NEAREST NEIGHBOR SEARCH WITH K-DIMENSION TREE APPROACH

MY PROJECT DSA(CO2003)

NGUYEN DAC HOANG PHU
Ho Chi Minh City University of Technology
phu.nguyen7122002hp@hcmut.edu.vn

ABSTRACT

The problem of finding similarities is a well-known challenge in computer science. k Nearest Neighbor is a method to solve that problem, it has achieved certain success in terms of accuracy. However, the main disadvantage of this method is still there as it requires too much data in a single computation (which is not efficient as the number of data points is increasing) leading to a very long time consuming. K-dimensional tree and Locality Sensitive Hashing are studied to store data in a reasonable way, creating a favorable premise in speeding up the search for nearest points. The kD-tree data structure and its performance in storing or searching multi-dimensional data. Besides, we also learn how K-dimension Tree (kDT) applies the methods of finding neighboring data points and compare its effectiveness with the Locality Sensitive Hashing (LSH) method in the K-nearest neighbor problem. Finally, we extract a conclusion as to which method is more efficient and used in all cases where the number of data points is different.

KEYWORDS

kD-tree, Locality sensitive hashing (LSH), k-Nearest Neighbor (kNN)

1 INTRODUCTION

k-Nearest Neighbor (kNN) is an algorithm chosen into the lazy-machine learning category. Its special feature is that it does not learn anything from the training data, instead all calculations are performed only when it is necessary to predict the outcome of a new data point based on the labels of k data points. closest to it. kNN has many real-life applications, most often in classification problems, label prediction, or even further, in problems of retrieving related information, or finding behavior. anomalies of a system,... Although there are many advantages in terms of training or data, this algorithm still faces many limitations such as being very noisy when k is small, or performing computations for too long when k is small. Having to store all the data at the time of computation.

Two efficient storage data structures for finding a multidimensional data point: the kDTree (which divides the data into neighboring regions) and the local hash table (which hashes the data into bins).). For kDTree, we also understand how to make the tree balanced (benefit in searching) and the heuristics in choosing how to partition the data in kDTree accordingly. Remaining with hash, we go into two main methods, flipped bits and random universe plane to improve the accuracy of the search more.

Supervised by TRAN NGOC BAO DUY and NGUYEN DUC DUNG.

Project in DSA, (CO2003), HCMUT
2022.

Related Work.

- We will first define how to generate data and how to preprocessing it to fit each data structure.
- Diving into the kD Tree, we will define the basic tree data structure as well as its basic elements and methods: insert, deletion, clear, search, .. There is also a way to process the data to help the tree balance.
- Similar to hash, we also define a hash method, a method to find neighboring bins, redefine the kNN method, etc.
- Next, we will compare the storage space, search speed, and computation speed of kD-Tree, LSH and kNN pairs as well as overall to find out which method is the most efficient. .
- Finally, find other ways to improve performance and find out if there are unknowns affecting the whole structure,...

Contribution Summary. Here are some useful tool and convenient way which i uses for my project:

- We implement it all with colab's ipython notbook <-> convenient for storing right on the drive as well as being able to speed up computation with GPU,...
- We have implemented 2 ways of creating kD-Tree with tree with split value and tree without split value.
- We develop data structures according to OOP <-> programming technique, which is convenient for abstracting functions as well as improving the readability and understanding of the model.
- Going further we can continue to study Ball-Tree, QuadTree as well as more complex hash locality methods to better serve to improve search performance in kNN problem.

2 BACKGROUND

2.1 kD-Tree

Definition: kD-Tree (also called as K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning(details below) data structure for organizing points in a K-Dimensional space. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points

with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x value of the point, and its normal would be the unit x axis [1].

Construct kD-Tree: kD tree is a non-dynamic model (difficult to remove elements) because all data must be made available (not as a data stream problem). Since there are many ways to construct data separation planes, there are many ways to construct kD-trees. The two main methods are: data is stored in both inner and leaf nodes or data is stored only in leaf nodes.

