A Simple, Efficient, Parallelizable Algorithm for Approximated Nearest Neighbors

Sebastián Ferrada[1], Benjamin Bustos[1], and Nora Reyes[2]

[1] Institute for Foundational Research on Data
Department of Computer Science
University of Chile
Santiago, Chile
{sferrada,bebustos}@dcc.uchile.cl

[2] Departamento de Informática
Universidad Nacional de San Luis
San Luis, Argentina
nreyes@unsl.edu.ar

Abstract. The use of the join operator in metric spaces leads to what is known as a similarity join, where objects of two datasets are paired if they are somehow similar. We propose an heuristic that solves the 1-NN self-similarity join, that is, a similarity join of a dataset with itself, that brings together each element with its nearest neighbor within the same dataset. Solving the problem using a simple brute-force algorithm requires $O(n^2)$ distance calculations, since it requires to compare every element against all others. We propose a simple divide-and-conquer algorithm that gives an approximated solution for the self-similarity join that computes only $O(n^{\frac{3}{2}})$ distances. We show how the algorithm can be easily modified in order to improve the precision up to 31% (i.e., the percentage of correctly found 1-NNs) and such that 79% of the results are within the 10-NN, with no significant extra distance computations. We present how the algorithm can be executed in parallel and prove that using $\Theta(\sqrt{n})$ processors, the total execution takes linear time. We end discussing ways in which the algorithm can be improved in the future.

1 Introduction

There are many problems in multiple domains that require similarity algorithms to be solved: Multimedia Retrieval, Pattern Recognition and Recommendation of Products to name a few, rely on different similarity notions, especially on similarity join problems. A similarity join between two datasets X and Y can be defined as a set of pairs, $S = X \bowtie Y$ where $(x, y) \in S$ iff $x \in X, y \in Y, x \neq y$ and x is similar to y. Similarity joins can be as costly as $O(n^2)$ distance computations when every object has to be compared against all others in the dataset, hence the interest in proposing new indexing methods and efficient algorithms.

The notion of similarity can vary through applications, however, metric spaces are often used as a similarity criterion, therefore if two elements are at a close distance in the given space, they can be seen as similar. A metric space is a pair $(\mathbf{U}, \delta)$, where $\mathbf{U}$ is a universe of objects and $\delta : \mathbf{U} \times \mathbf{U} \to \mathbf{R}^+$ is a metric function (i.e., a non-negative function that is reflexive, symmetric and holds the triangle inequality). Metric spaces are useful since they allow the use of many indexing

techniques and algorithms that take advantage of the triangle inequality to discard objects from comparison when performing similarity joins.

The final question to address regarding similarity joins is "how many results should be returned?" There are mainly 2 strategies to determine the number of results given a dataset $\mathbf{D} \subseteq \mathbf{U}$. One of them is that given a threshold r, for each element $x \in \mathbf{D}$ obtain all $u \in \mathbf{D}$ such that $\delta(x, u) < r$, which is called a range search. The other approach is to find the k nearest neighbors of each element i.e., the k other elements in $\mathbf{D}$ at the closest distances.

The problem we want to address is the generation of similarity graphs, which is a problem of self-similarity join. A similarity graph is a set of vertices, one for each element of the dataset, where two elements from it share an edge if they are part of the result of the join. Particularly we will work with the IMGpedia Dataset [4], a linked dataset that provides visual features and a similarity graph for the images of the Wikimedia Commons dataset. The similarity graph of IMGpedia is the result of a 10-NN self-similarity join; it has 15 million vertices (images) and 150 million edges (similarity relations), and was built using the Fast Library for Approximated Nearest Neighbors (FLANN) [9] that automatically chooses an efficient, scalable technique to solve the similarity join.

In this work we propose a simple heuristic that computes an approximation for the 1-NN self-similarity join using only $O(n^{\frac{3}{2}})$ distance computations and can be naturally parallelizable. We test the effectiveness of the algorithm using a subset of the visual descriptors provided by IMGpedia, by comparing it against other state-of-the-art algorithms. We also show how the algorithm can be modified in order to improve its effectiveness while maintaining the upper bound of distance computations.

