plusieurs nœuds de calcul de façon à optimiser l'analyse du graphe. Dans la deuxième partie, nous montrons que les graphes k-nn peuvent être utilisés avec succès pour effectuer le regroupement de grands ensembles de données textuelles (text clustering) et pour détecter les ordinateurs compromis dans un réseau.

Contents

Chapter 1

Introduction

One of the hypes in the IT field is currently the Big Data trend. To understand the phenomenon, one must return to its birth. During the 1990s, the price of data storage has dropped dramatically, from around 40,000€ per GB in 1987 to about 2€ per GB in 2002 and 0.05€ per GB in 2015 [42]. Due to the prohibitive storage cost, until then companies only stored the data that was strictly required to manage the relationship with their customers (name, address, services or products ordered, invoices, etc.). Besides, outdated data was quickly discarded to save space and money. With the democratization of mass storage, these companies started to also store all the information concerning interactions with their potential customers, without ever deleting anything, and without even worrying about their possible usefulness. Google and Yahoo! for example, the precursors of Big Data, have since stored all activity conducted by their users.

Tesco, the largest supermarket chain in the United Kingdom, was one of the first large companies to discover and successfully exploit the business benefits of this data. Starting in the mid-1990s, Tesco launched its own loyalty program, called "Clubcard". Many competitors had already used similar cards to encourage customer loyalty. However, Tesco was the first to store the data generated by the use of these cards: which customer purchased which product and when. The firm then used this data to better target the promotional mailings and coupons sent to customers, which resulted in a strong increase of their use (from 3% to 70%), as well as an increase of sales [28].

Seeing these results, Tesco has also applied this analytical approach to other areas. One of the most profitable uses for the company is the analysis of sales as a function of the weather to optimize their inventory management system. By being able to forecast sales by product for each store, Tesco was able to optimize logistic and save £100 million (about 127 million €).

At the moment, the phenomenon is only increasing. The amount of data produced every second in the world is exploding. IoT devices, e-commerce applications, security or financial services continue to generate huge amounts of data that is captured and stored. The use of this data is currently considered a key to the competitiveness and growth: just like Tesco did, analyzing these data allows companies to discover hidden trends and opportunities.

Processing these large amounts of data in an efficient way requires appropriate indexing

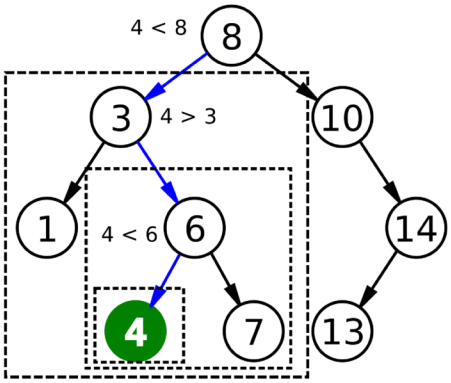

Figure 1.1: Example of a Binary Search Tree used to search the element with value 4 [102].

structures: without an index, every operation on the data (search, cluster,...) requires to scan the complete dataset, possibly multiple times.

Different indexing structures exist. The most simple kind of index is a sorted binary tree (also called binary search tree, BST) [27]. It assumes that the data to index can be ordered. A BST maintains a tree-like structure of the data where elements are sorted. This allows to search an item in time $O(\log n)$, where n is the number of elements in the tree. Figure 1.1 shows how a BST can be used to search the element with value 4.

B-trees [25], B+ trees and B* trees are generalizations of binary search trees that store multiple values at each level of the tree and have multiple branches. They are commonly used by database systems and file systems for example. For points in $\mathbb{R}^n$, like geographic data, k-dimensional trees (k-d trees) and R trees can be used.

Next to these classical options, a k-nn graph is an unusual but interesting candidate. A k-nn graph is a graph where each node has an edge (a link) to the k most similar other nodes in the dataset. Although it is very simple compared to some other indexes, it allows to perform some common analysis tasks like clustering or similarity search in an efficient way. It has three additional interesting characteristics:

1. it can be used with any measure of similarity, even non metric. A k-nn graph is thus a multipurpose index that can be used for multiple applications, like text datasets equipped with a non-metric measure of string similarity for example. These kinds of datasets are actually very common and will be used throughout this thesis for experimental evaluation;

2. it requires little memory: the memory requirement of a k-nn graph is proportional to the size of the dataset (i.e. $\propto kn$ where n is the size of the dataset). This makes them suitable for very large datasets;

3. although building a k-nn graph is usually a computationally intensive and time consuming operation, as we will show in Chapter 3, processing a k-nn graph is usually very fast. This makes k-nn graphs a premium choice for implementing interactive analysis tools, like the one we present in Chapter 7.

This thesis focuses precisely on the usage of k-nn graphs for processing large datasets.

Moreover, we make the assumption that the dataset is so large that it cannot be processed using a single compute node, either because it does not offer sufficient memory, or because it cannot process the data within a acceptable amount of time. Hence we focus on distributed algorithms, that can take advantage of multiple compute nodes to process the data.

In the first part of this thesis, *Fundamental algorithms*, we study:

— how to efficiently build k-nn graphs from large text datasets;

— how to update a k-nn graph when new data has to be inserted or removed;

— how to partition a large k-nn graph between multiple compute nodes.

In the second part, *Applications*, we show how to use k-nn graphs to perform clustering of large text datasets, and how to use them to detect compromised computers in a network.

Chapter 2

Background and related work

Both k-nn graphs and distributed processing are vast subjects. Concerning k-nn graphs, a vast literature exists that covers a lot of different algorithms and applications. Distributed processing is an even broader subject. Next to theoretical papers describing algorithms and approaches to process data in parallel, a lot of different technologies and frameworks exist that can be used to build distributed systems.

This thesis is of course based on a large quantity of this pre-existing work. In this Chapter we review them and present more in details the algorithms and technologies that we used.

2.1 k-nn graphs

In the general case, a graph is a mathematical structure made up of nodes (also called vertices) connected with edges. The edges of a graph may be directed or undirected. The edges may also indicate a weight, which results in a weighted graph. Graph theory is a very ancient topic, dating back to 1736 [15]. It has received a strong highlight these last years with the explosion of the Internet, search engines and social networks like Facebook and Twitter. Indeed, the data generated by these networks can easily be formatted as a graph, as does the web itself, making graph algorithms a premium analysis tool.

As a consequence, a lot of research has been devoted to efficiently analyze this kind of data, either sequentially or in parallel, like in [85], [60], [17] or [90].

Instead of physical relationships like "is a friend of", "likes" or "has an hyperlink to", edges may also represent the similarity between nodes. This results in a special kind of directed graph, called a nearest neighbors graph, of which two flavors exist. The most commonly used is the k-nearest neighbors graph (k-nn graph), where each node is connected to (has an edge to) its k nearest neighbors, according to a given measure of similarity.

Another possibility is to create an ϵ-nn graph, a graph where an edge exists between two nodes if their distance is less than a pre-defined threshold ϵ. However, it has been shown in [12] that ϵ-nn graphs easily result in disconnected components. Moreover, it is usually difficult to find a good value of ϵ which yields graphs with an appropriate number of edges [20]. From a practical point of view, it is more efficient to build a k-nn graph and afterward

filter the graph with different values of ϵ. Hence most of the research on graph building currently focuses on k-nn graphs.

Unlike "natural" graphs (like Facebook graph, twitter graph or web graphs [59]), k-nn graphs do not automatically emerge from data. They have to be built from a regular dataset, using an appropriate measure of similarity between elements. This is usually a CPU intensive and time consuming operation, although optimizations exist, as we will show in Chapter 3.

Once built, k-nn graphs are powerful tools to analyze data. For example, the authors in [46] build a nearest neighbor graph of landscape images to allow interactive exploration of large image databases. In [81] the authors also use a nearest neighbor graph of images, this time to automatically discover objects in images. In [91], a k-nn graph is used to search similar images in a large database. In [94] the authors use a nearest neighbor graph to perform dimensionality reduction. In [68], the authors use nearest neighbor graphs to perform clustering. In Chapter 6 we also show how to use k-nn graphs to cluster text datasets and in Chapter 7 we use them to detect compromised computers on a network.

2.2 Distributed computing

When a dataset is so large that it cannot be processed using a single computer, as we assume for this thesis, one has to rely on multiple computers to do the job. In a distributed system, a group of networked computers work together to achieve this common goal. Unlike parallel systems, they do not share a common memory. Distributed algorithms have to take this limitation into account.

Different models and technologies exist to achieve distributed processing, like the Message Passing Interface (MPI) for example. As the name states, MPI relies on the exchange of messages between the computers. This approach thus makes the implicit assumption that the network allows extremely fast communication (both low latency and high bandwidth).

However, when using commodity hardware, sending a message over the network is usually orders of magnitude slower than accessing the local memory. Hence for this thesis we focus on another model, that does take communication cost into account: the bulk synchronous parallel (BSP) model. In this model a computation proceeds in a series of *supersteps*, which consist of three stages:

1. concurrent computation;
2. communication between the computers;
3. barrier synchronization.

As communication actions are executed in bulk, an upper bound on the communication cost of a BSP algorithm can be computed. This cost will often account for a large part of the total processing time, and has to be minimized.

More specifically, for our experimental evaluation we implement our algorithms using the MapReduce model [29], and we use two different targets: Apache Hadoop MapReduce and Apache Spark [105].

2.3 MapReduce

`map` and `reduce` are originally array operators found in most (if not all) programming languages:

— `array.map(function)` applies the provided function to all elements of the array. Hence it returns an array of the same size.

— `array.reduce(function, initial_value)` iteratively reduces the array by applying the provided function to all elements of the array. In this case the function takes two arguments: the current value of the array, and the aggregated value obtained by applying the function to previous elements of the array. The result of `array.reduce` is hence a single aggregated value.

In their seminal paper [29], Jeffrey Dean and Sanjay Ghemawat from Google describe how to use `map` and `reduce` operations to simplify distributed computing on commodity hardware. This model naturally relies on some assumption: the input data is a large set of records that can be processed independently. In the MapReduce model, the map function takes a single record as parameter, and emits a key, value pair. The reduce function receives as parameter the list of values that share the same key and computes a single result corresponding to this key.

Although this model can seem quite simple and thus limiting, the range of possible applications was surprisingly broad: the MapReduce model was successfully applied to tasks such as distributed grep, count of URL access frequency, sort, inverted index building etc.

This model has the enormous advantage that it makes parallelization quite straightforward and transparent to the user: the input data is split between the different compute nodes, where the map function is applied independently, without requiring any additional communication between compute nodes. In the second stage, the reduce operation is also applied independently by the different compute nodes.

Overall, the only required communication consists in 1) distributing the input data between the compute nodes and 2) transporting the different key, value pairs emitted by the map function to the correct compute node, where the reduce function will be applied. This, however, is the responsibility of the MapReduce framework and is completely transparent to the user.

Moreover, the MapReduce model allows to easily achieve data locality: the map operation is preferably executed by the compute node that already holds the input data. This requires to 1) split the data in slices (called splits) and distribute them between the compute nodes and 2) orchestrate the execution such that each compute node applies the map function to the splits that it holds. These two tasks are transparent to the user as well, and are taken care of by the distributed filesystem and by the MapReduce framework respectively.

2.4 Hadoop

To support their seminal paper with experimental evaluation, Dean and Ghemawat implemented their own MapReduce framework, which was largely used by Google for different

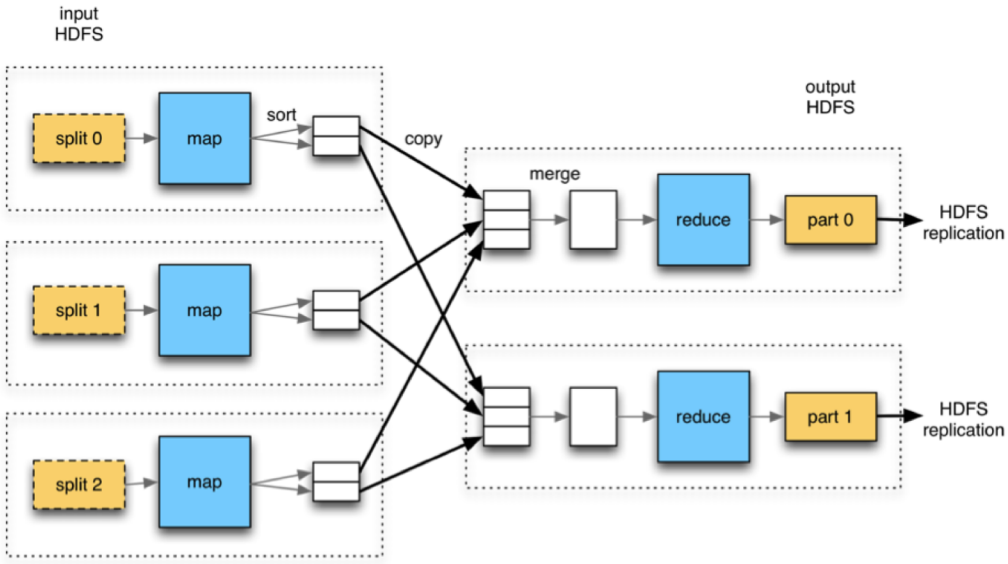

Figure 2.1: Example of data flow in a distributed MapReduce application [24].

applications. Others implementations also appeared quickly after the publication of the paper, based on the ideas described in the paper, in different programming languages.

Amongst all of these, the most used is probably Apache Hadoop MapReduce. As the name states, this implementation if part of the Apache Hadoop project. [1] The original Hadoop project has two components: 1) the Hadoop Distributed File System (HDFS) which is responsible for splitting the data and storing it on the different compute nodes and 2) the Hadoop MapReduce framework itself which is responsible for executing the MapReduce jobs submitted by the user [2]

The flow of data between the different components is illustrated in Figure 2.1.

In 2013, version 2 of Hadoop was released. The biggest difference between Hadoop 1 and Hadoop 2 is YARN technology, which is an acronym for Yet Another Resource Negotiator. YARN is a resource management technology which is deployed on a Hadoop cluster. It allows to effectively allocate the resources (compute nodes) to different user applications. It runs two daemons, which take care of two different tasks: job tracking and progress monitoring.

Hadoop is originally a project from Yahoo, and it is still massively supported by Yahoo. Among other things, Hadoop clusters from Yahoo won the TeraByte Sort challenge at multiple occasions [3]. Yahoo reports to have more then 40.000 computers running Hadoop [4]. The framework is also massively used by Facebook, with at least 1400 machines running Hadoop [5].

1. `https://hadoop.apache.org/`
2. For the sake of completeness, there is also a third component, called Hadoop Common, containing common utilities.
3. `http://sortbenchmark.org/`
4. `https://wiki.apache.org/hadoop/PoweredBy#Y`
5. `https://wiki.apache.org/hadoop/PoweredBy#F`

2.5 Spark

Despite the power of Hadoop MapReduce, it has two main drawbacks, which are actually related:

1. each job is limited to a single Map then Reduce sequence;
2. after the sequence, data can only be written to disk.

In a lot of applications, processing the data requires to perform additional steps (after the MapReduce sequence). With Hadoop MapReduce, the data has to be written to disk (and read) before these additional operations can be executed. This introduces a large IO penalty. Similarly, iterative algorithms require to write the data to disk at each iteration.

The Spark project [6] was developed to mitigate these limitations [105]. It allows to execute multiple successive operations on the dataset. To achieve this goal, however, Spark relies on a strong assumption: that the combined RAM memory of all compute nodes is large enough to hold the entire dataset.

In Spark, the dataset is represented by a Resilient Distributed Dataset (RDD). It is a programmatic construct that represents the data distributed between the different compute nodes. The user has the possibility to apply different operations to the RDD like map, filter, sample etc. This allows to nicely hide the complexity of distributed processing.

However, the distributed processing of the RDD introduces a synchronization cost: the framework has to wait for every compute node to finish an operation before moving to the next operation.

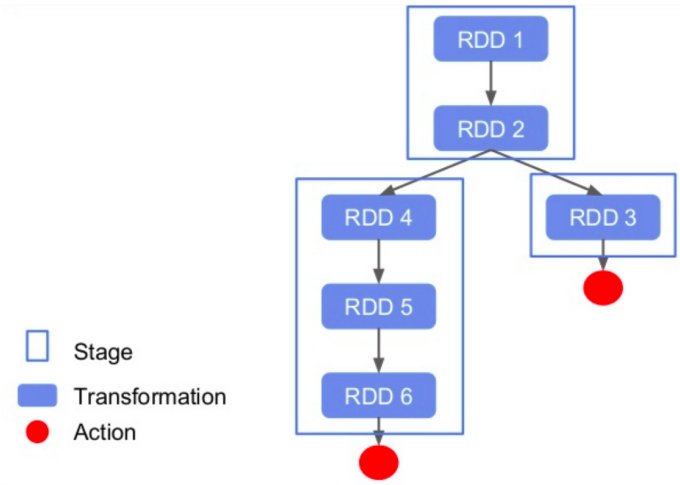

Figure 2.2: Example of a Directed Acyclic Graph in Spark [58].

To reduce this cost, Spark uses a lazy execution approach: it makes a distinction between transform operations (like map, filter and sample) and actions (like count, save or collect). When the user applies a transform operation to a RDD, it is actually not executed by the framework: it is simply added to an execution plan. When the user applies an action on a RDD, the execution plan is evaluated by the framework and optimized to reduce

6. `https://spark.apache.org/`

synchronization and execution time in general, and then the different operations are actually applied to the data.

The execution plan is thus a Directed Acyclic Graph (DAG): for each RDD, it indicates which transformation should be applied to obtain the next RDD, and finally to obtain the result. An example of Spark DAG is presented in Figure 2.2.

Part I

Fundamental algorithms

Chapter 3

Building k-nn graphs from large text datasets

3.1 Introduction

As stated previously, k-nn graphs are powerful tool for data analysis. However, as we show below, until recently no algorithm was able to quickly build a k-nn graph from a large text dataset. Hence we present here a new algorithm called NNCTPH. The algorithm first bins the input text data into buckets, then computes the subgraph inside each bucket and finally merges all subgraphs together.

The binning step relies on a variant of Context Triggered Piecewise Hashing (CTPH), a hashing function that tends to produce the same hash for similar input strings. However, we develop a custom CTPH function that allows to control the number of buckets. Moreover, our hashing function is designed to produce subgraphs that can be reconnected into one single graph.

The algorithm is based on the MapReduce programming model and can be executed on scalable computing frameworks such as Hadoop. Furthermore, it uses a single job (as opposed to iterative algorithms that usually require multiple jobs) which facilitates the task of the resource scheduler of systems like Hadoop or Spark. We experimentally test the algorithm on different datasets consisting of the subject of spam emails. We test the influence of the different parameters of the algorithm on the number of computed similarities, on processing time, and on the quality of the final graph. We also compare the algorithm with a sequential and a MapReduce implementation of NN-Descent, an efficient k-nn graph building algorithm, and show that NNCTPH largely outperforms state of the art approaches in terms of run-time.

The rest of this Chapter is organized as follows. In Section 3.2 we present existing algorithms to build a k-nn graph, and algorithms that perform nearest neighbor search in general. In Section 3.3 we present the implementation details of our algorithm. In Section 3.4 we show experimental results, and compare our algorithm with a sequential and a MapReduce implementation of NN-Descent. Finally, in Section 3.5 we present our conclusions.

3.2 Related work : building k-nn graphs

Different approaches exist to build a k-nn graph. The naive method, also called linear search, uses brute force to compute all pairwise similarities. Then, for each node, the algorithm keeps only the k edges with the highest similarity. This method has a computational cost of $O(n^2)$ and is thus very slow, even implemented in parallel.

Therefore multiple algorithms have been proposed in the literature to speedup the process. Some of them tolerate incorrect edges to further speedup the building process and produce an approximate graph, while others produce an exact graph. In both cases, these building algorithms are closely related to nearest neighbor search algorithms.

But when it comes to building a k-nn graph from a big unstructured text dataset, where each node consists of a string, none of these offers an efficient solution.

3.2.1 Exact algorithms

One efficient way to build a k-nn graph consists in iteratively using a nearest neighbor search algorithm to find the neighbors of all nodes in the dataset.

The nearest-neighbor search problem is formally defined as follows: given a set S of points in a space M and a so-called query point $q \in M$, find the closest point in S to q. The k-nn search is a direct generalization of this problem, where we need to find the k closest points.

k-nn search is also a method used for classification and regression. In k-nn classification, the algorithm assigns to an object the class which is most common amongst its k nearest neighbors in the training set. In the case of regression, the value computed by the algorithm is the average value of the k nearest neighbors of the object in the training set.

Nearest neighbor search algorithms usually use some kind of index to speedup the search. These techniques usually rely on the branch and bound algorithm, and the index is used to partition the data space. For example, a k-d tree [13], that recursively partitions the space into equally sized sub-spaces, can be used to speedup neighbor search, like proposed in [74]. R-trees [43] can also be used for euclidean spaces. In the case of generic metric spaces, vantage-point trees [41], also known as metric trees [99], and BK-trees [18] can be used. In [26], the authors present a distributed k-nn graph building algorithm that uses a shared memory architecture to store a kd-tree based index. But these approaches are hard to implement in parallel on a BSP architecture like MapReduce.

In [79], the authors propose two algorithms that first build an index of the dataset to reduce the number of distances that have to be computed:

— A recursive partition based algorithm: In the first stage, the algorithm builds the index by performing a recursive partition of the space. In the second stage, it builds the k-nn graph by searching the k-nn of each node, using the order induced by the partitioning.

— A pivot based algorithm: The algorithm first builds a pivot index. Then, the k-nn graph is built by performing range-optimal queries improved with metric and graph considerations.

Both algorithms are not limited to euclidean space and support metric spaces, which makes them suitable to build a k-nn graph from text data. For example, the authors tested the algorithms using a text dataset and edit distance as measure of similarity.

Finally, some algorithms use Locality-Sensitive Hashing (LSH), like [86]. LSH is originally a method used to perform probabilistic dimension reduction of high-dimensional data. The basic idea is to hash the input items such that similar items are mapped to the same bucket with a high probability. At the opposite of conventional hash functions, such as those used in cryptography, the goal is to maximize the probability of collision between similar items. But LSH functions are defined only for some similarity measures (l_p, Mahalanobis distance, kernel similarity, and χ^2 distance). The algorithms relying on LSH can thus not be used to build a k-nn graph from text data using an edit distance (Levenshtein distance) or another similar function (weighted Levenshtein distance, Jaro-Winkler distance, Hamming distance) as a similarity metric.

3.2.2 Approximate algorithms

In a lot of cases, to achieve a higher speedup, the designed algorithms focus on building an approximate k-nn graph.

A versatile algorithm to efficiently compute such a graph is described in [37]. The algorithm, called NN-Descent, starts by creating edges between random nodes. Then, for each node, it computes the similarity between all neighbors of the current neighbors, to find better edges. The algorithm iterates until it cannot find better edges. The main advantage of this algorithm is that it works with any similarity measure. Dong et al. experimentally found the computational cost of the algorithm is around $O(n^{1.14})$.

The paper also proposes a MapReduce version of the algorithm. Internally, the algorithm works with a kind of adjacency list called neighbors list. This structure holds the candidate neighbors of a node. The algorithm first creates a random neighbors list for each node. Then each iteration is realized with two MapReduce jobs. First, the mapper emits the initial neighbors list, and reverses the neighbors list to produce and emit new candidate neighbors. For each node, the reducer merges all candidate neighbors to produce a new extended neighbors list. In the second job, the mappers compute and emit the pairwise similarity between all elements of each neighbors list. Finally, the reducer merges the neighbors of each key node, keeping only the k neighbors with the highest similarity. A sequential C++ implementation of the algorithm is also available under the name KGraph [36].

Various authors propose approximate algorithms relying on locality-sensitive hashing.

In [48], the authors propose a MapReduce algorithm that first bins the scale-invariant feature transform (SIFT) description of images into overlapping pools. The algorithm then computes the pairwise similarity between images in the same bucket to build the k-nn graph of images. For binning, the algorithm uses MinHash, a variant of LSH that uses Jaccard coefficient as similarity measure.

Similarly, in [106], the authors use LSH to divide the dataset into small groups. Then, inside these small groups, the algorithm uses NN-Descent to build the k-nn graph. As groups are not overlapping, the constructed graph is a union of multiple isolated small graphs. To bind

the final graph, and improve the approximation quality, the division is repeated several times to generate multiple approximate graphs, which are finally combined to produce the final graph.

Furthermore, the authors propose a method to produce equally sized groups, thus alleviating the computational cost of skewed data. They first project the item's hash code on a random direction. Then they sort items by their projection values. Finally, they divide this sequence of items into equally sized buckets. Doing so, the items with same hash code still fall in the same bucket with a high probability. Finally, the authors show experimentally that their algorithm is much faster than existing algorithms, including NN-Descent, for similar quality of the built graph.

When it comes to building a k-nn graph from a big unstructured text dataset, none of these algorithms offer an efficient solution. Algorithms that rely on indexes are hard to implement in parallel on a shared nothing architecture like MapReduce. LSH functions are defined only for some similarity measures (l_p, Mahalanobis distance, kernel similarity, and χ^2 distance). The algorithms relying on LSH can thus not be used to build a k-nn graph from text data using edit distance (Levenshtein distance) or any similar distance metric (weighted Levenshtein distance, Jaro-Winkler distance, Hamming distance) as a similarity metric.

In the case of NN-Descent, the MapReduce (MR) version of the algorithm requires two MR jobs per iteration, and multiple iterations to converge. Moreover, the algorithm requires to read and write a lot of data on disk between jobs. Although the sequential version of the algorithm proved to be very efficient, these constraints make it inefficient when implemented in parallel. This will be confirmed by the experimental tests presented below.

3.3 NNCTPH

As no current algorithm is suited for building a k-nn graph from a big text dataset, we propose here a new algorithm. The algorithm requires a single iteration and a single MapReduce job, and it does not rely on a shared index. Internally, it uses a specific hashing scheme, called Context Triggered Piecewise Hashing (CTPH) to bin the input data into buckets. Hence we call the algorithm NNCTPH.

3.3.1 Context Triggered Piecewise Hashing

Context Triggered Piecewise Hashing (CTPH), also called Fuzzy Hashing or ssdeep [57], is a hashing function that tends to produce the same hash for similar input strings. It was originally developed by Tridgell as a spam email detector called SpamSum [98]. The algorithm is used to build a database of hashes of known spams. When a new email is received, its hash is computed, and compared with the spam database. If a similar hash is found, the incoming email is considered as spam, and discarded.

The algorithm works by splitting a character string in chunks of variable length. The end point of a chunk is determined by a rolling hash. This rolling hash is based on the Adler-32 checksum used in the zlib compression library [5]. It uses a window of 7 characters

that slides over the input string. By using a rolling hash, the algorithm can perform auto resynchronisation if characters are inserted or deleted between two strings. If the value of the rolling hash matches a given value, the end of the current chunk is found.

As the final hash of the input string is a sequence of characters that corresponds to Base64 encoding, each chunk is hashed into a single character out of 64 possible letters using the Fowler/Noll/Vo (FNV) hash function [40].

In [56], ssdeep is used to identify almost identical files to support computer forensics.

3.3.2 NNCTPH

We present here the design of NNCTPH. As shown in Algorithm 1, the algorithm requires a single MR job. In the map phase, the algorithm uses a modified CTPH function to produce a hash of each input string. This hash value is then used to bin the string into a bucket. Each reduce task builds a k-nn graph of the strings in the bucket. We experimentally found that, for small datasets, the naive brute force method requires less computations and processing time than sequential NN-Descent. Therefore, if the number of strings in the bucket is smaller than a given threshold θ, the reduce task uses brute force, otherwise it uses NN-Descent.

To control the number of buckets, and hence the number of strings per bucket, we modify the original CTPH function to: *i)* produce a hash of variable size; and *ii)* use only a subset of letters in the hash, instead of the 64 original letters.

Moreover, if we only emit each string once, we end up with a series of unconnected subgraphs, as each string is binned into a single bucket, and no edges are created between the strings of different buckets. To reconnect the graph, in the map phase the algorithm creates a longer hash (using a coefficient we call *stages*) and emits the input string once for each subpart of the hash.

The example on Figure 3.1 shows that the hash of `Lorem` is `ABC`. The original string is emitted twice by the mapper: once for bucket `AB`, and once for bucket `BC`. In this way, we can expect that the reduce task for bucket `BC` will produce edges to strings located outside bucket `AB`, hence reconnecting the subgraphs.

The algorithm thus requires three parameters: the number of stages (s), the number of characters of emitted hash (c), and the number of letters used to produce the hash (l). For each stage, the input strings are binned into l^c buckets, and the modified CTPH function must produce hashes with a length of $c + s - 1$ using an alphabet of l letters.

These parameters have an impact on the quality of the graph, on the quantity of data that has to be shuffled, on the parallelism of the algorithm, on the quantity of RAM required by the reducers, and on the number of similarities to compute.

The number of buckets produced by the hashing function is l^c. If we assume the input strings are uniformly distributed over the buckets (if data is not skewed), the number of strings per bucket is $\frac{n}{l^c}$. Dong *et al.* [37] experimentally found that the computational cost of NN-Descent is around $O(n^{1.14})$. As we use NN-Descent inside the buckets to build the subgraphs, the number of similarities to compute is: $O(s \cdot l^c \cdot (\frac{n}{l^c})^{1.14})$. If we choose l and c such that the number of strings per bucket (n/l^c) is constant (with a number of

Algorithm 1 NNCTPH

```
Input:
s the number of stages
c the number of characters of emitted hash
l the number of letters used to produce the hash

procedure MAP(string)
    h = CTPH(string, s, c, l)
    // string is emitted s times
    for i in 0..s do
        // As key, we concatenate the stage i and
        // a substring of c characters of the hash
        // starting at i^th character
        key = i _ SUBSTRING(h, i, c)
        EMIT(< key, string >)
    end for
end procedure

procedure REDUCE(key, < strings >)
    if SIZE(strings) < θ then BRUTEFORCE
    else NNDESCENT
    end if
    for string in strings do
        // Emits the k strings from this bucket
        // that have the highest similarity with string
        EMIT(< string, EDGES(string, k) >)
    end for
end procedure
```

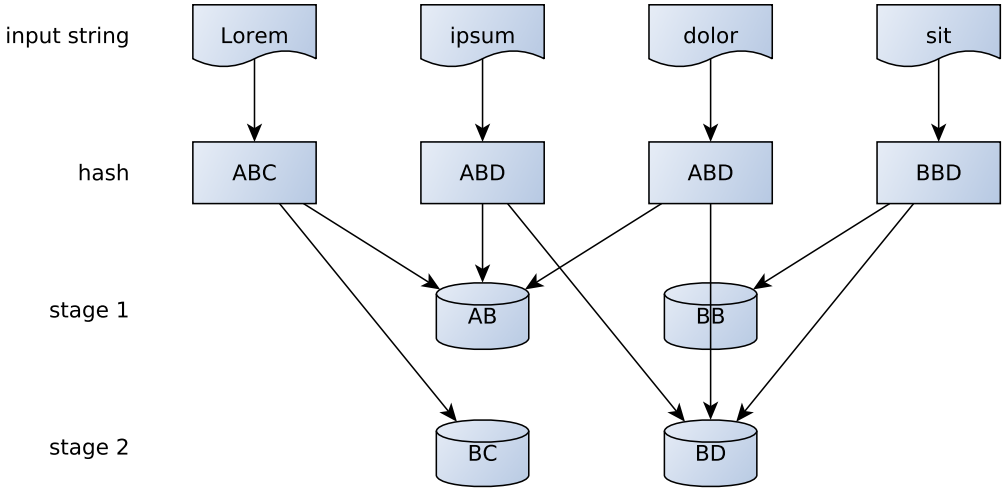

Figure 3.1: Example of how strings are binned when using two stages and hashes of two characters.

buckets proportional to the size of the dataset), this means that the computational cost of our algorithm is proportional to the number of buckets, and thus proportional to the size of the dataset. This is a lower bound. If the input data is skewed, which is the case of our test dataset, the total number of similarities to compute is higher.

The number of stages s will also have an impact on the quality of the final graph: if more stages are used, the same string will be emitted multiple times. The probability to discover correct edges will thus also rise. The number of stages is also directly proportional to the quantity of data to shuffle and transmit over the network: data to shuffle $= s \cdot n$ where n is the size of the dataset. Using a higher number of stages will thus slow down the algorithm.

In the next section, we perform a sensitivity analysis of the effects and interactions of these parameters.

3.4 Experimental evaluation

To analyze the performance of our algorithm, we implement it using Hadoop MapReduce and test it on datasets containing the subject of 200.000 spam emails. This dataset is a sample of spams collected by Symantec Research Labs in 2010. It is mainly used to improve spams signature definitions, and to analyze trends in spam campaigns. We also compare it against our Hadoop MapReduce implementations of NN-Descent and of the brute-force method. All algorithms are executed on a cluster of 20 worker nodes, each equipped with a four-core processor, 8GB of RAM, and four 1Gb ethernet cards.

To compute the similarity between spam subjects, we use the Jaro-Winkler distance [103]. This measure of string similarity is normalized such that zero equates to no similarity and one is an exact match.

To measure the accuracy of the output of each algorithm, like in [37], we use *recall*, which is

the ratio between the number of correct edges found by the approximate algorithm (where the ground truth are the edges found by the naive algorithm) and the total number of edges created by approximate algorithm.

The accuracy of information retrieval algorithms can also be measured using *precision*, the fraction of relevant documents among the retrieved documents.

In the case of a graph building algorithm, the number of created edges is defined by k. Hence, the number of edges created by an approximate algorithm (corresponding to the number of retrieved documents in the context of information retrieval) equals the number of edges created by a brute-force algorithm (which, in the context of information retrieval, corresponds to the number of relevant documents), and thus precision equals recall.

3.4.1 Number of stages

We first test the influence of the number of stages used to run the algorithm. The number of stages is the number of times each input string will be emitted by the mapper. We use this coefficient to reconnect the different subgraphs produced by the reducers.

We use NNCTPH to build a 10-nn graph from our dataset. We use hashes of two characters, with 32 possible letters. We thus create 1024 buckets, and we let the number of stages vary between one and ten. The quantity of data that has to be shuffled and transmitted over the network is directly proportional to the number of stages, which is confirmed by our experiments. Using a higher number of stages will thus slightly slow the algorithm down. At the same time, this will distribute the same input string into more buckets, thus increasing the probability to find correct edges.

The resulting running time and recall are shown on Figure 3.2. As we can see, as little as two stages suffice to correctly discover 50% of edges in the dataset. Increasing the number of stages increases the running time, as expected, but has only a limited effect on recall.

There is here a clear effect of diminishing return due to the mismatch between the CTPH function and the measure of similarity we use to build the graph: CTPH is a general purpose LSH function and there is no guarantee that items that are binned together by NNCTPH are actually the most similar according to Jaro-Winkler. This prevents NNCTPH from achieving a high recall.

3.4.2 Number of buckets

We now study the influence of the number of buckets on processing time and on the quality of produced graph. We have two ways to modify the number of buckets: varying the length of the hash, and varying the number of letters used to produce the hash. Therefore we run three series of tests. For each series, we use NNCTPH to build a 10-nn graph from our spam dataset. Given the results of our previous section, we use two stages as this offers the best trade-off between running speed and recall. In the first and second series, we use respectively one and two characters, and we let the number of possible letters used to compute the hash vary. In the third series, we use a fixed number of possible letters (two), and we let the number of characters of the hash vary. These values are summarized below.

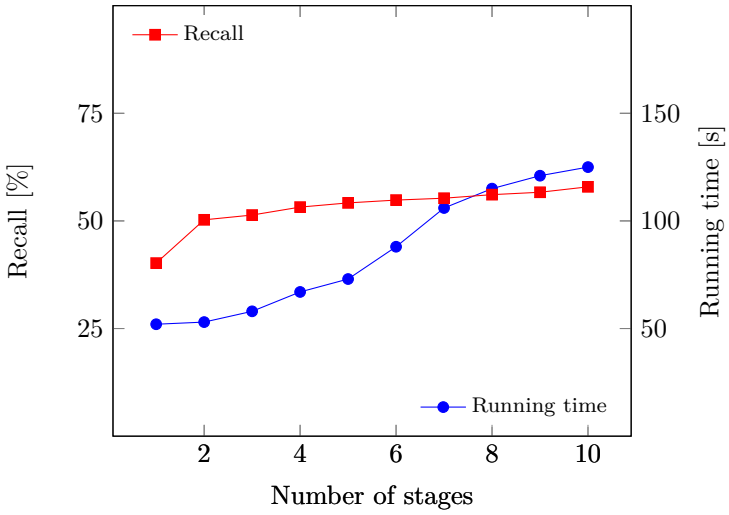

Figure 3.2: Influence of the number of stages on the running time and recall of NNCTPH.

	S1	**S2**	**S3**
Characters	1	2	7 to 12
Letters	20 to 44	4 to 40	2
Buckets	20 to 44	16 to 1600	128 to 4096

The resulting running time and recall of each series of experiments are displayed on Figure 3.3. As we can observe, the running time and recall both tend to decrease when the number of buckets increases, but the impact on recall is quite limited.

3.4.3 Comparison with NN-Descent

For this work, we also implement a complete MapReduce version of NN-Descent, such that we can compare it to our NNCTPH algorithm [1] The main idea of the algorithm is to iteratively improve an initial, random k-nn graph, by "swapping" the neighborhood of each node, searching for similar candidates among its two-hop neighborhood. Increasing the number of iterations allows the algorithm to converge to better and better approximations of the k-nn graph, at the cost of an increased convergence time.

We initialize the algorithm by building a random k-nn graph: [2] each node of the graph (*i.e.* a data item) is assigned k randomly chosen neighbors. The algorithm then iterates with "map" and "reduce" phases, which are illustrated in Algorithm 2.

In the "map phase", the algorithm processes independently each pair consisting of a node and its k neighbors. It uses all the $k+1$ nodes and computes all pairwise similarities. It emits triplets consisting of two nodes and their similarity.

Thanks to the MapReduce framework, the reducer receives a single node, and the list of other nodes for which a similarity was computed by the different mappers. Then each node

1. Although an Hadoop MapReduce version of NNDescent is discussed in [37], we are not aware of any experimental validation of it.
2. We omit the pseudo-code of this phase, as it is trivial.

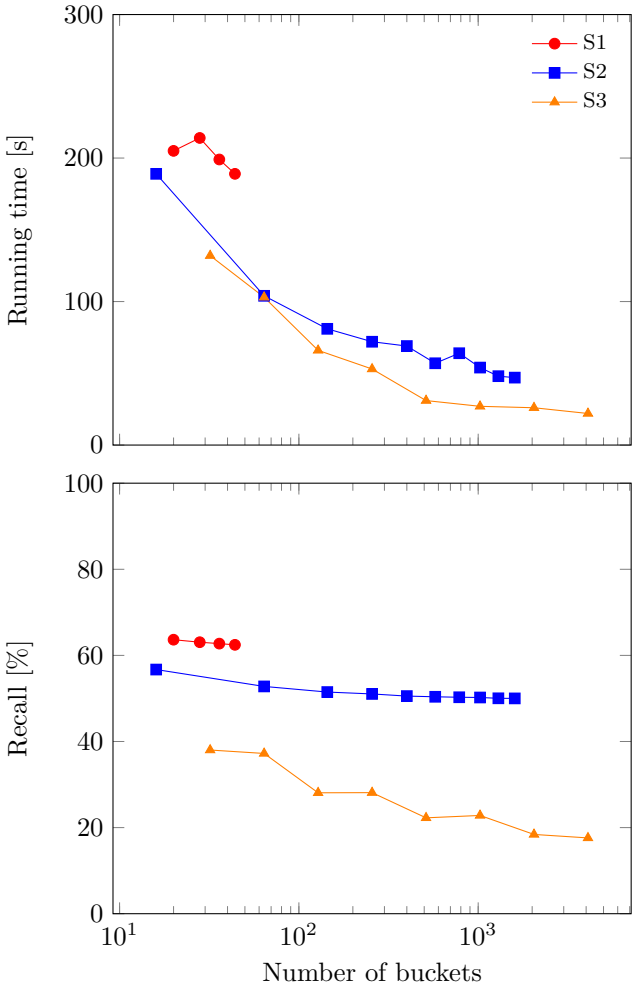

Figure 3.3: Influence of the number of buckets on the running time and recall of NNCTPH

selects its top-k similar nodes, and produces a new approximated k-nn graph that is used as an input for the next iteration of the algorithm. Note that `NeighborList` is a compound data structure, made of `(Node, Similarity)` pairs.

Algorithm 2 k-nn graph construction

```
procedure MAP(Node n, NeighborList(n))
    for all u in NeighborList(n) ∪n do
        for all v in NeighborList(n) ∪n\u do
            EMIT(u, (v, similarity(u, v)))
        end for
    end for
end procedure

procedure REDUCE(Node n, List[Node u, Similarity s] l)
    EMIT(n, ORDERDESCENDING(l).LIMIT(k))
end procedure
```

To compare MR NN-Descent to our NNCTPH algorithm, we run NN-Descent on our dataset, and for each iteration we measure the total running time and recall. The results are shown on Figure 3.4. On the same Figure, we also present the results of previous experiments. As we can see, in some cases NNCTPH runs 6 times faster than NN-Descent for the same quality of produced graph, but the attainable recall is limited.

This is mainly due to the principle of binning itself. At some point, the hashing function has to produce different hashes for different input strings. This means that two similar strings, that differ by only one letter, may receive different hashes. Therefore they will be binned into different buckets, which makes the creation of an edge between them impossible. We mitigate this effect using multiple stages, but the resulting attainable recall is still limited to roughly 50%, as shown on Figure 3.2. We can also reduce this effect by using less buckets, and more strings per bucket, but this increases the computational cost of the algorithm, as shown on Figure 3.3, and reduces the parallelism of the algorithm. A possible solution to increase recall would be to use different hashing functions in parallel. Furthermore, even if we have an idea of how many edges were correctly discovered by our algorithm, we would like to know which edges are correctly detected, and which ones are not. These are left as future work.

3.4.4 Scalability

We now test how the algorithm behaves when the size of the dataset increases. Therefore we use other datasets with up to 800.000 spam emails. Based on previous experiments, we use two stages, hashes of two characters, and we tune the number of letters used so that each buckets receives an average of 200 spams, as summarized below.

	T2	**T4**	**T6**	**T8**
Spams (x1000)	200	400	600	800
Characters	2	2	2	2
Letters	32	45	55	64
Buckets	1024	2025	3025	4096

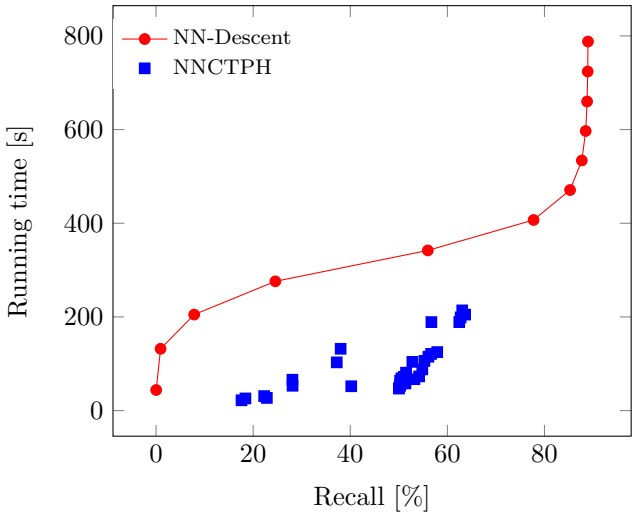

Figure 3.4: Comparison of NN-Descent and NNCTPH algorithms

We also compare NNCTPH with our MR implementation of NN-Descent and with our sequential implementation of NN-Descent. For both algorithms, we chose parameters that deliver approximately the same recall. The resulting running times and recalls are displayed on Figure 3.5. As we can see, the recall achieved by NNCTPH with these parameters is very stable, and the running time rises very slowly. As a result, the bigger the dataset is, the higher the speedup with respect to MR NN-Descent. With our dataset of 800,000 spams, we reach a speedup of nearly an order of magnitude for the same quality of the final graph. Clearly here MR NN-Descent suffers from its iterative structure, which is not well suited for the MR framework, and requires a lot of slow disk I/O operations.

3.5 Conclusions

In this Chapter we presented NNCTPH, a MapReduce algorithm that builds an approximate k-nn graph from large text datasets. We used datasets containing the subject of spam emails to experimentally test the influence of the different parameters of the algorithm on the quality on processing time and on the quality of the final graph. We also compared the algorithm with a sequential and a MapReduce implementation of NN-Descent. For our datasets, the algorithm proved to be up to ten times faster than the MapReduce implementation of NN-Descent, for the same quality of produced graph. Moreover, the speedup increased with the size of the dataset, making NNCTPH a perfect choice for very large text datasets.

3.6 Contributions

Our contributions are the following:

— Thibault Debatty, Pietro Michiardi, Olivier Thonnard, and Mees Wim. Scalable graph building from text data. In *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming*

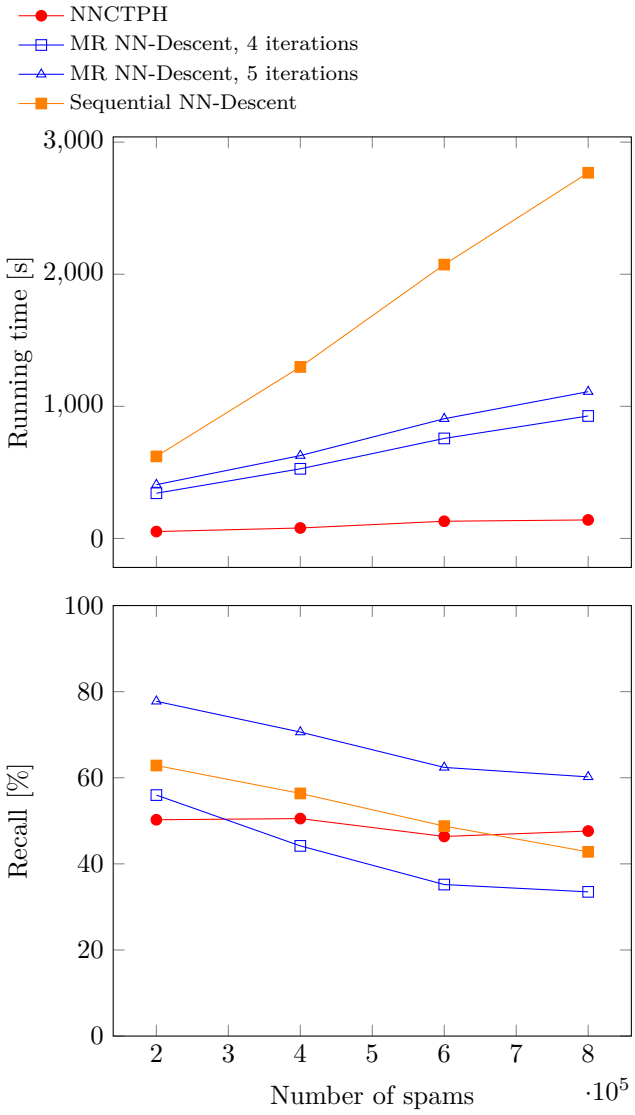

Figure 3.5: Comparison of NNCTPH and NN-Descent for larger datasets

Models and Applications, BigMine '14, 2014

— T. Debatty, P. Michiardi, O. Thonnard, and W. Mees. Building k-nn graphs from large text data. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 573–578, Oct 2014

— Our Hadoop implementation of NN-Descent and NNCTPH is available on GitHub: `https://github.com/tdebatty/hadoop-knn-graph`

— As a preparatory work for this subject, we implemented and compared a dozen string similarity algorithms (including Levenshtein edit distance and siblings, Jaro-Winkler, Longest Common Subsequence, cosine similarity etc.). The resulting code is available as a library on GitHub and Maven Central: `https://github.com/tdebatty/java-string-similarity` and `https://mvnrepository.com/artifact/info.debatty/java-string-similarity`

— As a preparatory work, we also investigated the usage of other LSH algorithms to build k-nn graphs. The implemented LSH algorithms are also available as a library on GitHub and Maven Central: `https://github.com/tdebatty/java-lsh`

Our java string library is used by other people for multiple projects, amongst which:

— GraphHopper, an API for route planning and optimization: `https://www.graphhopper.com/`

— OrientDB, a multi-model database that can store and query documents (like NoSQL databases) and relations (like graph databases), to provide different string similarity and distance measures as SQL functions: `https://orientdb.com/` and `https://github.com/orientechnologies/extra-functions`

— Teiid, a data broker that allows applications to use data from multiple, heterogeneous data stores by exposing "virtual databases": `http://teiid.jboss.org/`

— Alchemist, a distributed network simulator: `http://alchemistsimulator.github.io/`

Chapter 4

Online building of k-nn graphs

4.1 Introduction

In the general case, building a k-nn graph requires the computation of $O(n^2)$ similarities, where n is the size of the dataset. Indexes can be used to reduce the number of computations, but in any way building an exact k-nn graph remains a computationally heavy operation.

