OPTIMIZING PROXIMITY DETERMINATION FOR SEMISTRUCTURED DATA

Raymond K. Wong *Michael Barg*

School of Computer Science & Engineering
University of New South Wales
Sydney, NSW 2052, Australia
{wong, mbarg}@cse.unsw.edu.au

1. INTRODUCTION

This paper summarizes our mechanism for efficiently executing a proximity search in near linear time for semistructured database of moderate size, e.g., hundreds of thousands of XML elements. This includes a method for encoding graphs, and a family of encoding schemes for representing this information in a compressed space. Readers may refer to [1] for detailed description and applications. In particular, the encoding schemes are specifically designed not only to be as small as possible, but to facilitate the direct calculation of proximity.

Informally, we are looking for all F near N, where F and N each represent a set of nodes. It is important to realise that F and N may be specified by some inexact criteria (for example, find all elements containing ”ticket” near all elements containing ”price”), and so may not be disjoint. 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). The problem can now formally be stated as follows: we wish to return all elements of the *Find Set*, ranked by their proximity to the *nearest* element of the *Near Set*.

When considering solutions to this problem, there are two fundamental approaches. On one hand, we could pre-compute all pairwise shortest distances, and then 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($|F| \times |N|$) 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 for minimising this problem [2]. 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 single modified node.

The other approach is to calculate distances as required, using some form of 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 impractical for any real implementation.

Our approach fundamentally falls into the second category, calculating 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.

We utilise a two phase approach to avoid the need for performing O($|F| \times |N|$) comparisons. In practice, our approach tends towards either O($|F| + |N|$) comparisons if the *Near Set* encoding must be generated dynamically, or O($|F|$) comparisons if it can be retrieved from the cache. Our encoding schemes and the algorithms which use them to determine proximity are described in detail in [1].

2. THE PROXIMITY INDEX

The efficiency of our index relies largely on our encoding schemes. Minimising space is important as it allows more of the index to be held in main memory. Looking ahead a little, our encoding scheme can represent all subgraphs from the root to each of 1,000,000 XML nodes in 7.6 MB. Given current systems, it is not unreasonable to hold this entirely in main memory. 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.

2.1. Representing Single Paths

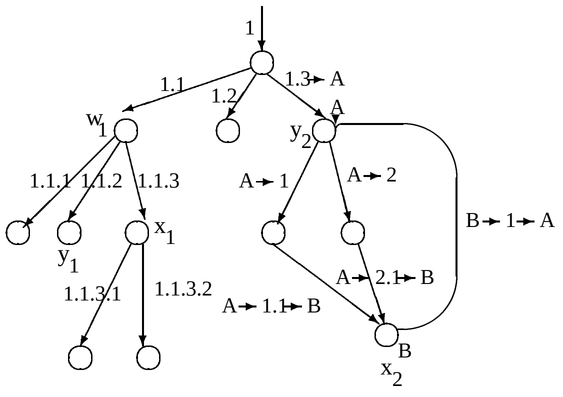

Figure 1: Graph with Indicated Edge Identifiers

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).

It is now possible to begin to see how the distance calculation works. As y_1 is encoded by ”1.1.2” and x_1 is encoded by ”1.1.3”, the distance between them can be determined by observing that the paths are the same for the first two edges (”1.1”), and so this contributes nothing to the shortest path between them. This information is found using a single bitwise exclusive or operation. After the paths diverge, the path from the root to y_1 contains 1 edge, as does the path from the root to x_1. This information is found using a non-iterative bit counting algorithm. We can thus determine the distance between these two nodes is 2 in constant time.

2.2. Representing Multiple Paths

The method described in section 2.1 is extended to represent general subgraphs in this section. Suppose we want to encode the subgraph containing all paths from the root to y_2. This must include not only the direct path to y_2, but the cycle from y_2 to itself. Obviously our method of listing edges sequentially is not sufficient when multiple paths are involved.

To deal with multiple paths in a subgraph, we number nodes which contain 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 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 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*. This is generally substantially smaller than the total number of such nodes within the entire graph. (Thus, for example, a different common node may also be identified as ”A” in a different subgraph).

The entire encoding for the subgraph of all paths from the root to y_2 is therefore given by:
1.2.$\rightarrow$A.A.1.1.$\rightarrow$B.2.1.$\rightarrow$B.B.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.

3. CONCLUSIONS

This extended abstract summarized the method that we proposed for implementing a fast, efficient proximity search in near linear time. The encoding scheme is 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 utilises bitwise operations and optimisations to substantially reduce the time taken by the algorithm in practice.

4. REFERENCES

[1] M. Barg, and R.K. Wong. Structural Proximity Searching for Large Collections of Semi-Structured Data. Technical Report, CSE, University of New South Wales, 2001.

[2] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. In *International Conference on VLDB*, 26–37, 1998.