Scalable Graph Building from Text Data

Thibault Debatty THIBAULT.DEBATTY@RMA.AC.BE
Royal Military Academy, Brussels, Belgium

Pietro Michiardi PIETRO.MICHIARDI@EURECOM.FR
EURECOM, Campus SophiaTech, France

Olivier Thonnard OLIVIER_THONNARD@SYMANTEC.COM
Symantec Research Labs, Sophia Antipolis, France

Wim Mees WIM.MEES@RMA.AC.BE
Royal Military Academy, Brussels, Belgium

Editors: Wei Fan, Albert Bifet, Qiang Yang and Philip Yu

Abstract

In this paper we propose NNCTPH, a new MapReduce algorithm that is able to build an approximate k-NN graph from large text datasets. The algorithm uses a modified version of Context Triggered Piecewise Hashing to bin the input data into buckets, and uses an exhaustive search inside the buckets to build the graph. It also uses multiple stages to join the different unconnected subgraphs. We experimentally test the algorithm on different datasets consisting of the subject of spam emails. Although the algorithm is still at an early development stage, it already proves to be four times faster than a MapReduce implementation of NN-Descent, for the same quality of produced graph.

Keywords: k-NN graph, graph building, MapReduce, text data, Context-Triggered Piecewise Hashing

1. Introduction

A graph is a mathematical structure used to represent relations between objects. A graph is made up of nodes (or vertices) connected with edges. In some cases, the edges have a weight, resulting in a weighted graph. Graph theory is a very ancient topic, dating back from 1736 Biggs and Wilson (1986). It has received a strong highlight these last years with the explosion of web 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 Rajaraman and Ullman (2010b), Liben-Nowell (2005), Broder et al. (2000) or Scott and Carrington (2011).

In a lot of cases however, we may wish to analyze unstructured data, that is not formatted as a graph. Nevertheless, it can be interesting to convert the dataset into a graph, to be able to use these algorithms. In most cases, the trick used is to build a 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 similarity metric.

Another possibility would be 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 Belkin and Niyogi (2003) 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 Chen et al. (2009). 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.

In this context, in this paper we present NNCTPH, a new MapReduce algorithm that is able to build an approximate k-NN graph from large text datasets. The algorithm uses Context Triggered Piecewise Hashing (CTPH), a hashing function that tends to produce the same hash for similar input strings, to bin the input text data into buckets. To control the number of buckets, we develop a custom CTPH function. The algorithm then uses an exhaustive search inside the buckets to build the graph. It also uses multiple stages to join the different unconnected subgraphs. We experimentally test the algorithm on different datasets consisting of the subject of spam emails. Although the algorithm is still at an early development stage, it already proves to be four times faster than a MapReduce implementation of NN-Descent, for the same quality of produced graph.

The rest of this paper is organized as follows. In Section 2 we present existing algorithms to build a k-NN graph, and algorithms that perform nearest neighbor search in general. In Section 3 we present the implementation details of our algorithm and context triggered piecewise hashing. In Section 4 we show test results, and compare our algorithm with a MapReduce implementation of NN-Descent. More precisely, we focus on the quality of computed graph, on execution time, and on scalability of the algorithms. Finally, in Section 5 we present our conclusions and future work.

2. Related work

2.1. Nearest neighbor search

k-NN graph building algorithms are of course closely related to nearest neighbor search algorithms, and also to the k-nearest neighbors method.

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.

But 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.

Different methods exist to find the nearest neighbor, or the k-nearest neighbors, of a point. The naive method, also called linear search, consists in computing the distance between the query point and every other point in the set, keeping track of the best node so far" (or k best nodes so far).

Some techniques rely on the branch and bound algorithm, with some kind of index to partition the space M. For example, a k-d tree, that recursively partitions the space into equally sized sub-spaces, can be used to speedup search, like proposed in Moore (1991).

R-trees can also be used for euclidean spaces. In the case of generic metric spaces, vantage-point trees and BK-trees can be used.

Finally, some algorithms use Locality-Sensitive Hashing (LSH), like Rajaraman and Ullman (2010a). LSH is originally a method used to perform probabilistic dimension reduction of high-dimensional data. The basic idea is to hash the input items so that similar items are mapped to the same buckets 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.

2.2. k-NN graph building algorithms

To build a complete k-NN graph, the most naive way to proceed is of course using brute force to compute all pairwise similarities. This method has a computational cost of $O(n^2)$.

A more subtle way consists in iteratively using a nearest neighbor search algorithm, like the ones presented above, to find the neighbors of all nodes in the dataset.

