ABSTRACT

The richness of the XML data format allows data to be structured in a way which precisely captures the semantics required by the author. It is the structure of the data, however, which forms the basis of all XML query languages. Without at least some notion of the structure, a user cannot meaningfully query the data. This problem is compounded when one considers that heterogeneous data adhering to different schema are likely to exist in the database(s) being queried. This paper proposes a solution based on an efficient proximity index. In particular, we describe a family of encoding and compression schemes which enable us to build an index to efficiently implement the proximity search. Our index is extremely small, and can reflect updates in the underlying database in modest time. Experiments show that our algorithm and implementation are fast and scale well.

1. INTRODUCTION

The richness of the XML data format allows data to be structured in a way which precisely captures the semantics required by the author(s) [2]. Such richness, however, poses substantial barriers to casual users and non-domain experts who wish to query this data. These people may know "what" they want, but may not know how that data is structured. As it is the structure of the data which forms the basis of all XML query languages [1], without at least some notion of the structure, a user cannot meaningfully query the data. This problem is compounded when one considers that heterogeneous data adhering to different schema are likely to be included in the same database (or even the same document) [7].

Suppose, for example, we are looking for trends in "insurance claims" related to "smoking". The information we are after may be contained in insurance company records, court transcripts, or even newspaper articles. Even if we decide we are only interested in examining court transcripts, we do not know the structural relationship between the terms of interest. We are left in the predicament of knowing exactly what we are looking for, but not knowing how to find it.

A number of solutions to this problem have already been explored. Naturally, all these solutions are less precise than an exact database query where the structure of the data is known. However, an increasing number of users are engaging in interactive searches, such as online web based searching, where queries are successively refined until the desired information is located. In such circumstances, a useful and accepted approach is to provide approximate or "likely" answers which can be refined by the user.

Previous approaches have adapted methods from information retrieval (IR), such as keyword searching [4] and building traditional IR indexes for different element types [6]. Such approaches, however, typically do not address index updates, and fail to fully consider the structure of the document. Navarro and Baeza-Yates [9] have proposed a generic model in which proximal nodes are used to query databases by content and structure. Their paper, however, does not focus on implementation and system details, efficiency, indexing, etc.

Goldman et al [5] have proposed a method for ranking one set of nodes based on their proximity (in the structural sense) to another set of nodes. Whilst their approach is useful when the structure is unknown, it relies on pre-computed values to determine proximity, and does not provide a mechanism for updating the index.

In our approach, we extend this idea of ranking results based on proximity. To increase the power of the proximity ranking, we enable results to be ranked according to either ascending or descending proximity. Rather than pre-compute distances, we propose a method of dynamically calculating distances as required.

Suppose, for example, we are interested in locating a "restaurant" in "Soho". As we are entering this request over the internet, we have no idea of the underlying structure of the data. Indeed, it is very possible that the web site accesses data from many different sources with vastly differing structures. For this reason we cannot easily issue a meaningful database query. However, if we can find "restaurants", ranked by proximity to the nearest occurrence of "Soho", we are likely to find what we are looking for.

Suppose further that we do not want any restaurant which serves seafood. We could refine the previous results by looking for all such "restaurants" not near "seafood". This raises the question, however, of how to determine what is meant by "near". We could decide that everything beneath some
arbitrary threshold is "near", or decide upon some other means of distinguishing between "near" and "far". This, however, entails the problem of imposing an artificial and possibly incorrect structure on the data, with the increased likelihood of returning inappropriate results. If we instead rank everything in decreasing order of proximity, so that the "restaurant" furthest away from "seafood" is listed first, we are likely to find a restaurant we want without the risk of distorting the underlying data.

For structural proximity determination, we consider the XML database (such as a Lore database [8]) as a general graph, comprised of one or more individual data sets. Our proximity search requires that we be able to find the distance between any two nodes in an arbitrary graph. Such graphs may easily contain a huge number of nodes (easily Θ(10^6) nodes for a database of 1GB). Any proximity search thus potentially contains a large number of calculations, and a single query may be comprised of more than one proximity search. Given that the proximity search itself is only part of the entire query processing, the speed of such determination is especially crucial.

