

CONCORDIA UNIVERSITY – Department of Electrical & Computer  
Engineering Fall 2015

Data Structures and Algorithms (COEN 352)

# **Content Addressable Memory**

Submitted by: Matthew Masi

Student Number: 27039956

Date Submitted: December 20, 2015

## ABSTRACT

Content-addressable memory (CAM) is a type of memory used in high-speed searching applications. It compares input search data against stored data, and returns the address of matching data. In this project, we will be using a form of content-addressable memory to output a greyscale image from an input image. Our approach to this project made use of the NB tree and B+ tree data structures. The NB tree will be used to map multidimensional points to a key by computing their Euclidean norm. The keys can then be inserted into a B+ tree where they can be easily retrieved through a linear search. We will see that every input image combined with a viable Euclidean distance will produce an output image that will greatly resemble the input.

## OBJECTIVE

To design, implement and test an algorithm that is able to output all existing vectors of a given Euclidean distance of a given input vector from a large number of highly-dimensional vectors of integers. The dataset, input, and output vectors to be used represent greyscale images of people where each vector contains 3600 pixels (number of dimensions) each ranging from 0 to 255. The outputted vector is to be summed up and normalized by the recalled images.

## METHOD

### Insertion of dataset:

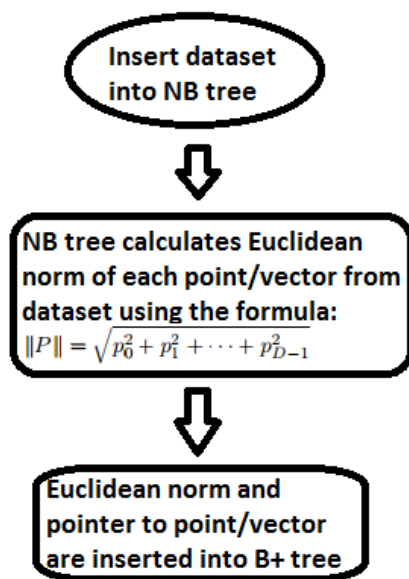


Figure 1: Insertion of dataset into B+ tree

Retrieval of output vector:

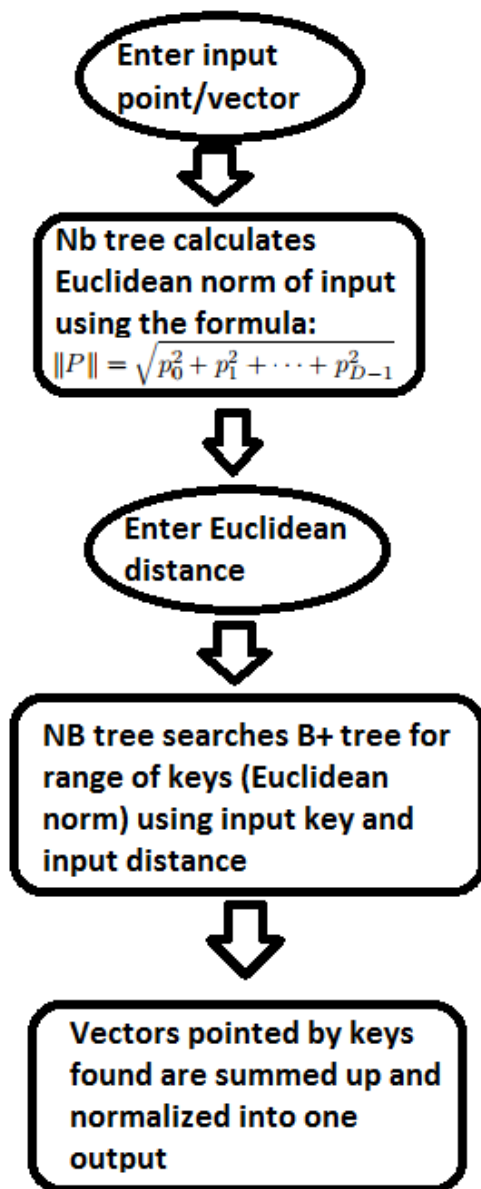


Figure 2: Outputting a vector using NB tree

### **Insertion into B+ tree:**

```
void insert(double k){
    if empty{
        node = root node
        keys[0] = k
        root->numberOfChild = 1
    }

    else not empty{
        if root is full(root->numberOfChild == 2*(degree-1){
            create new node;
            new node->child[0] = root;
            new node->splitchild()
            if(new node->keys[0] < k)
                x++
            new node->child[x]->insertNonFull[k]
            root = new node;
        }

        else if root not full
            root->insertNonFull(k)
    }
}

void insertNonFull(double k){
    int x = numberOfChild-1
    if node = leaf{
        while(x >= 0 && keys[x] > k){
            keys[x+1] = keys[x]
            x--;
        }
        keys[x+1] = k
        numberOfChild = numberOfChild + 1
    }
    else not a leaf{
        while(x >= 0 && keys[x] > k)
            x--;

        child[x+1]->insertNonFull(k)
    }
}
```

The above two functions provide insertion to the B+ tree. The first function is used to make sure that root is not empty, and that it is not full based on the degree of the B+ tree. If it is empty, it assigns the node as the root node. If it is not empty but the number of children is at a maximum (based on the degree of the tree), root is assigned to a newly created node, and the old root is split up to make room for the new child. Finally, when these two conditions are met, (not empty and not full) the key can be normally inserted into the B+ tree.

The second function inserts the key into the B+ tree when the root is not empty, and when the number of children nodes do not surpass the condition set by the degree of the tree. It first checks if the node is a leaf node. If it is, it first finds the location of where the new key will be inserted,

moves all greater keys one place ahead, and inserts the key into its place. If the node is not a leaf node, it then searches for which child is going to have the new key, and then uses a recursion call of the function using the selected child node.

### **Searching the B+ tree:**

```
vector <double> findKeys(double key, double range){
    for(int x = 0; x < numberOfChild; x++){
        if (not a leaf node)
            child[x]->findKeys(key,range)
        }

        else if(a leaf node){
            for (int x =0; keys[x] < keys.size(); x++){
                if(keys[x] > key-(range/2){
                    while(keys[x] < key+(range/2))
                        vec[i++] = keys[x++];
                    break;
                }
            }
        }
    }
}
```

The searching function first checks to see if it has a leaf node. If not, recursion is used to get the child node. If the child node is found to be a leaf node, it searches for the range of values indicated by the input range value. It does this by first finding the lower bound ( $\text{key} - (\text{range}/2)$ ), then sequentially finding the rest of the keys up until the upper bound ( $\text{key} + (\text{range}/2)$ ).

### **Calculating Euclidean norm:**








```
double calculateKey(){
    for i = 0 to 3600{
        key = key + vec[i]^2

    key = sqrt(key)
}
```

The Euclidean norm of each vector was calculated by taking the sum of squares of all integers, and then taking the square root of the sum. The result is then used as a key value representing the vector in the B+ tree.

RESULTS & DISCUSSION

The following are the results of 10 different inputs:

<u>Input</u>	<u>Output</u>
	
	
	
	
	
	
	
	



<u>Image</u>	<u>Key Value</u>	<u>Euclidean distance</u>
<b>1</b>	45.8913	0.5
<b>2</b>	39.0135	0.45
<b>3</b>	5.00629	2
<b>4</b>	37.7016	30
<b>5</b>	34.5021	0.05
<b>6</b>	38.9632	0.5
<b>7</b>	46.4322	0.5
<b>8</b>	28.566	0.1
<b>9</b>	47.7107	1
<b>10</b>	66.5519	1

Table 1: Key values and Euclidean distance

If we compare the Euclidean distance from table 1 with the results above, we can see that an Euclidean range of 0.5 – 2 gave the most accurate results. When the distance was too low, the results were a little less accurate. We can also see from image number 4, that with a really high Euclidean distance, the output was also less accurate.