Scalable Pattern Search Analysis

Eric Sadit Tellez, Edgar Chavez, and Mario Graff

Universidad Michoacana de San Nicolas de Hidalgo, México
{sadit,mgraffg}@lsc.fie.umich.mx,
elchavez@fismat.umich.mx

Abstract. Efficiently searching for patterns in very large collections of objects is a very active area of research. Over the last few years a number of indexes have been proposed to speed up the searching procedure. In this paper, we introduce a novel framework (**the K-nearest references**) in which several approximate proximity indexes can be analyzed and understood. The search spaces where the analyzed indexes work span from vector spaces, general metric spaces up to general similarity spaces.

The proposed framework clarify the principles behind the searching complexity and allows us to propose a number of novel indexes with high recall rate, low search time, and a linear storage requirement as salient characteristics.

1 Introduction

Proximity search, often generalized as similarity search, is present in many fields of computer science such as: pattern recognition, textual and multimedia information retrieval, query by content and classification, machine learning, lossless/lossy data streaming and compression, security (e.g. criminal record databases, biometric identification, biometric authentication, etc.) and bioinformatics, followed by a very large etcetera.

We need a formalization of the problem to continue the discussion. A metric workload is a triple (U, d, S) with U a domain, $S \subseteq_F U$ a finite subset of U, named the database and $d : U \times U \rightarrow \mathbb{R}^+$ a distance function. The distance function obeys the following properties $\forall x, y, z \in U$ (i) $d(x, y) \geq 0$ and $d(x, y) = 0 \iff x = y$, (ii) $d(x, y) = d(y, x)$, and (iii) $d(x, z) + d(z, y) \geq d(x, y)$. These properties are known as strict positiveness, symmetry, and the triangle inequality, respectively.

The two most common search queries in S are *range* and *nearest neighbor* (**NN**) queries. This paper focus on a natural extension of the later, the k-Nearest-Neighbors (**kNN**). Informally speaking the **kNN**$_d(q, S)$ retrieves the k closer objects to the query q in the database S.

Searching **kNN** have a well known linear worst case [1,2,3]. This problem gets worst in the case of high intrinsic dimensional dataset. Under this circumstance, traditional indexing techniques using the triangle inequality such as the families of indexes: *compact partition* and *pivot based* suffer from a condition known as the *curse of the dimensionality* (*CoD*) [2]. The problem comes from the phenomenon of *concentration of the measure*, informally characterized by an histogram of distances with small standard deviation and a large mean. We are not aware of any exact index able to deal with the curse of dimensionality.

A common approach to alleviate the CoD is to use *approximate* proximity search algorithms trading speed for accuracy. One way is to convert an exact algorithm to an approximate one using the procedure described in [4] which consists in aggressively reduce the radius by a stretching constant.

In this work, we present a new framework where a number of approximate indexes can be analyzed and understood. Our framework is simple, yet powerful enough to analyze different approximate indexes that work on different spaces such as: vector spaces, general metric spaces, and similarity spaces.

Furthermore, the simplicity of our approach allow us to propose novel approximate indexes that have a high recall rate, low search time, and a linear storage requirement, among other characteristics. As we will see, these novel indexes are competitive to the point that in the majority of cases tested they have a better performance[1] than previous approaches.

The rest of the paper is organized as follows. In Section 2 we introduce our framework and describe those indexes that can be analyzed under this framework. Section 3 shows the experimental results. A number of conclusions and possible directions for future work are described in Section 4.

2 K Nearest References

Our framework is composed by two functions: an encoding function and a similarity function. Each object u in the database is encoded using its K closer objects from a set of references R, i.e., $\mathbf{KNN}_d(u, R)$ where R is a small subset of objects randomly selected from the database. We call this representation the *K Nearest References* (**KNR**). Proximity between objects is approximate by proximity between encodings of the objects using a similarity function.

Let **encode** be an encoding function, the domain U is converted to the **KNR** space as: $\hat{U} = \{\hat{u} \leftarrow \mathbf{encode}(\mathbf{KNN}_d(u, R)) : u \in U\}$. Similarly $\hat{S}$ denote the encoded database.