In this paper we present a mechanism for efficiently executing such a proximity search in near linear time. We present a method for encoding graphs, and a family of encoding schemes for representing this information in a compressed space. Our encoding schemes are specifically designed not only to be as small as possible, but to facilitate the direct calculation of proximity. We describe further optimisations to perform these comparisons very quickly (Ranking 100,000 nodes according to their proximity to the closest element of a 100,000 node set in 0.44 seconds). Our index is extremely small, and can reflect updates in the underlying database in modest time. Experiments show that our algorithm and implementation are very fast and scale well.

In section 2 we discuss our conceptual model and formalise the problem. In section 3 we give an overview of our approach. In section 4 we informally describe our mechanism for encoding subgraphs, and the data structure used to hold the encoding, and begin to discuss how these combine to facilitate proximity determination. In section 5 we describe our proximity determination algorithm in detail. Section 6 discusses how updates are dealt with. Finally, section 7 presents the experimental results and section 8 concludes the paper.

2. CONCEPTUAL MODEL

A well defined and precise correspondence exists between a single XML document and its representation as a tree. This idea can easily be extended to a collection of documents by the addition of a root node, of which all documents are children. Similarly, multiple XML repositories can be combined with the inclusion of a further root node.

A collection of XML documents can therefore be viewed as a single document. We are unconcerned, therefore, whether the desired information is contained in a single document, in multiple documents in the same database, or in multiple databases. For this reason, we assume that all the XML data of interest has a single, common root.

Although the raw XML structure is guaranteed to be a tree, its logical or interpreted structure may not be. XML documents may contain links to portions of the same or other documents. Such links indicate some close relationship (in the semantic or logical sense) between the two portions, as determined by the author of the document. As we are concerned with determining proximity in the semantic and logical sense, it makes sense to materialise these links, and consider them as actual directed edges. We thus consider the XML repository as a directed, possibly cyclic, graph.

2.1 The System

A native XML database system called Soda2 [10] is used to test our prototype index engine. Soda2 is similar to Lore [8] except that it is designed and implemented with DOM as its physical model and XQL as its default query language. To process proximity queries, Soda2 evaluates two queries (the "Find" and "Near" queries) as normal. The results of these queries are passed to the proximity engine, which utilises the proximity index to rank the objects returned by the "Find" query. Database objects are opaque to the proximity engine, which only deals with object identifiers. The proximity engine retrieves the encodings for each object in constant time. These encodings are then used to compute the inter-object distances, again, in near constant time. Finally, objects that are ranked according to their inter-object distances are returned.

2.2 The Problem

Informally, we are looking for all F near N, where F and N each represent a set of nodes. We refer to F as the Find Set (i.e. what we want to find), and N the Near Set (i.e. what it is near).

Formally, we wish to rank all elements of the Find Set by their proximity to the nearest element of the Near Set.

According to this definition, we must be able to rank elements of the Find Set in both ascending and descending order. Proximity can be naturally defined as the shortest path between the two nodes.

If we consider, however, that the purpose of the proximity search is to find nodes which are semantically or conceptually related, it may make sense only to consider paths which pass through common ancestors. Common ancestors can be seen as concepts which include both target nodes. Proximity can then be defined as shortest path between two points, considering only paths through common ancestors.

For our implementation, we have chosen this second definition of proximity, with good results. Our mechanism works equally well for either definition of proximity, the former requiring slightly longer time when updating the index.

3. OVERVIEW

There are two fundamental approaches one can take to this problem. On one hand, we could pre-compute all pairwise shortest distances, and look these up as required. This method has the advantage of retrieving any distance in constant time. Algorithms which employ this method, however, necessarily involve O(|F| × |N|) comparisons. Furthermore, such pre-computed indexes are very large (|V|^2 in the worst case for a graph with V vertices), although methods have been proposed to address this [5]. Updating the index to reflect changes in the database is also expensive. The underlying database needs to be extensively examined to determine all shortest distances involving the modified node.

