Masters’s Thesis

Fast Search for Large Entries in a Matrix Product

submitted by

Olga Mykytiuk

on 31.07.2012

Supervisor
Dr. Rainer Gemulla

Advisor
Dr. Rainer Gemulla

Reviewers
Dr. Rainer Gemulla
Prof. Dr. Gerhard Weikum
Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 31.07.2012
(Datum / Date) (Unterschrift / Signature)
Abstract

In this thesis we formalize and solve the problem of finding large entries in a matrix product. This problem arises in Probabilistic Latent Semantic Analysis [Hof99]. Fast search for large entries in a matrix product can be used in recommendation systems like Netflix [KBV09]. A naive approach for this problem is to multiply both matrices and take all the entries in the resulting matrix larger than a predefined threshold. Since the naive approach takes an unacceptably large amount of time (of the order of days), we design and implement more efficient solutions. SPEEDY solution takes advantage of the data distribution and outperforms the naive approach by a factor of 27,000. SPEEDY is the most competitive algorithm of the considered algorithms. Optimized Locality sensitive hashing-based matrix multiplication Algorithm (OLGA) performs better than the naive approach but needs expensive input parameter tuning. THRESH based on the threshold algorithm [FLN01] outperforms the naive algorithm by a factor of 500 and doesn’t need any input parameters. We compare the strengths and weaknesses of these algorithms and suggest the best choice for each particular kind of input data. We justify all our statements by conducting multiple experiments with the discussed algorithms.
Acknowledgments

Many thanks to Alex, Victor, Benjamin and Maksim for their help and support while working at this thesis. I also thank to all my friends who visited me in the hospital.
1 Introduction

In the modern information society, we face an overwhelming amount of information every day. Despite the value of this information, if not properly analyzed and processed, it can be anything but useful. In this case, Information Retrieval comes into play trying to extract the meaningful part out of the huge amount of data in World Wide Web and other sources. In this thesis we consider a problem of finding large entries from a matrix product. This problem arises in different areas of Information Retrieval such as Probabilistic Latent Semantic Analysis, text mining and recommendation systems. A naive approach for this problem is to multiply both matrices and take all the entries in the resulting matrix larger than a predefined threshold. The naive approach takes unacceptably long for large matrix sizes and therefore we look for better solutions.

A Probabilistic Latent Semantic Analysis (PLSA) [Hof99] is a clustering technique of information extraction that is used to characterize documents and assign them to a certain category. Having a collection of \(w\) documents indexed with \(d\) words, PLSA models the word-document co-occurrence by matrix \(V\). Every \(v_{ij}\) in \(V\) shows the number of times a word \(w_i\) occurred in a document \(d_j\). PLSA decomposes \(V\) using a probabilistic model into a product of two matrices \(W\) and \(H\) such that \(V \approx WH\). In this case, \(W\) refers to the word-category relation, and \(H\) refers the document-category interaction. Finding these matrices \(W\) and \(H\) implies finding latent features (factors) of every document. Contrary to the original \(V\), the new product matrix \(WH\) is denoised and reflects only crucial word-document relations. Entries in the product \(WH\) correspond to the probability for a certain word to occur in a certain document. Therefore, the largest entries are the most meaningful. Commonly, \(W\) and \(H\) are large and finding these large entries is a challenge. After obtaining \(H\), one can find all similar documents to a particular one. Usually, the cosine similarity is used as a similarity measure between the documents [Hof99].

Matrix factorization is a set of methods of linear algebra that decomposes a matrix \(V\) into some approximation \(V \approx WH\). These methods are successfully applied in recommendation systems like Netflix [KBV09] where the matrix \(V\) reflects the users ratings of movies. Usually, this matrix \(V\) is incomplete, i.e. some entries of \(V\) are undefined, since not all users can rate all movies. \(V\) is decomposed into a product of the user preferences matrix and the movies factors matrix. The meaning of these latent factors is not always possible to determine, but they can refer, to some extent, to properties like genre, amount of action or depth of character.
Introduction

development. The matrix product $WH$ predicts which user prefers which movie. Therefore, the recommendation system can still predict how a particular user might rate an unrated movie. Entries in the matrix $WH$ with large values correspond to movies that appear to be valuable for the user. Commonly, both users matrix and the movies matrix are large. A fast search for those large entries is a challenging problem. Another challenge is to quickly find a similar movie to the one the user found interesting.