2 Related Work

There are many approaches to solve the similarity join problem in sub-quadratic time, both approximated and exact, that take advantage of different properties of the domain of the data. For example, there are techniques that use prefix filtering for string datasets or tokenized documents, such as web pages, to efficiently find near-copies[1,2,15]. They select a prefix of each object and prune object pairs whose prefixes have no overlap. There are others that use specific properties of the distance involved in the similarity computation; Xiao et al. [14] propose the ed-join using constraints on the edit distance, computing thus lower bounds in order to discard elements from comparison. On the other hand, set similarity joins can be addressed efficiently using inverted indexes [12].

However, we are interested in the similarity join problem in metric spaces. This problem has been addressed mainly by indexing one or both datasets, where the cost of building the index is sub-quadratic and the cost of querying, sub-linear. Classic multidimensional indexes are the R-Tree [6] for range queries and the k-d tree [10] for k-NN search, which divide the metric space so that complete regions of it are discarded from comparison. These kinds of multidimensional indexes do not perform well in high-dimensional spaces since we must eventually find the most discriminating dimensions, which is not trivial and also because the elements of the dataset may not always be vectors (e.g., words in a dictionary) hence multidimensional techniques do not apply. There are indexing techniques that attempt to overcome this problem, such as the ϵ-tree, which is similar to a R-tree, but when it has to split when a node is too full, it takes the i-th dimension and divides

the vectors into equally sized intervals of the dimension, overcoming the problems stated and performing better than R-trees [13]. Vantage Point Trees are similar to k-d trees, but instead of making orthogonal hyperplanar cuts to space, they take vantage points to define a sense of near and far points and making large spherical cuts; this change is an improvement for k-d trees in nearest neighbor retrieval [16].

There are other indexing techniques that make use of clustering properties instead of tree-like structures, such as in the case of the D-Index that uses a ρ-split clustering technique, which finds exclusion regions of the space for each cluster and later uses a pivot-based filtering within the cluster. It is suitable for both range queries and nearest neighbor search and is proven to perform better than tree-based approaches [3]. Another cluster-based approach is the List of twin clusters that indexes both of the datasets using a clustering algorithm and later finds matches between the clusters of the datasets, so it discards from comparison all elements from clusters that do not have a twin on the other side [11].

Quickjoin [8] is an algorithm for range similarity joins that does not require a multidimensional index and instead adapts techniques used in distance-based indexing in a method that is conceptually similar to the Quicksort [7] algorithm. Quickjoin recursively partitions the metric space using a marked object, called pivot, so that it divides the data into two sets: the objects closer to the pivot and those that are further to the pivot. The process continues until each partition contains fewer than c objects; then a nested-loop is performed in the subset and the recursive call returns; finally all the results are merged. There is a problem when objects that should be contained in the result end up assigned to different partitions which is addressed by keeping window partitions between every division so it is ensured that every pair of objects at ϵ distance are compared. In [8] it is shown that Quickjoin outperforms many index based approaches and other algorithms that do not use indexes. They prove that Quickjoin takes on average $O(n(1+w)^{\log n})$ comparisons, where w is the fractional average size of the window partitions (and is dependent on ϵ); however, as per Quicksort, its worst case can take $O(n^2)$ time. Fredriksson and Braithwaite [5] give an improved version of Quickjoin, proposing multiple changes: using pivot-based joins instead of nested-loops, using unbalanced partitions, and making it probabilistic. They also describe a way in which k-NN similarity joins can be computed using Quickjoin.

3 The Algortithm

The proposed algorithm computes an approximation for an 1-NN self-similarity join and is based on a divide and conquer design. The algorithm receives a set of points $\mathbf{D}$ as input, with size n and outputs a set of pairs $S \subset \mathbf{D} \times \mathbf{D}$ where every element of the set is paired with some other element expected to be its first nearest neighbor or a good approximation if not.

