Internship Report

GoldFinger, a Fast and Precise method for approximating Jaccard similarity

26. April - 23. July 2021

Guilhem Niot
Département Informatique Fondamentale
ENS de Lyon

Supervised by
Anne-Marie Kermarrec
EPFL, Lausanne, Switzerland

Olivier Ruas
Inria, Lille

François Taiani
IRISA, Rennes

Erick Lavoie
EPFL, Lausanne, Switzerland

Contents

As part of my first year of Master at ENS de Lyon, I had to do a 12-weeks internship abroad. I'm extremely grateful to Anne-Marie Kermarrec, professor at EPFL and director of the SaCS (Scalable Computing Systems) laboratory, for her supervision, and for warmly welcoming me in her laboratory. I also would like to express my deepest appreciation to Olivier Ruas, Postdoctoral fellow at Inria (Lille, France), and to François Taiani, Professor at the University of Rennes 1 and IRISA/Inria in Brittany, for co-supervising me and for joining Anne-Marie and I every week for a remote meeting where we discussed the results obtained and where I could head next. Our discussions were always both pleasant and enlightening. I would like to extend my gratitude to Erick Lavoie, Research Scientist at SaCS, EPFL. Erick supervised me at the end of my internship on an additional project about small-world connectivity using Raspberry Pis described in Appendix B. I worked on this side project to stay busy when I had to wait for long experiments to terminate on my main topic. I also had great pleasure working with all the members of the SaCS laboratory, who were very welcoming and with whom I had the chance to exchange regularly either on-site or during the weekly team meeting.

While my internship started well remotely, I was able to go on-site for six weeks from the middle of June. I really enjoyed being on-site, as it allowed me to appreciate the good humor in the lab and to meet interesting people. It also allowed me to discover how scientists actually organize and communicate about their work.

1 Introduction

K-Nearest Neighbors (KNN) graphs are widely used for recommendation systems [21, 7], for web duplicates detection [26] and for machine learning prediction models [27]. KNN graphs are directed graphs connecting nodes to their K nearest neighbors according to a given similarity metric.
However, their computation does not scale well to datasets with millions or billions of entries.

In order to make the computation scale, several approaches try to approximate the exact KNN graph by computing an Approximate Nearest Neighbors (ANN) graph. Those approaches often offer to tune their algorithm to obtain a tradeoff between quality and computation time adapted to the application.
We can observe two main leverages to reduce the computation time of ANN graphs:

- **Reducing the number of comparisons of pairs**. The computations of similarities are costly and limiting them often pays off.
- **Approximating the computation of the distance between two elements**. This approximation can be done by performing dimensionality reduction on the dataset: each entry of the dataset is reduced into a vector of small dimension while preserving as much as possible the distances between the different entries. For instance, Spotify [3] uses sparse matrix factorization to reduce the dimension of the set of listened songs of its users before computing the KNN graph on the transformed dataset.

A lot of research has been published on reducing the number of similarity computations, with graph-based methods [15, 12, 7, 23] (successive improvements of a random KNN graph), hash based methods [19] (hash functions which hash similar elements to the same value with high probability), and further improvements seem complicated.
Surprisingly, there has been less interest in approximating the computation similarity while it still represents a significant part of the computation time.

In this regard, my supervisors developed an approximation of the Jaccard similarity metric (when given two sets S_1 and S_2, $Jaccard(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$) called GoldFinger [17], and the goal of my internship was to compare it to other techniques for approximating Jaccard similarity in terms of

speed and closeness to the true value, to benchmark it on real datasets in combination with other techniques, and to explore possible improvements.

During my internship, I implemented state-of-the-art techniques for approximating Jaccard, and I analyzed the compromise between performance and quality of each of these techniques. The conclusion was that GoldFinger was still competitive against them for computing KNN graphs. Then, I applied GoldFinger to different problems relying on the Jaccard similarity, and I showed that GoldFinger was providing interesting performance improvements at a low implementation cost. This step was the occasion to contribute major changes to KNN libraries, described in Appendix A. Finally, I studied whether GoldFinger quality could be further improved, and I proposed and analyzed several strategies to fit the method to the data. One of the three strategies analyzed has interesting results and improves significantly the quality on datasets with non-linear frequency.

In link with my internship, I opened **7 pull requests** to acknowledged libraries, totaling **565 line additions, and 187 line deletions**. In addition, I also wrote **thousands of lines of code** on repositories dedicated to this internship (5636 lines in `https://gitlab.aliens-lyon.fr/gniot/SamplingKNN`, 1019 lines in `https://github.com/GuilhemN/ann-benchmarks/tree/PAPER`, and 1345 lines in `https://github.com/GuilhemN/nmslib/tree/FASTSIM`).

Those results will be included in a journal version of the conference version of Goldfinger [16] which will be submitted to Transactions on Knowledge and Data Engineering Journal (TKDE).

1.1 Problem statement

We first define what a similarity function is, as it is central to the definition of KNN graphs and of the problems we will study in this report.

Definition (Similarity function)**.** *Noted*

$$\begin{aligned} sim : U \times U &\longrightarrow \mathbb{R}^+ \\ (u, v) &\mapsto sim(u, v) \end{aligned}$$

It can be seen as the inverse of a distance, it takes large values when u and v are similar, and values near 0 when they are dissimilar.

There are plenty of similarity functions used in the literature such as cosine[1], or based on Euclidean distance, or Hamming distance[2].
Since GoldFinger is used to approximate the Jaccard similarity, we will focus on the Jaccard similarity in the rest of this document.

Definition (Jaccard similarity function)**.** *Given two sets S_1, S_2, we define:*

$$Jaccard(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

It is a value between 0 and 1, and the closer to 1, the more similar the 2 sets.

Besides, there are actually several close but different problems studied on datasets with similarity functions.

Definition (K-nearest neighbor graph)**.** *Given a dataset of points U, we want to compute the oriented K-regular graph where each node $u \in U$ is linked to the K nodes $v_1, ..., v_K \in U$ with largest values $sim(u, v_i)$. Those nodes are called the neighbors of u.*

[1] `https://en.wikipedia.org/wiki/Cosine_similarity`
[2] `https://en.wikipedia.org/wiki/Hamming_distance`

Definition (K-nearest neighbor search)**.** *Given a dataset of points U, we want to build an index such that we can efficiently compute the K-nearest neighbors on subsequent queries:*

$$search(index(U), q_i) = K \text{ nearest neighbors of } q_i \text{ in } U$$

Definition (Similarity Join)**.** *Given a dataset of points U and a threshold t, we want to find all the couples $(u_1, u_2) \in U^2$ such that $sim(u, v) \geq threshold$.*

All these problems have relaxed variants (for the KNN problems, they are called ANN for Approximate Nearest Neighbors) where we drop the exactitude constraint on the result and instead try to achieve a trade-off between efficiency and quality metrics. For instance, for the KNN problems, we may try to keep a high proportion of true nearest neighbors, while for Set Similarity Join, we may try to retrieve a high proportion of the pairs having a similarity above the threshold.

The main purpose of the our research is to decrease the computation time while maintaining a high quality on the results.