Lately, dedicated algorithms have also been proposed to efficiently build a k-NN graph from a dataset. Most of them naturally share similarities with nearest neighbor search algorithms.

In Paredes et al. (2006), 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.

In Connor and Kumar (2009), the authors present a distributed algorithm, but that requires a shared memory architecture to store a shared kd-tree based index.

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 Dong et al. (2011). 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, Dong (2014).

Various authors propose algorithms relying on locality-sensitive hashing.

In Hsieh and Wu (2012), 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 Zhang et al. (2013), 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, Zhang et al. 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 offers an efficient solution. Algorithms that rely on indexes are hard to implement in parallel on a shared nothing architecture like MapReduce (MR). 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 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 during the experimental tests presented below.

3. Building a graph from a big text dataset

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 bins. Hence we call the algorithm NNCTPH.

3.1. Context Triggered Piecewise Hashing

Context Triggered Piecewise Hashing (CTPH), also called Fuzzy Hashing, 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, Tridgell (2002). 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, Adler (1995). 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 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. Therefore the block hash function is based on the Fowler/Noll/Vo (FNV) hash function, Fowler et al. (1991). In the original algorithm, the matching value for the rolling hashing is chosen to produce a final of 64 characters.

Fuzzy hashing is also known under the name ssdeep, Kornblum (2006b), which is the name of the tool implemented by Kornblum. In Kornblum (2006a), ssdeep is used to identify almost identical files to support computer forensics.

3.2. NNCTPH

We propose here to use CTPH to build a k-NN graph from text data using MapReduce. The algorithm, NNCTPH, is presented in Algorithms 1 and 2. It requires a single MapReduce job. In the map phase, the algorithm uses CTPH to produce a hash of each input string. This hash value is then used to bin the string into a bucket. Each reduce task performs an exhaustive search to compute pairwise similarities between all strings of a single bucket, and for each node emits the k edges with the highest similarity.

Algorithm 1 NNCTPH Map

```
Input: $stages, hash\_length, hash\_letters$

procedure Map($string$)
    $hash$ = CTPH($string$, $stages$, $hash\_length$, $hash\_letters$)
    for $s$ in 0..$stages$ do
        Emit($s\_hash[s] \Rightarrow string$)
    end for
end procedure
```

Algorithm 2 NNCTPH Reduce

```
Input: $k, stages$

procedure Reduce($key, < strings >$)
    $k' = k/stages$
    ReadAllStrings()
    ComputePairwiseSimilarities()
    for $s$ in $strings$ do
        edges = FindKNN($s, k'$)
        Emit($edges$)
    end for
end procedure
```

To control the number of buckets, and hence the average number of nodes per bucket, we modified 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.

For example, by using a hash of two characters with ten possible letters (`A` to `J`), we create 100 buckets. The number of buckets will have an influence on the average number of nodes per bucket and on the number of similarities to compute, but also on the parallelism of the algorithm, and hence on the processing time and on the quality of the final graph.

Doing this, we would end up with a series of unconnected subgraphs, as no edges are created between the nodes of different buckets. To avoid this and reconnect the graph, in the map phase we create a longer hash (using a coefficient we call *stages*) and emit the input string once for each subpart of the hash. Then, each reducer emits $stages/k$ edges for each node. Specifically, our CTPH function directly returns an array of *stages* elements.

For example, to run the algorithm with 100 buckets and two stages, our CTPH function produces a hash of four characters, using ten letters. The returned value is an array of two strings, each consisting of two characters. If the hash of an input string is `ABCD`, the returned value is an array `AB`, `CD`, and the original string will be emitted twice by the mapper: once for `AB`, and once for `CD`. In this way, we can expect that the reduce task for bucket `CD` will produce edges to nodes located outside bucket `AB`, hence reconnecting the graph.

The number of stages used will also have an impact on the quality of the graph, and on the quantity of data to shuffle and transmit over the network.

Our algorithm thus requires three parameters: the number of stages, the number of characters in a hash, and the number of letters used to produce the hash. In summary, these 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 number of similarities to compute, and on the quantity of RAM required by the reducers. In the future, we plan to perform a sensitivity analysis of the effects and interactions of these parameters.

4. Experimental evaluation

To experimentally test our algorithm, we implement it using Hadoop and test it on datasets containing the subject of spam emails. We also compare it against a Hadoop implementation of NN-Descent, and a Hadoop implementation of the brute-force method. All algorithms are executed on a cluster of four servers, each equipped with two quad-core processors and 16GB of RAM.

To compute the similarity between spam subjects, we use Jaro-Winkler distance, Winkler (1990). This measure of string similarity is normalized such that zero equates to no similarity and one is an exact match.

