Australian National University

School of Computer Science

College of Engineering and Computer Science

Nearest Neighbour Search with Zero-Suppressed Decision Diagram for Text Retrieval

Yong Boon, Lim – U5122243

COMP8740 - Artificial Intelligence Project

Supervisor: Dr. Scott Sanner

Abstract

Exact NNS on text document becomes very expensive for its high dimensionality and the volumes. We demonstrate Exact NNS on text can be performed in linear time by using Zero-Suppressed Binary Decision Diagram (ZDD). We also demonstrate ZDD combined with Multi-hash technique can perform Approximate NNS on text with sublinear time. The exact NNS Naïve NN-ZDD model query time is order of magnitude faster than exhaustive NN for text, and approximate NNS using ZDD Multi-hash's query time is closer to LSH with less build time and better accuracy.

Acknowledgements

Thanks Dr. Scott for sharing his view and precious time to honestly guide a rookie.

Thanks Dr. Weifa for sharing his project experience and living tips in Canberra.

Thanks God for everything.

Table of Contents

1 Introduction

Nearest Neighbour Search (NNS) is a search problem which effectively finds the pre-processed objects in a database closest to a query object measured by distance/similarity metrics. In this report, we focus NNS on text domain which exhibits *high dimensional* and *sparse* properties. NNS on text has many applications [1], for example, de-duplication for web crawling, plagiarism detection, personalized news aggregation, related page/document search, and text clustering.

In general, the complexity of NNS increases exponentially with the dimensionality (curse of dimensionality). Exact NNS on text document becomes very expensive for its high dimensionality and the volumes. Most of the Branch and Bound NNS techniques (i.e. set partitioning) perform only as good as linear search on text document.

Approximate NNS techniques attract attentions in high dimensionality NNS. Approximate NNS finds the nearest neighbour with high probability in sublinear time in the cost of allowing false positive. Locality Sensitivity Hashing (LSH) is one of the popular Approximate NNS algorithms. Google [4] used Approximate NNS algorithms Simhash for duplication for web crawling [5] and MinHash and LSH for Google News personalization [6].

In this report, we demonstrate Exact NNS on text can be performed in linear time by using Zero-Suppressed Binary Decision Diagram (ZDD) [2]. ZDD is a compact and efficient data structure of Binary Decision Diagram (BDD) to represent high dimensional sparse set. Each ZDD node represents the feature of the text document, and all ZDD nodes in a valid path consist of all the features of a text document.

Besides, we also demonstrate ZDD combined with Multi-hash technique can perform Approximate NNS on text with sublinear time. Mult-hash technique hashes a document multiple times to cluster more similar one from the less similar document, and perform NN in a smaller set.

This paper main research contribution is,

- For exact NNS, we propose using the ZDD to perform NNS.
- For approximate NNS, we propose a ZDD with Multi-hash to partition the smaller set to improve the performance for both build and query time with higher accuracy.

The rest of the paper is organized as follows. Chapter 2 provides a brief background in various document representations and basic idea of ZDD. Chapter 3 describes the design of various NNS using ZDD (NN-ZDD models). Chapter 4 presents the experimental study. The final chapter offers conclusions and research direction.

2 Background

In this chapter, we provide the background knowledge to lay the foundation for various NN-ZDD models in Chapter 3. We give overview of the NN methods. Then we go through various document term representation and weighting metrics in the vector space models. Last, we present ZDD data structure and its associated set operations.

2.1 Overview

Various NN methods have been proposed to reduce the complexity under 2 major categories [1],

- Branch and Bound Techniques
  Branch and bound techniques construct tree-based structures which partition the set into disjoint subsets. Every node contains the threshold information which determines the subset closer to the query point. Branch and Bound NNS methods include K-D Tree, Vantage Point Tree (VP-Tree), and Generalized Hyperplane Tree (GH-Tree).

- Mapping-based Techniques
  Mapping-based techniques map the set of high dimensions into lower dimensions with high probability of the closer objects staying near to each other. In Locality Sensitivity Hashing (LSH) [10], the objects are hashed into lower dimensions and another hash function maps each object into different buckets. The objects closer to each other have higher probability to fall into the same bucket.

Branch and Bound techniques which normally use for are Exact NNS are not efficient in the high dimensional space. In contrast, mapping-base techniques are efficient in high dimensional space performs only Approximate NNS.

2.2 Problem Setting

The NNS problem can be formalized [1] as, in a metric space M with similarity function σ/distance function d, given a set of input $S = \{p^1, \dots, p^n\} \in M$, and a query $q \in M$. Find $argmax_{p^i}\sigma(p^i, q)/argmin_{p^i}d(p^i, q)$ as illustrated in Figure 2-1.

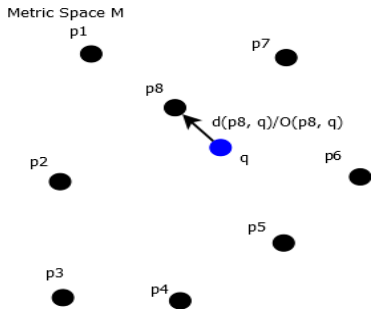

Figure 2-1 Nearest Neighbour Search in 2D

2.3 Metric and Similarity Functions

A distance function d on metric space M satisfies [1],

- Non-negative : $\forall p^i, q \in M,\ d(p^i, q) \geq 0$
- Symmetry : $\forall p^i, q \in M,\ d(p^i, q) = d(q, p^i)$
- Identity : $\forall p^i, q \in M,\ d(p^i, q) = 0 \Leftrightarrow p^i = q$
- Triangle Inequality : $\forall p^i, p^j, q \in M,\ d(p^i, p^j) \leq d(p^i, q) + d(q, p^j)$

Similarity function $\sigma(p^i, q)$satisfies all the properties above except the Triangle Inequality. We discuss various distance/similarity functions in Section 2.4.

2.4 Document Model

In NNS on text, the documents in document set S and the query are represented as vectors, $p^i = (p_1^i, \dots, p_k^i) \in S$ and $q = (q_1, \dots, q_k)$ [8]. Each dimension in a vector represents a document feature. This model is referred as vector space model M. The value represents either existence or weighting of a document feature depending on the schemes

2.4.1 Document Term Features

Each term in a document represents a feature/dimension. The dimension *k* of the vector space model M equals to the number of unique terms in all the documents.

2.4.1.1 Boolean Scheme

In the simplest scheme, the value of each vector j in document p^i and query q are expressed in Boolean as $p_j^i, q_j \in \{0, 1\}$ representing the absent or present of a feature. The distance between document p^i and query q is measured by Hamming Distance.

$$Hamming\ Distance = \sum_{j=1}^{k} \mathrm{I}(p_j^i, q_j) \qquad \text{and } I(x, y) = \begin{cases} 1, if\ x \neq y \\ 0, if\ x = y \end{cases}$$

In Chapter 3, we use the Boolean scheme and Hamming Distance to construct NN-ZDD.

2.4.1.2 Term Frequency Scheme[8]

In term frequency scheme, each dimension of term t is weighted by the term occurrences in document p, and denoted as $tf_{t,p}$. However, rare term is also important to distinguish the documents. Therefore, a measure Document frequency for term t (df_t) represents the total term occurrences in all the documents, and the inverse of df_t give weight to the rare term, and defined as, $idf_t = log\frac{N}{df_t}$ where N is the total number of documents. $tf_{t,p} * idf_t$ reflects the weight of each term by considering both term frequency at the document itself, and overall document set.