Each object $r_j \in R$ can be unequivocally identified using j. We will use j and r_j without distinction when there is no confusion. Also, for notation convenience, we define $\mathbf{KNR}(u) = \hat{u}$. In general, the **KNR** sequences can be encoded as vectors, strings or sets. Each encoding gives a tradeoff between space, search time and accuracy. To fix ideas, think on a string representation and the amount of space needed for the whole database S. We will need at most $O(nK \log |R|)$ bits to store $\hat{S}$.

To complete our framework we need a distance function $d' : \hat{U} \times \hat{U} \rightarrow \mathbb{R}^+$. This function accepts two **KNR** sequences encoded as $\hat{u}$ and $\hat{v}$ and if the corresponding objects are close under d, then the encoded sequences will be close under d'.

To search with a **KNR** method we complete three steps: a) map the query to the **KNR** domain, b) search for the closest γ candidates under d' in the mapped space, and finally, c) verify the candidates using the original distance. Typically, γ is a tuning parameter that optimize the desired performance. That is, it is optimized to a particular data set and **kNN** queries in order to maximize performance.

[1] Here performance is measured as the tradeoff among storage, recall and search time.

2.1 Describing Existing Proximity Indexes Using KNR

The first use of our framework is in describing and analyzing existing indexes. We are aware of four proximity indexes that can fit under our framework. In the next paragraphs, we briefly describe them.

Permutation Based Index (PBI) is the first **KNR** index reported in the literature (see [5]). The idea behind this is to describe an object by capturing the perspective of it. That is, it measures how a set of references (R), called permutants, are seen by each object in the database. The proximity is computed using the relative movements of the permutants. We must notice that $K = |R|$ in PBI. Here $|R|$ is very small, this makes it particularly efficient for expensive distances.

The index uses the inverse of the permutation to create a vector space as follows: given a $\mathbf{KNR}(u) = x_1 x_2 \cdots x_{|R|}$ where $x_i \in R$. We define $\hat{u}$ as the permutation of u. The inverse of $\hat{u}$ is defined as $\hat{u}^{-1}[x_i] = i$, and they are measured with Minkowski's L_1 or L_2 distances (i.e. Spearman Footrule and Spearman ρ, respectively[2]) [5]. Please note that each permutation requires $|R|$ distances, a linear sort $O(|R|)$,[3] and a linear pass to find the inverse. In order to find the candidate list, one needs to perform n permutation distance comparisons (L_1 or L_2) each one of them requires $O(|R|)$ basic arithmetic operations. This yields to a total cost of $O(n|R|)$ basic operations. The space complexity is $n|R| \log |R|$ bits.

Metric Inverted File. It is based on a simplification of the Spearman Footrule [5,6], using only the first K references closer to an object and its positions in the full permutation. This information is used to compute approximately the Spearman Footrule (L_1) distance of the permutations. Indexes are small and fast since $K \ll |R|$.

Since many permutants cannot be found to compute L_1, the blank positions should be filled with a penalization constant ω (e.g. $\omega = |R|/2$). To provide a scalable representation, the authors use an inverted file to represent R (as the *thesaurus*), and list of tuples $(object, position)$ as *posting lists* [6]. A detailed explanation about inverted indexes can be found in [7,8].

The computational cost to represent each object is equivalent to the permutation based index; however, in this case the plain mapping (without inverted index representation) requires $Kn \log K|R|$ bits i.e., each object requires K tuples of $(referenceId, position)$. These tuples are sorted by $referenceId$ (adding an additional sort over K items). The candidate list requires n evaluation of the partial Spearman Footrule distance which requires in the worst case $O(K)$ and in the best scenario $O(1)$. Please note that when the objects are not related the best scenario (or something close to it) is frequently found. This yields to a worst case of $O(nK)$ and a smaller average worst case for obtaining the candidate list. Using an inverted index, requires $Kn \log(Kn)$ bits of space, and the total cost is driven by the cost to obtain the candidate list and $O(\gamma)$ distance computations.

The Brief Permutation Index. As the original PBI, the brief permutation index [9] uses $K = |R|$, but it speeds up searches, in total time, while reducing the space needed

[2] Another option is Kendall τ [5], but its usage it is limited by the high cost of the computation.