4.1. Graph quality

To compare the accuracy of algorithms, like in Dong et al. (2011), we use recall to measure the quality of produced graphs. The ground truth is the true k-NN graph obtained using the naive, brute-force, algorithm. The recall of a single node is the number of its correct edges divided by k. The recall of an approximate k-NN graph is the average recall of all nodes.

For the tests, we use a dataset containing the subject of 200,000 spam emails, and want to build a 10-NN graph. For NNCTPH, we use two stages, hashes of two characters, and we let the number of possible letters vary between 31 and 40. Doing so, we let the number of buckets vary between 961 and 1600, and the average number of spams per bucket vary between 208 and 125. The resulting execution time and recall are displayed in Table 1.

Table 1: Running time and recall of NNCTPH algorithm on a dataset of 200,000 spams with two stages and hashes of two characters, to build a 10-NN graph

	T1	**T2**	**T3**	**T4**	**T5**
Letters	31	32	33	34	35
Buckets	961	1024	1089	1156	1225
Running time (sec)	153	207	136	142	118
Similarities (x 1000.000)	160	164	156	146	143
Recall	23.39%	23.40%	23.17%	23.24%	23.25%
	T6	**T7**	**T8**	**T9**	**T10**
Letters	36	37	38	39	40
Buckets	1296	1369	1444	1521	1600
Running time (sec)	129	141	115	157	174
Similarities (x 1000.000)	138	138	133	129	133
Recall	23.18%	23.05%	23.04%	23.08%	23.08%

As we can notice, the number of edges that are correctly found is fairly stable, around 23%. This might seem low, but given the size of our dataset, rapidly discovering this amount of correct edges is not trivial. The average running time of our algorithm for these tests is

140 sec. As a matter of comparison, the brute-force algorithm we use to compute ground truth takes approximatively 9 hours to complete on the same hardware.

We also compare our algorithm with a MapReduce implementation of NN-Descent. Table 2 shows the running time and recall of 10 iterations of NN-Descent on the same dataset. As we can see, the algorithm requires 8 or 9 iterations to achieve the same result. For the same quality of the resulting graph, our algorithm is thus on average four times faster than NN-Descent.

Table 2: Running time and recall of NN-Descent algorithm on a dataset of 200,000 spams, to build a 10-NN graph

Iterations	**1**	**2**	**3**	**4**	**5**
Running time (sec)	134	203	272	339	406
# similarities (x 1000.000)	22	44	66	88	110
Recall	0.053%	0.37%	1.60%	5.57%	12.37%
Iterations	**6**	**7**	**8**	**9**	**10**
Running time (sec)	472	537	604	670	736
Similarities (x 1000.000)	132	154	176	198	220
Recall	18.20%	21.57%	23.09%	23.77%	24.07%

Moreover, even when we execute more iterations of NN-Descent, the algorithm has difficulties to reach a recall higher than 24%. This is shown on Figure 1, which illustrates the evolution of recall for up to 30 iterations of NN-Descent. As we can see, starting from iteration 9, the recall only rises very slowly. This is probably due to the structure of our dataset, where additional edges are difficult to find because they are hidden amongst a lot of other edges of similar but slightly inferior similarity.

On Table 1 we can also observe that, as we could expect, when the number of letters and thus the number of buckets rise, the number of computed similarities roughly decreases. But it is not necessarily the case of the computing time. This is mainly due to the fact that the data sent by the mappers to the reduce tasks is skewed. The computing time is thus dominated by these few reduce tasks that have a lot more similarities to compute than the others.

This is confirmed on Figure 2 that shows the frequency distribution of the number of nodes per bucket (and thus per reduce task). NNCTPH is used on the dataset of 200,000 spams, with two stages, hashes of two characters, and 32 letters. We thus create 1024 buckets, and an average of 195 strings per bucket. As we can see, there is a peak around 100 nodes per bucket, but there are also a lot of buckets with more than 1000 nodes, and even one with approximatively 7000 nodes. As the algorithm uses brute-force to compute edges inside buckets, these one will have much more similarities to compute, and will mostly be responsible for the execution time.

