Fast distributed k-nn graph update

Thibault Debatty, Fabio Pulvirenti, Pietro Michiardi, Wim Mees

Fast distributed k-nn graph update

Thibault Debatty*, Fabio Pulvirenti†, Pietro Michiardi‡, Wim Mees*
*Royal Military Academy, Brussels, Belgium
Email: firstname.name@rma.ac.be
†Polytechnic University of Turin
Email: fabio.pulvirenti@polito.it
‡EURECOM, Campus SophiaTech, France
Email: pietro.michiardi@eurecom.fr

***Abstract*—In this paper, we present an approximate algorithm that is able to quickly modify a large distributed k-nn graph by adding or removing nodes. The algorithm produces an approximate graph that is highly similar to the graph computed using a naïve approach, although it requires the computation of far fewer similarities. To achieve this goal, it relies on a novel, distributed graph based search procedure. All these algorithms are also experimentally evaluated, using both euclidean and non-euclidean datasets.**

I. Introduction

A k-nn graph is a data structure where each element (called node or vertex) has a link (an edge) to the k most similar elements of the dataset. Building a k-nn graph is a time consuming operation while, at the opposite, analysing it is usually very fast. Therefore, k-nn graphs are often used for interactive data analytics, like clustering for example.

In the general case, building a k-nn graph requires the computation of $n \cdot (n-1)/2$ similarities. If the similarity used is a metric, the triangle inequality can be used to reduce the number of computations. In any way, building an exact k-nn graph remains a computationally heavy process. Therefore, research mainly focuses on building approximate k-nn graphs, where each node has edges to the k most similar nodes with a high probability.

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 paper we propose an 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 paper is organized as follows. In Section II we present existing graph based search algorithms and graph partitioning algorithms. In Sections III and IV we show how we add and remove nodes from the distributed k-nn graph. In Section V 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 VI. In Section VII we perform an experimental evaluation, where we perform a parameter study of the algorithm. Finally, in Section VIII, we present our conclusions and propositions for future work.

II. Related work

We present here related work in the domain of k-nn graph building. One important step in our algorithm consists in searching the nearest neighbours of the new data point using the existing graph. Therefore, we also present existing graph based nearest neighbour search algorithms. Finally, searching the graph in a distributed fashion requires a specific partitioning of the graph. Hence, we

present here existing graph partitioning algorithms.

A. 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 [6], 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 [5], also created a software called KGraph [4] 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.

B. 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 [10], 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 [3], 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 [7], 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 V, 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 centres 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) [11]. To the best of our knowledge, the most efficient algorithm for performing k-medoids clustering is currently the Voronoi iteration method proposed in [9], 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 [8], the authors proposed a method that has a complexity $O(n^3)$, which makes it too complex for large graphs. In [2], 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 [1], 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.

III. 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 V and 2) update the graph using the procedure presented in Algorithm 2. 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 the 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 1.

Algorithm 1 Distributed online k-nn graph building: 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 2
Update(*graph*, *node*, *neighbours*, 0)

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

Algorithm 2 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 $k^{\text{DEPTH}+1}$ 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.

IV. 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 if 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 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.

V. 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 [6]: 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 $\text{similarity}(query, r) < s_{\max}/e$. 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 a euclidean space of d dimensions and uniformly randomly distributed data points. In such a space, observe that for any node, 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. As the improved algorithm iterates as soon as a node with higher similarity is found, the expected number of similarities to compute for each analysed node is:

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

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

Table I
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

These additional approximations allow 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 [6] 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.

VI. 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 [9]. 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 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 $w(m_j, t)$:

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

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

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

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 ($\mathit{imbalance} \geqslant 1$):

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

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

In a distributed shared nothing environment the different compute nodes don't 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:

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

where $\mathit{capacity}_L$ is the contribution of each compute node to total size of the final cluster:

$$\mathit{capacity}_L = \frac{n \cdot \mathit{imbalance}}{k \cdot c}$$

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.

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:

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

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

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

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 1, 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.

VII. 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 [1].

The experiments are run using two datasets, with different similarity measures[2].

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-euclidean similarity measures.