2.4.2 Hash Features

One of the effective ways to reduce the dimension of the document is through hashing. Image we assign a unique ID to each of the English alphabet,

A	B	C	D	E	F	G	H	I	J	K	L	M
1	2	3	4	5	6	7	8	9	10	11	12	13

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
14	15	16	17	18	19	20	21	22	23	24	25	26

Table 2-1 ID for English Alphabets

To represent any English word, we need a vector of 26 dimensions, for example, “APPLE”, {A=>1, P=>16, L=>12, E=>5}.

If we apply a hash function 31*x mod 5 to all the English alphabets' ID,

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	0	1	2	3	4	0	1	2

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
3	4	0	1	2	3	4	0	1	2	3	4	1

Table 2-2 Hash ID for English Alphabets with 31*x mod 5.

"APPLE" can now be represented by a vector of 5 dimensions {A, P=>0, L=> 1, E=>4}. So do "ALE", and "PLE" which has the same vector as "APPLE".

If we apply a new hash function 31*x mod 6 to all the English alphabets' ID,

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	0	1	2	3	4	5	0

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	3	4	5	0	1	2	3	4	5	1	2

Table 2-3 Hash ID for English Alphabets with 31*x mod 6.

"APPLE" can now be represented by a vector of 6 dimensions {A=>1, P=>4, L, E=>5}.

By combining both hash vectors (from 31*x mod 5 and 31*x mod 6), we could distinct each word with the smaller vector sizes, and form the clusters of 31*x mod 5 with 3 distinct words surround it.

English Word	31*x mod 5	31*x mod 6	Original
APPLE	(0, 1, 4)	(1, 4, 5)	(1, 5, 12, 16)
ALE	(0, 1, 4)	(1, 5)	(1, 5 12)
PLE	(0, 1, 4)	(4, 5)	(5, 12, 16)

Table 2-4 Combined Hash ID for English Alphabets

The useful property of partition large set of document into small sets for NN, and it motivates the NN-ZDD Multi-hash in Section 3.2.1.

2.5 Zero-Suppressed Decision Diagram

The Zero-suppressed Decision Diagram (ZDD) is a canonical directed acyclic graph (DAG) [0] compactly represents a set of objects S. ZDD represents the sparse set efficiently and makes it a potential candidate for the representation of document. ZDD consists of one root node, a set of decision nodes (circle) and two terminal nodes (box) as shown in Figure 2-2. The terminal node 0 (N^0) represents the empty set 'Ø' and terminal node 1 (N^1) represents '{Ø}'.

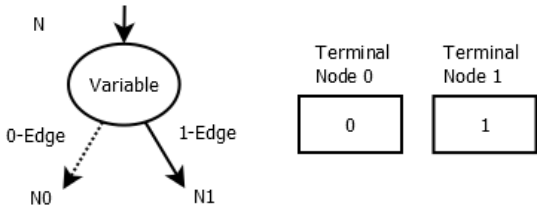

Figure 2-2 Basic Elements of ZDD

An object $p_j \in S$can be represented by an k-bit Boolean vector $p_j = (x_1, \dots, x_k)$ where $x_i \in \{0, 1\}$. Each path from the root node to the terminal node 1 represent an object $p_j \in S$. A variable is present in object p_j if the path go through the 1-edge of the node represents it.

Each decision node N^j for a variable x_i and two subset N_0^j and N_1^j denoted as $N^j = Node(x_i, N_0^j, N_1^j)$ representing the Boolean function $N^j = (x_i \wedge N_0^j) \vee (x_i \wedge N_1^j)$. The decision node N^j contains 1-edge (solid line) and 0-edge (dash line) for variable $x_i \in p_j$ or $x_i \notin p_j$ on the path representing p_j. In Figure 2-3 a), ZDD set S represents 3 object set {{A, C}, {B, C}, {A, B, C}} corresponding to all 3 paths from the root decision node to terminal node 1.

ZBDD has an important property [3], i.e. all the equivalent decision nodes are shared and reusable. If two decision nodes N_i and N_j represent variable x_i with two subsets N_0 and N_1, they are considered equivalent, canonical and shared, i.e. $N^i = N^j = Node(x_i, N_0^j, N_1^j)$ Therefore a vector space of 2^k can be represented by lesser nodes compactly.

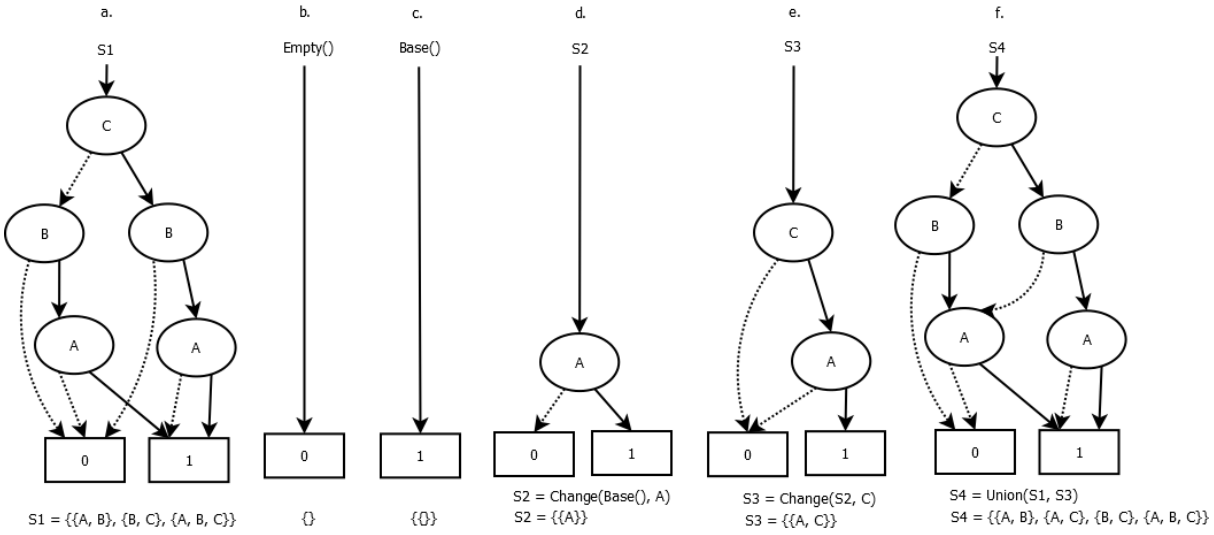

Figure 2-3 Examples of ZDD Basic Operations

The basic operation of the ZDD is shown in Table 2-5. A simple example in Figure 2-3 b) to e) illustrates how the set S3 is constructed using the ZDD basic operations. In Figure 2-3 f), set S4 are the union of set S1 and S3. Note that ZDD for S1 and S4 have the same number of decision nodes even though they represent different sets as some of the decision nodes are reused.

Operation	Description
Empty()	Return empty set Ø
Base()	Return set contains empty set {Ø}
Change(S_j,x_i)	If set S_j does not contain x_i, return the set of subset of S_j with x_i, else return the set of subset of S_j without x_i
Subset0(S_j, x_i)	Return set of subsets of S_j not containing variable x_i
Subset1(S_j, x_i)	Return set of subsets of S_j containing variable x_i
Union(S_1, S_2)	Return set of $S_1 \cup S_2$
Intersec(S_1, S_2)	Return set of $S_1 \cap S_2$
Diff(S_1, S_2)	Return set of $S_1 \setminus S_2$

Table 2-5 ZDD Basic Operations

Variable ordering affects the size of the ZDD of the same sets as demonstrate in Figure 2-4 a) and b). Effective variable ordering for compact ZDD [3] should, 1) groups of variables with high co-occurrence rate should be put together; 2) Variables are ordered by their frequency. It's a hard problem to find out the optimal variables ordering, especially in the text domain where new documents incrementally adding to the ZDD, and influence the relative frequency of variables. One of the heuristics used [3] is to order the variables by their frequency. We show in the experiments (section 4.4.1.2) the variable ordering has slightly influences on the ZDD node sizes.