In the same way, modifying an existing graph by adding or removing nodes is a computationally heavy process. For example, adding a single node requires: 1) to compute the edges of the new node and 2) to update the edges of existing nodes. Each new data point thus requires to compute the similarity between the new point and every node in the existing graph. This is thus very slow, and better alternatives that can achieve higher speedups w.r.t. a naïve approach are truly desirable.

Therefore, in this chapter we propose a fast approximate k-nn graph modification algorithm, which is able to update a k-nn graph by quickly adding or removing nodes. To the best of our knowledge, this is the first algorithm of this kind. Moreover, the algorithm can be run in a distributed, shared-nothing environment to process very large, distributed k-nn graphs. Finally, our algorithm is independent of the similarity measure used to build or query the graph.

The distributed algorithm starts by partitioning the k-nn graph using a balanced k-medoids algorithm. This partitioning is used to improve the nearest neighbour search based on the existing graph. Indeed, the algorithm has two main steps to add a new point to the graph: 1) use the distributed graph to search the k nearest neighbours of the new point and 2) update the graph: these neighbours are used as starting points to search existing nodes for which the new point is now a nearest neighbour. To search the nearest neighbours, inside each partition the algorithm uses a fast sequential graph based nearest neighbour search procedure.

The rest of this chapter is organized as follows. In Section 4.2 we present existing graph based search algorithms and graph partitioning algorithms. In Sections 4.3 and 4.4 we show how we add and remove nodes from the distributed k-nn graph. In Section 4.5 we explain the distributed graph based search algorithm together with the sequential graph based

search procedure we use inside each partition. The distributed search relies on a k-medoids based partitioning method that we present in Section 4.6. In Section 4.7 we perform an experimental evaluation, where we perform a parameter study of the algorithm. Finally, in Section 4.8, we present our conclusions.

4.2 Related work

One important step in our algorithm consists in searching the nearest neighbours of the new data point using the existing graph. Therefore we present here existing graph based nearest neighbour search algorithms.

Searching the graph in a distributed fashion requires a specific partitioning of the graph. Hence, we alse present existing graph partitioning algorithms.

4.2.1 Graph based nn-search

The nearest-neighbour search problem (NN search) is formally defined as follows: given a set S of points in a space M and a so-called query point $q \in M$, find the closest point in S to q, according to some similarity metric. The k-nn search is a direct generalization of this problem, where we need to find the k closest points. A lot of algorithms exist to find the k nearest neighbours of a point. They are generally very similar to those used to build a k-nn graph. However, only a few of them rely on an existing k-nn graph to find the nearest neighbours of a query point.

In [44], Hajebi et al. proposed a sequential approximate NN search algorithm that relies on k-nn graphs. The algorithm, called Graph Nearest neighbour Search (GNNS), works by selecting initial nodes at random. For each node, the algorithm computes the similarity between query point and every neighbour. The most similar neighbours are selected, and the algorithm iterates until a depth of search d is reached. It is thus a "hill climbing" algorithm. The most promising nodes are searched first, using the similarity between the query point and the node as a heuristic. It was tested against different datasets. Without taking graph building phase in account, the search algorithm achieved a speedup of up to 80 over linear search, and a speedup of two over randomized KD-tree.

Dong, the co-author of the paper on nn-descent [37], also created a software called KGraph [36] which is able to search the nearest neighbours of a query point using a precomputed k-nn graph. However, the search algorithm used by the program was never published.

4.2.2 Graph partitioning

As will be shown below, performing a distributed graph-based nn search requires a specific partitioning of the data, based on k-medoids clustering. Hence, we present here related existing graph partitioning algorithms.

The classical definition of graph partitioning consists in splitting the graph data between partitions, minimizing the number cross-partition edges, while keeping the number of nodes

in every partition approximately even. Multiple algorithms exist to perform this type of partitioning. In [83], the authors proposed a distributed iterative algorithm that iteratively swaps the partition of two nodes to minimize the number of cuts. The algorithm is heavily based on MPI and requires a lot of communication between all nodes of the graph. In [19], the authors proposed and tested a Bulk Synchronous Parallel (BSP) version of the algorithm which makes it suitable for shared nothing architectures like Apache Spark. In [55], the authors proposed a streaming algorithm, that requires a single iteration to partition the graph. They experimentally compared various heuristics to assign nodes to a partition. They found the best performing heuristic was linear weighted deterministic greedy. This one assigns each node to the partition where it has the most edges, weighted by a linear penalty function based on the capacity of the partition.

As we show in Section 4.5, to improve distributed graph based search, the partitioning scheme should minimize the number of steps between any two nodes in the partition. In this case the partitioning becomes a k-medoids clustering problem. It is a variation of k-means clustering, where the centers are points from the dataset. It also minimizes the sum of pairwise distances, while k-means minimizes the sum of squared Euclidean distances. Just like k-means clustering, various algorithms were proposed in the literature to perform k-medoids clustering, like Partitioning Around Medoids (PAM) [95]. To the best of our knowledge, the most efficient algorithm for performing k-medoids clustering is currently the Voronoi iteration method proposed in [80], which is very similar to the classical Lloyd's algorithm used to compute k-means. However, these algorithms cannot be executed in a distributed environment.

Moreover, until now no balanced version of k-medoids was published, although a few balanced versions of k-means exist. In [69], the authors proposed a method that has a complexity $O(n^3)$, which makes it too complex for large graphs. In [9], the authors proposed the Frequency Sensitive Competitive Learning (FSCL) method, where the distance between a point and a centroid is multiplied by the number of points already assigned to this centroid. Bigger clusters are therefore less likely to win additional points. In [7], the authors used FSCL with additive bias instead of multiplicative bias. However, both methods offer no guarantee on the final number of points in each partition, and experimental results have shown the resulting partitioning is often largely unbalanced. Hence, we present below our own algorithm, that offers a guarantee on the maximum number of items per cluster.

4.3 Adding nodes

The algorithm we propose has two main steps to add a new point to the graph: 1) use the current graph to search the k nearest neighbours of the new point, using the distributed algorithm presented in Section 4.5 and 2) update the graph using the procedure presented in Algorithm 4. The update procedure is actually a propagation algorithm. It starts with the discovered neighbours of the new node, and recursively explores the neighbours of neighbours, up to a fixed depth, to check if existing edges should be modified. In addition, the new node is assigned to the compute node corresponding to the most similar medoid.

Finally, the medoids may be recomputed once a given number of new nodes have been added to the graph. This is actually not mandatory, and depends on the dataset: if the

characteristics of the dataset are fixed over time, adding new nodes will not induce a displacement of the medoids. Otherwise, the medoids update rate should be consistent with the expected rate of change of the dataset. The automatic estimation of the update rate is left as as future work. The complete procedure used to add a new node to the graph is shown in Algorithm 3.

Algorithm 3 Add a node to the graph

Inputs:
graph: current graph
node: a new node

In parallel: ▷ Distributed search
neighbourlist = Search(*graph*, *node*, k)

In parallel: ▷ Update with Algorithm 4
Update(*graph*, *node*, *neighbours*, 0)

medoid = NearestMedoid(*node*) ▷ Shuffle
assign $\langle node, neighbourlist \rangle$ to the compute node
corresponding to *medoid*

Algorithm 4 Update

Inputs:
graph: the current graph
new: the new node to add in the graph
neighbours: the list of nodes to analyse
depth: the current depth
MAX_DEPTH: the maximum depth of exploration

for *node* in *neighbours* **do**
  if *depth* < *MAX_DEPTH* **then** ▷ Recursion
    Update(*graph*, *new*, *node.neighbours*, *depth*++)
  end if
  compute similarity(*node*, *new*)
  if needed, add *new* to the neighbourlist of *node*
end for

The most computation intensive steps of the algorithm are the search and update steps. This latter requires a maximum of $k^{\mathrm{DEPTH}+1}$ similarity computations. To reduce the space requirement of the graph, k is generally kept small. A value of 10 is very often seen. With this value, experimental evaluation shows that a depth of three is sufficient to update the graph. The resulting number of similarity computations (1000) is thus small compared to the size of the graphs targeted by this update algorithm. The computation cost of the algorithm will thus be dominated by the search step, hence the need for a very efficient algorithm.

4.4 Removing a node

When a node n_d is deleted from the graph, some other nodes which have n_d as neighbour have to be updated to assign them a new neighbour. These nodes to update are easily identified by scanning the complete graph. As these nodes to update all had n_d as neighbour, they are very likely to be highly similar to each other. Hence, we do not process them individually, but as a group.

Indeed, a common set of candidates is first identified using a distributed propagation algorithm similar to the one used to update the graph after adding a node: the graph is explored up to a fixed depth, starting from each node to update and from the node to delete.

Finally, for each node to update, the most similar node from the set of candidates is used to replace n_d.

When removing a node, we can expect an average of k nodes will have to be updated. As we also use the node to delete n_d as a starting point for the propagation algorithm, we can expect to find $(k+1)^{\text{DEPTH}+1}$ candidates. The algorithm will thus require to compute $k \cdot (k+1)^{\text{DEPTH}+1} \simeq k^{\text{DEPTH}+2}$ similarities between nodes. Experimental evaluation showed a small DEPTH value is enough to get good results. This represents a huge speed improvement compared to the naïve approach that requires comparing the k nodes to update to the n nodes in the graph and would thus require $k \cdot n$ similarity computations.

4.5 Distributed nearest neighbours search

As stated above, the computation cost for adding a new node to the graph is mainly influenced by the search step. To the best of our knowledge, no algorithm exists in the literature that allows to quickly search the nearest neighbours of a point using a distributed graph. So we present here our own algorithm, which can support any similarity measure, even non metric.

The procedure we propose to perform a k-nn search is actually very simple: the p partitions are searched independently using an efficient graph based sequential algorithm, then the $k \cdot p$ nearest neighbour candidates are filtered to keep the k most similar to the query point. The data exchanged is very limited: the compute nodes send the $k \cdot p$ candidate neighbours to the master, that produces the final output.

The procedure we use inside each partition to search the nearest neighbours of a query point q is inspired by the hill climbing approach presented in [44]: the algorithm selects a random node n from the graph, computes the similarity between q and every neighbour of n, and iterates with the most similar neighbour. While iterating, it keeps a set of the most similar points to q. When a local maximum is reached, the algorithm restarts with another random node. This search algorithm is thus an approximate algorithm, as it does not necessarily find the most similar node in the graph. It does however find the nearest neighbour with a high probability, while analysing only a fraction of the nodes in the graph.

In our sequential search procedure we introduce two additional approximations, which allow to further reduce the number of computed similarities compared to the naïve hill climbing

approach.

The first improvement relies on the observation that by the definition of a k-nn graph, each node only has edges to other very similar nodes. Hence, the increase of similarity at each iteration of the search can be very small. As a consequence, the number of iterations i (the number of nodes to analyse) before finding the nearest neighbours of a query point can be very large. In the worst configuration of the graph, $i = n/k$. This requires computing a lot of similarities, $i \cdot k$. To avoid this situation, the randomly chosen starting node r is skipped if it is situated too far from the query point.

Formally, we keep track of the similarity of the most similar neighbour found so far $s_{\max}$, and we introduce an expansion coefficient $e > 1$. As stated above, when a local maximum is reached, the algorithm restarts with another random node r. We immediately discard r and select a new random node if similarity$(query, r) < s_{\max}/e$. In this way, we avoid analysing a potentially very long chain of i nodes before reaching the neighbourhood of the query point, which would be computationally very expensive ($i \cdot k$). Instead, we focus on exploring the vicinity of the query point (the nodes for which similarity with query point is at least $s_{\max}/e$).

Secondly, to further reduce the number of computed similarities, for each analysed node, we eagerly iterate using the first neighbour that provides an increase in similarity compared to the currently analysed node. We thus try to analyse only a few of the k neighbours of the analysed node. The improvement provided can be calculated for an Euclidean space of d dimensions and uniformly randomly distributed data points. For any node, let H be the number of neighbors that have a higher similarity with the query point, and L the neighbors that have a lower similarity. In such a space, observe that for any node, out of 2^d directions, only one leads in the direction of the query point. For example, for $d = 1$, they are two possible directions (left and right), and only one goes in the direction of the query point. As each node has k neighbors, on average only $\mathbf{E}(H) = k/2^d$ edges lead to a node with higher similarity, and $\mathbf{E}(L) = k - k/2^d$ edges lead to a node with lower similarity.

We now compute $\mathbf{E}(S)$, the expected number of similarities that have to be computed for each node. As the improved algorithm iterates as soon as a node with higher similarity is found, this is actually equivalent to the bag of balls problem: imagine a bag with w white balls and r red balls. We search the expected number of white balls to pick before we pick a red ball. Each of the w white balls can be picked instead of a red ball with probability $1/(r+1)$, so we pick an expected number of $w/(r+1)$ white balls. For $\mathbf{E}(S)$ this yields:

$$\mathbf{E}(S) = \frac{\mathbf{E}(L)}{1 + \mathbf{E}(H)} = \frac{k - k/2^d}{1 + k/2^d} \tag{4.1}$$

At the opposite, the original hill climbing approach requires computing k similarities for each analysed node. Hence, this results in an expected speedup of $k/\mathbf{E}(S)$ compared to the original hill climbing approach. The resulting speedup for some values of k and d is shown in Table 4.1, which shows a substantial speedup can be achieved in some cases. The drawback is that in some cases the algorithm might pick a neighbour that improves the similarity, but without maximizing it (there was another neighbour which is more similar to the query point). However, a node has edges only to other highly similar nodes, hence those neighbours are also similar to each other. The difference of similarity improvement when

Table 4.1: Speedup compared to classical hill-climbing search achieved by iterating as soon as a node with higher similarity is found, for various values of k and d (dimensionality).

k	4	10	10	10
d	2	2	3	4
$\mathbf{E}(H)$	1	2.5	1.25	0.625
$\mathbf{E}(L)$	3	7.5	8.75	9.375
$\mathbf{E}(S)$	1.5	2.14	3.88	5.77
$\mathbf{E}$(speedup)	2.66	4.66	2.57	1.73

choosing a sub optimal neighbour remains thus limited, and globally the efficiency of the algorithm increases.

These additional approximations allow us to reduce the number of computed similarities, while they do not prevent the convergence of the hill climbing approach. The demonstration of this latter can be found in [44] and is not repeated here.

To increase the efficiency of the distributed search, the graph is also partitioned in a way that maximizes the probability of finding the most similar nearest neighbours. This partitioning scheme aims to fulfil two conditions: 1) the distance (measured as the number of edges) between two nodes in the same sub-graph should be as low as possible, to maximize the probability of quickly finding "good" candidates and 2) the number nodes in each sub-graph should be similar to balance the work load between compute nodes during the search.

The first condition corresponds to the definition of k-medoids clustering, a variation of k-means clustering where the centres are data points. It also minimizes the sum of pairwise distances, while k-means minimizes the sum of squared Euclidean distances.

The impact of the partitioning on the search algorithm is actually very dependent on the geometry of the graph. Let p_g be the number of clusters that the graph naturally contains (the number of connected components). If the algorithm splits the graph into $p = p_g$ partitions, the partitioning used by the distributed search algorithm will reflect the natural partitioning of the graph. In this case, it will ensure that the starting points used by the sequential search will be nicely distributed over the different clusters. We can thus expect the distributed algorithm to produce better results then the sequential algorithm.

In other cases (when $p \neq p_g$), the clusters of the graph will be distributed over the p partitions. The partitioning will thus potentially cause a lot of cross-over edges. Statistically, these will shorten the chain of nodes that the sequential search procedure can run through and cause the algorithm to restart from a random node in the partition. It will eventually reduce the probability to find the nearest neighbours of the query point.

4.6 Balanced k-medoids partitioning of a distributed k-nn graph

We present here the procedure we use to partition the distributed k-nn graph. It is inspired by the Voronoi iteration method used by the classical k-means algorithm and in [80]. It has the additional advantage that it offers a guarantee on the maximum size of each cluster,

which guarantees a fair distribution of the work load between the compute nodes.

In our case, we wish to cluster the graph in order to optimize the distributed k-nn search. Hence, the distance measure used to assign each node to a medoids and to compute the new medoids should be the length of the shortest path in the graph between two nodes. However, computing the shortest path from a node to all medoids requires access to all adjacency lists, which is impossible in a distributed, shared-nothing infrastructure. Therefore, instead of computing the shortest path to every medoid, we use the similarity between the node and every medoid as a heuristic. An intuitive example is to consider the simple case of a uniform distribution over an Euclidean space. In such a space, all edges have the same length. Hence, the most similar medoid, measured as the shortest path from the node to the medoid, is also the most similar medoid, measured using the euclidean distance.

To achieve a balanced distribution of n points between the k clusters (not to confuse with the k edges per node of a k-nn graph), we use a linear weighted deterministic greedy heuristic to assign each point to a cluster. Let $p_0 \dots p_i \dots p_n$ be the set of points to cluster and $m_0 \dots m_j \dots m_k$ the set of medoids at any iteration of the algorithm. Each point p_i is assigned to the cluster $C(p_i)$ corresponding to the most similar medoid weighted by a penalty function $w(m_j, t)$:

$$C(p_i) = \arg\max_{m_j}(\text{similarity}(p_i, m_j) \cdot w(m_j, t)) \tag{4.2}$$

This penalty function is based on the capacity and current size of the cluster in order to penalize large clusters:

$$w(m_j, t) = 1 - \frac{|C(m_j, t)|}{\mathit{capacity}} \tag{4.3}$$

where $C(m_j, t)$ is the cluster corresponding to medoid m_j at time t and *capacity* is the maximum size of each cluster.

Depending on the application, a small size difference can be tolerated between clusters, hence the capacity of clusters is usually computed using an imbalance factor ($\mathit{imbalance} \geqslant 1$):

$$\mathit{capacity} = \frac{n \cdot \mathit{imbalance}}{k} \tag{4.4}$$

A perfectly balanced clustering can be achieved using $\mathit{imbalance} = 1$.

In a distributed shared nothing environment the different compute nodes do not have access to total size of each cluster at time t. Each compute node can only rely on the local number of points already assigned to a cluster $C_L(m_j, t)$.

Hence, to execute the algorithm in parallel we first randomly distribute the input dataset between the c compute nodes. At each iteration, this randomized dataset is used to assign each point using a modified weight function that relies solely on the local size of clusters:

$$w_L(m_j, t) = 1 - \frac{|C_L(m_j, t)|}{\mathit{capacity}_L} \tag{4.5}$$

where $capacity_L$ is the contribution of each compute node to total size of the final cluster:

$$capacity_L = \frac{n \cdot imbalance}{k \cdot c} \tag{4.6}$$

If the dataset is large enough (with respect to c, the number of compute nodes) and randomly distributed, we can assume that the resulting error (relative number of points that are not assigned to the correct cluster) can be kept arbitrarily low.

If the clustering of the data points is perfectly balanced, during the update step the size of each cluster is n/k. For each cluster, computing the new medoid requires to compute every pairwise similarity:

$$\frac{n}{k} \cdot \frac{\frac{n}{k} - 1}{2} \approx O(\frac{n^2}{k^2}) \tag{4.7}$$

The lower bound on the total computational cost for computing the k new medoids is thus

$$O(\frac{n^2}{k}) \tag{4.8}$$

This is completely unacceptable for large datasets. Therefore, we sample the dataset and run the balanced k-medoids algorithm against the downsized data to compute the medoids. We only use the complete dataset once, to assign each node to a medoid using the constraint in equation 5.3, and thereby to a compute node.

It is a well known technique for the family of k-means clustering algorithms to use a sampled dataset to compute the initial centres. The impact of this technique depends naturally on the final goal of the clustering. For our use case, experimental evaluation shows it is perfectly valid and offers the same results as performing multiple iterations of distributed k-medoids clustering with the complete dataset, while it requires the computation of far less similarities.

4.7 Experimental evaluation

To perform the experimental evaluation, we implemented all algorithms using the Spark parallel processing framework. All experiments are run on a cluster consisting of 16 compute nodes plus one master node, each equipped with a quad-core processor and 8GB of RAM memory. Each experiment is repeated 10 times, and we present below the averaged values.

The experiments are run using two datasets, with different similarity measures.

The synthetic dataset consists of points in R^3 which are randomly generated according to a mixture of gaussian distributions. The similarity measure used to build and query the graphs is the classical Euclidean distance.

The SPAM dataset contains the subject of approximatively 1 million spams collected by Symantec Research Labs in 2010. Domain knowledge suggests that the most relevant similarity measure for building and querying the graph built from this dataset is Jaro-Winkler. This dissimilarity measure is similar to classical Levenshtein edit distance, but it

allows character substitution. Moreover, the substitution of two close characters is considered less important then the substitution of two characters that are located far from each other. Jaro-Winkler is not a metric distance as it does not abide by triangle inequality. It confirms our algorithm can also be used with non-metric similarity measures.

4.7.1 Graph quality

We first evaluate the quality of the graph produced by the fast distributed algorithm. Therefore, we build an initial k-nn graph consisting of 50000 nodes using a naïve brute force algorithm. Then we progressively add new points to the graph and regularly compare the updated graph to a graph built from the same points using the brute force algorithm. At each step we count the number of edges that are correct in the updated graph e_c and compute the quality of the graph, as defined below.

If the initial graph has n nodes and we add n_a nodes to the graph, the algorithm has to create $n_a \cdot k$ new edges. We can also expect that a number of edges from the initial graph have to be modified. Hence, the total expected number of modified edges is:

$$\mathbf{E}(e_m) = n_a \cdot k + n \cdot k \cdot \frac{n_a}{n_a + n} \tag{4.9}$$

The expected number of unmodified edges in the final graph is simply $\mathbf{E}(e_u) = (n + n_a) \cdot k - \mathbf{E}(e_m)$. Hence, the quality of the produced graph can be measured as the number of correctly modified edges divided by the expected number of edges that the algorithm should modify:

$$Q = \frac{e_c - \mathbf{E}(e_u)}{\mathbf{E}(e_m)} \tag{4.10}$$

where e_c is the real number of correct edges in the graph. Hence, Q is also the ratio of correct edges in the graph when $n_a \to \infty$:

$$Q = \lim_{n_a \to \infty} \frac{e_c}{k \cdot (n + n_a)} \tag{4.11}$$

The result of this experimental evaluation is shown in Figure 4.1. It confirms that the quality value we defined above is an accurate measure of the correctness of the updated graph. It also shows that the algorithm produces graphs that are highly similar to the exact graphs (produced by a brute force algorithm), although it introduces multiple approximations to reduce the number of computed similarities. The algorithm is however more efficient with the synthetic dataset, that relies on an euclidean similarity measure, than with the SPAM dataset, that uses a non-euclidean similarity measure. This will also be confirmed by following experiments.

4.7.2 Parallelism

The first source of approximation is due to the partitioning of the graph, which is required to execute the nearest neighbour search in parallel. Indeed, increasing the number of partitions

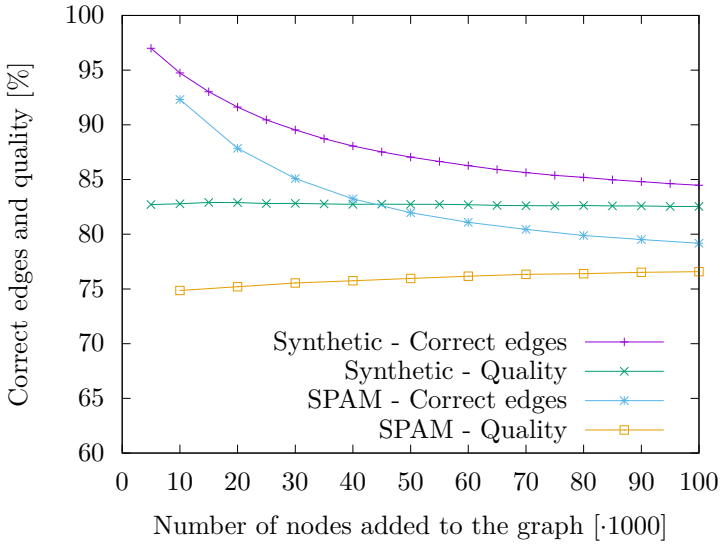

Figure 4.1: Graph quality

reduces the size of each sub-graph, and thus increases the probability for the sequential search procedure to reach the boundary of the sub-graph, which reduces the probability to find the most similar nodes in the graph.

To evaluate the effect of parallelism on the algorithm, we build an initial graph and add a fixed number of nodes while we vary the number of partitions. Hereby we thus also vary the parallelism. We start with one single partition, which means the processing is sequential, and not distributed. For each value, we measure the quality of the produced graph, the time required to add the nodes, and we count the number of times the sequential search procedure has to restart because it reaches the boundary of the partition. The measured values are shown in Figure 4.2.

As we can see, increasing the number of partitions (and thus the parallelism) effectively reduces the amount of time required to add points to the graph. However, once a certain amount of parallelism is reached, the cost of distributing the operations overtakes the speed increase offered by a higher parallelism. This threshold depends mainly on the cost of computing the similarity between points. For the synthetic dataset, for which the similarity is very easy to compute (Euclidean distance in R^3), the best parallelism appears to be 8.

The Figure also confirms that the graph quality decreases with the number restarts due to the partitioning, which increases with the number of partitions. Once again, the effect is more pronounced with the spam dataset.

4.7.3 Update depth

The second approximation lies in the update depth used to modify the edges of existing nodes when a new node is added to the graph. Increasing the update depth will of course increase the quality of the produced graph, but it also requires to compute more similarities. To evaluate its impact we vary the update depth and measure for each one the quality of the produced graph and total number of computed similarities. This last includes the

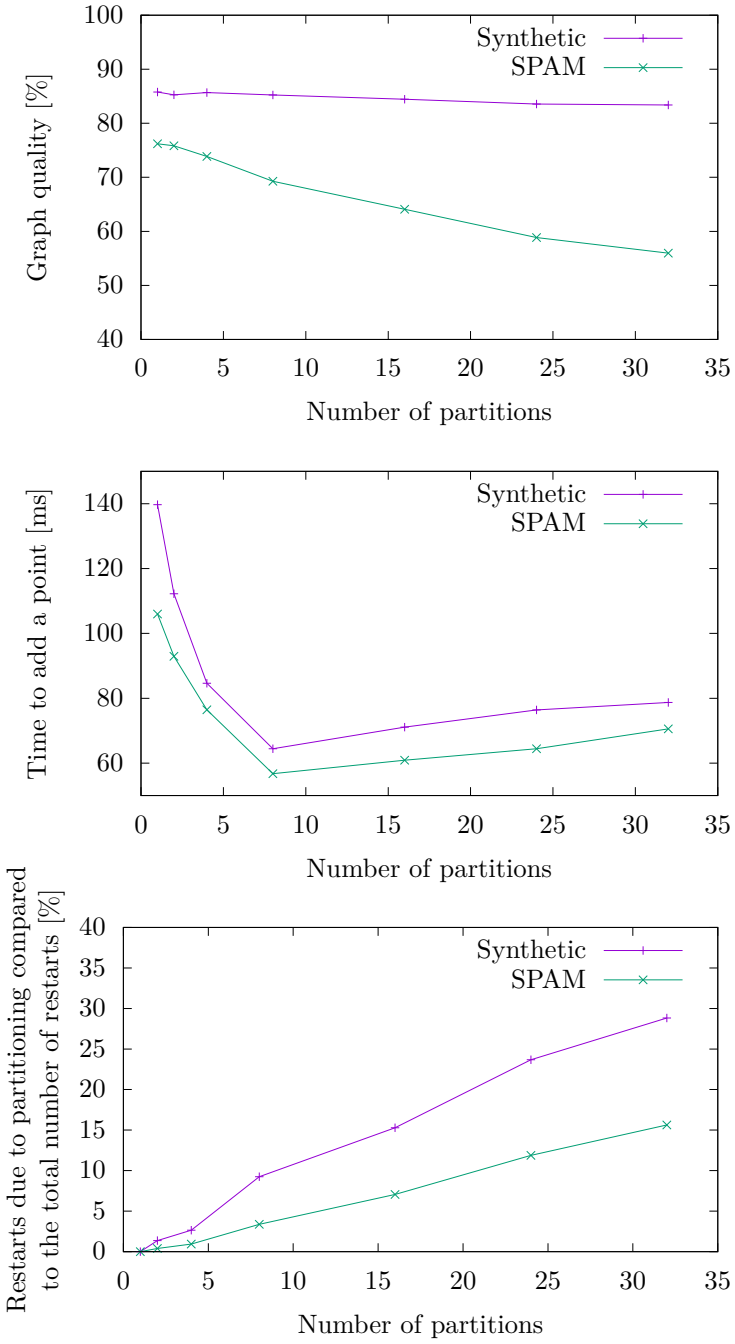

Figure 4.2: Influence of the number of partitions

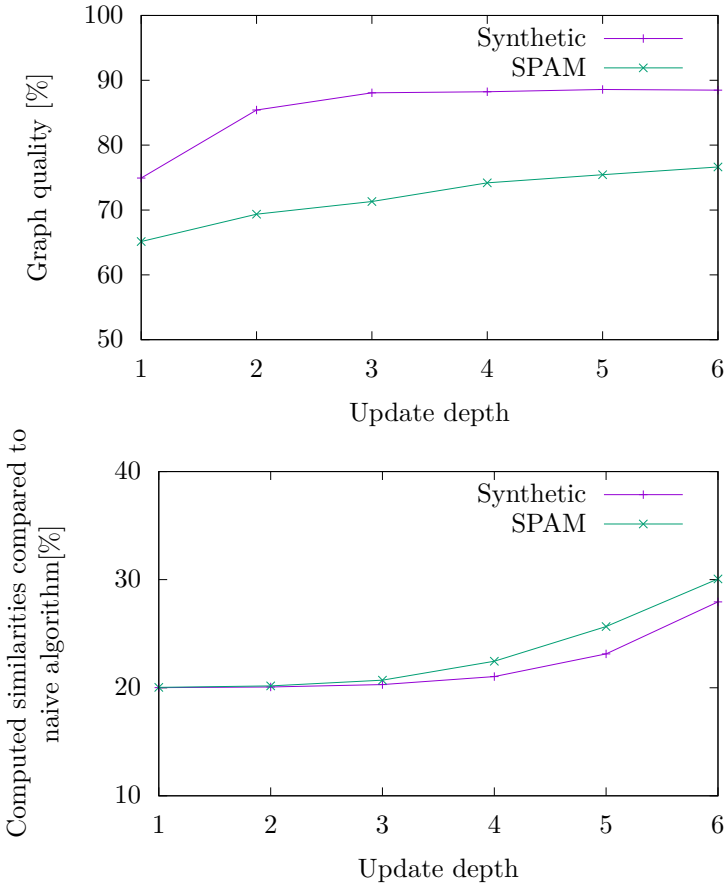

Figure 4.3: Influence of the update depth

number of similarities computed to search the neighbours of the new node, and the number of similarities computed to update the edges of existing nodes in the graph.

As we can see on Figure 4.3, there is a clear effect of diminishing return, and an update depth of three seems to be a good compromise.

4.7.4 Search speedup

Finally, the main source of approximation relies in the sequential search procedure used inside each partition. This one is responsible for finding the nearest neighbours of the new node added to the graph. It is itself influenced by multiple parameters. The first and most important one is the search speedup. As we could expect and is confirmed in Figure 4.4, increasing the search speedup allows to reduce the number of similarities to compute, while the produced graphs remain highly similar to the graphs produced by a brute force algorithm, especially with the euclidean dataset. There is however a minimal number of similarities that have to be computed to update the edges of existing nodes.

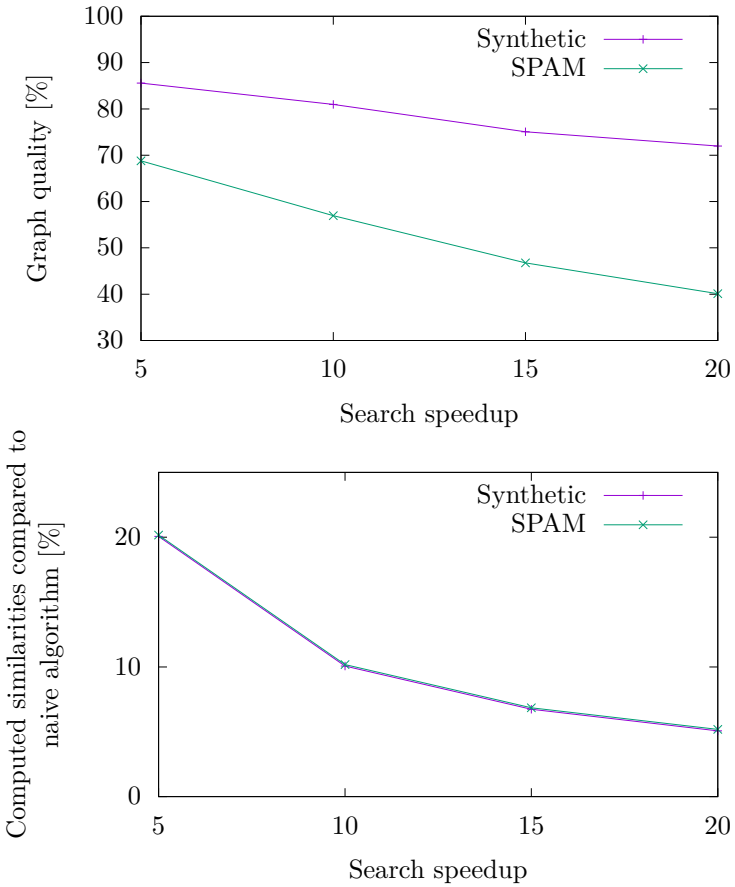

Figure 4.4: Influence of the search speedup

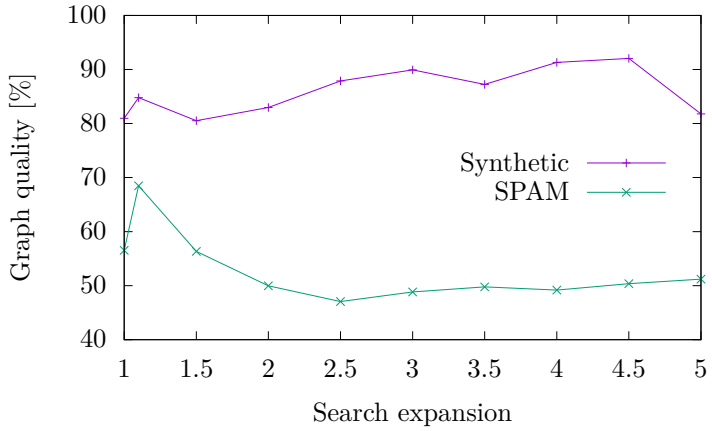

Figure 4.5: Influence of the search expansion

4.7.5 Search expansion parameter

Another important parameter of the sequential search procedure is the expansion parameter. As we can see in Figure 4.5, a well chosen value (1.1 for the SPAM dataset and 4.5 for the synthetic dataset) effectively increases the quality of the search procedure by reducing the number of nodes to analyse, and eventually increases the quality of the produced graph. This parameter is however very dependent on the dataset and can be quite difficult to tune.

4.7.6 Dimensionality and k

Finally, the dimensionality of the dataset and the number of edges per node k also have a strong influence on the search procedure. Remember that the search relies on a hill climbing approach. Hence, if k is inferior to the dimensionality of the dataset, the probability is high that a node has no neighbour that increases the similarity with the query point. This will cause the search procedure to restart, and will eventually reduce the probability to find the nearest neighbours of the query point.

This effect is illustrated in Figure 4.6. We build a synthetic dataset in $\mathbf{R}^{20}$, then build a k-nn graph with different values of k, add a fixed number of nodes to the graph, and finally evaluate the quality of the produced graph. As we can see, increasing k also increases the quality of the produced graph.

Regarding the SPAM dataset, although it is not euclidean and the real dimensionality is not defined, its behaviour is roughly similar to a high dimensionality dataset.

4.7.7 Partitioning

We compare here the quality of the graph produced after adding a fixed number of nodes if: 1) we do not partition the graph and leave the nodes randomly distributed between the compute nodes, 2) we use the complete dataset and run one to ten iterations of k-medoids to compute the medoids and 3) we sample the dataset to compute the medoids and partition the original graph.

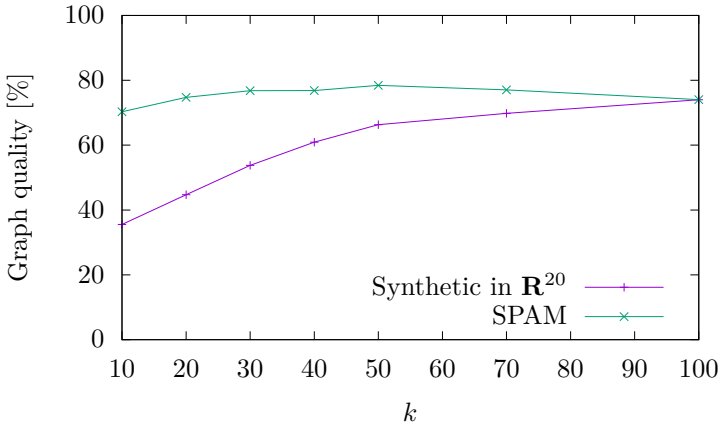

Figure 4.6: Influence of the dimensionality

As we can see on Figure 4.7, the partitioning allows to improve the quality of the graph, and the medoids computed by sampling the dataset eventually produce the same quality of the final graph.

4.7.8 Scalability

We evaluate here the scalability of the algorithm. Therefore, we increase the number of partitions, and keep the size of the graph proportional to the number of partitions (10000 nodes per partition). We add a fixed number of nodes to the graph, and compute the quality of the produced graph. The results in Figure 4.8 show the size of the dataset has a very limited impact on the quality of the produced graph, which makes the algorithm suitable for very large datasets.

4.7.9 Removing nodes

Finally, we evaluate the quality of the produced graph when we remove nodes. In this case, the quality factor is computed as follows. We can expect each node in the graph has an average of k incoming edges. Hence, deleting a single node will require to compute k new edges. When we remove n_r nodes from the graph, the expected total number of modified edges is thus $\mathbf{E}(e_m) = k \cdot n_r$, the expected number of unmodified edges is $\mathbf{E}(e_u) = (n - n_r) \cdot k - \mathbf{E}(e_m)$ and the quality of the graph can be computed using Formula 4.10.

As we can see, the algorithm allows to remove nodes from the graph computing only a few similarities (less than 2% of the number of computations required by the naïve approach). The produced graph is highly similar to the graph produced by a brute-force algorithm, although the quality of the built from the SPAM dataset is once again slightly inferior to the the quality of the graph built from the synthetic dataset.

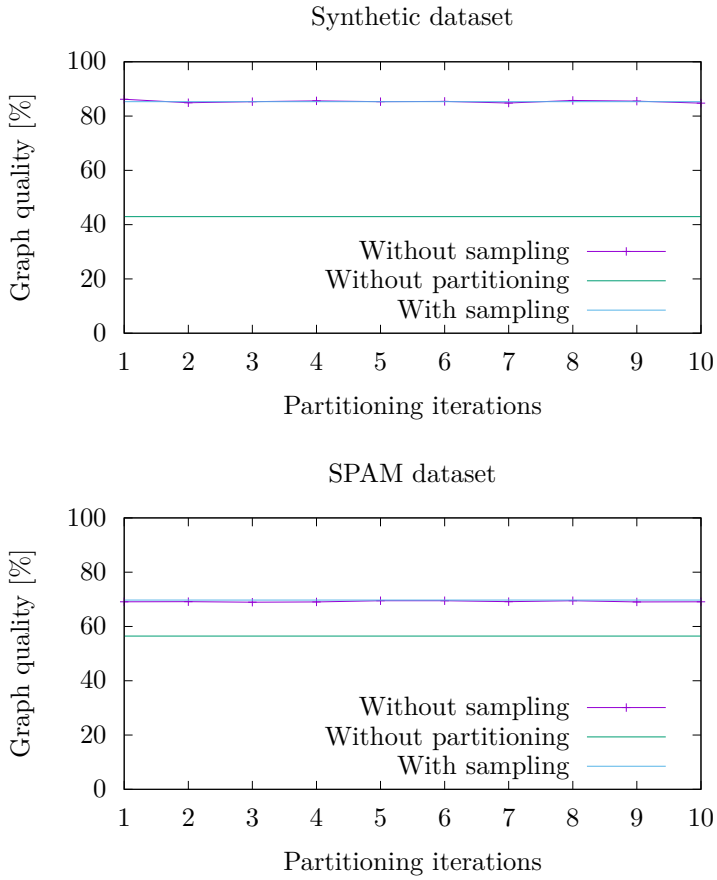

Figure 4.7: Influence of the partitioning

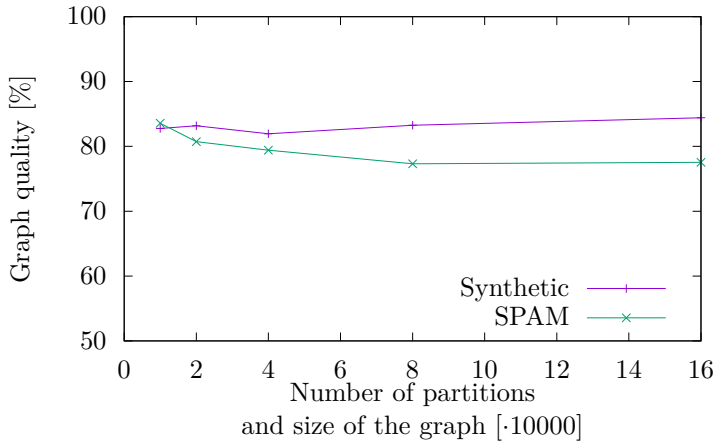

Figure 4.8: Scalability

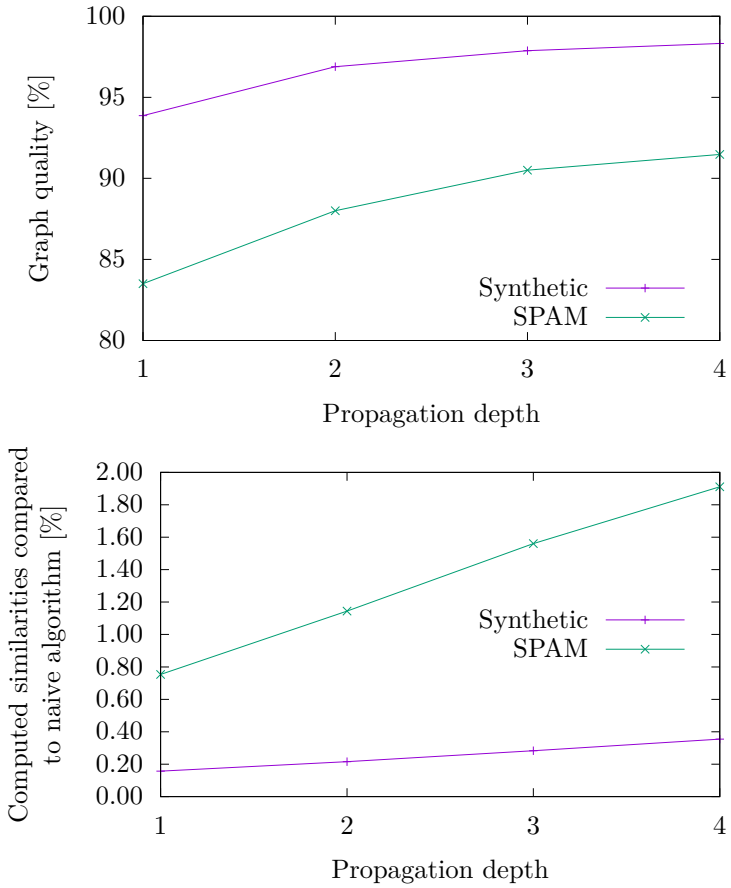

Figure 4.9: Removing nodes

4.8 Conclusions

In this chapter we proposed an algorithm that is able to update a distributed k-nn graph by quickly adding or removing nodes. We performed an experimental evaluation that shows the algorithm can be used with very large datasets and produces graphs that are highly similar to the graphs produced by a brute-force algorithm, while it requires the computation of far less similarities.

4.9 Contributions

— Thibault Debatty, Fabio Pulvirenti, Pietro Michiardi, and Wim Mees. Fast distributed k-nn graph update. In James Joshi, George Karypis, Ling Liu, Xiaohua Hu, Ronay Ak, Yinglong Xia, Weijia Xu, Aki-Hiro Sato, Sudarsan Rachuri, Lyle H. Ungar, Philip S. Yu, Rama Govindaraju, and Toyotaro Suzumura, editors, *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pages 3308–3317. IEEE, 2016

— Our Spark implementation of k-nn building and processing algorithms (including NN-Descent, NNCTPH, LSH, Online, NN-search etc.) is available on GitHub, Maven Central and Spark Packages: `https://github.com/tdebatty/spark-knn-graphs`

Chapter 5

Partitioning k-nn graphs

5.1 Introduction

In this chapter, we study more deeply the partitioning of k-nn graphs: if the graph is so large it cannot be processed on a single machine within a reasonable amount of time, it has to be distributed between multiple compute nodes and processed in parallel. However, the partitioning of the graph has a large impact on the performance of processing algorithms. This was one of our observations in the previous chapter and in [31].

Existing graph partitioning algorithms mainly focus on optimizing two metrics: reducing the number of cross-partition edges, and producing equally sized partitions. The impact of optimizing these metrics on different graph processing algorithms has already been studied in previous works like [75]. However, there is no guarantee that optimizing these metrics will indeed produce the optimal configuration of the graph [47]. In the previous chapter for example, we show that the optimal partitioning for searching the distributed k-nn graph actually corresponds to the definition of k-medoids clustering of the nodes. This shows that another approach for partitioning graphs, based on k-medoids clustering, is desirable.

In this chapter we study the usage of k-medoids clustering to partition large k-nn graphs with Bulk Synchronous Parallel (BSP) processing frameworks. We first focus on k-medoids clustering itself and propose two new optimized procedures. Then, in the second part of the chapter, we propose a method for using k-medoids clustering to partition large k-nn graphs.