1.2 How to evaluate the quality of ANN graphs

Since the ANN graphs are not exact, we need to define metrics which capture how good the approximations of the exact graphs are. Those metrics will be used to evaluate the trade-off between the computation time and the quality provided by the ANN approaches.

Definition (Recall)**.** *A widely used metric in the literature is the recall. It is the proportion of correct nearest neighbors in the ANN graph:*

$$Recall = \frac{|\{(u, v) \in ANN \cap KNN\}|}{|KNN|}$$

The closer it is to 1, the better.

The recall is however too restrictive for some applications using KNN graphs where we only need to have similarity values close to their optimal values, and not the exact edges.
We can thus also adopt a relative error metric:

Definition (Quality)**.**

$$quality = \frac{\sum_{(u,v)\in ANN} sim(u, v)}{\sum_{(u,v)\in KNN} sim(u, v)}$$

The closer it is to 1, the better.

1.3 Approximating Jaccard similarity

Computing the exact Jaccard similarity of two sets is expensive when the sets contain tens or hundreds of items. To speed it up, an idea is to produce a vector of small dimension, called a sketch, representing the user's set, and from which the Jaccard similarity can be approximated at a low cost.

To approximate the Jaccard similarity, a sketching technique first defines a function producing sketches:

$$produce\_sketch : \mathcal{P}(\text{items set}) \to \mathbb{R}^{\text{small dimension}}$$

This function is applied to the whole dataset during its loading, and then approximate similarities are computed using the sketches produced:

$$\widehat{Jaccard}(produce\_sketch(S_1), produce\_sketch(S_2)) \approx Jaccard(S_1, S_2)$$

We will present three sketching techniques for Jaccard similarity: MinHash, Fast Similarity Sketching and GoldFinger.

1.3.1 MinHash Similarity Sketching

Min-wise Hashing [9] is the most popular sketching technique for approximating Jaccard similarity. It is based on the fact that when sorting two similar sets by fixing an order on items, the sorted sets are likely to have the same minimal element. More formally, given a random permutation on the set of items π, and two sets S_1, S_2 to be compared, we have

$$\mathbb{P}\left(\min_{s_1 \in S_1} \pi(s_1) = \min_{s_2 \in S_2} \pi(s_2)\right) = J(S_1, S_2)$$

We leverage this fact to define a sketching technique using l permutations of the items set noted $\pi_1, ..., \pi_l$:

$$produce\_sketch(S) = (min_{s \in S} \pi_i(s))_{1 \leq i \leq l}$$

and,

$$\widehat{Jaccard}(sketch_1, sketch_2) = \frac{\sum_{i=1}^{l} \mathbb{1}_{sketch_{1,i} = sketch_{2,i}}}{l}$$

However this version has at least two drawbacks :

- It is costly to draw random permutations, to store them and to apply them to each set of the dataset.
- It still takes a lot of space to store the minimal items of each set.

Several works tried to overcome these limitations, with two main strategies:

- By reusing permutations several times [29, 30].
- By keeping only some bits for each minimal ID, and compensating the statistical bias introduced [9, 22, 26].

1.3.2 Fast Similarity Sketching

Fast similarity sketching [11] was designed to alleviate the expensive pre-processing cost of MinHash. Instead of sampling with replacement one item for each element of the sketch, as MinHash does, Fast Similarity Sketching mixes sampling with replacement and sampling without replacement. For a sketch of length t, Fast Similarity Sketching uses $2t$ random hash functions that are used to assign the items to a position in the sketch and to a random value. The first t hash functions randomly assign items to positions while the last t hash functions deterministically assign items to a set position –one position per hash function– to ensure that every position has at least one item associated to it. At each position of the sketch, the minimum value of the associated items is stored. To improve the performances, the sketch is filled with items one hash function at a time and the process is stopped whenever every position has at least one value.
We can describe this algorithm with pseudo-code, using $2t$ hash functions $h_1, ..., h_{2t}$: items set $\rightarrow \{0, ..., t\} \times [0, 1]$:

The similarity using Fast Similarity Sketching is then estimated in a similar way as MinHash: by counting the proportion of equal values in the sketch, position-wise:

$$\widehat{Jaccard}(sketch_1, sketch_2) = \frac{\sum_{i=1}^{l} \mathbb{1}_{sketch_{1,i} = sketch_{2,i}}}{l}$$

Constructing a sketch of length t from a set S has a complexity of $t+|S|$, against $t \times |S|$ for MinHash. Albeit its lower complexity, Fast Similarity Sketching is not as compact as densified variations of MinHash such as b-bit Minwise Hashing [22] as it stores floats instead of bits.

Algorithm 1 Pseudo-code of the function *produce_sketch* of Fast Similarity Sketching

```
1: function produce_sketch(S)
2:     sketch ← +∞^t
3:     counter ← 0                ▷ Count the number of positions filled in the sketch to stop early
4:     for i ∈ {1, ..., 2t} do
5:         for item ∈ S do
6:             b, v ← h_i(item)
7:             if i > t then      ▷ The hash functions t, ..., 2t make sure the sketch is entirely filled
8:                 b ← i − t
9:             end if
10:            if sketch[b] = ∞ then
11:                counter ← counter + 1
12:            end if
13:            sketch[b] = min(sketch[b], v)
14:        end for
15:                                                          ▷ early return
16:        if counter = t then return sketch
17:        end if
18:    end for
19: end function
```

1.3.3 GoldFinger

GoldFinger is a sketching technique that was designed with efficiency and simplicity in mind. It leverages a technique called feature hashing, consisting in transforming a set of features into the set of the hashes of these features, allowing to efficiently approximate the properties of the initial set, to drastically accelerate similarity computations on sparse datasets (few items per set compared to the number of items available).
Using a hash function $h : I \mapsto [0, B-1]$, we produce a binary sketch of size B called *Single Hash Fingerprint* (SHF for short):

$$\forall i \in [0, B-1], produce\_sketch(S) = SHF(S)_i = \begin{cases} 1 & \text{if } \exists e \in S \text{ such that } h(e) = i \\ 0 & \text{otherwise} \end{cases}$$

When there are few collisions in this bit array, it was experimentally verified [17] that we can approximate the cardinal of a user set:

$$||S|| \approx ||SHF(S)||_1$$

And we can approximate the intersection of two user sets,

$$||S_1 \cap S_2|| \approx ||SHF(S_1) \textbf{ AND } SHF(S_2)||_1$$

From these observations arise the following approximation for Jaccard similarity, dubbed GoldFinger:

$$\widehat{Jaccard}(SHF(S_1), SHF(S_2)) = \frac{||SHF(S_1) \textbf{ AND } SHF(S_2)||_1}{c_1 + c_2 - ||SHF(S_1) \textbf{ AND } SHF(S_2)||_1}$$

where $c_1 = ||SHF(S_1)||_1$, $c_2 = ||SHF(S_2)||_1$.

A very interesting property of GoldFinger is that when consider the set of positions of the ones in the GoldFinger sketch, we obtain another formula for the approximation:

$$\widehat{Jaccard}(SHF(S_1), SHF(S_2)) = Jaccard\left(\{i \mid SHF(S_1)_i = 1\}, \{i \mid SHF(S_2)_i = 1\}\right)$$

Thus, by replacing the item sets of the dataset with the set of positions of the ones of the dataset sketches, GoldFinger allows to perform dimensionality reduction on the dataset without changing the original problem: the goal is to retrieve the same edges, and the similarity metric is unchanged. A direct implication is that GoldFinger can be applied as a preprocessing layer to any KNN algorithm using the Jaccard similarity to improve its performance, without any modification to the underlying algorithm.