Method 1: We insert into the kD-tree the data points sequentially according to the above rule. When going further down the tree, we get the depth d, want to know which coordinates are chosen to create the data separation plane, we do modulo d and k (k is the number of dimensions of the data point) will be the result is that a or the a coordinate will be selected. However, the disadvantage of this method is that the complete tree can be unbalanced and easily degenerate into a linked list if all the data revolves around a region which is detrimental to the element search. The most effective way to overcome this drawback is that for each dimension we choose the median of the data as the intersection point, continue to recursively choose the median point with the left and right parts so we will get a balance tree. To do that logically, we should sort the data first using the heap sort or quicksort algorithm so that the complexity of building the tree is O(nlogn) (where n is the number of data points). Format tree with example: a 3-d tree, there will be a root node containing the plane selected on the x-axis, with 2 left and right children selected on the y-axis, with all grandchildren being selected along the z-axis, otherwise x-axis and so on,... The implementation will be in the following section.

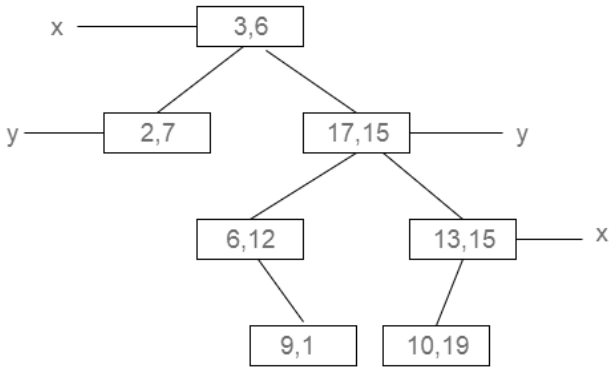

Figure 1: 2d-tree after inserting the sequence of elements: (3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19) [2]

Method 2: Instead of each inner node containing data in it, we only store there the value of split dim (selected direction), split value (value selected to split according to selected coordinates) and a Some other attributes will be defined later. This kDtree with each node will represent a bounding box and the recursion will create smaller bounding box regions so that similar datasets will stay together. A special feature is that a leaf node instead of only containing a single data point, with this method we can create a list of data in a leaf node. To help decide a good split value, we will be interested in heuristics such as: the chosen dimension can be the dimension with the largest variance (ie easy to split the data into 2 separate parts), split -value can be the median of the data in the selected dimension or the midpoint of the data range (regardless of whether this value is in the data or not), tree construction stops until the bounding box is reached. the required number of points or when the split results in too little data to split. The implementation of this method will bring the tree to near equilibrium and the construction time with O(nlogn) complexity.

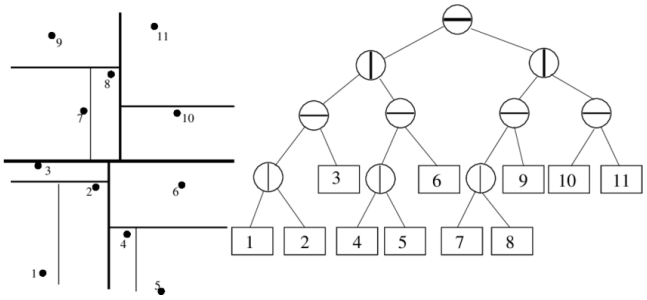

Figure 2: kD-tree with split value

Operation: I write ### in the cells below because the last 2 structures are used for all the provided data, the insert is calculated at construction time, and if insert and remove separately, it will lead to the data-stream problem. to unbalance the tree, and make no sense for method 2.

Method	1-not balance	1-balance	2-nearly balance
Construction	O(nlogn)	O(nlogn)	O(nlogn)
Insert	O(logn) - O(n)	###	###
Search	O(logn) - O(n)	O(logn)	O(logn)
Remove	O(logn) - O(n)	###	###

Nearest Neighbor Search with kDTree: To find k nearest neighbors, we will initialize a queue, or stack of size k to store the k closest elements (by euclidean, or hamming distance) to the point to be queried. With method 1, each node has a chance to compare the distance with the query node, and with method 2, only the leaf nodes have a chance to compare.