In Algorithm 1 we show the pseudo-code of the proposed algorithm, which works as follows: first we select a set of centers among the elements of the input; second, the rest of the elements are partitioned in different groups, one group per center; third, within each group we compute the brute-force algorithm to obtain the 1-NN for each of the elements, which is shown in Algorithm 2; and finally, all partial results are aggregated and returned.

It is pending to address the strategies for picking the centers and for partitioning the data. For simplicity, we choose to select a random set of M elements as centers and to partition the rest of the elements into groups of size n/M. An element

Algorithm 1: Algorithm for approximated 1-NN self-similarity join.

```
  Input: Data, a set of objects
  Output: result, a set of pairs of objects
1 centers ← select_centers(Data);
2 groups ← partition(Data, centers);
3 result ← φ ;
4 for group ∈ groups do
5 |  partial ← brute_force_1NN_join(group);
6 |  result ← result ∪ partial ;
7 end
8 return result
```

Algorithm 2: Brute-force 1-NN similarity join.

```
   Input: Data, a set of n objects
   Output: R, a set of pairs of objects
1  D[i] ← ∞ ∀i, 0 < i < n ;
2  R[i] ← φ ∀i, 0 < i < n ;
3  for o_i ∈ Data do
4  |  for o_j ∈ Data, i < j do
5  |  |  if δ(o_i, o_j) < D[i] then
6  |  |  |  D[i] = δ(o_i, o_j);
7  |  |  |  R[i] = (o_i, o_j)
8  |  |  end
9  |  end
10 end
11 return R
```

is assigned to a group, preferably to the group of the closest center, if not, to the second closest. If the second closest group is also full, we keep repeating with the next closest group until a group with room for the element is found. In the worst case, the element will be assigned to the furthest group, hence computing the distance from the point to all the centers. The pseudo-code for the partition algorithm can be found in Algorithm 3.

3.1 Complexity Analysis

The most costly operation of these kind of algorithms is the computation of distances between objects. The number of distance computations of our algorithm depends strongly on the number and size of the formed groups. We can see that the partition algorithm computes M distances per element in the worst case, giving a total of $O(n \cdot M)$ computations. On the other hand, the quadratic 1-NN join of Algorithm 2 computes $(n/M)^2$ distances. If we use $M = \sqrt{n}$ we can prove the following theorem.

Theorem 1. *The proposed heuristic for 1-NN self-similarity join computes $O(n^{\frac{3}{2}})$ distances.*

Proof. Selecting random $\sqrt{n}$ centers does not involve distance computations. As stated before, in the worst case, partitioning the data into $\sqrt{n}$ groups takes $O(n \cdot M)$ distance computations i.e., $O(n \cdot \sqrt{n}) = O(n^{\frac{3}{2}})$. Here we note that sorting the

Algorithm 3: Proposed partition strategy

```
   Input: data, the set of elements; centers, the list of centers
   Output: groups, a list of sets that partition the data
1  groups[i] ← {centers[i]}, ∀i, 0 < i < |centers|;
2  maxSize ← |data|/M;
3  for obj_i ∈ data do
4  |   D ← φ ;
5  |   for center_j ∈ centers do
6  |   |   D ← D ∪ δ(obj_i, center_j);
7  |   end
8  |   sort(D[i]);
9  |   bestGroup ← the group of the center in D[0];
10 |   repeat
11 |   |   if not isFull(bestGroup) then
12 |   |   |   bestGroup ← bestGroup ∪ {obj_i};
13 |   |   else
14 |   |   |   bestGroup ← the group of the center in next(D);
15 |   |   end
16 |   until obj_i is added to a group;
17 end
```

distances to all centers does not involve distance computations. The brute-force join computes $(n/\sqrt{n})^2 = n$ distances per group, as there are $\sqrt{n}$ groups, there are $n \cdot \sqrt{n} = n^{\frac{3}{2}}$ distance computations. Therefore, the algorithm computes $O(n^{\frac{3}{2}})$ distances in total.