The other approach is to calculate distances as required, using a graph algorithm. This has the advantage of virtually no overhead to reflect changes to the database, as well as much more reasonable space requirements (O(|V|). However,
as any graph algorithm requires arbitrary traversal through an arbitrary graph, such an algorithm could require \(O(|V| \times (|F| + |N|))\) random disk seeks in the worst case. Thus this solution is unfeasible in practice.

Our approach fundamentally falls into the second category, computing the distances as required using a graph algorithm. Instead of directly examining the graph, however, we use a family of encoding schemes to represent the relevant subgraphs in a very small space (typically no more than 20 bytes for a single subgraph). The distance is then calculated by directly comparing these encodings. As the encodings are so small, the entire subgraph comparison can be performed in main memory, often utilising only the CPU cache. As the comparisons themselves heavily utilise bitwise comparisons and optimisations, distance calculations are performed very quickly.

The index contains an entry for each node in the graph. This entry contains the encoded subgraph containing all paths from the root to the given node. As many XML documents (or portions of these documents) are trees, many of these subgraphs will be a single path.

Proximity is determined using the following approach:

1. Obtain encoded subgraph containing all elements of \(\text{Near Set}\)
2. Compare the encoded subgraph of each element of \(\text{Find Set}\) with the encoding obtained in step [1]

The encoded subgraph obtained in step [1] can either be generated dynamically or retrieved from a cache. Dynamic generation is achieved by retrieving the encoding for each element of \(\text{Near Set}\) and combining them into a single subgraph. Again, this process only involves the encodings themselves, and not the underlying graph. This process heavily utilises bitwise operations and optimisations, giving very fast performance. If dynamic generation of this encoding is required, this process approaches \(O(|N|)\) in practice.

For each element of \(\text{Find Set}\), step [2] involves a comparison with the encoded subgraph obtained in step [1]. Conceptually, this step involves “overlaying” the subgraph for the \(\text{Find Set}\) element with the subgraph of the \(\text{Near Set}\), and seeing where they diverge. In practice, this overlaying is done using bitwise comparisons, and so many edges are typically compared in a single, cheap operation. In practice, this step tends toward \(O(|F|)\).

Utilising this approach, we avoid the need for performing \(O(|F| \times |N|)\) comparisons. In practice, this approach tends toward \(O(|F| + |N|)\) comparisons if the subgraph in step [1] must be dynamically generated, or \(O(|F|)\) comparisons if the subgraph in step [1] is retrieved from the cache.

4. REPRESENTING SUBGRAPHS

In order to efficiently encode a subgraph, each edge in the main graph is assigned the smallest unused positive number which is unique only amongst all edges originating from a given node. This means that two edges can be assigned the same number as long as they originate from different nodes. This number is referred to as the edge identifier.

This decision has the important consequence that all such numbers will be relatively small compared to the total number of edges/vertices in the graph.

4.1 Representing Single Paths

![Figure 1: Graph with Indicated Edge Identifiers](image)

Paths in the main graph are identified by the sequence of individual edge identifiers, which implicitly start from a (virtual) incoming edge to the root. Nodes are identified as being the terminus of one or more paths. This concept is illustrated in figure 1. The node \(y_1\) is identified by the sequence of edge identifiers "1.1.2". Note that this sequence of edge identifiers both uniquely identifies the node itself and the path from the root to the node.

Our encoding scheme exploits the low numerical value of the edge identifiers, by only allocating twice the minimum space required to store the numbers. For example, as the number "1" is represented by 1 bit, and the number "2" by 2 bits, the path "1.1.2" is represented in only 8 bits (2 \(\times\) 4 bits). This approach offers a great space saving over methods which typically use a 4 byte integer to represent each node (thus requiring 12 bytes instead of 1 to represent the previous path).

4.2 Representing Multiple Paths

The method described in section 4.1 is extended to represent general subgraphs. To deal with multiple paths in a subgraph, we specially number nodes which contain either more than 2 incoming or more than 2 outgoing edges, within a single subgraph. Note that we are not concerned about the total number of incoming and outgoing edges from a node. We are only concerned with the number of incoming and outgoing edges which are included in the subgraph of interest. This is illustrated in figure 1 when considering all paths from the root to \(y_2\), by the nodes labeled "A" and "B". Note that even though many nodes in the graph have more than 2 incoming or outgoing edges, within the subgraph containing all paths from the root to \(y_2\), there are only 2 such nodes.