1.4 Datasets

We based our evaluations on six publicly available datasets. To apply Jaccard's index, we binarize each dataset by only keeping in a user's profile P_u those items that user u has rated higher than 3.

Movielens Movielens [18] is a group of anonymous datasets containing movie ratings collected online between 1995 and 2015 by GroupLens Research [28]. The datasets (before binarization) contain movie ratings on a 0.5-5 scale by users who have at least performed 20 ratings. We use 3 versions of the dataset, movielens1M (ml1M), movielens10M (ml10M) and movielens20M (ml20M), containing between 575,281 and 12,195,566 positive ratings (i.e. higher than 3).

AmazonMovies AmazonMovies [24] (AM) is a dataset of movie reviews from Amazon collected between 1997 and 2012. We restrain our study to users with at least 20 ratings (before binarization) to avoid users with not enough data (this problem, the *cold start problem*, is generally treated separately [20]). After binarization, the dataset contains 57,430 users; 171,356 items; and 3,263,050 ratings.

DBLP DBLP [32] is a dataset of co-authorship from the DBLP computer science bibliography. In this dataset, both the user set and the item set are subsets of the author set. If two authors have published at least one paper together, they are linked, which is expressed in our case by both of them rating each other with a rating of 5. As with AM, we only consider users with at least 20 ratings. The resulting dataset contains 18,889 users, 203,030 items; and 692,752 ratings.

Gowalla Gowalla [10] (GW) is a location-based social network. As DBLP, both the user set and the item set are subsets of the set of the users of the social network. The undirected friendship link from u to v is represented by u rating v with a 5. As previously, only the users with at least 20 ratings are considered. The resulting dataset contains 20,270 users, 135,540 items; and 1,107,467 ratings.

2 Computing KNN graphs with GoldFinger

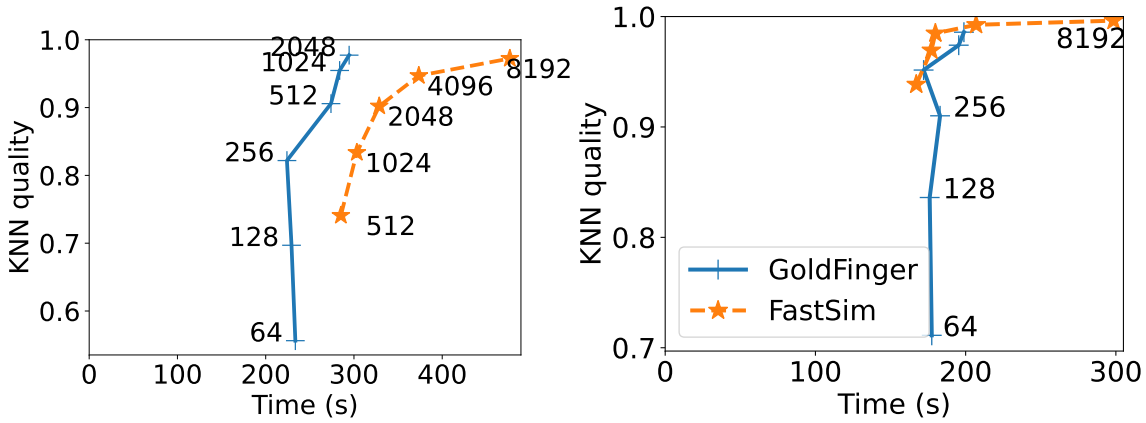
In this section, we compare GoldFinger to the sketching techniques MinHash and Fast Similarity Sketching on the KNN graph problem. During my internship, I showed that GoldFinger remained competitive against the most recently published state-of-the-art sketching technique Fast Similarity Sketching.

2.1 MinHash and b-bit minwise hashing

GoldFinger was compared to MinHash and b-bit MinHash [17]. GoldFinger significantly outperforms both by providing a better compromise KNN quality versus computation time (excluding initialization time).
Furthermore, the initialization cost of MinHash is prohibitive in practice, and makes it unsuitable for large datasets.

Table 1: Preprocessing time for the native approach, 2048 bits Fast Similarity Sketching (with $t = 64$) and 1024 bits GoldFinger. GoldFinger is significantly faster but Fast Similarity Sketching is competitive.

Dataset	**Native**	**FastSim**	**GolFi**	speedup/ FastSim (×)
ml10M	5.27s	6.45s	4.01s	1.6
AM	2.68s	3.65s	2.12s	1.7
DBLP	0.27s	0.47s	0.21s	2.3



(a) movielens10M (b) AmazonMovies

Figure 1: Relation between the computation time and the KNN quality for different sketch sizes (in number of bits) using Fast Similarity Sketching and GoldFinger.

2.2 Fast Similarity Sketching

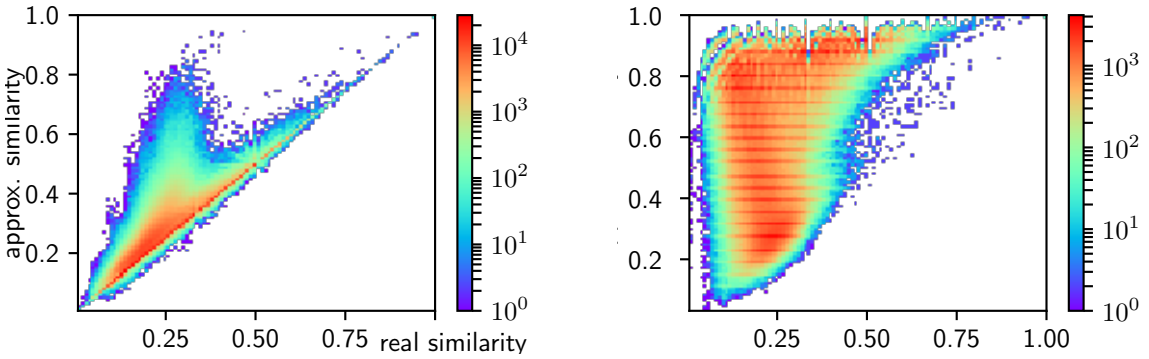
In order to compare Fast Similarity Sketching to GoldFinger, I first implemented Fast Similarity Sketching in Java to compare it on the same ground as the Java implementation of GoldFinger [17]. My work on Fast Similarity Sketching was then divided into two parts: evaluating the performance of GoldFinger and Fast Similarity Sketching on the different datasets studied, and trying to explain these results.

The experiments[3] were run on an Intel Xeon E5-2630@2.40GHz using 8 cores (out of the 16 available cores) with 125GB of memory.