[3] In general, if distances cannot be discretized, we need a comparison based sorting, i.e. $O(|R| \log |R|)$.

for the index. The main idea behind the *brief index* is to lossy encode the inverse of the permutation with a single bit per permutant, using the information about how much it moves from its position in the identity permutation. The encoding yields a binary Hamming space of dimension $|R|$.

The codification of each object requires $O(|R|)$ distances plus $O(|R|)$ basic operations in the same way as the permutation based index. $|R|$ bits are needed to represent each object in the mapped space. The total space is then $n|R|$ bits. Assume that w is the size of the computer word then the computational cost of each distance d' is $O(\frac{|R|}{w/2})$ using an additional table of size $2^{w/2}$ to pre-compute hamming.[4]

Prefix Permutations Index: PP-Index. The mapping used by the PP-Index [10] is as follows: for every object $u \in U$ we compute $KNR(u)$. The proximity between objects is measured by the length of the shared prefixes of the corresponding strings, larger shared prefixes means high proximity and short or zero length prefixes reflects low proximity. The strict notion of closeness yields to low recall (ranging from 0.3 to 0.5) [10]. In contrast, it is really fast and can be represented efficiently using a compressed trie data structure [7,8].

In order to overcome the low recall several strategies are possible, increasing search time and memory usage (see [10]). While some of these enriching strategies are common to all **KNR** indexes and we discuss and generalize these strategies in section 3.3 for all **KNR** methods. The basic implementation used in the PP-Index will be referred as **KNR** prefixes.

The computational cost of this index is similar to the metric inverted file explained above; however, the constants involved in this index are lower and the best scenario (for d') is even more frequent in the PP-Index than in the Metric Inverted File.

2.2 Using the KNR Framework to Create Proximity Indexes

Using the described framework, we create seven new indexes. We present them in a classification based on the mapped space, obtaining three main **KNR** mappings: vector spaces, strings, and sets. In general the computational cost of computing the mapping per object is composed by the number of distances to obtain the K nearest references, plus the cost of the **encode** function. The search process has a computational cost of n evaluations of d' (using the plain mapping without an index) plus at most γ evaluations of d. The storage requirements are linked to the particular **encode** function, as well the specific cost of d'.

Vector Space Mappings. In this class of mappings, the target is to obtain vectors from the sequences **KNR**(u). One way to do so is by fixing the attention to the positions of the references in $KNR(u)$, or the actual distances to the references u_i. The permutation index [5], the brief index [9], and the metric inverted index [6] using the Spearman Footrule measure (L_1) are **KNR** vector space mappings. We add the vector space mapping for Spearman ρ (L_2), as a natural variation to L_1. Another possibility is to use the cosine between the $|R|$-dimensional **KNR** vectors having the distances to the K closer

[4] When $w = 32$ this scheme produces tables of 2^{16} entries of 5 bits each one, i.e. $\log(w/2 + 1)$ bits in general. For very large w one needs to divide w in smaller pieces.

references. These methods would have the same computational cost than the Spearman Footrule. The space complexity is $nK(\log |R| + w)$ bits for the **KNR** cosine.

String Mappings. **KNR** sequences are strings in R^K (using R as an alphabet), where $K \ll |R|$. So, each $\hat{u}$ is used as a short string, then objects are compared using distances defined in the string domain. The main point here would be to measure the order of shared references, and how much effort it must be done to convert one string into another.

The PP-Index [10] is a string mapping. We augment the list showing the performance for Levenshtein (edit distance), longest common subsequence (*LCS*), both distances require $O(K^2)$ operations, for detailed description of the distances the reader should see [11]. The **KNR** string mappings uses only the information found by the **KNR** sequence, and a string distance function to predict proximity. The space complexity is $nK \log |R|$.

Set Mapping. Up to the best of our knowledge, there are no **KNR** indexes based on sets in the literature. Our idea is to use the **KNR** sequence as a set. We will keep the same notation for the **KNR**(u) and the encoded version $\hat{u}$. Since each reference can appear only once in **KNR**(u), the main difference with respect to string distances is the lack of order in the representation.

For set mappings we studied three distances, the Jaccard coefficient, the Dice coefficient, and the cardinality of the intersection as similarity measures. We found that set **KNR** mappings produces small and fast indexes with excellent recall.