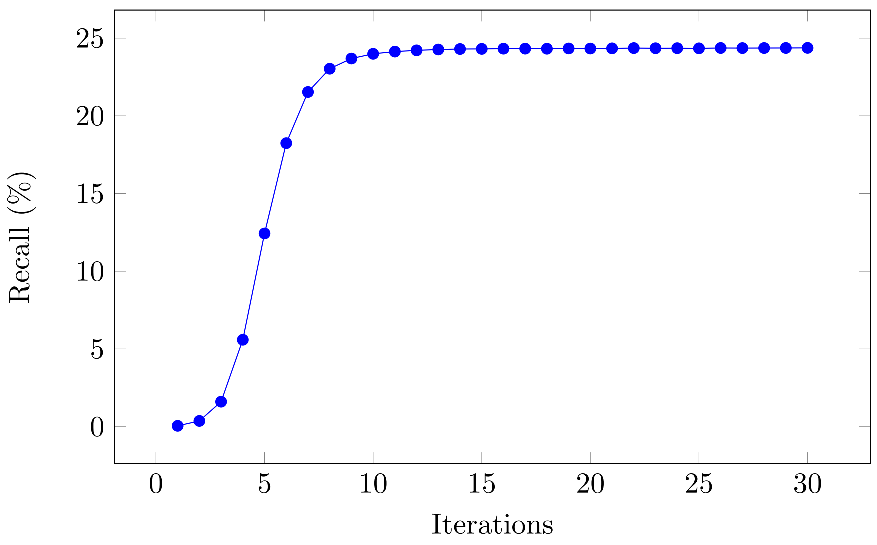

Figure 1: Evolution of the recall of NN-Descent algorithm with up to 30 iterations for a dataset of 200,000 spam subjects

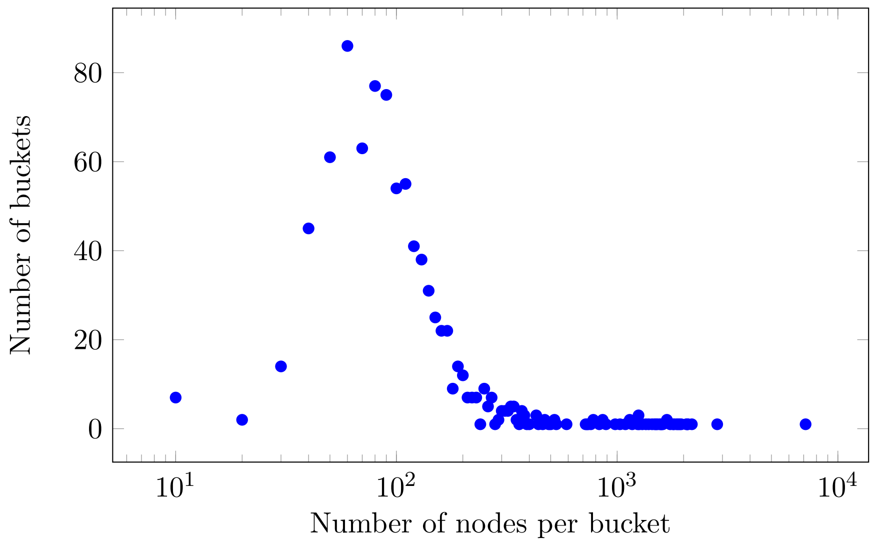

Figure 2: Frequency distribution of the number of nodes per bucket when using NNCTPH to create a graph from a dataset of 200,000 spams with two stages, hashes of two characters, and 32 letters (1024 buckets)

To alleviate this problem, in a future version of the algorithm we plan to use NN-Descent instead of brute-force to find nearest-neighbors inside buckets. This algorithm has proved to be very efficient when implemented sequentially, and does not require $O(n^2)$ similarity computations. Generally, handling skewed data is not a trivial problem, and has already been studied in the context of distributed databases, as shown in Xu et al. (2008). To improve the performance of NNCTPH we will have to tackle this problem. Amongst other possibilities we plan to apply extendible hashing.

4.2. Scalability

To test the scalability of our algorithm, we run it against different datasets containing 200,000 to 600.000 spam emails. We use two stages, hashes of two characters, and 64 letters (4096 buckets). Due to hardware constraints, for these datasets we don't have the ground-truth to compute recall. Instead we compute the average similarity of edges. A higher average similarity indicates a better quality of the generated graph.

To compare the performance and computational cost of the algorithms, as the number of similarities to compute depends on the size of the dataset, we use the scan rate, defined as follows:

$$\text{scan rate} = \frac{\#\text{ similarity evaluations}}{n(n-1)/2}$$

where $n(n-1)/2$ is the number of similarities computed by the naive algorithm.

Table 3: Running time, scan rate and average similarity of edges found by NNCTPH for different datasets

Dataset	**spam200k**	**spam400k**	**spam600k**
Spams	200,000	400.000	600.000
Running time (sec)	97	316	675
Similarities (x 1000.000)	88	359	773
Scan rate	0.44%	0.45%	0.43%
Average similarity	0.843	0.863	0.860

As we could expect, the number of similarities to compute and the processing time rise proportionally with n^2 when using a fixed number of buckets. If the problem of skewed data is set aside, reducing the processing time simply requires to use more buckets. Quite surprisingly however, the average similarity of the edges, and thus the quality of the graph, does not really evolve, even if the number of strings per bucket rises. Therefore, there must exist a sweet spot that maximize the processing speed, while preserving an acceptable graph quality. The study of the influence of these parameters is left as a future work.