The rest of this chapter is organized as follows. In section 5.2 we present existing algorithms to partition graphs and to compute k-medoids clustering. In section 5.3 we show that the current state of the art algorithm for computing k-medoids clustering has a very slow convergence time, so in section 5.4 we propose two optimizations that allow to speedup this processing, and we experimentally compare them. In section 5.5 we propose a partitioning procedure based on k-medoids clustering, which we experimentally compare to other classical partitioning algorithms in two different ways. First we use the usual measures of partitioning quality: cross-partition edges and partitions size. Then we study the actual impact of the partitioning by measuring the performance of a distributed search running on the partitioned graph. Finally, in section 5.6 we present our conclusions and propositions for future work.

Our contributions in this chapter are the following:

— we provide lower and upper bounds for the convergence time of CLARANS, the current state of the art algorithm for k-medoids clustering;

— we propose two new distributed algorithms for computing k-medoids clustering and perform an experimental evaluation;

— we propose a new method for partitioning k-nn graphs that relies on a k-medoids clustering of the nodes and we perform an experimental evaluation of this partitioning, which confirms that it allows to improve the results of subsequent algorithms applied to the k-nn graph, like distributed search.

5.2 Related work

5.2.1 Graph partitioning

A graph partitioning algorithm splits a large graph into several sub-graphs. From the mathematical point of view, there is a vast literature devoted to graph partitioning, see for instance [14] for a review. These consider different kinds of graphs, like trees or special types of trees, different type of constraints, and different optimality criterion.

From the distributed computing point of view, we are interested in distributed balanced k-way graph partitioning. These algorithms partition a large graph in a fixed number of partitions and in a way that allows other graph processing algorithms to achieve a better performance: if the graph is correctly partitioned graph analysis algorithms achieve a better performance (speed or accuracy) then if the graph is poorly partitioned. The efficiency of these partitioning algorithms is usually measured using balance and communication cost:

$$\text{balance} = \frac{\text{size of largest partition}}{\text{average partition size}}$$

$$\text{communication cost} = \text{number of cross-partition edges}$$

Two approaches exist to partition a graph: vertex and edge partitioning. In the vertex partitioning approach (also called edge-cut partitioning), the vertices (nodes) are assigned to different partitions. An edge is cut if its nodes belong to two different components. In the case of edge partitioning (or vertex-cut partitioning), edges are mapped to partitions and vertices are cut if their edges happen to be assigned to different components.

It has been shown that edge partitioning is a better approach to process graphs with a power-law degree distribution, which are common in real-world datasets [38, 4]. Hence most of the recent literature focuses on this area. There is a large variety of these algorithms available, like Random Vertex Cut (RVC), Canonical Random Vertex Cut (CRVC), Ja-Be-Ja-VC [82] and Hybrid-cut [21], and implementations exist for most distributed graph processing frameworks like Pregel, Giraph or GraphX [100].

In the case of a k-nn graph, the graph is stored using adjacency lists of fixed size k (usually called neighbor lists), hence only edge-cut partitioning algorithms are applicable. To the best of our knowledge, only a few of these algorithms currently exist that support distributed

execution: EdgePartition1D, PT-Scotch, parMETIS, Multi-level Label Propagation (MLP) and Ja-Be-Ja.

EdgePartition1D [76] is actually an edge (vertex-cut) partitioning algorithm. It assigns each edge to a partition using a hash value computed from the id of the source node of the edge. Hence all outgoing edges of a node are assigned to the same partition, which makes it suitable for partitioning k-nn graphs as well.

PT-Scotch [22], parMETIS [52] and MLP [101] belong to the same family of multi-level graph partitioning algorithms. They first reduce the size of the graph by collapsing vertices and edges. Then they partition the smaller graph, and finally they map back this partitioning to the original graph. These however rely on the Message Passing Interface (MPI) protocol and assume a fast communication between all nodes in the graph. For this chapter we target a different programming model, namely Bulk Synchronous Parallel (BSP) processing.

Finally, the Ja-Be-Ja [82] algorithm initially assigns each node to a random partition, then the algorithm iteratively swaps nodes between partitions to increase the number of neighbors they have in the same partition as themselves. A BSP implementation of Ja-Be-Ja has also been proposed in [19].

5.2.2 k-medoids clustering

k-medoids is a clustering algorithm that is very similar to k-means as they both attempt to minimize the distance between points labeled to be in a cluster and a point designated as the center of that cluster. However, there are two main differences between k-means and k-medoids: 1) in k-medoids centers are actual points from the dataset, called medoids and 2) k-medoids minimizes the sum of pairwise dissimilarities instead of the sum of squared distances. This gives k-medoids two advantages over k-means: 1) it is more robust to noise and outliers and 2) it can be used with datasets governed by a non-euclidean distance measure.

The four classical algorithms for performing k-medoids clustering are PAM, CLARA, Lloyd iteration and CLARANS.

Partitioning Around Medoids (PAM) is one of the first published algorithms for computing k-medoids clustering. It is an iterative algorithm. At each iteration the algorithm searches which of the current medoids should be replaced to obtain a better solution. The iterations are actually greedy, as the algorithm computes the gain obtained by swapping each medoid with each point in the dataset. Each iteration requires to compute the cost of kn swaps, where n is the size of the dataset. Some geometric reasoning is used to speed up the computation of the cost of each swap, but the algorithm remains quite slow and is typically used for small datasets only (less then 1000 objects) [45].

Clustering LARge Applications (CLARA) is an optimization of PAM that samples the dataset to perform the clustering. The algorithm draws multiple samples and gives the best clustering as the output, using the entire dataset to compute the cost of each solution.

Clustering Large Applications based on RANdomized Search (CLARANS) [78] is currently considered as the most efficient algorithm for performing k-medoids clustering. It considers the space of solutions as a graph, where each node is a set of k medoids and two nodes are

neighbors if they differ by one medoid. Hence, from a current solution, $k(n-k)$ neighbor solutions can be built by swapping one medoid with a random point from the dataset. Starting from a current solution, CLARANS randomly selects a random neigbor solution, compute its cost, and move to the neighbor solution if it has a lower cost. Otherwise it picks another random neighbor, and so on.

PAM and CLARANS are actually different implementations of the local search optimization strategy: they both try to improve the current solution by swapping a medoid with a point from the dataset. The main difference between them is that PAM works in a purely greedy way: for each medoid it tests all points in the dataset before performing the swap, while CLARANS performs the swap as soon as a point is found that will decrease the cost of the solution. This allows CLARANS to converge more quickly. As we will see, there is however room for improvement.

The Lloyd iteration algorithm [80] uses the same approach as the Lloyd algorithm used for k-means clustering: at each iteration, the algorithm searches the most central point in each cluster. If all clusters have the same size (n/k), finding the center of each cluster requires to compute at least $O(n^2/k^2)$ distances. Hence each iteration requires to compute $O(n^2/k)$ distances, which makes the algorithm quite slow as well.

In [89] and [23], the authors use a simulated annealing based technique to perform k-medoids clustering. In these papers the authors apply the simulated annealing acceptance criteria after the CLARANS algorithm computes the cost of the neighbor solution, to escape a possible local minimum. According to our experience, local minimums are relatively rare when using CLARANS, hence using simulated annealing this way only slows down the convergence of the algorithm. In section 5.4 we propose another approach, where we apply the decision criteria before we compute the cost of the neighbor solution. This allows to avoid computing the cost of the current solution, which is a costly operation.

5.3 Convergence time of CLARANS

We define an iteration of CLARANS as the process required to improve the current solution. Hence it consists in finding a new solution with a smaller cost.

A trial is defined as selecting a candidate solution and computing its cost. It requires to compute kn similarities with the standard algorithm. This can be reduced to n if appropriate caching is used.

Consider the simple case of a uniform dataset $\mathbf{D}$ in $\mathbb{Z}$ with $k = 2$ and unitary distance between points, like depicted on Figure 5.1. The space of solutions is the set of (x, y) from $\mathbf{D} \times \mathbf{D}$ where $x \neq y$. It forms a grid in $\mathbb{Z}^2$.

The set of the costs of every candidate solution forms a grid. It is symmetric as the solutions (x, y) and (y, x) are equivalent, and it has only two equivalent maxima: (x_o, y_o) and (y_o, x_o).

Consider the grid formed by the solution space. At any iteration, $k(n-k) = O(nk)$ candidate solutions can be built from the current solution.

We now compute an upper bound for the number of trials required by CLARANS at any iteration. The progression of CLARANS can be measured using d, the Manhattan distance

between the current solution and the optimal solution in the solution space, as shown in Figure 5.1. More formally, d is defined as follows: let $o_0 \dots o_i \dots o_k$ be the optimal solution and $m_0 \dots m_i \dots m_k$ be the current solution (the set of medoids at any iteration of the algorithm). Then:

$$d = \sum_{i=1}^{k} \text{distance}(m_i, o_i) \tag{5.1}$$

By the definition of CLARANS, at each iteration this distance will decrease by at least 1 (the distance between points in the dataset).

Consider any iteration of the algorithm where $d > k$. The worst configuration for the convergence of the algorithm is when the candidate solution is located in a diagonal with respect to the optimum solution, like in Figure 5.1. This means that, from the current solution, all medoids have to be swapped before we reach the optimal solution. In this configuration, in each direction (for each medoid) there are $2d/k - 1$ points that lead to an improved solution, like presented in Figure 5.1. As there are k directions (medoids), the number of neighbor solutions that have a smaller cost (in red) is $(2d/k - 1)k$, and the total number of neighbor solutions is $O(nk)$. Hence this iteration will require $O(nk/(2d - k))$ trials on average to progress.

At any iteration, and for any value of d (also if $d \leq k$), the best configuration is achieved when the candidate and the optimal solution differ by only one medoid. In this configuration, there are $2d - 1$ solutions that will reduce the cost compared to the current solution. The cost for completing the iteration is hence $o(nk/2d)$ on average.

Remark that in both cases, **the average number of trials required by CLARANS to complete the iteration, and hence to progress towards the optimum solution, is proportional to** nk.

As an example, suppose we have a solution where $d = 1$ (right next to the optimal solution), only one solution will improve the total similarity compared to the current solution (the optimal solution). Hence CLARANS will require up to nk trials to find this one, which will require to compute n^2k^2 similarities.

This shows that **CLARANS is actually slow to converge because the neighbor solution is selected purely at random**.

5.4 Efficient k-medoids clustering

The slow convergence rate of CLARANS is due to the fact it tries a lot of candidate solutions before finding one that is actually better, and computing the cost of a solution is a heavy operation that requires computing nk similarities. However, a lot of candidate solutions can be quickly and cheaply discarded. Indeed, recall that CLARANS builds a new candidate solution by swapping a medoid from the current solution with a random point from the dataset. From Equation 5.1, we can infer that:

$$\forall i \ : \ \text{distance}(m_i, o_i) \leq d \tag{5.2}$$

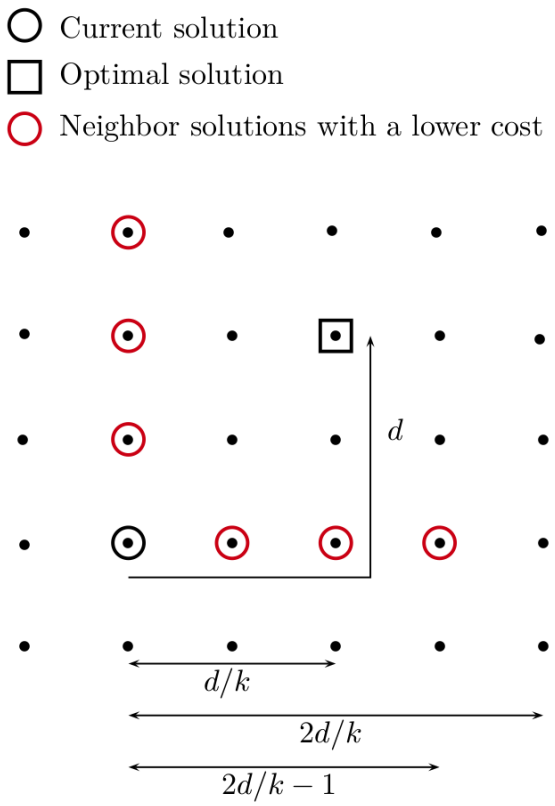

Figure 5.1: Convergence of CLARANS

Namely, the distance between a current medoid and the optimal medoid is at most d. This allows to quickly discard candidate solutions that would not improve the current solution.

For any configuration, the number of neighbor solutions that obey this condition is dk. Hence the average number of trials required to progress is reduced to $O(dk/(2d-k)$ for the worst configuration and $o(dk/2d) = o(k/2)$ for the best configuration. Remark that this time **the number of trials is not proportional to** n. This shows that we can spare a lot of trials and computations if we only consider neighbor solutions located at a distance $\leq d$ from the current solution.

Unfortunately, there is no easy way to compute d as we do not know in advance where the optimum solution is located. Hence we propose two procedures to perform efficient k-medoids clustering that **discard neighbor solutions that are probably located too far from the current solution**, without actually relying on the value of d.

The first one is inspired by simulated annealing. In the experimental evaluation we show that, with correct parameters, it converges faster than the current state-of-the-art. However, like all simulated annealing algorithms, parameter tuning is very difficult. We also present another approach that relies on a simple to compute heuristic that does not require any configuration parameter. According to our experiments, it performs better than the state-of-the-art in all our evaluations.

5.4.1 Simulated annealing neighbor generation

Our approach to compute k-medoids clustering is based on local search but, instead of selecting a random neighbor solution like CLARANS, we use an algorithm inspired by simulated annealing to select the neighbor. In the graph of solutions, we try to discard solutions that do not fulfill Equation 5.2.

To build a neighbor solution we select a random medoid from the current solution and a random point from the dataset. The distance between these two points is the measure of the energy that has to be minimized when the algorithm converges. Hence we use an initial temperature T_0 that will decrease at each iteration according to a cooling factor γ, and the probability to accept this random point is computed using the formula of Metropolis and Kirkpatrick [53]: $P = e^{-distance/T}$.

Algorithm 5 Simulated annealing neighbor solution generator

Input: current solution S
$m \leftarrow$ a random medoid from S
$p \leftarrow$ a random point from the dataset
$d \leftarrow distance(m, p)$
$T \leftarrow T_0 \cdot \gamma^{iteration}$
swap m and p with probability $e^{-d/T}$

At the first iterations the temperature is high, hence we accept neighbors located at a large distance from the current solution. As the algorithm progresses towards the optimal solution, the temperature decreases such that we only generate neighbors located in the vicinity of the current solution.

After the neighbor solution is generated as depicted in algorithm 5, we continue with the classical local search procedure: the algorithm computes the cost of the neighbor solution, which is accepted if it has a lower cost. The advantage compared to classical CLARANS is that our neighbor solution generators allows to discard neighbor solutions that are located too far from the current solution in the solution space, before we compute their cost, thereby sparing unnecessary computation.

Unlike the algorithms presented in [89] and [23], we apply simulated annealing before and not after the cost of the neighbor solution is computed. Indeed, we try to reduce the number of computed similarities and to speed up the algorithm by reducing the search space, while the two aforementioned papers use simulated annealing to escape a possible local minimum, after computing the cost of the solution.

However, like all methods based on simulated annealing, our method is efficient only if correct values for T_0 and γ are used, and there are no known approaches for estimating these values based on specific characteristics of the dataset. Hence we also propose another neighbor solution generator, based on a simple heuristic that requires no input parameter.

5.4.2 Heuristic based neighbor generation

To explain the idea behind the heuristic based neighbor generator, we consider the situation illustrated in Figure 5.1. We consider all the neighbors of the current solution along the

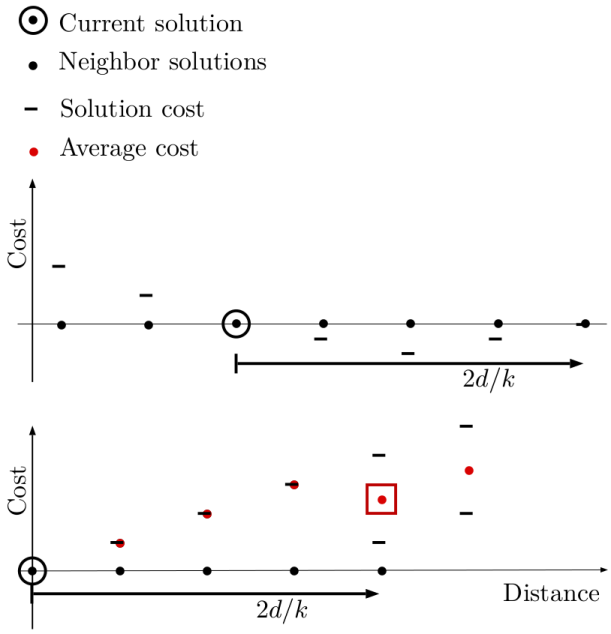

Figure 5.2: Heuristic based neighbor generator

horizontal axis (all the solutions that can be built by swapping one medoid with a random point from the dataset). On the top of Figure 5.2, we plot the cost of each neighbor solution. The neighbors with a negative cost are the ones that allow the algorithm to progress, but they have not been discovered yet (otherwise the algorithm would use one of them as the current solution). They are all situated at a distance $\leq 2d/k$ from the current solution.

On the bottom of Figure 5.2 we plot the cost of the discovered neighbors as a function of the distance with the current solution. In red we plot the average cost of solutions situated at the same distance. This plot has local minimum (identified with a red square) that also corresponds to the maximum distance for which solutions exist that will reduce the cost of the current solution ($2d/k$).

This is the approach used in the heuristic based neighbor generator: we keep track of the distance and cost of every evaluated candidate solution and search a local minimum in the resulting curve. The distance corresponding to this local minimum is used as the maximum distance for trying a candidate solution.

5.4.3 Experimental evaluation

We now perform an experimental evaluation of our algorithms. All tests are executed on a Spark cluster composed of one master and 16 workers. Each test is executed twenty times and we show the average result. We use different datasets to run the tests:

— **synthetic-3-12** is a dataset composed of 100.000 points in R^3 distributed around 12 centers according to a gaussian law;

— **synthetic-10-12** is a dataset of 100.000 points in R^{10} distributed around 12 centers;

— **synthetic-3-100** is a dataset of 100.000 points in R^3 distributed around 100 centers;

Table 5.1: Optimal parameters identified for simulated annealing neighbor generator

Dataset	γ	T_0 [·1000]
synthetic-3-12	0.990	60
synthetic-10-12	0.993	15
synthetic-3-100	0.995	10
commercials	0.980	64
spam	0.985	64

— **commercials** is a dataset from the UCI Machine Learning Repository[61] that contains features extracted from 129.685 TV sequences;

— **spam** is a dataset composed of the subject of 100.000 spam emails. The similarity between spams is computed using Jaro-Winkler string similarity, which does not belong to a metric space.

A lot of formulas exist to measure the quality of a clustering, like Davies–Bouldin index, Dunn index or Silhouette coefficient. However, by definition the goal of k-medoids clustering is to maximize the total similarity between each point in the dataset and its associated cluster. As a consequence, depending on the geometric configuration of the points in the dataset, a clustering with higher total similarity may result in a lower Silhouette coefficient for example. For this chapter, we want to know if the solution found by the different algorithms actually corresponds to the definition of k-medoids clustering. Hence we use the total similarity to measure the quality of clustering.

5.4.3.1 Parameter study

We first study the influence of the two parameters required by the simulated annealing approach: initial temperature T_0 and cooling rate γ. For this test we let the algorithm compute a fixed number of similarities (1E9) while we vary the cooling rate and the initial temperature. For each experiment we record the sum of the similarity between each point and corresponding medoid (hence higher is better). The results are presented in Table 5.1 and Figure 5.3. For the sake of visibility we only plot a subset of the results and we shift the curves vertically to fit on a single Figure. This clearly shows that the optimal value depends on the dataset: if the initial temperature is too low or the cooling rate too high, the algorithm wastes time searching for a neighbor solution located too close to the current solution, which prevents the algorithm from progressing towards the optimum solution. At the opposite, if the initial temperature is too high or the cooling rate too low, the algorithm accepts any neighbor solution, which is the same behavior as CLARANS.

5.4.3.2 Comparison with the state-of-the-art

We now experimentally compare the Spark implementation of the three k-medoids clustering algorithms: CLARANS, our Simulated Annealing approach, and our heuristic approach. To compare the convergence time of the algorithms we first need to compute the exact solution of the k-medoids clustering problem. This requires to evaluate all possible solutions:

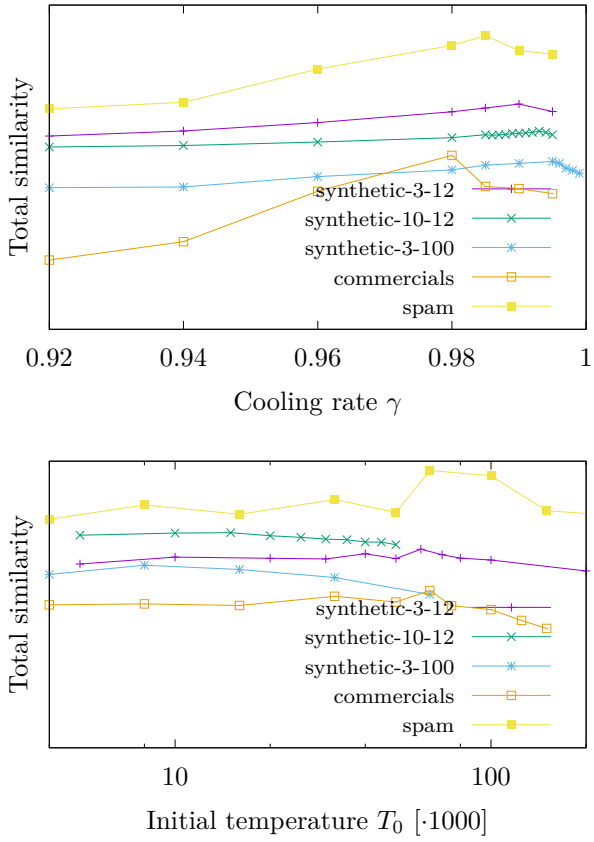

Figure 5.3: Influence of initial temperature and cooling rate

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

This is only feasible within a reasonable amount of time for toy problems. For our datasets, this would require to compute roughly 10^{43} solutions ($n = 100000$ and $k = 10$). Instead we show here how fast the algorithms converge by measuring the quality of the clustering after a fixed number of similarities have been computed. For the simulated annealing approach we use the optimal parameters identified above. Figures 5.4 and 5.5 show the quality of the clustering when we vary the number of computed similarities for the five datasets.

We can clearly see that the heuristic based approach performs at least as well as the classical CLARANS. In some cases the algorithm requires to compute twice less similarities than CLARANS to achieve the same quality of clustering. The simulated annealing performs even better, but only if the configuration parameters (T_0 and γ) are precisely tuned for the dataset to process and for the number of similarities that can be computed during analysis (1E9 in our tests).

5.5 k-medoids based graph partitioning

In this section, we propose to use k-medoids clustering to partition a large k-nn graph between multiple compute nodes. Then we compare our approach to existing algorithms in two different ways. First we use the classical metrics used to measure the performance of partitioning algorithms: balance and communication cost. Then we compare the performance of the distributed search algorithm we presented in Section 4.5 using graphs partitioned with the different algorithms.

5.5.1 Balanced k-medoids based graph partitioning

We propose here to use a k-medoids based approach to partition a large k-nn graph. Therefore we first compute the k-medoids clustering of the nodes using the optimized method presented above, and then we assign each node and its corresponding adjacency list to the partition corresponding to the most similar medoid.

To achieve a balanced distribution of points between the k partitions (not to confuse with the k edges per node of the k-nn graph), we use a modified heuristic to assign each point to a partition. Let $p_0 \dots p_i \dots p_n$ be the set of nodes in the graph and $m_0 \dots m_j \dots m_k$ the set of medoids at any iteration of the algorithm. Each point p_i is assigned to the partition $P(p_i)$ corresponding to the most similar medoid weighted by a penalty function $w(m_j, t)$:

$$P(p_i) = \arg\max_{m_j}(\text{similarity}(p_i, m_j) \cdot w(m_j, t))$$

This penalty function is based on the capacity and current size of the partition, in order to penalize large clusters.

In a distributed shared nothing environment like Spark, the different workers do not have access to the total size of each partition at time t. Each compute node can only rely on the

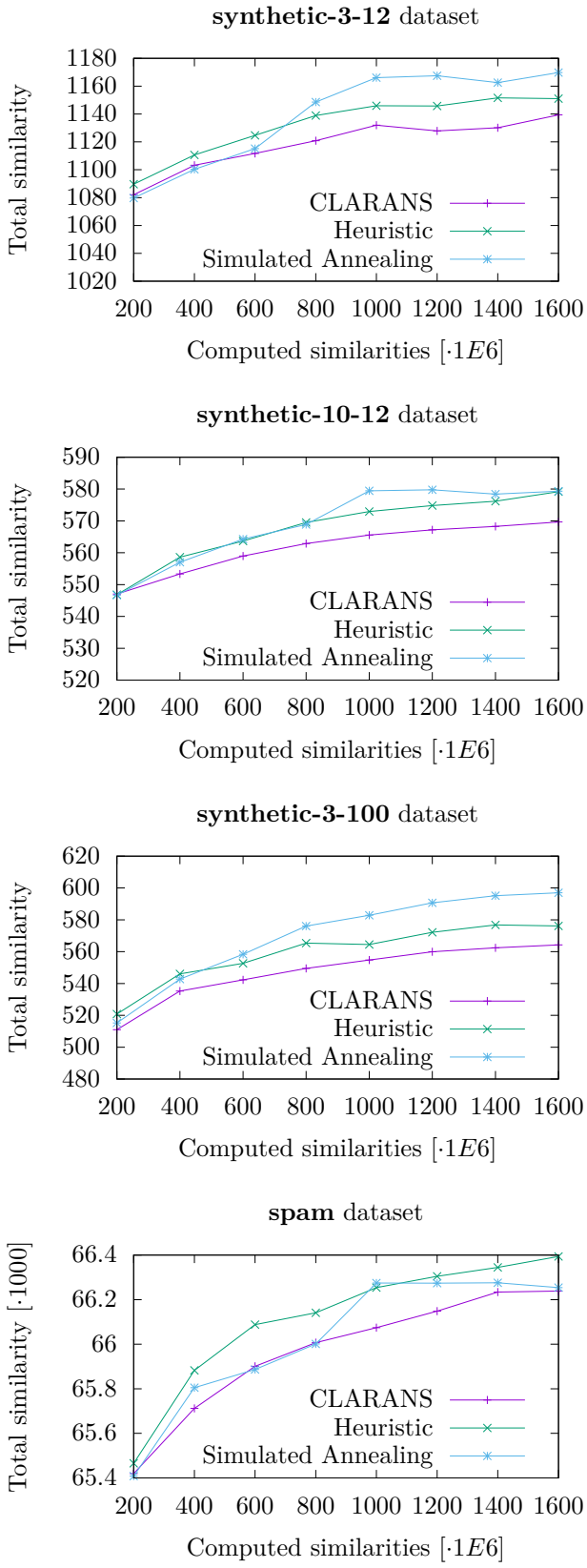

Figure 5.4: Quality of clustering with different datasets

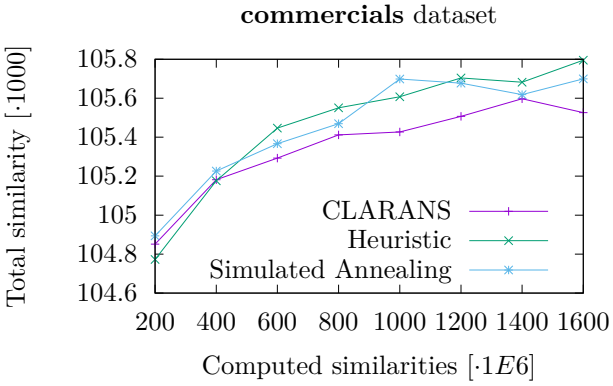

Figure 5.5: Quality of clustering with different datasets

local number of points already assigned to a partition $P_L(m_j, t)$.

Hence, to execute the algorithm in parallel we first randomly distribute the nodes between the c workers. At each iteration, this randomized dataset is used to assign each node using a weight function that relies solely on the local size of clusters:

$$w_L(m_j, t) = 1 - \frac{|P_L(m_j, t)|}{capacity_L} \tag{5.3}$$

where $capacity_L$ is the contribution of each compute node to total size of the final partition:

$$capacity_L = \frac{n \cdot imbalance}{k \cdot c}$$

Depending on the application, a small size difference can be tolerated between partitions, hence the capacity of partitions is usually computed using an imbalance factor ($imbalance \geqslant 1$). At the opposite, a perfectly balanced partitioning can be achieved using $imbalance = 1$.

If the dataset is large enough (with respect to c) and randomly distributed, we can assume that the resulting error (relative number of points that are not assigned to the correct cluster) can be kept arbitrarily low.

5.5.2 Experimental evaluation

We now experimentally compare the Spark implementation of our k-medoids based graph partitioning with Ja-Be-Ja and Edge1D [1]. For the sake of fairness, the Ja-Be-Ja algorithm implements the optimizations proposed in [19] and [75]. We first build the 10-nn graph corresponding to each dataset. The resulting characteristics of the graphs are shown in Table 5.2. Then we partition the graph between our 16 workers, letting each partitioning algorithm run for a fixed amount of time, and finally we compute the number of cross-partition edges (communication cost) and the imbalance of the resulting partitioning. For the k-medoids partitioning, we use a target imbalance parameter of 1.2. As Edge1D is a

1. `https://github.com/tdebatty/spark-knn-graphs`

Table 5.2: Characteristics of the graphs used for experimental evaluation

Dataset	Vertices (nodes)	Edges
synthetic datasets	100.000	1.000.000
commercials	129.685	1.296.850
spam	100.000	1.000.000

single pass algorithm, we simply let it run until it completes, which requires approximatively 3 seconds on our cluster. The results are presented in Figures 5.6 and 5.7.

As we could expect, the quality of the partitioning performed by Edge1D is consistent with a random distribution of the nodes. We can see that the k-medoids approach manages to find a good partitioning from the very first iterations, and then keeps slowly improving the partitioning. At the opposite, Ja-Be-Ja starts with a random partitioning and then very slowly improves the solution, swapping only a few hundreds of nodes at each iteration. On the figures we also show the average quality of partitioning achieved by Ja-Be-Ja after running for 1 hour. We can see that even after 1h of computation, Ja-Be-Ja is not even close to the quality of the k-medoids based partitioning.

5.5.3 Graph partitioning for efficient distributed search

We now turn back to our initial problem: how to partition a large k-nn graph such that we can efficiently run other processing algorithms, such as distributed search? We showed previously that these kind of algorithms require the graph to be partitioned according to a k-medoids clustering.

We perform here an experimental evaluation to show that a k-medoids based approach indeed allows the search algorithm to deliver better results. We use the different algorithms to partition the graph, and this time we measure the accuracy of the distributed search algorithm. We first split each dataset between a training and a query set. We build the graph from the training set and we partition it with the different algorithms, then we use the query set to perform distributed search with the algorithm presented in Section 4.5.

This search algorithm assumes that each partition contains a subgraph of the distributed graph. Each subgraph is searched independently using a hill-climb approach. When the search reaches a boundary of the subgraph, it simply restarts from a random node. Finally the best solution from all partitions is returned. This algorithm has the advantages that 1) it requires only one iteration and 2) it requires almost no communication.

In Figures 5.8 and 5.9 we show the proportion of correct search results for the different partitioning algorithms.

The results clearly show that the k-medoids approach largely outperforms the other partitioning algorithms. We can also notice that the results are less good for the **spam** and the **synthetic-10-12** datasets than for the others. For the **spam** dataset, this is due to the non-euclidean character of the dataset. For the **synthetic-10-12** dataset, this is due to the higher dimensionality of the dataset. As we also observed in [31], these two configurations make searching more challenging.

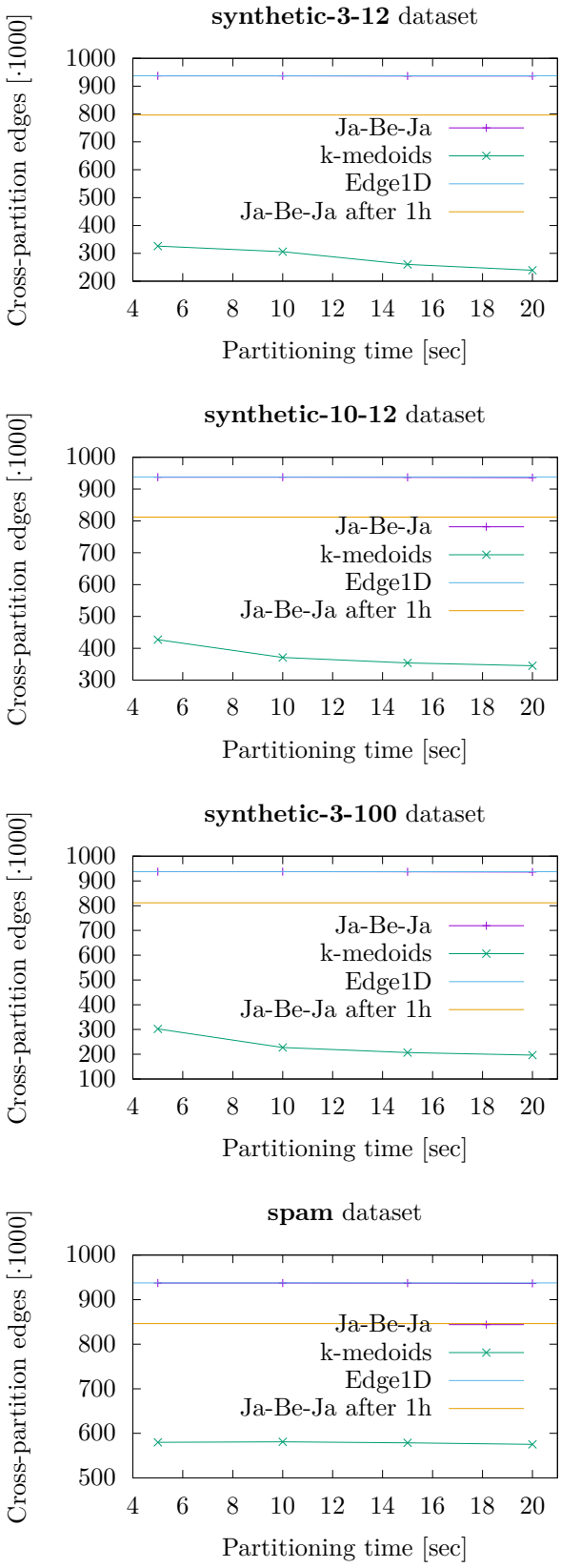

Figure 5.6: Quality of partitioning with different datasets (lower is better)

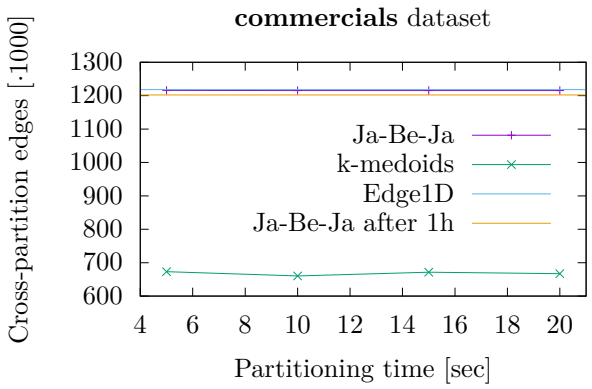

Figure 5.7: Quality of partitioning with different datasets (lower is better)

5.6 Conclusions and future work

In this chapter we studied the usage of k-medoids clustering to partition large k-nn graphs. We proposed two new optimized procedures for performing k-medoids clustering. Then we proposed a method relying on k-medoids clustering to partition large k-nn graphs. Our experimental evaluation showed that 1) our clustering procedures outperform the current state-of-the-art and 2) k-nn clustering is an excellent approach for partitioning large k-nn graphs as it is at the same time faster and more efficient than current partitioning algorithms.

As a future work, we plan to evaluate the quality of the k-medoids partitioning when other algorithms are executed on the partitioned graph, like Connected Components.

5.7 Contributions

— Thibault Debatty, Pietro Michiardi and Wim Mees. Efficient k-medoids based graph partitioning. Submitted to the *24th International Conference on Parallel and Distributed Systems (ICPADS2018)*. Notification expected on 01 September 2018.

— Our Spark implementation of k-medoids based graph partitioning is available at `https://github.com/tdebatty/spark-knn-graphs`

— Our Spark implementation of k-medoids clustering algorithms (CLARANS, Lloyd iteration, Simulated Annealing and Heuristic) is available on GitHub and Maven Central: `https://github.com/tdebatty/spark-kmedoids`

— To support our experimental evaluation, we also implemented a Java library to easily parse various datasets (DBLP bibliograph database, Reuters-21578 news dataset, ENRON email dataset, synthetic datasets generated according to a gaussian law, Wikipedia web pages dataset etc.). This library is also available on GitHub and Maven Central: `https://github.com/tdebatty/java-datasets`

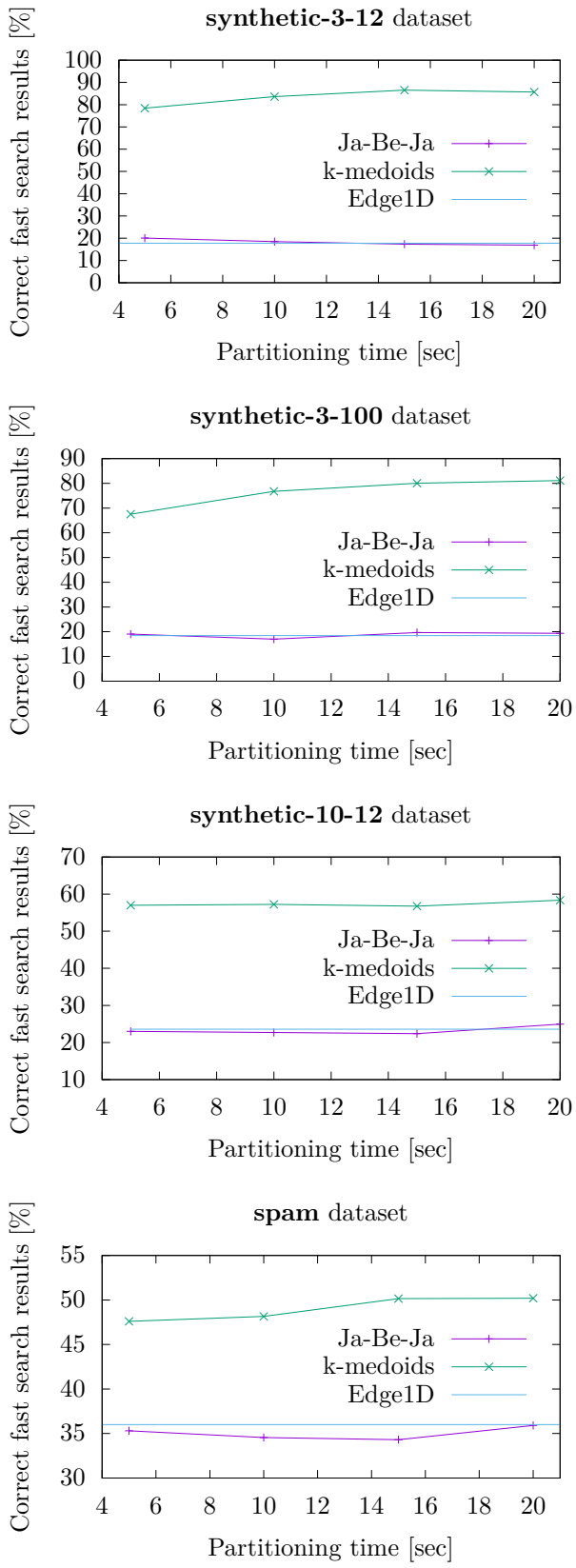

Figure 5.8: Effect of graph partitioning on the results of the fast distributed search algorithm

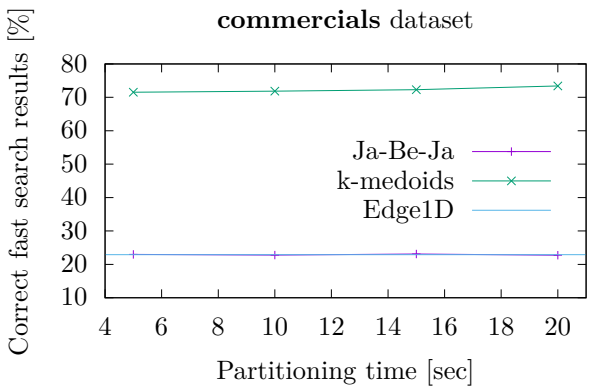

Figure 5.9: Effect of graph partitioning on the results of the fast distributed search algorithm

Part II

Applications

Chapter 6

Scalable k-nn based text clustering

6.1 Introduction

Data clustering is a fundamental analysis task that consists in finding groups of related data items. The definition of similarity used to perform the clustering is usually determined by the application domain at hand. In the same way, measuring the quality of the clustering obtained by applying one of the numerous clustering techniques that exist is often dependent on the application domain. A clustering considered as good for one application may not suit the requirements of another one. This often calls for manual inspection by domain experts.

In this Chapter, we focus on a particular data clustering task, which involves grouping text data items. The application domain of our work stems from the objective of identifying SPAM campaigns: we focus on data collected by Symantec Research Labs to perform root-cause analysis of large scale SPAM email campaigns originating from bot networks.

This a very adversarial context as spammers usually try manipulate text to avoid SPAM emails being identified as originating from the same campaign, which makes data clustering even more challenging. As a consequence, the similarity metrics used for clustering must cope with text mangling. They must be able to detect similar strings, even if typos, character swapping and other techniques to avoid detection have been applied. According to our experience, Jaro-Winkler [49, 104] text similarity is suitable for this purpose, although it has the drawback that is not a metric distance, as it does not obey to triangle inequality.

In addition, data clustering is challenging due to the large volume of data that is collected nowadays: this calls for the design of scalable algorithms, capable of ingesting millions of data points and cluster them in meaningful ways.

Current approaches fall short in addressing the above challenges.

For example, the widespread K-means clustering algorithm requires text data to be transformed into d-dimensional vectors to operate correctly. However, such transformations generally imply high-dimensional vectors, which render distance functions problematic, an

issue known as the "curse of dimensionality". Other common approaches for text clustering based on frequency analysis of shingles also suffer from high-dimensionality problems. Moreover, for applications like the ones we consider in this work where strings are short sentences (like the subject of an email), frequency analysis is problematic.

Also, the common techniques discussed above are not amenable to non-metric spaces, which is a requirement for the application domain we study. As a consequence, clustering performs poorly: data items that should be considered similar and fall in the same clusters are instead assigned to different groups.

Finally, it is generally very challenging to design scalable clustering algorithms. For example, K-means – albeit easy to parallelize – requires a large number of similarity computations and suffers from a long runtime. Frequency based methods are also difficult to scale as it is generally cumbersome to parallelize frequent itemset mining algorithms.

This is why in this chapter we present a clustering approach that addresses the above concerns: *i)* it produces high quality clusters that are easy to interpret, *ii)* it accommodates any kind of similarity functions, even non-metric ones, and *iii)* it is scalable. The gist of our clustering algorithm consists in building an *approximate* k-nn graph of the input text data, and compute its connected components, which identify data clusters.

In particular, we aim at understanding the trade-off that exists between accuracy, scalability and, ultimately, clustering quality. To do so, we proceed with a thorough experimental evaluation of our clustering method with real, large-scale datasets, using our implementation – which is publicly available – for the Apache Spark computing framework.

The contributions of this chapter are the following:

— We design and implement a *scalable* algorithm for text clustering, which works in an "adversarial" application scenario, and that produces high quality, and interpretable clusters.

— We perform a detailed experimental analysis, where we show the impact of the parameters that govern the degree of approximation of our method. Our results indicate that even rough approximations are sufficient to obtain high quality clusters, which is also beneficial for the algorithm runtime.

— We use real-life datasets and evaluate the overall clustering quality of our approach both using traditional metrics, and with the help of domain experts through manual investigation, highlighting the interpretability of clustering results.

The remainder of the chapter is organized as follows. In section 6.2 we survey related works, and discuss the differences with our approach. In section 6.3 we present our approach in detail. Then, in sections 6.4 and 6.5 we present the experimental setup and our main results, respectively. We conclude the chapter in section 6.6, where we also discuss our future work.

6.2 Related Work

The problem of clustering data has been widely studied in the computer science literature at large, with nuances ranging from graph theoretic and data mining principles [39, 6] to

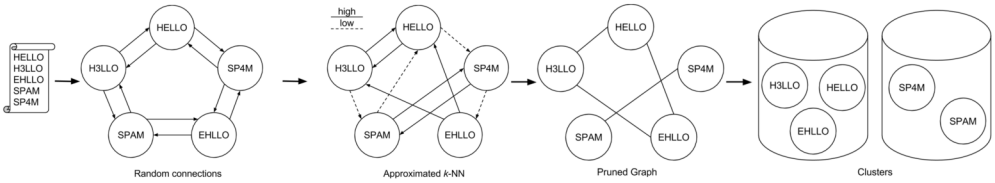

Figure 6.1: Illustration of our approach

experimental approaches [93, 107]. Since it is outside the scope of this paper to give justice to all work that has been done in the domain, here we focus on approaches that are closest to our work.

One of the most popular clustering algorithms is the Lloyd iteration algorithm [64], which performs K-means clustering. This is a simple approach that operates on d-dimensional vectors. It can also be used to perform text clustering, but this requires a *transformation phase* to encode sentences and words into vectors [51, 97]. For example, Mikolov et al. [73] present an efficient implementation of the continuous bag-of-words and skip-gram architectures for computing vector representations of words. In our work, we use such approach as a baseline to which we compare our clustering algorithm. In the experimental evaluation we show that it suffers from its inability to accept *non-metric* distance measures, which are essential to detect similarity between mangled sentences, and from its poor scalability.

Alternative approaches search for frequent terms in the dataset to identify clusters [11, 10, 72]: essentially, the idea is to find subsets of frequent term sets, which are a proxy for clusters, and map data items containing elements of such subsets to the same cluster. Such approaches suffer from their inability to scale well (in some cases, clusters may overlap, and a disambiguation phase based on greedy heuristics is required, which is difficult to scale), and do not take into account text similarity that is resilient to mangling.

Other approaches aim to optimize the computation of pairwise similarity between text items using matrix computations [77]. In this category, Lin et al. [62] presented an optimized algorithm to retrieve clustering of text data from a similarity matrix using cosine similarity. Once again, such approaches do not accommodate non-metric similarity measures and are difficult to scale, although recent work [16] has shown the benefits of approximate matrix operations which scale better than exact, all-pair similarity computations.

An approach that targets goals that are similar to ours is Triage [96], which addresses the same application domain we target in this paper. However, the focus of Triage is on multi-feature data items, and not on scalability: as such, the authors mainly address problems related to information fusion, by defining a method to merge several different distance metrics operating on text, categorical and numerical values. Recently, a parallel version of Triage has been proposed [92], which partially addresses scalability issues. However, the approach still computes all-pair similarity among representative, prototype items, with an $O\left(n^2\right)$ complexity that still makes handling very large data sets difficult.

6.3 k-nn based clustering

We now present our approach for scalable text clustering and we provide a parallel implementation. We also underline the important role played by *approximation* in our algorithm. Besides the algorithm design itself, using approximation techniques constitutes an important contribution to our analysis of the trade-off that exists between clustering quality and the scalability of our method.

The problem of text clustering we consider is particularly challenging due to the application scenario we study. We face an adversarial setting in which text data is generated such that finding similar items is cumbersome: SPAM campaigns use text mangling, spelling errors and generally variations on some baseline text which makes SPAM items belonging to the same campaign appear different one from each other. As a consequence, we need to use a similarity metric between items that can overcome, or at least mitigate, the problem.