Such nodes (referred to as common nodes) are numbered separately from the edge identifier numbering. In figure 1, the node \(y_2\) is labeled "A" for clarity. In the encoding scheme, \(y_2\) is implemented as the number "1", with a marker bit set to indicate this number refers to a common node and not an edge identifier. Common nodes are given numbers which are unique within the subgraph being encoded.

The method of labeling multiple paths can now be seen.

1. The method used in section 4.1 (path encoding) is used as long as the subgraph contains only a single path.
2. Each common node is included in the encoding using a marker bit.

3. Each path encoding segment (except the first) is assumed to originate from the previous common node in the encoding. The first path encoding segment is assumed to begin with the (virtual) incoming root edge.

4. Each path encoding segment is followed by a number which indicates the common node where the encoded segment terminates. This number is differentiated from an edge identifier by utilising a special “marker bit”.

The entire encoding for the subgraph of all paths from the root to \( y_2 \) is therefore given by the mapping:

\[
1.3 \rightarrow A \rightarrow A.1.1. \rightarrow B.2.1. \rightarrow B.B.1. \rightarrow A
\]  

(1)

where \( A \) indicates a common node, and \( \rightarrow A \) indicates that a path encoding segment terminates at common node \( A \).

Tracing this example from left to right, we see that 1.3 corresponds to the edge identifiers from the root to node \( A \). The fact that this path encoding segment terminates at node \( A \) is indicated by \( \rightarrow A \).

The next \( A \) we encounter indicates that we have come to an common node in the encoding. All path encoding segments which exist between this common node and the next (indicated by \( B \)) are assumed to originate at node \( A \). In this example, we see that the next 2 such segments, 1.1 and 2.1, both originate at common node \( A \) (the previous annotated node), and both terminate at common node \( B \) (indicated by \( \rightarrow B \)). Similarly we see the next common node \( B \), followed by the path which originates from node \( B \) and terminates at node \( A \) (indicated by \( 1.\rightarrow A \)).

Given that each of these numbers are represented in the minimum possible space, the entire subgraph is represented in only 48 bits (shown in figure 2).

### 4.3 Representing Multiple Elements

So far the encoding methods we have looked at have all had only a single node of interest (the terminal node of all the paths). However, if we wish to encode the subgraph containing the *near set* in such a way as to determine distance, we need to have some mechanism for representing a subgraph with multiple terminal nodes. We achieve this with a very minor addition to the method described in section 4.2, which additionally helps to make distance determination more efficient.

As we want to find distances quickly, we wish to avoid the need for traversing all paths of a subgraph. Indeed, doing so would make this technique less efficient than many others. To alleviate this need, we pre-compute the minimum distance from each common node to the terminal node, considering only forward edges. (If the first definition of proximity in section 2.2 is chosen, this pre-computed distance must consider all edges). We store this pre-computed value in the encoding, immediately after the occurrence of the common node as the path origin. (\( A \) or \( B \) without the arrows in mapping (1)). Note that this pre-computed distance is local to the encoded subgraph. Changes in the database only need to be reflected here if the encoded subgraph is involved. Even when this does occur, the pre-computed distance is local, and the value can be directly computed from the encoding. As such, re-computing this distance is relatively cheap.

Pre-computed values are included for both the multi-path encoding described in section 4.2 and the multi-element encoding described here.

Once we have this notion of including pre-computed distances, identifying multiple terminal nodes becomes easy. For multi-path encoding, we only include the pre-computed value for common nodes. For multiple terminal nodes we include pre-computed values for both common nodes and terminal nodes. A terminal node is therefore identified as any node with a pre-computed distance of zero. As both common nodes and terminal nodes have the shortest distance explicitly stored, they are collectively referred to as annotated nodes. When dealing with multiple elements, common nodes and terminal numbers are assigned identifying numbers so as to be unique amongst annotated nodes.

