DIMACS Technical Report 99-50

Cluster-preserving embedding of proteins

by

Gabriela Hristescu[1] Martin Farach-Colton[2]

[1]hristesc@cs.rutgers.edu
[2]Permanent Member, farach@cs.rutgers.edu
Department of Computer Science, Rutgers University; Piscataway, NJ 08855, U.S.A.
Supported by NSF Career Development Award CCR-9501942, an Alfred P. Sloan Research Fellowship and NATO Grant 960215.

ABSTRACT

Similarity searching in protein sequence databases is a standard technique for biologists dealing with a newly sequenced protein. Exhaustive search in such databases is prohibitive because of the large sizes of these database and because pairwise comparisons are slow. Heuristic techniques, such as FASTA and BLAST, are useful because they are fast and accurate, though it has been shown that exhaustive search is more accurate. Therefore, there are times when one would like to perform an exhaustive search.

We propose an efficient method, called SparseMap, for preprocessing a database of proteins to support efficient similarity searches using expensive but sensitive distance functions, such as those based on Smith-Waterman similarity. Our method is based on a *Low-dimensional Euclidean Embedding* approach. We compare our method with other embedding approaches, and show that our method is faster and produces embeddings which preserve more biological information about the proteins, such as pairwise distance and biological clusters.

1 Introduction

When protein sequencing was first developed, it was a laborious lab technique. Only well studied proteins were sequenced at first. As sequencing became commonplace and inexpensive, poorly characterized proteins were sequenced. Rather than starting the biological investigation from scratch on each novel protein, biologists found that by comparing the sequence of such proteins with the sequences in a database of known proteins they could extrapolate protein function and other information to guide in the design of experiments to characterize the novel protein, thus speeding up the process of protein exploration. This time saving device has become part of every lab's arsenal. Therefore, designing efficient tools for similarity querying in protein database is crucial for molecular biology research.

Proteins are molecules consisting of one or more chains of amino acids. There are 20 different amino acids. The sequence of amino acids in the chain is called the protein sequence, while the folding of the chain is called the protein structure. For newly discovered proteins, it is trivial to find the sequence, while determining their structure requires the use of X-ray crystallography or NMR spectroscopy, lab techniques which are not applicable to all proteins and which are substantially more time-consuming than sequencing.

As noted above, useful relationships between proteins can be detected by sequence comparison, though the proteins must be relatively closely related for such similarities to show up at the sequence level. In order to detect more distant relationships, structural comparison is needed [Ore94]. Because of the more complicated and time consuming methods for determining the structure of proteins, databases of protein structures is almost two orders of magnitude smaller than that of protein sequences. The sequence similarity between proteins can be computed using quadratic-time dynamic programming methods like Smith-Waterman [SW81], if local similarity is sought, or Needleman-Wunsch [NW70] for global similarity. Even though sequence similarity is expensive to compute, it is still much faster than structural similarity computation [SB98, SB97]. Therefore, similarity querying is usually performed on sequence databases.

Because of the large sizes of protein databases and expensive similarity computation, programs that perform an exhaustive search of the database, like SSEARCH [SW81] or PSW [BH96], which are based on the Smith-Waterman [SW81] method, are accurate but infeasible. Another approach taken by similarity retrieval systems in protein databases, like FASTA [PL88], BLAST [Alt+90] or WU-BLAST2 [Alt+96], is to use heuristics to guide the search. For example FASTA first finds locally similar regions between the query sequence and database sequences using lookup tables, based on ungapped identity, joins nearby high scoring regions and applies a dynamic programming method on a band around the best region found in order to compute a similarity score. One of the shortcomings of these methods is that they trade accuracy for speed. For example, FASTA will fail to find a highly similar protein if gaps are scattered evenly in the matching region.

Recent studies [BCH98, AS98] show that, even though these heuristic methods give good results, the results are not as precise as exhaustive search. Thus, while BLAST and related methods are very successful, there are times when one might be interested in a secondary exhaustive search. We consider the problem of speeding up searches in databases in which objects in the database have expensive pairwise distance function. In particular, we will be interested in finding the *Nearest Neighbor (NN)* to a query protein in the database, where neighborhood is determined by our given distance function. We will base our experiments in this paper on protein databases with a Smith-Waterman (SW) based distance function.

To speed up NN searching, it is necessary to preprocess the database. For some special cases of distance functions, quite a lot is known about NN searching. For example, suppose the elements of the database are *points in low dimensional Euclidean space*, that is, the points $P = \{p_1, \ldots, p_n\}$ of the database are represented as vectors $p_i = [c_{i,1}, c_{i,2}, \ldots, c_{i,d}]$ such that the distance from p_i to p_j is given by

$$L_2(p_i, p_j) = \sqrt{\sum_{k=1}^{d}(c_{i,k} - c_{j,k})^2}$$

(P, L_2) forms a d-dimensional Euclidean space, and of course, if d is small enough, we call this a low-dimensional Euclidean space. For such spaces, we can answer NN queries via methods known as *Spatial Access Methods* [KS97, BB98, KAS98].

Protein sequences, along with an SW distance function, do not form a Euclidean space. While protein sequences can be trivially represented as vectors, the Euclidean distance between such vectors is meaningless, and will, in general, be quite different from any SW distance function. Thus, if we hope to use Spatial Access Methods for NN searching on proteins, we need to *embed* the proteins into a low-dimensional Euclidean space. By such an embedding, we mean that we must find a function $\Phi : P \rightarrow \Re^d$ such that, if $S(\cdot, \cdot)$ is our biologically meaningful distance function, then $S(p_i, p_j)$ closely approximates $L_2(\Phi(p_i), \Phi(p_j))$. Under the right circumstances, the existence of such a $\Phi()$ will allow us to use Spatial Access Methods for NN searching in a protein database. Also, we can use such an embedding to determine biologically significant clusters of proteins, which correspond to protein families. The question is how to perform this mapping of the protein database into a low-dimensional space fast, while preserving the biologically significant relationships between proteins. This mapping procedure is also referred to as *feature extraction*, where each of the dimensions of the embedded space corresponds to a feature of the protein space. Note that features, as used here, do not refer to biological features, such as soluble versus trans-membrane, but rather to a mathematically abstract feature corresponding to a dimension of the embedding.