The drawback of this algorithm is that, when the groups start filling, the probability that an object and its 1-NN are assigned to different groups increases, hence the precision of the algorithm decreases. One way to handle this issue is to increase the size of the groups. To do so without changing the upper bound of distance calculations, we can enlarge the group size in a constant factor. We show later that using groups of size $2\sqrt{n}$ improves notably the efficacy of the algorithm without a high impact on the number of computed distances.

In practice, the running time can be decreased if instead of performing a brute-force self-similarity join in each group, other techniques are applied. For example, using a pivot-based approach or building a sub-quadratic construction time index in order to discard objects from comparison leads to a lower number of computed distances. However, the cost of distributing the objects in groups still has the complexity upper bound of $O(n^{\frac{3}{2}})$ which dominates the cost no matter how much the similarity joins are reduced. Nevertheless, such an optimization can diminish the total execution time.

The algorithm can be naturally executed in parallel. If p processors are used, each one can distribute n/p elements through the groups computing nM/p distances. Afterwards, the processors can execute Algorithm 2 in parallel, one group each, computing $O((n/M)^2)$ distances per processor. We will use this statements to prove the following theorem:

Theorem 2. *The proposed parallel algorithm computes $O(n)$ distances per processor when using $\Theta(\sqrt{n})$ processors.*

Proof. Since $p = \Theta(\sqrt{n})$, the partition of the objects takes $nM/p = n\sqrt{n}/\Theta(\sqrt{n}) = \Theta(n)$ distance computations for each processor. The brute-force part of the algorithm takes $O((n/\sqrt{n})^2)) = O(n)$ distance computations per processor. Therefore, the overall process takes $O(n)$ distance computations per processor, thus providing linear speedup, i.e., the speedup is equal to the number of processors used, hence having an efficiency of $O(1)$.

4 Experimental Results

We test the effectiveness and efficiency of the proposed algorithm, comparing the resulting pairs of objects and the number of computed distances with the ones produced by the brute-force algorithm (which we use as a ground truth), and with the approximated version of Quickjoin proposed by Fredriksson and Braithwaite [5][3]. We choose Quickjoin, since it is the best algorithm (that does not use indexing) for similarity joins that was found [8] and that can be adapted to find nearest neighbors. The important thing about comparing against a non-indexing method is that both the data and the distance functions are not set beforehand and can be chosen at the moment by the system or the user, which is relevant when resolving similarity queries on multimedia databases. The data used for the experiments are 928,276 feature vectors of 192 dimensions, taken from the IMGpedia Dataset [4], corresponding to the Histogram of the Orientations of the Gradient, which are available in the IMGpedia project site[4]. We run the experiments on a machine with Debian 4.1.1, a 2.2 GHz 24-core Intel® Xeon® processor, and 120 GB of RAM. The proposed algorithm, the brute-force algorithm and Quickjoin, along with the experiments were implemented in Python and are publicly available at `https://github.com/scferrada/self-sim-join`.

We run two experiments. The first of them was comparing the result of our algorithm against the brute-force ground truth in order to measure the precision of the algorithm in two settings, one with groups of size $\sqrt{n}$ and another with size $2\sqrt{n}$. Since performance of the algorithm depends on the order of the data, such that if the objects cannot be assigned to the best fitting group this can result in 1-NN relations to be lost, we run the algorithm 100 times and compute the average precision. We also compute the real position of the found match in order to quantify the error of the algorithm. In Table 1 we show a histogram where each bin indicates if the algorithm's response was: the 1-NN, within the 10-NN, between 10-NN and 50-NN, between 50-NN and 90-NN, and beyond 90-NN. There we see that in only 14.6% of the cases we retrieved the correct answer, while 54% of the time, the answer was within the 10-NN. In Table 2 we show the same results, for the modified algorithm that uses groups with the double of size. We see that the precision is doubled respect to the original algorithm, reaching 30.9% with 79% of the answers within the 10-NN. In both tables we also show the 95% confidence interval for the true mean cumulative value for the amount of elements within the 10-NN.