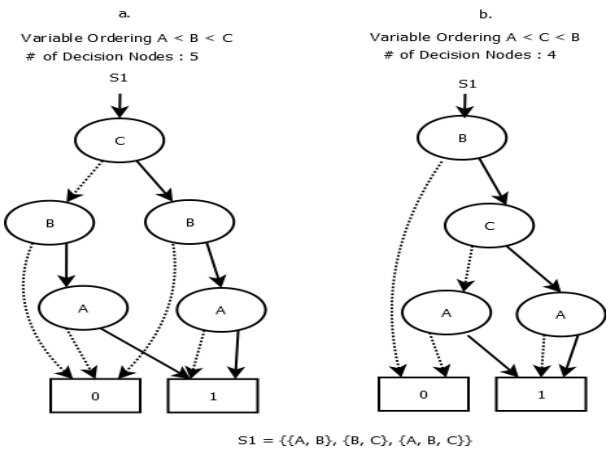

Figure 2-4 ZDD with variable ordering

The theoretical maximum number of decision node for the ZDD is the total number of variables presented in all subset of a set. However, the actual sizes of the ZDD nodes are smaller due to the node reusability.

3 Design and Theory

3.1 Exact NNS

3.1.1 Exhaustive NN Model

The exhaustive NN does not pre-process the document set S to build data structure or index to speed up the searching. In this model, a document p^i and query q is represented as Boolean vector $p^i = (p_1^i, \dots, p_k^i)$ and $q = (q_1, \dots, q_k). p_j^i, q_j \in \{0, 1\}$.

The model compares the similarity of each document features p_j^i with the query features q_i, and return the closest document p'. The time complexity of the exhaustive NN is O(nk) where n is the number of documents in document set S, and the k is the number of feature in the target document.

$$p' = argmax_{p^i \in S} \sum_{j=1}^{k} \sigma(p_j^i, q_j)$$

$$\sigma(p_k, q_k) = \begin{cases} 1 & p_k = q_k \\ 0 & p_k \neq q_k \end{cases}$$

3.1.2 Naïve NN-ZDD Model

The naïve NN-ZDD model is the fundamental NN-ZDD model and serves as a framework for other NN-ZDD models in the subsequence discussion. In this model, a document is represented as Boolean vector $p^i = (p_1^i, \dots, p_k^i). p_j^i \in \{0, 1\}$ represents the existence of term x_j in document i. The similarity between the documents and the query is measured by Hamming distances. In Section 3.1.3, we replace the Hamming Distance with Idf, and TF-IDF.

3.1.2.1 Build Phase

In the build phase, each document $p^i \in S$ is converted to a ZDD denoted as $ZDD(p^i)_{X_{p^i}}$ where $X_{p^i}$is the set of document features for p^i using ZDD Change operation (Algorithm 2) for each document feature $x_j \in p^i$,

$$ZDD(p^i)_{X_{p^i} \cup \{x_j\}} = ZDD(p^i)_{X_{p^i}} \cup \{x_j\}$$

an example is illustrated in Figure 3-1. The number of nodes in $ZDD(p^i)_{X_{p^i}}$ define number of document features.

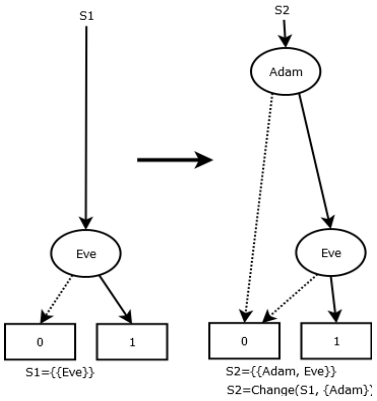

Figure 3-1 ZDD Change Operation on Text Doc

A Naïve NN-ZDD model is built by union all the $ZDD(p^i)_{X_{p^i}}$ (Algorithm 1),

$$ZDD(S) = \bigcup_{p^i \in S} ZDD(p^i)_{X_{p^i}}$$

and illustrated Figure 3-2 which union set S1 and S2 to form S3, and also union set S3 and S4 to form S5.

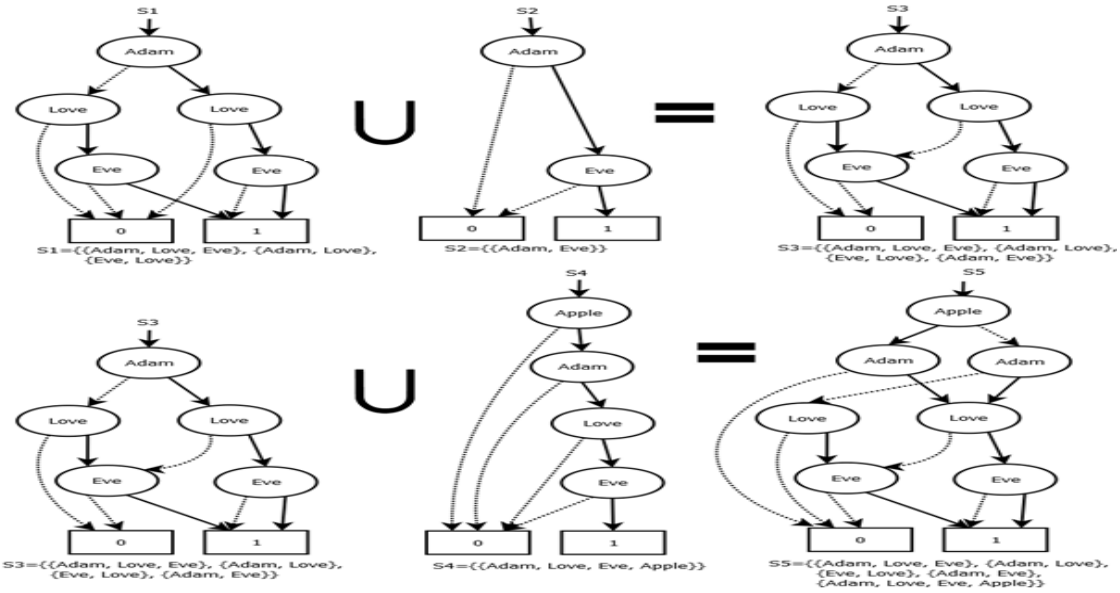

Figure 3-2 Example of ZDD Union for document set

3.1.2.2 Query Phrase

Finding a nearest neighbour for a query document in naïve NN-ZDD model is equivalent to finding the path from root node N^n of ZDD(S) to terminal node N^1 with maximum score. Dynamics programming calculates the maximum score from the

terminal nodes N^0 and N^1 back to the root node N^n. Each node N^j gets the maximum score $s(N^j)$ from its 0-edge node (N_0^j) and 1-edge node (N_1^j) with additional hamming similarity score $\sigma(x_k, q_k)$ if the term $x_k^{N^j}$ represented by N^j matches query term q_k. The $maxChild(N^j)$ is the child branch with maximum score (Algorithm 3 and Algorithm 4).

$$s(N^j) = \max(s(N_0^j), s(N_1^j) + \sigma(x_k^{N^j}, q_k))$$

$$\sigma\left(x_k^{N^j}, q_k\right) = \begin{cases} 1 & x_k^{N^j} = q_k \\ 0 & x_k^{N^j} \neq q_k \end{cases}$$

$$maxChild(N^j) = argmax_{b \in \{N_0^j, N_1^j\}} s(N^j)$$

The document features (terms) of the nearest neighbour document p_i to query q can be extracted from the 1-edge (N_1) with maximum score ($maxchild(N^i) = N_1$) from terminal node N^1 to root node N^n of ZDD(S). X_{N^i} is the set of extracted document features for p^i (Algorithm 5 and Algorithm 6).