5. Conclusions and future work

In this paper we propose NNCTPH, a MapReduce algorithm that builds an approximate k-NN graph from large text datasets. The algorithm uses a modified version of Context Triggered Piecewise Hashing to bin the input data into buckets, and uses brute-force inside the buckets to build the graph. It also uses multiple stages to join the different unconnected subgraphs.

We experimentally tested the algorithm on different datasets containing the subject of spam emails. Although the algorithm is still at an early development stage, it already proves to be on average four times faster than a MapReduce implementation of NN-Descent, for the same quality of produced graph.

In the future we plan to use NN-Descent inside the buckets, instead of exhaustive search, as this algorithm has proved to be very efficient when implemented sequentially. We will also study the influence of the different parameters of the algorithm on the quality of the resulting graph, and on the performance of the algorithm (computational cost, parallelization, network traffic), and we will study the influence of graph quality on the algorithms applied afterward on the graph (connected components for example). We will have to tackle the problem of skewed data, and test the scalability of the algorithm on bigger datasets and bigger clusters. Finally, we plan to compare our algorithm with algorithms that build a k-NN graph using the bag-of-words (BOW) model.

Acknowledgments

This work has been partially supported by the EU project BigFoot (FP7-ICT-317858).

References

Mark Adler. Adler-32 checksum, 1995. URL https://github.com/madler/zlib/blob/master/adler32.c.

Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Comput.*, 15(6):1373–1396, June 2003. ISSN 0899-7667. doi: 10.1162/089976603321780317. URL http://dx.doi.org/10.1162/089976603321780317.

E. Keith Biggs, Norman L. nd Lloyd and Robin J. Wilson. *Graph Theory 1736–1936*. Oxford University Press, 1986.

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. URL http://dl.acm.org/citation.cfm?id=347319.346290.

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. URL http://dl.acm.org/citation.cfm?id=1755852.

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. ISSN 1077-2626. doi: 10.1109/TVCG.2010.9. URL http://www.ncbi.nlm.nih.gov/pubmed/20467058.

Wei Dong. Kgraph, 2014. URL http://www.kgraph.org/.

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. doi: 10.1145/1963405.1963487. URL http://portal.acm.org/citation.cfm?doid=1963405.1963487.

Glenn Fowler, Phong Vo, and Landon Curt Noll. Fowler/noll/vo hash, 1991. URL http://www.isthe.com/chongo/tech/comp/fnv/.

LC Hsieh and GL Wu. Two-stage sparse graph construction using MinHash on MapReduce. In *ICASSP*, pages 1013–1016, 2012. ISBN 9781467300469. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6288057.

Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3, Supplem(0):91–97, 2006a. ISSN 1742-2876. doi: http://dx.doi.org/10.1016/j.diin.2006.06.015. URL http://www.sciencedirect.com/science/article/pii/S1742287606000764.

Jesse Kornblum. ssdeep, 2006b. URL http://ssdeep.sourceforge.net/.

David Liben-Nowell. *An Algorithmic Approach to Social Networks*. PhD thesis, Massachusetts Institute of Technology, 2005.

Andrew Moore. An introductory tutorial on kd trees, 1991. URL http://www.autonlab.org/autonweb/14665/version/2/part/5/data/moore-tutorial.pdf.

Rodrigo Paredes, E Chávez, K Figueroa, and Gonzalo Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. *Experimental Algorithms*, 2006. URL http://link.springer.com/chapter/10.1007/11764298_8.

A. Rajaraman and J. Ullman. *Mining of Massive Datasets*, chapter 3. Cambridge University Press, 2010a.

A. Rajaraman and J. Ullman. *Mining of Massive Datasets*, chapter 10. Cambridge University Press, 2010b.

John P. Scott and Peter J. Carrington. *The SAGE Handbook of Social Network Analysis*. Sage Publications Ltd., 2011. ISBN 1847873952, 9781847873958.

Andrew Tridgell. Spamsum, 2002. URL http://www.samba.org/ftp/unpacked/junkcode/spamsum/.

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.

Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. Handling data skew in parallel joins in shared-nothing systems. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1043–1052, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376720. URL http://doi.acm.org/10.1145/1376616.1376720.

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. ISBN 978-3-642-40990-5.