Table 1 summarizes the time required to construct the sketches for Fast Similarity Sketching, GoldFinger and the speed-up of GoldFinger compared to Fast Similarity Sketching on movielens10M, AmazonMovies and DBLP. For most of the experiments, we used Fast Similarity Sketching with $t = 64$ as it was the best trade-off between quality and computation time. As each stored value is encoded as a float, the resulting sketches were $64 \times 32 = 2048$ bits long. On the other hand, we used 1028 bits GoldFinger as it was as well a trade-off promise between quality and computation time. During the preprocessing step, GoldFinger is significantly faster than Fast Similarity Sketching (×1.6) for the initialization part. Still, Fast Similarity Sketching remains an acceptable competitor as the preprocessing step is negligible compared to the total KNN graph computation.

Figure 1 shows how the trade-off between the computation time and the KNN quality evolves when we increase the size of the sketches when computing KNN graphs with the brute force approach on movielens10M and AmazonMovies. The size ranges from 64 to 2048 bits for GoldFinger and from 512 to 8192 for Fast Similarity Sketching, which corresponds to a number of hash functions ranging from 16 to 256. The results are averaged over four runs. On movielens10M, GoldFinger significantly outperforms Fast Similarity Sketching while on AmazonMovies Fast Similarity Sketching

[3]Code is available at `https://gitlab.aliens-lyon.fr/gniot/samplingknn`



(a) GoldFinger (1024 bits)

(b) Fast Similarity Sketching (2048 bits)

Figure 2: Estimated similarity in function of the real similarity on movielens10M using 2048 bits Fast Similarity Sketching and 1024 bits GoldFinger

seems slightly better, but the differences are negligible. This difference can be explained by how GoldFinger behaves depending on the dataset: GoldFinger is more efficient on denser datasets. By increasing artificially the density of a dataset –by randomly merging items– the KNN quality increases for both approaches but GoldFinger widen the gap with Fast Similarity Sketching and is the clear winner for denser datasets.

These results show that not only GoldFinger is competitive against one of the most up-to-date sketching approach but it clearly outperforms it for denser datasets.

We also analyzed the difference in results between these two sketching techniques.

Figure 2 shows the repartition of the estimated similarity in function of the true similarity on ml10M for both 1024 bits GoldFinger and 2048 bits Fast Similarity Sketching. We observe a poorer approximation from Fast Similarity Sketching on this dataset, which is much further from a linear expectation and explains the better quality obtained with GoldFinger.

3 KNN search and set similarity join with GoldFinger

As the original paper presenting GoldFinger focused on its impact on KNN graph computation, I studied whether GoldFinger could also improve state-of-the-art algorithms for KNN search and Set Similarity Join.

In spite of intense research in the field of KNN search, I showed that using GoldFinger could significantly improve the performance of state-of-the-art algorithms with little development efforts. I also obtained promising results for Set Similarity Join that could be investigated further.

3.1 Leveraging GoldFinger to improve state-of-the-art KNN search algorithms

The KNN search problem is a fundamental problem related but different from the KNN graph problem.

In the KNN graph problem, the k closest neighbors are searched for every user in the dataset. The graph can be built user by user or can be built as a whole. In that setting, the computation time of both the sketches and the KNN graph are paramount performance metrics.

On the other hand, the KNN search problem aims at finding the k closest neighbors for users which are not known beforehand and which may not be part of the initial set of users. In this case,

the main metric for KNN queries is the number of queries that can be answered per second. Complex indexes and datastructures can be built offline, once and for all, to speed up the KNN queries.

For instance, KNN search can be used to produce recommendations for giant datasets. To limit the cost, it is possible to extract a small part of the dataset at random, and to find the k nearest neighbors of a user at runtime in this smaller dataset. This limits the size of the index and speeds up the process.

3.1.1 Evaluation performance of KNN search implementations

The implementation used in [17] for benchmarking KNN graph algorithms was not adapted to the KNN search problem. Indeed, benchmarking KNN search algorithms require techniques closer to those used by the ML community, with the separation into training and test sets, and the graph structure is less apparent.

I decided to leverage the benchmarking tool ANN-benchmarks [5] for this study. It aims at comparing ANN querying algorithms, by comparing state of the art implementations. The idea to compare implementations without worrying about them being implemented in the same language, with equal conditions, is that users care about the actual performance they will obtain.

However, before I used it, its support of the Jaccard similarity was limited and very inefficient making it infeasible to benchmark KNN algorithms on large datasets such as movielens10M. This was the occasion to contribute some improvements to ANN-benchmarks described more precisely in Appendix A.1.

3.1.2 Impact of using GoldFinger for KNN search

In this section, we evaluate the impact of the use of GoldFinger for KNN search.

ANN-benchmarks evaluates algorithms with parameters optimized empirically by the libraries authors corresponding to different compromises between quality and performance. For the Jaccard Similarity metric, ANN-benchmarks[4] compares the following implementations: NMSLIB [8], PyNNDescent [25, 12], Datasketch [13] and PUFFINN [4].

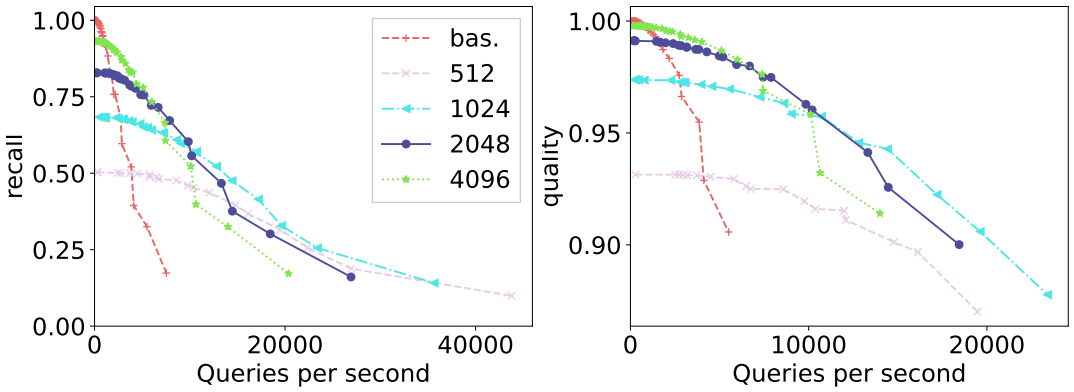
In the following I focus on the implementation of HNSW (Hierarchical Navigable Small World graphs) [23] from NMSLIB as it significantly outperforms the other competitors in all our experiments, in both its base version, and the version with a GoldFinger preprocessing. The library NMSLIB was adopted by Amazon [1] and HNSW was implemented at Facebook [2]. ANN-benchmarks provide a list of good parameters configurations for the implementation HNSW in NMSLIB: this list of configurations will be used for all our experiments.

For optimal performances, I implemented GoldFinger directly inside NMSLIB[5], in C++. During my testing, I observed that the Python interface exposed by NMSLIB was not optimized and that it could be drastically improved (performance improved up to a factor 10). This was another occasion to contribute to a public library, described in Appendix A.2.

Figure 3 shows the trade-off between quality of the resulting queries, expressed as recall and KNN quality, and the performances, expressed as the number of queries per second, for the different parameters configurations provided by NMSLIB for HNSW on the dataset movielens10M. We plot both results in HNSW original form (dubbed bas.), and with a pre-treatment to compute GoldFinger with a size ranging from 512 to 4096. **The use of GoldFinger provides an important speed-up of the queries while maintaining a similar level of quality.**

[4]The version I used is available at `https://github.com/GuilhemN/ann-benchmarks/tree/PAPER`. It leverages some improvements I contributed to the NMSLIB library.