Traditional embedding methods, such as Singular Value Decomposition, have two shortcomings that we must overcome. First, almost all such methods are *off-line*, that is, they require all points to be embedded before the computation of the embedding of the query begins. Such methods are not useful for NN searching, since one must first embed the points of the database, then embed query points into the same space as they are presented. One does not know all query points at the time of the embedding. Second, the embedding will depend on pairwise distances, and we cannot afford to compute all such pairwise distances. To see why, consider that if we compute the distance from the query point to all points in the database, we are simply back to sequential search. Thus, the embedding technique will have to be *sparse* in the sense that not all distances between the query point and database points can be computed. Furthermore, we will want a technique which is sparse in that not all pairwise distances are computed during the original embedding phase, thus the all-pairs distance computation is prohibitively expensive.

In this paper we propose SparseMap, a method to embed proteins into a low-dimensional space. SparseMap scales very well while preserving distances between proteins, as measured both directly on the distances, as well as in that the embedding preserves biological clusters. Even though the method that we developed is a general method that can be used with any kind of similarity measure between proteins, for practical reasons we chose to test it with sequence similarity. We hope to extend this work to structural similarity measures in future work.

Recently, Faloutsos and Lin [FL95] proposed a scalable method, called FastMap and showed that it preserves distances approximately. Compared to FastMap, SparseMap scales better because it computes a sparser subset of all pairwise distances between proteins in the database and requires a smaller number of scans of the protein database in order to compute the embedding. Furthermore, it yields embeddings of substantially higher quality. To compare the two methods in terms of the quality of the embeddings they produce, we used two metrics. First, we used the *Stress*, a standard criterion proposed by [Kru64] and the *Cluster Preservation Ratio*, a measure for testing the preservation of biologically significant families of proteins as defined in the PROSITE database [PRO99].

Outline. The rest of the paper is organized as follows. In Section 2 we discuss other embedding methods. In Section 3 we present our method, SparseMap. In Section 4 we discuss analytical differences between the methods. In Section 5 we present an experimental comparison of these methods and in Section 6 the conclusions.

2 Embedding Methods

In this section, we introduce the Bourgain embedding method on which our approach is based. We begin by giving some background. We will then give an example to illustrate the method followed by a brief discussion of its practical limitations.

Suppose we have a distance function $D : X \times X \rightarrow \Re^+$, that is, $D(\cdot,\cdot)$ is a distance function between the elements of X. We can extend D as follows: let X' be a subset of X. Then $D(x, X') = \min_{y \in X'}\{D(x, y)\}$, that is, $D(x, X')$ is the distance from x to its closest neighbor in X'. Let $R = \{X_1, X_2, \ldots, X_k\}$ be a set of subsets of X. Then we can define an embedding with respect to R as follows: $E_R(x) = [D(x, X_1), D(x, X_2), \ldots, D(x, X_k)]$. It is not obvious at first why such an embedding might have any reasonable properties. For one thing, it is highly non-linear, and so visual intuition tends not to be too helpful in understanding the behavior of such an embedding. However, we can gain some understanding of this type of embedding by considering the case where the points of the metric space are packed tightly into well separated clusters. Then, if some set X_1 has a point which is in one cluster C_1 and not in another C_2, the distance $D(x, X_1)$ will be small for $x \in C_1$ and large for $x \in C_2$. The dimension corresponding to X_1 will then induce a large distance between points in C_1 and C_2. Obviously, we cannot count on the input points being packed into well-separated clusters, or that the sets X_i have the right arrangements with respect to the clusters. However, Bourgain showed that there is a set of reference sets which works for general inputs.

The Bourgain embedding, then, is a choice of reference sets, R, and the corresponding embedding E_R. The choice of the sets in R will be randomized via a procedure to be described below:

R consists of $O(\log^2 n)$ sets $X_{i,j}$, which we can think of as being naturally organized into *columns* and *rows*. We call the number of columns in the embedding the *width* of the embedding and denote it by κ and the number of rows the *length* β of the SparseMap. Let $R = \{X_{1,1}, X_{1,2}, ..., X_{1,\kappa}, X_{2,1}, ..., X_{2,\kappa}, X_{\beta,1}, ..., X_{\beta,\kappa}\}$, where $\kappa = O(\log n)$ and $\beta = O(\log n)$. The Bourgain embedding will thus define $O(\log^2 n)$ dimensions. We select the elements of $X_{i,j}$ to be any random subset of X of size 2^i. Thus, we get κ reference sets of size 2, κ of size 4, etc., up to β of size approximately n.

Bourgain showed [Bou85] that if R is selected as above, then the embedding E_R has *distortion* $O(\log n)$, where the distortion of an embedding is the maximum stretch of any distance, that is $D(i,j) \leq \log n l_2(E_R(x_i), E_R(x_j))$, and $D(i,j) \geq l_2(E_R(x_i), E_R(x_j))/\log n$.

It seems at first glance that modifying a distance by as much as a $\log n$ *multiplicative* factor would almost completely destroy the original information contained in the distance function $D(\cdot,\cdot)$. For example, if the we are embedding 1024 points, and $D(\cdot,\cdot)$ ranges over a factor of 10, a $\log n$ distortion embedding could arbitrarily reorder the distance between objects in our dataset. However, as with many worst case bounds, the Bourgain guarantee of no more than $\log n$ distortion is often very far from the actual behavior on real data. Furthermore, the $\log n$ bound is tight in the sense that there are metrics which require this much distortion for any Euclidean embedding. Experimental analysis on meaningful data rather than on concocted counterexamples is the *sine qua non* of feature extraction evaluation.

Example. Consider the following example of the Bourgain embedding. Let $D(\cdot,\cdot)$ be given by the following matrix:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
x_1	0	12	10	9	10	6	3	10
x_2		0	8	13	14	10	13	8
x_3			0	11	12	8	11	2
x_4				0	3	7	10	11
x_5					0	8	11	12
x_6						0	7	8
x_7							0	11
x_8								0

The Bourgain algorithm specifies that we pick 6 reference sets at random, 3 with 2 elements each, and 3 with 4 elements each. Here is one particular choice:

x_3, x_4	x_1, x_8	x_2, x_6
x_4, x_5, x_2, x_3	x_1, x_4, x_6, x_3	x_7, x_5, x_3, x_6

We can now compute the embedding of, for example, x_3 and x_5 as follows.

$$E_R(x_3) = \begin{array}{|c|c|c|}\hline 0 & 2 & 8 \\ \hline 0 & 0 & 0 \\ \hline\end{array} \qquad E_R(x_5) = \begin{array}{|c|c|c|}\hline 3 & 10 & 8 \\ \hline 0 & 3 & 0 \\ \hline\end{array}$$

Then, $l_2(E_R(x_3), E_R(x_5)) = \sqrt{3^2 + 8^2 + 0 + 0 + 3^2 + 0} \approx 9.055$. The original distance between x_3 and x_5 is 12.

