ΔB^+ Tree: Indexing 3D Point Sets for Pattern Discovery

Xiong Wang
Department of Computer Science
California State University, Fullerton
Fullerton, CA 92834-6870, USA
wang@ecs.fullerton.edu

Abstract

Three-dimensional point sets can be used to represent data in different domains. Given a database of 3D point sets, pattern discovery looks for similar subsets that occur in multiple point sets. Geometric hashing proved to be an effective technique in discovering patterns in 3D point sets. However, there are also known shortcomings. We propose a new indexing technique called ΔB^+ Trees. It is an extension of B^+-Trees that stores point triplet information. It overcomes the shortcomings of the geometric hashing technique. We introduce four different ways of constructing the key from a triplet. We give analytical comparison between the new index structure and the geometric hashing technique. We also conduct experiments on both synthetic data and real data to evaluate the performance.

1. Introduction

Three-dimensional point sets can be used to describe data in different domains, e.g. scientific data mining, computer-aided design, computer vision, etc. Pattern discovery is one of the problems that arise in these domains. It is concerned with similar substructures that occur in multiple point sets. For example, a motif is a substructure in proteins that has specific geometric arrangement and, in many cases, is associate with a particular function, such as DNA binding. Active sites are another type of patterns in protein structures. They play an important role through protein-protein and protein-ligand interaction, i.e. the binding process.

Similarity search in 3D point sets has been studied extensively. There are roughly three categories of approaches: volume-based approaches, feature-based approaches, and interactive approaches. Volume-based approaches use well-defined 3D structures to approximate the shapes of the point sets. They do not consider points inside the volumes. Thus they are not suitable for pattern discovery that is also concerned with those points. Feature-based approaches capture the shapes of the 3D point sets by descriptors. The descriptors are essentially very high dimensional spaces and indexing them is a well known difficult problem, due to "the curse of dimensionality" [1]. The interactive approaches rely on the user to distinguish differences and provide feedbacks that were then used to refine the search. In molecular data, the differences between two molecules are often so subtle that bare eyes can hardly detect them.

We propose a new index structure called ΔB^+ Tree that facilitates pattern discovery in 3D point sets. We introduce three variants of ΔB^+ Trees and compare the effectiveness of them on both synthetic data and real data. The rest of the paper is organized as follows. In Section 2, we describe our approach to pattern discovery in 3D point sets. In Section 3, we present the structure of a ΔB^+ Tree. In Section 4, we compare ΔB^+ Trees with the geometric hashing technique. In Section 5, we report some experimental results. We conclude the paper in Section 6.

2. Finding Patterns in 3D Point Sets

The pattern discovery process consists of three phases: (1) decompose the point sets to candidate patterns; (2) index the candidate patterns to a ΔB^+ Tree; and (3) calculate the occurrence numbers of the candidate patterns.

The first phase is domain dependent. For example, chemical compounds often have substructures that are connected by double bonds. The building block of such substructures is referred to as a block in graph theory. We used an adapted depth-first search algorithm to identify these substructures [5]. These substructures are considered candidate patterns. To discover active sites on the protein surfaces, we extracted the surfaces of the proteins and constructed candidate patterns using the k-nearest neighbors of any given surface atom [3].

In the second phase, all the candidate patterns are indexed to a ΔB^+ Tree. The minimum match identified by the algorithm is a triplet match. A *triplet* is an ordered set of

0-7695-1754-4/02 $17.00 © 2002 IEEE

three points $\{P_i, P_j, P_k\}$, such that $d_{ij} \leq d_{ik} \leq d_{jk}$, where $d_{ij} = \|P_i - P_j\|$, $d_{ik} = \|P_i - P_k\|$, and $d_{jk} = \|P_j - P_k\|$ stand for the Euclidean distances between each pair of the points. Given a collection $\mathcal{D}$ of candidate patterns, we decompose each candidate pattern $D \in \mathcal{D}$ to triplets and index the triplets to a ΔB^+ Tree. The key of the ΔB^+ Tree is the three-dimensional vector (d_{ij}, d_{ik}, d_{jk}). Two keys are compared according to lexicographic ordering. For every triplet $\{P_i, P_j, P_k\}$ in D, A tuple in the ΔB^+ Tree has the structure $(CPID, GCF)$, where $CPID$ is the identification number of the candidate pattern and GCF is a 3×3 matrix. All triplets of the same shape share the same key and are grouped together. $CPID$ tells where this triplet comes from and GCF serves as a reference frame in constructing matches in the third phase. The tuples are stored in a disk file, called the tuple file. Tuples with the same key are stored contiguously.