- **Step1:** Starting with the root node, the algorithm moves down the tree recursively, in the same way that it would if the search point were being inserted (i.e. it goes left or right depending on whether the point is lesser than or greater than the current node in the split dimension).
- **Step2:** Once the algorithm reaches a node, it checks that node point and if the distance is better, that node point is saved as the "current best".
- **Step3:** To know the next move direction, we will rely on the split-dimension value, we will compare the coordinates according to this value with that split-value value, if the side gives the smaller value, we will choose the direction to

go. there. That good direction is called the good-side, the opposite is the bad-side.
- **Step4:** However, the bad-side direction still has a chance to be moved if the absolute difference between the query coordinates in terms of split-dimension and split value <= best_distance.
- **Step5:** Repeat this until we have reached all the leaf nodes, then the algorithm will end

2.2 Locality Sensitive Hashing (LSH)

Hash table is a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE with the average time to search for an element O(1) but requires much less storage than the universe U of all possible keys. Hash table consists of n buckets (n is the size of the table), the data will be stored in the bucket. Hash function as a function that takes data of arbitrary sizes and maps it to a fixed value. The values returned are known as hash values or even hashes. When hashing you sometimes want similar words or similar numbers to be hashed to the same bucket. To do this, you will use "locality sensitive hashing." Locality is another word for "location". So locality sensitive hashing is a hashing method that cares very deeply about assigning items based on where they're located in vector space. Locality sensitive hashing is a technique that allows you to hash similar inputs into the same buckets with high probability. In this method each data point is treated as a vector. For each vector, we need to get a unique number associated to that vector in order to assign it to a "hash bucket". (the entire itemize section below is referenced from [3])

- **Hyperlanes in vector spaces:** In 3-dimensional vector space, the hyperplane is a regular plane. In 2 dimensional vector space, the hyperplane is a line. A hyperplane is uniquely defined by its normal vector. Normal vector n of the plane w is the vector to which all vectors in the plane w are orthogonal (perpendicular in 3 dimensional case).
- **Split the vector space:** All vectors whose dot product with a plane's normal vector is positive are on one side of the plane. Conversely, if the dot product is negative, then the vector lies on the other side of the plane .
- **Encoding hash buckets:** For a vector, we can take its dot product with all the planes, then encode this information to assign the vector to a single hash bucket. When the vector is pointing to the opposite side of the hyperplane than normal, encode it by 0. Otherwise, if the vector is on the same side as the normal vector, encode it by 1. If you calculate the dot product with each plane in the same order for every vector, you've encoded each vector's unique hash ID as a binary number, like [0, 1, 1, ... 0]. At the end, we convert this sequence of binary values to dec values to get the hash value

Search problem with LSH:. First, It is almost impossible to find planes that properly divide similar regions, so we must choose planes with the principle of completely random randomness, leading to the following problem: if two data points have a high degree of similarity, when random exists a plane is located between those two data points, accidentally causing them to be located at random. The 2 bins are completely different, leading to inefficient search, when the points with the highest similarity may not be together in the same buckets. Second, for data points that are not uniformly distributed (completely skewed towards a positive or negative part) can result in a bin containing a lot of data, while a bin containing no points at all, so still searching over large set for each NN query.