Making Quickjoin probabilistic, means that it can miss pairs of the result, but all retrieved pairs are correct. Because of time restrictions, and the complexity of

[3] We use the following parameters: $c = 5000$, the number of points to stop the recursion; $\epsilon = 0.5$, the error margin for the probabilistic algorithm; $k = 10$ the number of elements used in the pivot-based join.

[4] `http://imgpedia.dcc.uchile.cl`

Table 1: Histogram of the actual ranking of the results of the algorithm

k-NN	Frequency	Percentage	Cumulative	95% Confidence interval
1	135,222	0.145670	0.145670	0.1450 - 0.1463
10	365,805	0.394069	0.539739	0.5380 - 0.5412
10 - 50	275,478	0.296763	0.836502	0.8340 – 0.8386
50 - 90	57,080	0.061490	0,897992	0.8951 – 0.9004
> 90	94,678	0.101993	1	–

Table 2: Histogram of the actual ranking of the results of the modified algorithm

k-NN	Frequency	Percentage	Cumulative	95% Confidence interval
1	282,045	0.303837	0.303837	0.2990 - 0.3090
10	443,395	0.477654	0.781491	0.7710 - 0.7901
10 - 50	150,188	0.161792	0.943283	0.9341 – 0.9526
60 - 90	14,798	0.015941	0,959224	0.9499 – 0.9686
> 90	14,414	0.015527	1	–

the implemented solution[5] we could only run Quickjoin over a sample of the 10% of the data (92,827 vectors), and we found that after 20 executions, the average fraction of retrieved results was 76% with an average execution time of 93 minutes.

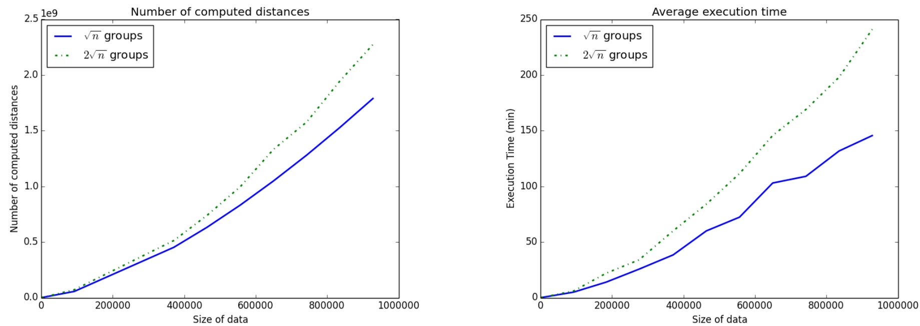
In Figure 1a we show the number of distance computations for the algorithms used: the proposed algorithm and its modified version. Since the brute-force algorithm is known to be quadratic, we omitted its curve so the difference between the other methods could be appreciated. We could not obtain the distances computed for Quickjoin for all sizes because of the problems stated before; however, for the subset where Quickjoin was computed it took three times more distance computations (163,902,044) compared with our algorithm (56,584,929). The extra distance computations are because in order to make Quickjoin finding k-NN it must run two times. We run 100 times the heuristics, and show the average number of distances. The experiments were executed using a 10% of the data, later the 20% and so on, until using the full dataset. There we can see that indeed the modified algorithm computes more distances than the original algorithm; however the difference is a fair trade-off considering that the precision increases to almost double.

5 Conclusions and Future Work

We have presented a simple approximated algorithm for solving the 1-NN self-similarity join problem that computes $O(n^{\frac{3}{2}})$ distances and can be naturally parallelizable in such a way that if $O(\sqrt{n})$ processors are used it computes $O(n)$ distances per processor. Such kinds of processing can enable running the algorithm over millions of objects in linear time using a GPU. We remark the simplicity of the algorithm, since it makes the implementation easier (which is also provided in a public repository) and requires only one parameter (the size of the groups) in comparison with other algorithms or indexing techniques that can be bug prone, hard to understand, and require many parameters.

We showed in experimental results that with a minimal modification and at a reasonable extra cost in terms of distance computations, our algorithm achieves

[5] The implemented solution copies the data of the different partitions on each recursive call, making the operative system kill the process when using larger samples of the data.



