CAM^2 Report

Database Report on the efficacy of LSH vs kd Trees for building a database which supports efficient searching for k-Nearest Neighbors

Amol Moses Jha

July 5, 2018

To aid in our team's goal for solving the re-identification problem, we needed to come up with efficient data structures for storing feature vectors extracted from each frame of a video.

Our requirements dictated that in addition to being fast and highly optimized for inserting and retrieval of objects, the data structure in question should also help us to find the k nearest neighbors of an object in question. This would greatly reduce the overhead of comparing the current object against multiple unrelated entities, and enable faster performance.

We narrowed our search down to two efficient data structures, and this report rationalized our choice for the final decision.

kd Trees

kd Trees are k-dimensional trees, with every level of a tree representing partitioning along a certain dimension. Each level of a kd tree splits all children along a specific dimension, using a hyperplane that is perpendicular to the corresponding axis. This makes this a very fast to access data structure, as traversal through the tree is the same as traversal through a binary tree. This makes searches (and kNN searches by the same extension) have a time complexity of $\mathbf{O}(\boldsymbol{log\ n})$.

While relatively easy to understand, kd Trees have a major flaw. They are meant for static data, which is, a predefined numbers of points from which to build the tree from. This means that insertion of new points ends just adding points to the lowest level as a new leaf. This results in the resulting tree to be unbalanced. An unbalanced tree results in widely inaccurate results for kNN queries. Balancing a kd tree, unfortunately is an active research area, and the only way to do so now would be to simply append the new point to the already existing list of points and to rebuild the tree from scratch. This results in the kd tree having abysmal performance for insertion and deletion with a theoretical lower bound of $\mathbf{O}(\boldsymbol{n^2 log\ n})$. This could possibly changed with some heuristics on when to rebalance the tree, such as rebalancing only after n insertions and/or deletions, but that ends up being very detrimental to applications requiring a high degree of accuracy for kNN queries, such as ours.

LSH (Locality-Sensitive Hashing)

Locality sensitive hashing is a very effective method to reduce the dimensionality of multi-dimensional data such as we are dealing with. While regular hashing algorithms aim to hash input points to different values, LSH on the other hand aims to hash all similar values to the same hash. This results in a theoretical lower bound of $\mathbf{O(\mathit{1})}$ for any kNN search and randomized insertions. Deletions and searches for a particular node have a theoretical lower limit of $\mathbf{O(\boldsymbol{n})}$, resulting from a case when all points hash to the same "bucket"; however this is extremely rare in practice, as most well known LSH algorithms do a great job in separating differentiable data.

Metrics

The following are the timings for insertions and deletions on n points conducted on LSH and kd tree data structures, for a typical use case of our required. The time taken by both (within a reasonable degree of error) are formulated below:

n	**kd Tree (s)**	**LSH (s)**
50	0.020974	0.044184
100	0.089230	0.085083
200	0.472756	0.247437
500	3.518683	0.603336
1000	19.387651	1.491773
5000	300.432561	8.173680

The table above shows typical growth of algorithms between a typical $\mathbf{O(\boldsymbol{n^2 log\ n})}$ and an $\mathbf{O(\boldsymbol{n})}$ algorithm.

Therefore, as we can see LSH in addition to being faster offers higher flexibility in its ability to insert and delete points at will. These highly desirable features lead us to choose it for our caching/database implementation for our project.