$$X_{N^i} = \begin{cases} \emptyset & if\ N^i = N^1 \\ X_{maxchild(N^i)} \cup \{x^{N^i}\} & if\ N^i \notin \{N^0, N^1\} \wedge maxchild(N^i) = N_1 \\ X_{maxchild(N^i)} & if\ N^i \notin \{N^0, N^1\} \wedge maxchild(N^i) = N_0 \end{cases}$$

Algorithm 1: BuildNNZDD (D)

```
1  NNZDDTopic = ZDDBase                                  // Initialize ZDD structure
2  ZDDFeatureDict = {}                                   // Initialize ZDD Node Feature dictionary
3  while D is not empty do
4      d ← D.current()                                   // Assign document from set of documents
5      ZDDTopic ← DocumentToZDD(d, ZDDFeatureDict)       // Convert document to ZDD
6      NNZDDTopic = ZDDUnion(NNZDDTopic, ZDDTopic)
7      remove d from D
8  return NNZDDTopic
```

Algorithm 1 : Building NN-ZDD from documents

Algorithm 2: DocumentToZDD(d, ZDDFeatureDict)

```
1  ZDDTopic = ZDDBase                                    // Initialize ZDD structure
2  featues ← extractFeature(d)                           // Convert the document to a list of features
3  while feature is not empty do:
4      f ← feature.current()                             // Assign feature from a list of features
5      if not ZDDMember(ZDDFeatureDict, f) then          // If the feature not already the existing node
6          ZDDadd(ZDDFeatureDict, f)                     // Create a new node for this feature
7          ZDDNode ← ZDDLookup(ZDDFeatureDict, f)        // Assign the node for the feature.
8          ZDDTopic = ZDDChange(ZDDTopic, ZDDNode) // Use ZDD Change operation
                                                         // to add ZDDNode to ZDD Structure
9      remove f from features
10 return ZDDTopic
```

Algorithm 2 : Convert document to ZDD

Algorithm 3: ReduceScoreMain (Z, q)

```
1   MaxChild = {}                                  // Initialize dictionary to store high score edge
2   return ReduceScore(Z.root, q, MaxChild)        // Recursive call ReduceScore
```

Algorithm 3 : ReduceScore Main

Algorithm 4: ReduceScore (z, q, MaxChild)

```
1   if z is ZDDEmpty return ZDDEmptyCost           // If z reaches Terminal node 0 return the
2   if z is ZDDBase return ZDDBaseCost             // If z reaches Terminal node 1 return the
3   zLow ← 0-Edge of z                             // Assign 0-edge of z to zLow
4   zLowScore ← ReduceScore(zLow, q, MaxChild)     // Recursive call ReduceScore follow 0-edge
5   zHigh ← 1-Edge of z                            // Assign 1-edge of z to zHigh
6   zHighScore ← ReduceScore(High, q, MaxChild)    // Recursively call ReduceScore follow 1-edge
                + ZDDContain(z, q)                 // Give score if this node represent a doc feature
7   if zHighScore > zLowScore then                 // If the 1-edge score higher than 0-edge
8       MaxChild(z) = High                         // Set the child node to 1-edge
9   else
10      MaxChild(z) = Low                          // Set the child node to 0-edge
11  return Max(zHighScore, zLowScore)              // Return the score.
```

Algorithm 4 : ReduceSore to identify path with highest similarity to a Query

Algorithm 5: ExtractSetMain(Z, MaxChild, ZDDFeatureDict)

```
1   DocFeatures = {}                                                    // Initialize dictionary to store high score edge
2   z =Z.root                                                           // Start from the root node of ZDD
3   DocFeatures = ExtractSet(z, MaxChild, ZDDFeatureDict, DocFeatures)  // Recursively call ExtractSet
4   return DocFeatures                                                  // Recursive call extracted set
```

Algorithm 5 : Extract Set

Algorithm 6: ExtractSet(z, MaxChild, ZDDFeatureDict, DocFeatures)

```
1   If z is not ZDDEmpty or ZDDBase then                                // If z is not terminal node
2       If MaxChild(z) = High then                                      // If the child with higher score is 1-edge
3           f ←ZDDFeatureDict(z)                                        // Extract the feature represented by node z
4           DocFeatures.add(f)                                          // Add the feature to the Feature set
5           zHigh ← 1-Edge of z                                         // Assign 1-edge of z to zHigh
6           DocFeatures = ExtractSet(zHigh, MaxChild, ZDDFeatureDict, DocFeatures) // Recursive call
7       else
8           zLow ← 0-Edge of z                                          // Assign 0-edge of z to zLow
9           DocFeatures = ExtractSet(zLow, MaxChild, ZDDFeatureDict, DocFeatures) // Recursive call
10  return DocFeatures
```

Algorithm 6 : Extract the most similar features to a Query

3.1.2.3 Complexity

In this section, we analyse the space and time complexity of the Naïve NN-ZDD model.

3.1.2.3.1 Space Complexity

In the worst case scenario, each document p contains n distinct unique document features. The space complexity is O(dn) where d is the total number of document in set S. In the real world, documents share some common features which promote the node reusability, therefore the space complexity is much smaller than O(dn).

3.1.2.3.2 Time Complexity

3.1.2.3.2.1 Build Phase

Construct a ZDD chain for a document of n features required O(n). Union a document of n features to a ZDD of total m distinct feature in the worst case required max(O(n), O(m)). Therefore the complexity of building a Naïve NN-ZDD with d number of documents, each with n distinct feature would require d * (O(n) + max(O(n), O(m)).

3.1.2.3.2.2 Query Phase

To identify the path with maximum similarity score to the query document, we need to traverse to all the ZDD nodes in the Naïve NN-ZDD. It's proportional to the ZDD nodes size of the Naïve NN-ZDD. In the worst case scenario, the query complexity is O(dn) where d is the number of document and n is the number of distinct features of each document.

3.1.3 NN-ZDD Model for Different Metrics

In this section, we introduce two variants of NN-ZDD model with different scoring metrics, i.e. IDF and TF-IDF.

3.1.3.1 IDF

In this model, we replace the hamming similarity score $\sigma(x_k, q_k)$ in the naïve NN-ZDD with IDF score so the rare term get high score. The total number of nodes in ZDD(S) is same with Naïve NN-ZDD.

$$\sigma\left(x_k^{N^j}, q_k\right) = \begin{cases} idf_k & x_k^{N^j} = q_k \\ 0 & x_k^{N^j} \neq q_k \end{cases}$$

3.1.3.2 TF-IDF

Document is represented as $p^i = \left(p_1^i, \ldots, p_k^i\right), p_j^i \in N$ is the TF-IDF weight of $tf_{j,p^i} * idf_j$ in TF-IDF. To integrate the term frequency to the ZDD, we adopt the concept from Frequent Pattern (FP-tree) [9] to include the frequency information in the ZDD nodes (Figure 3-3), and represent the variable as $x_k : freq$.

$$\sigma\left(x_k^{N^j}, q_k\right) = \begin{cases} tf_k^{N^j} * idf_k & x_k^{N^j} = q_k \\ 0 & x_k^{N^j} \neq q_k \end{cases}$$

where $tf_k^{N^j}$ represents the term frequency of node N^j for variable $x_k : freq$

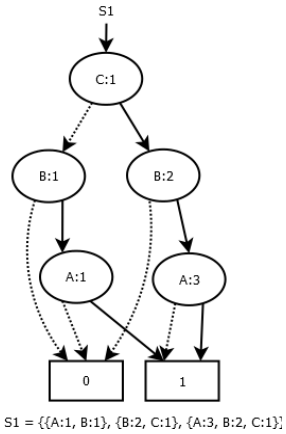

Figure 3-3 ZDD with Frequency Variables

3.2 Approximate NNS

3.2.1 NN Multi-hash Model

The time complexity of the exhaustive NN is $O(nk)$ where n is the number of documents in document set S, and the k is the number of feature in the target document. One way to improve the performance of exhaustive NN is to reduce size for both n and k during NN while preserving features for each document. Hash representation has the nice property of partition the set while preserving (with error) the features as we discussed in Section 2.4.2.

