Hashing and Trees for Fast Approximate Nearest Neighbor Search

Project Proposal

Fan Cao
Lingli Mei
Wenpeng Wang
Xin Kan
Zhenjie Cao

Introduction

When given the object and a database, the most common and original way to quickly retrieve the most similar object to the query from the database would require one to linearly scan through the entire database, which is typically too costly for online applications. Implementing a more efficient way that requires less time and space is very useful and necessary nowadays. In this project, we are going to implement two approaches to solve this problem. One is locality-sensitive hashing and the other one is k-d trees. Both are data structures for finding an approximate nearest neighbor to a query. We will also compare the performance of these approaches in terms of speed and accuracy. If possible, Firstly, we will explore extensions beyond basic LSH and k-d trees, and find an extension that yields better performance. Secondly, we will develop an interactive application that allows one to search quickly through a large database of files. Lastly, we'd like to do some improvement to one of these algorithms.

Background

1. Locality-Sensitive Hashing:

The locality sensitive hashing is a promising method to resolve the approximate nearest neighbor search. While for normal hashing, the collide that is different keys have the same value pointing to the same "bucket", the locality sensitive hashing utilize this collision. Two points that is close have higher chance to be place to the same bucket, their key have higher chance to collide, when the two points that are far away from each other tend to end up in different bucket. That is when we want to query something, we hashing it, and the value it corresponds will find you a bucket fill of points that have high chance to be a close neighbor of the query point. Here is an simplicity of this method:

1. if $d(O1,O2)<r1$, then $Pr[h(O1)=h(O2)]\geq p1$

2. if $d(O1,O2)>r2$, then $Pr[h(O1)=h(O2)]\leq p2$

2. K-d Tree:

a k-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. k-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches). k-d trees are a special case of binary space partitioning trees.

Procedures

1. locality-sensitive hashing:

Consider the data set of the MIT image set is S and the distance function is D. We will consider an LSH family F. And the algorithm will have two main parameters: the width parameter k and the number of hash tables L.

1. In the first step, define a new family G of hash functions g, where each function g is obtained by concatenating k functions $h_1,\ldots,h_k$ from F. The algorithm then constructs L hash tables, each corresponding to a different randomly chosen hash function g.
2. In the preprocessing step we hash all n points from the data set S into each of the L hash tables. Given that the resulting hash tables have only n non-zero entries, one can reduce the amount of memory used per each hash table to O(n) using standard hash functions. Given a query point q from S, the algorithm iterates over the L hash functions g. For each g considered, it retrieves the data points that are hashed into the same bucket as q. The process is stopped as soon as a point within distance cR from q is found.

2. k-d tree:

K-d tree is used to store spatial data, and it is used to approximate nearest neighbor search queries. This is steps to implement this data structure and test its ability:

1. An algorithm to construct k-d tree using the test database should be implemented, and check for correctness based on its features(complexity, node fields, storage, codebook, etc.).
2. Study and implement other k-d tree operations: add element, remove element, balancing, range search.
3. Design and implement k-d tree Nearest Neighbor Search operation, which in k-d tree is by finding the nearest point of a given point.
4. Test k-d tree nearest neighbor search performance (speed and accuracy) under different conditions to find it's best applicable cases.
5. Study k-d tree extensions for better nearest search performance.
6. Implement an extended k-d tree and evaluate its performance if possible.

Problems we might have

1. K-d Tree:

k-d tree construction method can lead to very unbalanced trees, and resulted in slow speed of processing. This can be avoided by stick to typical k-d tree construction: portioning points sets; instead of insert points and then divided into appropriate cells.

For a image database, it is time consuming to compare each pixels of each image for a NNS query. The solution is to define a classifier to process the image first, and then apply the NNS algorithm to search within the results of classification.

2. locality-sensitive hashing:

In order to reach a high performance and high efficiency, LSH method is using large scale memory space to build it algorithm, besides, the frame of this LSH algorithm itself have some limitations, which need a lot of hash table to ensure its efficiency, and is not that efficient when searching points that are far from the search point. A new algorithm named FBLSH has been designed to solve all this problems, we may have time to compare this with the original LSH algorithm after we finish the original task.

Division of Work

At phase 1, 2 student will focus on the k-d tree, and 2 student focus on the Locality-sensitive hashing. 1 student will need to find data set, write codes to access the dataset and process them, take the gist feature out or other ways to reduce the dimension of pictures, and find ways to provide the correct solution, the nearest neighbor for different point, which will be needed when we try to test out the performance of two algorithms.

At phase 2, we will try to find an improvement for the algorithms, or to see if there are better solution. Possible choice includes AND, OR construction of LHS, using multiple LHS table at same time. Also we can come up with an interactive application for our algorithms. A Gui, command line interface or a web application, that can find you a similar picture among certain data set. 3 can focus on the improvement, 2 can work on the application.

Time-line

Phase	Enduring Time
1	March. 15—April. 3
2	April. 4—April. 30