6.3.1 Similarity measure

There exist numerous algorithms to measure the similarity between items. In particular, for text data, the Hamming distance and Levenshtein distance have been extensively used.

In this work, for the reasons illustrated above, we choose the Jaro-Winkler [49, 104] similarity measure which, simply stated, counts the common characters between two strings even if they are misplaced, misspelled, and mangled by a "short" distance. Jaro-Winkler is not however a metric distance as it does not obey to the triangle inequality.

Given the choice of the similarity metric we use in this work, the wide spectrum of techniques to find clusters of similar text items reduces to few methods. This is the main driving factor that steers the algorithmic design choices we make in this work.

6.3.2 Illustrative example

Before delving into the technical details of our method, we first provide an overview of our algorithm, and proceed with an illustrative example. Our text clustering approach works in two phases. In the first phase, it builds an approximate k-nn graph of text dataset. The first phase concludes with a pruning stage, which strives at eliminating spurious links between items with low similarity. In the second phase, we use a parallel approach to identify connected components in the k-nn graph, which are a proxy for clusters of similar items.

Figure 6.1 illustrates the process, where we consider 5 SPAM email subjects: note the swap of letters and typos, which are typical of SPAM emails. Each email subject corresponds to a node in the intermediate graph structure our approach builds.

Initially, each node connects to $k = 2$ randomly chosen nodes. First, our algorithm iteratively builds an approximate 2-nn graph. On the Figure, edges having low similarity are dashed. The number of iterations used to buid the graph constitutes one parameter of our approach. At this point, the pruning phase eliminates edges between nodes that have a low similarity, using a threshold value that constitutes the second parameter of our algorithm. Finally, using the pruned 2-nn graph, our method finds its connected components, which we use as

a proxy of the clusters in the original dataset.

We now describe each step of our clustering algorithm in detail.

6.3.3 Phase 1: k-nn Graph Construction

In the first phase of our method, we build a k-nn graph. In this work, we limit our attention to text features: for example, we extract the subject of a SPAM email as the only representative feature of the item. Considering additional or heterogeneous features is outside the scope of this work, and we defer it to an extension of our approach.

The naïve approach to build a k-nn graph consists in finding all-pairs similarity among all items of a dataset, then select the k most similar items to each item. Clearly, this "brute-force" approach is not scalable, as it requires $O(n^2)$ similarity computations, where n is the number of items in the dataset. Note that the "brute-force" algorithm produces *exact* k-nn graphs, which we use in this work as a baseline to determine the approximation quality of our method.

Instead, we use a Spark implementation of NN-Descent, similar to the Hadoop MapReduce implementation that we already presented in Chapter 3. Remember that this algorithm uses an iterative approach to progressively refine the k-nn graph and eventually build an approximation of the exact k-nn graph.

In this work we are particularly interested in the role of the number of iterations of the algorithm, which determines its approximation quality. We claim that even rough approximations of the k-nn graph are sufficient for the ultimate goal of our clustering method. Intuitively, the existence of a path between similar items on the k-nn graph is sufficient for the last phase of the clustering algorithm we propose.

6.3.4 k-nn Graph Pruning

The iterative procedure to build an approximate k-nn graph may induce neighboring relations between text items that have a low pairwise similarity. Indeed, the algorithm necessarily outputs the k most similar neighbors for each item: any skew in the distribution of the pairwise similarities may produce a k-nn graph in which some node are only "loosely" similar. We thus use a *pruning phase*, with parameter $\theta \in [0, 1]$ to determine a cut-off similarity value, below which edges between any pair of nodes are eliminated. The pruning phase inspects each node of the k-nn graph, and prunes such edges.

Choosing an appropriate threshold θ determines the final output of our clustering method: as $\theta \to 1$ clustering is *strict*, which leads to a large number of small clusters of essentially identical items; as $\theta \to 0$ clustering is *loose*, leading toward the degenerate case of a single, giant cluster.

6.3.5 Phase 2: Connected Components

The second and last phase of our approach uses the pruned k-nn graph, and outputs its connected components, that we use as a proxy for identifying clusters of similar items. Recall

that the problem of finding the connected components of a graph amounts to searching for sub-graphs in which any two vertexes are connected to each other by paths.

Finding connected components in large-scale graphs using scalable algorithms is a well studied and understood problem, the abundant literature on the subject bears witness to this [87, 54, 67].

In this work we use a parallel implementation of the Cracker algorithm [67], wherein each node is tagged with the smallest node identifier of its component called seed node identifier. The key idea of Cracker is to reduce the graph size in each step of the computation, by employing a technique inspired by the "contraction algorithm" to compute minimum cuts of a graph [50]. At termination, all nodes tagged with the same seed identifier are grouped in the same component.

Table 6.1: Symantec Dataset: characteristics

Feature	**Unique**	**Value 1**		**Value 2**		**Value 3**		**Value 4**		**Value 5**	
bot	11669	Lethic	20.41%	Unclassified	18.63%	Bagle	11.24%	Cutwail	9.36%	Grum	9.20%
city	33905		16.22%	Seoul	3.61%	Kiev	2.16%	Hanoi	1.76%	Moscow	1.70%
country	224	Russian Federation	10.46%	India	8.53%	Brazil	5.83%	Korea, Republic of	5.73%	Ukraine	4.62%
day	393	10/31/2010	0.31%	10/30/2010	0.30%	10/23/2010	0.30%	2/19/2011	0.30%	11/3/2010	0.29%
fromDomain	241117	domain555065.com	11.10%	domain359761.com	2.22%	domain572911.com	0.90%	domain425436.com	0.86%	domain384117.net	0.56%
host	32915		42.09%	airtelbroadband.in	1.71%	ukrtel.net	1.67%	localhost	1.62%	hinet.net	1.40%
ip	2227174	anonymous_IP_1	0.04%	anonymous_IP_2	0.03%	anonymous_IP_3	0.03%	anonymous_IP_4	0.02%	anonymous_IP_5	0.02%
rcptDomain	8641	domain555065.com	81.66%	domain806676.fr	3.27%	domain946987.org	2.18%	domain240360.br	1.93%	domain801669.com	1.64%

6.4 Experimental Setup

This section provides details about our experimental setup, including datasets used, evaluation metrics, parameters and system environment.

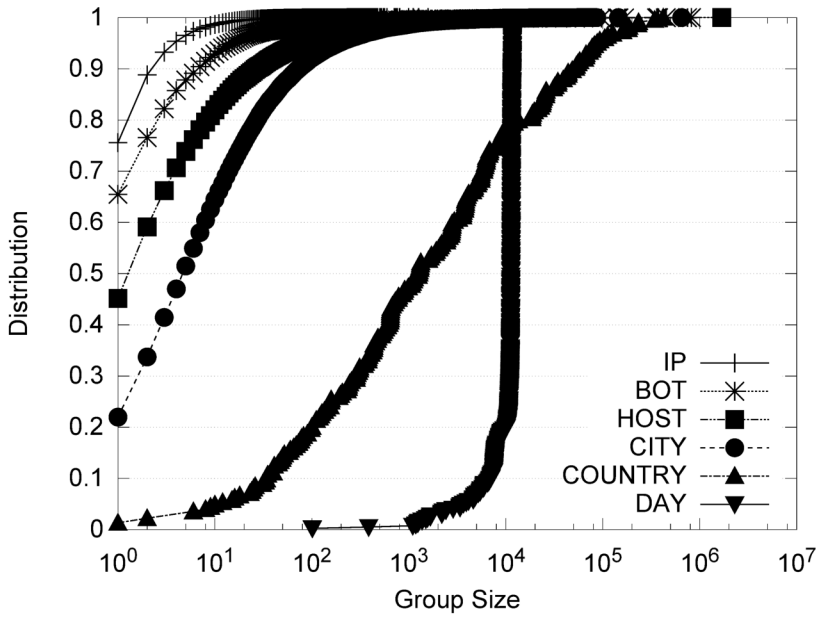

Figure 6.2: Symantec dataset: feature distribution

Table 6.2: Symantec Dataset: average similarity

Feature	**Similarity**
bot	0.6
city	0.49
country	0.49
day	0.5
fromDomain	0.54
host	0.5
ip	0.51
rcptDomain	0.5

Experimental platform. All the experiments have been conducted on a cluster consisting of 17 nodes running Ubuntu Linux (1 master and 16 slaves), each equipped with 12 GB of RAM, a 4-core CPU and a 1 Gbit interconnect.

We implemented our approach and the baseline method we use for our comparative analysis using Apache Spark [8] and our source code is publicly available.

Evaluation Metrics. We now discuss the metrics we use to analyze the parameter space of our approach, and for its global validation in terms of clustering quality. Also, we manually investigate the clusters we obtain, using domain knowledge to evaluate the goodness of clustering.

We study the role of the parameters of our approach –namely the number of iterations used

to build the k-nn graph and the pruning threshold θ– using the following metrics:

— **Number of clusters**: measures the number of clusters identified by the clustering algorithm. If not otherwise stated, we only consider clusters to be "useful" if they have more than 1,000 elements;

— **Largest cluster size**: measures the size of the largest cluster identified by the algorithm.

We compute clustering quality using well-known metrics [35, 63]:

— **Compactness** measures how closely related the items in a cluster are. We obtain the compactness by computing the average pairwise similarity among items in each cluster. Higher values are preferred.

— **Separation** measures how well clusters are separate from each other. Separation is obtained by computing the average similarity between items in different clusters. Lower values are preferred.

— **Silhouette** [88] constitutes an aggregate metric, that takes into account the inter- and intra-cluster pairwise similarity between items. Higher values are preferred.

— **Recall** relates two data clustering obtained by different methods. Using clustering C as a reference, we compute the recall of clustering D by computing the fraction of items that belong to the same cluster in both C and D. In particular, we use as a reference the exact clustering we obtain with the "brute force" approach to compute the k-nn graph. Higher values of recall are preferred.

It is important to notice that computing the above metrics is computationally as hard as computing the clustering we intend to evaluate. For this reason, we resort to uniform sampling: instead of computing the all-to-all pairwise similarity between items, we pick items uniformly at random, with a sampling rate of 1%. [1]

The datasets. The main dataset we use in our evaluation consists of a subset of SPAM emails collected by Symantec Research Labs, between 2010-10-01 and 2012-01-02, which is composed by $3,886,371$ email samples. Each item of the dataset is formatted according to JSON and contains the common features of an email, such as: subject, sending date, geographical information, the bot-net used for the SPAM campaign as labeled by Symantec systems, and many more. For instance, a subject of an email in the dataset is "19.12.2011 Rolex For You -85%" and the sending day is "2011-12-19".

In this work, we are interested in identifying clusters of SPAM emails using subjects alone, as they constitute a compact description of the email.

Next, we provide an overview of the dataset we use, to gain a better understanding of its characteristics; we also proceed with a naïve approach to clustering emails, by grouping them according to some of their fields.

Table 6.1 illustrates such a preliminary analysis: the "Unique" column identifies the number of distinct values for each feature we use, whereas additional columns in the table indicate the top individual values for each feature. For instance, "grouping by" the feature "bot" indicates that there are 11,669 unique bot-nets in the dataset, with "Lethic" taking roughly

1. We increase the sampling rate up to 10% for small clusters.

20% of the emails, followed by 18% of "Unclassified" bot-nets and 11% from the "Bagle" bot-net. The dataset contains emails sent from 224 different countries, as shown when "grouping by" the feature "country".

Figure 6.2 describes how the size of each group is distributed in the dataset. For example, 90% of the groups having the same bot-net value have a size lower than 10 emails, indicating skewness when considering the "bot" feature. Instead, groups having the same "day" feature are more uniformly distributed: only 10% of the days have less than 10^4 emails, while the remaining 90% of the days have similar group size around the value 10^4.

Finally, we verify if grouping emails according to the features of Table 6.1 results in email having similar subjects: in other words, we are interested in understanding if using similar subjects to cluster emails would boil down to simply grouping them by some other features. The right side of Figure 6.2 pinpoints at a negative answer: essentially, grouping by any of such features results in email subjects being very loosely similar, which is not sufficient to consider such groups a useful proxy for email clusters.

To cross-validate our approach on a different dataset, we also use a data obtained using the Twitter API, and consisting of $1,530,623$ tweets in JSON format.

6.5 Results

In this section we present our result, and we organize it as follows. First, we analyze the parameter space of our algorithm, and discuss the impact of such parameters on the metrics we defined above. Then, we focus on clustering quality, and compare the performance of our approach to that of the *baseline* algorithm we discuss in section 6.2. Finally, we study the clustering scalability.

6.5.1 Analysis of the parameter space

First, we summarize the parameters underlying our algorithm and discuss about their role. Our approach has 3 main parameters: k, the number of neighbors to construct the k-nn graph; the number of *iterations* of the first phase of the algorithm; and θ, the pruning threshold.

The experimental results we show in this section are obtained with a sampled version of the Symantec dataset, and account for 800,000 data items. A sampled dataset allows us to execute the "brute force" method to compute the k-nn graph.

In what follows, we let the number of *iterations* and θ to be free parameters, and instead select a few representative values for k. We chose k to be small, *i.e.*, we allow a few neighbors per node in the k-nn graph. First, each *iteration* of the k-nn graph construction step has $O(k^2 n)$ complexity [37]: lower values of k lead to better runtime performance. Note also that the approximation quality of the k-nn graph, that we measure using *recall*, increases with k: larger values imply a better approximation. Since we are interested in understanding the approximation / clustering quality trade-off of our algorithm, we deliberately chose lower values of k. Finally, consider that large values of k produce dense k-nn graphs, which remain so even after pruning. For all these reasons, in what follows, we consider $k \in \{5, 10, 15\}$.

Impact of the number of iterations. Next, we show that the number of iterations of the k-nn graph building phase, which is proportional to the accuracy of the k-nn graph, does not need to be large for the algorithm to settle to a "steady state". Even rough approximations of the k-nn graph, obtained with a small number of *iterations*, are sufficient for the algorithm to stabilize.

We show this in Figure 6.3 representing the *Silhouette* of the clustering and the number of unique clusters (with more than 1,000 items), as a function of the number of *iterations*. In particular, the top Figure 6.3 indicates that clustering quality stabilizes already after roughly 5 *iterations*, for a range of choices of k and θ. Bottom Figure 6.3 focuses on $k = 5$, which produces very rough k-nn graph approximations: in this case, roughly 10 *iterations* are sufficient for the number of unique clusters to stabilize. We note that the range displayed in the figure is suitable for a manual investigation from domain experts to further dig into understanding the clustered data.

Impact of the pruning threshold θ. We now discuss how the pruning mechanism modifies the k-nn graph, and what is the impact on clustering. As discussed in section 6.3, as θ tends to 0, pruning is less effective, and the k-nn graph tends to have a single giant component. Instead, when θ tends to one, only very similar neighbors survive pruning, and the k-nn graph is fractioned in a large number of small clusters.

Figure 6.4 shows the fraction of nodes for which a given number of edges are removed after pruning, as a function of θ. For values of $\theta < 0.8$, pruning is less effective, as the number of pruned edges is small. Instead, for $\theta > 0.9$, a large fraction of nodes remain with one or fewer edges after the pruning phase. This translates in sizes of the largest clusters to approach the entire dataset, for $\theta = 0.5$ already, or to be extremely small, for $\theta = 1$, as shown in Figure 6.4.

Overall impact of approximation. We now study the impact of the k-nn graph approximation on clustering quality, by analyzing the deviation of our approach from the results obtained from an *exact* k-nn graph computed using the "brute force" approach. Our results indicate that approximate k-nn graphs obtained with a low k and few *iterations* are sufficient to obtain data clustering that is practically indistinguishable from that obtained by an onerous $O(n^2)$ k-nn graph construction phase.

Figure 6.5 shows how clustering recall varies as a function of the k-nn *iterations*. As a reminder, clustering recall accounts for the fraction of items belonging to the same clusters, when comparing the results obtained by the approximate algorithm to those of the "brute force". As shown in the Figure, $k = 10$ and 5 *iterations* are sufficient to obtain a clustering which is *essentially identical* to that obtained with *exact* k-nn graph. Even a very low value of $k = 5$ settles to a 0.8 recall, after roughly 10 *iterations*.

We conclude our analysis of the parameter space by evaluating the cluster quality from the application perspective, using domain knowledge. In general, a SPAM campaign is likely to have originated from one or few bot-nets, and to last for only a few days. For this reason, we consider a clustering result to be useful if clusters contain a substantial number of emails (1,000 in our case) that have been sent only from two or less bot-nets, within a time-frame of less than a week. Figure 6.6 compares the number of "good" clusters obtained with our approach to those obtained with *exact* k-nn graphs, for a range of k and θ parameters, and 10 *iterations*. Clearly, the approximation introduced by our approach does not substantially

ruin clustering quality, while – as we show in the following sections – it is very beneficial for algorithmic runtime.

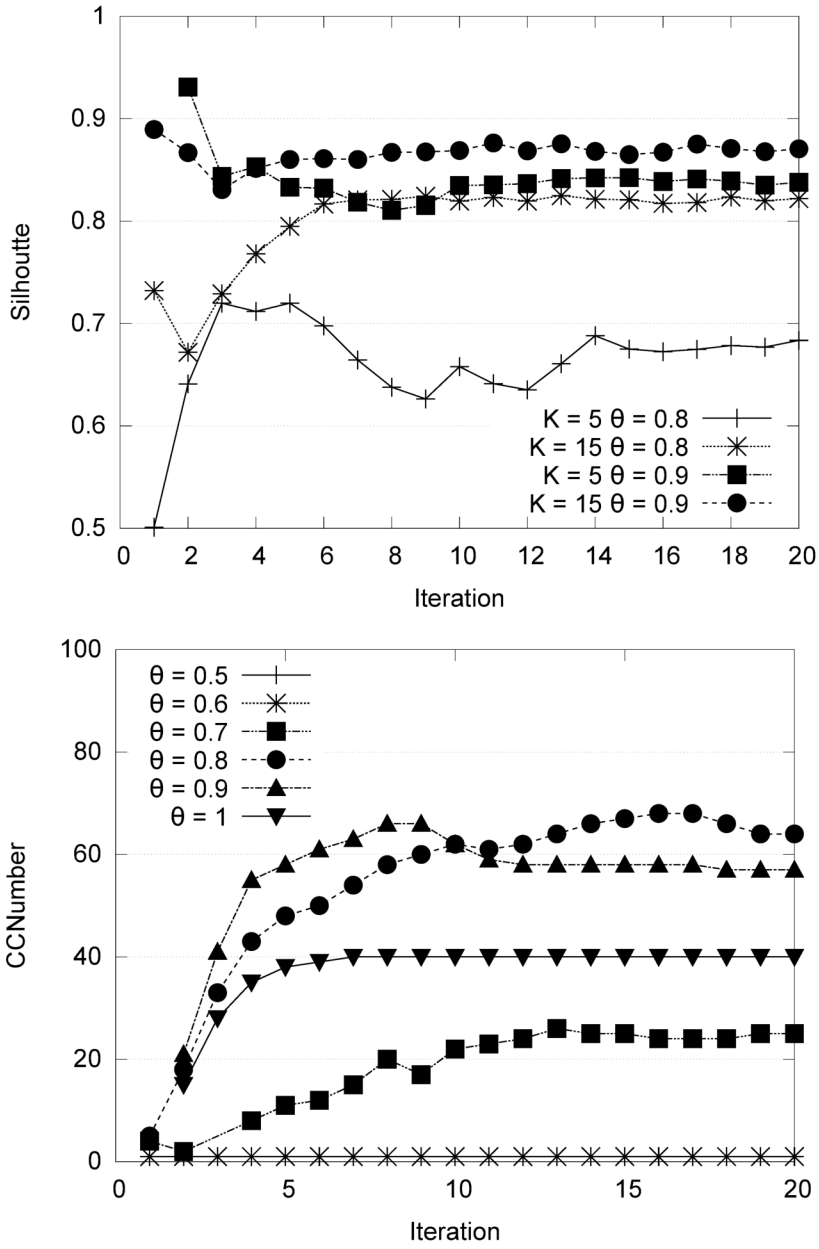

Figure 6.3: Impact of number of *iterations* for the k-nn graph construction phase of our algorithm. Clustering quality "stabilizes" after roughly 10 iterations, indicating that approximate k-nn graphs are good enough.

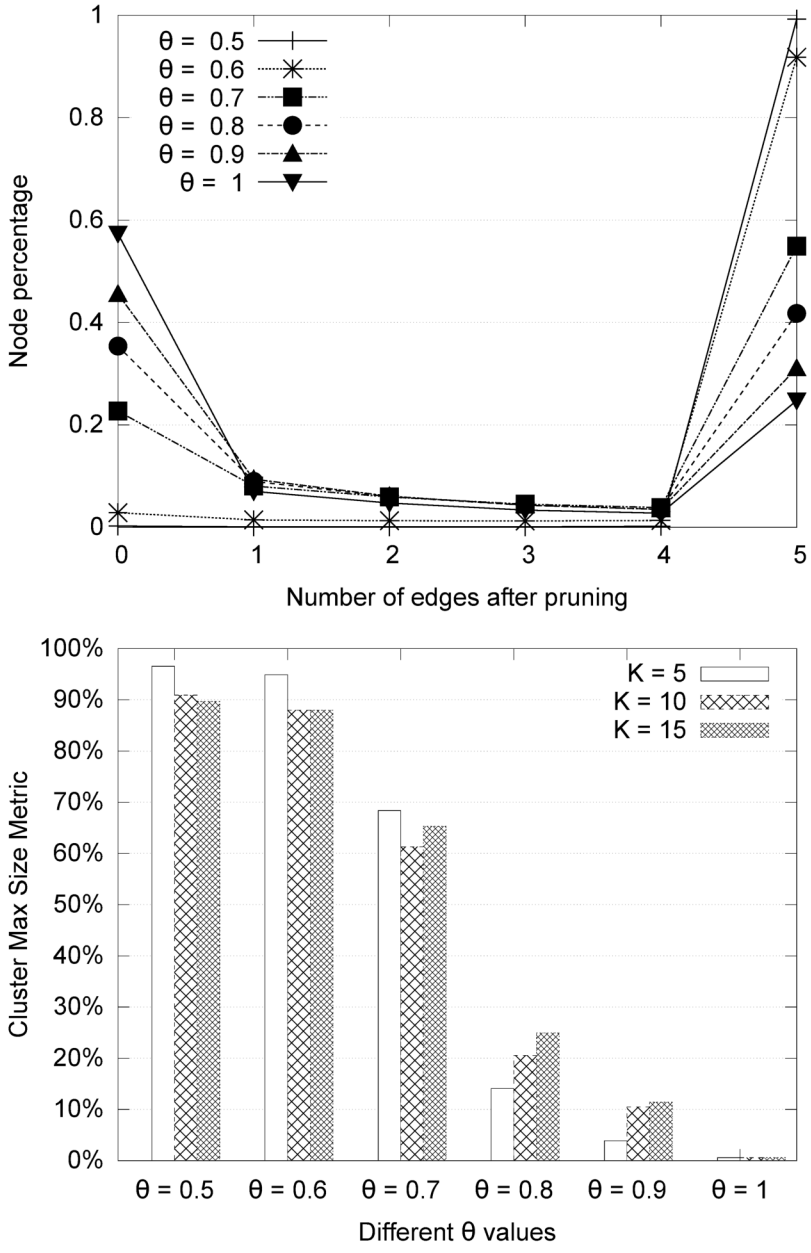

Figure 6.4: Impact of the pruning phase threshold θ. Extreme values of θ either produce a single, giant cluster, or too many small clusters. In our experiments, and with our dataset, $\theta \in [0.8, 0.9]$ produces useful clustering results.

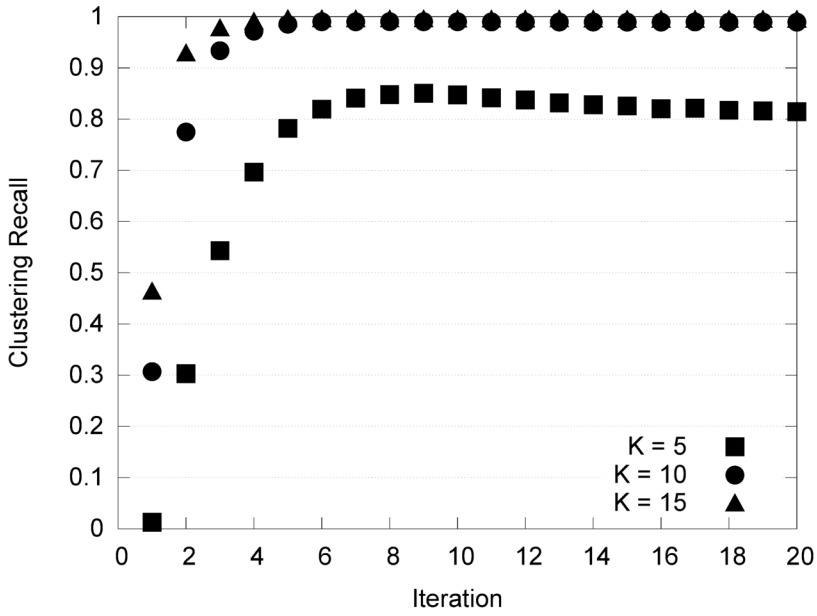

Figure 6.5: Iterations vs. Clustering recall.

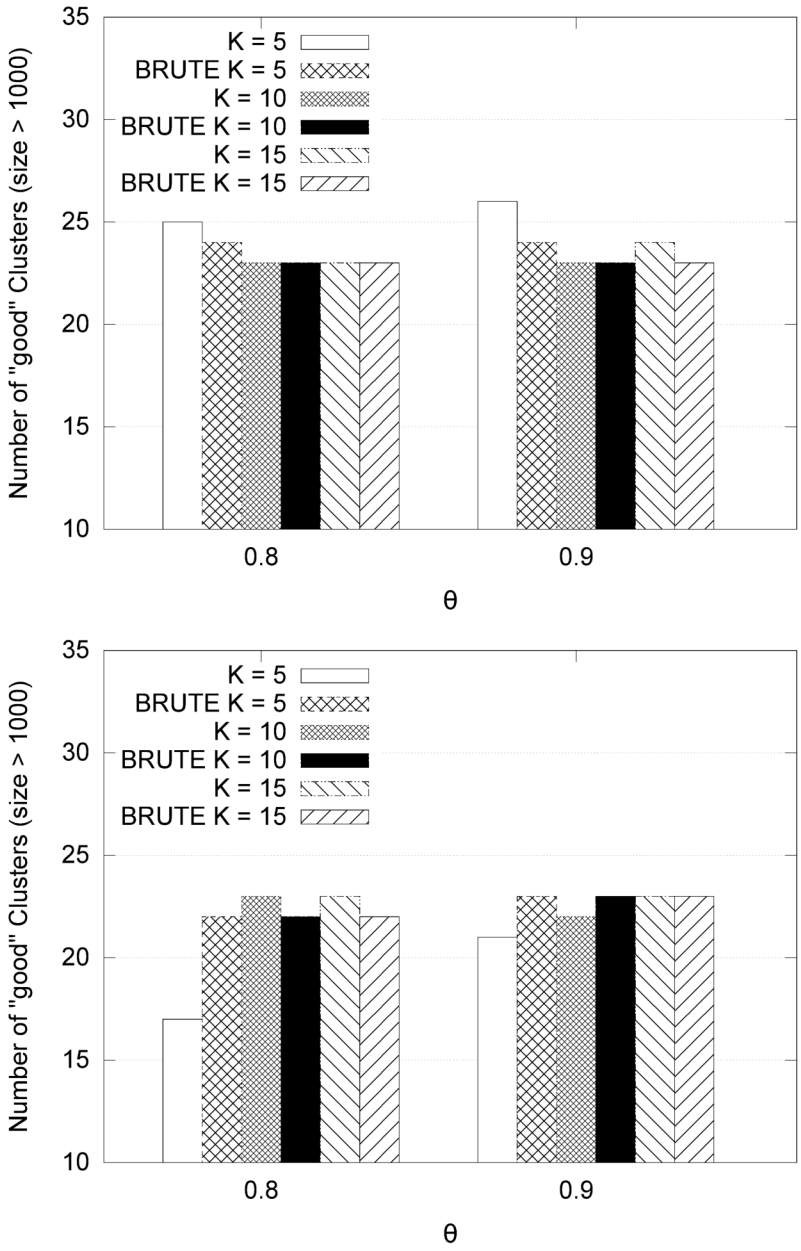

Figure 6.6: Overall impact of approximation quality for two or less bot network identifiers (top) and 7 days or less (bottom). Few *iterations* and small values of k are sufficient to produce output clusters that are practically indistinguishable from those obtained with a "brute-force", *exact* k-nn graph.

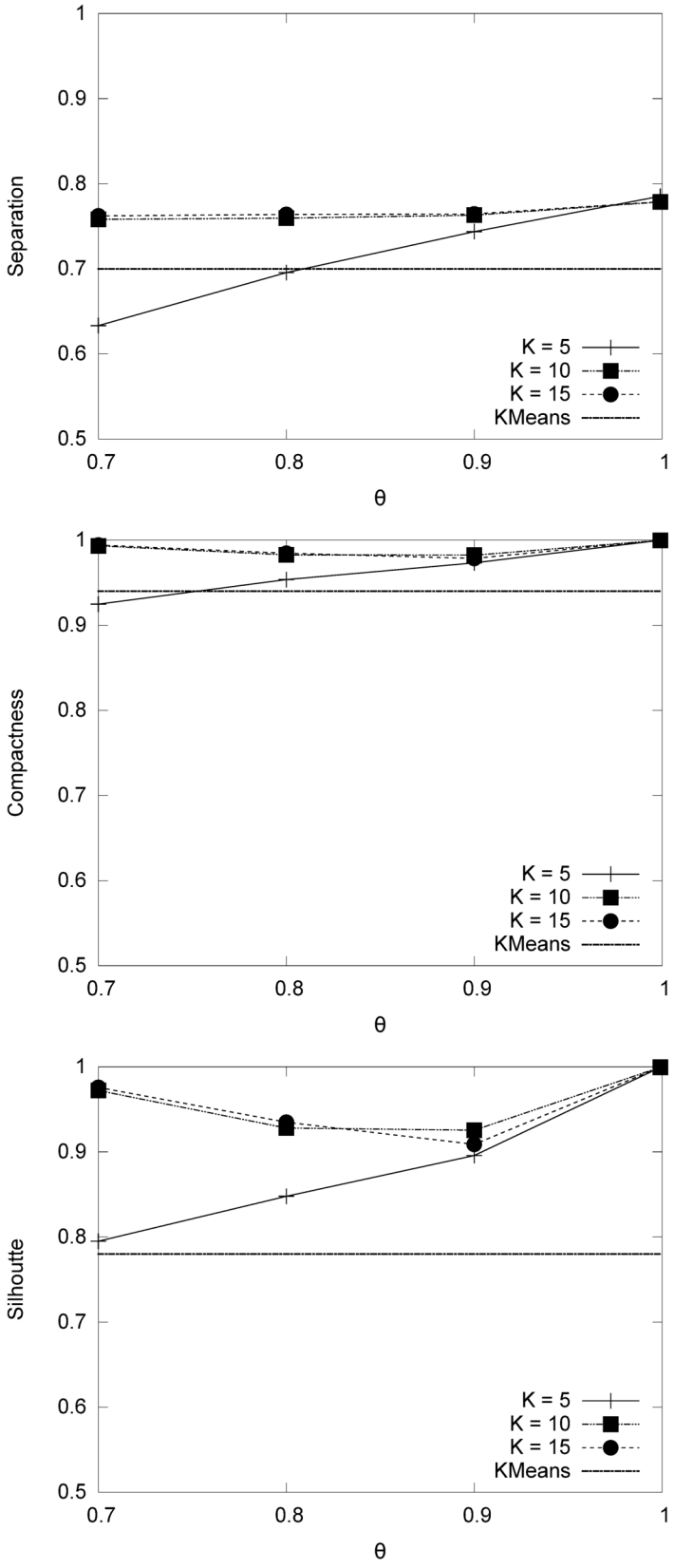

Figure 6.7: Symantec Dataset: Clustering quality in terms of inter and intra cluster similarity, and clustering Silhouette. The algorithm proposed in this work outperforms the baseline method, and is only marginally affected by approximation quality.

Table 6.3: Symantec Dataset: Manual Investigation

cluster size	bot	days	subjects sample
7255	Grum, Unclassified	2011/12/14-2011/12/20	"17.12.2011 Rolex For You -73%" "15.12.2011 Rolex For You -89%" "19.12.2011 Rolex For You -85%"
4512	Rustock, Unclassified	2010/12/04-2010/12/06	"jadevnn, Alena (status-online) invites you for chat." "Hi zmes40, Alena (status-online) invites you for chat." "keumd„Alena (status-online) invites you for chat."
4412	Rustock, Unclassified	2011/01/28-2011/02/01	"Re: User kilmernn" "Re: User anguinet" "Re: User hudnalli"
4116	Rustock, Unclassified	2011/03/12-2011/03/14	"tdwilkey, you have a new PRIVATE MESSAGE", "dbeltondd, you have a new PRIVATE MESSAGE" "bn, you have a new PRIVATE MESSAGE"
2992	Grum, Unclassified	2011/08/19-2011/08/23	"cseeberd@Amega.com VIAGRA ? 84% consensus!" "Maia@Amega.com VIAGRA ? 50% consensus!" "zelmo38dd@Amega.com VIAGRA ? 16% consensus!"

6.5.2 Analysis of the clustering quality

We now move to a global evaluation of the algorithm we present in this chapter, and compare clustering quality to the **baseline** algorithm described in section 6.2. In particular, we use the efficient K-means implementation available in Spark's MLLib package [1], and the `word2vec` package [3], as illustrated in [2]. It is important to note that the parameter K, in K-means is substantially different from the parameter k of the k-nn graph algorithm: it indicates the number of clusters the K-means algorithm is set to produce. If not otherwise specified, we set $K = 1000$ such that the baseline algorithm output 1,000 clusters. This configuration yields the best result in term of Silhouette metric.

For the sake of completeness, we first focus on the *full* Symantec dataset of more than 3 million emails, and corroborate our results with the Twitter dataset. In addition, in what follows and if not otherwise specified, we set the operating parameters of our algorithm as follows: $k = 10$, and 10 *iterations*, which are the parameters that offer a good trade-off between clustering quality, approximation quality, and algorithm runtime.

Symantec dataset. Figure 6.7 shows the three main metrics we use to judge clustering quality, namely *separation*, *compactness* and *Silhouette*, as a function of θ, and for various values of k. Such metrics are computed both for our approach, and for the *baseline* algorithm based on K-means.

The separation metric evaluates how "far apart" the clusters output by the algorithms are: lower values of separation indicate that the inter-cluster distance is large, which is a desirable property to distinguish clusters well. As shown in Figure 6.7, both our method and the baseline algorithm achieve good separation, with a slight advantage for the baseline method, that produces clusters that are more pairwise dissimilar. [2]

On the other hand, the compactness metric indicates how similar are the items within a cluster: larger values of compactness are desirable, because they are indicative of the absence of outliers that could "pollute" the quality of individual clusters with unrelated items. Figure 6.7 indicates that our approach is superior to the baseline method with respect to this metric, the latter producing clusters with emails that are unrelated to the majority of other items in a cluster.

Figure 6.7 illustrates that the clustering Silhouette obtained by our approach is superior to the baseline algorithm, and this holds for all parameter choices. This is confirmed also in Figures 6.8 and 6.8, which show the number of "good" clusters (with at least 1,000 items) as determined by domain knowledge metrics. Essentially, these figures report the number of clusters amenable to manual inspection of the results, as a function of features such as the number of bot-nets and the time-frame of a SPAM campaign. For example, in Figure 6.8, domain experts can extract valuable information when the number of SPAM bots in a cluster is small, in the 1-2 range: in this case, our approach is superior to the baseline algorithm, which performs slightly better for the less interesting cases of 3-4 and 5-6 bots. Similarly, Figure 6.8 shows that the number of "good" clusters identified by our approach is always better than that of the baseline algorithm, and this is especially true for the 1-7 range, indicating cluster with emails spanning a 1 week time-frame. In summary, we find that the

2. An "artifact" due to the distance metric used in K-means, which separates text items even if they differ because of mangling.

algorithm we present in this work performs well, especially when selecting an appropriate operating point, with $\theta \in [0.8, 0.9]$.

Finally, we proceed with a manual inspection of the clusters we obtain with our approach, to further illustrate the "goodness" of the clustering we achieve, with $k = 10$, 5 *iterations* and $\theta = 0.9$.

Table 6.3 illustrates a few email samples in clusters where both the number of bot-nets is less or equal to 2, and all emails are all sent within one week time-frame. For instance, we obtain a cluster of 7255 emails sent from the Grum bot-net, between 2011/12/14 and 2011/12/20: the subjects of the email are related to a SPAM campaign involving a Rolex discount. Note that subjects are all related, albeit not identical.

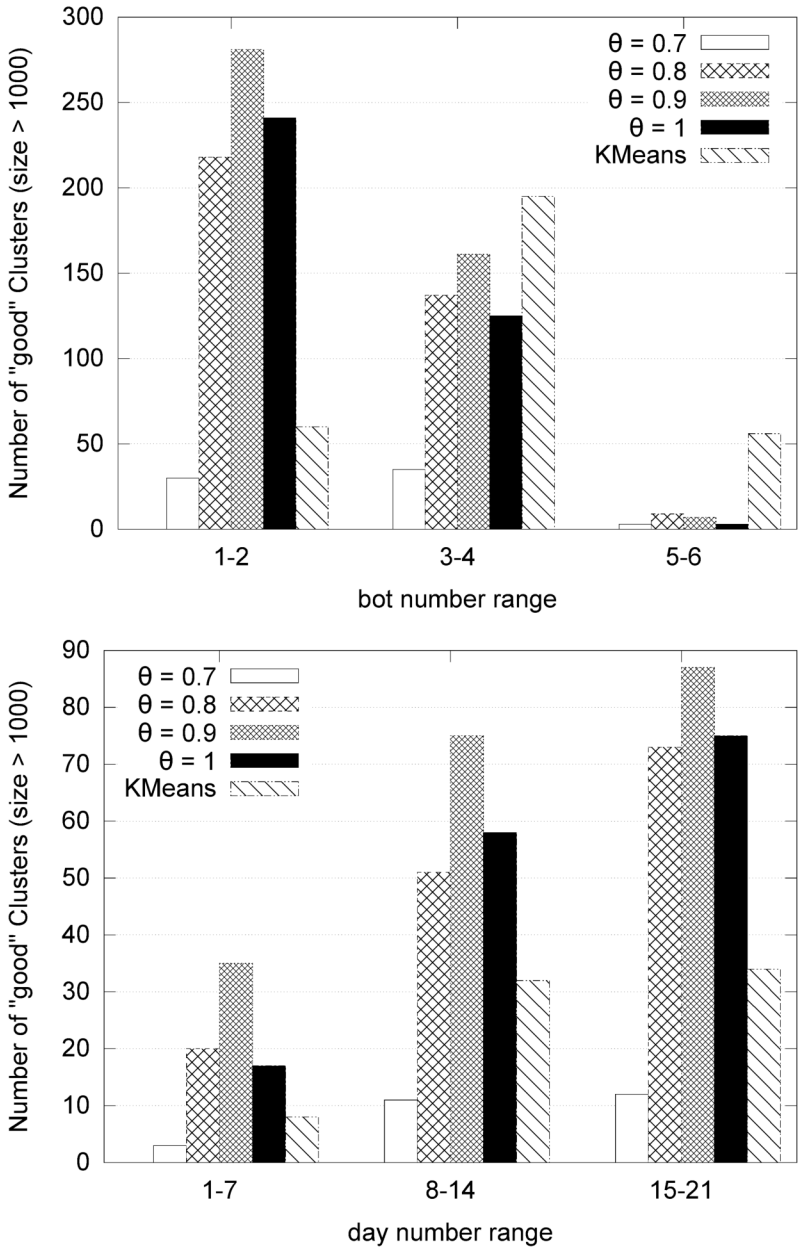

Figure 6.8: Clustering quality in terms of the unique number of features in each cluster output by clustering algorithms. For manual inspection to be useful, a small number of unique feature, such as bot-nets or time-frame of each cluster item, is preferred. Our approach outperforms the baseline algorithm.

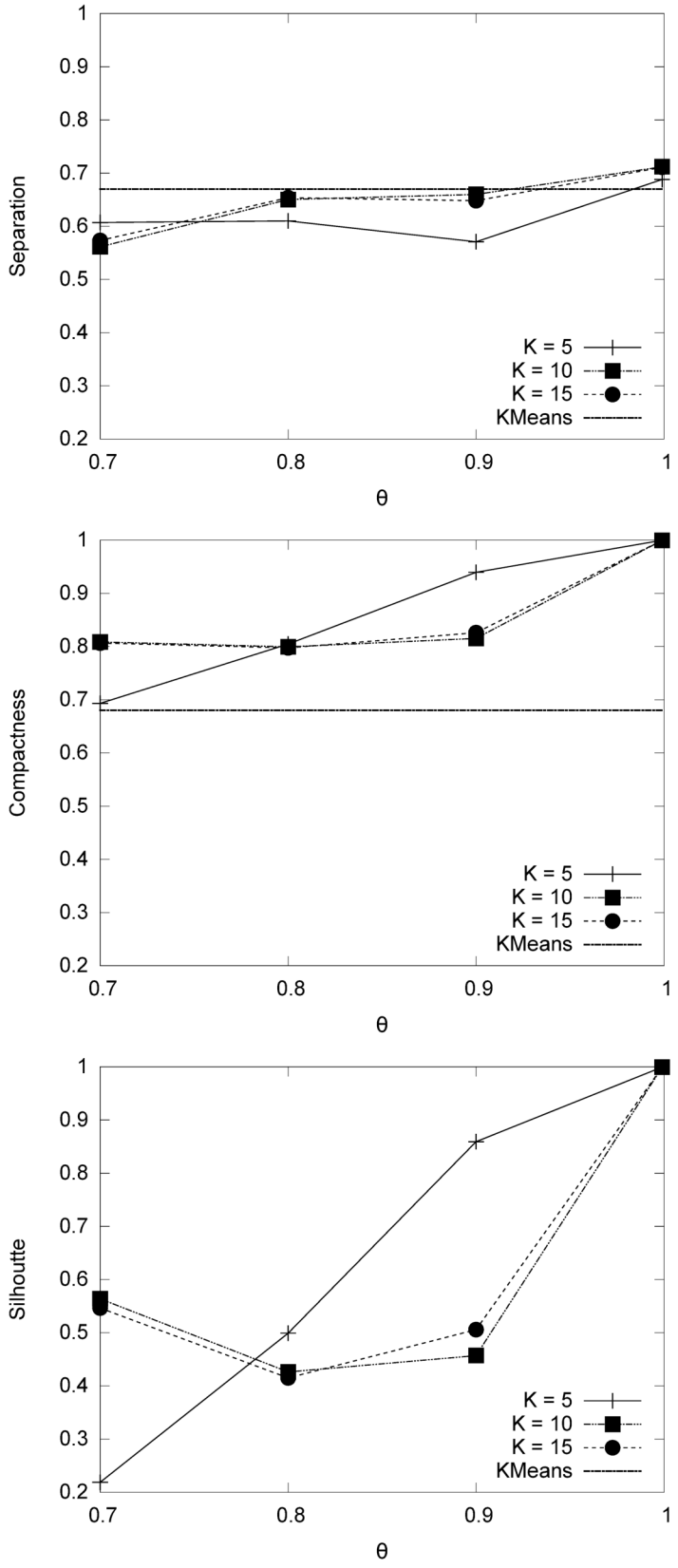

Figure 6.9: Twitter Dataset: Clustering quality in terms of inter and intra cluster similarity, and clustering Silhouette. As for the Symantec dataset, the algorithm proposed in this work outperforms the baseline method in all metrics.

Table 6.4: Twitter Dataset: Manual Investigation

cluster size	sample
1542	Wind 9.1 km/h WSW. Barometer 1002.0 hPa, Falling. Temperature 0.5C. Rain today 0.3 mm. Humidity 34% Wind 0.0 mph —. Barometer 1022.4 mb, Steady. Temperature 21.4F. Rain today 0.00 in. Humidity 79% Wind 1.9 km/h S. Barometer 1026.5 hPa, Falling. Temperature -9.6C. Rain today 0.0 mm. Humidity 84%
194	"#IfIHadItMyWay I would have a fast metabolism so I could eat MORE!" "#IfIHadItMyWay Chicken strips would be served everyday at ranger" "#IfIHadItMyWay school would be just gym. And back home"
564	Miami Valley Hospital: CLINICAL NURSE - CNN (#TROY , OH) http://t.co/adVBYzCK #Nursing #Job #Jobs #TweetMyJobs Miami Valley Hospital: OB SURGICAL TECH (#DAYTON , OH) http://t.co/Bh80vn0k #Healthcare #Job #Jobs #TweetMyJobs Miami Valley Hospital: CLINICAL NURSE-CN (#FRANKLIN , OH) http://t.co/ry2YR2yK #Nursing #Job #Jobs #TweetMyJo
228	"#PrettyLittleLiars was intense!!" "#prettylittleliars marathon!" "Pretty Little Liars with breakfast :)"
773	"I just became the mayor of Roszkowski Haus on @foursquare! http://t.co/4RzAisZg" "I just became the mayor of Bertha's Place on @foursquare! http://t.co/t07P13EL" I just became the mayor of Mini Mini Mart on @foursquare! http://t.co/d6xFIIQM

Table 6.5: Symantec dataset: breakdown of the algorithm runtime (in seconds)

k-nn graph phase	**Iteration**				
k	5	10	15	20	**CC**
5	675	2293	3185	4897	66
10	2281	4610	5320	7061	81
15	4061	8475	13594	18203	107

Twitter dataset. In this section, we cross-validate our results with the Twitter dataset, which includes $1,530,623$ tweets sent in USA and collected on the day 2012/02/21. This dataset is fundamentally more diverse than the Symantec dataset: the average weight on the generated k-nn graph (*i.e.,* the similarity of the neighbours) is 0.75, as compared to 0.96 for the Symantec dataset.

Figure 6.9 shows separation, compactness and Silhouette, as a function of θ, and for various values of k, for both our approach, and for the baseline algorithm based on K-means (with $K = 1000$). A glance at the Figure indicates that our approach achieves similar or better performance than the baseline method for clustering. It is interesting to notice that for the Twitter dataset, $k = 5$ – which produces very rough approximations of the k-nn graph – achieves better performance than for larger values of k.

We conclude the analysis of the Twitter dataset with a manual investigation of the clusters, as shown in Table 6.4. We focus on clusters with at least 100 tweets: in this case, a smaller cluster size is justified by the lower similarity between tweets as compared to the Symantec dataset. Our results confirm that clusters are meaningful, for example clustering weather forecasts in one case and job positions at the Miami hospital in another. We also have identified clusters related to hashtags such as *#IfIHadItMyWay* and a popular TV-series, *#PrettyLittleLiars*.

6.5.3 Analysis of algorithm scalability.

Next, we study the scalability of our approach and compare it to the baseline algorithm discussed earlier: first, we vary the dataset size maintaining the same number of compute machines that execute the parallel algorithms, then we keep the dataset size constant, and increase the level of parallelism by adding compute machines to our compute-cluster.

Figure 6.10 shows the algorithm runtime with varying dataset sizes, using 5 different samples of the Symantec dataset of size 100,000, 200,000, 400,000, 800,000 and 1,600,000 emails respectively. All values plotted are the average of 5 independent executions. Our results indicate roughly a linear scalability with respect to dataset size, an observation that holds irrespectively of the value of k.