Limitations. The Bourgain embedding has two very serious drawbacks. First, it produces $O(\log^2 n)$ dimensions. Again, if we are embedding 1024 points, we could end up with 100 dimensions, which would be far too many. Second, with high probability, every point in the dataset will be selected in some reference set $X_{i,j}$. Thus, we will need to compute all $\binom{n}{2}$ distances in order to perform the Bourgain embedding. If we are using the embedding for similarity searching, we would need to compute $E_R(q)$ for a query point q. This would require a distance evaluation between q and every point in X. There would be no point in performing an embedding if we were willing to spend the time to evaluate all n distances between q and the points in X, since we are using the embedding in order to facilitate a nearest neighbor search.

3 Our Approach: SparseMap

Our research goal is to investigate how to make the Bourgain method practical. We propose a heuristic modification of the Bourgain method which is designed to perform very few distance evaluations. The key idea is a loop interchange. Bourgain's method is typically thought of as having an outer loop which runs through all the points, and then an inner loop which computes all the features. We can equivalently imagine computing the first feature for every point, then the second feature, etc., until we have computed however many features we want. So far, we have not changed anything substantial: whether we loop on all points and then on all reference sets, or on all reference sets and then on all points, we end up doing the same distance evaluations.

Notice that the features are defined by a two-dimensional array. Thus, the order in which we loop through the features is not determined. For reasons which are suggested by the proof of the distortion bound of Bourgain's algorithm, we will proceed row-by-row, that is, we will compute features for all reference sets of size 2, then for size 4, etc.

So finally, we need to know how to actually save on distance evaluations. The main idea is as follows: Suppose we already computed k' features for every point. Then we have a rough estimate on the distances between points. If the first k' features are good enough, then we can very quickly compute the approximate distance from some point to all points in a reference set, using these first k' features. We can then consider just the best few candidates, and perform full distance evaluations to these objects. Concretely, we get

- For every reference set S_k (taken in row-major order)
  - For every point p
    - For every point $q \in S_k$, compute the approximate distance $d'(p,q) = \sqrt{\sum_{l=1}^{k-1}(p_l - q_l)^2}$, where p_l is the l^{th} feature of p, and similarly for q.
    - Find the σ qs with smallest d' distance to p.
    - Evaluate the true distance $D(p,q)$ for each such q.

* Take $p_k = D(p, q')$, where q' is the q with smallest $D(p, q)$.

Thus, for every point, and for every feature, we compute σ distances. We get $O(nk\sigma)$ distance evaluations, if there are n points and k features.

A Greedy Resampling Heuristic. Once we have extracted k features, we can even further reduce the dimensionality of the embedding space by selecting a smaller set k' of high quality features from the initial k. The quality of a feature can be quantified in terms of how well it reduces the stress. We propose the *Greedy Resampling* algorithm for picking out the best subset of features.

Having extracted k features, each single feature is compared in terms of the stress of the embedding using only that feature. If the database has n protein sequences, computing the stress of all pairwise distances requires $\binom{n}{2}$ distance evaluations, but we can simply pick out a random sample of distances and compare the stress on just these distances. Thus, picking out the best feature is very fast compared to finding the k features to begin with. Once some feature f_1 is selected as the best, we can pick the second feature as the one which produces the best stress when combined with f_1. We can proceed in this greedy manner until we have reordered all k features by decreasing order of quality.

Having done this, if we need $k' < k$ features, we can simply take the first k' features of the reordered embedding. Thus, Greedy Resampling is a dimensionality reduction method. Note that we are not guaranteeing that we will always pick out the best k' features, but picking out the best k' features takes exponential time, and so we suggest greedy resampling as a heuristic. We implemented the greedy resampling in SparseMap.

There are two questions one might ask about our embedding method: does it do well in theory and does it do well in practice. We address these questions in the next two sections by comparing SparseMap and FastMap both analytically and experimentally. The FastMap method is described in Appendix A. In order to provide a fair comparison of the two methods, we have also modified FastMap to include greedy resampling.

4 Analytical Comparisons

In this section, we compare SparseMap and FastMap in terms of the following three criteria: quality, sparsity and locality.

Quality. As stated above, SparseMap is based on the Bourgain algorithm, which guarantees a distortion of $O(\log n)$ for any metric, and there are some metrics which require $\Omega(\log n)$ distortion. It is easy to show that no sparse method can have such a bound. Consider, for example, the *Needle-in-a-haystack* distance function. All distances are some large Δ, except, perhaps, for some unknown pair a, b which are at distance $\epsilon \ll \Delta$ apart. Unless we find the pair a, b, we do not know on any unevaluated distance if it is Δ or ϵ. Thus we cannot provide an embedding with distortion of less than Δ/ϵ, a quantity which is not even bounded. But this is not a shortcoming of SparseMap, it is a shortcoming of *every sparse method*, including FastMap.

Similarly, FastMap is based on the Karhunen-Loève transform, which gives some optimality guarantees about picking out the best *linear projections.* Notice that it gives no guarantees on finding the best features overall, only the best linear features, under some criterion. FastMap is a heuristic and gives no guarantees.

Two metrics were used to test the quality of the embedding, the *Stress* and the *Cluster Preservation Ratio.* The stress of a distance function $d(\cdot,\cdot)$ with respect to another distance function $D(\cdot,\cdot)$ is defined as:

$$Stress(d, D) = \sqrt{\frac{\sum_{i,j}(d(i,j) - D(i,j))^2}{\sum_{i,j} D(i,j)^2}}$$

The *Cluster Preservation Ratio* measures the preservation of biologically significant clusters and is defined as:

$$CPR = \frac{1}{n}\sum_{i=1}^{n} \frac{|E_{q_i} \cap C_{q_i}|}{|C_{q_i}|}$$

where q_i is a query point in cluster C_{q_i} in the protein space, $|C_{q_i}| = m_{q_i}$, and E_{q_i} is the set of closest m_{q_i} points to q_i in the embedded space.