Matrix factorization is used in knowledge extraction and text mining. Assume, one has retrieved a number of subject-object pairs and patterns between them from a large corpus of text like Wikipedia. Furthermore, one has learned a number of factor features for every pattern. These factor features were learned from a matrix factorization model, which maps both subject-object and pattern spaces to a joint latent factor space. Inner product in this space models the subject-object-pattern interaction [KBV09]. In this setting, $W$ corresponds to the subject-object matrix and $H$ to the pattern matrix. Using any metric one can establish similarities (semantic correlations) between the patterns. For example, a pattern chief executive of is similar to president of in the business domain.

These semantic correlations can be used to enrich the results of a database query. Suppose, one is looking for the place where Albert Einstein was born by issuing a SPARQL query [PAG09]: "Albert Einstein" "was born in" ?x, where $x$ is an output variable. If there is no exact match for the pattern was born in in the data, then the result is empty. However, if instead of was born in one would also look for place of birth, the probability of finding a non-empty result would increase. In our example it is possible, because was born in and place of birth are conceptually the same.

The product $WH$ corresponds to the estimated probability that a particular pattern exists between a certain subject and object, i.e. to the existence of a certain fact. Large entries from the product matrix $WH$ refer to the facts that are assumed to be true. However, the matrices $W$ and $H$ are of enormous size and finding those large entries fast is a challenging problem.

In this thesis we formalize and solve the problem of quickly finding large entries in a matrix product. We conduct our experiments in the aforementioned subject-object-pattern setting. In order to solve this main problem we define and solve a subproblem of finding similar vectors in a set of vectors. We designed three new algorithms for finding large entries in a matrix product: SPEEDY, THRESH and OLGA. Our best algorithm achieves a speed-up by a factor of 27000 over the naive approach. We have conducted many experiments in order to benchmark the implemented algorithms. The weaknesses as well as the strengths of every algorithm are studied and described. In conclusion we point out which algorithm solves a particular problem the best.

In Chapter 2 we formalize the problem of finding similar patterns in the subject-object-pattern settings and describe its possible solutions. We examine possible solutions for our main problem of fast finding large entries in matrix product and discuss their strengths and weaknesses in Chapter 3. In Chapter 4 we compare different approaches and their execution times. Finally, we conclude and discuss directions for future work in Chapter 5.
2 Efficient Similar Patterns Search

2.1 Problem Definition

In this section we present the problem statement, notation and terminology.

The inner product is an algebraic operation that takes two vectors coordinates and returns a number obtained by multiplying corresponding entries and then summing those products. The inner product of vectors $a$ and $b$ is defined as follows: $ab^T = \sum_i a_i \cdot b_i$. The cosine similarity for vectors $a$ and $b$ is defined as cosine of the angle between them: $\cos \theta(a, b) = \frac{ab^T}{\|a\|\|b\|}$ [RPH05].

The Euclidean distance between points $a$ and $b$ is defined as $\|a - b\| = \sqrt{\sum_i (a_i - b_i)^2}$.

In this thesis we deal with specific kind of algorithms, so called anytime algorithms. Anytime algorithm is an algorithm that returns a valid solution to a problem even if it is interrupted at any time before it stops. Such an algorithm is expected to find better or more precise solutions the more time it keeps running.

We formalize two problems in the domain of subject-object-pattern triples. We are given two matrices, an $(n \times r)$ matrix of patterns $H$ and an $(m \times r)$ matrix $W$ consisting of subject-object pairs. These matrices are the resulting matrices of the decomposition process of matrix $V$, where $V = WH^T$.

**Similarity threshold problem:** Given a query vector $p \in \mathbb{R}^r$, which denotes a pattern in a latent factor space, and a number $\theta \in \mathbb{R}$, find all such vectors $w_i$ in $W$ (or $h_i$ in $H$) such that $pw_i^T > \theta$.

So, we are interested in all vectors, which similarity to the query vector is larger than $\theta$.