[1]The source code of algorithms and evaluation scripts can be found at https://github.com/tdebatty/spark-knn-graphs

[2]Instructions to download and process the datasets can be found at https://github.com/tdebatty/java-datasets

A. 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:

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

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 produce graph can be measured as the number of correctly modified edges divided by the expected number of edges that the algorithm should modify:

$$Q = \frac{e_c - \mathbf{E}(e_u)}{\mathbf{E}(e_m)} \tag{2}$$

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

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

The results are shown in Figure 1. It confirms 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),

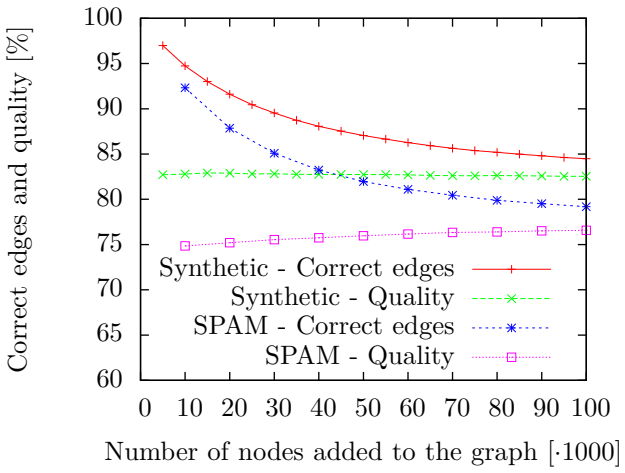

Figure 1. Graph quality

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.

B. 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 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 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 outreaches the speed increase

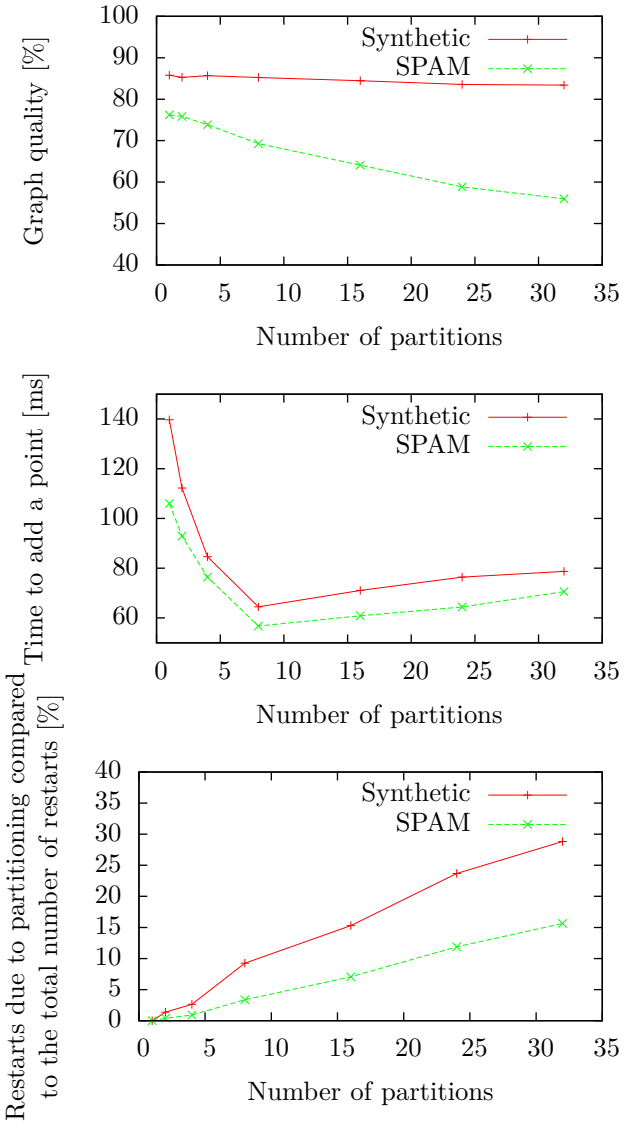

Figure 2. Influence of the number of partitions

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.