Sparsity. Above, we showed that SparseMap performs $O(\sigma kn)$ distance evaluations while FastMap performs $O(tkn)$ evaluations. The greedy resampling optimizations of these algorithms do not add any significant number of distance evaluations. A comparison of the exact number of distance evaluations performed requires knowing t and σ, and analyzing the hidden constants in the O notation. Without getting into the details of the analysis, we can conclude that FastMap performs more or less $(t + 2)kn$ distance evaluations and SparseMap performs approximately $(\sigma + 2)kn$ distance evaluations. For a more detailed discussion on the number of distance evaluations see Appendix C.

Faloutsos and Lin suggested that $t = 7$, though we found very little advantage is gained by setting $t > 2$. We found that setting $\sigma = 1$ gave very reasonable result, so we set $t = 2$ and conclude that as n increases, FastMap will perform $\frac{t+2}{\sigma+2} = \frac{4}{3}$ times as many distance evaluations.

Locality. SparseMap randomly selects the reference sets that correspond to features. We distinguish two cases: either the reference sets fit in memory or they don't. Consider the second case in which not all reference sets fit in memory. The size of the reference sets grows exponentially with the row index of the feature. The smallest sets can be considered to fit in memory. Since the only information needed to compute the embedding corresponding to the bootstrapping features are the points in the reference sets, two scans, one to load them and one to compute the embedding for each point in the dataset, are needed. To compute the embedding of a point for each extension feature, the embedding for the previous features for that point and for the reference set points are needed. This can be achieved by loading a few reference sets during a scan and then computing their embedding, followed by another scan to compute the embedding of the rest of the points in the dataset. The reference sets for the next feature can also be prefetched during the computation of the embedding for the current feature. The worst case number of scans is therefore: $S_{SparseMap} = k + 1$. In the case where all points in the reference sets fit in memory, that is, where we compute sufficiently few features so that we need not generate very large reference sets, a scan for loading the reference set and the computation of their embedding followed by a scan during which the whole embedding for each point in the dataset is computed suffices. Only 2 scans are required in this case.

In order to select one feature, FastMap has to scan the data t times and then compute the embedding (projection) using one more scan. If the number of features extracted is k, then the number of scans is: $S_{FastMap} = (t+1)k$. In the worst case, and setting $t = 2$, FastMap performs 3 times as many scans.

5 Experimental Comparison

We compared SparseMap and FastMap on several datasets by evaluating the quality of the computed embedding using two metrics. We also compared the performance of the two implementations by looking at the execution time for computing the embedding and the number of distance evaluations performed during the computation.

Selecting the Protein Datasets. We ran experiments on datasets selected randomly from the SwissProt protein database [BA98]. The size of the datasets ranged from 48 to the whole SwissProt database consisting of 66,745 proteins. The platform we used for these experiments was a dual-processor 300 MHz Pentium II with 512 MB of memory. SparseMap was implemented in C and tested under Linux. We used the original FastMap code supplied by Faloutsos and Lin [FL95].

Due to excessive memory requirements, we were not able to run FastMap on datasets larger than 511. Therefore, most of our comparisons will be the average of 10 random subsets of SwissProt, each of size 255. We find the performance of FastMap and SparseMap do not degrade significantly with size, and so these data are quite representative.

The Protein Distance Function. The biological relationship between proteins is captured in the representation of the protein space through the distance function. When two proteins are compared, their relationship is naturally expressed as similarity and not distance. Extensive work has been done in designing methods that compute the similarity between proteins. Global similarity can be computed using the method of Needleman and Wunsch [NW70]; a variant of this method, that ignores end gaps computes semi-global similarity; and for local similarity the Smith-Waterman method [SW81] is used. Global similarity may overlook local similarities, which are biologically important, but it is simpler to transform into global distance, even though it does not always yield a metric [Y99]. For local similarity, there is no straightforward transformation from similarity to distance, but Linial et al. [LLTY97] have proposed one that defines a pseudo-metric.

We used the Smith-Waterman [SW81] measure for local similarity with affine gap cost function $g(x) = \alpha + \beta x$ ($\alpha = 8$, $\beta = 4$) and the BLOSUM62 similarity matrix. The similarity measure is based on a dynamic programming method that takes $O(mn)$ time, where m and n are the lengths of the two protein sequences. Since this is a very expensive operation, it will significantly affect the running time of the methods that use it. If $s(a, b)$ is the similarity value between proteins a and b, the distance measure between the two proteins we considered is based on the similarity measure and is defined as $d(a, b) = s(a, a) + s(b, b) - 2s(a, b)$, as proposed by Linial & al [LLTY97].

The PROSITE Database. PROSITE [PRO99] is a database of biologically significant sites and patterns that can be used by specialized computational tools to identify quickly and accurately to which known family of proteins a new sequence belongs or which proteins in the database are part of the same family. We used PROSITE database release 15.0, which has 1352 entries that describe different patterns, rules and profiles/matrices. In order to test how well the embedding preserves biologically significant clusters in the protein space, defined by families in the PROSITE database, we selected a set of 255 proteins, of which 245 belonged to 14 protein families of sizes ranging from 4 to 34 proteins and 10 were not related to any other protein in the dataset.

5.1 Evaluating the Quality of Embeddings

Stress. The figures report results for embeddings of sizes ranging from one to the maximum number of features considered. We used $t = 2$ for FastMap. Note that in SparseMap we can select a number of rows to evaluate and a number of columns. If α is the number of rows and κ is the number of columns, then $k = \alpha\kappa$ will be the number of features extracted. We used the full Bourgain dimensionality, that is $\kappa = \alpha = \log n$ and $\sigma = 1$ for SparseMap. We measured the stress on 4,000 distances between randomly selected pairs of proteins from the dataset. For the comparison of the two methods we present the average over results on ten sets of 255 proteins. Comparing the two methods (Figure 5.1), we notice the quality