By hashing the document from $p^i = (p_1^i, \dots, p_k^i), p_j^i \in \{0,1\}$ to $h^l(p^i) = \{h_1^l, \dots, h_{m_l}^l\}$,

$h_n^l \in \{0,1\}$, m_l is the dimension of $h^l(p^i)$, and $m_l \ll k$, we could reduce the dimension of the document p^i from k to m_l. Because of the possible 'hash collision' (happened when mapping higher dimension vector to lower dimension vector), two or more documents might have the same hash representation, i.e. $h^l(p^i) = h^l(p^j) = \{h_1^l, \dots, h_{m_l}^l\}$, the maximum hash document size $|h^l|$ is 2^{m_l} but normally the actual $|h^l| \leq n$ and the equal hold if no hash collision. The time complexity of the NN query is reduced from $O(nk)$ to $O(|h^l|m_l)$.

However, number of documents which share the same hash representation might still be large, therefore, we repeating the same process for documents sharing the same hash representation, we also increase the hash size to separate the less similar from the more similar documents, i.e. $m_{l-1} < m_l$. Each of the increased size new hash representation h^l forms the branch of the previous sharing hash representation h^{l-1}.

We discuss the details of query phase in section 3.2.1.2.

3.2.1.1 Build Phase

In the build phase, each document p^i is initially hashed to root tree $h^1(p^i)$. The $h^1(p^i)$ is added to a set of root trees $F(S)$.

$$l = 1,$$

$$F(S) = F(S) \cup h^l(p^i)$$

The document p^i is then hashed to $h^2(p^i)$, and added to the branch of $h^1(p^i)$ (i.e. $b\left(h^1(p^i)\right) = b(h^1(p^i)) \cup h^2(p^i))$ where the hash size $m_2 > m_1$ (and $m_l > m_{l-1}$) .

The same process is repeated for document p^i from $h^2(p^i)$ to $h^{L-1}(p^i)$ where L is the depth of the tree, and $L > 2$.

$$l = 2..L-1,$$

$$b\left(h^{l-1}(p^i)\right) = b(h^{l-1}(p^i)) \cup h^l(p^i)$$

At the leaf, document p^i is added to the branch $b\left(h^{L-1}(p^i)\right)$.

$$l = L,$$

$$b\left(h^{l-1}(p^i)\right) = b(h^{l-1}(p^i)) \cup p^i$$

The next document p^j in the document set S goes through the same process from root to leaf till all the documents in S is placed at one of the leaf. Figure 3-4 illustrates the trees structure of $L = 4$. The algorithm is shown in Algorithm 7.

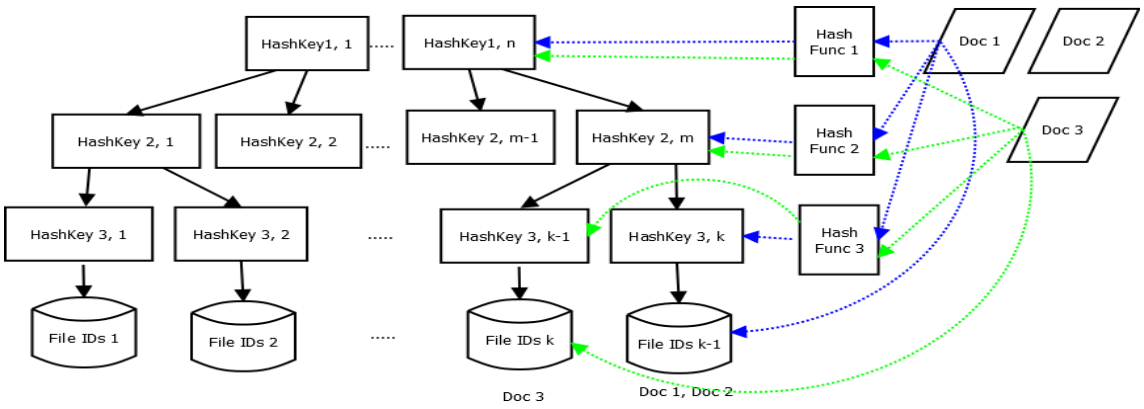

Figure 3-4 Build Phase for Multi-Hashing

3.2.1.2 Query Phase

In the query phase, the query q is initially hashed to $h^1(q)$ and performed exhaustive NN on $F(S)$ and identified the nearest neighbour t'^1. m_1 is the dimension of $h^1(q)$.

$$l = 1,$$

$$t'^l = argmax_{t^l \in F(S)} \sum_{j=1}^{m_l} \sigma(t_j^l, h_j^l(q))$$

Where

$$F(S) = \bigcup_{p^i \in S} h^l(p^i)$$

$$\sigma(t_k, h_k(q)) = \begin{cases} 1 & t_k = h_k(q) \\ 0 & t_k \neq h_k(q) \end{cases}$$

The query q is then hashed to $h^2(q)$ and performed exhaustive NN with all the hash values of the branch of t'^1 ($b(t'^1)$) and identified the nearest neighbour t'^2. The same process is repeated till $L-1$.

$$l = 2..L-1,$$

$$t'^l = argmax_{t^l \in b(t'^{(l-1)})} \sum_{j=1}^{m_l} \sigma(t_j^l, h_j^l(q))$$

$$\sigma(t_k, h_k(q)) = \begin{cases} 1 & t_k = h_k(q) \\ 0 & t_k \neq h_k(q) \end{cases}$$

At the leaf, the query q is performed the exhaustive NN search on all the documents $p^i \in b(t'^{(l-1)})$.

$$l = L,$$

$$p' = argmax_{p^i \in b(t'^{(l-1)})} \sum_{j=1}^{k} \sigma(p_j^i, q_j)$$

$$\sigma(p_k, q_k) = \begin{cases} 1 & p_k = q_k \\ 0 & p_k \neq q_k \end{cases}$$

An example is illustrated in Figure 3-5, and the algorithm is shown in Algorithm 8.