Figure 6.10 also shows the algorithm runtime as the number of cores we devote to the computation varies between 4 and 64, considering datasets 400,000 and 800,000 items; in both cases, our results indicate a quasi-linear speed-up, especially for the biggest dataset. For example, increasing doubling the number of cores from 8 to 16, for the large dataset, cuts almost in half the algorithm runtime.

Finally, Table 6.5, reports the runtime breakdown of the k-nn graph construction phase

Table 6.6: Symantec dataset: K-means, baseline algorithm runtime (in seconds)

	K	**time**
K**-means**	1000	3498 (1.53×)
	2000	10004 (4.39×)
	3000	28411 (12.46×)
	4000	56008 (24.55×)
Our approach, $k = 10$ **and 5 iterations**		**2281**

for various numbers of *iterations*, and of the connected component phase of our algorithm, for several values of k and for $\theta = 0.9$. Again, all values are the average of 5 independent executions.

As expected, the k-nn graph construction runtime increases both with k and with the *iteration* number, although more slowly than the worst scale asymptotic analysis and experimental results presented in [37].[3] In addition, it is important to notice that the first phase of our approach dominates the overall algorithm runtime, as computing the connected components is fast. Once the k-nn graph is built, it is possible to quickly proceed with various versions of the pruning phase (tuning θ for the application at hand) and obtain different clusters.

Table 6.6 illustrates the runtime of the baseline algorithm that uses K-means: the table reports the "slow-down" of the baseline algorithm with respect to our approach, when $k = 10$ and with 10 *iterations*, and for different values of K, the number of clusters K-means constructs. Our approach outperforms the baseline algorithm in terms of end-to-end clustering times, even for small values of K.

6.6 Conclusion

Exploratory data analysis requires fundamental techniques to understand, describe and eventually extract value from large amounts of data. One of such techniques is data clustering. In this chapter we presented a scalable approach for text data clustering, that accommodates arbitrary similarity measures and that produces high quality clusters.

To overcome the quadratic nature of typical approaches to text clustering, we studied the role of approximation in establishing a trade-off between high clustering quality and fast algorithmic runtime. We showed, through a detailed experimental campaign, that our method does not require accurate representations of pairwise similarity across data items to produce high quality, interpretable clusters. We supported our claims using real traces covering adversarial applications aiming at identifying SPAM campaigns, and through manual inspection by domain experts of the clusters output by our algorithm.

Our next steps include the design of an LSH-based k-nn graph algorithm supporting arbitrary similarity metrics to push approximation even further, the extension of our method to a density-based approach and, ultimately, to take into account multiple, heterogenous features of the data.

3. Recall that we use a different parallel execution framework in our work, Spark, which is geared towards efficient execution of iterative algorithms.

6.7 Contributions

— Alessandro Lulli, Thibault Debatty, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. Scalable k-nn based text clustering. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*, BIG DATA '15, pages 958–963, Washington, DC, USA, 2015. IEEE Computer Society

— Our Spark implementation of k-nn graph based text clustering is available at `https://github.com/alessandrolulli/knnMeetsConnectedComponents`

This idea has been further developed in a follow-up paper, although the indexing structure is not a pure k-nn graph anymore: Alessandro Lulli, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. Ng-dbscan: Scalable density-based clustering for arbitrary data. *Proc. VLDB Endow.*, 10(3):157–168, November 2016

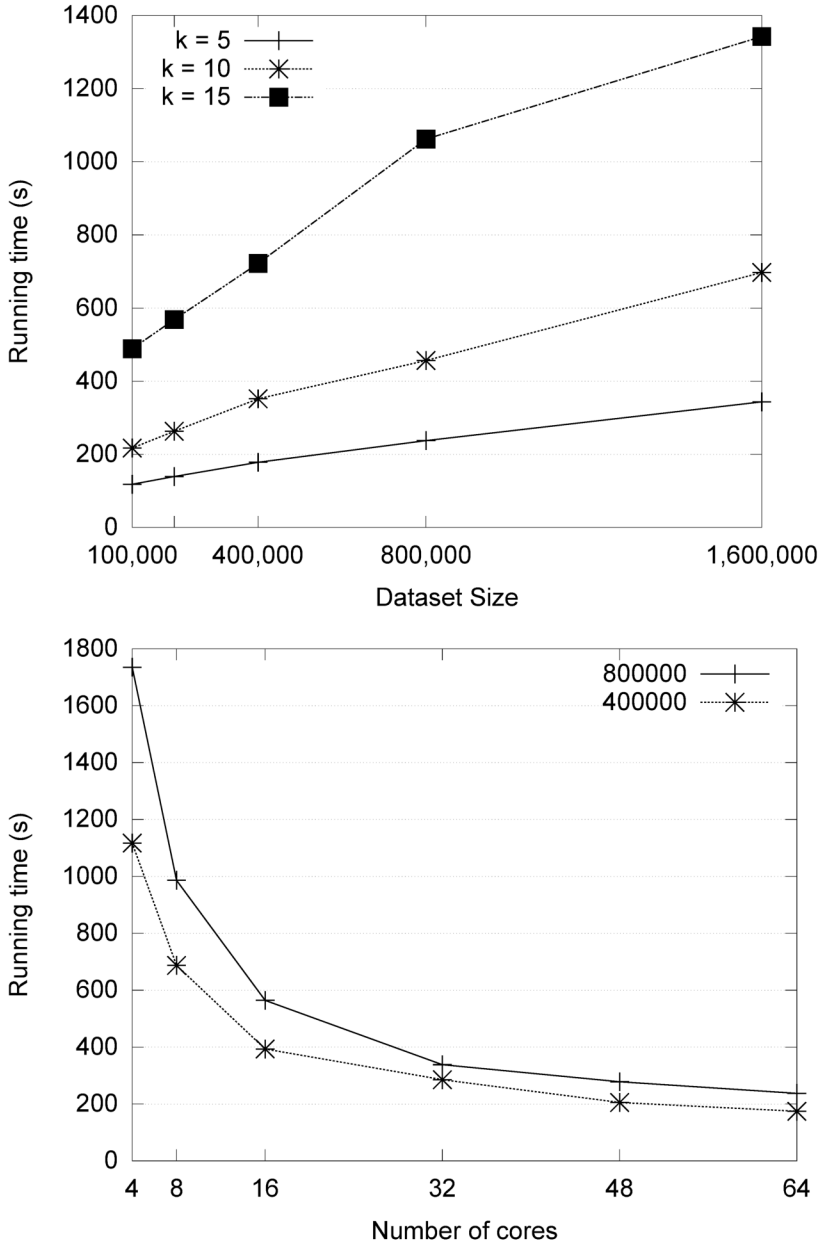

Figure 6.10: Scalability analysis, varying dataset sizes and compute-cluster sizes. Our approach scales roughly linearly.

Chapter 7

Graph based APT detection

7.1 Introduction

Advanced Persistent Threats (APT) are targeted cyber attacks committed over a long period of time by highly skilled attackers.

An example of an APT is the Miniduke attack [84] that targeted the governments of at least 20 countries including the Czech Republic, Ireland, Portugal, Romania and the United States. The malware infected PCs when victims opened a cleverly disguised Adobe PDF attachment to an email, which was specifically tailored to the target. The attachment referred to highly relevant subjects like foreign policy, a human rights seminar, or NATO membership plans.

Such attacks are becoming evermore sophisticated and manage to bypass the state of the art commercial-off-the-shelf protections that are currently in place. The attackers regularly succeed in remotely controlling hosts in our networks long enough to locate the information they are after, gain access to it and finally exfiltrate sensitive data. APT attacks have therefore become a major concern for network security professionals around the world.

All APT's have some characteristics in common. First, they use advanced techniques like 0-day attacks and social engineering to infect the target organization. This makes them impossible to detect using regular, signature based, detection tools like antiviruses and intrusion detection systems (IDS).

Second, once a computer is infected, an APT will try to establish a communication channel with a command and control (C2) server outside the organization. This channel will be used to download a payload, to download further instructions or to exfiltrate data. This link can be established using any protocol allowed through the borders of the organization: HTTP, DNS, SMTP, or even SIP.

In a security conscious organization, however, all these protocols should take place through some sort of choke-point. For the HTTP protocol, this would be a proxy server installed in the demilitarized zone (DMZ). This allows to log all connections taking place with servers located outside the organization, and offers a chance to detect the activity of the APT.

In this chapter, we focus on the detection of APT's that use the HTTP protocol to establish a communication channel with their C2 server. Naive APTs perform connections to their C2 server at fixed or variable time intervals. This results in traces in the proxy logs that are quite easy to detect, like on top of Figure 7.1. The most evolved APT's however are able to sense user activity and wait for outgoing traffic to perform connections to the C2 servers. This makes them much more difficult to detect, like on the bottom of Figure 7.1. In this chapter we focus on detecting the latter, activity-sensing APT's.

Therefore we build a graph of HTTP traffic. In this graph, the APT becomes an anomaly that can be detected. The rest of this chapter is organized as follows: in Section 7.2 we show how we model HTTP traffic, and we explain how this graph can be used to detect APT's; in Section 7.3 we present how we implemented this algorithm in a complete detection system; in Section 7.4 we perform an experimental evaluation and we study the impact of the different parameters of the algorithm using data collected on a real network; finally, in Section 7.5 we present our conclusions.

7.2 Graph modeling of HTTP traffic

7.2.1 Modelization of legitimate traffic

When a user is browsing the Internet, each page requested by the browser triggers multiple HTTP request. Usually, the first request contains HTML code, then the browser downloads javascript, CSS, font files and images referenced in the code. These may further trigger the download of other files, or trigger AJAX requests, and so on. Each page visited by a client can thus be represented by a tree, where the root is the original page requested by the user, and each subsequent request has a single edge (a link) to the request that triggered this download. These trees can be built from the logs of the proxy server. This is depicted on Figure 7.2. The user successively opened three pages (pink, green and blue), which triggered a number of requests that can be observed in the logs of the proxy server.

In real-life, reconstructing the HTTP graph is much more complicated. First, requests belonging to multiple pages do frequently take place at the same moment. This can lead to the construction of incorrect trees. This is illustrated in Figure 7.3.

Second, as most pages on the Internet are built dynamically, loading the same page multiple times usually results in the execution of different, although similar, series of requests.

Finally, browsers try to keep in cache memory content that is not supposed to change, like images, javascript and css files. This also modifies the requests executed to display the same page.

Therefore we actually build a weighted graph, a graph where each node may have edges (links) to multiple other nodes, and each edge has a weight. In our graph, the weight of the edge from request B to request A indicates the probability that request B is a consequence of request A. How we exactly compute these probabilities is explained below. Hence, a request (B) may have edges to multiple other requests, each indicating the probability that request A is a consequence of each other request.

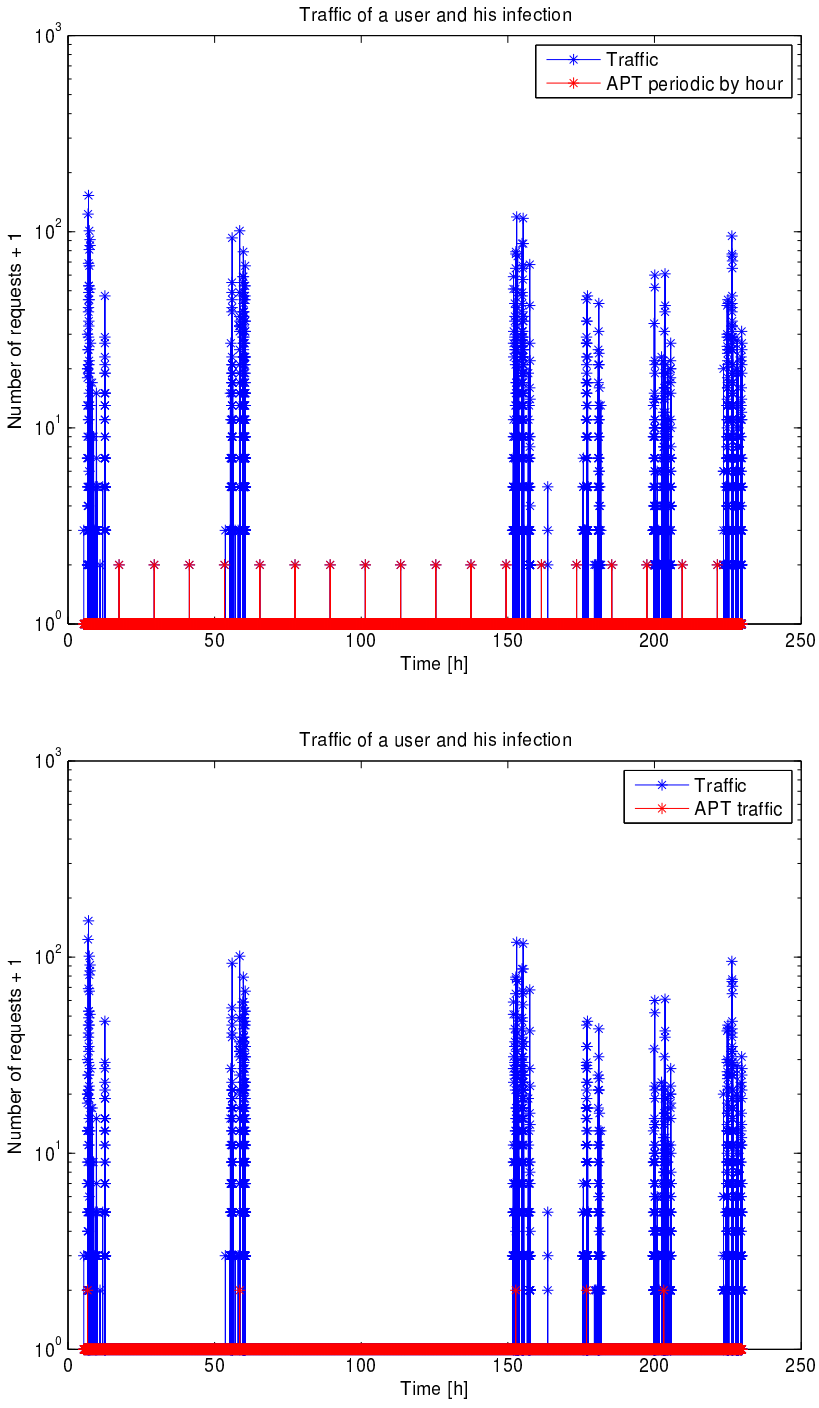

Figure 7.1: HTTP traffic generated by a single computer, infected by a frequency-based APT (top) and a sensing APT (bottom).

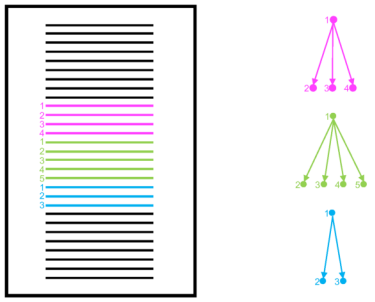

Figure 7.2: Graph model of HTTP traffic reconstructed from the logs of a proxy server

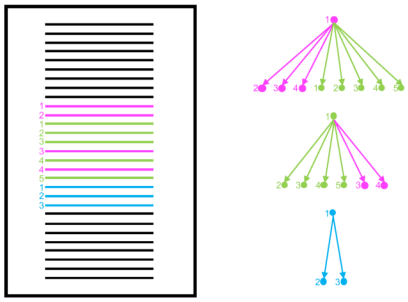

Figure 7.3: Graph model of HTTP traffic reconstructed from the logs of a proxy server when multiple pages are loaded in parallel.

7.2.2 Detection of APT traffic

When an APT waits for legitimate traffic (caused by a user browsing the Internet) to contact its C2 server, the requests performed by the APT take place roughly at the same moment as other requests from the graph. Hence, these requests have weak edges to multiple other requests.

This is illustrated in Figure 7.4: the requests performed by the APT happen together with the requests caused by the pink page, the green page and the blue page. As a consequence, the APT request has weak edges to all three root-requests.

A simple way to detect those requests is thus to prune the graph, which means to cut all edges whose value is lower then a threshold. The APT requests then appear as isolated

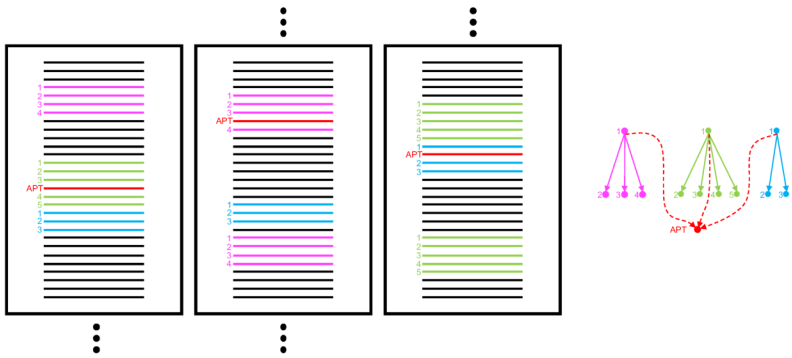

Figure 7.4: Graph model of HTTP traffic reconstructed from the logs of a proxy server with the impact of a sensing APT.

Figure 7.5: Web interface of the detection system

nodes, while other legitimate requests are linked together into clusters.

7.3 Implementation

To test the algorithm, we implemented a complete detection system. The system offers a web interface that allows the analyst to interactively analyze the requests taking place on the network using his browser. The detection relies on multiple parameters that the analyst can easily modify to spot hidden APT's. A screenshot of the web interface is presented in Figure 7.5. This view was generated using log files from a real network, hence the domains were anonymized. APT's were simulated by injecting requests in the log.

The system consists in two components: 1) the batch processor and 2) the web interface that allows interactive analysis of the data.

7.3.1 Batch processing

The system is designed to allow the analysis of HTTP traffic generated by large networks of computers. In such networks, millions of requests are generated each day. Storing a complete (fully connected) graph requires to store $O(n^2)$ values, where n is the number of nodes in the graph. Hence, storing a graph containing just 1 million requests requires roughly 1TB of storage. With current hardware, this is heavy to store on disk and almost impossible to store in memory unless a supercomputer is used. Hence, we instead build a k-nearest neighbors (k-nn) graph, a graph were each node has an edge to the k most similar other nodes in the graph. These graphs are a close approximation of a complete graph if k is large enough. However, they have the huge advantage that their memory requirement is linear in n (namely kn), which makes them also faster to process [65].

Before the k-nn graph is stored, it has to be computed, which requires to compute $O(n^2)$ similarities using a naive algorithm. This is not acceptable either for the large graphs we envision. Hence we use instead a fast approximate algorithm called nn-descent that requires to compute $O(n^{1.14})$ similarities to build the k-nn graph [37]. Moreover, this algorithms can easily be implemented in parallel, which allows to take advantage of all the cores available on the processing server.

Even using theses two optimizations (k-nn graphs and nn-descent algorithm), building the graph is a computationally heavy process that requires a non-negligible amount of time. Hence the system is split in two separate components: a batch processor and a web interface to perform the analysis.

The batch processor is responsible for computing the initial graph, without applying any detection. As the name states, this time consuming processing is meant to be run only once for each dataset.

Moreover, the system is designed to let the analyst tweak the definition of similarity between requests. Indeed, multiple criteria can be used to measure the probability that request B is a consequence of request A:

— requests A and B belong to the same domain;

— request B took place shortly after request A;

— etc.

In a naive implementation, modifying the definition of similarity, even slightly, requires to recompute the complete k-nn graph. Once again, this is a time consuming operation that would not allow to interactively analyze the data. Instead, during batch processing, we compute multiple graphs: one for each elementary similarity.

The time graph is built using the following measure of similarity:

$$\mu_{\text{time}} = \frac{1}{1 + |\delta t|}$$

where δt is the time difference (in seconds) between two requests.

The domain graph, however, is built using the following measure of similarity:

$$\mu_{\text{domain}} = \frac{\beta(A, B)}{\max(\gamma(A), \gamma(B))}$$

In this equation, $\beta(A, B)$ is the number of labels in common in the domain names of request A and B, starting from the Top Level Domain (TLD), and excluding the TLD itself. For example, $\beta(\text{cnn.com}, \text{www.cnn.com}) = 1$ because they have the label "cnn" in common. $\gamma(A)$ is the number of labels in the domain name of A, without taking the TLD into account. For example, $\gamma(\text{www.cnn.com}) = 2$

The complete batch processing thus involves the following steps:

— **Split** The data is first split between the different clients in the network, which correspond to the "source IP" field in the proxy logs;

— **Graphs** For each client, the different k-nn graphs are computed

— **Domains** As we suspect the APT will use different URL's belonging to the same domain, the different requests from the same domain are merged together to build graphs of domains;

— **Save** The graphs of domains are saved to disk.

The batch processor only takes one parameter, k, the number of edges per node in the k-nn graph. The impact of varying this parameter is shown below.

7.3.2 Interactive analysis

Once the batch processor has computed the k-nn graphs, these can be interactively analyzed using the web interface. The analysis mainly requires to: 1) merge the different k-nn graphs, 2) remove weak edges (pruning), 3) cluster the graph, 4) filter the graph to show only isolated domains and 5) rank the remaining suspicious domains.

In this process, the operator can provide several parameters to improve the detection.

First, he can choose which **clients** from the network are analyzed. Merging the graphs corresponding to multiple clients reinforces the edges between naturally related domains. This makes the domains contacted by the APT more isolated and hence easier to detect.

The analyst can also modify the definition of **similarity** used to link requests by providing a different weight for the time similarity and for the domain similarity.

He can provide a **pruning threshold** to remove weak edges from the the final graph. A high threshold removes a lot of edges in the graph, which leaves a lot of requests isolated. This causes a higher number of false positives. At the opposite, a lower value leaves almost all edges unaffected. Hence the requests generated by the APT have a higher probability of remaining connected to other requests, which decreases the probability of detection. The pruning threshold can be defined as an absolute value or as a z-score. The relation between a z-score z and an absolute value x is defined as follows:

$$z = \frac{x - \mu}{\sigma}$$

where μ is the average of the values and σ is the standard deviation.

Using z-scores allows to specify values that are independent of the data.

For the filtering step, the analyst can provide a **maximum cluster size**. Theoretically, after the pruning step the APT is supposed to be completely isolated. However, if the pruning threshold is chosen slightly too low, the APT may remain connected to some other domains, thus creating a small cluster. By using a filter to show only small clusters, the analyst may be able to spot the APT.

To further filter the results, the analyst can specify a **minimum number of requests per domain**. During our experimental evaluation with real data, we discovered that regular HTTP traffic contains a lot of domains that have very few requests each. Because they have very few requests, they are weakly connected to other domains, and are thus considered

as suspicious by our system. An APT, however, has to regularly contact its C2 server to download further instructions or to exfiltrate data. Hence we give the analyst the possibility to filter out domains with very few requests.

Finally, the system performs a **ranking** of remaining suspicious domains. Therefore we use three parameters, with weights provided by the analyst: 1) the number of requests to this domain, as a stealthy APT is supposed to perform only a few requests, 2) the number of child domains in the graph and 3) the number or parent domains in the graph, as the domain used by an APT is supposed to be weakly connected to multiple other domains.

In summary, the following steps are executed to perform the analysis:

— **Load** The k-nn graphs are read from the disk;

— **Feature fusion** The domain and time graphs are merged using the weights provided by the analyst;

— **Users fusion** The graphs corresponding to the different users selected by the analyst are merged;

— **Pruning** Weak edges are removed from the graph;

— **Clustering**;

— **Filtering** Clusters larger then a threshold provided by the analyst are removed;

— **Filtering** Domains that don't have enough requests are removed;

— **Ranking** The remaining suspicious domains are sorted using weights provided by the analyst.

Although this may seem heavy, processing k-nn graphs is actually extremely fast. This allows to interactively analyze huge amounts of data.

7.4 Parameter study and experimental evaluation

7.4.1 Test setup

To test the system, we use the logs from the proxy server of a real, large organization. From this dataset, we chose a subnet consisting of 26 computers, and a time period of 10 days. This subset contains a total of 721 921 requests.

In this log file, requests are inserted to simulate the activity of 4 APT's in the network. These APT's are chosen to exhibit different typical behaviors. The least stealthy APT generates 239 requests to its C2 server while the most stealthy APT performs only 13 requests in total, which makes it very difficult to detect.

To test the quality of detection, we build the receiver operating characteristic (ROC) curve of the detection system: we generate a ranking of the requests according to their suspiciousness using the provided parameters set. Then we walk the list from the top, and we compare each request to the known list of requests performed by the APT's. At each level of the list, we compute the probability of detection (p_d) and probability of false alarm (p_{fa}):

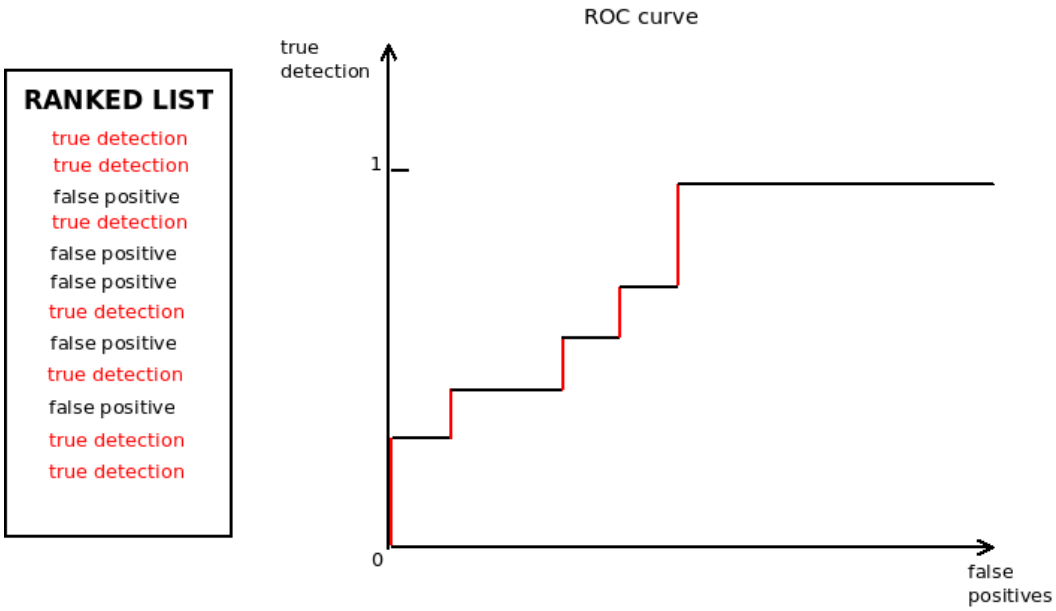

Figure 7.6: Building the receiver operating characteristic (ROC) curve of a detection system

$$p_d = \frac{\text{number of domains corresponding to an APT}}{\text{number of domains considered in the list}}$$

$$p_{fa} = \frac{\text{number of legitimate domains}}{\text{number of domains considered in the list}}$$

This allows to draw the ROC of the system, like depicted in Figure 7.6.

Finally, to compute the quality of the detection, we compute the area under the curve (AUC). Indeed, for a perfect detection system $p_d = 1 \; \forall \; p_{fa}$, hence AUC $= 1$. At the opposite, the worst detection system has $p_d = 0 \; \forall \; p_{fa}$, hence AUC $= 0$.

7.4.2 Number of edges per node k

For the first test, with vary values for k, the number of edges per node in the initial graphs.

We use the following values for the tests:

Weight time similarity	0.5
Weight domain similarity	0.5
Pruning (z-score)	0.0
Filtering: max cluster size	1000000
Filtering: requests/domain/client	1
Ranking: weight parents	0.35
Ranking: weight children	0.35
Ranking: weight number of requests	0.3

The resulting ROC and AUC are presented on Figure 7.7.

Surprisingly, $k = 40$ seems to give better results then $k = 100$. To study this effect, we perform additional tests where we compare these two cases by varying the other parameters.

The values used for the different tests are presented in Table 7.1 and the results are shown on Figure 7.8.

Test	1	2	3	4	5	6	7	8	9	10
k	40	100	40	100	40	100	40	100	40	100
Weight time similarity	0.1	0.1	0.4	0.4	0.1	0.1	0.1	0.1	0.1	0.1
Weight domain similarity	0.9	0.9	0.6	0.6	0.9	0.9	0.9	0.9	0.9	0.9
Pruning (z-score)	0.0	0.0	0.0	0.0	-0.1	-0.1	-0.5	-0.5	0.0	0.0
Filtering: max cluster size					1000000					
Filtering: requests/domain/client					5					
Ranking: weight parents	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.5	0.5
Ranking: weight children	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.5	0.5
Ranking: weight number of requests	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.0	0.0

Table 7.1: Parameters used to compare $k = 40$ and $k = 100$

As we can see, although $k = 40$ does provide slightly better results in some cases, $k = 100$ is generally better, as we expected. From now on, we use $k = 100$ for all our tests.

7.4.3 Fusion weights

We now study the impact of varying the weights used to merge the graph built using time similarity and the graph built using domain similarity. To perform the test, we have to fix a value for all other parameters. Therefore we choose neutral values, even though we know these are suboptimal. For example, we use a large value for the maximum cluster size, such that we don't filter clusters out:

k	100
Pruning (z-score)	0.0
Filtering: max cluster size	1000000
Filtering: requests/domain/client	1
Ranking: weight parents	0.35
Ranking: weight children	0.35
Ranking: weight number of requests	0.3

The results are shown on Figure 7.9.

The best result is achieved when the weight for time based similarity is 0.1 and the weight for domain based similarity is 0.9. This shows that the fact that two requests belong to the same domain is a better indicator of the link between the requests than the fact that these two requests happen slightly at the same moment. This was expected, as APT's wait for activity to contact their C2 servers. From now on we use these values for other tests.

7.4.4 Pruning

We now vary the pruning threshold. To keep the test independent of the data, we actually vary the z-score of the pruning threshold. The other parameters used for the test are:

k	100
Weight time similarity	0.1
Weight domain similarity	0.9
Filtering: max cluster size	1000000
Filtering: requests/domain/client	1
Ranking: weight parents	0.35
Ranking: weight children	0.35
Ranking: weight number of requests	0.3

The results are shown on Figure 7.10.

7.4.5 Size of clusters

Now we vary the maximum size of clusters using following parameters:

k	100
Weight time similarity	0.1
Weight domain similarity	0.9
Pruning (z-score)	0.0
Filtering: requests/domain/client	1
Ranking: weight parents	0.35
Ranking: weight children	0.35
Ranking: weight number of requests	0.3

The results are presented on Figure 7.11. Interestingly, the best result is obtained when the maximum cluster size is 1 000,000, which shows that even after pruning, the APT's are usually still linked to some much larger clusters.

7.4.6 Minimum number of requests per client

This parameter is studied using the setup:

k	100
Weight time similarity	0.1
Weight domain similarity	0.9
Pruning (z-score)	0.0
Filtering: max cluster size	1000000
Ranking: weight parents	0.35
Ranking: weight children	0.35
Ranking: weight number of requests	0.3

The results are presented in Figure 7.12.

As we explained above, filtering out the domains that receive very few requests per day (less then 5) allows to drastically reduce the background "noise" and improves the quality of detection.

7.4.7 Ranking weights

We now vary the weights used to perform the ranking of remaining domains. We use the following parameters:

k	100
Weight time similarity	0.1
Weight domain similarity	0.9
Pruning (z-score)	0.0
Filtering: max cluster size	1000000
Filtering: requests/domain/client	5

We first vary the weight of the number of requests, and set the two other weights accordingly:

$$\frac{1 - \text{weight number of requests}}{2} = \text{weight number of children} = \text{weight number of parents}$$

The results are shown in Figure 7.13.

Using a negative value for the number of requests seems to provide the best results. However, this also first enlightens the less stealthy APT's, that perform a lot of requests to their C2 servers. Hence we recommend here to use a slightly positive value, namely 0.2.

7.4.8 Cross validation

We have now identified the parameters that seem to offer the best quality of detection. These parameters must of course be tuned by the analyst for the network under test, and for the kind of APT he tries to detect on the network (is the APT supposed to be more or less stealthy, etc.). However, we assume these are robust enough to represent a relevant starting point for the analyst.

To test this hypothesis, we use another subnet of the proxy log. This time, the subnet contains 66 clients, 10 days of data and 1 032 021 requests. We simulate the infection with 8 APT's and we analyze the data with the optimal parameters identified previously.

The resulting ROC is presented in Figure 7.14 and has an AUC of 0.9036. This shows the algorithm is very resilient and performs equally well with a different dataset. It also shows that the system quickly discovers 90% of the APT's. This is equivalent to the performance of most antiviruses on the market.

7.4.9 Detected domains

As we stated above, the dataset used to perform the tests is produced from the logs of the proxy server of a real network. Hence we manually analyzed the domains that were ranked as APT's by the system. These are mainly:

— Content Delivery Networks (CDN);

— domains that display advertising on multiple websites;

— domains that deliver Javascript libraries to multiple websites;

— websites with very few visits.

Although these are not C2 servers per se, they are characterised by the same behavior as the APT's that we are trying to detect: they have weak links with a lot of other domains. This shows that the algorithm is actually performing very well at detecting domains that behave like the domains used by an APT to contact its C2 server.

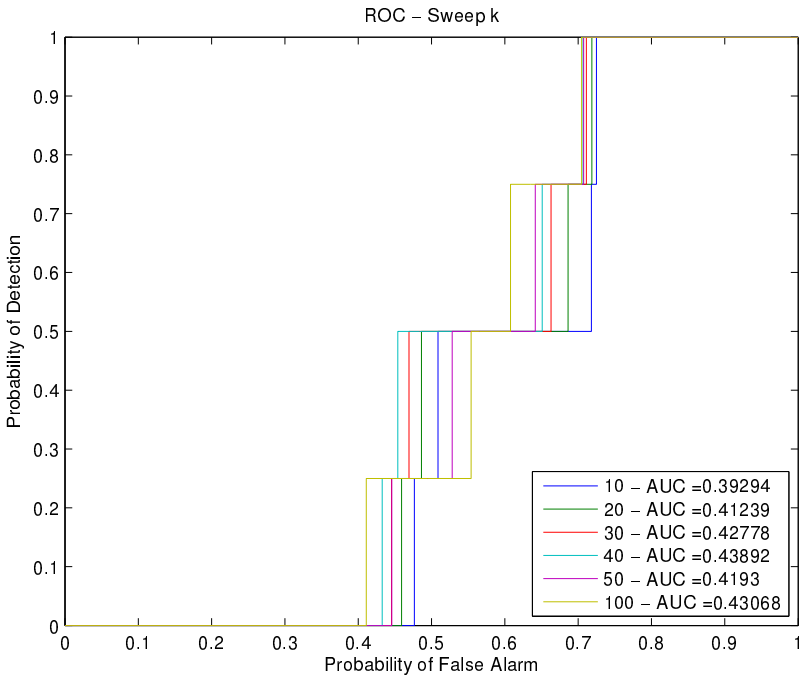

Figure 7.7: Result of varying k when building the graphs.

7.5 Conclusion and future work

The results obtained by our detection algorithm are very promising as they do not rely on a previous knowledge of the attacks. As a future work we plan to further tune the detection parameters, for example using deep learning, to further improve the quality of detection.

7.6 Contributions

— Thibault Debatty, Wim Mees, and Thomas Gilon. Graph Based APT Detection. In *Proceedings of the International Conference on Military Communications and Information Systems ICMCIS*, 2018

— The source code of the APT detection system is available at `https://github.com/RUCD/apt-graph`

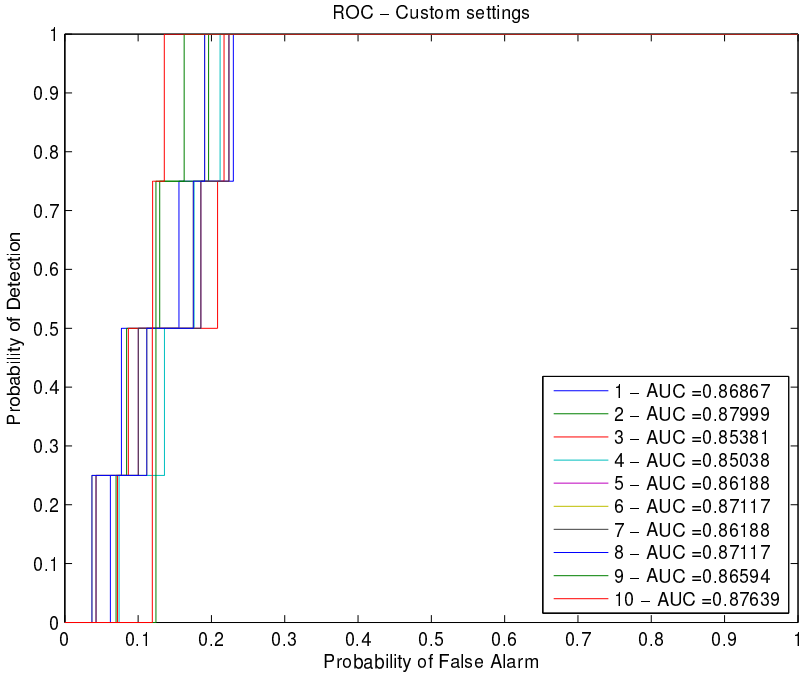

Figure 7.8: Result of comparing $k = 40$ and $k = 100$.

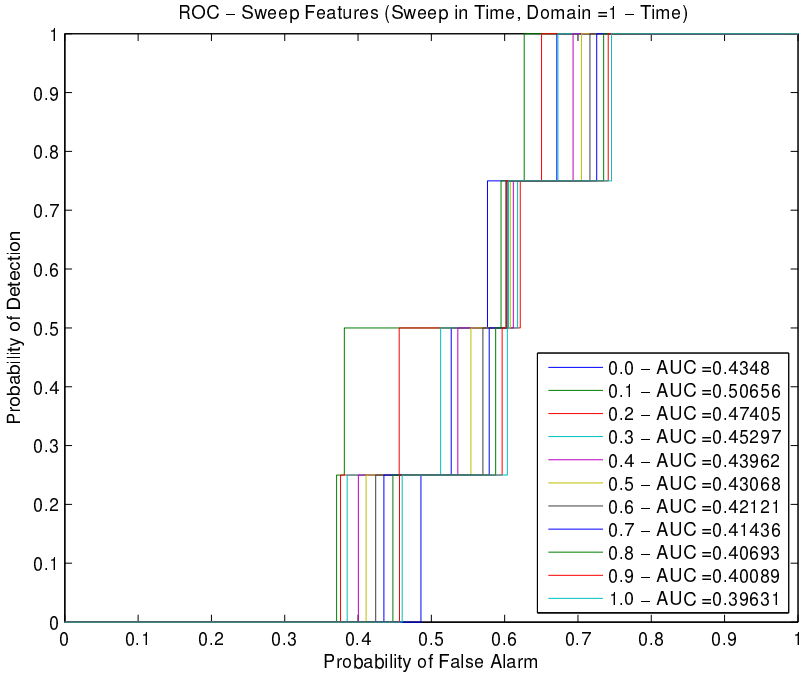

Figure 7.9: Result of varying the weights for merging the graphs.

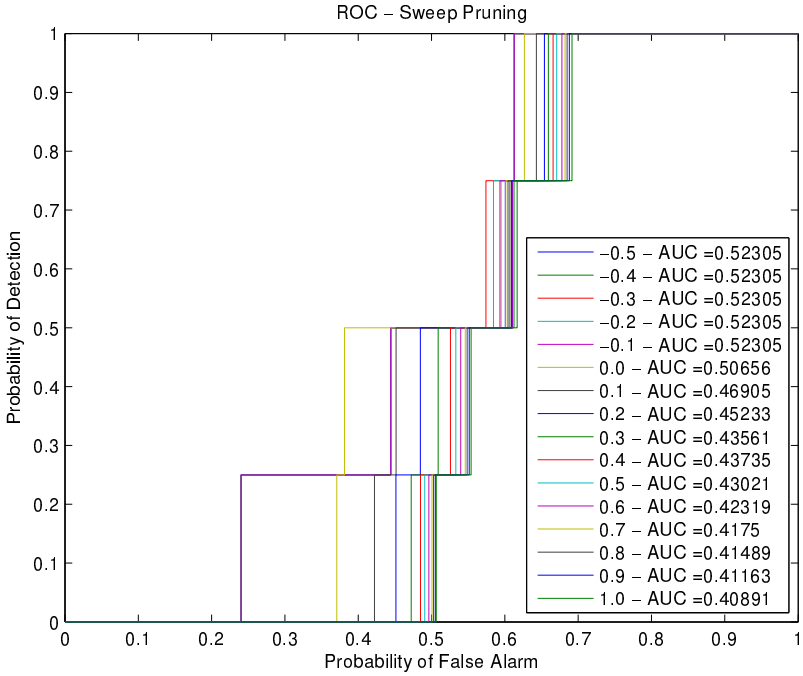

Figure 7.10: Result of varying the pruning threshold.

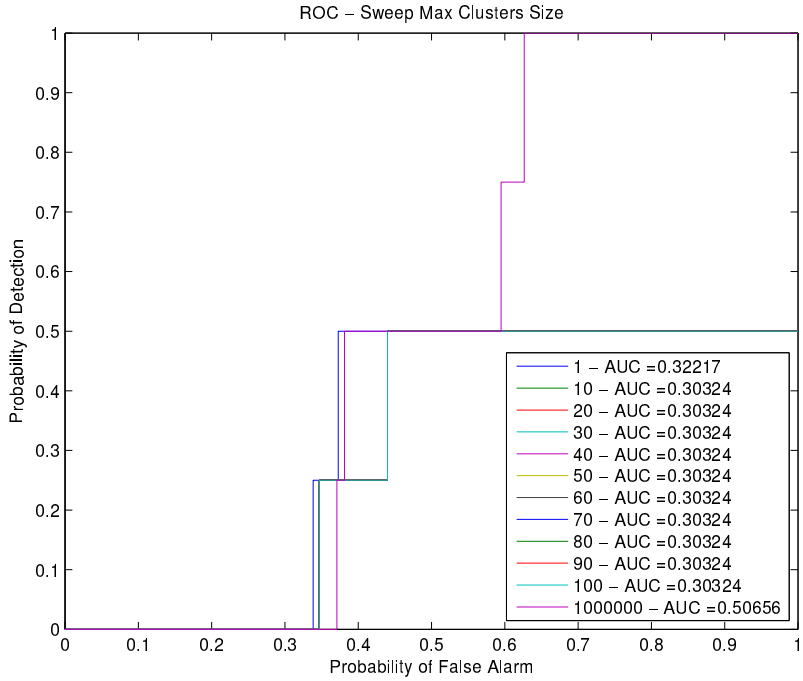

Figure 7.11: Result of varying the maximum cluster size.

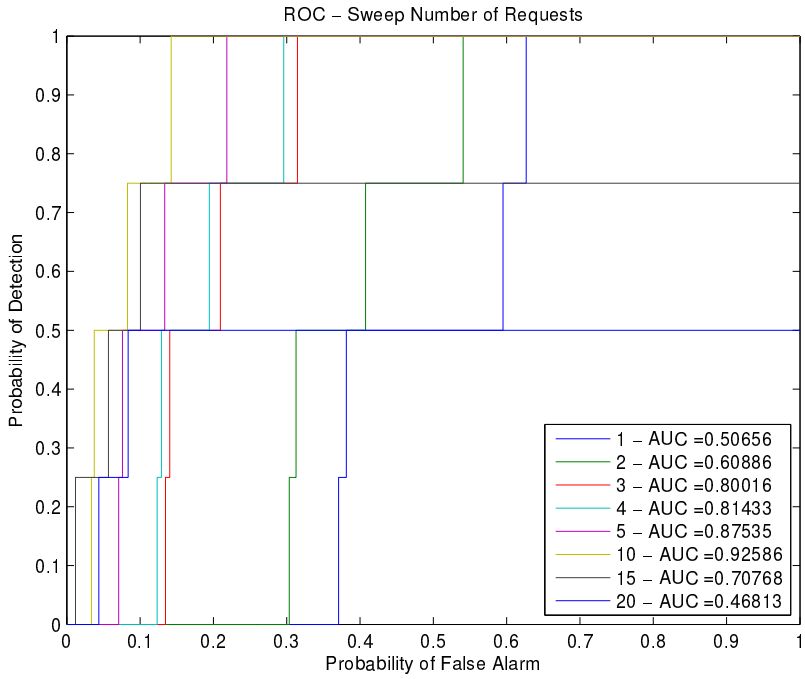

Figure 7.12: Result of varying the minimum number of requests per client.

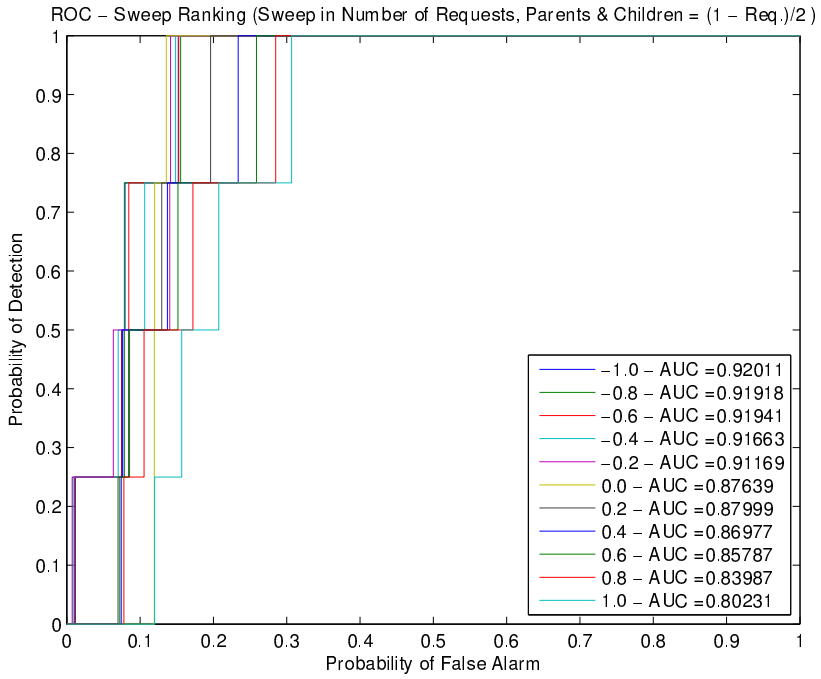

Figure 7.13: Result of varying the ranking weights.

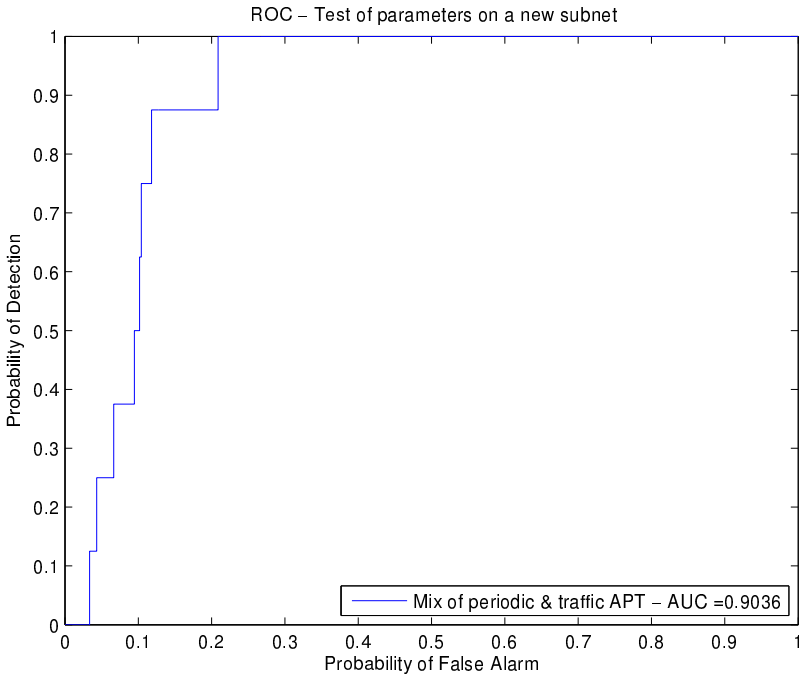

Figure 7.14: Result with a different dataset.