(a) Number of distances computed versus the size of the dataset.

(b) Execution time in minutes versus the size of the dataset.

Fig. 1: Experimental results of the algorithm

30.9% precision and 79% of the results are within the 10-NN. Compared with Quickjoin, we found that our algorithm is simpler to implement (short operations, few parameters, easy to understand) and requires less memory. A comparison in terms of precision can be unfair, since Quickjoin does not retrieves results for all elements, but all pairs retrieved are correct [5]. Nevertheless, we think our approach is more useful in practice, it takes less distance computations, it does not have a quadratic worst case and gives a good approximation for the complete result.

As future work we plan study if the goal precision can be set beforehand, finding a fitting function on the size of the groups or finding an optimal value for the group size. Along these lines, it would be interesting to find an analytical expression for the probability to find an exact match and therefore give an approximation bound. We plan on finding if the precision can be improved by using some heuristics for choosing evenly distributed centers for the groups, and maybe heuristics on the assignment of the elements in the groups, for example, swapping a new element with other already assigned in the group if such swap diminishes the radius of the group. We will also run benchmarks on the execution time of the algorithm, using other join techniques, rather than the brute-force approach, for example, when all distances to one element are computed, said element can be used as a pivot to prune further pairs from comparison.

We also plan on modifying the algorithm so that it supports k-NN similarity joins so we can compare the output of this algorithm against the whole similarity graph of IMGpedia. Finally we plan to propose metrics that can enable the comparison of the quality of different similarity graphs.

Acknowledgments This work was supported by the Millennium Institute for Foundational Research on Data and CONICYT-PFCHA/2017-21170616.

References

1. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th international conference on World Wide Web. pp. 131–140. ACM (2007)

2. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: Data Engineering, 2006. ICDE'06. proceedings of the 22nd International Conference on. IEEE (2006)
3. Dohnal, V., Gennaro, C., Savino, P., Zezula, P.: D-index: Distance searching index for metric data sets. Multimedia Tools and Applications 21(1), 9–33 (2003)
4. Ferrada, S., Bustos, B., Hogan, A.: IMGpedia: a linked dataset with content-based analysis of Wikimedia images. In: International Semantic Web Conference. pp. 84–93. Springer (2017)
5. Fredriksson, K., Braithwaite, B.: Quicker similarity joins in metric spaces. In: International Conference on Similarity Search and Applications. pp. 127–140. Springer (2013)
6. Guttman, A.: R-trees: A dynamic index structure for spatial searching, vol. 14. ACM (1984)
7. Hoare, C.A.: Quicksort. The Computer Journal 5(1), 10–16 (1962)
8. Jacox, E.H., Samet, H.: Metric space similarity joins. ACM Transactions on Database Systems (TODS) 33(2), 7 (2008)
9. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. VISAPP (1) 2(331-340), 2 (2009)
10. Ooi, B.C., McDonell, K.J., Sacks-Davis, R.: Spatial kd-tree: An indexing mechanism for spatial databases. In: IEEE COMPSAC. vol. 87, p. 85. sn (1987)
11. Paredes, R., Reyes, N.: Solving similarity joins and range queries in metric spaces with the list of twin clusters. Journal of Discrete Algorithms 7(1), 18–35 (2009)
12. Sarawagi, S., Kirpal, A.: Efficient set joins on similarity predicates. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data. pp. 743–754. ACM (2004)
13. Shim, K., Srikant, R., Agrawal, R.: High-dimensional similarity joins. In: Proceedings of the 13th International Conference on Data Engineering. pp. 301–311. IEEE (1997)
14. Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. Proceedings of the VLDB Endowment 1(1), 933–944 (2008)
15. Xiao, C., Wang, W., Lin, X., Yu, J.X., Wang, G.: Efficient similarity joins for near-duplicate detection. ACM Transactions on Database Systems (TODS) 36(3), 15 (2011)
16. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: SODA. vol. 93, pp. 311–321 (1993)