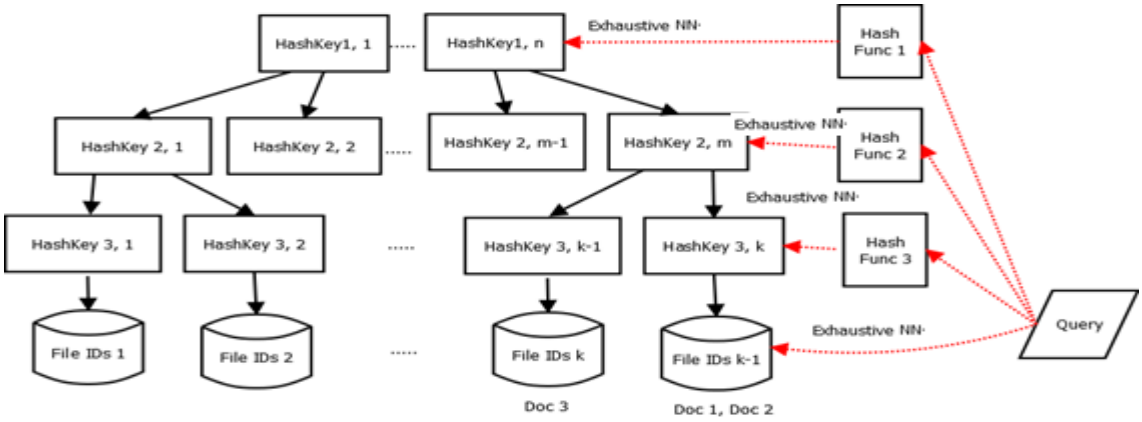

Figure 3-5 Query Phase for Multi-Hashing

Algorithm 7: BuildNNMultiHash(D, L)

```
1   while D is not empty do
2       h_p(d) = null                                   // Initialize parent tree
3       d ← D.current()                                 // Assign document from set of documents
4       for each l ∈ L do
5           if l = 1 and h_l(d) ∉ F(s) then             // if the hash value of d is not exist at Level 1
6               F(s) ← F(s) ∪ h_l(d)                    // Add hash value of d to the hash set at Level 1
7           else if l = L then                          // if it's the leaf
8               b(h_p(d)) ← b(h_p(d)) ∪ d               // Add the doc to the branch of parent tree h_p(d)
9           else
10              b(h_p(d)) ← b(h_p(d)) ∪ h_l(d)          // Add hash value to the branch of parent tree h_p(d)
11          end if
12          h_p(d) ← h_l(d)                             // Assign the current hash value as parent tree.
13      end for
14
```

Algorithm 7 : Build NN Multi-hash

Algorithm 8: QueryNNMultiHash(q)

```
1   for each l ∈ L do
2       h_p(t) = null                                   // Initialize parent tree
3       if l = 1 then then                              // if Level 1
4           t = nn(F(s), h_l(q))                        // Perform nn on F(S) with h_l(q)
5       else if l = L then                              // if leaf
6           p = nnFile(b(h_p(t), q))                    // Perform nn on all the files from branch of parent tree
7                                                         h_p(t) with q
8           return p                                    // return the closest files to q
9       else
10          t = nn(b(h_p(t), h_l(q)))                   // Perform nn on all hash values from branch of parent tree
11                                                        h_p(t) with h_l(q)
12      end if
13      h_p(t) ←t                                       // Assign the current hash value as parent tree.
14  end for
15
```

Algorithm 8 : Query NN Multi-hash

3.2.1.3 Complexity

In this section, we analyse the space and time complexity of the NN Multi-hash model.

3.2.1.3.1 Space Complexity

In the worst case scenario, all documents n spans all hash values $|h^l|$ (with hash collision) if $|h^l| \leq n$ or n of them have its owner hash value if $|h^l| \geq n$ in Level l. Therefore, the space complexity in Level l is $\min(|h^l|, n)$.

For NN Multi-hash with Level L, the space complexity is, $O(\prod_{l=1}^{L} \min(|h^l|, n))$, which is much higher than exhaustive NN of O(dn). However, by proper selecting the hash function which balance the tree to achieve $|h^l| \ll n$.

3.2.1.3.2 Time Complexity

3.2.1.3.2.1 Build Phase

Construct a NN Multi-hash for n documents with L Level, each document is required to hashed L times only, therefore the overall time complexity for build phase is $O(nL)$.

3.2.1.3.2.2 Query Phase

Assume that n documents are evenly distributed at each Level l during build phase, the complexity of the query time for performing exhaustive NN for all level is

$$O(|F(S)| * m_1 + \sum_{l=2}^{L-1} \left|b\left(t'^{(l-1)}\right)\right| * m_l + k n_{b\left(t'^{(L-1)}\right)}).$$

The $|F(S)|$ is the size of the root of trees, $\left|b\left(t'^{(l-1)}\right)\right|$ is the branch which parent branch has the most similar hash value to the query. m_l is the dimension of hash value at Level l. k is the dimension of the document, and $n_{b\left(t'^{(L-1)}\right)}$ is the number of documents that fall under this particular branch.

3.2.2 NN-ZDD Multi-hash Model

In the NN Multi-hash Model, we perform exhaustive NN on each branch of the tree from the root to leaf. As we've seen in Section 3.1.2.2, the Naïve NN-ZDD has better query time than the exhaustive NN. Therefore, we construct ZDD at each branch during the build phase and perform query on the ZDD which is more efficient as shown in Section 4.4.3.2

4 Evaluation

In this section, we introduce the details of our experiments and show the results.

4.1 Data Set Description

These models are evaluated with the newsgroups data. The newsgroups data set is a collection of approximate 10,000 newsgroup documents which contains 11 different newsgroups of different topics in the text file format. The filtering process performs tokenizing, stop word removal and stemming on text files and passes the result to NN models.

4.2 Experiment Setup

We construct 3 types of experiments to demonstrate the ZDD properties and the performances of NN-ZDD models. In section 4.4.1, we demonstrate two important properties of ZDD, i.e. relationship of ZDD node and file sizes, and variable ordering to ZDD node sizes. In section 4.4.2, we focus on exact NNS. We evaluate the performance of NN-ZDD with Exhaustive NN, and also demonstrate NN-ZDD models of various metrics. In section 4.4.3, we compare the performance of NN-ZDD-MLH with LSH.

4.3 Performance Metrics

Four performance metrics measure and compare the performances of NN models as defined in Table 4-1.

Performance Metric	Description
Build Time (ms)	Time to build the data structure for effective querying.
Query Time (ms)	Time retrieve the nearest neighbour document.
Accuracy (%)	The percentage of the nearest neighbour docs correctly matches the queries.
Node Size	Number of Nodes in the ZDD (ZDD only).

Table 4-1 Performance Metrics

4.4 Results

4.4.1 ZDD Properties

4.4.1.1 ZDD Node Size and No of Files

The ZDD node size is increasing linearly with the number of file. The ZDD search is as good as linear search in the worst case.

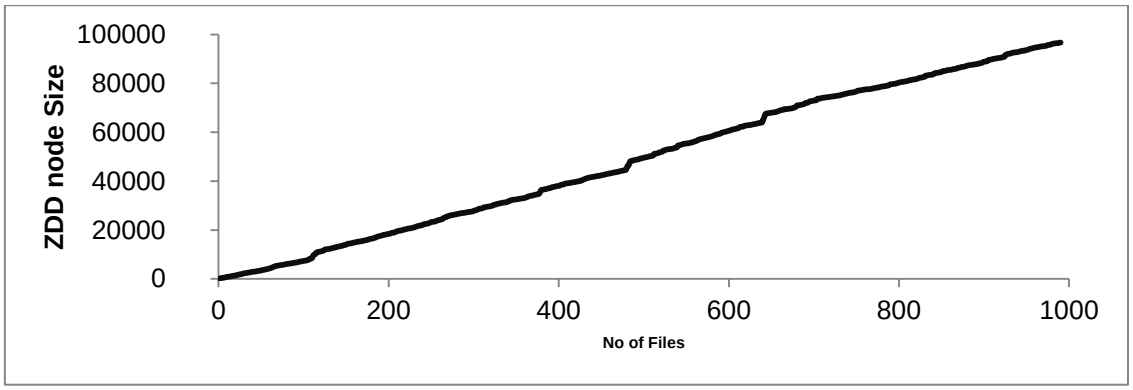

Figure 4-1 ZDD Node Size and number of Files Relation

4.4.1.2 Variable Ordering and ZDD Node Size

We've discussed the variable ordering influences ZDD node size in section 2.5. In this experiment, we are using data set sci.med which contains 16914 distinct document terms. We sort the variable ordering based on inverse document frequency (IDF) in both ascending and descending order, and compared with case without ordering. The result is shown in Table 4-2.