The Jaccard distance is computed as $d_J(\hat{u}, \hat{v}) = 1 - \frac{|\hat{u} \cap \hat{v}|}{|\hat{u} \cup \hat{v}|}$, the distance gives values between $[0, 1]$ where 0 means equality and 1 means disjointness. The Dice coefficient, $d_D(\hat{u}, \hat{v}) = \frac{2|\hat{u} \cap \hat{v}|}{|\hat{u}| + |\hat{v}|}$, is used in many information retrieval tasks [12,7]. For similarity functions, a zero value means no closeness.

The computational cost is in the same order than the metric inverted file, but simpler, since it does not requires additional operations than the union, i.e., the metric inverted file computes $O(K)$ arithmetic operations to partially compute the L_1, then the constants involved are smaller for this case. The space complexity is smaller too, it is $O(nK \log(|R|))$ bits. As we experimentally prove in the experimental section below, these facts and a higher recall makes the set mappings a better option.

3 Experimental Results

In order to study the behavior of the different **KNR** mapping methods, we performed a series of experiments using as benchmarks three real datasets. The first one (**documents**) is a collection of 25157 short news articles in the $TFIDF$ format from Wall Street Journal $1987 - 1989$ files taken from TREC-3 collection. We use the angle between vectors as distance measure [7] and extracted 100 random documents from the collection as queries (note: these documents were not indexed). Each query searches for 30NN. The objects are vectors of thousand of coordinates. The second benchmark (**vectors**) is a set of 112544 color histograms (112-dimensional vectors) from an image database[5]. We choose randomly 200 histogram vectors and applied a *perturbation*

[5] The original database source is `http://www.dbs.informatik.uni-muenchen.de/~seidl/DATA/histo112.112682.gz`

of ± 0.5 on one random coordinate. The search consists on finding 30NN under L2 distance. The third one (**CoPhIR**) consists of 10 million of objects selected from the CoPhIR project [13]. Each object is a 208-dimensional vector and we use the L_1 distance. Each vector was created in a linear combination of five different MPEG7 vectors as described in [13]. We choose the first 200 vectors from the database as queries. Searches for the 30NN were performed.

The algorithms were implemmented in the C# programming language, running under the Mono framework.[6] The experiments were performed in an Quadcore Intel Xeon 2.33 GHz workstation with 8GiB of RAM, running Ubuntu Linux 8.04. We kept the entire database and indexes in main memory and without exploiting parallel capabilities of the workstation.

3.1 Recall of the Indexes

Figure 1 shows the recall rate when the number of references is varied. The Figure presents three **KNR** mappings: vector, string and set mappings (rows) on two different data sets, namely **documents** and **CoPhIR** (columns). As we can see on figure 1(b) the permutation index and the brief index have a perfect recall for a small $|R|$. The other indexes performed below these two. On the other hand, for **documents** dataset, the lowest performance is presented by the permutation index and the brief index (figure 1(a)). This behavior is consequence of the high dimensionality of the **documents** data set.

String mappings (Figures 1(c) and 1(d)) show a diversity of performance. That is, **KNR** *LCS* has the highest recall rate in both data sets, this is followed by **KNR** *Levenshtein*. The worst performance obtained by **KNR** prefixes, note that for this method the recall get worst as the number of references increases.

In the set mappings, namely Figures 1(e) and 1(f), all the indexes share almost an identical recall rate. **KNR** set methods need larger values of $|R|$ to achieve its optimal value. This is a good characteristic, because larger $|R|$ means faster inverted indexes (which is the underlaying data structure for set mappings, Section 3.3).

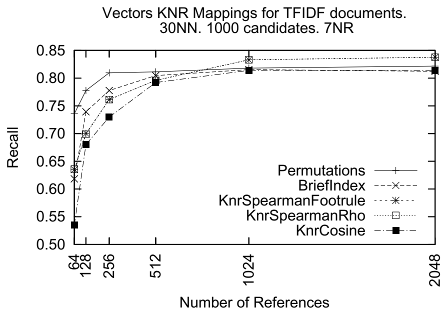
3.2 Increasing Recall Using Several KNR Indexes

A general technique to increase the recall in **KNR** methods is the usage of several indexes, as reported by [10,6]. Unfortunately, as expected, the recall increases but also the time and the storage requirements.