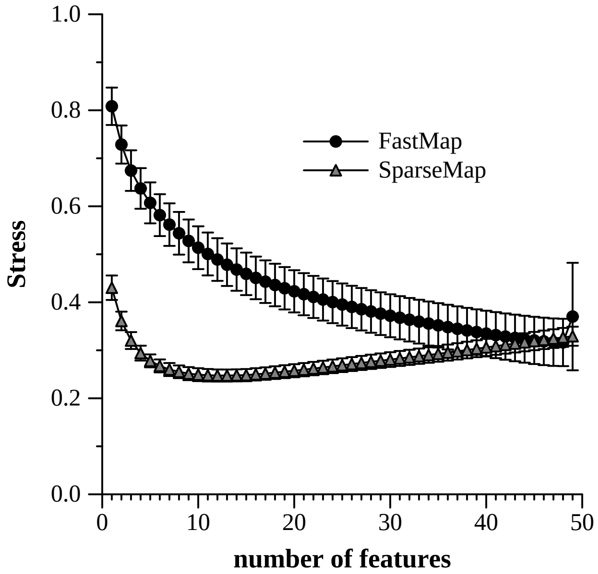

Figure 5.1: Average quality of SparseMap vs. FastMap on 10 sets of 255 proteins, with error bars.

difference between them. SparseMap starts by being 46% better than FastMap for the first extracted feature and stays around 50% for the first 10 features, after which the gap in quality decreases to almost 0 for 46 features. The thing to note is that SparseMap has its best stress at 10 dimensions, and this stress is not matched by FastMap, even out to 49 dimensions (though if sufficient dimensions were used FastMap would probably produce a lower-stress embedding). As noted above, it is quality of an embedding with a small number of dimensions which is most important, and so we conclude that SparseMap produces very high quality embeddings with a small number of features.

Cluster Preservation Ratio. We tested the preservation of biological clusters on the test set of 255 proteins, by randomly selecting 100 query points. 93 of the queries belonged to protein families and hence clusters in the protein space, while the other 7 queries were completely unrelated to any other protein in the dataset. Of the 93 queries, for 68 queries, the protein clusters defined by PROSITE families were preserved by the Smith-Waterman based distance function used, while for the rest of 25 queries the clusters were not preserved. The results reported are only on queries in preserved clusters. Note that the percentage of preserved clusters (73% in this case) also gives a possible measure for the quality of the (dis)similarity function used.

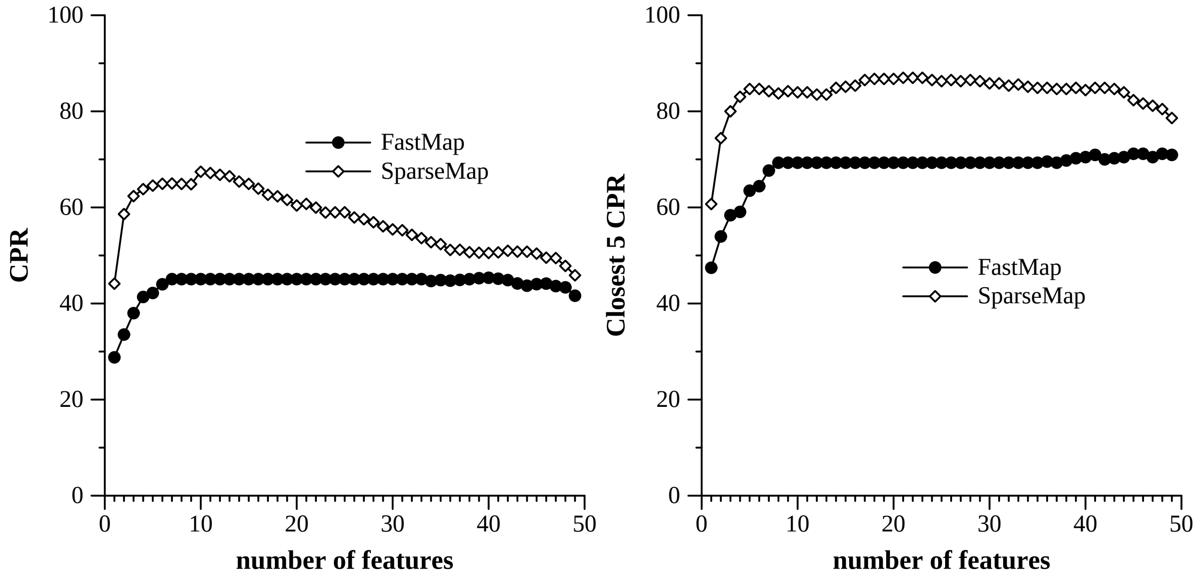

Figure 5.2: Cluster Preservation Ratio (CPR) of SparseMap vs. FastMap on a set of 255 proteins

Figure 5.2 compares the quality of the two methods through the *Cluster Preservation Ratio* percentage measure. The quality of SparseMap is about 50% better than FastMap for up to 10 features and stays better afterwards, although the gap between the quality of the two methods diminishes. Since the purpose of similarity querying is to retrieve the closest or approximately the closest protein in the database, we also investigated if the closest k proteins to a query protein belong to the same family. This means a relaxation in the definition of the *Cluster Preservation Ratio* measure, by having $m_{q_i} = min\{|C_{q_i}|, k\}$.

Figure 5.2 shows the results of this quality measure for $k = 5$. Again, SparseMap proves to generate a better embedding than FastMap, but the important thing to note is that more than 80% of the closest proteins for a query protein in the SparseMap embedding belong to the PROSITE indicated family for the query. This means that even an approximate nearest neighbor method would find with high probability a protein sequence with the right family classification for the query.

5.2 Evaluating the Performance

In addition to comparing the quality of the two methods, we were also interested in comparing the execution time of the two methods. We instrumented the code to get timing information and also profiled the execution in order to understand where the time was spent. Recall that greedy resampling does not require a significant number of distance evaluations and very little run time and is only a post-processing step. We will therefore discuss only the execution time of the two methods, disregarding the post-processing step. so we compare only SparseMap with FastMap.

Execution Time. Figure 5.3 presents the average execution time over runs on ten datasets of size 255. It shows results for increasing number of features for SparseMap and FastMap. SparseMap is 2.3 times faster than FastMap for the first feature and 2.8 times faster for all 49 features.