### 4.4 Compressed Arrays

In order to efficiently implement this encoding scheme, we utilise a data structure called a *compressed array*. Compressed arrays are designed to store numbers in a small amount of space, in a manner which supports our encoding mechanisms. Operations on compressed arrays make heavy use of bitwise operations and optimisations for increased efficiency. A compressed array is shown in figure 2, storing the encoding (from right to left) shown in mapping (1).

Conceptually, a compressed array can be thought of as a pair of parallel bit patterns. One pattern, the *identifier pattern*, contains the bit patterns necessary for representing numbers. The second bit pattern, the *boundary pattern*, contains a set bit which denotes the boundary (or most significant bit) of a particular cell. This has the useful property that the number of elements in a compressed array can be determined by counting the number of set bits in the boundary pattern. Looking ahead a little, each such set bit of a path encoding corresponds to an edge in the graph. Counting the number of set bits in the encoding therefore corresponds to counting the number of edges. Using a non-iterative bit counting algorithm, we are therefore able to count the number of edges in an encoding in constant time.

The purpose of the boundary pattern is to determine the range of bits which are relevant for an individual cell. Considering only these bits in the identifier pattern thus yields the value of an individual cell. As cells are generally only "wide" enough to contain necessary bits for representing a given number (e.g. 1 bit wide for the number "1", 2 bits wide for the number "3"), they frequently have different widths. These features are clearly seen in figure 2, where each "cell" is indicated by different shading.

We can now see how the encodings are actually stored in the data structure. Figure 2 shows the encoding (from right to left) shown in mapping (1). Note that first 3 bits of this encoding correspond to a path encoding segment (1.3). Edge identifiers (used in path encoding segments) are always stored in cells of minimum width. As edge identifiers are guaranteed to be positive numbers, this has the useful property that for all cells in a path encoding segment, both the most significant bit in the identifier pattern and the corresponding bit in the boundary pattern are always set.

In section 4.2 we mentioned that a special bit marker was included to indicate that a path terminates at an annotated node. This is achieved by including a single leading zero (bit indexes 4, 11, 17 and 23 in figure 2). Thus, if a cell has a single leading zero, it indicates that the number iden-
5. PROXIMITY DETERMINATION

The most fundamental algorithm, described in section 5.1, identifies where two paths diverge in constant time. The main algorithm, described in section 5.2, contains a loop. In the worst case, this executes $E$ times, where the subgraph representing the element of Find Set has $E$ edges. In practice, however, this rarely occurs. Wherever possible, the algorithm considers multiple edges in a single execution of a loop cycle. Furthermore, loops are only repeated if the path segment being considered is exactly the same in both Near Set, $N$, and the element from Find Set, $F$. Even though in the worst case this algorithm is $O(|E| \times |F|)$ for the entire Find Set, in practice, performance tends to be $O(|F|)$.

5.1 Path Divergence

The path divergence algorithm takes constant time to determine where two paths diverge, and is shown in figure 3. Understanding the algorithm is much easier if we recall that for path encoding, every set bit in the boundary pattern of a compressed array corresponds to an edge in the graph.

Step [1] finds the first bit where the compressed arrays differ, considering them from the start of the compressed array (which corresponds to the incoming root edge). Conceptually, this corresponds to the outgoing edges of the node where the paths diverge. Note that we are unconcerned about the value of the numbers stored in the compressed array. From the perspective of this algorithm, the only purpose of unique edge identifiers is to guarantee different bit patterns where appropriate. By treating the compressed arrays as bit patterns, we can efficiently compare and count many edges in a single comparison.

LowBit($n$) is a non-iterative algorithm which returns the index of the lowest set bit in $n$, or -1 if $n = 0$. Step [2] thus returns the index where the compressed arrays diverge.