Ordering	Number of ZDD Nodes
IDF Descending	97789
IDF Ascending	97795
No Ordering	99727

Table 4-2 : ZDD sizes with various Ordering

In the text document ZDD, the variable ordering has slightly influences on the ZDD node sizes (2% less than no ordering). Therefore, our NN-ZDD models do not implement the variable ordering to improve the build time.

4.4.2 Exact Search

4.4.2.1 Query Time: Exhausting NN vs Naïve NN-ZDD.

We compare the total times required for building and querying 701 files (alt.atheism) for the exhaustive NN and Naïve NN-ZDD. The Naïve NN-ZDD is order of magnitude faster than the exhaustive NN (Figure 4-2).

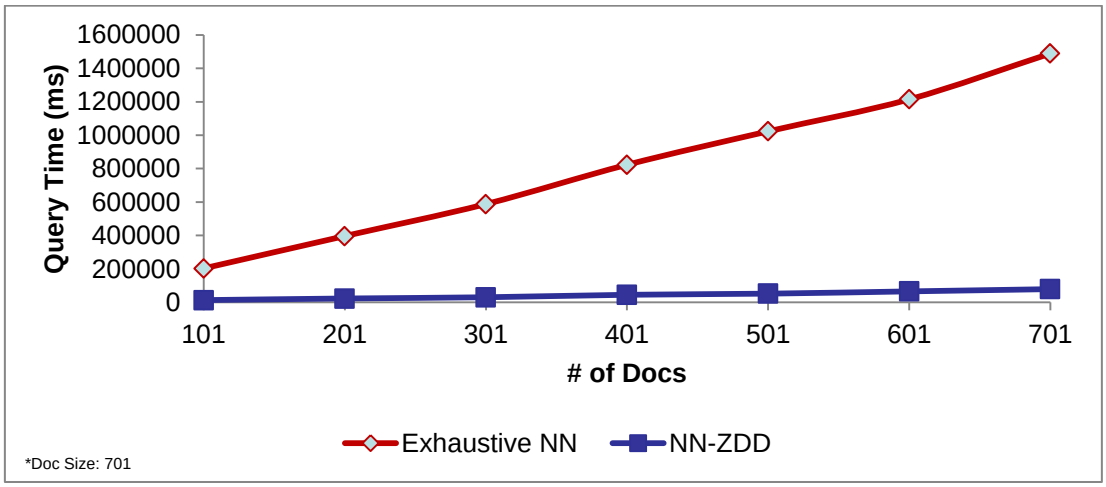

Figure 4-2 Query Time for Exhaustive NN and Naive NN-ZDD

4.4.2.2 Comparison of Exact NNS for different metrics

We compare NN-ZDD models with three different metrics (i.e. Hamming Distance, IDF, and TF-IDF). The Hamming Distance and IDF have similar query time and identical number of nodes, and 100% accuracy. However, TF-IDF has slight higher number of nodes, longer query time, and degraded accuracy. NN-ZDD models are suitable for non-frequency term.

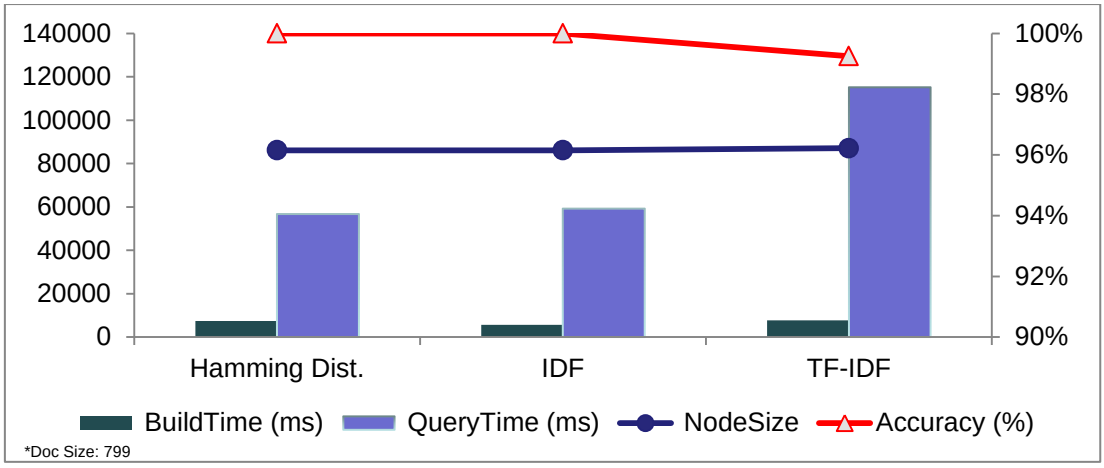

Figure 4-3 Exact NN-ZDD for Different Metrics

The degraded accuracy of NN-ZDD TF-IDF is because the query scores high in certain documents that the terms matched have higher frequency without containing all the terms as shown in Figure 4-4. One way to reduce this effect is to multiply the TF-IDF score with the fraction of all query terms matched (as Apache Lucene does).

Doc1 = {Adam:10, Eve:5}
Doc2= {Adam:5, Love:5, Eve:5}

TF	Adam	Love	Eve
1	10	0	5
2	5	5	5

Idf	Adam	Love	Eve
N = 2	2	1	2

Query = {Adam, Love, Eve}

TF-IDF	Adam	Love	Eve	Score
1	**6.02**	**0**	**3.01**	**9.03**
2	3.01	2.39	3.01	8.4

Figure 4-4 TF-IDF Issues: The similar doc. to query should be D2, but D1 has higher TF-IDF.

4.4.3 Approximate Search

4.4.3.1 Comparison of Approximated NNS for Small Data Set

We compare the accuracy, build and query time for 799 files (alt.atheism) among NN-ZDD Hash, Idf App(roximate), LSH Minhash (Baseline), Multi-Hash and ZDD Multi-Hash (Figure 4-5).

The LSH Minhash, Multi-Hash, and ZDD Multi-Hash have 100% accuracy and LSH Minhash has the lowest query time. But ZDD Multi-Hash has least build time.

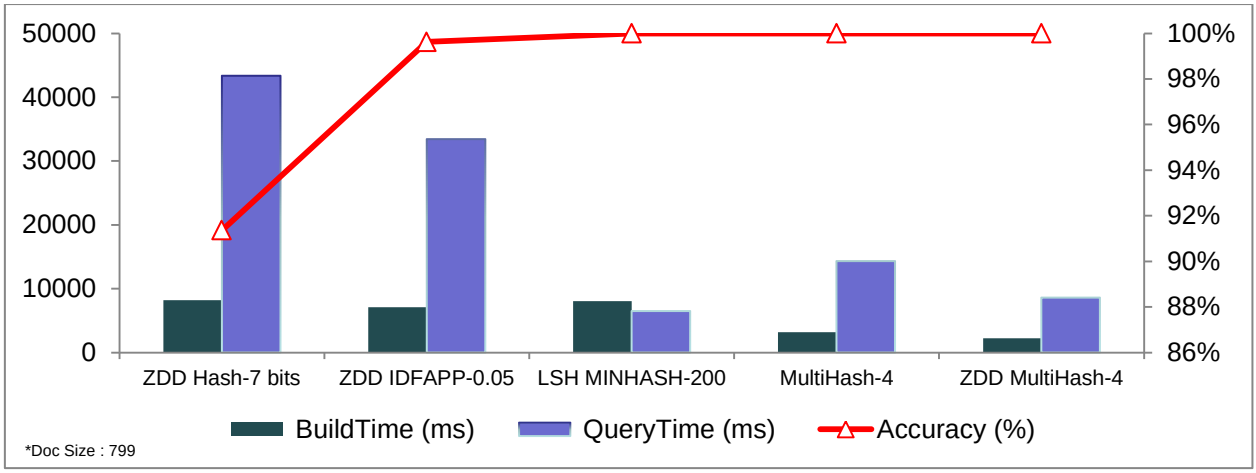

Figure 4-5 Approximate NNS (Smaller Data Sets)

4.4.3.2 Comparison of Approximated NNS for Large Data Set

We compare the accuracy, build and query time for 2,000 files among LSH Minhash (Baseline), Multi-Hash and ZDD Multi-Hash (Figure 4-6). The accuracy of LSH Minhash drops to 99.90%. The query time of ZDD Multi-Hash and LSH MinHash are comparable.