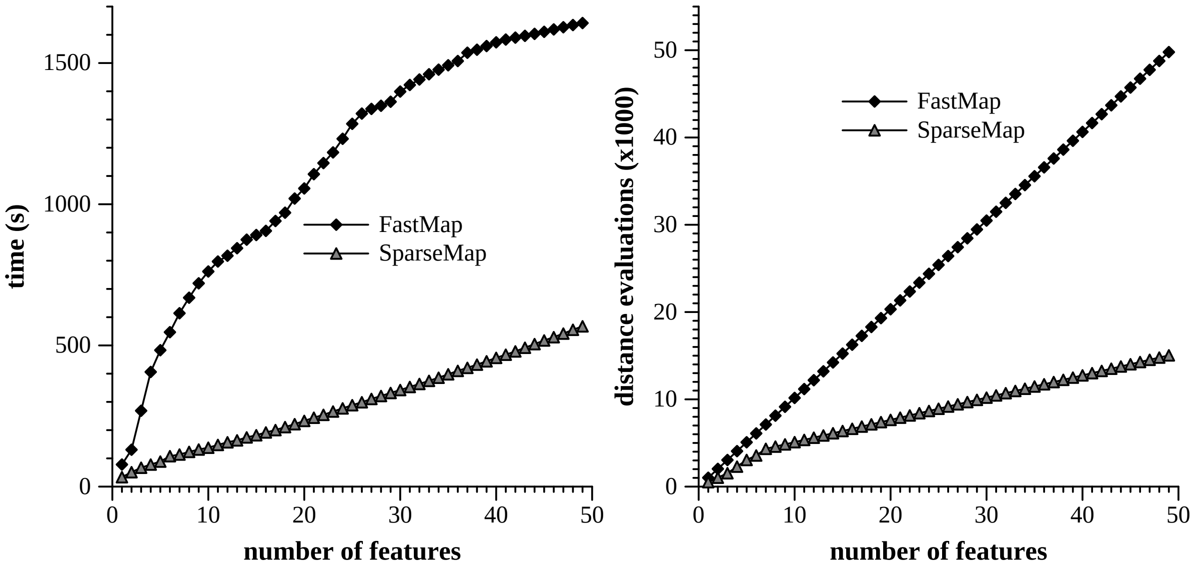

Figure 5.3: Average run time and number of distance evaluations for SparseMap and FastMap on a set of 255 proteins.

Number of Distance Evaluations. We profiled both methods to determine where the bottlenecks were in the computation. As expected, in both cases, more than 90% of the execution time was spent in the computation of the distance between data points (proteins).

Therefore, we were able to explain the difference in execution time by the difference in the frequency of distance computations between the two methods. However, the run times measured require some explanation. We would have expected FastMap to run about 33% slower than SparseMap since it performs about 4/3 as many distance evaluations in theory. However, two effects come into play here. First, FastMap performs more scans of the data, and therefore pays a penalty in terms of slower memory access. Also, the reference set is known ahead of time in SparseMap. This allows many optimizations, both in terms of possible parallelization, as well as in savings of repeated distance evaluations. Such optimizations are not possible for FastMap because the reference sets are only determined dynamically. Thus, Figure 5.3 presents actual measured data of optimized code.

The number of distance calls for a dataset of 255 protein sequences for a 49 feature embedding were 46% of the total number of pairwise distances for SparseMap and 153% for FastMap. Even for one feature FastMap computes 3.1% of distances compared to 1.5% for SparseMap. Even if we use the lowest value for $t = 1$, which means that one reference point is completely random, FastMap makes 115% of the total number of pairwise distance evaluations for 49 features extracted and 2.3% for the first feature.

6 Conclusions

We have proposed a fast algorithm, SparseMap, which maps protein sequences into k-dimensional space such that distances, as well as biologically significant protein families, are well preserved. Since distance evaluations in the protein sequence space are very expensive, the main goal in designing our method was a reduced number of distance evaluations, while maintaining the biological relationships between proteins.

We compared our SparseMap method with a previously proposed method, FastMap [FL95], on sets of protein sequences selected from the SwissProt database and found that SparseMap performs substantially fewer distance evaluations than FastMap. Furthermore, the *Stress* of the embedding is about half that of FastMap's for a reasonable number of dimensions. Testing how protein families from the PROSITE database are preserved, using a measure called the *Cluster Preservation Ratio*, also proved our method to be qualitatively superior. SparseMap and FastMap have very different approaches: SparseMap performs random non-linear projections while FastMap performs geometric projections. We conclude that SparseMap is an efficient and accurate method of embedding a protein database in a low-dimensional space, which, combined with an efficient Nearest Neighbor searching method, gives an effective method for similarity querying in protein databases.

References

[Alt+90] S.F. Altschul, R.J. Carrol, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[Alt+96] S.F. Altschul, and W. Gish. Local alignment statistics. *Methods Enzymol.*, 266:460–480, 1996.

[AS98] P. Agarwal, and D.S. States. Comparative accuracy of methods for protein sequence similarity search. *Bioinformatics*, 14:40–47, 1998.

[BA98] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence data bank and its supplement TrEMBL in 1998. *Nucleic Acids Res.*, 26:38–42, 1998.

[BB98] S. Berchtold and C. Bőhm. The pyramid-technique: Towards breaking the curse of dimensionality. *Proc. ACM SIGMOD Int. Conf. on Management of Data, Seatle, WA*, pages 142–176, 1998.

[BH96] P. Bucher, and K. Hoffman. A sequence similarity algorithm based on a probabilistic nterpretation of an alignment scoring system. *ISMB-4*, AAAI Press, Menlo Park, CA, 1996.

[Bou85] J. Bourgain. On lipschitz embedding of finite metric spaces in hilbert space. *Israel Journal of Mathematics*, 52:46–52, 1985.

[BCH98] S.E. Brenner, C. Chothia, and T.J. Hubbard. Assessing sequence comparison methods with reliable structurally identified distant evolutionary relationships. *Proc. Natl. Acad. Sci. USA*, 95:6073-6078 , 1998.

[FL95] C. Faloutsos and King-Ip Lin. FastMap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2):163–174, June 1995.

[GR99] V. Ganti, R. Ramakrishnan, J. Gehrke, A. Powell and J. French. Clustering large datasets in arbitrary metric spaces. *Proc. 15th Int. Conf. on Data Engineering (ICDE), Sydney, Australia*, 1999.

[PRO99] K. Hofmann,P. Bucher,L. Falquet, and A. Bairoch. The PROSITE database, its status in 1999. *Nucleic Acids Res*, 27:215–219, 1999.

[KAS98] K. V. R. Kanth, D. Agrawal, and A. Singh. Dimensionality reduction for similarity searching in dynamic databases. *Proc. ACM SIGMOD Int. Conf. on Management of Data, Seatle, WA*, pages 142–176, 1998.

[KS97] N. Katayama and S. Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 369–380, 1997.