K-Nearest Neighbor with LSH:. We can overcome the above disadvantages in a few ways as follows: for the uneven distribution of points, we can move the coordinate axis, translate the vector to another coordinate to make the data uniform. more (for example, if all 2D vectors lie completely to one side, we can move the center O of the x, y axis to the center center of that region,...) As for the disadvantage of random planes, we have two directions The main law is as follows:

- **Flip Method:** Due to random planes, the similarities may be in the wrong bin position, but this difference can only be 1 bit or at most 2 bits, so to improve the search results, when the hash is value , we search in bucket[hash_value], then find other hash-values that differ from this hash-value by exactly 1 bit or at most 2 bits and then access the search in that bucket[hash_value']. This can be done by inverting the bits in that hash_value. => Cost of binning points is lower, but likely need to search more bins per query.
- **Using multiple tables for even greater efficiency in NN search:** Instead of just using one hash_table, we will build multiple hash tables. For the query point, in each hash table we will get a hash_value, we collect the points to be searched and then put into the kNN model to find the k closest points. Cost of binning points is higher, but likely need to search fewer bins per query.

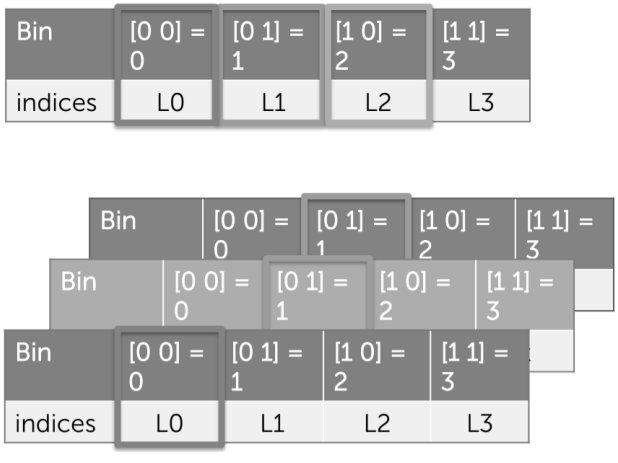

Figure 3: 2 method with LSH [4]

3 IMPLEMENTATION

3.1 Data Preprocessing

Data: Data consists of data points in 3D space, the range of values in the range [0...100] in each dimension, the accuracy of the points is 0.001.

Preprocessing:

- **For the kD-tree**, we will implement it in two different ways. The first way is to use vector to represent each data point (method 2), the second way is to use struct Point float x, y, z;; elucidated. (method 1)

```
if method == "classic" or method is "re_balance":
        point <- Point(x,y,z)
else:
        point <- [x,y,z]
```

- **For LSH**, we also process data points represented by vector form to facilitate dot product calculation. Besides, because the data points all have positive coordinates, the distribution is skewed to one side, so using multiple planes can cause buckets that do not contain any points. So we move the axis so that the new center O' has coordinates (50,50,50) so that the points will be more evenly distributed in this new coordinate system.

```
point <- [x - 50,y - 50,z - 50]
points.append(point)
```

3.2 kD-Tree Construction

Method1 - Not Balance kD-Tree: We insert sequentially according to the pseudo code below

```
function insert(root,data,depth):
    if root is None:
       return Node(data)
    axis <- depth % dim(data)
    root_cor <- root.data.axis
    data_cor <- data.axis

    if root_cor > data_cor:
       root.left <- insert(root.left,data,level + 1)
    else:
       root.right <- insert(root.right,data,level + 1)
    return root
endFunction

function buildkDTree(model,all_data):
    for data in all_data:
        model.insert(data)
endFunction
```

Method1 - Balance kD-Tree: We will first rearrange the data by the median of the coordinates for the selected split-dimension. Recursively down the same tree.

```
function re_balance(data,start,end,depth)
    if start >= end
       return None
    axis <- depth % dim(data)
    sub_data <- data[start->end]
    sub_data <- sort_follow_axis(sub_data,axis = axis)

    mid <- (begin + end)/2
    re_balance(data,start,mid,depth + 1)
    re_balance(data,mid,end,depth + 1)
endFunction
```

When building the tree, we also use the insert function functionined as above, but the order of points will be different according to the sorted order as above (select the median value of each region)

```
function buildkDTree_pre(model,points,begin,end)
    if begin >= end:
       return None
    mid <- (begin + end)/2
    model.insert(points[mid])
    model.buildkDTree_pre(points,begin,mid)
    model.buildkDTree_pre(points,mid + 1,end)
endFunction
```