[5]The version I used is available at `https://github.com/GuilhemN/nmslib/tree/PAPER`. It integrates GoldFinger in the NMSLIB library itself.



(a) Recall (b) KNN quality

Figure 3: Relation between the quality metrics and the number of queries per second in function of the parameter configuration on the dataset movielens10M. Using GoldFinger highly increases the number of queries per second at the expense of a slight decrease in quality.

This shows that the use of GoldFinger is not limited to KNN graph computation but can be extended to other problems where the Jaccard similarity is used, such as KNN search.

Remark. *While we wanted to compare the impact of Fast Similarity Sketching on NMSLIB to the impact of GoldFinger, I was not able to have a working implementation in time. The version available at* `https://github.com/GuilhemN/nmslib/tree/FASTSIM` *ships a C++ version of Fast Similarity Sketching I developed but the quality metrics are inexplicably low and we thus decided to not include them in the paper produced nor in this report.*

3.2 GoldFinger and Set Similarity Join

Another interesting problem that GoldFinger applies to is Similarity Join. An application of this problem is to find duplicates and merge similar sets in a dataset.

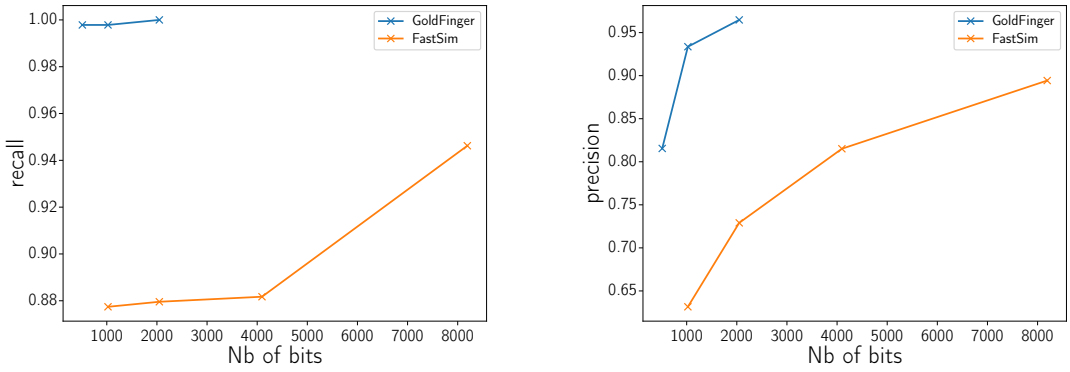
Researchers recently showed [6] that, when comparing exact and approximate algorithms for Similarity Join, the exact algorithm was the fastest on most datasets. However, the algorithms compared limit their optimizations to reducing the number of similarity computations, without approximating the similarity itself.

While I did lack time to evaluate whether we could obtain a performance gain for Similarity Join using GoldFinger, as GoldFinger would not have been sufficient used alone and it would have required also reducing the number of similarity computations to compete with other methods, I showed that using GoldFinger allowed to obtain a high-quality approximation of the set $\{(u, v) | sim(u, v) \geq threshold\}$, and was performing better than Fast Similarity Sketching at it.

3.3 Quality metrics for Set Similarity Join

Benchmarking this problem is different than for KNN problems in that each node does not have a fixed number of neighbors, so we need to define two adapted evaluation metrics, depending on whether we care about false positives.

Definition (Recall)**.** *The recall for this problem differs from the recall used for KNN graphs. Given the set* $S = \{(u, v) \mid sim(u, v) \geq threshold\}$*, and an approximation of it noted* $\widehat{S}$*, we define the recall*



(a) Recall (b) Precision

Figure 4: Quality of the approximate Similarity Join computed by GoldFinger and Fast Similarity Sketching on DBLP. GoldFinger achieves high quality with a low number of bits, and outperforms Fast Similarity Sketching.

as:

$$recall(\widehat{S}) = \frac{|S \cap \widehat{S}|}{|S|}$$

This definition ignores false positives, it only takes into account the retrieval of the actual edges.

Definition (Precision)**.** *Given the set* $S = \{(u, v) \mid sim(u, v) \geq threshold\}$*, and an approximation of it* $\widehat{S}$*, we define the recall as:*

$$precision(\widehat{S}) = \frac{|S \cap \widehat{S}|}{|S \cup \widehat{S}|}$$

This definition penalizes false positives. The approximated set must not include too many additional elements to keep a high precision.

3.4 Impact of using GoldFinger for Set Similarity Join

The protocol I adopted was to compute the approximated sets

$$\widehat{S}_1 = \{(u, v) \mid GoldFinger(u, v) \geq threshold\}$$

and

$$\widehat{S}_2 = \{(u, v) \mid FastSim(u, v) \geq threshold\}$$

on the datasets using a brute force method before computing their recall and precision.

Figure 4 shows our results on DBLP. **GoldFinger achieves both a high recall and precision with a low number of bits, and significantly outperforms Fast Similarity Sketching.** We obtain similar results on movielens10M and Gowalla.

This can be explained by the fact that this problem is very sensible to errors on the similarity value, when only keeping the relative order between elements was important for KNN graphs. While GoldFinger both provides a precise approximation of the similarity value and keeps the relative order between elements, Fast Similarity Sketching performs well at the former, but provides a poor absolute approximation of the similarity value as observed on Figure 2.

We thus obtained promising preliminary results on this problem, and it could be interesting to further investigate the performance gain obtained by plugging GoldFinger into existing Set Similarity Join methods. The strength of GoldFinger is that it would only require a pre-treatment layer replacing

the item sets with the sketchs produced by GoldFinger, just like for KNN search, without any change to the underlying algorithms. In comparison, the integration of other sketching techniques such as Fast Similarity Sketching or MinHash, would be much more complicated as many Set Similarity Join algorithms do not compute the Jaccard similarity directly, and rather do their own computations on the item sets. The source code of [6], available at `http://ssjoin.dbresearch.uni-salzburg.at` could be used as basis for this evaluation.

4 Improving GoldFinger by fitting the data for better quality

While constructing GoldFinger, we studied the choice of the hash function used. Indeed, GoldFinger assigns a hash to each item of the dataset, and two items with the same hash will induce errors in the computation of the approximated similarity. For instance, if two frequent items obtain the same hash, they will collide often and create a significant number of errors and thus decrease the KNN graph quality. The original paper [17] observed that using the Jenkins hash function[6], SHA-512[7] or a modulo as hash function resulted in a similar quality. Indeed, on real datasets, these functions behave as if hashes were randomly chosen for each item.

Replacing the random hash function by a function which is adapted to the data should lower the impact of those unavoidable collisions.

Intuitively, collisions occur when two items share the same hash so reducing the total frequency of the items associated to a hash should thus reduce the number of collisions. We consider a new hashing strategy, dubbed AdaptHash, which is based on the frequency of the items. The items are greedily assigned one by one by decreasing frequency to the least used hash. This is a $\frac{4}{3}$-approximation [14] of the classical scheduling problem where we minimize the makespan of scheduling the items on p processors based on their frequency. Hence, this strategy is efficiently distributing the item hashes while maintaining near the optimal the value $\max_{b\in[0,B-1]} \sum_{item|h(item)=b} \text{frequency}(item)$.