The input compressed arrays for this algorithm contain "normalised" path encoding segments. This means that the compressed arrays contain only path encoding segments, both starting at the least significant bit. In practice, these segments are often extracted from multi-path encoding. For a path encoding segment ranging from bit index $m$ to $n$, this is achieved by performing the following bitwise operation on $ca.bound$ (and symmetrically on $ca.ident$).

$$ca_{tmp}.bound = (ca.bound AND \_lmn(1 << n)) >> m$$

where $\_lmn(k)$ returns a bit mask which sets all bits less than or equal to the least significant set bit of $k$, and unsets all other bits. This allows us to unset all bits above the highest bit we are interested in. $\_lmn(n)$ is implemented by:

$$\_lmn(n) = NOT( n XOR NOT(n - 1))$$

5.2 Multi-Element Comparison

This algorithm finds the proximity of the element of Find Set to the nearest element of the encoded Near Set. The
Algorithm: Path Divergence

Input: Two compressed arrays, \(a_1\) and \(a_2\), which contain the normalised path encoding segments terminating at Node 1 and Node 2 respectively

Output: \(DPC_{\text{index}}\), the offset + 1 (in bits) from the start of \(a_1\) and \(a_2\) where the paths diverge (if they diverge),
0, if the paths do not diverge.

\[\text{MaxDiff} = (a_1, \text{bound XOR} a_2, \text{bound}) \text{ OR } (a_1, \text{ident XOR} a_2, \text{ident})\]

\[DPC_{\text{index}} = \text{LowBit} (\text{MaxDiff}) + 1\]

Figure 3: Path Divergence Algorithm

The algorithm is shown in figure 4. Conceptually, this algorithm works as follows:

a. The path to the element from \(\text{Find Set}\) is "overlayed" on the subgraph containing the \(\text{Near Set}\) until an annotated node (AN) is reached.

b. Our next actions depend on which graph the AN is in.

c. If the AN is only in the encoding for \(\text{Near Set}\),

   (i) First check if the paths diverge before the AN is reached. If they do, calculate the minimum distance. Continue considering other queued paths.
   (ii) Otherwise, store the minimum distance from this AN to the nearest terminal node (in \(\text{Near Set}\)). We then continue following the appropriate edge in \(\text{Find Set}\), if it exists. If it doesn’t exist, this means the paths diverge at this AN. In this case, we calculate the shortest distance for this path, and continue considering the other queued paths.

d. If the AN is only in the encoding for the \(\text{Find Set}\), we perform symmetrical steps to step c.

e. If we come to ANs in both \(\text{Find Set}\) and \(\text{Near Set}\), we also perform symmetric steps to step c. However, in this situation we also have the possibility that more than a single path will be found in common. For this reason, we check each edge which leaves the AN node in \(\text{Find Set}\). If this edge also appears in \(\text{Near Set}\), we add that edge to a queue. We also have the possibility that this node indicates a cycle. For this reason, we only add the edge to the queue if we have not visited this node before.


Many of the steps above require to store the minimum distance between the element of \(\text{Find Set}\) and the nearest element of \(\text{Near Set}\). Strictly speaking, for each iteration of the loop, we progressively obtain the minimum distance out of all paths considered so far. This allows us to use the second definition of proximity in section 2.2, which considers only paths through common ancestors.

When calculating distance, we are not sure if the shortest path passes through the previous annotated node we visited, or the next annotated node we would visit (or are visiting). We deal with this by calculating the minimum distance from the point of divergence to the element of \(\text{Find Set}\), the minimum distance from the point of divergence to the nearest element of \(\text{Near Set}\), and then summing these distances.

Each minimum distance is determined by considering the path through the previous (or next) annotated node and the number of edges between that node and the point of divergence. As we store the minimum distance at each annotated node, this calculation is simply a combination of retrieving the stored distance, the path divergence algorithm described in section 5.1, and a non-iterative bit counting algorithm. This method is illustrated step [9]. Note that in step [7], \(DPC_{\text{Index}}\) is assigned one greater than the index of the point of divergence. Thus, \(DPC_{\text{Index}} - 1\) is the index of the point of divergence, and \(DPC_{\text{Index}} - 2\) is the index prior to the point of divergence. The statement in step [9] thus conciselyAchieves the operations described above.

6. UPDATES

Updates to the index are performed in the following steps:

1. Obtain encodings affected by the changes
2. Reflect changes in the encoding schemes
3. Re-calculate shortest distances (if necessary).