In the third phase, each candidate pattern is used to access the ΔB^+ Tree to calculate its occurrence number. Let Q be a query pattern and ϵ a given range of tolerable errors. The algorithm decomposes Q to triplets and uses each triplet to access the ΔB^+ Tree. For each triplet $\{P'_l, P'_m, P'_n\}$ in Q, it looks for those keys (d_{ij}, d_{ik}, d_{jk}) in the ΔB^+ Tree, such that $|d_{ij} - d'_{lm}| \leq \epsilon$, $|d_{ik} - d'_{ln}| \leq \epsilon$, and $|d_{jk} - d'_{mn}| \leq \epsilon$. Triplet matches are augmented together through comparison of the GCFs. For any query pattern Q, the process finds *all* possible matches of *any size* simultaneously. In other words, it identifies every possible alignment between Q and any candidate pattern within the range of tolerable errors. In contrast, volume-based and feature-based approaches do not provide alignment information.

3. ΔB^+ Trees

Recall that tuples with the same key are stored contiguously in the tuple file. The tuple file is indexed using a ΔB^+ Tree. Each distinct key has an entry in the leaf nodes of the ΔB^+ Tree. The ΔB^+ Tree is stored in a disk file, called the ΔB^+ Tree file. When building the internal node, we adopt the structure of CSB Tree [2]. The child nodes of each internal node are stored contiguously and only the pointer to the first child node is stored in the node. Notice that CSB Tree is a main memory index structure, while ΔB^+ Tree is disk based. The size of the node is set to the page size of the disk file.

Searching the ΔB^+ Tree with $\epsilon = 0$ is straightforward. For each triplet $\{P'_l, P'_m, P'_n\}$ generated from the query pattern Q, the searching algorithm seeks exact match with the key $(d'_{lm}, d'_{ln}, d'_{mn})$. When $\epsilon > 0$, starting from the root, the algorithm checks the keys in each level and filters out subtrees that are certainly out of the range. Essentially, the algorithm does a breadth first search. We also implement another algorithm that uses a key $(d'_{lm} - \epsilon, d'_{ln} - \epsilon, d'_{mn} - \epsilon)$ to access the ΔB^+ Tree. The subsequent keys in the leaf nodes are then checked until a key (d_{ij}, d_{ik}, d_{jk}), such that $d_{ij} > d'_{lm} + \epsilon$. Those keys that are not within the range are filtered out. The breadth first approach performed slightly better.

Lexicographic ordering in essence partitions the keys using the first dimension first. It could be better if the values in the first dimension include information from all the three dimensions. We developed three variants of ΔB^+ Tree. They all use three-dimensional vectors as keys. The difference is in the values they use in the first dimension. Let $l = d_{ij} + d_{ik} + d_{jk}$. The first variant uses (l, d_{ik}, d_{jk}) as the key. We will call this ΔB^+ Tree-Length. Let a be the area of the triangle formed by the triplet $\{P_i, P_j, P_k\}$. The second variant uses (a, d_{ik}, d_{jk}) as the key. We will call this ΔB^+ Tree-Area. Let r be the radius of the circle that circumscribes the triangle formed by the triplet. The third variant uses (r, d_{ik}, d_{jk}) as the key. We will call this ΔB^+ Tree-Circle.

4. ΔB^+ Tree vs. Geometric Hashing

The framework we introduced in [4, 5] also has three phases. The difference is that we used a three-dimensional hash table to store the triplet information. The framework maintains two disk files, i.e. the header file and the tuple file. The tuple file is the same as discussed above. The header file stores a three-dimensional array. An entry in the header file has the format $(firstTuple, nTuples)$, where $firstTuple$ is the address of the first tuple and $nTuples$ is the number of tuples in that hash bin. For any tuple with key (d_{ij}, d_{ik}, d_{jk}), the hash bin addresses are calculated as follows:

$$\begin{aligned} l_1 &= Round(d_{ij}^2 \times Multiplier) \\ l_2 &= Round(d_{ik}^2 \times Multiplier) \\ l_3 &= Round(d_{jk}^2 \times Multiplier) \end{aligned}$$

$$\begin{aligned} index_1 &= (l_1 + l_2) \bmod Prime_1 \bmod Nrow \\ index_2 &= (l_2 + l_3) \bmod Prime_2 \bmod Nrow \quad (1) \\ index_3 &= (l_3 + l_1) \bmod Prime_3 \bmod Nrow \end{aligned}$$

where $Prime_1$, $Prime_2$, and $Prime_3$ are three primes. $Nrow$ is the cardinality of the hash table in each dimension. $Multiplier$ (e.g. 100) is used so that some digits after the decimal point can contribute to the distribution of the entries. It is set according to the range of tolerable errors. For example, if $\epsilon = 0.001$, $Multiplier$ is set to 1000. In the second phase, all candidate patterns are hashed to the hash table. In the third phase, each candidate pattern is used to probe the hash table to calculate its occurrence number.