Concerning computation time, this new hashing strategy mainly impact the initialization time, during which the sketches are computed, as the method to estimate the similarity remains unchanged. The initialization time remain negligible compared to the KNN graph computation time. On movielens10M, using AdaptHash doubles the initialization time, from 4s to 8s (independently of the sketch size), which is negligible compared to the computation time of the KNN graph (110s with Hyrec).

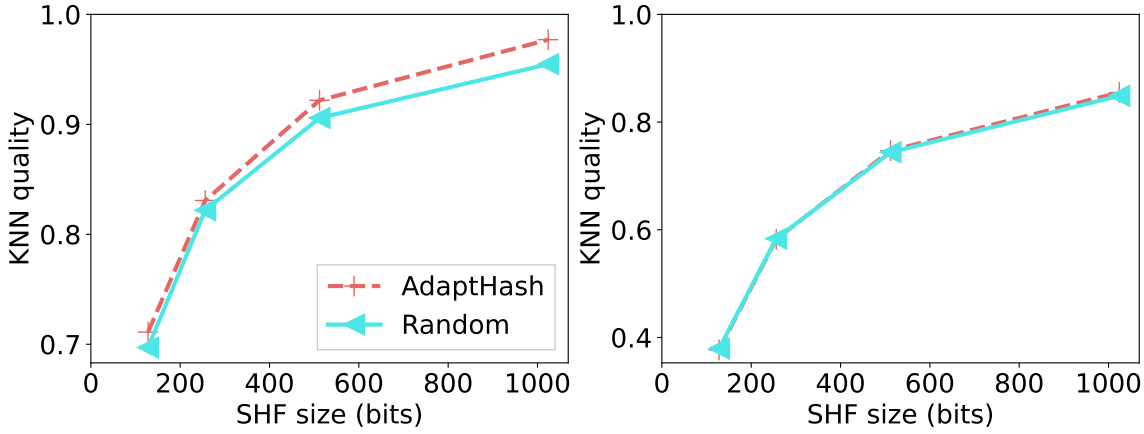
Figure 5 shows the impact of the hashing strategy on the KNN graph quality on movielens10M and Gowalla. **The use of AdaptHash provides a better KNN quality compared to the use of a random hash function, especially on movielens10M where the gain is substantial.** Similar results were obtained on AmazonMovies, ml1M, ml20M and DBLP: the AdaptHash strategy outperforms the use of a random hash function.

The gain compared to the use of a random hash function is tightly bounded to the frequency distribution of the items. Figure 6 shows the frequency distribution of the datasets movielens10M and Gowalla. On movielens10M, where the gain is important, the frequency is not linear, there are items which are much more frequent than others while the frequency is linearly distributed among items in Gowalla, resulting in a slight improvement in terms of KNN quality. AdaptHash is particularly interesting when the frequency distribution is not uniform.

Remark. *We experimented a second strategy for choosing the item hashes consisting in dedicating part of the sketch to the most frequent items. This strategy reserves k bits for the k most frequent items. The other items are hashed at random on the remaining bits. The intuition was that most errors are due to hash conflicts with frequent items and eliminating these conflicts should improve the quality.*

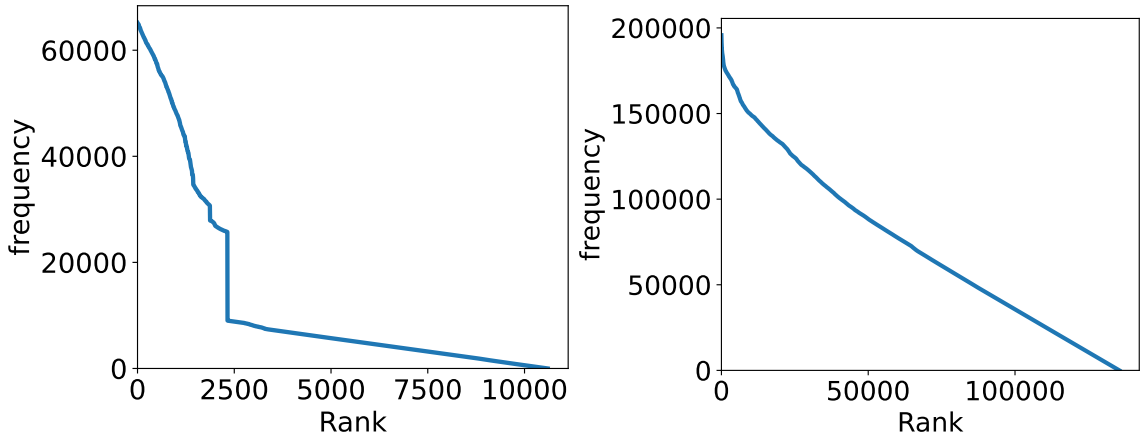
However, it was not performing predictably and was sometimes improving, and other times worsening the KNN graph quality, so we do not analyze it in depth here and focused our work on AdaptHash.

[6] `https://en.wikipedia.org/wiki/Jenkins_hash_function`
[7] `https://en.bitcoinwiki.org/wiki/SHA-512`



(a) movielens10M (b) Gowalla

Figure 5: Evolution of KNN quality with a random hash function and AdaptHash for GoldFinger. AdaptHash outperforms the use of a random hash function.



(a) movielens10M (b) Gowalla

Figure 6: Frequency distribution of movielens10M and Gowalla

5 Conclusion and possible future work

During this internship, I showed that GoldFinger remained competitive against the later released Fast Similarity Sketching. I have also illustrated the effectiveness of GoldFinger for KNN search, and the significant improvement it provides to state-of-art algorithms performance, which have not leveraged similarity approximation so far. Additionally, I showed that GoldFinger outperforms Fast Similarity Sketching for Set Similarity Join, in terms of quality. Complementary work could be realized to determine whether the performance gain obtained with GoldFinger balances the loss in quality, when using GoldFinger alongside state-of-the-art Set Similarity Join algorithms. Along with these results, I showed that GoldFinger quality could be slightly increased by using Adaptative hashing functions, with little increase in the initialization time.

The results obtained during this internship will be presented in a paper submitted to the Transactions on Knowledge and Data Engineering journal (TKDE).

A Improving existing KNN libraries

As I used existing libraries to answer KNN queries and to benchmark the impact of GoldFinger, I encountered a few issues and implementations that could be improved. It was the occasion to contribute to previous research papers and to improve their implementation.

I contributed to ANN-benchmarks the support of sparse datasets, and the correction of the support of Jaccard similarity with the ANN libraries NMSLIB and PUFFINN [4]. Besides, I contributed to NMSLIB a refactor of its C-to-Python interface that drastically improves the performances for several metrics, including Jaccard similarity.

A.1 Improving the support of Jaccard similarity in ANN-benchmarks

ANN-benchmarks [5] is a benchmarking tool aiming at drawing a fair comparison between KNN search libraries, and to provide people with a way to determine which algorithm and parameters suit their application best.

It supports the Angular, Euclidean and Jaccard metrics. Yet, its support of the Jaccard similarity was quite limited prior to my contributions due to the lack of support of sparse datasets, and several algorithms integrations being broken.