Perhaps surprisingly, if no new mechanism is introduced, the most expensive step in this process is step 1. Steps 2 and 3 modify existing graph algorithms. In practice, most subgraphs are single paths, or possibly DAGs. This means that whilst the worst case performance may not be good, the average case performance is indeed very good. With the optimisations and increased speed provided by our encoding schemes and bitwise operations, these algorithms have increased performance in practice.

To minimise the cost of step 1 above, we build an alternative index on the encodings, which allows indexes to be retrieved based on included nodes, rather than just on terminal nodes.

Reflecting changes in the encodings is efficiently performed by utilising optimised algorithms which cover the necessary 4 primitive structural operations.
If necessary, pre-computed distances are re-calculated using Dijkstra’s single-source shortest path algorithm [3], modified to work with our encodings and exploit all the bitwise optimisations our encodings provide. As such calculations need only consider the encodings (which can always fit into main memory), such re-calculation is very fast in practice.

7. EXPERIMENTAL RESULTS

We have implemented this system on a Pentium III 800MHz processor with 256 MB RAM running Linux Redhat 7.0. We generated XML data sets with random topology. Multiple data sets were generated and average performance were obtained by repeatedly running the program with different data sets. Choosing the metric of database size is somewhat misleading for such examples, as our index considers not only the structure itself. A document with 100 nodes can be as small as 1 KB or as large as 1 MB. To give a fair indication, therefore, we consider only the number of nodes in the graph, without any reference to the size of the data stored at a single node.

Figure 5 shows a comparison of the number of nodes and the size. We differentiate between the size of the encoding and the total size, which includes the space overhead of implementing the hash table. One reason for the precise linear nature of the space measurements is our implementation. In our implementation, we store the actual encoding in a multiple of 8 bytes (corresponding to 2 x 4 byte unsigned integers). Thus, if the actual encoding only requires 24 bits, we still use 8 bytes for storage. This is done for performance considerations. A result of this, however, is that whilst different graph topologies do have different space requirements, storing encodings in multiples of 8 bytes tends to “smooth” out this variation, resulting in the linear space measurements.

<table>
<thead>
<tr>
<th># Nodes</th>
<th>Total space for encodings</th>
<th>Total space for index</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>78 KB</td>
<td>159 KB</td>
</tr>
<tr>
<td>100,000</td>
<td>780 KB</td>
<td>1.55 MB</td>
</tr>
<tr>
<td>1,000,000</td>
<td>7.6 MB</td>
<td>15.5 MB</td>
</tr>
</tbody>
</table>

Figure 5: Number of Nodes vs Size

Figure 6(a) shows the time taken to perform the proximity ranking for various sizes of Near Set and Find Set, when the Near Set is cached. As can be seen, the time is linear in practice for this case. It is worth noting that for any method which requires \(O(|F| \times |N|)\) comparisons, then comparing Find Set and Near Set of 100,000 nodes each would require \(O(10^{10})\) comparisons. We manage to achieve the same result in 0.44 seconds.

Figure 6(b) shows the time taken to perform the proximity ranking for various sizes of Near Set and Find Set, when the Near Set is uncached. The increase in time is taken by the need to generate the single encoding of the Near Set. Once again, the time taken to generate the encoding for the Near Set is approximately linear. The larger jump which occurs as the size of Near Set and Find Set increase is due to the introduction of random disk accesses and paging caused by the query processing. The overall performance, however, is still impressive in practice.

8. CONCLUSIONS

In this paper we have presented a method for implementing a fast, efficient proximity search in near linear time. We have described a method of encoding graphs, and a family of encoding schemes to represent this information. The encoding schemes are focused on compressing the information as much as possible, whilst at the same time facilitating proximity determination. This is aided by the data structure we employ, which utilizes bitwise operations and optimisations to substantially reduce the time taken by the algorithm in practice. Our proximity determination algorithm exploits these bitwise operations and the ability to compare many edges in a single pass, to provide an algorithm that is near linear in practice. We show how to handle updates in modest time, and our experimental results prove that size and space performance are both good and scalable.

For future work, we are considering the following directions:

- We are exploring useful ways to integrate this proximity search into a Near operator. One challenge lies in knowing how to implement an operator that deals with ranked results, rather than just sets.
- We are considering ways to increase the performance of updates to the index.