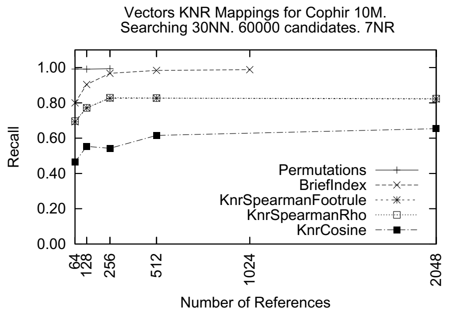
The increasing in the recall is produced by the diversity of several R sets. Each index retrieves a subset of the complete result, note that the size of the partial result depends on the particular **KNR** method. The union of partial results is our final result set.

Note that joining results is not tight in the length of different R sets or the particular **KNR** method, nor the mix of them. In order to show this behavior, Figure 2(a) shows the recall in each index. In the right side, Figure 2(b) shows the corresponding *cumulative* recall. The cumulative recall is obtained by using the union of the current and all the smaller results. For example the cumulative point for $|R| = 512$ needs the union of result from indexes working with $|R|$ as 64, 128, 256, and 512. For $|R| = 512$ four indexes are needed to work.

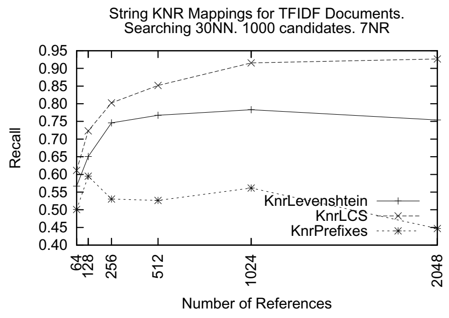
[6] `http://www.mono-project.org`



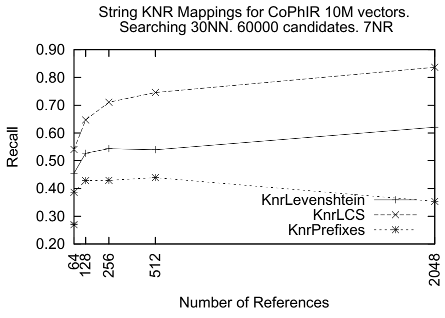
(a) Recall for TFIDF Documents. Vector mappings



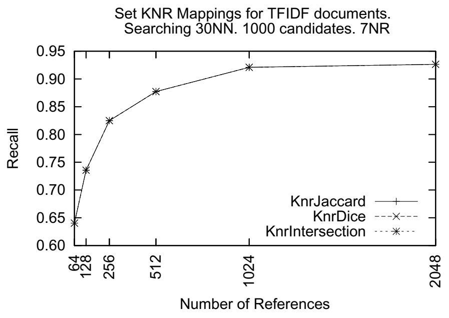
(b) Recall for CoPhIR 10M MPEG7 Vectors. Vector mappings



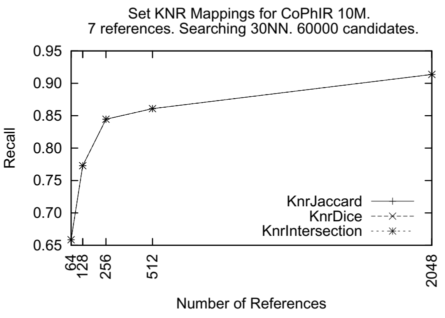
(c) Recall for TFIDF Documents. String mappings.



(d) Recall for CoPhIR 10M MPEG7 Vectors. String mappings.