The support of an adapted format for sparse datasets is paramount for the Jaccard similarity, given the size of the datasets used in practice, a dense format is in the best case terribly inefficient, and in the worse simply not possible with the actual technologies. The density is the ratio between the number of items possessed by the users, and the product of the number of users and the number of items, and it is proportional to the ratio between the size of a dataset stored in sparse format, and of the size of its dense representation. For instance, Movielens10M has a density of 1%, and DBLP even has a density of 0.02%!

Due to the importance of supporting sparse datasets, I integrated their support in the library and contributed it back in `https://github.com/erikbern/ann-benchmarks/pull/235`. It reduced the size of the dataset archive provided by the library for evaluating Jaccard similarity from 2GB to just 33MB!

I also fixed the support of the PUFFINN [4] library with Jaccard Similarity[8], and I added the support of the Jaccard similaritty with NMSLIB in another pull request[9].

In total, I opened **6 pull requests on ANN-benchmarks, totaling 435 line additions, and 78 deletions**.

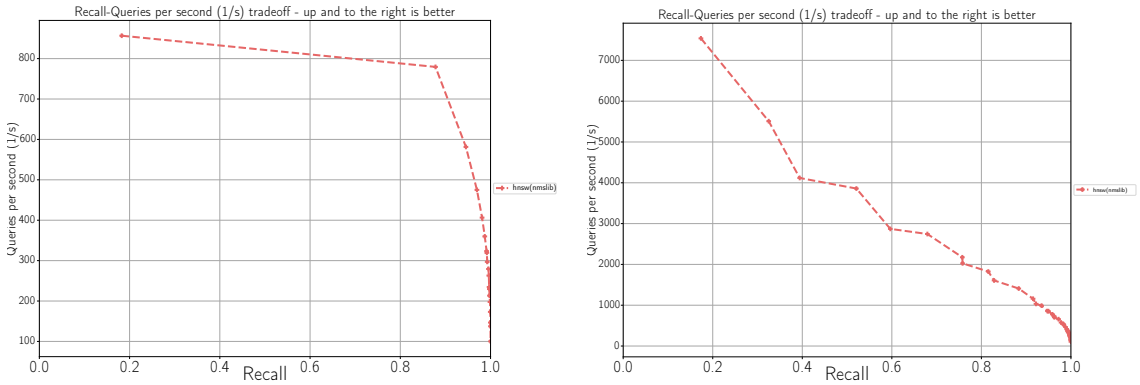
A.2 Speeding up the KNN search library NMSLIB

NMSLIB is a major library answering to the KNN search problem. Amazon AWS has dedicated offers implementing it for its users. And NMSLIB ranks among the best KNN search algorithms according to ANN-benchmarks[10].

It is a generic library implementing different KNN search algorithm (BallTree, SW-graph, HNSW), as well as different metrics (Jaccard, Hamming, Cosine, Euclidean, etc).

However, its implementations of the different metrics are not all as performant. Indeed it exposes a python interface where the user retrieves the data, and for some metrics like Euclidean or Cosine, the data are passed directly to the C++ program, but for some others, like Jaccard or Hamming, the data are first converted to string format before being passed to C++, where they are converted again to integers or floats.

[8] `https://github.com/erikbern/ann-benchmarks/pull/234`
[9] `https://github.com/erikbern/ann-benchmarks/pull/239`
[10] See their website `http://ann-benchmarks.com/`.



(a) Before (b) After

Figure 7: Measurement of the improvement of the performance of NMSLIB on movielens10M. For clarity reasons, we only show results for HNSW, these improvements also apply to the other algorithms provided by NMSLIB. We observe a performance gain of a factor up to 10.

While it simplifies the codebase by avoiding the need for managing polymorphic C++ classes, it is highly inefficient.

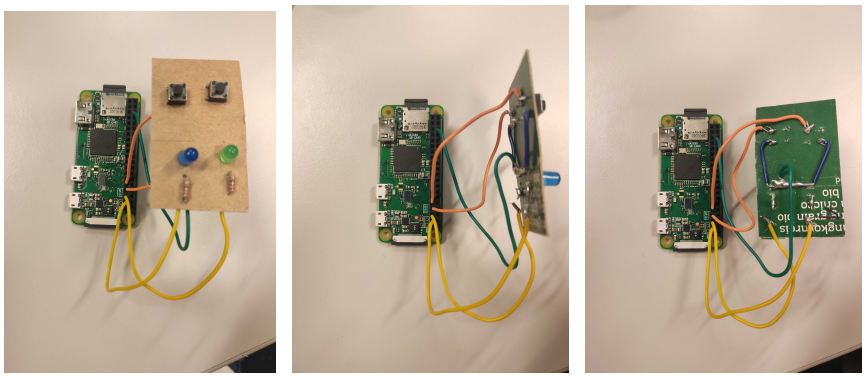
In the scope of my internship, I wanted to measure the impact of GoldFinger on NMSLIB with the minimum of noise possible, and I wanted to diminish the impact of using the Python interface. Thus, I worked on passing the raw data directly from Python to C++ to avoid unecessary conversions and opened a pull request at `https://github.com/nmslib/nmslib/pull/484`. Despite being minimalist as it only includes the changes to the C++ interfaces and leaves the changes to the metric classes separate, this pull request already adds **130 lines of code, and deletes 109 others**.

Figure 7 shows the performance improvement obtained with this change on movielens10M, with the Jaccard similarity metric. We observe an impressive gain in the performances: up to a factor 10. This improvement makes the performance gap between NMSLIB and other libraries even greater for the Jaccard similarity.

B Side project on small-world networks and peer-to-peer application

While not directly related to the initial subject of my internship, as I had to wait for experiments to terminate at the end of my internship, I worked part time with Erick Lavoie, Research Scientist at SaCS, and with Paulette Vazquez, summer Intern at SaCS, on integrating new applications into a small-world connectivity project using Raspberry Pis.

The observation at the root of this project is that it is difficult to create applications such as social networks or file sharing for local communities, and decentralized networks due to the multitude of device technologies involved, with the need to define a communication protocol for every pair of devices (Android-Windows, Android-iOS, iOS-Linux, and so on...). And it should be simpler to use an intermediary device, in our case a Raspberry Pi, to limit the number of possible interaction to those of the form $\forall device\_type, device\_type$ - Raspberry Pi. Each individual could then have their own Raspberry Pi and use it as an intermediary to communicate with other people, with the additional advantage that the Raspberry Pi can be used as a replicating node when the user is offline and cannot directly share their data.



(a) (b) (c)

Figure 8: Pictures of the assembled Raspberry Pi

This project was separated into two main tasks. First, replicating and clarifying the report[11] of the semester project of a bachelor student at EPFL, Romain Kuenzi, who designed the physical interface of the Raspberry Pis and implemented a first decentralized application using them. Then, I integrated another application onto the Raspberry Pi called Hypercore[12]. It is a peer-to-peer (P2P) protocol allowing to share data on a decentralized network, it provides high-level interfaces to simplify the development of P2P applications, and among them, Hyperdrive, a P2P file-sharing application, which we will cover more precisely.