Chapter 8

Conclusions and perspectives

8.1 Conclusions

In this thesis we studied the usage of k-nn graphs to process large datasets.

In Chapter 3 we presented NNCTPH, a MapReduce algorithm that builds an approximate k-nn graph from large text datasets. We used datasets containing the subject of spam emails to experimentally test the influence of the different parameters of the algorithm on processing time and on the quality of the final graph. We also compared the algorithm with a sequential and a MapReduce implementation of NN-Descent. For our datasets, the algorithm proved to be up to ten times faster than the MapReduce implementation of NN-Descent for the same quality of produced graph. Moreover, the speedup increased with the size of the dataset, making NNCTPH a perfect choice for very large text datasets.

In Chapter 4 we proposed an algorithm that is able to update a distributed k-nn graph by quickly adding or removing nodes. We performed an experimental evaluation that shows the algorithm can be used with very large datasets and produces graphs that are highly similar to the graphs produced by a brute-force algorithm, while it requires the computation of far less similarities.

In Chapter 5 we studied the usage of k-medoids clustering to partition large k-nn graphs. We also proposed two new optimized procedures for performing k-medoids clustering. Then we proposed a method relying on k-medoids clustering to partition large k-nn graphs. Our experimental evaluation showed that 1) our clustering procedures outperform the current state-of-the-art and 2) k-medoids clustering is an excellent approach for partitioning large k-nn graphs as it is at the same time faster and more efficient than current partitioning algorithms.

In Chapter 6 we presented a scalable approach for text data clustering that relies on k-nn graphs. This new method accommodates arbitrary similarity measures and produces high quality clusters. To overcome the quadratic nature of typical approaches to text clustering, we studied the role of approximation in establishing a trade-off between high clustering quality and fast algorithmic runtime. We showed, through a detailed experimental campaign, that our method does not require accurate representations of pairwise similarity across data

items to produce high quality, interpretable clusters. We supported our claims using real traces covering adversarial applications aiming at identifying SPAM campaigns, and through manual inspection by domain experts of the clusters output by our algorithm.

Finally, in Chapter 7 we proposed an interactive detection system relying on k-nn graphs to detect APT's in a network. We also performed a complete experimental evaluation, using logs from the proxy server of a real network. The results obtained are very promising as we reached a high probability of detection, without relying on a previous knowledge of the attacks.

8.2 Perspectives

As we could show in this thesis, k-nn graphs are a powerful tool for data analysis. There are however other siblings data structures that are also good candidates for indexing large datasets.

A small-world network (SWN) is also a type of graph, but the neighbors of a node are not necessary the most similar other nodes. On the contrary, neighbors are built in such a way that most nodes can be reached from every other node by a small number of hops (or steps). Specifically, a small-world network is defined to be a network where the typical distance L between two randomly chosen nodes (the number of steps required) grows proportionally to the logarithm of the number of nodes N in the network: $L \propto \log n$.

This property makes the SWN suitable as an indexing structure for nearest-neighbors search, as the average number of hops required to reach a node from any random node in the graph is L. This has been shown in [70]. Nonetheless this also makes the efficient partitioning of a SWN much more challenging. As neighbor nodes can be very dissimilar, a large SWN has much more cross-partition edges if it is partitioned according to the k-medoids rule. A SWN might thus be much more efficient than a k-nn graph for sequential processing, but may not be an appropriate choice if the graph has to be distributed between different compute nodes.

In [71] the authors propose a related data structure, called a Hierarchical Navigable Small World graph (HNSW or Hierarchical NSW). This structure has multiple layers where:

— each layer holds a subset of the nodes in the lower layer and;

— each layer is a nearest neighbor graph.

This allows to nicely separate long edges (in the upper layers of the structure) from short edges (in lower layers of the structures). When the HNSW is used to perform nearest neighbor search, the upper layers can be used to quickly find a rough estimate of the nearest neighbors, then the lower layers are used to refine the solution.

At the same time, properly choosing the ratio for keeping nodes in the upper layers allows to contain the memory requirement of the data structure. In their paper, the authors show that HNSW outperforms other state-of-the-art approaches for performing nearest neighbor search.

However, just like SWN, the question remains to know whether this data structure remains as efficiently when implemented in parallel. Moreover, this data structure has been evaluated

only in the context of performing nearest neighbor search. Is it usable for performing other analysis like clustering for example?

It is interesting to note that, just like we did, the authors of these papers underline one main advantage of graph based indexes: they can be used with any measure of similarity, even non-metric.

An interesting extension of this work would be to broaden the scope of the analysis to graph based indexing structures in general (k-nn graphs, NSW, HNSW and possibly others).

In this thesis we focused solely on k-nn graphs, from two different viewpoints:

1. how to build or update them efficiently and;
2. how to use them.

These two aspects, although completely different, are also strongly related: one has no need to build a data structure if it cannot be used afterward to process the data, and a data structure that is too heavy to compute is equally useless.

Our perspective for the future is that those two aspects should be further investigated: how to build graph-based indexing structures, and how to use them for concrete applications. Of course, these graphs should also be compared against each other, and against other classical indexing structures.

Troisième partie

Résumé en français

Chapitre 9

Introduction

L'une des tendances actuelles dans le domaine des technologies de l'information est celle des grands ensembles de données (Big Data). Pour comprendre le phénomène, il faut revenir à sa naissance. Au cours des années 1990, le prix du stockage des données a chuté de façon spectaculaire, passant d'environ 40 000 euros par Go en 1987 à environ 2 euros par Go en 2002 et à 0,05 euros par Go en 2015 [42]. En raison des coûts de stockage prohibitifs, les entreprises ne conservaient jusqu'alors que les données strictement nécessaires à la gestion de la relation avec leurs clients (nom, adresse, services ou produits commandés, factures, etc.). De plus, les données obsolètes étaient rapidement supprimées pour économiser de l'espace et de l'argent. Avec la démocratisation du stockage de masse, ces entreprises ont également commencé à stocker toutes les informations concernant les interactions avec leurs clients potentiels, sans jamais rien effacer, et sans même se soucier de leur utilité éventuelle. Google et Yahoo ! par exemple, les précurseurs de Big Data, ont depuis stocké toutes les activités menées par leurs utilisateurs.

A l'heure actuelle, le phénomène ne fait que s'amplifier. La quantité de données produites chaque seconde dans le monde explose. Les objets connectés, les applications de commerce électronique, la sécurité ou les services financiers continuent de générer d'énormes quantités de données qui sont capturées et stockées. L'utilisation de ces données est actuellement considérée comme une clé de la compétitivité et de la croissance : l'analyse de ces données permet aux entreprises de découvrir des tendances et opportunités cachées.

Le traitement efficace de ces grandes quantités de données nécessite des structures d'indexation appropriées : sans index, chaque opération sur les données (recherche, cluster,...) nécessite de scanner l'ensemble des données, éventuellement plusieurs fois.

Il existe différentes structures d'indexation. Le type d'index le plus simple est un arbre binaire trié (aussi appelé arbre de recherche binaire, BST) [27]. Il suppose que les données à indexer peuvent être ordonnées (triées par ordre croissant). Une BST maintient une structure arborescente des données où les éléments sont triés. Ceci permet de rechercher un élément avec un temps de recherche $O(\log n)$, où n est le nombre d'éléments dans l'arbre. Figure 9.1 montre comment un BST peut être utilisé pour rechercher un élément dont la valeur est 4.

Les arbres B (B-trees) [25], les arbres B+ et B* sont des généralisations des arbres de recherche binaires qui stockent plusieurs valeurs à chaque niveau de l'arbre et ont plusieurs

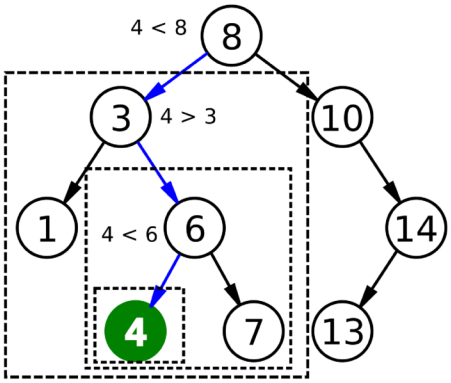

FIGURE 9.1: Exemple d'un arbre de recherche binaire utilisé pour rechercher l'élément d'une valeur de 4 [102].

branches. Ils sont couramment utilisés par les systèmes de bases de données et les systèmes de fichiers par exemple. Pour les points dans $\mathbb{R}^n$, comme les données géographiques, les arbres kd (kd-trees) et les arbres R peuvent être utilisés.

A côté de ces options classiques, un graphe k-nn est un candidat inhabituel mais intéressant. Un graphe k-nn est un graphe où chaque nœud possède une arête (un lien) vers les k autres nœuds les plus similaires de l'ensemble de données. Bien qu'il soit très simple comparé à d'autres index, il permet d'effectuer certaines tâches d'analyse courantes comme le clustering ou la recherche de similarité d'une manière efficace. Il possède trois autres caractéristiques intéressantes :

1. il peut être utilisé avec n'importe quelle mesure de similarité, même non métrique. Un graphe k-nn est donc un index polyvalent qui peut être utilisé pour de multiples applications, comme des ensembles de données textuelles équipés d'une mesure non métrique de similarité de chaîne par exemple. Ces types d'ensembles de données sont en fait très courants et seront utilisés tout au long de cette thèse pour l'évaluation expérimentale ;

2. il nécessite peu de mémoire : l'exigence de mémoire d'un graphe k-nn est proportionnelle à la taille de l'ensemble de données (i.e. $\propto kn$ où n est la taille de l'ensemble de données). Cela les rend appropriés pour de très grands ensembles de données ;

3. bien que la construction d'un graphe k-nn soit généralement une opération qui prend beaucoup de temps et de calculs, comme nous le montrerons au chapitre 10, traiter un graphe k-nn est généralement très rapide. Cela fait des graphes k-nn un choix intéressant pour l'implémentation d'outils d'analyse interactifs, comme celui que nous présentons dans le Chapitre 14.

Cette thèse porte précisément sur l'utilisation des graphes k-nn pour le traitement de grands ensembles de données. De plus, nous supposons que l'ensemble de données est si important qu'il ne peut être traité à l'aide d'un seul nœud de calcul, soit parce qu'il n'offre pas suffisamment de mémoire, soit parce qu'il ne peut traiter les données dans un laps de temps acceptable. C'est pourquoi nous nous concentrons sur les algorithmes distribués, qui peuvent tirer parti de plusieurs nœuds de calcul pour traiter les données.

Dans le reste de cette thèse, nous étudions :

— comment construire efficacement des graphes k-nn à partir de grands ensembles de données texte ;

— comment mettre à jour un graphe k-nn lorsque de nouvelles données doivent être insérées ou supprimées ;

— comment partitionner un grand graphe k-nn entre plusieurs nœuds de calcul ;

— comment utiliser les graphes k-nn pour effectuer le clustering de grands ensembles de données textuelles, et

— comment les utiliser pour détecter des ordinateurs compromis dans un réseau.

Chapitre 10

Construction à partir de données textuelles

Tel que mentionné précédemment, les graphes k-nn sont un outil puissant pour l'analyse de données. Cependant, comme nous le montrons ci-dessous, jusqu'à récemment, aucun algorithme n'était capable de construire rapidement un graphe k-nn à partir d'un grand ensemble de données textuelles. C'est pourquoi nous présentons ici un nouvel algorithme, appelé NNCTPH. Notre algorithme distribue les données textuelles en plusieurs groupes, puis calcule le sous-graphe à l'intérieur de chaque groupe et enfin fusionne tous les sous-graphes ensemble.

10.1 Travaux connexes : construction de graphes k-nn

Différentes approches existent pour construire un graphe k-nn. La méthode naïve, aussi appelée recherche linéaire, utilise la force brute pour calculer toutes les similarités par paires. Ensuite, pour chaque nœud, l'algorithme ne conserve que les liens k avec la plus grande similarité. Cette méthode a un coût de calcul de $O(n^2)$ et est donc très lente, voire implémentée en parallèle.

C'est pourquoi de multiples algorithmes ont été proposés dans la littérature pour accélérer le processus, par exemple en utilisant un index. Certains d'entre eux tolèrent des liens incorrects pour accélérer davantage le processus de construction et produire un graphe approximatif, tandis que d'autres produisent un graphe exact. Dans les deux cas, ces algorithmes de construction sont étroitement liés aux algorithmes de recherche du voisin le plus proche.

Un algorithme polyvalent pour calculer efficacement un tel graphe est décrit dans [37]. L'algorithme, appelé NN-Descent, commence par créer des liens entre des nœuds aléatoires. Ensuite, pour chaque nœud, il calcule la similarité entre chaque paire de voisins, afin de trouver de meilleurs liens. L'algorithme itére jusqu'à ce qu'il ne puisse plus trouver de meilleurs liens. Le principal avantage de cet algorithme est qu'il fonctionne avec n'importe quelle mesure de similarité. Les auteurs ont montré expérimentalement que le coût de calcul de l'algorithme est d'environ $O(n^{1.14})$.

L'article propose également une version MapReduce de l'algorithme. En interne, l'algorithme fonctionne avec une sorte de liste d'adjacence appelée liste des voisins (neighborlist). Cette structure contient les voisins d'un nœud. Une implémentation séquentielle C++ de l'algorithme est également disponible sous le nom KGraph [36].

Lorsqu'il s'agit de construire un graphe k-nn à partir d'un grand ensemble de données textuelles non structurées, aucun de ces algorithmes n'offre une solution efficace. Les algorithmes qui s'appuient sur des index sont difficiles à implémenter en parallèle sur une architecture de rien partagée comme MapReduce. Les fonctions LSH ne sont définies que pour certaines mesures de similarité (l_p, distance Mahalanobis, similarité du noyau, et χ^2 distance). Les algorithmes s'appuyant sur LSH ne peuvent donc pas être utilisés pour construire un graphe k-nn à partir de données textuelles en utilisant la distance d'édition (distance de Levenshtein) ou toute autre mesure de distance similaire (distance Levenshtein pondérée, distance Jaro-Winkler, distance Hamming) comme mesure de similitude.

Dans le cas de NN-Descent, la version MapReduce (MR) de l'algorithme nécessite deux tâches MR par itération et plusieurs itérations pour converger. De plus, l'algorithme nécessite de lire et d'écrire beaucoup de données sur disque entre les travaux. Bien que la version séquentielle de l'algorithme se soit avérée très efficace, ces contraintes la rendent inefficace lorsqu'elle est mise en œuvre en parallèle. Ceci sera confirmé par les tests expérimentaux présentés ci-dessous.

10.2 NNCTPH

Comme aucun algorithme courant n'est adapté pour construire un graphe k-nn à partir d'un grand ensemble de données textuelles, nous proposons ici un nouvel algorithme. L'algorithme nécessite une itération unique et un travail MapReduce unique, et il ne repose pas sur un index partagé. En interne, il utilise un schéma de hachage spécifique, appelé Context Triggered Piecewise Hashing (CTPH) pour distribuer les données d'entrée en différents groupes. C'est pourquoi nous appelons l'algorithme NNCTPH.

10.2.1 Context Triggered Piecewise Hashing

Context Triggered Piecewise Hashing (CTPH), aussi appelé Fuzzy Hashing ou ssdeep [57], est une fonction de hachage qui tend à produire le même hachage pour des chaînes d'entrée similaires. Il a été développé à l'origine par Tridgell comme un détecteur de spam appelé SpamSum [98]. L'algorithme est utilisé pour construire une base de données de hachs des spams connus. Lorsqu'un nouvel email est reçu, son hachis est calculé, et comparé à la base de données des spams. Si un hachis similaire est trouvé, le courrier entrant est considéré comme du spam, et jeté.

10.2.2 NNCTPH

Notre algorithme nécessite une seule tâche MapReduce. Dans la phase *map*, l'algorithme utilise une fonction CTPH modifiée pour produire un hachis de chaque chaîne d'entrée.

Cette valeur de hachis est ensuite utilisée pour distribuer les textes entre plusieurs groupes. Chaque tâche de réduction construit un graphe k-nn à partir des chaînes de caractères de chaque groupe.

Pour contrôler le nombre de groupes, et donc le nombre de textes par groupe, nous modifions la fonction CTPH d'origine pour : *i)* produire un hachis de taille variable ; et *ii)* utiliser seulement un sous-ensemble de lettres dans le hach, au lieu des 64 lettres originales.

De plus, si nous n'émettons chaque texte qu'une seule fois, nous obtenons une série de sous-graphies non connectés, car chaque texte est placée dans un groupe unique, et aucun lien n'est créé entre les texte des différents groupes. Pour reconnecter les graphes, l'algorithme crée un hachis plus long (en utilisant un coefficient que nous appelons étapes ou stages s) et émet la chaîne d'entrée une fois pour chaque sous-partie du hach.

L'algorithme nécessite donc trois paramètres : le nombre d'étapes (s), le nombre de caractères du hachis émis (c) et le nombre de lettres utilisées pour produire le hachis (l). Pour chaque étape, les chaînes d'entrée sont stockées dans un des l^c groupes, et la fonction CTPH modifiée doit produire des hachis d'une longueur de $c + s - 1$ en utilisant un alphabet de l lettres.

Ces paramètres ont un impact sur la qualité du graphe, sur la quantité de données à transmettre sur le réseau, sur le parallélisme de l'algorithme, sur la quantité de mémoire RAM requise pour l'analyse, et sur le nombre de similitudes à calculer.

Le nombre de groupes produits par la fonction de hachage est l^c. Si nous supposons que les chaînes d'entrée sont uniformément réparties dans les groupes (si les données ne sont pas biaisées), le nombre de chaînes par groupe est $\frac{n}{l^c}$. Dong *et al.* [37] ont montré expérimentalement que le coût de calcul de NN-Descent est d'environ $O(n^{1.14})$. Comme nous utilisons NN-Descent à l'intérieur des groupes pour construire les sous-graphes, le nombre de similarités à calculer est : $O(s \cdot l^c \cdot (\frac{n}{l^c})^{1.14})$. Si nous choisissons l et c de telle sorte que le nombre de chaînes par groupe (n/l^c) soit constant (avec un nombre de groupes proportionnel à la taille du jeu de données), cela signifie que le coût de calcul de notre algorithme est proportionnel au nombre de groupes, et donc proportionnel à la taille du jeu de données. Cec est naturellement une limite inférieure. Si les données d'entrée sont biaisées, ce qui est le cas de notre ensemble de données de test, le nombre total de similitudes à calculer est plus élevé.

Le nombre d'étapes s aura également un impact sur la qualité du graphe final : si plusieurs étapes sont utilisées, la même chaîne sera émise plusieurs fois. La probabilité de découvrir les liens corrects augmentera donc également. Le nombre d'étapes est également directement proportionnel à la quantité de données à lire et à transmettre sur le réseau. L'utilisation d'un nombre plus élevé d'étages ralentira donc l'algorithme.

10.3 Evaluation expérimentale

Pour analyser la performance de notre algorithme, nous l'implémentons à l'aide de Hadoop MapReduce et le testons sur des ensembles de données contenant le sujet de spam. Cet ensemble de données est un échantillon de spams collectés par Symantec Research Labs en 2010. Il est principalement utilisé pour améliorer les définitions des signatures de spams et pour analyser les tendances des campagnes de spam.

Tous les test sont exécutés sur un cluster de 20 nœuds de calcul, chacun équipé d'un processeur quatre cœurs, de 8 Go de RAM et de quatre cartes Ethernet 1 Go.

Pour calculer la similitude entre les sujets de spam, nous utilisons la distance Jaro-Winkler [103]. Cette mesure de similarité de chaîne de caractères est normalisée de sorte que zéro équivaut à aucune similarité et un est une correspondance exacte.

Pour mesurer l'efficacité de chaque algorithme, comme dans [37], nous calculons le rappel (recall), qui est le rapport entre le nombre de liens corrects trouvés par l'algorithme et le nombre total des liens créés.

Nous effectuons d'abord une analyse des paramètres pour identifier les valeurs optimales. Ensuite nous comparons notre algorithme avec notre implémentation Hadoop MapReduce de NN-Descent et avec la méthode de la force brute. Finalement, nous analysons l'efficience de l'algorithme lorsque la quantité de données à analyser augmente.

Les temps d'exécution et les rappels résultants sont affichés sur la Figure 10.1. Comme nous pouvons le voir, le rappel réalisé par NNCTPH avec ces paramètres est très stable, et le temps de fonctionnement augmente très lentement. Par conséquent, plus l'ensemble de données est grand, plus l'accélération est élevée par rapport à MR NN-Descent. Avec notre ensemble de données de 800 000 spams, nous atteignons une accélération de près d'un ordre de grandeur pour la même qualité du graphe final. On peut clairement constater que NN-Descent souffre de sa structure itérative, qui n'est pas bien adaptée au framework Map Reduce, et exige beaucoup d'opérations lentes de lecture et d'écriture sur le disque.

10.4 Conclusions

Dans ce chapitre, nous avons présenté NNCTPH, un algorithme MapReduce qui construit un graphe k-nn approximatif à partir de grands ensembles de données textuelles. Nous avons utilisé des ensembles de données contenant le sujet du spam pour tester expérimentalement l'influence des différents paramètres de l'algorithme sur la qualité du graphe et le temps de traitement. Nous avons également comparé l'algorithme avec une implémentation séquentielle et MapReduce de NN-Descent. Pour nos ensembles de données, l'algorithme s'est avéré jusqu'à dix fois plus rapide que l'implémentation MapReduce de NN-Descent, pour la même qualité de graphe produit. De plus, l'accélération augmente avec la taille de l'ensemble de données, ce qui fait de NNCTPH un choix parfait pour les très grands ensembles de données textuelles.

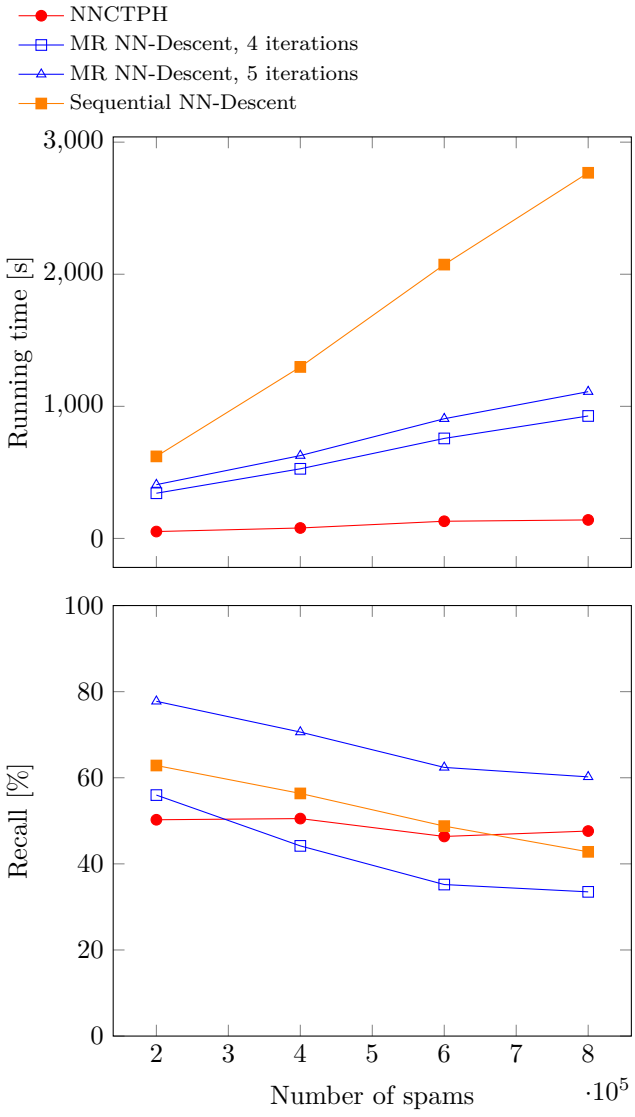

FIGURE 10.1: Comparaison entre NNCTPH et NN-Descent

Chapitre 11

Construction en ligne

Dans le cas général, la construction d'un graphe k-nn nécessite le calcul de $O(n^2)$ similarités, où n est la taille de l'ensemble de données. Un index peut être utilisé pour réduire le nombre de calculs, mais la construction d'un graphe k-nn exact reste une opération lourde.

De même, la modification d'un graphe existant par l'ajout ou la suppression de nœuds est un processus lourd en termes de calcul. Par exemple, l'ajout d'un seul nœud nécessite : 1) de calculer les liens du nouveau nœud et 2) de mettre à jour les liens des nœuds existants dans le graphe. Chaque nouveau point nécessite donc de calculer la similarité entre le nouveau point et chaque nœud du graphe existant. C'est donc très lent, et de meilleures alternatives permettant d'accélérer le processus sont donc souhaitables.

Par conséquent, dans ce chapitre, nous proposons un algorithme de modification rapide d'un graphe k-nn, qui est capable de mettre à jour un graphe k-nn en ajoutant ou en supprimant rapidement des nœuds. A notre connaissance, c'est le premier algorithme de ce type. De plus, l'algorithme peut être exécuté dans un environnement distribué, sans partage, pour traiter de très grands graphes distribués. Enfin, notre algorithme est indépendant de la mesure de similarité utilisée pour construire ou interroger le graphe.

11.1 Travail lié

Une étape importante de notre algorithme consiste à rechercher les plus proches voisins d'un nouveau point, en utilisant le graphe existant. C'est pourquoi nous présentons ici les algorithmes de recherche basés sur les graphes k-nn. Cette opération de recherche, lorsqu'elle est appliquée à un graphe distribué, nécessite un partitionnement spécifique du graphe. Par conséquent, nous présentons également les algorithmes de partitionnement de graphes.

11.1.1 Algorithmes de recherche

Dans [44], Hajebi et al. ont proposé un algorithme de recherche des plus proches voisins qui repose sur les graphes k-nn. Il s'agit d'un algorithme approximatif : les points (nœuds) trouvés par l'algorithme ne sont pas toujours les points les plus proches du point utilisé

pour la requête. Par contre, ils ont une forte probabilité d'être les points les plus proches. L'algorithme, appelé Graph Nearest neighbour Search (GNNS), fonctionne en sélectionnant les nœuds initiaux au hasard. Pour chaque nœud, l'algorithme calcule la similarité entre le point utilisé pour la requête et chaque voisin. Les voisins les plus similaires sont sélectionnés et l'algorithme itére jusqu'à ce qu'une profondeur de recherche d soit atteinte. Il s'agit donc d'un algorithme du type "hill climbing". Les nœuds les plus prometteurs sont recherchés en premier, en utilisant la similarité entre le point utilisé pour la requête et le nœud comme heuristique. Il a été testé sur différents ensembles de données. Sans tenir compte de la phase de construction du graphe, l'algorithme de recherche beaucoup plus rapide que la recherche linéaire ou qu'un algorithme de recherche utilisant un arbre KD (KD tree).

11.1.2 Partitionnement de graphes

Comme nous le verrons plus loin, l'exécution d'une recherche sur un graphe distribué nécessite un partitionnement spécifique des données, basé sur le clustering k-médoïdes. C'est pourquoi nous présentons ici les algorithmes de partitionnement de graphes existants.

La définition classique du partitionnement d'un graphe consiste à répartir les données entre plusieurs partitions de manière à minimiser le nombre de liens entre les partitions (le nombre de coupes), tout en maintenant le nombre de nœuds dans chaque partition à peu près égal. Plusieurs algorithmes existent pour effectuer ce type de partitionnement. Dans [83], les auteurs proposent un algorithme itératif distribué qui échange itérativement la partition de deux nœuds pour minimiser le nombre de coupes. L'algorithme est fortement basé sur MPI et nécessite beaucoup de communication entre tous les nœuds du graphe. Dans [19], les auteurs ont proposé et testé une version Bulk Synchronous Parallel (BSP) de l'algorithme qui le rend adapté aux architectures MapReduce ou Apache Spark. Dans [55], les auteurs ont proposé un algorithme de streaming, qui nécessite une seule itération pour partitionner le graphe. Ils ont comparé expérimentalement diverses heuristiques pour assigner des nœuds à une partition. Ils ont trouvé que l'heuristique la plus performante était l'heuristique déterministe linéaire pondérée gourmande. Celle-ci assigne chaque nœud à la partition où il a le plus de liens, pondéré par une fonction de pénalité linéaire basée sur la capacité de la partition.

Comme nous le montrons dans Section 11.4, pour améliorer la recherche des plus proches voisins basée sur des graphes distribués, le schéma de partitionnement doit minimiser le nombre de nœuds intermédiaires entre deux nœuds de la partition. Dans ce cas, le partitionnement devient un problème de clustering k-médoïdes. Il s'agit d'une variation de k-means clustering où les centres sont des points de l'ensemble de données. Elle minimise également la somme des distances entre les paires de points, tandis que k-means minimise la somme des distances au carré. Tout comme pour le clustering k-means, divers algorithmes ont été proposés dans la littérature pour effectuer du clustering k-médoïdes, comme Partitioning Around médoïdes (PAM) [95] et la méthode d'itération de Voronoi proposée dans [80]. Celle-ci est très similaire à l'algorithme classique de Lloyd's utilisé pour calculer les k-means. Cependant, ces algorithmes ne peuvent pas être exécutés dans un environnement distribué. De plus, aucune version équilibrée de k-médoïdes (qui maintnent le même nombre de points dans chaque partition) n'a été publiée jusqu'à présent, bien qu'il existe quelques versions équilibrées de k-means. Dans [69], les auteurs ont proposé une méthode qui a une complexité $O(n^3)$, ce qui la rend trop complexe pour les grands graphes. Dans [9], les auteurs ont

proposé la méthode FSCL (Frequency Sensitive Competitive Learning), où la distance entre un point et un centroïde est multipliée par le nombre de points déjà attribués à ce centre. Les plus grands clusters ont donc moins de chances de recevoir des points supplémentaires. Dans [7], les auteurs ont utilisé FSCL avec biais additif au lieu du biais multiplicatif. Cependant, les deux méthodes n'offrent aucune garantie sur le nombre final de points dans chaque partition, et les résultats expérimentaux ont montré que le partitionnement résultant est souvent largement déséquilibré. C'est pourquoi nous présentons ci-dessous notre propre algorithme, qui offre une garantie sur le nombre maximum d'éléments par cluster.

11.2 Ajouter des nœuds

L'algorithme que nous proposons comporte trois étapes principales pour ajouter un nouveau point au graphe : 1) utiliser le graphe actuel pour rechercher les k plus proches voisins du nouveau point, en utilisant l'algorithme distribué présenté à la section 11.4 ; 2) mettre à jour le graphe en suivant la procédure présentée en algorithme 6 et 3) assigner le nouveau point à l'un des nœuds de calcul.

La procédure de mise à jour est en fait un algorithme de propagation. Elle commence avec les voisins découverts du nouveau nœud, et explore récursivement les voisins des voisins, jusqu'à une profondeur fixe, pour vérifier si les liens existants doivent être modifiés. Finalement, le nouveau nœud est affecté au nœud de calcul correspondant au médoïde le plus similaire.

Une fois qu'un nombre donné de nouveaux nœuds ont été ajoutés au graphe, les médoïdes peuvent être recalculés. Ceci n'est en fait pas obligatoire et dépend de l'ensemble de données : si les caractéristiques de l'ensemble de données sont fixes dans le temps, l'ajout de nouveaux nœuds n'induira pas un déplacement des médoïdes. Sinon, le taux de mise à jour des médoïdes devrait correspondre à la vitesse de variation de l'ensemble de données. L'estimation automatique du taux d'actualisation est laissée comme travail futur.

Algorithm 6 Update

```
Inputs :
graph : the current graph
new : the new node to add in the graph
neighbours : the list of nodes to analyse
depth : the current depth
MAX_DEPTH : the maximum depth of exploration

for node in neighbours do
    if depth < MAX_DEPTH then                                  ▷ Recursion
        Update(graph, new, node.neighbours, depth++)
    end if
    compute similarity(node, new)
    if needed, add new to the neighbourlist of node
end for
```

Les étapes de calcul les plus intensives de l'algorithme sont les étapes de recherche et de mise à jour. Ce dernier nécessite de calculer au maximum $k^{\mathrm{DEPTH}+1}$ similarité. Afin de réduire

l'espace requis pour le graphe, k est généralement maintenu petit. Une valeur de 10 est très souvent observée. Avec cette valeur, l'évaluation expérimentale montre qu'une profondeur de trois est suffisante pour mettre à jour le graphe. Le nombre de calculs de similarité qui en résulte (1000) est donc faible par rapport à la taille des graphes visés par cet algorithme de mise à jour. Le coût de calcul de l'algorithme sera donc dominé par l'étape de recherche, d'où la nécessité d'un algorithme très efficace.

11.3 Suppression d'un nœud

Lorsqu'un nœud n_d est supprimé du graphe, d'autres nœuds qui ont n_d comme voisin doivent être mis à jour pour leur attribuer un nouveau voisin. Ces nœuds à mettre à jour sont facilement identifiables en scannant le graphe complet. Comme ces nœuds à mettre à jour avaient tous n_d comme voisin, ils sont très susceptibles d'être très similaires les uns aux autres. C'est pourquoi nous ne les traitons pas individuellement, mais en tant que groupe.

En effet, un ensemble commun de candidats est d'abord identifié à l'aide d'un algorithme de propagation distribué similaire à celui utilisé pour mettre à jour le graphe après l'ajout d'un nœud : le graphe est exploré jusqu'à une profondeur fixe, à partir de chaque nœud à mettre à jour et du nœud à supprimer.

Enfin, pour chaque nœud à mettre à jour, le nœud le plus similaire de l'ensemble des candidats est utilisé pour remplacer n_d.

Lors de la suppression d'un nœud, on peut s'attendre à ce qu'une moyenne de k nœuds doivent être mise à jour. Comme nous utilisons aussi le nœud pour supprimer n_d comme point de départ de l'algorithme de propagation, nous pouvons nous attendre à trouver $(k+1)^{\text{DEPTH}+1}$ candidats. L'algorithme devra donc calculer $k \cdot (k+1)^{\text{DEPTH}+1} \simeq k^{\text{DEPTH}+2}$ similarités entre nœuds. Une évaluation expérimentale a montré qu'une faible valeur DEPTH est suffisante pour obtenir de bons résultats. Cela représente une énorme amélioration de la vitesse par rapport à l'approche naïve qui exige de comparer les k nœuds à mettre à jour aux n nœuds du graphe et nécessiterait donc de calculer $O(k \cdot n)$ similarité .

11.4 Recherche distribuée des voisins les plus proches

Comme indiqué ci-dessus, le coût de calcul pour ajouter un nouveau nœud au graphe est principalement influencé par l'étape de recherche des plus proches voisins. A notre connaissance, il n'existe dans la littérature aucun algorithme permettant de rechercher rapidement les voisins les plus proches d'un point à l'aide d'un graphe distribué. Nous présentons donc ici notre propre algorithme, qui peut supporter toute mesure de similarité, même non métrique.

La procédure que nous proposons pour effectuer une recherche distribuée est en fait très simple : les p partitions sont recherchées indépendamment en utilisant un algorithme séquentiel efficace, puis les $k \cdot p$ candidats sont filtrés pour garder le k nœuds les plus proches du point utilisé pour la requête. Les données échangées sont très limitées : les nœuds de calcul envoient les $k \cdot p$ candidats voisins au maître, qui produit le résultat final.

La procédure que nous utilisons à l'intérieur de chaque partition pour rechercher les voisins les plus proches d'un point de requête q est inspirée de l'approche "hill climbing" présentée dans [44] : l'algorithme sélectionne un nœud aléatoire r dans le graphe, calcule la similarité entre q et chaque voisin de r, et itère avec le voisin le plus similaire. Pendant l'itération, il conserve un ensemble des points les plus similaires à q. Lorsqu'un maximum local est atteint, l'algorithme redémarre avec un autre nœud choisi au hasard. Cet algorithme de recherche est donc un algorithme approximatif, car il ne trouve pas nécessairement le nœud le plus similaire dans le graphe. Il trouve cependant le voisin le plus proche avec une probabilité élevée, tout en n'analysant qu'une fraction des nœuds du graphe.

Dans notre procédure de recherche séquentielle, nous introduisons deux approximations supplémentaires, qui permettent de réduire encore le nombre de similitudes à calculer.

La première amélioration repose sur l'observation que par la définition d'un graphe k-nn, chaque nœud possède des liens vers d'autres nœuds qui lui sont très similaires. Par conséquent, l'amélioration de la similarité à chaque itération de la recherche peut être très faible. Par conséquent, le nombre d'itérations i (le nombre de nœuds à analyser) avant de trouver les voisins les plus proches d'un point de requête peut être très important. Dans la pire configuration du graphe, $i = n/k$. Ceci nécessite de calculer beaucoup de similarités, $i \cdot k$. Pour éviter cette situation, le nœud de départ choisi au hasard r est directement éliminé s'il est situé trop loin du point de requête.

Formellement, nous gardons une trace de la similarité du voisin le plus similaire trouvé jusqu'ici $s_{\max}$, et nous introduisons un coefficient d'expansion $e > 1$. Comme indiqué ci-dessus, lorsqu'un maximum local est atteint, l'algorithme redémarre avec un autre nœud aléatoire r. Nous rejetons immédiatement r et sélectionnons un nouveau nœud aléatoire si $\text{similarity}(query, r) < s_{\max}/e$. De cette façon, nous éviton l'analyse d'une chaîne potentiellement très longue de i nœuds avant d'atteindre le voisinage du point de requête, ce qui serait très coûteux en calcul ($i \cdot k$). Au lieu de cela, nous nous concentrons sur l'exploration du voisinage du point de requête (les nœuds pour lesquels la similarité avec le point de requête est au moins $s_{\max}/e$).

Deuxièmement, pour réduire davantage le nombre de similarités calculées, pour chaque nœud analysé, nous choisissons directement le premier voisin qui fournit une augmentation de la similarité par rapport au nœud actuellement analysé. Nous essayons donc d'analyser seulement quelques-uns des k voisins du nœud actuel. L'amélioration fournie peut être calculée pour un espace euclidien de dimension d et pour des points uniformément répartis de façon aléatoire. Pour n'importe quel nœud, H est le nombre de voisins qui ont une plus grande similarité avec le point de requête, et L le nombre des voisins qui ont une plus petite similarité (et donc $k = H + L$). Dans un tel espace, observez que pour n'importe quel nœud, sur les 2^d directions, une seule mène dans la direction du point de requête. Par exemple, pour $d = 1$, ce sont deux directions possibles (gauche et droite), et une seule va dans la direction du point de requête. Comme chaque nœud a k voisins, en moyenne seulement $\mathbf{E}(H) = k/2^d$ les liens mènent à un nœud avec une plus grande similarité, et $\mathbf{E}(L) = k - k/2^d$ les liens mènent à un nœud avec une similarité plus faible.

Nous calculons maintenant $\mathbf{E}(S)$, le nombre attendu de similarités qui doivent être calculées pour chaque nœud. Comme l'algorithme amélioré itére dès qu'un nœud avec une plus grande similarité est trouvé, c'est en fait l'équivalent du problème du sac de billes : imaginez un

TABLE 11.1: Accélération de la recherche si l'on saute au nœud voisin dès qu'un voisin avec une plus grand similarité est trouvé, comparé à l'approche classique hill climb, pour différentes valeurs de k et différentes dimensionalités d.

k	4	10	10	10
d	2	2	3	4
$\mathbf{E}(H)$	1	2.5	1.25	0.625
$\mathbf{E}(L)$	3	7.5	8.75	9.375
$\mathbf{E}(S)$	1.5	2.14	3.88	5.77
$\mathbf{E}$(speedup)	2.66	4.66	2.57	1.73

sac avec w billes blanches et r billes rouges. Nous cherchons le nombre attendus de billes blanches à piocher avant de tomber sur une bille rouge. Chacune des w billes blanches peut être choisie au lieu d'une bille rouge avec une probabilité de $1/(r+1)$, et donc le nombre attendu de billes blanches à piocher est $w/(r+1)$. Pour $\mathbf{E}(S)$ cela donne :

$$\mathbf{E}(S) = \frac{\mathbf{E}(L)}{1+\mathbf{E}(H)} = \frac{k - k/2^d}{1 + k/2^d} \tag{11.1}$$

A l'inverse, l'approche originale hill climbing nécessite de calculer k similarités pour chaque nœud analysé. Il en résulte donc une accélération prévue de $k/\mathbf{E}(S)$ par rapport à l'approche originale. L'accélération résultante pour certaines valeurs de k et d est montrée dans le tableau 11.1, qui montre qu'une accélération substantielle peut être réalisée dans certains cas. L'inconvénient est que dans certains cas, l'algorithme peut choisir un voisin qui améliore la similarité, mais sans la maximiser (il y avait un autre voisin qui est plus similaire au point de requête). Cependant, un nœud n'a des liens que vers d'autres nœuds très similaires, donc ces voisins sont aussi similaires les uns aux autres. La différence d'amélioration de la similarité lors du choix d'un voisin sous-optimal reste donc limitée, et globalement l'efficacité de l'algorithme augmente.

Ces approximations supplémentaires nous permettent de réduire le nombre de similitudes calculées, tout en n'empêchant pas la convergence de l'approche hill climb. La démonstration de cette dernière peut être trouvée dans [44] et n'est pas répétée ici.

Afin d'accroître l'efficacité de la recherche distribuée, le graphe est également distribué (partitionné, ou partitioning) entre les différents nœuds de calculs de manière à maximiser la probabilité de trouver les voisins les plus proches. Cette distribution vise à remplir deux conditions : 1) la distance (mesurée comme le nombre de nœuds intermédiaires) entre deux nœuds d'un même sous-graphe doit être aussi faible que possible, afin de maximiser la probabilité de trouver rapidement de "bons" candidats et 2) le nombre de nœuds dans chaque sous-graphe doit être semblable pour équilibrer la charge de travail entre les nœuds de calcul pendant la recherche.

La première condition correspond à la définition du clustering k-médoïdes, une variation de k-means où les centres sont des points du jeu de données. Le clustering k-médoïds minimise également la somme des distances par paires, alors que k-means minimise la somme des distances au carré.

L'impact du partitionnement des données sur l'algorithme de recherche est en fait très

dépendant de la géométrie du graphe. Soit p_n le nombre de clusters que le graphe contient naturellement (le nombre de composants connectés). Si l'algorithme divise le graphe en $p = p_n$ partitions, le partitionnement utilisé par l'algorithme de recherche distribué reflétera le partitionnement naturel du graphe. Dans ce cas, il s'assurera que les points de départ utilisés par la recherche séquentielle seront bien répartis entre les différents clusters. On peut donc s'attendre à ce que l'algorithme distribué produise de meilleurs résultats que l'algorithme séquentiel.

Dans d'autres cas (lorsque $p \neq p_n$), les clusters du graphe seront répartis entre les p partitions. Le partitionnement risque donc de provoquer de nombreuses coupes (des liens entre des nœuds situés dans des partitions différentes). Statistiquement, cela raccourcira la chaîne de nœuds que la procédure de recherche séquentielle peut parcourir et obligera l'algorithme à redémarrer à partir d'un nœud aléatoire de la partition. Cela réduira finalement la probabilité de trouver les voisins les plus proches du point utilisé pour la requête.

11.5 Partitionnement équilibré d'un graphe k-nn en utilisant k-médoïdes

Nous présentons ici la procédure que nous utilisons pour partitionner le graphe k-nn distribué. Il s'inspire de la méthode d'itération de Voronoi utilisée par l'algorithme classique k-means et dans [80]. Il a l'avantage supplémentaire d'offrir une garantie sur la taille maximale de chaque partition, ce qui garantit une répartition équitable de la charge de travail entre les nœuds de calcul.

Dans notre cas, nous souhaitons découper le graphe afin d'optimiser la recherche distribuée k-nn. Par conséquent, la mesure de distance utilisée pour attribuer chaque nœud à un médoïde et pour calculer les nouveaux médoïdes devrait être la longueur du chemin le plus court dans le graphe entre deux nœuds. Cependant, pour calculer le chemin le plus court entre un nœud et tous les médoïdes, il faut avoir accès à toutes les neighborlists, ce qui est impossible dans une infrastructure distribuée et sans partage de mémoire. Par conséquent, au lieu de calculer le chemin le plus court vers chaque médoïde, nous utilisons la similarité entre le nœud et chaque médoïde comme une heuristique. Un exemple intuitif est le cas simple d'une distribution uniforme dans un espace euclidien. Dans un tel espace, tous les liens ont la même longueur. Par conséquent, la médoïde la plus semblable, mesurée comme le chemin le plus court entre le nœud et la médoïde, est aussi la médoïde la plus semblable mesurée en utilisant la distance euclidienne.

Pour obtenir une distribution équilibrée des points entre les k partitions (à ne pas confondre avec les k liens par nœud du graphe k-nn), nous utilisons une heuristique modifiée pour affecter chaque point à une partition. Soit $p_0 \dots p_i \dots p_n$ l'ensemble des nœuds du graphe et $m_0 \dots m_j \dots m_k$ l'ensemble des medoïdes. Chaque point p_i est attribué à la partition $P(p_i)$ correspondant à la médioïde la plus similaire pondérée par une fonction de pénalité $w(m_j, t)$:

$$P(p_i) = \arg\max_{m_j}(\text{similarity}(p_i, m_j) \cdot w(m_j, t))$$

Cette fonction de pénalité est basée sur la capacité et la taille actuelle de la partition, afin de pénaliser les grands clusters.

Dans un environnement distribué sans mémoire partagée, comme Spark, les différents nœuds de calcul n'ont pas accès à la taille totale de chaque partition à l'instant t. Chaque nœud de calcul ne peut s'appuyer que sur le nombre local de points déjà attribués à une partition $P_L(m_j, t)$.

Par conséquent, pour exécuter l'algorithme en parallèle, nous distribuons d'abord aléatoirement les nœuds entre les nœuds de calcul c. À chaque itération, cet ensemble de données mélangées est utilisé pour attribuer chaque nœud à l'aide d'une fonction de pondération qui repose uniquement sur la taille locale des partitions :

$$w_L(m_j, t) = 1 - \frac{|P_L(m_j, t)|}{capacity_L} \tag{11.2}$$

où $capacity_L$ est la contribution de chaque nœud de calcul à la taille totale de la partition finale :

$$capacity_L = \frac{n \cdot imbalance}{k \cdot c}$$

En fonction de l'application, une petite différence de taille peut être tolérée entre les partitions, donc la capacité des partitions est généralement calculée en utilisant un facteur de déséquilibre ($imbalance \geqslant 1$). A l'inverse, un partitionnement parfaitement équilibré peut être réalisé en utilisant $imbalance = 1$.

Si l'ensemble de données est suffisamment grand (par rapport à c) et distribué aléatoirement, on peut supposer que l'erreur résultante (nombre relatif de points qui ne sont pas attribués à la bonne grappe) peut être maintenue arbitrairement faible.

Si le partitionnement des points de données est parfaitement équilibré, lors de l'étape de mise à jour, la taille de chaque cluster est de n/k. Pour chaque partition, le calcul du nouveau médioïde nécessite le calcul de la similarité entre chaque paire de point appartement à la partition :

$$\frac{n}{k} \cdot \frac{\frac{n}{k} - 1}{2} \approx O(\frac{n^2}{k^2}) \tag{11.3}$$

La limite inférieure du coût total pour calculer les k nouveaux médoïdes est donc :

$$O(\frac{n^2}{k}) \tag{11.4}$$

C'est tout à fait inacceptable pour les grands ensembles de données. Par conséquent, nous échantillonnons l'ensemble de données et exécutons l'algorithme équilibré k-médoïdes par rapport aux données réduites pour calculer les médoïdes. Nous n'utilisons le jeu de données complet qu'une seule fois, pour assigner chaque nœud à un médoïde et à un nœud de calcul, en utilisant la contrainte de l'équation 11.2.