[Kru64] J.B. Kruskal. Multidimensional Scaling by Optimizing Goodness of Fit to a Non-metric Hypothesis. *Psychometrika*, 29:1–27, 1964.

[KW78] J.B. Kruskal and M. Wish. Multidimensional Scaling. *Sage University Paper series on Quantitative Applications in the Social Sciences,Beverly Hills, CA*, 07-011, 1978.

[LLR94] Nathan Linial, Eran London, and Yuri Rabinovich. The geometry of graphs and some of its algorithmic applications. *Proc. of the 35th IEEE Annual Symp. on Foundation of Computer Science*, pages 577–591, 1994.

[LLTY97] M. Linial, N. Linial, N. Tishby, and G. Yona. Global self organization of all known protein sequences reveals inherent biological signatures. *Journal of Molecular Biology*, 268:539–556, 1997.

[NW70] S.B. Needleman, and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[Ore94] C.A. Orengo. Classification of protein folds. *Current Opinion in Structural Biology*, 4:429–440, 1994.

[P95] W.R. Pearson. Comparison of methods for searching protein sequence databases. *Protein Science*, 4:1145–1160 , 1995.

[PL88] W.R. Pearson, and D.J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. USA*, 85:2444–2448, 1988.

Ilya N. Shindyalov and Philip E. Bourne (1998) Protein Engineering 11(9) 739-747.

[SB98] I.N. Shindyalov, and P.E. Bourne. Protein structure alignment by incremental combinatorial extension (CE) of the optimal path. *Protein Engineering*, 11(9):739–747, 1998.

[SB97] A.P. Singh, and D.L. Brutlag. Hierarchical Protein Structure Superposition using both Secondary Structure and Atomic Representations. *Fifth Intl. Conf. on Intelligent Systems for Molecular Biology*, Menlo Park, CA, 284–293, 1997.

[SW81] T. Smith and M. Waterman. The identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[Y99] G. Yona. Methods for global organization of the protein sequence space. *Ph.D. thesis, Hebrew University*, 1999.

A FastMap

The approach taken in FastMap for embedding points into k-dimensional Euclidean space is based on 3-point linear projections. Consider three objects O_a, O_b and O_i, and the distances, $D(a, b), D(a, i)$ and $D(b, i)$, between them. Any such distances which satisfy the triangle inequality can be represented exactly in two-dimensional Euclidean space. See Figure A.4. Point O_a can be assumed to be at position $(0, 0)$, and O_b at $(0, D(a, b))$. To find O_i's two coordinates (x_i, y_i), we can solve two equations with two unknowns, plugging in the values $D(a, i)$ and $D(b, i)$.

FastMap uses the following procedure to find a feature:

1. Pick a set of reference points, O_a and O_b.

2. For each other point O_i, its feature is the projection onto the line induced by O_a and O_b when these three points are embedded as above. In the figure, it is denoted x_i.

So each feature is a linear projection defined by a reference set O_a and O_b. The question is then how to pick such a reference pair. Faloutsos and Lin's aim was to select a pair along the principal component, *a la* S.V.D. Since it would be computationally expensive to find the principal component of the points, they suggest finding the pair which is furthest apart, which they suggest should lie more or less along the principal component. But finding such pair is also computationally expensive! So the following heuristic is used: Pick a point at random. Compute its distance to all others. Find the furthest point. Compute this new point's distance to all others. Repeat t times, where t is a parameter of the algorithm. The two last points found this way are taken to be the reference points.

Note that this sketch only describes how the first feature is selected. All distance calculations for the second features factor out the part of the distance function which has already been captured by the first feature. Otherwise, if O_a and O_b are the furthest points originally, they will remain so and all features will be identical! We refer the reader to[FL95] for the details of how this is accomplished. We simply note here that the selection of a reference set always requires t one-against-all sets of distance evaluations.

Thus, the selection of the reference points requires $O(tn)$ distance evaluations. Computing the feature once the reference set is known requires another $2n$ distance evaluations. We conclude that k dimensions require $O(tkn)$ distance evaluations.

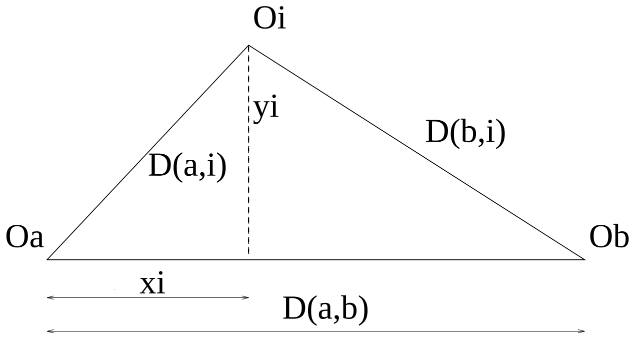

Figure A.4: Projecting point O_i on the line O_aO_b.

B Scalability

We studied how the SparseMap method scales with increasing dataset sizes both in terms of quality and running time. Figure B.5 shows the quality of the first 15 dimensions of SparseMap. We note very consistent behavior, with basically no degradation in the solution over a wide range of dataset sizes.

Figure B.6 shows the measured run time of SparseMap over a variety of data set sizes and number of features extracted. The flat initial part of the curve is due to startup costs. Finally, we show in Figure B.6 the estimated number of distance evaluations performed by SparseMap and FastMap. As noted above, we could not run FastMap on moderate size instances because of memory requirements, so this graph is plotting the calculated number of distance evaluations, without the SparseMap optimizations. These latter were omitted because the amount of their improvement is data-dependent, and so cannot be calculated.

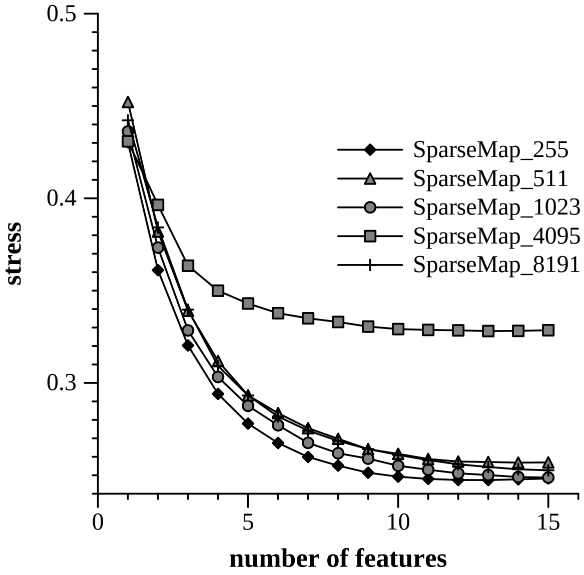

Figure B.5: The quality of SparseMap features for protein sets of sizes varying from 255 to 8191.