B.1 Using Raspberry Pis to simplify small-world network connectivity and application to Scuttlebutt, a decentralized social network

The semester project of Romain Kuenzi aimed at designing a physical interface to connect one's device to a Raspberry Pi, and showcasing the usage of Raspberry Pis to simplify the usage of Scuttlebutt [31] on local networks. Scuttlebutt is a decentralized gossip platform, providing social networking features. Data spread as users copy the data of friends and of friends of friends they encounter in the network (when they meet another user), hence the term gossip.

Paulette and I were able to successfully replicate the assembled device Romain designed (see Figure 8), and its usage to communicate with the decentralized social network Scuttlebutt.

We then extended Romain's report at `https://gitlab.epfl.ch/sacs/ssb/smallworld/report-romain` to clarify the setup procedure, and we documented additional procedures to connect the Raspberry Pi with Android phones, and Windows computers.

On top of that, Romain noted in his report that the user was lacking feedback to determine when the synchronization between their device and the Raspberry Pi was finished. In order to improve this point, I created a script[13] listening to Scuttlebutt events and alerting the user when new content is received.

B.2 Integrating the Hypercore protocol with small-world Raspberry Pis

For this second part of the project, I wanted to show that Hyperdrive could be used in a small-world setup to easily share files that could be often updated among peers, similarly to a Dropbox instance.

[11] The report of Romain Kuenzi on SmallWorld is available at https://gitlab.epfl.ch/sacs/ssb/smallworld/report-romain/-/tree/1c3c37bea3949826313e34cdaeec5f9a4d85ba55

[12] `https://hypercore-protocol.org/`

[13] The script I created to listen to Scuttlebutt publish events is available at `https://github.com/GuilhemN/ssb-copy-follows/blob/POSTS`.

To do so, I designed a demonstration where the goal is to synchronize files between two peers, with no simultaneous connexion. To achieve this, the Raspberry Pi acts both as a replicating node of the users' file and needs to be remotely controllable to determine what data is synced.

My report is available at `https://gitlab.epfl.ch/niot/smallworld-dat-report` and explains in detail how to integrate Hypercore on the Raspberry Pi, and how to remotely control it to achieve the desired goal.

References

[1] Build k-nearest neighbor (k-nn) similarity search engine with amazon elasticsearch service. https://aws.amazon.com/about-aws/whats-new/2020/03/build-k-nearest-neighbor-similarity-search-engine-with-amazon-elasticsearch-service/. last accessed July 23, 2021.

[2] Faiss: A library for efficient similarity search and clustering of dense vectors. `https://github.com/facebookresearch/faiss`. last accessed July 23, 2021.

[3] Mohan A. How does spotify's recommendation system work? `https://www.univ.ai/post/spotify-recommendations`.

[4] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. Puffinn: Parameterless and universally fast finding of nearest neighbors. In 27th Annual European Symposium on Algorithms (ESA 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[5] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. Information Systems, 87:101374, 2020.

[6] Christos Bellas and Anastasios Gounaris. An empirical evaluation of exact set similarity join techniques using gpus. Information Systems, 89:101485, 2020.

[7] Antoine Boutet, Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Rhicheek Patra. Hyrec: Leveraging browsers for scalable recommenders. In Proceedings of the 15th International Middleware Conference, pages 85–96, 2014.

[8] Leonid Boytsov and Bilegsaikhan Naidan. Engineering efficient and effective non-metric space library. In Nieves Brisaboa, Oscar Pedreira, and Pavel Zezula, editors, Similarity Search and Applications, pages 280–293, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[9] Andrei Z Broder. On the resemblance and containment of documents. In Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171), pages 21–29. IEEE, 1997.

[10] Eunjoon Cho, Seth A Myers, and Jure Leskovec. Friendship and mobility: user movement in location-based social networks. In Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 1082–1090, 2011.

[11] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Mikkel Thorup. Fast similarity sketching. In 2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS), pages 663–671. IEEE, 2017.

[12] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In Proceedings of the 20th international conference on World wide web, pages 577–586, 2011.

[13] Zhu E. Datasketch. `https://github.com/ekzhu/datasketch`.

[14] M. R. Garey, R. L. Graham, and D. S. Johnson. Performance guarantees for scheduling algorithms. Operations Research, 26(1):3–21, February 1978.

[15] George Giakkoupis, Anne-Marie Kermarrec, Olivier Ruas, and François Taïani. Cluster-and-conquer: When randomness meets graph locality, 2020.

[16] Rachid Guerraoui, Anne-Marie Kermarrec, Olivier Ruas, and François Taïani. Smaller, faster & lighter knn graph constructions. In Proceedings of The Web Conference 2020, WWW '20, page 1060–1070, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Rachid Guerraoui, Anne-Marie Kermarrec, Olivier Ruas, and François Taïani. Fingerprinting big data: The case of knn graph construction. In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 1738–1741, 2019.

[18] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. Acm transactions on interactive intelligent systems (tiis), 5(4):1–19, 2015.

[19] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In Proceedings of the thirtieth annual ACM symposium on Theory of computing, pages 604–613, 1998.

[20] Xuan Nhat Lam, Thuc Vu, Trong Duc Le, and Anh Duc Duong. Addressing cold-start problem in recommendation systems. In Proceedings of the 2nd international conference on Ubiquitous information management and communication, pages 208–211, 2008.

[21] Justin J Levandoski, Mohamed Sarwat, Ahmed Eldawy, and Mohamed F Mokbel. Lars: A location-aware recommender system. In 2012 IEEE 28th international conference on data engineering, pages 450–461. IEEE, 2012.

[22] Ping Li and Christian König. B-bit minwise hashing. In Proceedings of the 19th International Conference on World Wide Web, WWW '10, page 671–680, New York, NY, USA, 2010. Association for Computing Machinery.

[23] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence, 42(4):824–836, 2020.

[24] Julian John McAuley and Jure Leskovec. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In Proceedings of the 22nd international conference on World Wide Web, pages 897–908. ACM, 2013.

[25] L. McInnes. Pynndescent. `https://github.com/lmcinnes/pynndescent`.

[26] Michael Mitzenmacher, Rasmus Pagh, and Ninh Pham. Efficient estimation for high similarities using odd sketches. In Proceedings of the 23rd International Conference on World Wide Web, WWW '14, page 109–118, New York, NY, USA, 2014. Association for Computing Machinery.

[27] N. Nodarakis, S. Sioutas, D. Tsoumakos, G. Tzimas, and E. Pitoura. Rapid aknn query processing for fast classification of multidimensional data in the cloud. CoRR, abs/1402.7063, 2014.

[28] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In Proceedings of the 1994 ACM conference on Computer supported cooperative work, pages 175–186. ACM, 1994.

[29] Anshumali Shrivastava and Ping Li. Densifying one permutation hashing via rotation for fast near neighbor search. In Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32, ICML'14, page I–557–I–565. JMLR.org, 2014.

[30] Anshumali Shrivastava and Ping Li. Improved densification of one permutation hashing, 2014.

[31] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. Secure scuttlebutt: An identity-centric protocol for subjective and decentralized applications. In Proceedings of the 6th ACM Conference on Information-Centric Networking, ICN '19, page 1–11, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. Knowl. Inf. Syst., 42(1):181–213, 2015.