James Bach

COT5405

Advanced Algorithms Project High Dimensional Inversion Distance

The goal of this project was to develop a system of being able to query, given a multidimensional (32-D) string, finding the same or closest inversion relationship among a large data set, specifically an upper limit of 1,000,000 entries consisting of each entry being the same 32-dimensional space. The assignment consists of two phases (and executables): preprocessing and query. The main constraints were the query must be in O(log(n)) time and that the running executable for query should not be loading up the entire preprocessed dataset and not exceeding 32MB. Many possible solutions without the constraint of O(log(n)) time can be made but with the high dimensions and large data set, most operations from insert or find are brutally slow because they end up being O(n), O(n log(n)), or even O(n2) such as the brute force example in the given resources.

My original thought process was assuming this would not work with Euclidean type algorithms, especially in higher dimensions. I considered other methods such as sorting the algorithms in some way to make any sense based on a specific dimension, leading to nothing fruitful. The main metric we should work with is primarily the inversion distance, which is most definitely not a cheap operation in terms of large datasets. I explored, researched, and discussed other options such as R trees, B trees, M trees, KD trees, Ball trees, Fenwick Trees, approximate nearest neighbor, and some clustering approaches. I have experimented with a few of these concepts in Python and performed some time checks on their feasibility for this project. Most of them seemed rather hard to optimize as well towards this problem set. My own personal inclination, to achieve this project’s goal, was to apply a complex data structure of making a modified K-D type tree to support a Ɵ (log(n)) query or search. I further explored the data structure known as an M-tree, not to be confused with m-ary tree, which is used to index metric datasets. The M-tree uses a metric, such as inversion distance in this case, and determines a radius coverage of subsequent M-trees.

Once I decided on a potential, promising data structure to be able to query from, I decided on the language. Although the Python requirements were more relaxed (64mb RAM) I wanted to have all the control I could get and a (hopefully) much speedier approach, so I chose to do a C++ implementation. I used mostly C syntax and several C libraries to achieve the M-Tree implementation. The M-tree has similar operations as most trees which I found pseudocode/algorithms for such as Add, Split, and Query. The short version of the general algorithm for the tree was basically add nodes (Ɵ (log(n)) to the tree, when it became full, then perform a split Ɵ (log(n)), creating a new tree, and there are some special cases based on if the tree is a leaf, non-leaf, or root. The tree performs a recursive search, Ɵ (log(n)), for the minimum inversion distances or matching minimum distance nodes using decision making metrics such as radius cover to determine the best paths for traversal in the query. The worst case in this is O(n) time because this would mean that every node had equal inversion distance because the nodes in the tree are not required to be unique. For any further details see the documentation with the source code. Additionally, the tree itself is O(n).

The M-tree is designed such that a tree acts as a container to 32 nodes maximum per tree and each node can act as a direct connection to another tree. A node itself, on top of being a connection route to another tree carries a row of the 32-D data from the input file. This node connection is important and carries special attributes for radius coverage to determine if a query should traverse that path. Eventually these non-leaf trees will connect to the outer most leaf trees where the path can end.

As previously mentioned, each tree consists of a maximum of 32 nodes. The decision to use 32 nodes was due to optimizing this problem set and was determined by multiple tests on a variety of datasets and extended profiling that this was the optimal time, preprocessing space, RAM usage, and query speed. I think it is just coincidence that this is a 32-dimensional data set. Additionally, through profiling this was determined to have about 22 nodes per tree on average. The selection of this size also may rely more on other aspects as well such as the type of data set, caching, etc.

The design and modularity of this M-Tree structure is leaps and bounds more time and space efficient than any brute forcing method. The RAM constraint of 32MB has been addressed in both the preprocess and extraction process. In the preprocessing phase of the project, the input file is read, the data is tokenized by row to create the previously mentioned nodes and used sequentially to build the tree. Before serialization is completed, the trees and nodes are all assigned a fixed identifier. The trees are placed in a guaranteed size struct that is placed in a serialization file in byte format to reduce storage size on disk. The 1 million size preprocessing file is 10MB smaller than the actual input size. Additionally, this method allows addressable tree extraction from the preprocessing file called “data.mtree”. The query portion allows the program to essentially find tree[i] where i is the identifier of the tree. Each connection node knows where to go to find the next tree. Due to this aspect, only a few trees are loaded in RAM given time. Each tree is insignificantly small individually. The total RAM usage profiled for the executable was generally no more than 15.5MB. In a simple program with the included libraries only and a print Hello World statement, the RAM usage was about 15MB, thus making the net usage around 500KB in the upper limit. The contents of the serialization file were originally written out for human readability and was used for debugging and validating correctness. In later revisions, due to necessary optimizations against slowness and size, it was refactored to be machine ready.

The tree building process is also substantially faster than brute force methods because of the Ɵ (log(n)) insertions. The add node operation finds the best location for the node by recursively traveling the tree to find the best fit. In this process, upon reaching capacity of a given tree, the split operation will be invoked, and this effectively will end with a new parent with two children based on their split. I found that with no previous knowledge of the data set, such as sortedness, choosing which nodes to split on would be best if chosen at random, much like quick sort can choose a random pivot. A couple other powerful aspects of the M-tree are using the radius cover of each tree to determine the farthest inversion point from each tree and tracking the inversion distance to the parent, which enables finding the best-case inversions, or the match.

In developing this project for a C++ implementation and performing many variations of sections and validating faster approaches through profiling, I found that most of the application’s time was spent in the calculation of inversion distances. Therefore, I spent a little further time researching potential optimizations for this operation for this problem set because it could drastically improve the overall time. Through previous researches mentioned such as the Fenwick Trees, I came across an O(nlog(n)) approach for inversion distance that was different from the merge sort version and was a great fit for C++ optimization. This inversion calculation algorithm used a Binary Indexed Tree (or BIT) method to calculate the inversion distance. This method uses a significantly faster method of calculating inversions geared towards bit operations and simple calculations which have a huge performance boost in C++.