(e) Recall for TFIDF Documents. Set mappings

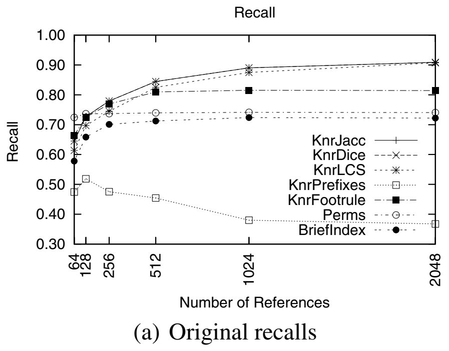


(f) Recall for CoPhIR 10M MPEG7 Vectors. Set mappings

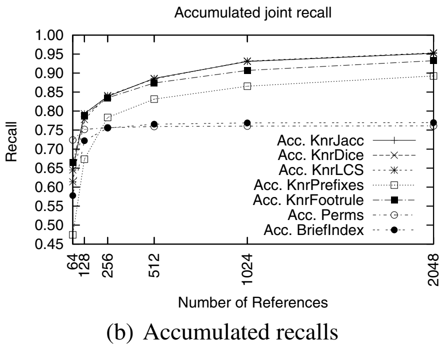
Fig. 1. Recall behavior of **KNR** mappings

This solution can be expensive, but it is a simple and effective solution to the problem of low recall of some **KNR** indexes and it can be used to index *very* large databases.

As shown in Figure 2(a), the best single recall is smaller than the corresponding point in 2(b). This is evident, since partial results are joined. A particular large improvement was found for **KNR** Prefixes (PP-Index), which improves dramatically the recall,



(a) Original recalls



(b) Accumulated recalls

Fig. 2. Joining **KNR** results for 112544 color's histograms, searching 30NN. The *accumulated* curves uses the *union* of current and smaller $|R|$ results.

transforming the index into an appealing option for high quality requirements. Other indexes present a gain of $5 - 15\%$, which is still very important.

The same strategy is valid to speed up searches, using a partition of the database (a disjoint collection of subsets) across several search servers. As usual, hybrid approaches can be used to achieve both recall and speed enhancements.

3.3 Improvements Using K and γ Variations

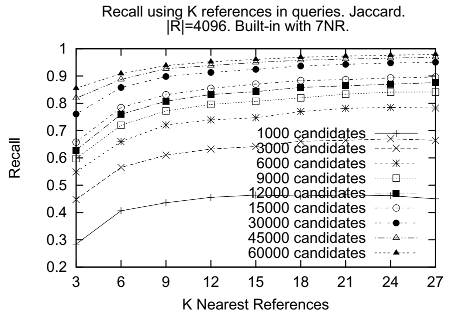
The optimization of K and γ parameters can be used to control the tradeoff between time and recall.

The parameter K has two roles, for building and for searching. At building time K modifies the size of the index, because it increase or decrease $|\hat{u}|$. When searching, increasing K allow to increase the recall without growing the index size and using the same index. The cost of searching-K shows up in increasing the number of set operations in the inverted index, which impacts the real time used for searching. Due to these characteristics, our experiments are focused on searching-K.[7]

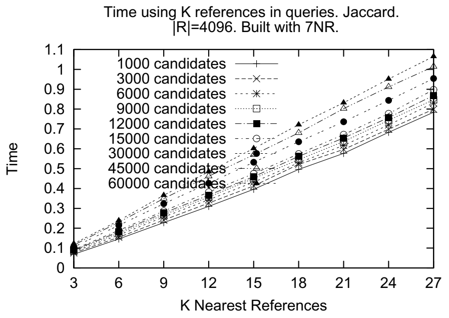
About the Implementation of this Experiment. In this experiment we used a simple inverted index to index the **KNR** Jaccard mapping. References are the thesaurus, and objects identifiers are inserted in the posting lists. The search algorithm consists in computing the set *union* and counting the cardinality of the union of the inverted lists (which is in fact the cardinality of the intersection of corresponding **KNR** sets).

Using an inverted index increases the space complexity from $nK \log |R|$ bits to $nK \log n$ bits, but reduces the cost of computing the candidate list. Let us define γ' as the maximum number of possible candidates, $\gamma' = |\bigcup_{x_i \in \mathbf{KNN}_d(q \in U, R)} \{\hat{v} \in \hat{U} : x_i \in \hat{v}\}|$, then we perform $O(\gamma')$ implicit evaluations of d'. The union uses $O(\log K)(\sum_{x_i \in \mathbf{KNN}_d(q \in U, R)} |\{\hat{v} \in \hat{U} : x_i \in \hat{v}\}|)$ comparisons. Finally we perform $\min\{\gamma', \gamma\}$ distances d. Notice that all these costs are not directly dependent of n.

[7] Increasing searching-K is meaningless for **KNR** prefixes, permutations and brief permutations.