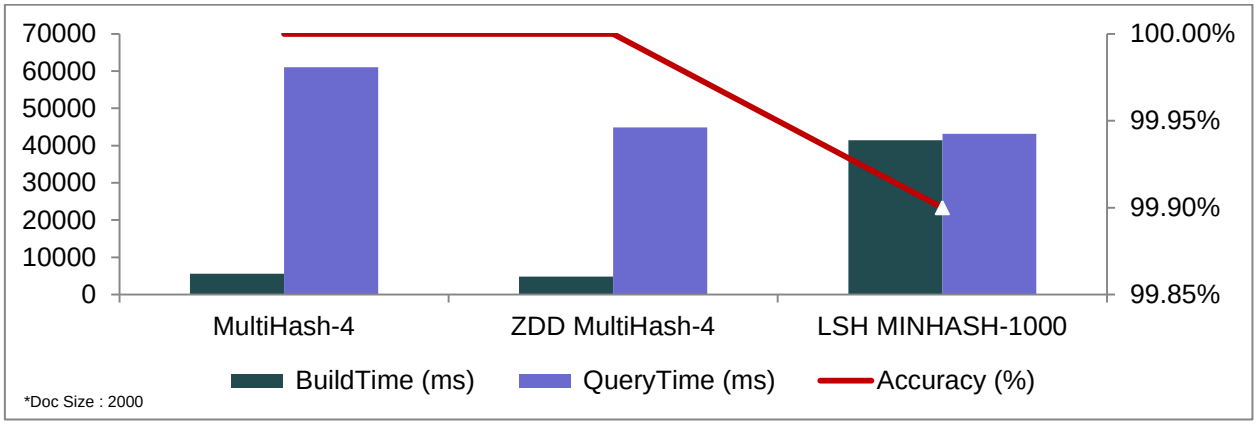

Figure 4-6 Approximate NNS (Large Data Sets)

4.4.3.3 Comparison of Approximated NNS for Huge Data Set

We compare the accuracy, build and query time for 10,003 files for LSH Minhash (Baseline), and Multi-Hash (Figure 4-7). ZDD Multi-Hash is excluded because the data structure used in the 3rd party packages (initialize large integer array, hash map should be used instead) is inefficient. However, we could infer the results based on the fact that

ZDD Multi-Hash performs query more effective than Multi-Hash, and get the glimpse of ZDD Multi-Hash performance at huge data set. The conclusion is same as the large data set.

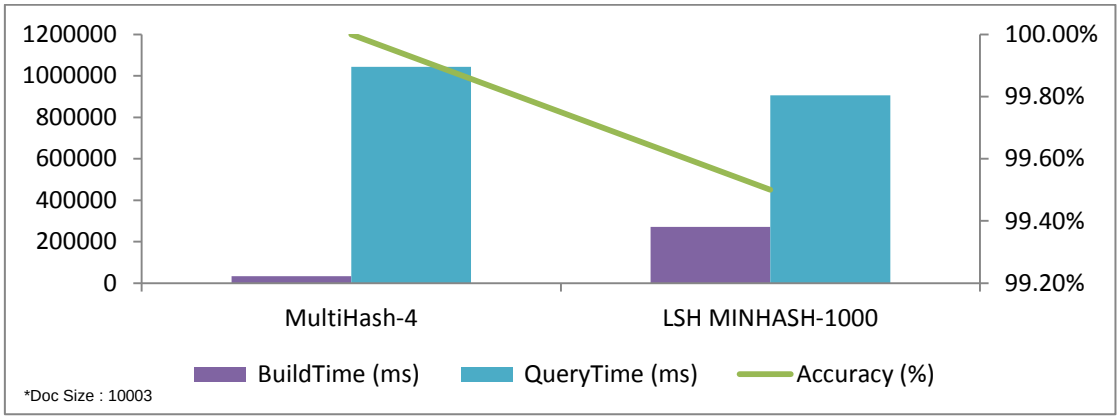

Figure 4-7 Approximate NNS (Huge Data Sets)

5 Conclusion and Future Work

In this paper, we focus on NNS for text. We propose an exact NNS Naïve NN-ZDD model. Its query time is order of magnitude faster than exhaustive NN for text. We also propose an approximate NNS ZDD Multi-hash model. The query time is closer to LSH with less build time and better accuracy.

In the future work, for exact search, we would seek the better heuristic for finding the path with highest score for Naïve NN-ZDD model using A*Star + DFS. Besides, we would explore using Weighted Zero-suppressed Binary Decision Diagram (WZDD) [3] for TF-IDF. For Approximate search, we would explore other hashing functions (Minhash/Simhash) which preserve more document features than the simple hashing use for NN-ZDD Multi-hash.

6 References

[0] Mishchenko, Alan. "An introduction to zero-suppressed binary decision diagrams." URL: http://www. ee. pdx. edu/alanmi/research. htm (2001).

[1] Algorithms for Nearest Neighbor Search, tutorial by Yury Lifshits. RuSSIR'07, Ekaterinburg, September 2007

[2] Minato, Shin-ichi. "Zero-suppressed BDDs for set manipulation in combinatorial problems." Design Automation, 1993. 30th Conference on. IEEE, 1993.

[3] Loekito, Elsa, and James Bailey. "Are zero-suppressed binary decision diagrams good for mining frequent patterns in high dimensional datasets?." Proceedings of the sixth Australasian conference on Data mining and analytics-Volume 70. Australian Computer Society, Inc., 2007.

[4] "MinHash" Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc., date last updated (15 April 2013). Web. Date accessed (26 May 2013). http://en.wikipedia.org/wiki/MinHash

[5] Manku, Gurmeet Singh, Arvind Jain, and Anish Das Sarma. "Detecting near-duplicates for web crawling." Proceedings of the 16th international conference on World Wide Web. ACM, 2007.

[6] Das, Abhinandan S., et al. "Google news personalization: scalable online collaborative filtering." Proceedings of the 16th international conference on World Wide Web. ACM, 2007.

[7] Goodman, Jacob E., and Joseph O'Rourke, eds. Handbook of discrete and computational geometry. Chapman and Hall/CRC, 2004.

[8] Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze. Introduction to information retrieval. Vol. 1. Cambridge: Cambridge University Press, 2008.

[9] Han, Jiawei, et al. "Mining frequent patterns without candidate generation: A frequent-pattern tree approach." Data mining and knowledge discovery 8.1 (2004): 53-87.

[10] Rajaraman, Anand, and Jeffrey David Ullman. Mining of massive datasets. Cambridge University Press, 2011.