C'est une technique bien connue pour la famille des algorithmes k-means d'utiliser un ensemble de données échantillonnées pour calculer les centres initiaux. L'impact de cette

technique dépend naturellement de l'objectif final du clustering. Pour notre cas d'utilisation, l'évaluation expérimentale montre qu'il est parfaitement valide et offre les mêmes résultats que l'exécution de multiples itérations de k-médoïdes distribués en grappes avec l'ensemble complet des données, alors qu'il nécessite le calcul de beaucoup moins de similarités.

11.6 Evaluation expérimentale

Pour effectuer l'évaluation expérimentale, nous avons implémenté tous les algorithmes en utilisant le framework Spark. Toutes les expériences sont exécutées sur un cluster de calcul composé de 16 nœuds de calcul plus un nœud maître, chacun équipé d'un processeur quad-core et de 8 Go de mémoire RAM. Chaque expérience est répétée 10 fois, et nous présentons ci-dessous les valeurs moyennes.

Les expériences sont menées à l'aide de deux ensembles de données, avec des mesures de similarité différentes.

L'ensemble de données synthétiques se compose de points dans R^3 qui sont générés de façon aléatoire selon un mélange de distributions gaussiennes. La mesure de similarité utilisée pour construire et interroger les graphes est la distance euclidienne classique.

Le jeu de données SPAM contient environ 1 million de spams collectés par Symantec Research Labs en 2010. D'après notre expérience, la mesure de similarité la plus pertinente pour construire et utilisé le graphe construit à partir de cet ensemble de données est Jaro-Winkler. Cette mesure de similarité est similaire à la distance d'édition classique de Levenshtein, mais elle permet la substitution de caractères. De plus, la substitution de deux caractères proches est considérée comme moins importante que la substitution de deux caractères éloignés l'un de l'autre. Jaro-Winkler n'est pas une distance métrique car elle ne respecte pas l'inégalité du triangle. Cela confirme que notre algorithme peut également être utilisé avec des mesures de similarité non métriques.

Pour cette évluation expérimentale, nous étudions la qualité du graphe lorsque nous ajoutons des nœuds dans le graphe, nous analysons l'impact des différents paramètres (nombre de partitions, profondeur d'update, vitesse de recherche, paramètre d'expansion de la recherche, dimensionalité des données, k). Nous analysons aussi l'impact du partitionnement sur les résultats de l'algorithme de recherche et la scalabilité lorsque plus de données doivent être analysées. Enfin, nous analysons la qualité du graphe lorsque nous retirons des nœuds du graphe.

Les différents résultats montrent que nos différents algorithmes peuvent être utilisés avec de très grands ensembles de données et donnent des résultats très similaires à ceux produits par un algorithme de force brute, alors qu'ils nécessitent le calcul de beaucoup moins de similarités.

Chapitre 12

Partitionnement

Dans ce chapitre, nous étudions plus en profondeur le partitionnement des graphes k-nn : si le graphe est si grand qu'il ne peut être traité sur une seule machine dans un délais raisonnable, il doit être distribué entre plusieurs nœuds de calcul et traité en parallèle. Cependant, le partitionnement du graphe a un impact important sur la performance des algorithmes de traitement. C'était l'une de nos observations dans le chapitre précédent et [31].

Les algorithmes de partitionnement de graphes existants se concentrent principalement sur l'optimisation de deux métriques : la réduction du nombre de liens entre partitions et la production de partitions de même taille. L'impact de l'optimisation de ces métriques sur différents algorithmes de traitement des graphes a déjà été étudié dans des travaux antérieurs comme [75]. Cependant, il n'y a aucune garantie que l'optimisation de ces métriques produira effectivement la configuration optimale du graphe [47]. Dans le chapitre précédent par exemple, nous montrons que le partitionnement optimal d'un graphe k-nn afin de l'utiliser pour la recherche des plus proches voisins correspond en fait à la définition un clustering k-médoïdes des nœuds. Cela montre qu'une autre approche de partitionnement des graphes, basée sur le clustering k-médoïdes, est souhaitable.

Dans ce chapitre, nous nous concentrons d'abord sur le clustering k-médoïdes et proposons deux nouvelles procédures optimisées. Ensuite, dans la deuxième partie du chapitre, nous proposons une méthode utilisant le clustering k-médoïdes pour partitionner les graphes k-nn.

12.1 Travail lié

12.1.1 Partitionnement de graphes

Un algorithme de partitionnement de graphes divise un grand graphe en plusieurs sous-graphes. Du point de vue mathématique, il existe une vaste littérature consacrée au partitionnement de graphes, voir par exemple [14] pour une liste.

Du point de vue de l'informatique distribuée, nous sommes intéressés par le partitionnement équilibré de graphes. Ces algorithmes partitionnent un grand graphe en un nombre fixe de partitions et d'une manière qui permet à d'autres algorithmes de traitement de graphes

d'atteindre une meilleure performance : si le graphe est correctement partitionné, les algorithmes d'analyse atteignent une meilleure performance (vitesse ou précision) que si le graphe est mal partitionné. L'efficacité de ces algorithmes de partitionnement est généralement mesurée à l'aide de l'équilibre (balance) et du coût de communication :

$$\text{balance} = \frac{\text{taille de la partition la plus grande}}{\text{taille moyenne des partitions}}$$
$$\text{coût de communication} = \text{nombre de liens entre deux partitions différentes}$$

Deux approches existent pour partitionner un graphe : le partitionnement des nœuds (sommets, ou vertex) et le partitionnement des liens. Dans l'approche de partitionnement des sommets (aussi appelée partitionnement en coupe de lien ou edge cut), les sommets (nœuds) sont affectés à différentes partitions. Un lien (ou arête) est coupé si ses nœuds appartiennent à deux partitions différentes. Dans le cas d'un partitionnement des liens (ou vertex cut), les liens sont assignés aux partitions et les sommets (nœuds) sont coupés si leurs liens sont affectés à des partitions différentes.

Il a été démontré que le partitionnement des liens est une meilleure approche pour traiter les graphes avec une distribution des degrés obéissant à une loi de puissance (power law), qui sont courants dans les ensembles de données du monde réel [38, 4]. C'est pourquoi la plupart de la littérature récente se concentre sur ce domaine. Il existe une grande variété de ces algorithmes disponibles, comme Random Vertex Cut (RVC), Canonical Random Vertex Cut (CRVC), Ja-Be-Ja-VC [82] et Hybrid-Cut [21], et des implémentations existent pour la plupart des frameworks de traitement de graphes distribués comme Pregel, Giraph ou GraphX [100].

Dans le cas d'un graphe k-nn, le graphe est stocké en utilisant des listes d'adjacence (neighborlists) de taille fixe k, donc seuls les algorithmes de partitionnement à coupe de lien sont applicables. A notre connaissance, il n'existe actuellement que quelques-uns de ces algorithmes qui supportent l'exécution distribuée : EdgePartition1D, PT-Scotch, parMETIS, Multi-level Label Propagation (MLP) et Ja-Be-Ja.

EdgePartition1D [76] est en fait un algorithme de partitionnement des liens (vertex-cut). Il assigne chaque lien à une partition en utilisant une valeur de hachage calculée à partir de l'identifiant du nœud source du lien. Par conséquent, tous les liens sortants d'un nœud sont assignés à la même partition, ce qui permet de partitionner également les graphes k-nn.

PT-Scotch [22], parMETIS [52] et MLP [101] appartiennent à la même famille d'algorithmes de partitionnement de graphes multi-niveau. Ils réduisent d'abord la taille du graphe en rabattant les sommets et les liens. Ensuite, ils partitionnent le graphe le plus petit, et finalement, ils remettent en correspondance ce partitionnement avec le graphe d'origine. Ceux-ci s'appuient cependant sur le protocole MPI (Message Passing Interface) et supposent une communication rapide entre tous les nœuds du graphe. Pour ce chapitre, nous ciblons un modèle de programmation différent, à savoir le traitement parallèle synchrone en masse (BSP).

Enfin, l'algorithme Ja-Be-Ja [82] attribue initialement chaque nœud à une partition aléatoire, puis l'algorithme échange itérativement les nœuds entre les partitions pour augmenter le nombre de voisins qu'ils ont dans la même partition comme eux-mêmes. Une implémentation BSP de Ja-Be-Ja a également été proposée dans [19].

12.1.2 Clustering k-medoïdes

k-médoïdes est un algorithme de clustering qui est très similaire à k-means car ils tentent tous les deux de minimiser la distance entre les points étiquetés comme étant dans un cluster et un point désigné comme le centre du cluster. Cependant, il y a deux différences principales entre k-means et k-médoïdes : 1) dans k-médoïdes les centres sont des points réels de l'ensemble de données, appelés médoïdes et 2) k-médoïdes minimise la somme des différences par paires plutôt que la somme des distances au carré. Cela donne à k-médoïdes deux avantages par rapport à k-means : 1) il est plus robuste au bruit et aux valeurs aberrantes et 2) il peut être utilisé avec des ensembles de données régis par une mesure de distance non euclidienne.

Les quatre algorithmes classiques pour effectuer le clustering k-médoïdes sont PAM, CLARA, l'itération de Lloyd et CLARANS.

Clustering Large Applications based on RANdomized Search (CLARANS) [78] est actuellement considéré comme l'algorithme le plus efficace pour effectuer un clustering k-médoïdes. Il considère l'espace des solutions comme un graphe, où chaque nœud est un ensemble de médoïdes k et deux nœuds sont voisins s'ils diffèrent par un médoïde. Ainsi, à partir d'une solution actuelle, les solutions voisines $k(n-k)$ peuvent être construites en échangeant un médoïde avec un point aléatoire de l'ensemble de données. A partir d'une solution actuelle, CLARANS sélectionne au hasard une solution neigbor aléatoire, calcule son coût et passe à la solution voisine si elle a un coût inférieur. Sinon, il choisit un autre voisin au hasard, et ainsi de suite.

L'algorithme d'itération de Lloyd [80] utilise la même approche que l'algorithme de Lloyd utilisé pour k- signifie clustering : à chaque itération, l'algorithme recherche le point le plus central dans chaque cluster. Si tous les clusters ont la même taille (n/k), trouver le centre de chaque cluster nécessite de calculer au moins $O(n^2/k^2)$ les distances. Par conséquent, chaque itération nécessite de calculer des distances $O(n^2/k)$, ce qui rend l'algorithme assez lent.

Dans [89] et [23], les auteurs utilisent une technique basée sur un recuit simulé pour effectuer le clustering k-médoïdes. Dans ces articles, les auteurs appliquent les critères d'acceptation du recuit simulé après que l'algorithme CLARANS ait calculé le coût de la solution voisine, pour échapper à un éventuel minimum local. D'après notre expérience, les minimums locaux sont relativement rares lors de l'utilisation de CLARANS, donc l'utilisation du recuit simulé de cette façon ne fait que ralentir la convergence de l'algorithme. Dans la section **??** nous proposons une autre approche, où nous appliquons les critères de décision avant de calculer le coût de la solution voisine. Cela permet d'éviter de calculer le coût de la solution actuelle, qui est une opération coûteuse.

12.2 Temps de convergence de CLARANS

Nous définissons une itération de CLARANS comme le processus nécessaire pour améliorer la solution actuelle. Il s'agit donc de trouver une nouvelle solution possédant un coût inférieur.

Un essai est défini comme le choix d'une solution candidate et le calcul de son coût. En

utilisant l'algorithm standard, ceci nécessite de calculer kn similarités. Ceci peut être réduit à n si une mise en cache appropriée est utilisée.

Considérons le cas simple d'un ensemble de données uniforme $\mathbf{D}$ dans $\mathbb{Z}$ avec $k = 2$ et distance unitaire entre points, comme illustré sur la Figure 12.1. L'espace des solutions est l'ensemble de (x, y) de $\mathbf{D}. \times \mathbf{D}$ où $x \neq y$. Il forme une grille en $\mathbb{Z}^2$.

L'ensemble des coûts de chaque solution candidate forme une grille. Elle est symétrique car les solutions (x, y) et (y, x) sont équivalentes, et elle n'a que deux maxima équivalents : (x_o, y_o) et (y_o, x_o).

Considérons la grille formée par l'espace des solutions. A n'importe quelle itération, $k(n-k) = O(nk)$ solutions candidates peuvent être construites à partir de la solution actuelle.

Nous calculons maintenant une limite supérieure pour le nombre d'essais requis par CLARANS à chaque itération. La progression de CLARANS peut être mesurée en utilisant d, la distance de Manhattan entre la solution actuelle et la solution optimale dans l'espace de solution, comme le montre la Figure 12.1. Plus formellement, d est défini comme suit : $o_0 \dots o_i \dots o_k$ est la solution optimale et $m_0 \dots m_i \dots m_k$ est la solution actuelle (l'ensemble des médoïdes à toute itération de l'algorithme). Alors :

$$d = \sum_{i=1}^{k} \text{distance}(m_i, o_i) \tag{12.1}$$

Par la définition de CLARANS, à chaque itération, cette distance diminuera d'au moins 1 (la distance entre les points du jeu de données).

Considérer toute itération de l'algorithme où $d > k$. La pire configuration pour la convergence de l'algorithme est lorsque la solution candidate est située en diagonale par rapport à la solution optimale, comme dans la Figure 12.1. Cela signifie qu'à partir de la solution actuelle, tous les médicaments doivent être échangés avant d'atteindre la solution optimale. Dans cette configuration, dans chaque direction (pour chaque médoïde) il y a des points $2d/k - 1$ qui mènent à une solution améliorée, comme présentée dans la Figure 12.1. Comme il y a des directions k (médoïdes), le nombre de solutions voisines qui ont un coût inférieur (en rouge) est $(2d/k - 1)k$, et le nombre total de solutions voisines est $O(nk)$. Par conséquent, cette itération nécessitera en moyenne des essais $O(nk/(2d - k))$ pour progresser.

A n'importe quelle itération, et pour n'importe quelle valeur de d (même si $d \leq k$), la meilleure configuration est obtenue lorsque le candidat et la solution optimale diffèrent par un seul medoid. Dans cette configuration, il existe des solutions à $2-1$ qui réduiront le coût par rapport à la solution actuelle. Le coût pour compléter l'itération est donc de $o(nk/2d)$ en moyenne.

Remarquez que dans les deux cas, **le nombre moyen d'essais requis par CLARANS pour compléter l'itération, et donc pour progresser vers la solution optimale, est proportionnel à** nk.

Supposons, par exemple, que nous ayons une solution où $d = 1$ (juste à côté de la solution optimale), une seule solution améliorera la similarité totale par rapport à la solution actuelle (la solution optimale). Par conséquent, CLARANS aura besoin d'essais jusqu'à nk pour trouver celui-ci, ce qui nécessitera de calculer les similarités n^2k^2.

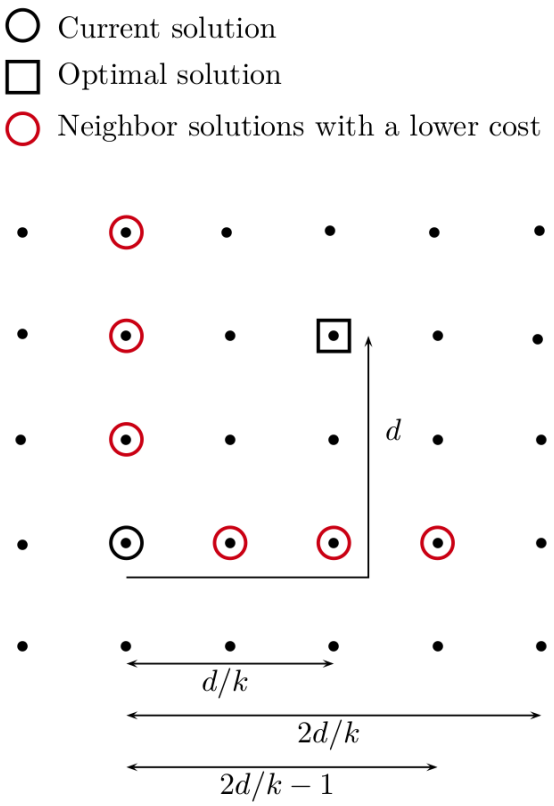

FIGURE 12.1: Convergence of CLARANS

Cela montre que **CLARANS est en fait lent à converger car la solution voisine est choisie au hasard**.

12.3 Clustering k-médoïdes

La lenteur du taux de convergence de CLARANS est due au fait qu'il essaie beaucoup de solutions candidates avant d'en trouver une qui soit réellement meilleure, et le calcul du coût d'une solution est une opération lourde qui nécessite de calculer des similarités nk. Cependant, beaucoup de solutions candidates peuvent être rejetées rapidement et à peu de frais. En effet, rappelons que CLARANS construit une nouvelle solution candidate en échangeant un médoïde de la solution courante avec un point aléatoire du jeu de données. D'après l'équation, on peut le déduire :

$$\forall i \ : \ \text{distance}(m_i, o_i) \leq d \tag{12.2}$$

En d'autres termes, la distance entre un médoïde actuel et le médoïde optimal est d'au plus d. Cela permet de rejeter rapidement les solutions candidates qui n'amélioreraient pas la solution actuelle.

Pour toute configuration, le nombre de solutions voisines qui obéissent à cette condition est dk. Par conséquent, le nombre moyen d'essais requis pour progresser est réduit à $O(dk/(2d - k)$

pour la pire configuration et $o(dk/2d) = o(k/2)$ pour la meilleure configuration. Remarquez que cette fois **le nombre d'essais n'est pas proportionnel à** n. Cela montre que nous pouvons épargner beaucoup d'essais et de calculs si nous considérons seulement les solutions voisines situées à distance $\leq d$ de la solution actuelle.

Malheureusement, il n'existe pas de méthode facile pour calculer d car nous ne savons pas à l'avance où se trouve la solution optimale. C'est pourquoi nous proposons deux procédures pour effectuer un clustering k-médoïdes efficace qui **éliminent les solutions voisines qui sont probablement trop éloignées de la solution actuelle**, sans avoir besoin de la valeur de d.

La première est inspirée de la simulation de recuit (simulated annealing). Dans l'évaluation expérimentale, nous montrons qu'avec des paramètres corrects, elle converge plus rapidement que CLARANS. Cependant, comme tous les algorithmes de recuit simulés, l'ajustement des paramètres est très difficile. Nous présentons également une autre approche qui repose sur une heuristique simple à calculer qui ne nécessite aucun paramètre de configuration. D'après nos expériences, il est plus performant que l'état de l'art dans toutes nos évaluations.

12.3.1 Recuit simulé

Notre approche pour calculer le clustering k-médoïdes est basée sur la recherche locale mais, au lieu de sélectionner une solution aléatoire comme CLARANS, nous utilisons un algorithme inspiré de la simulation de recuit (simulated annealing) pour sélectionner le voisin. Dans le graphe des solutions, nous essayons de rejeter les solutions qui ne remplissent pas l'équation 12.2.

Pour construire une solution voisine, nous sélectionnons un médoïde aléatoire dans la solution actuelle et un point aléatoire dans l'ensemble de données. La distance entre ces deux points est la mesure de l'énergie qui doit être minimisée lorsque l'algorithme converge. On utilise donc une température initiale T_0 qui diminuera à chaque itération selon un facteur de refroidissement γ, et la probabilité d'accepter ce point aléatoire est calculée selon la formule de Metropolis et Kirkpatrick [53] : $P = e^{-distance/T}$.

Algorithm 7 Générateur de solution inspiré par le recuit simulé

Input : current solution S
$m \leftarrow$ a random medoid from S
$p \leftarrow$ a random point from the dataset
$d \leftarrow distance(m, p)$
$T \leftarrow T_0 \cdot \gamma^{iteration}$
swap m and p with probability $e^{-d/T}$

Lors des premières itérations, la température est élevée, c'est pourquoi nous acceptons des voisins situés à une grande distance de la solution actuelle. Au fur et à mesure que l'algorithme progresse vers la solution optimale, la température diminue de telle sorte que nous ne générons que des voisins situés à proximité de la solution actuelle.

Une fois que la solution du voisin est générée comme décrit dans l'algorithme 7, nous continuons avec la procédure classique de recherche locale : l'algorithme calcule le coût de la solution du voisin, qui est acceptée si elle a un coût inférieur. L'avantage par rapport

aux CLARANS classiques est que nos générateurs de solutions voisines permettent de se débarrasser des solutions voisines trop éloignées de la solution actuelle dans l'espace solution, avant de calculer leur coût, évitant ainsi tout calcul inutile.

Contrairement aux algorithmes présentés dans [89] et [23], nous appliquons un recuit simulé avant et non après calcul du coût de la solution voisine. En effet, nous essayons de réduire le nombre de similitudes calculées et d'accélérer l'algorithme en réduisant l'espace de recherche, alors que les deux documents précités utilisent un recuit simulé pour échapper à un éventuel minimum local, après calcul du coût de la solution.

Cependant, comme toutes les méthodes basées sur le recuit simulé, notre méthode n'est efficace que si des valeurs correctes pour T_0 et γ sont utilisées, et qu'il n'existe aucune approche connue pour estimer ces valeurs en fonction des caractéristiques spécifiques du jeu de données. C'est pourquoi nous proposons également un autre générateur de solution voisin, basé sur une heuristique simple qui ne nécessite aucun paramètre d'entrée.

12.3.2 Heuristique

Pour expliquer l'idée derrière le générateur de voisin heuristique, nous considérons la situation illustrée dans Figure reffig :clarans. Nous considérons tous les voisins de la solution courante le long de l'axe horizontal (toutes les solutions qui peuvent être construites en échangeant un médoïde avec un point aléatoire du jeu de données). En haut de Figure 12.2, nous traçons le coût de chaque solution voisine. Les voisins avec un coût négatif sont ceux qui permettent à l'algorithme de progresser, mais ils n'ont pas encore été découverts (sinon l'algorithme utiliserait l'un d'eux comme solution actuelle). Ils sont tous situés à une distance de $\leq 2d/k$ de la solution actuelle.

En bas de Figure 12.2 nous traçons le coût des voisins découverts en fonction de la distance avec la solution actuelle. En rouge, nous traçons le coût moyen des solutions situées à la même distance. Cette placette a un minimum local (identifié par un carré rouge) qui correspond également à la distance maximale pour laquelle il existe des solutions qui réduiront le coût de la solution actuelle ($2d/k$).

C'est l'approche utilisée dans le générateur de voisin basé sur l'heuristique : nous suivons la distance et le coût de chaque solution candidate évaluée et recherchons un minimum local dans la courbe résultante. La distance correspondant à ce minimum local est utilisée comme distance maximale pour essayer une solution candidate.

12.3.3 Évaluation expérimentale

Nous effectuons maintenant une évaluation expérimentale de nos algorithmes. Tous les tests sont exécutés sur un cluster Spark composé d'un maître et de 16 ouvriers. Chaque test est exécuté vingt fois et nous montrons le résultat moyen. Nous utilisons différents ensembles de données pour effectuer les tests :

— **synthetic-3-12** est un ensemble de données composé de 100.000 points en R^3 répartis dans 12 centres selon une loi gaussienne ;

— **synthetic-10-12** est un ensemble de données de 100.000 points dans R^{10} distribué

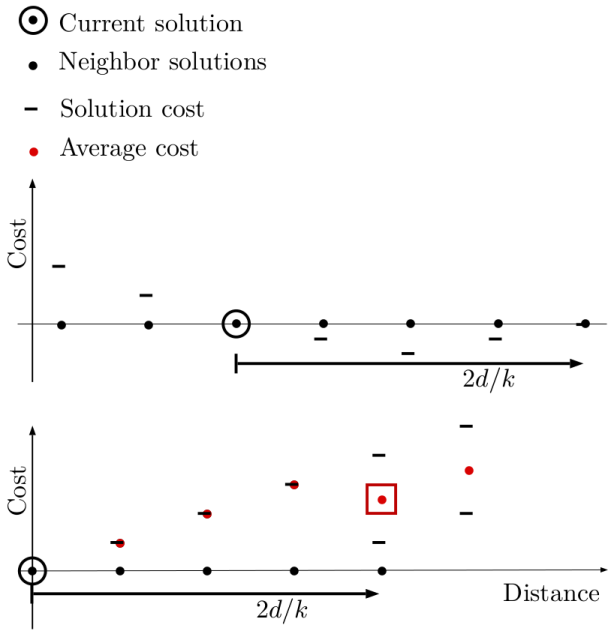

FIGURE 12.2: Heuristic based neighbor generator

autour de 12 centres ;

— **synthetic-3-100** est un ensemble de données de 100.000 points en R^3 distribué autour de 100 centres ;

— **commercials** est un ensemble de données du Référentiel d'apprentissage machine UCI[61] qui contient des fonctionnalités extraites de 129.685 séquences TV ;

— **spam** est un ensemble de données composé du sujet de 100.000 spams. La similarité entre les spams est calculée en utilisant la similarité des chaînes Jaro-Winkler, qui n'appartient pas à un espace métrique.

Il existe de nombreuses formules pour mesurer la qualité d'un clustering, comme l'indice de Davies-Bouldin, l'indice de Dunn ou le coefficient de Silhouette. Cependant, par définition, l'objectif du clustering k-médoïdes est de maximiser la similarité totale entre chaque point de l'ensemble de données et sa grappe associée. Par conséquent, en fonction de la configuration géométrique des points de l'ensemble de données, un clustering avec une similarité totale plus élevée peut se traduire par un coefficient de Silhouette inférieur, par exemple. Pour ce chapitre, nous voulons savoir si la solution trouvée par les différents algorithmes correspond bien à la définition du clustering k-médoïdes. Nous utilisons donc la similitude totale pour mesurer la qualité du clustering.

Pour cette étude expérimentale, nous effectuons d'abord une étude des paramètres afin de trouver les valeurs optimales pour l'approche basée sur le recuit simulé. Ensuite nous comparons la performance de nos deux algorithmes à l'état de l'art (CLARANS).

Les résultats montrent clairement que l'approche heuristique fonctionne au moins aussi bien que les CLARANS classiques. Dans certains cas, l'algorithme nécessite de calculer deux fois moins de similarités que CLARANS pour obtenir la même qualité de clustering. Le recuit simulé est encore plus performant, mais seulement si les paramètres de configuration (T_0

et γ) sont précisément ajustés pour l'ensemble de données à traiter et pour le nombre de similitudes qui peuvent être calculées pendant l'analyse (1E9 dans nos tests).

12.4 Partitionnement de graphe basé sur le clustering k-médoïdes

Dans cette section, nous proposons d'utiliser le clustering k-médoïdes pour partitionner un grand graphe k-nn entre plusieurs nœuds de calcul. Par conséquent, nous calculons d'abord le clustering k-médoïdes des nœuds en utilisant la méthode optimisée présentée ci-dessus, puis nous assignons chaque nœud et sa liste de contiguïté correspondante à la partition correspondant au médoïde le plus similaire.

Ensuite, nous comparons notre approche aux algorithmes existants de deux façons différentes. Tout d'abord, nous utilisons les métriques classiques utilisées pour mesurer la performance des algorithmes de partitionnement : l'équilibre et le coût de communication. Ensuite, nous comparons la performance de l'algorithme de recherche distribuée que nous avons présenté dans Section 11.4 en utilisant des graphes partitionnés avec les différents algorithmes.

12.4.1 Évaluation expérimentale

Nous comparons ici expérimentalement l'implémentation Spark de notre partitionnement de graphes basé sur k-médoïdes avec Ja-Be-Ja et Edge1D [1]. Par souci d'équité, l'algorithme Ja-Be-Ja met en œuvre les optimisations proposées dans [19] et [75]. Nous construisons d'abord le graphe 10-nn correspondant à chaque ensemble de données. Ensuite, nous partitionnons le graphe entre nos 16 travailleurs, en laissant chaque algorithme de partitionnement fonctionner pendant une durée fixe, et enfin nous calculons le nombre de fronts de partitionnement croisés (coût de communication) et le déséquilibre du partitionnement résultant. Pour le partitionnement k-médoïdes, nous utilisons un paramètre de déséquilibre cible de 1,2. Comme Edge1D est un algorithme à passage unique, nous le laissons simplement tourner jusqu'à ce qu'il soit terminé, ce qui prend environ 3 secondes sur notre cluster. Les résultats sont présentés sur la figure 12.3.

Comme on pouvait s'y attendre, la qualité du partitionnement effectué par Edge1D est cohérente avec une distribution aléatoire des nœuds. Nous pouvons voir que l'approche k-médoïdes parvient à trouver un bon partitionnement dès les toutes premières itérations, puis continue à améliorer lentement le partitionnement. Au contraire, Ja-Be-Ja commence par un partitionnement aléatoire, puis améliore très lentement la solution, en ne permutant que quelques centaines de nœuds à chaque itération. Sur les chiffres, nous montrons également la qualité moyenne de partitionnement obtenue par Ja-Be-Ja après 1 heure de fonctionnement. Nous pouvons voir que même après 1h de calcul, Ja-Be-Ja n'est même pas proche de la qualité du partitionnement basé sur k-médoïdes.

1. `https://github.com/tdebatty/spark-knn-graphs`

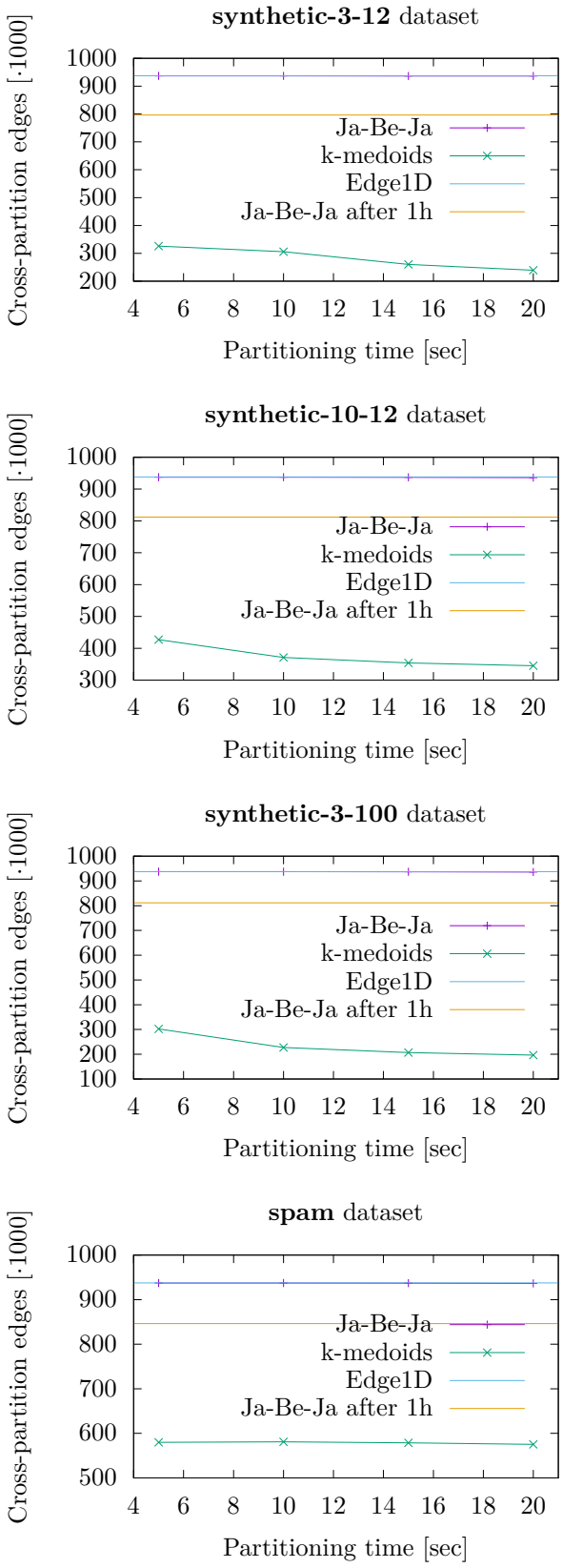

FIGURE 12.3: Quality of partitioning with different datasets (lower is better)

12.4.2 Impact du partitionnement sur la recherche distribuée

Nous revenons maintenant à notre problème initial : comment partitionner un grand graphe k-nn de manière à pouvoir exécuter efficacement d'autres algorithmes de traitement, comme la recherche distribuée ? Nous avons montré précédemment que ce type d'algorithmes nécessite que le graphe soit partitionné selon un cluster k-médoïdes.

Nous effectuons ici une évaluation expérimentale pour montrer qu'une approche basée sur k-médoïdes permet effectivement à l'algorithme de recherche d'obtenir de meilleurs résultats. Nous utilisons les différents algorithmes pour partitionner le graphe, et cette fois nous mesurons la précision de l'algorithme de recherche distribué. Nous avons d'abord divisé chaque ensemble de données entre un ensemble de formation et un ensemble de requêtes. Nous construisons le graphe à partir de l'ensemble d'apprentissage et nous le partitionnons avec les différents algorithmes, puis nous utilisons l'ensemble de requêtes pour effectuer une recherche distribuée avec l'algorithme présenté dans Section 11.4.

Cet algorithme de recherche suppose que chaque partition contient un sous-graphe du graphe distribué. Chaque sous-graphe fait l'objet d'une recherche indépendante à l'aide d'une approche ascensionnelle. Lorsque la recherche atteint une limite du sous-graphe, elle redémarre simplement à partir d'un nœud aléatoire. Enfin, la meilleure solution de toutes les partitions est renvoyée. Cet algorithme présente les avantages suivants : 1) il ne nécessite qu'une itération et 2) il ne nécessite pratiquement aucune communication.

Sur la figure 12.4 nous montrons la proportion de résultats de recherche corrects pour les différents algorithmes de partitionnement.

Les résultats montrent clairement que l'approche k-médoïdes surpasse largement les autres algorithmes de partitionnement. On peut aussi remarquer que les résultats sont moins bons pour les ensembles de données **spam** et **synthetic-10-12** que pour les autres. Pour le jeu de données **spam**, cela est dû au caractère non euclidien du jeu de données. Pour l'ensemble de données **synthetic-10-12**, cela est dû à la dimensionnalité supérieure de l'ensemble de données. Comme nous l'avons également observé dans [31], ces deux configurations rendent la recherche plus difficile.

12.5 Conclusions et travaux futurs

Dans ce chapitre, nous avons étudié l'utilisation du clustering k-médoïdes pour partitionner de grands graphes k-nn. Nous avons proposé deux nouvelles procédures optimisées pour la mise en grappe k-médoïdes. Puis nous avons proposé une méthode s'appuyant sur le clustering k-médoïdes pour partitionner les grands graphes k-nn. Notre évaluation expérimentale a montré que 1) nos procédures de clustering surpassent l'état actuel de la technique et 2) le clustering k-nn est une excellente approche pour partitionner de grands graphes k-nn car il est en même temps plus rapide et plus efficace que les algorithmes de partitioning actuels.

Comme travail futur, nous prévoyons d'évaluer la qualité du partitionnement k-médoïdes lorsque d'autres algorithmes sont exécutés sur le graphe partitionné, comme Connected Components.

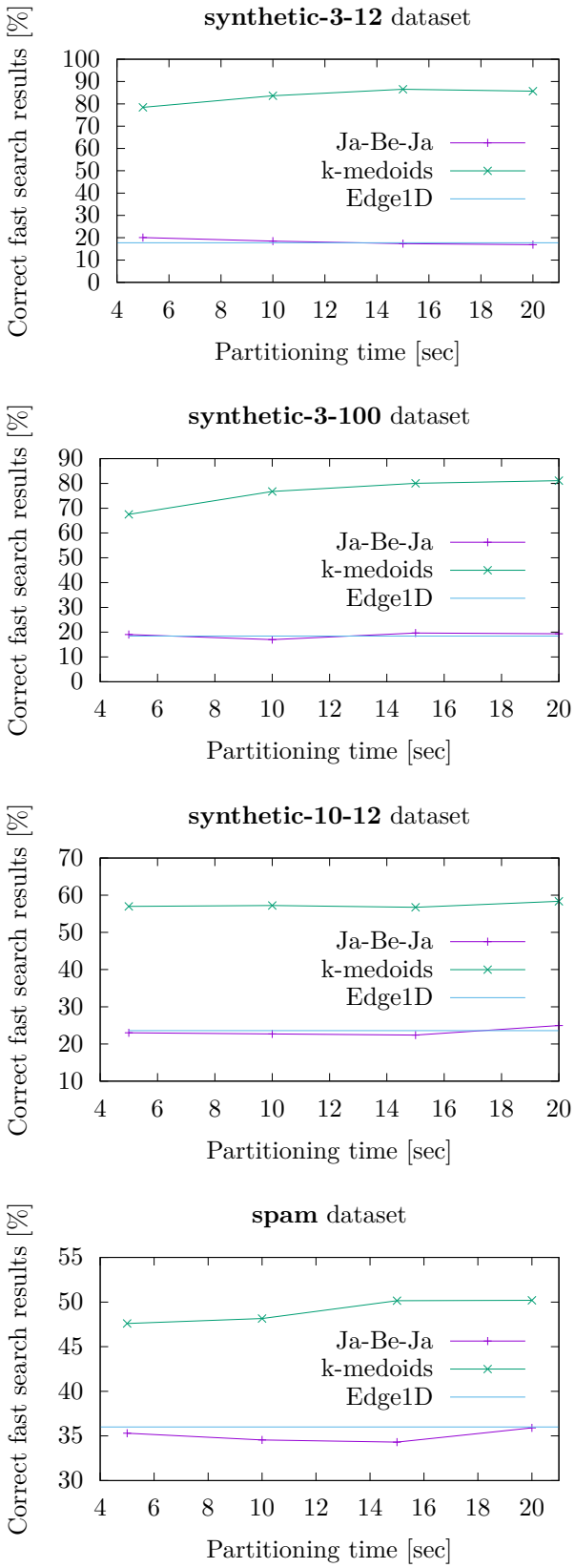

FIGURE 12.4: Effet du partitionnement de graphes sur les résultats de l'algorithme de recherche rapide distribué

Chapitre 13

Clustering de données textuelles

Le clustering de données textuelles est une tâche d'analyse fondamentale qui consiste à trouver des groupes d'éléments semblables. La définition de la similarité utilisée pour effectuer le clustering est généralement déterminée par le domaine d'application concerné. De même, la mesure de la qualité du clustering obtenu en appliquant l'une des nombreuses techniques de clustering existantes dépend souvent du domaine d'application. Un clustering considéré comme bon pour une application peut ne pas convenir aux exigences d'une autre. Cela nécessite souvent une inspection manuelle par des experts du domaine.

Dans ce chapitre, nous nous concentrons sur une tâche particulière de clustering de données textuelles. Cet algorithme : *i)* produit des clusters de haute qualité qui sont faciles à interpréter, *ii)* prend en charge toutes sortes de fonctions de similarité, même non métriques, et *iii)* supporte les grands ensembles de données. L'essentiel de notre algorithme de clustering consiste à construire un graphe k-nn *approximatif* des données textuelles reçues, et à calculer ses composants connectés, qui identifient les clusters de données.

En particulier, nous visons à comprendre le compromis qui existe entre la précision, l'évolutivité et, en fin de compte, la qualité du clustering. Pour ce faire, nous procédons à une évaluation expérimentale approfondie de notre méthode de clustering avec des ensembles de données réelles et à grande échelle, en utilisant notre implémentation - qui est disponible en ligne - pour le framework Apache Spark.

13.1 Travail lié

Le problème de la clusterings de données a été largement étudié dans la littérature informatique en général, avec des nuances allant de la théorie des graphes et des principes du data mining [39, 6] aux approches expérimentales [93, 107]. Étant donné qu'il n'entre pas dans le cadre du présent document de rendre justice à tous les travaux réalisés dans ce domaine, nous nous concentrons ici sur les approches qui se rapprochent le plus de notre travail.

L'un des algorithmes de clustering les plus populaires est l'algorithme d'itération de Lloyd [64], qui effectue le clustering des k-moyennes (k-means). Il s'agit d'une approche simple qui fonctionne sur des vecteurs d-dimensionnels. Il peut aussi être utilisé pour faire du clustering

de texte, mais cela nécessite une *phase de transformation* pour encoder les phrases et les mots en vecteurs [51, 97]. Par exemple, Mikolov et al. [73] présentent une implémentation efficace des architectures continues de poches de mots et de sauts d'images pour le calcul des représentations vectorielles des mots. Dans notre travail, nous utilisons cette approche comme référence à laquelle nous comparons notre algorithme de clustering. Dans l'évaluation expérimentale, nous montrons qu'il souffre de son incapacité à accepter des mesures de distance non métriques, qui sont essentielles pour détecter les similitudes entre les phrases mutilées, et de sa faible évolutivité.

Les approches alternatives recherchent des termes fréquents dans l'ensemble de données pour identifier les clusters [11, 10, 72] : essentiellement, l'idée est de trouver des sous-ensembles de termes fréquents, qui sont un proxy des clusters, et de mapper les éléments de données contenant des éléments de ces sous-ensembles sur le même cluster. De telles approches souffrent de leur incapacité à bien s'adapter (dans certains cas, les clusters peuvent se chevaucher, et une phase de désambiguïsation basée sur des heuristiques gourmandes est nécessaire, ce qui est difficile à adapter), et ne tiennent pas compte de la similitude du texte qui est résistante à la confusion.

D'autres approches visent à optimiser le calcul de la similarité par paires entre les éléments de texte à l'aide de calculs matriciels [77]. Dans cette catégorie, Lin et al. ont présenté un algorithme optimisé pour récupérer le clustering de données textuelles à partir d'une matrice de similarité en utilisant la similarité cosinus. Encore une fois, de telles approches ne tiennent pas compte des mesures de similarité non métriques et sont difficiles à mettre à l'échelle, bien que des travaux récents [16] aient montré les avantages des opérations matricielles approximatives qui mettent à l'échelle mieux que les calculs exacts de similarité toutes paires.

Une approche qui cible des objectifs similaires aux nôtres est le Triage [96], qui aborde le même domaine d'application que nous ciblons dans ce document. Cependant, le triage est axé sur les éléments de données à fonctions multiples, et non sur l'extensibilité : les auteurs abordent principalement les problèmes liés à la fusion de l'information, en définissant une méthode pour fusionner plusieurs mesures de distance différentes fonctionnant sur des valeurs textuelles, catégorielles et numériques. Récemment, une version parallèle de Triage a été proposée [92], qui traite en partie des questions d'évolutivité. Cependant, l'approche calcule toujours la similarité de toutes les paires parmi les éléments prototypes représentatifs, avec une complexité $O\left(n^2\right)$ qui rend toujours difficile le traitement de très grands ensembles de données.

13.2 Clustering basé sur les graphes k-nn

Nous présentons maintenant notre approche pour la clustering de texte évolutive et nous fournissons une implémentation parallèle. Nous soulignons également le rôle important joué par l'accentuation dans notre algorithme. Outre la conception de l'algorithme lui-même, l'utilisation de techniques d'approximation constitue une contribution importante à notre analyse de l'arbitrage qui existe entre la qualité du clustering et l'extensibilité de notre méthode.

Le problème du clustering de textes que nous considérons est particulièrement difficile en

raison du scénario d'application que nous étudions. Nous sommes confrontés à un contexte conflictuel dans lequel les données textuelles sont générées de telle sorte qu'il est difficile de trouver des éléments similaires : Les campagnes de SPAM utilisent des erreurs de texte, des fautes d'orthographe et généralement des variations sur certains textes de base, ce qui fait que les éléments SPAM appartenant à la même campagne semblent différents les uns des autres. Par conséquent, nous devons utiliser une mesure de similarité entre les éléments qui peuvent surmonter, ou du moins atténuer, le problème.

Il existe de nombreux algorithmes pour mesurer la similitude entre les éléments. En particulier, pour les données textuelles, la distance de Hamming et la distance de Levenshtein ont été largement utilisées.

Dans ce travail, notre expérience a montre que la mesure de similarité la plus adaptée à notre ensemble de données est la distance de Jaro-Winkler [49, 104] qui, simplement dit, compte les caractères communs entre deux chaînes même si elles si elles sont mal orthographiées. Jaro-Winkler n'est cependant pas une distance métrique car elle n'obéit pas à l'inégalité du triangle.

Étant donné le choix de la mesure de similarité que nous utilisons dans ce travail, le large éventail de techniques pour trouver des clusters de textes similaires se réduit à quelques méthodes.

13.2.1 Construction du graphe k-nn

Dans la première phase de notre méthode, nous construisons un graphe k-nn. Dans ce travail, nous limitons notre attention aux éléments textuels : par exemple, nous extrayons l'objet d'un courriel SPAM comme étant l'unique élément représentatif de l'élément. La prise en compte de caractéristiques supplémentaires ou hétérogènes n'entre pas dans le cadre de ce travail, et nous nous en remettons à une extension de notre approche.

L'approche naïve pour construire un graphe k-nn consiste à trouver la similarité de toutes les paires entre tous les éléments d'un ensemble de données, puis sélectionner les éléments k les plus similaires à chaque élément. De toute évidence, cette approche de " force brute " n'est pas évolutive, car elle nécessite des calculs de similarité $O(n^2)$, où n est le nombre d'éléments dans l'ensemble de données. Notez que l'algorithme de " force brute " produit des graphes *exact* k-nn, que nous utilisons dans ce travail comme référence pour déterminer la qualité approximative de notre méthode.

Au lieu de cela, nous utilisons une implémentation Spark de NN-Descent, similaire à l'implémentation Hadoop MapReduce que nous avons déjà présentée dans Chapter 10. Rappelez-vous que cet algorithme utilise une approche itérative pour affiner progressivement le graphe k-nn et éventuellement construire une approximation du graphe k-nn exact.

Dans ce travail, nous nous intéressons particulièrement au rôle du nombre d'itérations de l'algorithme, qui détermine sa qualité approximative. Nous prétendons que même des approximations approximatives du graphe k-nn sont suffisantes pour atteindre l'objectif ultime de notre méthode de clustering. Intuitivement, l'existence d'un chemin entre des éléments similaires sur le graphe k-nn est suffisante pour la dernière phase de l'algorithme de clustering que nous proposons.

13.2.2 Elagage du graphe k-nn

La procédure itérative pour construire un graphe k-nn approximatif peut induire des relations de voisinage entre des éléments de texte qui ont une faible similarité de paires. En effet, l'algorithme produit nécessairement les voisins les plus similaires k pour chaque élément : toute asymétrie dans la distribution des similarités par paires peut produire un graphe k-nn dans lequel certains nœuds ne sont que "approximativement" similaires. On utilise donc une *phase d'élagage*, avec le paramètre $\theta \in [0, 1]$ pour déterminer une valeur de similarité de coupure, en dessous de laquelle les liens entre deux nœuds sont éliminés. La phase d'élagage inspecte chaque nœud du graphe k-nn, et élague ces liens.

Choisir un seuil approprié θ détermine le résultat final de notre méthode de clustering : comme $\theta \to 1$ clustering is *strict*, qui conduit à un grand nombre de petits clusters de items essentiellement identiques ; comme $\theta \to 0$ clustering is *loose*, menant au cas dégénéré d'un cluster unique, géant.

13.2.3 Composants connectés

La deuxième et dernière phase de notre approche utilise le graphe taillé k-nn, et produit ses composantes connectées, que nous utilisons comme proxy pour identifier des grappes d'éléments similaires. Rappelons que le problème de trouver les composantes reliées d'un graphe revient à rechercher des sous-graphies dans lesquelles deux sommets sont reliés l'un à l'autre par des chemins.