Method2 - Nearly Balance kD-Tree: If a bounding-box reaches the amount of data <= data limit -> that is the leaf node of the kD-Tree, now we will save the data. Otherwise, we only save the split-value values (find the median according to the coordinates with max-variance) and split dimension (save the full coordinate dimension).

```
function buildkDTree(model,all_data)
    if len(all_data) is 0:
       return None
    if len(all_data) <= model.min_split:
        return Node(all_data,0,all_data[0],None,None)

    split_value <- model.get_variance_dimension(all_data)
    left_data,median,right_data <-
         model.split(all_data,split_value)
    left_sub <- model.buildkDTree(left_data)
    right_sub <- model.buildkDTree(right_data)
    node <- Node([],split_value,median,left_sub,right_sub)
    if node.left is None:
       node.left.parent <- node
    if node.right is None:
       node.right.parent <- node
    return node
endFunction
```

3.3 KNearestSearch with kD-Tree:

Both methods use the same method as mentioned above, the only difference is that method 1, all nodes can have a chance to be considered with target-node while method 2 only has leaf node. opportunity to participate.

```
function
     findNN(root,query_point,best_node,best_distance,depth)
    if not root
       return best_distance
    dist <- euclide_distance(root.data,query_point)

    if dist < best_distance
       best_node.append(root)
       best_distance <- dist
```

```
    axis = depth % dim(query_point)
    root_cor <- root.data.axis
    data_cor <- data.axis

    if data_cor < root_cor
       good_side <- root.leftChild
       bad_side <- root.rightChild
    else
       good_side <- root.rightChild
       bad_side <- root.leftChild

    best_distance <- findNN(good_side,query_point,best_node
    ,best_distance,depth + 1)
    if abs(root_cor - data_cor) < best_distance
       best_distance <-
            findNN(bad_side,query_point,best_node,
       best_distance,depth + 1)
    return best_distance
endFunction

function findkNN(model,query_point,k):
     initialize stack
     findNN(model.root,query_point,stack,infinity,0)
     for i in range(k)
        if size(stack) >= 1
           point = stack.pop()
           print(point)
endFunction
```

3.4 LSH Construction:

Construct: First, we will randomize planes, which can generate a list of planes using the universe method or just one plane for the flip-method. Then for each plane, we will create 2 directories, one is to store the vector, the other is to store the index of the vector.

```
planes_l <- [random_with_size(N_DIMS,N_PLANES) for _ in
     range(N_UNIVERSE)]

function create_hash_table(model,vecs,planes)
    num_of_planes <- number(planes)
    num_of_buckets <- 2^num_of_planes

    #dictionary hash_table store value
    hash_table = {i : [] for i in range(num_of_buckets)}

    #dictionary id_table store id of value
    id_table = {i : [] for i in range(num_of_buckets)}

    for i,v in enumerate(vecs)

        h = model.hash_v(v,planes)

        #store v in "h" buckets
        hash_table[h].append(v)

        #store id of v in "h" id_buckets
        id_table[h].append(i)

    return hash_table,id_table
endFunction
```

Hash function: To compute the hash function for a given vector_query, we will compute its dot product for each plane and then insert the sign value into a list. When we have gone through all the planes, we will convert that binary list into a single decimal hash value.

```
function hash_v(model,v,planes)
    dot_product <- np.dot(v,planes) # (1,3) * (3,4)
    sign_of_dot_product <- np.sign(dot_product)
    h <- (sign_of_dot_product >= 0)

    hash_value <- 0
    n_planes <- number(planes)

    for i in range(n_planes)
        hash_value += pow(2,i) * h[i]
    return hash_value
endFunction
```