C. 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

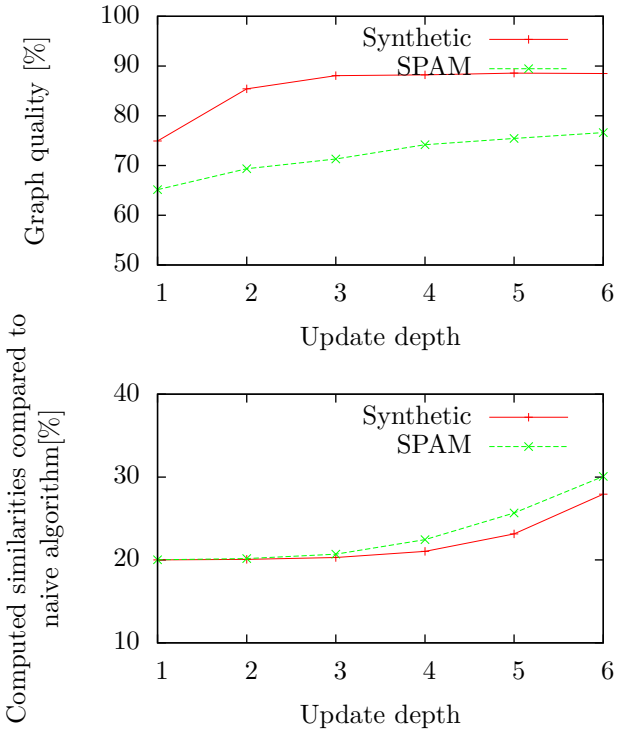

Figure 3. Influence of the update depth

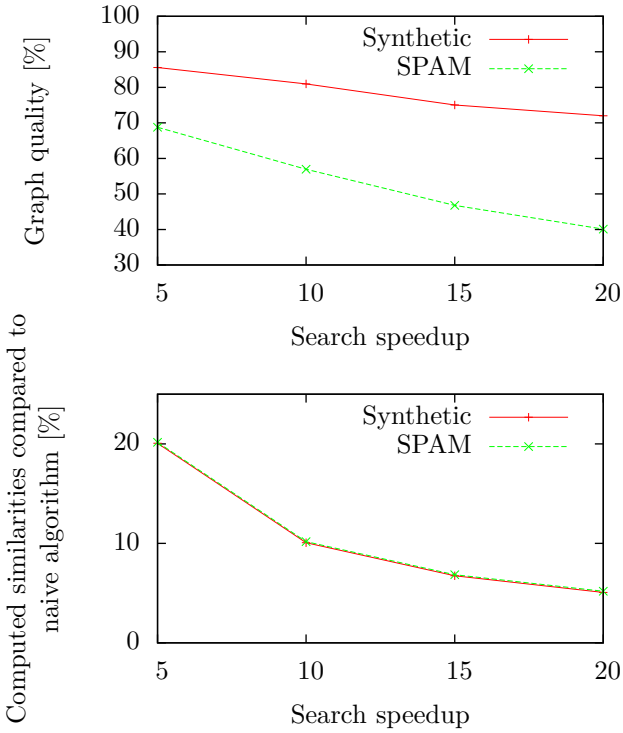

Figure 4. Influence of the search speedup

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 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 3, there is a clear effect of diminishing return, and an update depth of three seems to be a good compromise.

D. 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, 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 floor on the number of computed similarities due the similarities that have to be computed to update the edges of existing nodes.

E. Search expansion parameter

Another important parameter of the sequential search procedure is the expansion parameter. As we can see in Figure 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.

F. 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 6. We build a synthetic dataset in $\mathbf{R}^{20}$, then build a k-nn graph with different

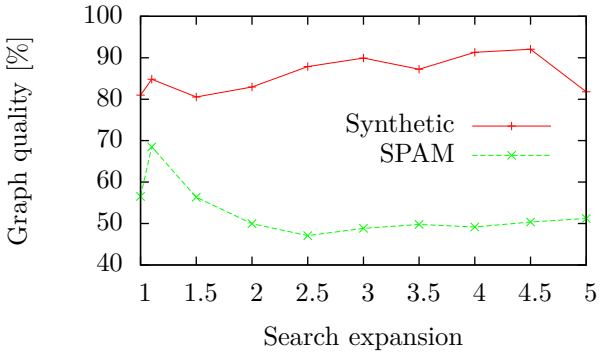

Figure 5. Influence of the search expansion

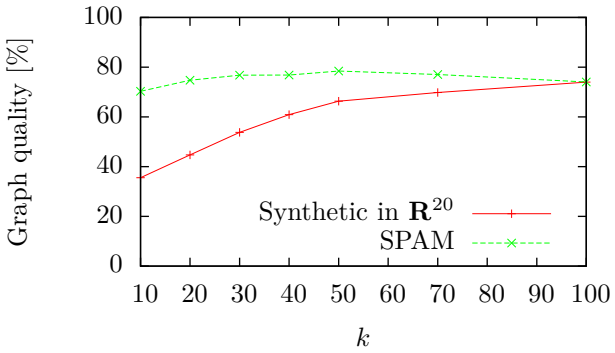

Figure 6. Influence of the dimensionality

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.

G. Partitioning

We compare here the quality of the graph produced after adding a fixed number of nodes if: 1) we don't 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.

As we can see on Figure 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.

