JMLR: Workshop and Conference Proceedings 29:1–6, 2014 BIGMINE 2014

# Efficient 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*

## Abstract

This is the abstract for this article.
**Keywords:** List of keywords

## 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. It has received a strong highlight these last years with the explosion of social networks like Facebook and Twitter. Indeed, the data generated by these networks can easily be formatted as a graph, making graph algorithms a premium choice for processing the data.

As a consequence, a lot of algorithms have been developed, and implemented, to efficiently analyze this kind of data, either sequentially or in parallel. XXXXXXX add examples of graph algorithms connected components, clustering, ...

In a lot of cases, we may have 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 (XXXXXXXXX add reference), the trick used is to build a k-nearest neighbors graph (k-NN graph), a graph where each node is connected to (has an edge to) its k nearest neighbors, according to a given similarity metric.

## 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 so far" (or k "best so far").

Some techniques rely on the branch and bond algorithm, with some kind of index to partition the space $M$. For example, a k-d tree, that recursively partition the space into equally sized sub-spaces can be used to speedup search, like 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.

XXXXXX add explanation and references!

Finally, some algorithms use Locality-Sensitive Hashing (LSH), like Rajaraman and Ullman (2010). 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 high probability. At the opposite of conventional hash functions, such as those used in cryptography, in the case of LSH the goal is to maximize the probability of "collision" between similar items.

### 2.2. k-NN graph building algorithms

The most naive way to build a k-NN graph 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, different algorithms have been proposed specifically to efficiently build a k-NN graph from a dataset. Most of them will 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 build 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. The authors tested the algorithms using text data 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 an approximate k-NN 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 anymore. The main advantage of this algorithm is that it works with any similarity measure. Dong et al. experimentally found the computational cost is around $O(n^{1.14})$.

They also propose a MapReduce version of the algorithm. Internally, instead of working with edges, each node has a neighbors list, a list of candidate neighbors. 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 new candidate neighbors. The reducer merges all neighbors for each node, 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.

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, they use the algorithm proposed by Dong et al. 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. By doing so, the items with same hash code will fall in the same bucket with a high probability.

Finally, Zhang et al. show experimentally that their algorithm is much faster than existing algorithms, for similar quality of the built graph.

When it comes to building a k-nn graph from a big unstructured text dataset, none of the previous algorithms offers a efficient solution. Algorithms that rely on indexes are hard to implement in parallel on a distributed memory architecture like MapReduce. As LSH functions are defined only for some similarity measures ($l_p$, Mahalanobis distance, kernel similarity, and $\chi^2$ distance), the algorithms relying on LSH cannot be used when it comes 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 measure.

In the case of NNDescent, 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 very inefficient when implemented in parallel. This will be confirmed during the experimental tests presented below.

## 3. NNCTPH

### 3.1. Context Triggered Piecewise Hashing

Context Triggered Piecewise Hashing (CTPH), also called Fuzzy Hashing, 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 zlib Adler (1995) and uses a windows 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.

Each chunk is hashed into a single character (out of 64). The block hash used is based on the Fowler/Noll/Vo (FNV) hash function Fowler et al. (1991).

The final hash of the input string is the sequence of these segment hashes. In the original algorithm, the matching value for the rolling hashing is chosen in such a way that the produced hash has a length of 64 characters.

Kornblum later implemented the algorithm under the name ssdeep Kornblum (2006b). In Kornblum (2006a), he uses the algorithm to identify almost identical files to support computer forensics.

## 4. Experimental evaluation

### 4.1. Performance measure

Like in Dong et al. (2011), we use recall to measure the accuracy of algorithms. 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.

To compare computational cost, 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.

### 4.2. Results

Table 1: Running time and scan rate of NNCTPH algorithm

| | **spam200k** | **spam400k** | **spam600k** | **spam800k** |
|---|---|---|---|---|
| # spams | 200.000 | 400.000 | 600.000 | 800.000 |
| Running time (sec) | 97 | 316 | 675 | 1093 |
| # similarities | 88.262.345 | 359.577.959 | 773.200.303 | 1.346.569.275 |
| Scan rate | 0.44% | 0.45% | 0.43% | 0.42% |

## 5. Conclusions

## 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.

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, 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/.

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, 2010.

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

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.