3.5 Method Search for kNN with LSH:

Flip Method: We will use this method to invert 1 or 2 bit values in the hash_value, so we can create other hash_values' to increase the accuracy of the search (because in addition to looking in the bucket[hash_value] we also find spread around the bucket[hash_value'])

```
function flipped_one_bits(hash)
    #convert decimal value to binary value
    bits <- binary(hash)
    #drop part 0b in bin value
    initialize list hashs
    for i in range(len(bits))
        bits_temp <- list(bits)
        #invert bit
        if bits[i] is '0'
          bits_temp[i] <- '1'
        else
          bits_temp[i] <- '0'
        hashs.append(decimal(bits_temp))
    return hashs
endFunction

function flipped_two_bits(hash):
    #convert decimal value to binary value
    bits <- binary(hash)
    #drop part 0b in bin value
    initialize list hashs
    for i in range(len(bits))
        for j in range(i + 1,len(bits))
            bits_temp <- list(bits)

            #invert bit
            bits_temp[i] <- flip_bit(bits[i])
            bits_temp[j] <- flip_bit(bits[j])

            hashs.append(decimal(bits_temp))
    return hashs
```

```
endFunction
```

Universe: We will generate more than 1 hash_table, so that the random lines can be located in more locations, increase the diversity in the search, as well as suggest more nearby points.

```
function create_hash_tables(model)
    initialize list hash_tables
    initialize list id_tables

    for id in range(N_UNIVERSE)
        planes <- model.planes_l[id]
        hash_table,id_table <-
             model.create_hash_table(model.data,planes)
        hash_tables.append(hash_table)
        id_tables.append(id_table)

    model.hash_tables <- hash_tables
    model.id_tables <- id_tables
endFunction
```

3.6 K Nearest Neighbor in LSH:

kNN function: We will refunctionine the kNN method, but instead of using the entire data set to search, we will use the two methods above to extract part of the search data. There are 2 main methods to measure similarity: 1 is using euclidean distance, 2 is cosine_similarity

```
function kNN(vector, vector_neighbor_l, k<-1)
    initialize list similarity_l

    for element in vector_neighbor_l
        cos_similarity <- cosine_similarity(vector,element)
        similarity_l.append(cos_similarity)

    sorted_idxs <- sorted_id(similarity_l)
    k_idx <- sorted_idxs[-k:] #last_k_element

    return k_idx
endFunction
```

We will create 2 lists - 1 is a list to store the vectors, 2 is a list to store the indexes of that vector, besides, for convenience of checking whether that vector exists in the list or not, we can create another set to store the unique index values in the list.

```
initialize list points_neighbor_l
initialize list idxs_neighbor_l
initialize set idxs_neighbor_set
```

Extract subset data: We'll cycle through all hash_tables with universe, and with flip_method we'll cycle through hash_values searched in a 1- or 2-bit inversion, for each point appearing in the buckets: we'll check if it exists in the set, go to the next point, otherwise we will add that point.

```
for uni_id in range(N_UNIVERSE)
    planes <- model.planes_l[uni_id]

    #get hash_value of point_query to find neareast
         neighbor of this
    hash_value <- model.hash_v(point_query,planes)

    hash_table <- model.hash_tables[uni_id]
    id_table <- model.id_tables[uni_id]

    #get subset of hash_table to lookup
    new_ids_table_consider <- id_table[hash_value]
    new_points_table_consider <- hash_table[hash_value]

    #if id_query exist in hash_table
    if id_query != -1 and id_query in new_ids_table_consider
      new_ids_table_consider.remove(id_query)

    for i,new_id in enumerate(new_ids_table_consider)
        if new_id not in idxs_neighbor_set:
          point_i <- new_points_table_consider[i]

          #add list to consider
          points_neighbor_l.append(point_i)
          idxs_neighbor_l.append(new_id)

          #add set to faster checking
          idxs_neighbor_set.add(new_id)
```

Find the nearest points: After extracting the subset data, we will put it into the kNN function to find the nearest neighbors

```
points_neighbor_l_arr <- np.array(points_neighbor_l)

k_nearest_neighbor_idx <-
     kNN(point_query,points_neighbor_l_arr,k)
k_nearest_neighbor_idx_l <- [idxs_neighbor_l[idx] for idx
     in k_nearest_neighbor_idx]

k_nearest_neighbor_idx_point <- [model.points[idx] for idx
     in k_nearest_neighbor_idx_l]
```