(a) Recall changing K ($|\hat{u}|$) to compute queries mappings



(b) Time changing K ($|\hat{u}|$) to compute queries mappings

Fig. 3. γ and K strategies enhancing **KNR** methods. CoPhIR 10M 30NN.

This analysis is important since we present real time results along the recall in the CoPhIR 10M database (which can be considered a large database). The index construction uses seven nearest references ($K = 7$).

Time and Recall Tradeoff Induced by Searching-*K*. Figure 3(a) shows the behavior for **KNR** Jaccard for different γ and K values. These results show that a large K value produce higher recalls, even in moderate γ values (e.g. 15000 candidates). The improvements by γ are rapidly stabilized and gives low increments for $\gamma \leq 30000$ candidates, under this configuration, recall values are up to 0.9 for $K = 6$, an being close to perfect for larger configurations. In the other hand, Figure 3(b) shows the impact of the K and γ variations in the search time. On the same Figure, large K values imply higher cost than increasing the number of candidates, i.e. there are more inverted lists (and objects) to compute the union operation.

4 Conclusions and Future Work

In this work, we presented a novel framework for approximate proximity search algorithms called K Nearest References (**KNR**) methods. This framework consists in mapping spaces from a general metric space or similarity space to a simpler space using **KNN** queries in a set of references R. The produced mapped spaces have a simple and well defined structure, allowing the creation of string indexes and inverted indexes [7,12].

As part of our study, we described and analyzed previous methods and explained how they belong to **KNR** indexes. Furthermore, we presented several enhancements for **KNR** methods based on parallelization, distribution and parameter optimizations. These enhancements reduce the search time, increase the recall, and highlight the scalability properties of the **KNR** mappings.

Notice that the set of references R can be indexed to search for the K nearest references. Since R is relatively small, we can use an exact index like AESA [2] for the searching. Using an approximate index to search for the **KNR** should be also consider, but may lower the recall of the subsequent index.

Finally, the present work focus on static collections, as a consequence a deep study on dynamic collections would be interesting. That is, **KNR** algorithms supporting efficient updates, inserts and deletion of items.

Acknowledgements

The first author acknowledges support from the National Council of Science and Technology (CONACyT) of Mexico to pursue graduate studies at the Universidad Michoacana de San Nicolas de Hidalgo. Also, part of this work was developed during his stay at the DCC of the University of Chile.

References

1. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, San Francisco (2006)
2. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. 33(3), 273–321 (2001)
3. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Computing Surveys 33(3), 322–373 (2001)
4. Chávez, E., Navarro, G.: Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. Information Processing Letters 85, 39–46 (2003)
5. Chavez, E., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. IEEE Transactions on Pattern Analysis and Machine Intelligence 30(9), 1647–1658 (2008)
6. Amato, G., Savino, P.: Approximate similarity search in metric spaces using inverted files. In: InfoScale 2008: Proceedings of the 3rd international conference on Scalable information systems, ICST, Brussels, Belgium, Belgium, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 1–10 (2008)
7. Baeza-Yates, R.A., Ribeiro-Neto, B.A.: Modern Information Retrieval. ACM Press / Addison-Wesley, New York (1999)
8. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing documents and images, 2nd edn. Morgan Kaufmann Publishing, San Francisco (1999)
9. Téllez, E.S., Chávez, E., Camarena-Ibarrola, A.: A brief index for proximity searching. In: Bayro-Corrochano, E., Eklundh, J.-O. (eds.) CIARP 2009. LNCS, vol. 5856, pp. 529–536. Springer, Heidelberg (2009)
10. Esuli, A.: Pp-index: Using permutation prefixes for efficient and scalable approximate similarity search. In: LSDS-IR Workshop (2009)
11. Navarro, G.: A guided tour to approximate string matching. ACM Computing Surveys 33(1), 31–88 (2001)
12. Grossman, D.A., Frieder, O.: Information Retrieval: Algorithms and Heuristics. Springer, Heidelberg (2004)
13. Bolettieri, P., Esuli, A., Falchi, F., Lucchese, C., Perego, R., Piccioli, T., Rabitti, F.: Cophir: a test collection for content-based image retrieval. CoRR abs/0905.4627v2 (2009)