9. REFERENCES

Algorithm: Multi Element Comparison

Input: A compressed array, \(ca_N\), containing the multi-element encoding of the Near Set.
A compressed array, \(ca_F\), containing the multi-path encoding or path encoding of all paths from the root to the specified element of the Find Set, \(E_F\).

Output: \(dist\), the shortest path from \(E_F\) to the closest element in Near Set

Notes: In step [3], we assume the path encoding segment found is bounded by annotated nodes \(A_i\) and \(A_j\), and goes from bit indexes \(m\) to \(n\).
In step [5], we assume the path encoding segment found bounded by annotated nodes \(A_g\) and \(A_h\), and goes from bit indexes \(o\) to \(p\).

1. Initialise \(Prev_Tmp_Dist_F = Prev_Tmp_Dist_N = MAXINT\), \(PE_{Start_F} = PE_{Start_N} = 0\)
2. Loop
3. Find \(PE_F\), the maximal path encoding segment starting from \(PE_{Start_F}\)
4. \(Next_Tmp_Dist_F = MIN(Prev_Tmp_Dist + No\ of\ set\ bits\ in\ ca_F.bound\ between\ m\ and\ n,\)
   
   Stored distance for \(A_j\)
6. If \(Length(PE_F) > Length(PE_N)\)
7. \(DPCIndex = Index\ of\ path\ divergence\ +\ 1,\)
   
   (found using path divergence algorithm), for corresponding portions of \(ca_F\) and \(ca_N\).
8. If \(DPCIndex > 0\)
9. \(Find_Dist = MIN(Next_Tmp_Dist_F + No\ of\ set\ bits\ in\ ca_F.bound\ between\ (DPCIndex - 1)\ and\ n,\)
   
   \(Prev_Tmp_Dist + No\ of\ set\ bits\ in\ ca_F.bound\ between\ m\ and\ (DPCIndex - 2)\)
10. Symmetric step to [9] for \(ca_N\)
11. \(dist = MIN(dist, Find_Dist + Near_Dist)\)
12. If no paths in queue, exit loop. Otherwise, restore state from head of queue and continue loop.
13. \(Find_Dist = MIN(Prev_Tmp_Dist_F + No\ of\ set\ bits\ in\ ca_F.bound\ between\ m\ and\ m + p - 0,\)
   
   Stored distance for \(A_j\) + No of set bits in \(ca_F.bound\ between\ m + p - o + 1\ and\ n)\)
14. \(Near_Dist = Next_Tmp_Dist_N\)
15. \(dist = MIN(dist, Find_Dist + Near_Dist)\)
16. \(Next_Edge_ID = element\ in\ ca_F\ starting\ at\ index\ m + p - o + 1\)
17. If a path originating from \(A_j\) has initial edge identifier \(Next_Edge_ID\)
18. \(PE_{Start_F} = m + p - o + 1,\) Reset values to correspond to new path encoding segment.
else
19. If no paths in queue, exit loop. Otherwise, restore state from head of queue and continue loop.
else if \(Length(PE_F) < Length(PE_N)\), perform symmetric steps to [7] through [16]
else // \(Length(PE_F) = Length(PE_N)\)
24. If we have visited this annotated node before
25. If no paths in queue, exit loop. Otherwise, restore state from head of queue and continue loop.
26. For each path encoding segment originating from \(A_j\) with initial edge identifier \(Next_Edge_ID\)
27. If path with initial edge identifier \(Next_Edge_ID\) originates fro \(A_h\), add edge and current state to queue.
28. If no paths in queue, exit loop. Otherwise, restore state from head of queue and continue loop.

---

**Figure 4:** Multi Element Comparison Algorithm

<table>
<thead>
<tr>
<th># Nodes vs Speed (sec)</th>
<th># Nodes in Find Set - Near Set Cached</th>
<th># Nodes in Find Set - Near Set Uncached</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td># Nodes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>10</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>Near Set</td>
<td>100</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>Set</td>
<td>1,000</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>&lt; 0.001</td>
</tr>
</tbody>
</table>

**Figure 6:** Number of Nodes vs Speed