We observed three shortcomings in the framework.

1. The hash functions (1) do not preserve accurate information of the data. It is not easy to determine an optimal value for $Multiplier$. Furthermore, once $Multiplier$ is fixed, the information stored in the hash table only approximates the raw data.

2. The hash function is not suitable for answering queries that allow variable ranges of tolerable errors when matching the points.

3. False matches have to be filtered out. Our experiments indicated that almost one quarter of the retrieved tuples are false matches. For large point set, efficiency deteriorates severely.

The first two shortcomings limit application of the framework to bioinformatics. Due to regularity of biological and chemical structures, dissimilarity is often very subtle. Inaccuracy introduced by the scanning devices adds noise to the data. It is extremely difficult to choose a fixed ϵ, i.e. the range of tolerable errors, especially, when the data are collected by different domain experts, using different equipments, such as in the case of Protein Data Bank [6]. It is highly desirable that ϵ be set to a tunable parameter, so that the domain expert can choose an optimal value according to the context. The ΔB^+ Trees overcome these shortcomings.

5. Experimental Results

We have implemented both the ΔB^+ Tree technique and the geometric hashing technique using GNU C++ Language. All experiments were conducted on a Sun Ultra10 workstation with 440 MHz CPU and 512 Megabyte memory. Two data sets were used in the experiments. The first data set includes 20 randomly generated point sets, each has 15,000 points. Since randomly generated point sets are rarely similar to each other, to make the experiments more interesting, we generated 2 point sets first. We then moved the X-coordinate of half of the points in each point set 0.001, 0.002, 0.003, 0.004, 0.01, 0.02, 0.03, and 0.04 respectively to generate 8 copies of these point sets. We generated 2 more point sets to make the total of 20 point sets. The second data set includes 140 proteins downloaded from the Protein Data Bank [6]. The proteins have 1,000 atoms on the average. We segmented both the point sets and the proteins to consecutive substructures of M points, namely the size of the candidate patterns was M. The parameters of ΔB^+ Tree were $PageSize = 1024$, $InternalNodeSize = 84$, and $LeafSize = 63$. The three primes of the geometric hashing framework were 276527, 387659, and 498761. $Nrow$ was 251 and $Multiplier$ was 1000.

We also implemented a typical bulkloading algorithm that keeps inserting sorted leaf entries into the rightmost path from the root. The heights of the ΔB^+ Tree for both data sets were 4. The heights of the ΔB^+ Tree with bulkloading were also 4 for both data sets, even though the ΔB^+ Tree files were much smaller. Our first observation in the synthetic data was that, although the size of the header file for geometric hashing was twice of the number of distinct keys, the number of non-empty bins was 23.1% less. These keys must have been hashed to the same hash bins of some other keys. In other words, about 23.1% of the matches were false matches in the third phase. Similarly, for the protein data, the header file for geometric hashing was 1.9 times larger than the number of distinct keys, while the number of non-empty bins was also 23.1% less.

Our first experiment compared ΔB^+ Tree with geometric hashing in terms of their response time in the third phase. Figure 1 shows response time as a function of M, the size of the candidate patterns for the synthetic data. Figure 2 shows the response time as a function of M for the proteins. Since all the variants had very similar response time, we only pictured one of them. The ΔB^+ Tree with bulkloading also performed very similarly, probably because they had the same heights.

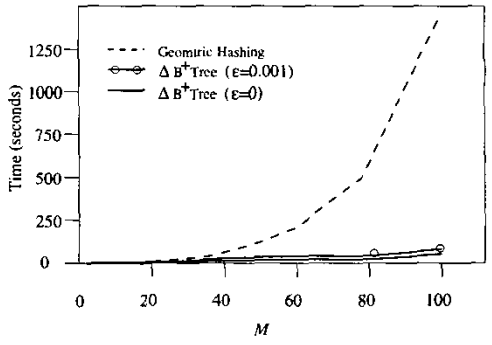

Figure 1. Response time as a function of M for the synthetic data

In the second experiment, we evaluated the performance of the ΔB^+ Tree technique with variable ranges of tolerable errors. We compared ΔB^+ Tree with its three variants. We fixed the size of the candidate patterns to 15. Figure 3 shows response time as a function of ϵ for the synthetic data. Figure 4 shows response time as a function of ϵ for the proteins. For all the variants, the ΔB^+ Tree with bulkloading did slightly better in answering range queries. We did not show the results here.