3.7 Main

kD-Tree: We will have build_method functions with methods here that can have 3 types: classic, re_balance, flip respectively as mentioned above.

```
#build_method
function build_method(filename,method <- "classic")
     points <- readData(filename,method)
     if method is "re_balance"
        preprocess(points,0,0,len(points))
        model <- kDTree()
        model.buildkDTree_pre(points,0,len(points))
     else if method is "classic"
        model <- kDTree()
        model.buildkDTree(points)
     else
        model <- kDTree_split(points,3)
     model.setMethod(method)
```

```
    return model,points
endfunction
```

If we want to test kNN-search, we will create random points and then preprocess it to match the type of kD-Tree to be processed.

```
function gen_query_point(method)
    initialize list l_gen
    for j in range(0,3):
        g is round(random_from(0,100),3)
        l_gen.append(g)
    if method is "classic" or method is "re_balance"
       x <- l_gen[0]
       y <- l_gen[1]
       z <- l_gen[2]
       point <- Point(x,y,z)
    else:
       point <- l_gen #x,y,z
    return point
endFunction
```

LSH:. Both methods can be used when only building a single model, the difference is that each time we use the flip-method we will create a new hash_table that matches the random_plane. As for the Universe, it was already built.

```
model is LSH(points)

model.create_planes(10,25)
model.create_hash_tables()
model.view_result("universe",3,2)
model.view_result("flipped",3,2)
```

Same as kD-tree, to random create a new query_point and then preprocess

```
function gen_random_point(dims)
    initialize list l_gen
    for j in range(0,dims)
        g <- round(random.uniform(0,100),3)
        l_gen.append(g)
    point <- [x - 50.000 for x in l_gen]
    return point
endFunction
```

4 RESULTS

Description: We will measure the time to compare the efficiency of data structures with each other with the main operations: construct, search point in data, search random point, kNN-search with the number of data points respectively. We use k = 3 for kNN-search method anh number-universe is 25. Data is generated from the genData function defined above with the number n shown below. We will use Python's time module to measure time, then summarize the time into a dictionary to create a csv file as you can see below. Log file goes through the process of building the data structure and using the methods located here: log.txt

- **n = 10:** data from 'ten.csv'.

	ds	construct	search	search_rd	kNN-search
0	**kD-nb**	0.002038	0.000411	0.000013	0.000168
1	**kD-b**	0.001205	0.000264	0.000013	0.000183
2	**hash_u**	0.812732	0.037199	0.049695	0.059717
3	**hash_f**	0.812732	0.048004	0.032468	0.084016

- **n = 100:** data from 'one-hundred.csv'

	ds	construct	search	search_rd	kNN-search
0	**kD-nb**	0.00369	0.000231	0.000032	0.000252
1	**kD-b**	0.008505	0.000248	0.000022	0.000184
2	**hash_u**	0.795465	0.028845	0.024303	0.035404
3	**hash_f**	0.795465	0.024345	0.024812	0.113437

- **n = 1000:** data from 'one-thousand.csv'

	ds	construct	search	search_rd	kNN-search
0	**kD-nb**	0.01953	0.000051	0.000045	0.000323
1	**kD-b**	0.018832	0.000059	0.000025	0.000191
2	**hash_u**	0.904691	0.056740	0.040146	0.058719
3	**hash_f**	0.904691	0.055766	0.032251	0.088397

- **n = 10000:** data from 'ten-thousand.csv'

	ds	construct	search	search_rd	kNN-search
0	**kD-nb**	0.1938	0.00026	0.000051	0.00063
1	**kD-b**	0.2014	0.00023	0.000031	0.00043
2	**hash_u**	0.8738	0.04002	0.053602	0.06153
3	**hash_f**	0.8738	0.03684	0.052217	0.12047