C Comparison of the number of distance evaluations

In this Appendix we analyze in more detail the number of distance evaluations performed by the SparseMap method and compare it with FastMap [FL95]. If we consider a dataset with n elements, the average computational cost of a distance evaluation in the distance space to be D and the cost of computing the Euclidean distance in an i-dimensional space to be $d_i = id$ where d is the cost of computing a squared difference. We consider the number of features extracted to be m.

The FastMap method consists of first choosing a pair of points that define a reference line and then projecting all points in the dataset on that line. At the next step, this procedure is repeated, except that the distances between points are considered projected in a hyperplane orthogonal to the chosen reference line. This introduces, in addition to a distance evaluation in the distance space, an extra d computation for each feature already extracted. The time to execute FastMap consists of the time $T_{extract}$ to extract the m features and the time $T_{project}$ to project the n points onto them. If for each feature extracted a number of t passes over the dataset are executed while on each pass the furthest point to a selected reference point is computed, then $T_{extract} = t\sum_{i=1}^{n}\sum_{j=1}^{m}(D + (j-1)d)$ and if c distance computations are needed to project a point onto a reference line, then $T_{embed} = c\sum_{i=1}^{n}\sum_{j=1}^{m}(D + (j-1)d)$. The number of distance computations c for projecting a point on a line is 2, considering that the distance between the reference points is known.

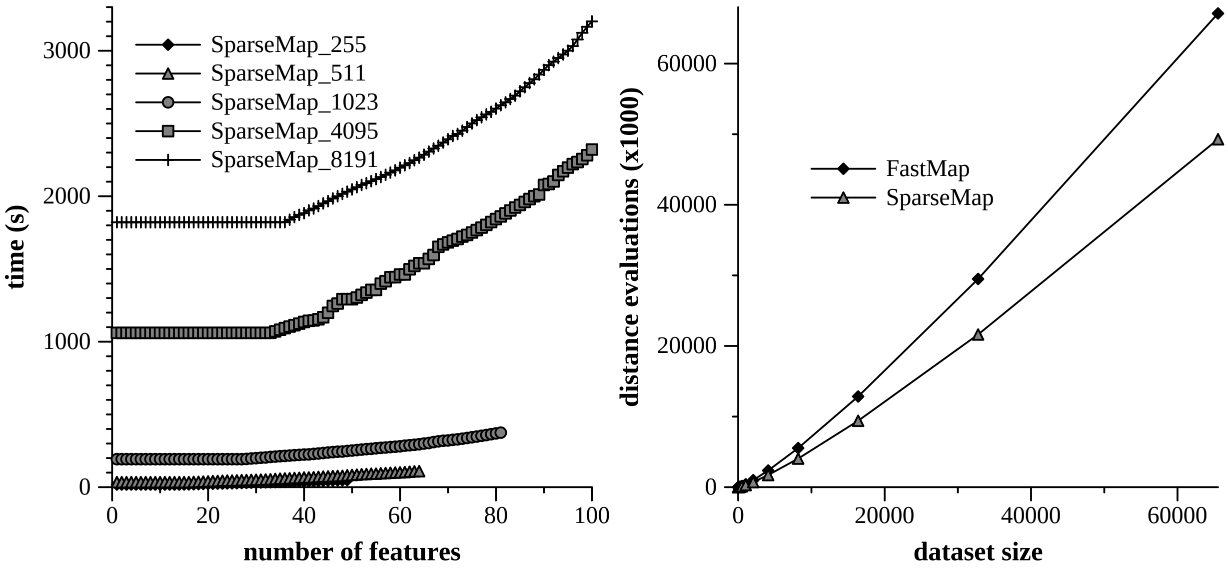

Figure B.6: Run time for SparseMap of sizes ranging from 255 to 8191 and the number of distance evaluations for SparseMap and FastMap.

Therefore:

$$T_{FastMap} = T_{extract} + T_{embed} = nm(t+c)\left(D + \frac{m-1}{2}d\right)$$

Computing the Bourgain transform would require randomly selecting the reference sets and then computing the embedding of a point in the image space as the smallest distance between the point and the points in the reference sets. Selecting the reference sets does not require any computation, and therefore for an embedding with κ columns and α rows: $T_{Bourgain} = 2n\kappa(2^{\alpha} - 1)D$

Instead of the α rows Bourgain would compute, SparseMap computes only β rows using the distance in the distance space. For the rest of $\alpha - \beta$ rows, the coordinate for the corresponding dimension i is approximated by first selecting the σ closest points in the already computed $(i-1)$-dimensional image space, then computing the smallest distance in the distance space among the σ points.

Extending the β row embedding to an α row embedding takes therefore:

$$T_{extension} = n\kappa\sigma(\alpha - \beta)D + n\sum_{i=\beta+1}^{\alpha}\left(2^i \sum_{j=\kappa(i-1)}^{\kappa i-1}(jd)\right)$$

Using the approximation $j \leq \kappa i$ we get:

$$T_{SparseMap} \leq n\kappa\left(2^{\beta+1} - 2 + \sigma(\alpha-\beta)\right)D + n\kappa\left((\alpha-1)2^{\alpha+1} - \beta 2^{\beta+2} + 4\right)d$$

Because computing D is by far more expensive than d, the dominant terms for the two methods are: $T'_{FastMap} = n\kappa\alpha(t+c)D$ and $T'_{SparseMap} = n\kappa\left(2^{\beta+1} + \sigma(\alpha-\beta)\right)D$ where the size of the embedding is $m = \kappa\alpha$.

For $\alpha = logn$ and $\beta = loglogn$:

$T'_{FastMap} = n\kappa(t + c)lognD$ and $T'_{SparseMap} = n\kappa((\sigma + 2)logn - \sigma loglogn)D$.

For this choice of parameters, both methods compute $O(logn)$ real distances. Considering that $c = 2$, the values for t that would give comparable numbers of distance computations in the distance space are $t = \sigma$ or $t = \sigma + 1$.