Notice that the response time for the protein data increased very fast when ϵ increased. This most likely is

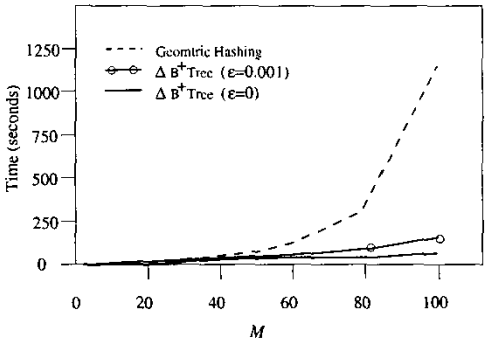

Figure 2. Response time as a function of M for the proteins

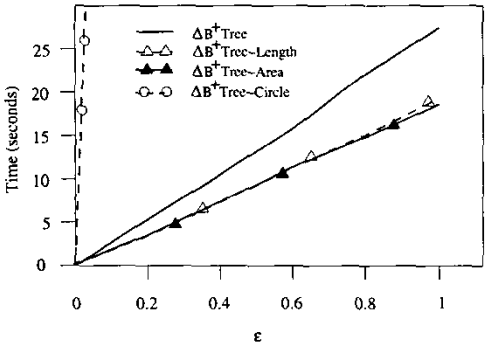

Figure 3. Response time as a function of ϵ for the synthetic data

caused by the fact that proteins are very similar in terms of inter-atom distances. There are some typical values of the bonds. This phenomenon makes it very difficult to set an optimal ϵ that is both large enough to cover data errors and small enough to detect the subtle differences among the data. This is the major motivation triggering the development of ΔB^+ Tree.

6. Conclusion

We have proposed a new index structure called ΔB^+ Tree. ΔB^+ Tree performs triplet matching and merging like in [4, 5]. In stead of using geometric hashing, we use a B^+ Tree to store the triplet information. The proposed indexing technique has all the advantages a B^+ Tree has compared with hashing techniques. We introduced three variants of ΔB^+ Tree and evaluate the effectiveness of them on both synthetic data and real data.

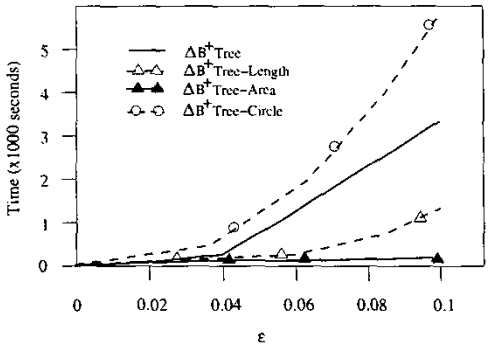

Figure 4. Response time as a function of ϵ for the proteins

Our future work includes using three dimensional R*-trees and other index structures to index the triplets and conducting experiments to compare the performance. We will conduct large scale experiments to index the Protein Data Bank for pattern discovery.

References

[1] S. Berchtold, C. Bhm, B. Braunmller, D. Keim, and H. Kriegel. Fast parallel similarity search in multimedia databases. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 1–12, Tucson, Arizona, 1997.

[2] J. Rao and K. A. Ross. Making b^+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486, Dallas, TX, 2000.

[3] X. Wang. α - surface and its application to mining protein data. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 659–662, San Jose, California, 2001.

[4] X. Wang, J. Wang, D. Shasha, B. Shapiro, I. Rigoutsos, and K. Zhang. Finding patterns in three dimensional graphs: Algorithms and applications to scientific data mining. *IEEE Transaction on Knowledge and Data Engineering*, 14(4):731–749, July/August 2002.

[5] X. Wang, J. T. L. Wang, D. Shasha, B. A. Shapiro, S. Dikshitulu, I. Rigoutsos, and K. Zhang. Automated discovery of active motifs in three dimensional molecules. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 89–95, Newport Beach, California, 1997.

[6] J. Westbrook, Z. Feng, S. Jain, T. N. Bhat, N. Thanki, V. Ravichandran, G. L. Gilliland, W. Bluhm, H. Weissig, D. S. Greer, P. E. Bourne, and H. M. Berman. The protein data bank: unifying the archive. *Nucleic Acids Research*, 30(1):245–248, 2002.