- **n = 100000:** data from 'one-hundred-thousand.csv'

	ds	construct	search	search_rd	kNN-search
0	**kD-nb**	2.634	0.00013	0.000059	0.00106
1	**kD-b**	2.762	0.000076	0.000093	0.00091
2	**hash_u**	0.7691	0.044114	0.0558	0.07264
3	**hash_f**	0.7691	0.029161	0.0552	0.08643

- **n = 1000000:** data from 'one-million.csv'

	ds	construct	search	search_rd	kNN-search
0	**kD-nb**	36.7317	0.000104	0.000079	0.000464
1	**kD-b**	54.1090	0.000081	0.000155	0.001374
2	**hash_u**	0.7007	0.033805	0.052698	0.045029
3	**hash_f**	0.7007	0.030671	0.051788	0.088662

where notation: kD-nb is kD-tree not balance, kD-b is kD-tree balance, hash-u is hash-universe, hash-f is hash-flip method.

5 CONCLUSION

We don't use kD-Tree split-value becase while measuring performance: tree building takes quite a long time sometimes up to 20, 30 minutes. This happens because the data is random and splitting the data through parts to go down sometimes takes a long time. Because we have to align the right amount of data for each side. Besides, looking around the remaining data structures, the build time is very fast, showing that this method is not suitable for this problem.

In general, the methods of kD-tree are better in terms of time than LSH, although only slightly better (due to the time being too short and fast). But when the number of data points increases from about 100000 onwards, the time to build the kD-tree is very long and expensive while the LSH is very fast. So customizing when the number of data points is small, we can use kD-tree to give better results, but when the number of data points is large, LSH is an effective method.

When searching with Knn method. kD-tree can give more accurate results, because according to the principle of tree traversing, in LSH we use random_plane, so points with high similarity may be located in bins far apart, so the results are not has high accuracy, I tried to find and compare it with normal Knn, the accuracy of the points found is only 0.2 to 0.3, and when k is small, the accuracy is almost zero (Accuracy at This is the similarity between the two sets of points found by the two methods).

In the future, we plan to continue reading the future reading below to study other data structures that store multidimensional data to see if we can increase the search speed and accuracy with the kNN algorithm.

6 FURTHER READING

Ball-Tree, Quad-Tree - data structures that store multidimensional data.

- **Ball*-tree:** Efficient spatial indexing for constrained nearest-neighbor search in metric spaces. https://arxiv.org/pdf/1511.00628.pdf
- **An effective way to represent quadtrees:** https://www.researchgate.net/publication/220421408_An_effective_way_to_represent_quadtrees
- **Performance Evaluation: Ball-Tree and KD-Tree in the context of MST:** https://arxiv.org/ftp/arxiv/papers/1210/1210.6122.pdf

Nearest Neighbor Search techniques survey: https://arxiv.org/ftp/arxiv/papers/1007/1007.0085.pdf

Hashing:

- **A Survey on Locality Sensitive Hashing Algorithms and their Applications** https://arxiv.org/pdf/2102.08942.pdf
- **Super-Bit Locality-Sensitive Hashing** https://proceedings.neurips.cc/paper/2012/file/072b030ba126b2f4b2374f342be9ed44-Paper.pdf

7 REFERENCES

[1]: **K-d tree:** https://en.wikipedia.org/wiki/K-d_tree

[2]: **Set K Dimensional Tree** https://www.geeksforgeeks.org/k-dimensional-tree/?ref=gcse

[3]: **Natural Language Processing with Classification and Vector Spaces - week 4** https://www.coursera.org/learn/classification-vector-spaces-in-nlp/home/week/4

[4]: **Machine Learning: Clustering & Retrieval - week2** https://www.coursera.org/learn/ml-clustering-and-retrieval/home/week/2

[5]: **Locality-sensitive hashing** https://en.wikipedia.org/wiki/Locality-sensitive_hashing

[6]: **K-d Tree implemented Split-value** https://github.com/HolyChen/KdTree