Trouver des composants connectés dans des graphes à grande échelle utilisant des algorithmes évolutifs est un problème bien étudié et compris, la littérature abondante sur le sujet en témoigne [87, 54, 67].

Dans ce travail, nous utilisons une implémentation parallèle de l'algorithme de Cracker [67], dans lequel chaque nœud est marqué avec le plus petit identifiant de nœud de son composant appelé identifiant de nœud de graine. L'idée clé de Cracker est de réduire la taille du graphe à chaque étape du calcul, en utilisant une technique inspirée de l'"algorithme de contraction" pour calculer les coupes minimales d'un graphe [50]. À la terminaison, tous les nœuds étiquetés avec le même identificateur de semence sont regroupés dans le même composant.

13.3 Evaluation expérimentale

Cette section fournit des détails sur notre configuration expérimentale, y compris les ensembles de données utilisés, les mesures d'évaluation, les paramètres et l'environnement système.

Toutes les expériences ont été menées sur un cluster composé de 17 nœuds exécutant Ubuntu Linux (1 maître et 16 esclaves), chacun équipé de 12 Go de RAM, d'un CPU 4 cœurs et d'une interconnexion 1 Gbit.

Nous avons implémenté notre approche et la méthode de base que nous utilisons pour notre analyse comparative en utilisant Apache Spark [8] et notre code source est disponible

publiquement.

Nous étudions le rôle des paramètres de notre approche - à savoir le nombre d'itérations utilisées pour construire le graphe k-nn et le seuil d'élagage θ - en utilisant les paramètres suivants :

— **Nombre de clusters** : mesure le nombre de clusters identifiés par l'algorithme de clustering. Sauf indication contraire, nous considérons que les clusters ne sont "utiles" que s'ils ont plus de 1 000 éléments ;

— **Taille du plus grand cluster** : mesure la taille du plus grand cluster identifié par l'algorithme.

Nous calculons la qualité de clustering en utilisant des métriques bien connues [35, 63] :

— **Compactness** mesure à quel point les éléments d'un cluster sont étroitement liés. Nous obtenons la compacité en calculant la similarité moyenne par paire entre les éléments de chaque groupe. Des valeurs plus élevées sont préférables.

— **Séparation** mesure à quel point les clusters sont séparés les uns des autres. La séparation est obtenue en calculant la similarité moyenne entre les items de différents groupes. Des valeurs inférieures sont préférables.

— **Silhouette** [88] constitue une métrique agrégée, qui tient compte de la similitude entre les éléments et à l'intérieur d'un même groupe. Des valeurs plus élevées sont préférables.

— **Recall** relie deux clusters de données obtenus par des méthodes différentes. En utilisant la classification C comme référence, nous calculons le rappel de la classification D en calculant la fraction des éléments qui appartiennent à la même classification dans C et D. En particulier, nous utilisons comme référence le clustering exact que nous obtenons avec l'approche " force brute " pour calculer le graphe k-nn. Des valeurs de rappel plus élevées sont préférables.

Il est important de noter que le calcul des mesures ci-dessus est aussi difficile que le calcul de la classification que nous avons l'intention d'évaluer. Pour cette raison, nous avons recours à l'échantillonnage uniforme : au lieu de calculer la similarité de tous les items par paires, nous choisissons les items uniformément au hasard, avec un taux d'échantillonnage de 1%. [1]

Le principal ensemble de données que nous utilisons dans notre évaluation consiste en un sous-ensemble de courriels de pourriels recueillis par Symantec Research Labs, entre le 2010-10-01 et le 2012-01-02, qui est composé d'échantillons de courriel de 3886371. Chaque élément de l'ensemble de données est formaté selon JSON et contient les caractéristiques communes d'un e-mail, telles que : l'objet, la date d'envoi, les informations géographes, le bot-net utilisé pour la campagne anti-spam tel qu'étiqueté par Symantec systems, et bien plus. Par exemple, l'objet d'un e-mail dans l'ensemble de données est “19.12.2011 Rolex For You -85%” et le jour d'envoi est “2011-12-19”.

Dans ce travail, nous nous intéressons à l'identification de groupes de courriels SPAM en utilisant uniquement des sujets, car ils constituent une description compacte de l'email.

Pour valider notre approche sur un ensemble de données différent, nous utilisons également

1. Nous augmentons le taux d'échantillonnage à 10% pour les petits groupes.

des données obtenues à l'aide de l'API Twitter, et consistant en des tweets de $1,530,623$ en format JSON.

Tout d'abord, nous analysons l'impact des différents paramètres de notre algorithme sur les mesures de qualité que nous avons définies ci-dessus. Ensuite, nous nous concentrons sur la qualité du clustering, et comparons la performance de notre approche à celle de l'algorithme *baseline* dont nous parlons dans la section 13.1. Enfin, nous étudions l'évolutivité de notre algorithme lorsque la quantité de données à analyser augmente.

Les différents tests montrent que notre algorithme produit des clusters de haute qualité, même avec un graphe k-nn approximatif, tout en nécessitant beaucoup moins de temps de calcul que k-means (jusqu'à 25x plus rapide pour nos données). De plus, la partie la plus coûteuse en calcul de notre algorithme est le calcul du graphe. Ceci permet de tester rapidement plusieurs valeurs pour l'élagage. Finalement l'accélération de notre algorithme par rapport à k-means ne fait qu'augmenter lorsque la quantité de données à analyser augmente.

Chapitre 14

Détection d'APT

Les menaces persistantes avancées (Adevanced Persistent Threats, APT) sont des attaques ciblées, perpétrées sur une longue période par des attaquants talentueux et généralement financés par des organisations puissantes.

De telles attaques sont de plus en plus sophistiquées et parviennent à contourner les meilleures protections disponibles dans le commerce. Les attaquants réussissent régulièrement à prendre à distance le contrôle d'ordinateurs situés à l'intérieur de nos réseaux, à garder ce contrôle suffisamment longtemps pour localiser l'information qu'ils recherchent, pour y accéder, et finalement exfiltrer des données sensibles.

Tous les APT ont quelques caractéristiques en commun. Premièrement, ils utilisent des techniques avancées comme les attaques 0-day et l'ingénierie sociale (social engineering) pour infecter l'organisation cible. Cela les rend impossibles à détecter à l'aide d'outils de détection classiques, qui utilisent des signatures, comme les antivirus ou les systèmes de détection d'intrusion (IDS).

Deuxièmement, une fois qu'un ordinateur est infecté, un APT essaiera d'établir un canal de communication avec un serveur de commandement et de contrôle (C2) à l'extérieur de l'organisation. Ce canal sera utilisé pour télécharger une charge utile, pour télécharger d'autres instructions ou pour exfiltrer des données. Ce lien peut être établi à l'aide de n'importe quel protocole autorisé sur le réseau de l'organisation : HTTP, DNS, SMTP, ou même SIP.

Cependant, dans une organisation soucieuse de la sécurité, tous ces protocoles devraient passer par une sorte de point de contrôle. Pour le protocole HTTP, il s'agirait d'un serveur proxy installé dans la zone démilitarisée (DMZ). Cela permet d'enregistrer toutes les connexions se déroulant avec des serveurs situés à l'extérieur de l'organisation, et offre la possibilité de détecter l'activité de l'APT.

Dans ce chapitre, nous nous concentrons sur la détection des APT qui utilisent le protocole HTTP pour établir un canal de communication avec leur serveur C2. Les APT naïfs effectuent des connexions à leur serveur C2 à des intervalles de temps fixes. Il en résulte des traces dans les journaux de proxy qui sont assez faciles à détecter. Les APT les plus évolués sont cependant capables de détecter l'activité des utilisateurs et d'attendre du trafic généré par l'activité de l'utilisateur pour se connecter aux serveurs C2. Cela les rend beaucoup plus

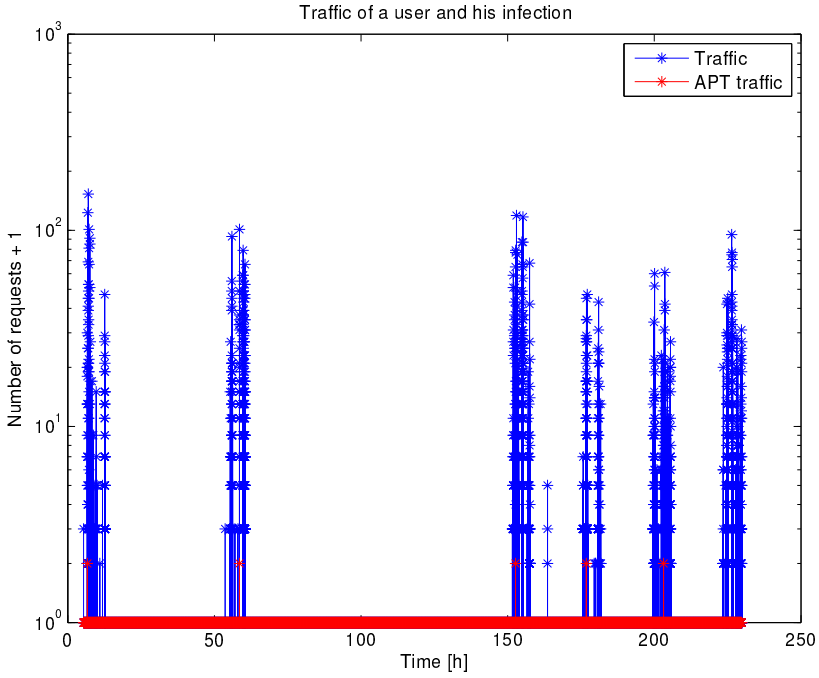

FIGURE 14.1: Traffic HTTP généré par un seul ordinateur infecté par une APT.

difficiles à détecter, comme sur la figure 14.1.

Dans ce chapitre, nous nous concentrons sur la détection de ces derniers, les APT à détection d'activité. Pour ce faire, nous construisons un graphe du trafic HTTP. Dans ce graphe, l'APT devient une anomalie détectable.

14.1 Modélisation du trafic HTTP

14.1.1 Modélisation du trafic légitime

Lorsqu'un utilisateur navigue sur Internet, chaque page demandée par le navigateur déclenche plusieurs requêtes HTTP. Habituellement, la première requête contient du code HTML, puis le navigateur télécharge javAUCript, CSS, fichiers de polices et images référencées dans le code. Ceux-ci peuvent en outre déclencher le téléchargement d'autres fichiers, ou déclencher des requêtes AJAX, et ainsi de suite. Chaque page visitée par un client peut ainsi être représentée par une arborescence, dont la racine est la page d'origine demandée par l'utilisateur, et chaque demande ultérieure possède un seul lien (un lien) vers la demande qui a déclenché ce téléchargement. Ces arbres peuvent être construits à partir des journaux du serveur proxy. Ceci est représenté sur la Figure 14.2. L'utilisateur a successivement ouvert trois pages (rose, vert et bleu), ce qui a déclenché un certain nombre de requêtes qui peuvent être observées dans les journaux du serveur proxy.

Dans la vie réelle, reconstruire le graphe HTTP est beaucoup plus compliqué. Tout d'abord, les requêtes appartenant à plusieurs pages ont souvent lieu au même moment. Cela peut

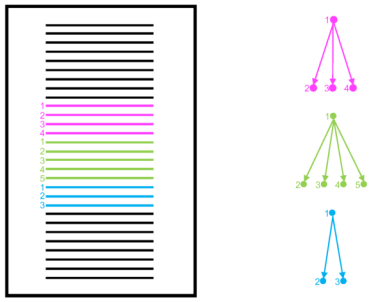

FIGURE 14.2: Graphe du traffic HTTP reconstruit à partir des journaux d'un serveur proxy.

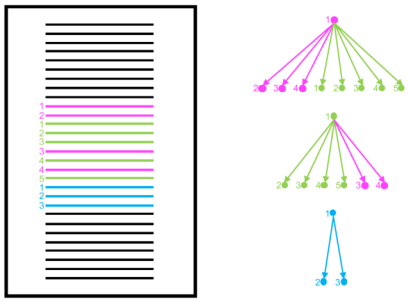

FIGURE 14.3: Graphe du traffic HTTP reconstruit à partir des journaux d'un serveur proxy lorsque plusieurs pages sont téléchargées en parallèle.

conduire à la construction d'arbres incorrects. Ceci est illustré dans la Figure 14.3.

Deuxièmement, comme la plupart des pages sur Internet sont construites dynamiquement, le chargement d'une même page plusieurs fois entraîne généralement l'exécution d'une série de requêtes différentes, bien que similaires.

Enfin, les navigateurs essaient de garder en mémoire cache le contenu qui n'est pas censé changer, comme les images, les fichiers javAUCript et css. Ceci modifie également les requêtes exécutées pour afficher la même page.

Par conséquent, nous construisons un graphe pondéré, un graphe où chaque nœud peut avoir des liens (liens) vers plusieurs autres nœuds, et chaque lien a un poids. Dans notre graphe, le poids du lien de la demande B à la demande A indique la probabilité que la demande B soit une conséquence de la demande A. La façon dont nous calculons exactement ces probabilités est expliquée ci-dessous. Par conséquent, une demande (B) peut avoir des limites par rapport à plusieurs autres demandes, chacune indiquant la probabilité que la demande A soit une conséquence de chaque autre demande.

14.1.2 Détection de l'APT

Lorsqu'un APT attend que le trafic légitime (causé par un utilisateur naviguant sur Internet) contacte son serveur C2, les requêtes effectuées par l'APT ont lieu approximativement au même moment que les autres requêtes du graphe. Par conséquent, ces demandes présentent des faiblesses par rapport à de multiples autres demandes.

Ceci est illustré dans la Figure 14.4 : les requêtes effectuées par l'APT se produisent avec les

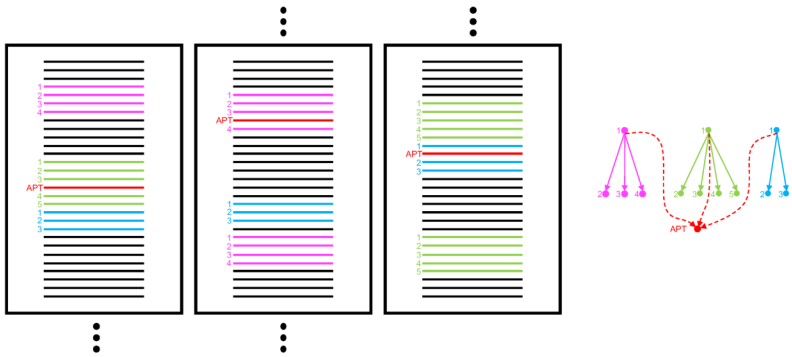

FIGURE 14.4: Graphe du traffic HTTP reconstruit à partir des journaux d'un serveur proxy lorsque l'ordinateur est infecté par une APT.

requêtes causées par la page rose, la page verte et la page bleue. En conséquence, la requête de l'APT présente des faiblesses par rapport aux trois demandes-racines.

Un moyen simple de détecter ces demandes est donc d'élaguer le graphe, c'est-à-dire de couper tous les liens dont la valeur est inférieure à un seuil. Les requêtes APT apparaissent alors comme des nœuds isolés, tandis que les autres requêtes légitimes sont reliées entre elles en clusters.

14.2 Implémentation

Pour tester l'algorithme, nous avons mis en place un système de détection complet. Le système offre une interface web qui permet à l'analyste d'analyser de façon interactive les requêtes se déroulant sur le réseau à l'aide de son navigateur. La détection repose sur de multiples paramètres que l'analyste peut facilement modifier pour repérer les APT cachés.

Le système se compose de deux éléments : 1) le processeur batch et 2) l'interface web qui permet une analyse interactive des données.

14.2.1 Traitement en batch

Le système est conçu pour permettre l'analyse du trafic HTTP généré par de grands réseaux d'ordinateurs. Dans ces réseaux, des millions de demandes sont générées chaque jour. Le stockage d'un graphe complet (entièrement connecté) nécessite de stocker des valeurs $O(n^2)$, où n est le nombre de nœuds dans le graphe. Ainsi, le stockage d'un graphe ne contenant que 1 million de requêtes nécessite environ 1 To de stockage. Avec le matériel actuel, c'est lourd à stocker sur disque et presque impossible à stocker en mémoire à moins d'utiliser un supercalculateur. Par conséquent, nous construisons plutôt un graphe k-voisins les plus proches (k-nn), un graphe où chaque nœud a un avantage sur les k autres nœuds les plus similaires dans le graphe. Ces graphes sont une approximation étroite d'un graphe complet si k est assez grand. Cependant, ils ont l'énorme avantage que leur besoin en mémoire est linéaire en n (à savoir kn), ce qui les rend aussi plus rapides à traiter [65].

Avant de stocker le graphe k-nn, il doit être calculé, ce qui nécessite de calculer les $O(n^2)$

similarités en utilisant un algorithme naïf. Cela n'est pas acceptable non plus pour les grands graphes que nous envisageons. Nous utilisons donc à la place un algorithme approximatif rapide appelé nn-descent qui nécessite de calculer seulement $O(n^{1.14})$ similarités pour construire le graphe k-nn [37]. De plus, ces algorithmes peuvent facilement être implémentés en parallèle, ce qui permet de profiter de tous les cœurs disponibles sur le serveur de traitement.

Même en utilisant ces deux optimisations (graphes k-nn et algorithme de nn-descente), la construction du graphe est un processus de calcul lourd qui nécessite un temps non négligeable. Le système est donc divisé en deux composants distincts : un processeur batch et une interface web pour effectuer l'analyse.

Le processeur batch est responsable du calcul du graphe initial, sans appliquer aucune détection. Comme son nom l'indique, ce traitement fastidieux ne doit être exécuté qu'une seule fois pour chaque ensemble de données.

Dans une implémentation naïve, modifier la définition de similarité, même légèrement, nécessite de recalculer le graphe k-nn complet. Encore une fois, il s'agit d'une opération qui prend beaucoup de temps et qui ne permettrait pas d'analyser les données de façon interactive. Au lieu de cela, pendant le traitement en batch, nous calculons plusieurs graphes : un pour chaque similarité élémentaire. La fusion des différents graphes est faite de façon interactive.

Le processeur batch ne nécessite qu'un seul paramètre, k, le nombre de liens par nœud dans le graphe k-nn.

14.2.2 Analyse interactive

Une fois que le processeur batch a calculé les graphes k-nn, ceux-ci peuvent être analysés de manière interactive à l'aide de l'interface Web. L'analyse nécessite principalement de : 1) fusionner les différents graphes k-nn, 2) supprimer les liens faibles (taille), 3) regrouper le graphe, 4) filtrer le graphe pour ne montrer que les domaines isolés et 5) classer les domaines suspects restants.

Dans ce processus, l'opérateur peut fournir plusieurs paramètres pour améliorer la détection.

Premièrement, il peut choisir quels **clients** du réseau sont analysés. La fusion des graphes correspondant à plusieurs clients renforce les limites entre les domaines naturellement liés. Cela rend les domaines contactés par l'APT plus isolés et donc plus faciles à détecter.

L'analyste peut également modifier la définition de **similarié** utilisée pour lier les requêtes en fournissant un poids différent pour la similarité temporelle et pour la similarité de domaine.

Il peut fournir un **seuil d'élagage** pour supprimer les liens faibles du graphe final. Un seuil élevé supprime beaucoup de liens dans le graphe, ce qui laisse beaucoup de requêtes isolées. Cela entraîne un plus grand nombre de faux positifs. A l'inverse, une valeur inférieure laisse presque tous les liens intacts. Par conséquent, les demandes générées par l'APT ont une probabilité plus élevée de rester en contact avec d'autres demandes, ce qui diminue la probabilité de détection.

Pour l'étape de filtrage, l'analyste peut fournir un **taille maximale du cluster**. Théori-

quement, après l'étape de taille, l'APT est censé être complètement isolé. Cependant, si le seuil d'élagage est choisi un peu trop bas, l'APT peut rester connecté à d'autres domaines, créant ainsi un petit cluster. En utilisant un filtre pour n'afficher que les petits groupes, l'analyste peut être en mesure de repérer l'APT.

Pour filtrer davantage les résultats, l'analyste peut spécifier un **nombre minimum de requêtes par domaine**. Lors de notre évaluation expérimentale avec des données réelles, nous avons découvert que le trafic HTTP régulier contient beaucoup de domaines qui ont très peu de requêtes chacun. Parce qu'ils ont très peu de requêtes, ils sont faiblement connectés à d'autres domaines, et sont donc considérés comme suspects par notre système. Un APT doit cependant contacter régulièrement son serveur C2 pour télécharger d'autres instructions ou pour exfiltrer des données. C'est pourquoi nous donnons à l'analyste la possibilité de filtrer les domaines avec très peu de requêtes.

Enfin, le système effectue un **ranking** des domaines suspects restants. Nous utilisons donc trois paramètres, avec des pondérations fournies par l'analyste : 1) le nombre de requêtes vers ce domaine, car un APT furtif est supposé n'exécuter que quelques requêtes, 2) le nombre de domaines enfants dans le graphe et 3) le nombre ou les domaines parents dans le graphe, car le domaine utilisé par un APT est censé être faiblement connecté à plusieurs autres domaines.

Bien que cela puisse sembler lourd, le traitement des graphes k-nn est en fait extrêmement rapide. Cela permet d'analyser interactivement d'énormes quantités de données.

14.3 Evaluation expérimentale

Pour tester le système, nous utilisons les logs du serveur proxy d'une grande et réelle organisation. A partir de cet ensemble de données, nous avons choisi un sous-réseau composé de 26 ordinateurs et d'une période de 10 jours. Ce sous-ensemble contient un total de requêtes 721 921.

Dans ce fichier journal, des requêtes sont insérées pour simuler l'activité de 4 APT dans le réseau. Ces APT sont choisis pour présenter différents comportements typiques. L'APT le moins furtif génère 239 requêtes vers son serveur C2 alors que l'APT le plus furtif n'effectue que 13 requêtes au total, ce qui le rend très difficile à détecter.

Pour tester la qualité de la détection, nous construisons la courbe caractéristique de fonctionnement du récepteur (Receiver Operating Curbe - ROC) du système de détection : nous générons un classement des requêtes en fonction de leur suspicion à l'aide des paramètres fournis. Ensuite, nous parcourons la liste en partant du haut, et nous comparons chaque demande à la liste connue des demandes effectuées par les APT. A chaque niveau de la liste, nous calculons la probabilité de détection (p_d) et la probabilité de fausse alarme (p_{fa}) :

$$p_d = \frac{\text{nombre de domaines correspondant à une APT}}{\text{nombre total de domaines présents dans la liste}}$$

$$p_{fa} = \frac{\text{nombre de domaines légitimes}}{\text{nombre total de domaines dans la liste}}$$

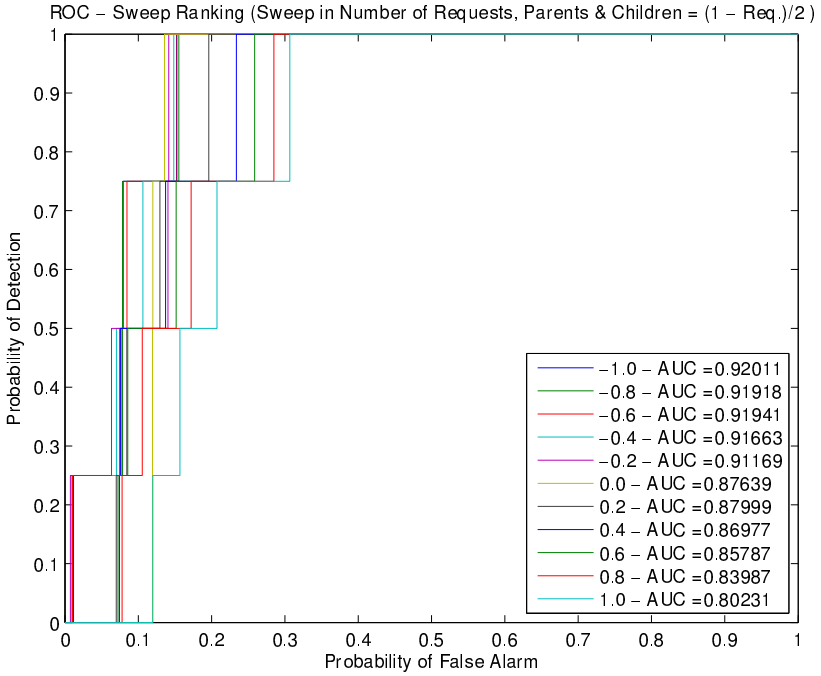

FIGURE 14.5: Résultats obtenus avec différentes valeurs pour les poids utilisés pour la classification

Enfin, pour calculer la qualité de la détection, nous calculons l'aire sous la courbe (Area Under the Curve - AUC). En effet, pour un système de détection parfait $p_d = 1 \forall p_{fa}$, donc AUC = 1. Au contraire, le système de détection le plus mauvais a $p_d = 0 \forall p_{fa}$, donc AUC = 0.

Nous avons successivement fait varier le nombre de voisins par nœud k, les poids utilisés pour la fusion, le seuil utilisé pour l'élagage du graphe, la taille maximale des clusters, le nombre minimal de requêtes par client, les poids utilisés pour le classement. Les résultats obtenus avec les valeurs optimales sur sont affichés dans la Figure 14.5.

L'utilisation d'une valeur négative pour le nombre de demandes semble donner les meilleurs résultats. Cependant, cela éclaire d'abord les APT moins furtifs, qui effectuent beaucoup de requêtes vers leurs serveurs C2. Nous recommandons donc d'utiliser ici une valeur légèrement positive, soit 0,2.

Nous avons maintenant identifié les paramètres qui semblent offrir la meilleure qualité de détection. Ces paramètres doivent bien sûr être réglés par l'analyste pour le réseau testé, et pour le type d'APT qu'il tente de détecter sur le réseau (l'APT est-il censé être plus ou moins furtif, etc.). Toutefois, nous supposons qu'ils sont suffisamment robustes pour représenter un point de départ pertinent pour l'analyste.

Pour tester cette hypothèse, nous utilisons un autre sous-réseau du journal de proxy. Cette fois, le sous-réseau contient 66 clients, 10 jours de données et 1 032 021 requêtes. Nous simulons l'infection avec 8 APT et nous analysons les données avec les paramètres optimaux identifiés précédemment.

La ROC résultante possède une AUC de 0,9036. Cela montre que l'algorithme est très résilient et qu'il fonctionne aussi bien avec un ensemble de données différent. Il montre également que le système découvre rapidement 90% des APT. Cela équivaut à la performance de la plupart des antivirus sur le marché.

Comme nous l'avons indiqué plus haut, l'ensemble de données utilisé pour effectuer les tests est produit à partir des journaux du serveur proxy d'un réseau réel. Nous avons donc analysé manuellement les domaines qui ont été classés comme APT par le système. Il s'agit principalement de :

— Réseaux de diffusion de contenu (CDN) ;

— Domaines d'articles qui affichent de la publicité sur plusieurs sites Web ;

— Domaines d'éléments qui fournissent des bibliothèques JavAUCript à plusieurs sites Web ;

— Sites Web d'articles avec très peu de visites.

Bien qu'il ne s'agisse pas de serveurs C2 en soi, ils se caractérisent par le même comportement que les APT que nous essayons de détecter : ils ont des liens faibles avec beaucoup d'autres domaines. Cela montre que l'algorithme fonctionne très bien pour détecter les domaines qui se comportent comme les domaines utilisés par un APT pour contacter son serveur C2.

Les résultats obtenus par notre algorithme de détection sont très prometteurs car ils ne reposent pas sur une connaissance préalable des attaques. Dans le cadre de travaux futurs, nous prévoyons d'affiner davantage les paramètres de détection, par exemple en utilisant l'apprentissage approfondi, afin d'améliorer encore la qualité de la détection.

Chapitre 15

Conclusions et perspectives

15.1 Conclusions

Dans cette thèse, nous avons étudié l'utilisation des graphes k-nn pour analyser de grands ensembles de données.

Dans le chapitre 10 nous avons présenté NNCTPH, un algorithme MapReduce qui construit un graphe k-nn approximatif à partir de grands ensembles de données textuelles. Nous avons utilisé des ensembles de données contenant le sujet de spam pour tester expérimentalement l'influence des différents paramètres de l'algorithme sur le temps de traitement et sur la qualité du graphe final. Nous avons également comparé l'algorithme avec une implémentation séquentielle et MapReduce de NN-Descent. Pour nos ensembles de données, l'algorithme s'est avéré jusqu'à dix fois plus rapide que l'implémentation MapReduce de NN-Descent pour la même qualité de graphe produit. De plus, l'accélération augmentait avec la taille de l'ensemble de données, ce qui fait de NNCTPH un choix parfait pour les très grands ensembles de données textuelles.

Au chapitre 11 nous avons proposé un algorithme capable de mettre à jour un graphe k-nn distribué en ajoutant ou en supprimant rapidement des nœuds. Nous avons effectué une évaluation expérimentale qui montre que l'algorithme peut être utilisé avec de très grands ensembles de données et produit des graphes qui sont très similaires aux graphes produits par un algorithme de force brute, alors qu'il nécessite le calcul de beaucoup moins de similarités.

Dans le chapitre 12 nous avons étudié l'utilisation du clustering (clustering) k-medoïdes pour partitionner de grands graphes k-nn. Nous avons également proposé deux nouvelles procédures optimisées pour le clustering k-medoïdes. Puis nous avons proposé une méthode s'appuyant sur le clustering k-medoïdes pour partitionner les grands graphes k-nn. Notre évaluation expérimentale a montré que 1) nos procédures de clustering surpassent les meilleurs algorithmes actuels et 2) le clustering k-medoïdes est une excellente approche pour partitionner de grands graphes k-nn car il est en même temps plus rapide et plus efficace que les algorithmes de partitionnement classiques.

Dans le chapitre 13 nous avons présenté une approche évolutive pour le clustering de données textuelles qui s'appuie sur les graphes k-nn. Cette nouvelle méthode supporte n'importe

quelle mesure de similarité et produit des clusters de grande qualité. Pour surmonter la nature quadratique des approches typiques du clustering de textes, nous avons étudié le rôle de l'approximation dans l'établissement d'un compromis entre une qualité de clustering élevée et une exécution rapide de l'algorithme. Nous avons montré, par le biais d'une campagne expérimentale détaillée, que notre méthode n'exige pas un calcul exact de la similarité entre chaque paire de points pour produire un clustering interprétable et de haute qualité.

Enfin, dans le chapitre 14 nous avons proposé un système de détection des attaques APT dans un réseau s'appuyant sur des graphes k-nn. Nous avons également effectué une évaluation expérimentale complète, en utilisant les journaux (logs) du serveur proxy d'un réseau réel. Les résultats obtenus sont très prometteurs car nous avons atteint une forte probabilité de détection.

15.2 Perspectives

Comme nous avons pu le montrer dans cette thèse, les graphes k-nn sont un outil puissant pour l'analyse des données. Il existe cependant d'autres structures de données similaires qui sont également de bons candidats pour l'indexation de grands ensembles de données.

Un Small World Network (SWN) est aussi un type de graphe, mais pour celui-ci les voisins d'un nœud ne sont pas nécessairement les autres nœuds les plus similaires. Au contraire, les liens sont construits de façon à minimiser le nombre de nœuds intermédiaires entre les nœuds du graphe. Plus précisément, un SWM est défini comme un réseau où la distance typique L entre deux nœuds choisis au hasard (le nombre de nœuds intermédiaires) augmente proportionnellement au logarithme du nombre de nœuds n dans le réseau : $L \propto \log n$.

Cette propriété rend le SWN approprié comme structure d'indexation pour la recherche des voisins proches, car le nombre moyen de sauts requis pour atteindre un nœud depuis n'importe quel autre nœud du graphe est L. Cela a été montré dans [70]. Néanmoins, cela rend aussi le partitionnement d'un SWN beaucoup plus difficile. Comme les nœuds voisins peuvent être très différents, un grand SWN a beaucoup plus de coupes (liens entre des partitions différentes) s'il est partitionné selon la règle k-medoïdes. Un SWN peut donc être beaucoup plus efficace qu'un graphe k-nn pour le traitement séquentiel, mais peut ne pas être un choix approprié si le graphe doit être distribué entre différents nœuds de calcul.

Dans [71] les auteurs proposent une structure de données apparentées, appelée graphe Hierarchical Navigable Small World (HNSW ou NSW). Cette structure possède plusieurs couches où :

— chaque couche contient un sous-ensemble des nœuds de la couche inférieure et ;

— chaque couche est un graphe k-nn.

Ceci permet de bien séparer les liens longs (dans les couches supérieures de la structure) des liens courts (dans les couches inférieures de la structure). Lorsque le graphe HNSW est utilisé pour effectuer la recherche du voisin le plus proche, les couches supérieures peuvent être utilisées pour trouver rapidement une estimation approximative des voisins les plus proches, puis les couches inférieures sont utilisées pour affiner la solution.

En même temps, en choisissant correctement le rapport entre le nombre de nœuds à chaque

étage, on peut contenir les besoins en mémoire de la structure de données. Dans leur article, les auteurs montrent que la technologie HNSW surpasse les autres approches de pointe en matière de recherche du plus proche voisin.

Cependant, la question demeure de savoir si cette structure de données demeure aussi efficace lorsqu'elle est mise en œuvre en parallèle. De plus, cette structure de données n'a été évaluée que dans le contexte de la recherche du plus proche voisin. Est-elle également utilisable pour effectuer d'autres analyses comme le clustering par exemple ?

Il est intéressant de noter que, tout comme nous, les auteurs de ces articles soulignent l'un des principaux avantages des graphes : ils peuvent être utilisés avec toute mesure de similarité, même non métrique.

Une extension intéressante de notre travail serait d'élargir la portée de l'analyse aux structures d'indexation basées sur des graphes en général (graphes k-nn, NSW, HNSW et autres).

Dans cette thèse, nous nous sommes concentrés uniquement sur les graphes k-nn, à partir de deux points de vue différents :

1. comment les construire ou les mettre à jour efficacement et
2. comment les utiliser.

Ces deux aspects, bien que complètement différents, sont aussi fortement liés : il est inutile de construire une structure de données si elle ne peut pas être utilisée par la suite pour traiter les données, et une structure de données qui est trop lourde à calculer est également inutile.

Notre perspective pour l'avenir est que ces deux aspects devraient être étudiés plus avant : comment construire des structures d'indexation basées sur les graphes, et comment les utiliser pour des applications concrètes. Bien entendu, ces graphes doivent également être comparés entre eux et avec d'autres structures d'indexation classiques.

Chapter 16

Bibliography

[1] Apache spark machine learning library. `https://spark.apache.org/mllib/`.

[2] Clustering the News with Spark and MLLib. `http://bigdatasciencebootcamp.com/posts/Part_3/clustering_news.html`.

[3] Word2vector package. `https://code.google.com/p/word2vec/`.

[4] Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani, and Bernardo A. Huberman. Search in power-law networks. *Phys. Rev. E*, 64:046135, Sep 2001.

[5] Mark Adler. Adler-32 checksum, 1995.

[6] Charu C Aggarwal and ChengXiang Zhai. *Mining text data*. Springer Science & Business Media, 2012.

[7] Tim Althoff, Adrian Ulges, and Andreas Dengel. Balanced Clustering for Content-based Image Browsing. *Artificial Intelligence*, pages 1–4, 2008.

[8] Apache Software Foundation. Spark. `http://spark.apache.org/`.

[9] A Banerjee and J Ghosh. Frequency Sensitive Competitive Learning for Balanced Clustering on High-dimensional Hyperspheres. *IEEE Transactions on Neural Networks*, 15(3):1590–1595, 2002.

[10] Hila Becker et al. Beyond trending topics: Real-world event identification on twitter. In *Proc. of ICWSM*, 2011.

[11] Florian Beil et al. Frequent term-based text clustering. In *Proc. of ACM SIGKDD*, 2002.

[12] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Comput.*, 15(6):1373–1396, June 2003.

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

[14] C.E. Bichot and P. Siarry. *Graph Partitioning*. ISTE. Wiley, 2013.

[15] E. Keith Biggs, Norman L. nd Lloyd and Robin J. Wilson. *Graph Theory 1736–1936*. Oxford University Press, 1986.

[16] Reza Bosagh-Zadeh and Ashish Goel. Dimension independent similarity computation. In *Journal of Machine Learning Research*, 2012.

[17] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Netowrking*, pages 309–320, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[18] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, April 1973.

[19] Emanuele Carlini, Patrizio Dazzi, Andrea Esposito, Alessandro Lulli, and Laura Ricci. Balanced Graph Partitioning with Apache Spark. In *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I*, pages 129—-140. Springer International Publishing, 2014.

[20] Jie Chen, H Fang, and Y Saad. Fast approximate k NN graph construction for high dimensional data via recursive Lanczos bisection. *The Journal of Machine Learning Research*, 10(2009):1989–2012, 2009.

[21] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

[22] Cédric Chevalier and François Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *CoRR*, abs/0907.1375, 2009.

[23] Shu-Chuan Chu, John Roddick, and Jeng-Shyang Pan. Improved search strategies and extensions to k-medoids-based clustering algorithms. 3:212–231, 01 2008.

[24] Cloudera. The truth about mapreduce performance on ssds. `https://blog.cloudera.com/blog/2014/03/the-truth-about-mapreduce-performance-on-ssds/`.

[25] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[26] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE transactions on visualization and computer graphics*, 16(4):599–608, 2009.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[28] Datafloq. Tesco and big data analytics, a recipe for success? `https://datafloq.com/read/tesco-big-data-analytics-recipe-success/665`.

[29] Jeffrey Dean and Sanjay Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):1–13, 2008.

[30] T. Debatty, P. Michiardi, O. Thonnard, and W. Mees. Building k-nn graphs from large text data. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 573–578, Oct 2014.

[31] T. Debatty, F. Pulvirenti, P. Michiardi, and W. Mees. Fast distributed k-nn graph update. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3308–3317, Dec 2016.

[32] Thibault Debatty, Wim Mees, and Thomas Gilon. Graph Based APT Detection. In *Proceedings of the International Conference on Military Communications and Information Systems ICMCIS*, 2018.

[33] Thibault Debatty, Pietro Michiardi, Olivier Thonnard, and Mees Wim. Scalable graph building from text data. In *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, BigMine '14, 2014.

[34] Thibault Debatty, Fabio Pulvirenti, Pietro Michiardi, and Wim Mees. Fast distributed k-nn graph update. In James Joshi, George Karypis, Ling Liu, Xiaohua Hu, Ronay Ak, Yinglong Xia, Weijia Xu, Aki-Hiro Sato, Sudarsan Rachuri, Lyle H. Ungar, Philip S. Yu, Rama Govindaraju, and Toyotaro Suzumura, editors, *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, pages 3308–3317. IEEE, 2016.

[35] Bernard Desgraupes. Clustering indices. *University of Paris Ouest*, 2013.

[36] Wei Dong. Kgraph. `http://www.kgraph.org/`, 2014.

[37] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. *Proceedings of the 20th international conference on World wide web - WWW '11*, page 577, 2011.

[38] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29(4):251–262, August 1999.

[39] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486, 2010.

[40] Glenn Fowler, Phong Vo, and Landon Curt Noll. Fowler/noll/vo hash, 1991.

[41] Ada Wai-chee Fu, Polly Mei-shuen Chan, Yin-Ling Cheung, and Yiu Sang Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *The VLDB Journal*, 9(2):154–173, July 2000.

[42] GinjFo. Stockage : En 60 ans le prix de go a été divisé par 520 millions. `http://www.ginjfo.com/actualites/composants/stockage/stockage-en-60-ans-le-prix-de-go-a-ete-divise-par-520-millions-20151113`.

[43] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[44] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI International Joint Conference on Artificial Intelligence*, pages 1312–1317, 2011.

[45] Jiawei Han, Micheline Kamber, and Anthony K. H. Tung. Spatial clustering methods in data mining: A survey. In Harvey J. Miller and Jiawei Han, editors, *Geographic Data Mining and Knowledge Discovery, Research Monographs in GIS*. Taylor and Francis, 2001.

[46] K Heath, N Gelfand, M Ovsjanikov, M Aanjaneya, and L J Guibas. Image Webs: Computing and Exploiting Connectivity in Image Collections. 2010.

[47] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Comput.*, 26(12):1519–1534, November 2000.

[48] LC Hsieh and GL Wu. Two-stage sparse graph construction using MinHash on MapReduce. In *ICASSP*, pages 1013–1016, 2012.

[49] Matthew A Jaro. Probabilistic linkage of large public health data files. *Statistics in medicine*, 14(5-7), 1995.

[50] David R Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In *SODA*, volume 93, pages 21–30, 1993.

[51] George Karypis and Eui-Hong Sam Han. Fast supervised dimensionality reduction algorithm with applications to document categorization & retrieval. In *Proc. of ACM CIKM*, 2000.

[52] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis * parallel graph partitioning and sparse matrix ordering library. 2003.

[53] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[54] Raimondas Kiveris et al. Connected components in mapreduce and beyond. In *Proc. of ACM SOCC*, 2014.

[55] Gabriel Kliot. Streaming Graph Partitioning for Large Distributed Graphs Categories and Subject Descriptors. *Acm Kdd*, pages 1222–1230, 2012.

[56] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3, Supplem(0):91–97, 2006.

[57] Jesse Kornblum. ssdeep, 2006.

[58] Tudor Lapusan. Mapreduce vs spark. `https://www.slideshare.net/TudorLapusan/map-reduce-vs-spark`.

[59] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[60] David Liben-Nowell. *An Algorithmic Approach to Social Networks*. PhD thesis, Massachusetts Institute of Technology, 2005.

[61] M. Lichman. UCI machine learning repository, 2013.

[62] Frank Lin and William W Cohen. A very fast method for clustering big text datasets. In *ECAI*, pages 303–308, 2010.

[63] Yanchi Liu et al. Understanding of internal clustering validation measures. In *Proc. of IEEE ICDM*, 2010.

[64] Stuart P Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2), 1982.

[65] Alessandro Lulli, Thibault Debatty, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. Scalable k-nn based text clustering. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*, BIG DATA '15, pages 958–963, Washington, DC, USA, 2015. IEEE Computer Society.

[66] Alessandro Lulli, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. Ng-dbscan: Scalable density-based clustering for arbitrary data. *Proc. VLDB Endow.*, 10(3):157–168, November 2016.

[67] Alessandro Lulli et al. Cracker: Crumbling large graphs into connected components. In *Proc. of IEEE ISCC*, 2015.

[68] Markus Maier, Matthias Hein, and Ulrike Von Luxburg. Cluster identification in nearest-neighbor graphs. *Algorithmic Learning Theory*, (May):196–210, 2007.

[69] Mikko I Malinen and Pasi Franti. Balanced K-Means for Clustering. *Structural, Syntactic, and Statistical Pattern Recognition*, 8621(February):32–41, 2014.

[70] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[71] Yury Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. Technical report.

[72] Yutaka Matsuo and Mitsuru Ishizuka. Keyword extraction from a single document using word co-occurrence statistical information. *International Journal on Artificial Intelligence Tools*, 13(1), 2004.

[73] Tomas Mikolov et al. Distributed representations of words and phrases and their compositionality. In *Proc. of NIPS*, 2013.

[74] Andrew Moore. An introductory tutorial on kd trees. `http://www.autonlab.org/autonweb/14665/version/2/part/5/data/moore-tutorial.pdf`, 1991.

[75] Hlib Mykhailenko. *Distributed edge partitioning.* Theses, Université Côte d'Azur, CNRS, I3S, France, June 2017.

[76] Meghanathan N. *Graph theoretic approaches for analyzing large-scale social networks.* IGI Global, 2017.

[77] Mark EJ Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3), 2006.

[78] R. T. Ng and Jiawei Han. Clarans: a method for clustering objects for spatial data mining. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1003–1016, Sep 2002.

[79] Rodrigo Paredes, E Chávez, K Figueroa, and Gonzalo Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. *Experimental Algorithms*, 2006.

[80] Hae Sang Park and Chi Hyuck Jun. A simple and fast algorithm for K-medoids clustering. *Expert Systems with Applications*, 36(2 PART 2):3336–3341, 2009.

[81] James Philbin, Josef Sivic, and Andrew Zisserman. Geometric latent dirichlet allocation on a matching graph for large-scale image datasets. *International Journal of Computer Vision*, 95(2):138–153, 2011.

[82] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, and Seif Haridi. Distributed vertex-cut partitioning. In *Proceedings of the 14th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - Volume 8460*, pages 186–200, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[83] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. JA-BE-JA: A distributed algorithm for balanced graph Partitioning. In *International Conference on Self-Adaptive and Self-Organizing Systems, SASO*, pages 51–60, 2013.

[84] Costin Raiu, Igor Soumenkov, Kurt Baumgartner, and Vitaly Kamluk. The MiniDuke Mystery: PDF 0-day Government Spy Assembler 0x29A Micro Backdoor. Technical report, Kaspersky Lab, 2013.

[85] A. Rajaraman and J. Ullman. *Mining of Massive Datasets*, chapter 10. Cambridge University Press, 2010.

[86] A. Rajaraman and J. Ullman. *Mining of Massive Datasets*, chapter 3. Cambridge University Press, 2010.

[87] Vibhor Rastogi et al. Finding connected components in map-reduce in logarithmic rounds. In *Proc. of IEEE ICDE*, 2013.

[88] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 1987.

[89] I. Saha and A. Mukhopadhyay. Genetic algorithm and simulated annealing based approaches to categorical data clustering. In *2008 IEEE Region 10 and the Third international Conference on Industrial and Information Systems*, pages 1–6, Dec 2008.

[90] John P. Scott and Peter J. Carrington. *The SAGE Handbook of Social Network Analysis*. Sage Publications Ltd., 2011.

[91] Thomas B. Sebastian and Benjamin B. Kimia. Metric-based shape retrieval in large databases. *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, (August 2002):291–296, 2002.

[92] Yun Shen et al. Mr-triage: Scalable multi-criteria clustering for big data security intelligence applications. In *Proc. of IEEE BigData*, 2014.

[93] Michael Steinbach et al. A comparison of document clustering techniques. In *Proc. of KDD workshop on text mining*, 2000.

[94] Josh B Tenenbaum, V de Silva, and J C Langford. A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science (New York, N.Y.)*, 290(5500):2319–2323, 2000.

[95] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition, Third Edition*. Academic Press, Inc., Orlando, FL, USA, 2006.

[96] Olivier Thonnard and Marc Dacier. A strategic analysis of spam botnets operations. In *Proc. of ACM CEAS*, 2011.

[97] Zhiqiang Toh and Wenting Wang. Dlirec: Aspect term extraction and term polarity classification system. In *Proc. of SemEval*, 2014.

[98] Andrew Tridgell. Spamsum, 2002.

[99] JK Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information processing letters*, 40(November):175–179, 1991.

[100] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *Proc. VLDB Endow.*, 10(5):493–504, January 2017.

[101] Lu Wang, Bin Shao, Yanghua Xiao, and Haixun Wang. How to partition a billion-node graph. Technical report, February 2013.

[102] Wikipedia. Binary search algorithm. `https://en.wikipedia.org/wiki/Binary_search_algorithm`.

[103] William E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research*, pages 354–359, 1990.

[104] William E Winkler. The state of record linkage and current research problems. In *Statistical Research Division, US Census Bureau*, 1999.

[105] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as*

part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pages 15–28, San Jose, CA, 2012. USENIX.

[106] Yan-ming Zhang, Kaizhu Huang, Guanggang Geng, and Cheng-lin Liu. Fast k NN Graph Construction with Locality Sensitive Hashing. In *Machine Learning and Knowledge Discovery in Databases*, pages 660–674. 2013.

[107] Ying Zhao and George Karypis. Empirical and theoretical comparisons of selected criterion functions for document clustering. *Journal of Machine Learning*, 55(3), 2004.