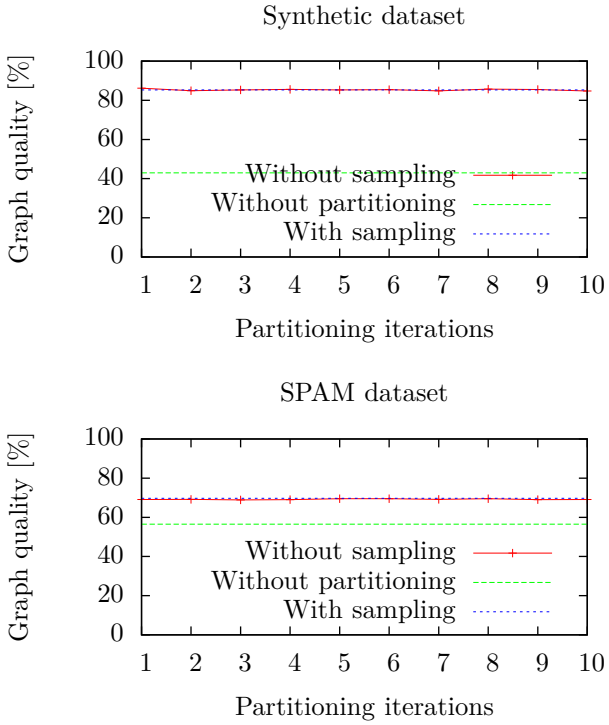

Figure 7. Influence of the partitioning

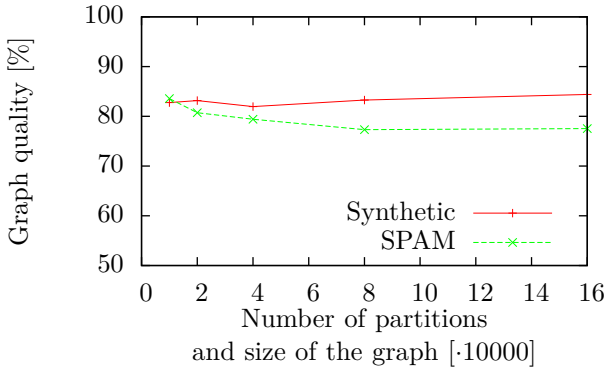

Figure 8. Scalability

H. 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 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.

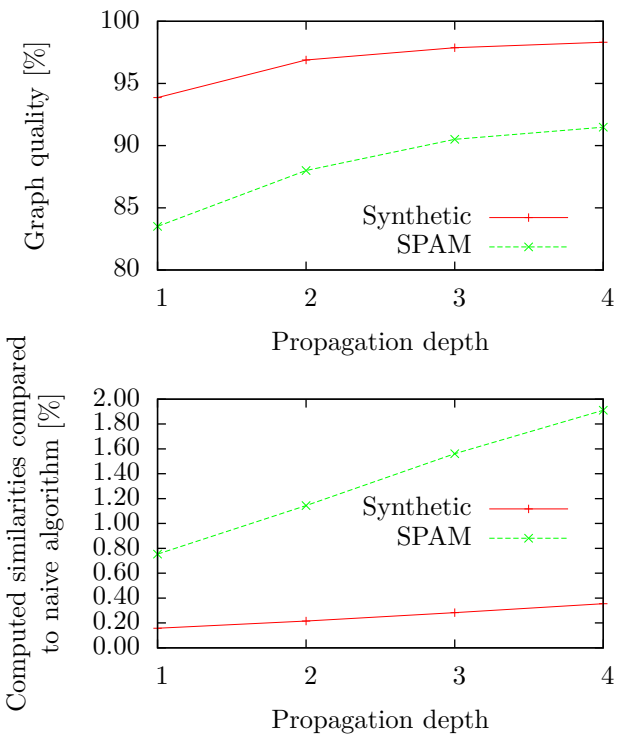

Figure 9. Removing nodes

I. 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 2.

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.

VIII. Conclusions and future work

In this paper 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.

As a future work, we plan to study the auto-evaluation of the expansion and medoids update interval parameters.

References

[1] Tim Althoff, Adrian Ulges, and Andreas Dengel. Balanced Clustering for Content-based Image Browsing. *Artificial Intelligence*, pages 1–4, 2008.

[2] 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.

[3] 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.

[4] Wei Dong. Kgraph. http://www.kgraph.org/, 2014.

[5] 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.

[6] 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.

[7] Gabriel Kliot. Streaming Graph Partitioning for Large Distributed Graphs Categories and Subject Descriptors. *Acm Kdd*, pages 1222–1230, 2012.

[8] Mikko I Malinen and Pasi Franti. Balanced K-Means for Clustering. *Structural, Syntactic, and Statistical Pattern Recognition*, 8621(February):32–41, 2014.

[9] 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.

[10] 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.

[11] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition, Third Edition*. Academic Press, Inc., Orlando, FL, USA, 2006.