**Similarity top-$k$ problem:** Given a query vector $p \in \mathbb{R}^r$, which denotes a pattern in a latent factor space, and a number $\theta \in \mathbb{R}$, find all such vectors $w_i$ in $W$ (or $h_i$ in $H$) such that $w_i \in top_k(pw_i^T)$.

In this problem we are interested in all top-$k$ similar vectors to the query vector.

One can use different distance or similarity measures between vectors in a multidimensional space. The possible choices are Euclidean distance, cosine similarity, L1 norm or others. We focus mainly on cosine similarity.
<table>
<thead>
<tr>
<th>pattern</th>
<th>feature 1</th>
<th>feature 2</th>
<th>...</th>
<th>feature r</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;wife of&quot;</td>
<td>4</td>
<td>6</td>
<td>...</td>
<td>80</td>
</tr>
<tr>
<td>&quot;place of birth&quot;</td>
<td>34</td>
<td>-7</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>&quot;drugs dealer&quot;</td>
<td>2</td>
<td>-34</td>
<td>...</td>
<td>-100</td>
</tr>
<tr>
<td>&quot;was born in&quot;</td>
<td>36</td>
<td>-10</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>&quot;grew up&quot;</td>
<td>42</td>
<td>-80</td>
<td>...</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2.1: Matrix of patterns $H$

### 2.2 Naive Solution

First we consider the **Similarity threshold problem** in this chapter. We search for similar vectors in a pattern matrix $H$ in the subject-object-pattern setting partially displayed in Table 2.1.

Each vector in the pattern matrix $H$ represents a certain pattern, where individual values correspond to the values of features (latent factors) for a particular vector.

**Algorithm 1 NaiveTopk**

1: input: $p$, $H$, $k$
2: $q ←$ empty priority queue
3: for all $v$ in $H$ do
4: $d ← pv^T$
5: maintainTop$_k(q, v, d)$
6: end for
7: return $q$

For the Similarity threshold problem we go through all the patterns in Algorithm 2 outputting only those, whose calculated similarity to a given vector is larger than a predefined threshold, see in line 3. The threshold value is specified by the user. We call the naive approach solution for Similarity threshold problem $NaiveThresh$.

**Algorithm 2 NaiveThresh**

1: input: $p$, $H$, $\theta$
2: for all $v$ in $H$ do
3: if $pv^T > \theta$ then
4: output $v, pv^T$
5: end if
6: end for

For solving Similarity top-$k$ problem we maintain a priority queue of cardinality $k$. If we find a vector with similarity larger than the minimum in the queue, we add the newly found
2.3 Threshold Algorithm

-vectorId | column 0 | column 1 \\
---|---|---
0 | 5 | 10 \\
1 | 8 | 3 \\
2 | 1 | -5 \\
3 | 0 | -10 \\

(a) Initial $H$

-vectorId | column 0 | column 1 \\
---|---|---
(8,1) | (10,0) \\
(5,0) | (3,1) \\
(1,2) | (-5,2) \\
(0,3) | (-10,3) \\

(b) Sorted matrix $H$

Table 2.2: Example of threshold algorithm execution

vector to the queue removing the minimal vector if necessary in line 5 of the Algorithm 1. We call the naive approach solution for the Similarity top-$k$ problem NaiveTopk.

The naive approach has complexity $nr + n \log k$ in case of the Similarity top-$k$ problem and $nr$ for the Similarity threshold problem. We focus on the cosine similarity as a similarity measure between two vectors. For normalized vectors, cosine similarity corresponds to the inner product between these vectors.

The threshold algorithm [FLN01] is aimed to return the points, corresponding to top-$k$ results of a function calculated on multidimensional data without having to compute the function for all existing data points. The important fact is that the function we want to estimate is monotonic. A function $f$ defined on a subset of the real numbers with real values is called monotonic, if for all $x$ and $y$ such that $x \leq y$ it holds $f(x) \leq f(y)$. Inner product is monotonic with respect to the vector coordinates. We call the algorithm TA, sketch TA in Algorithm 3 and introduce it on a small example. We denote the version of TA in the case of the Similarity top-$k$ problem as TAtopK and TAthresh in the case of the Similarity threshold problem.

Suppose, we have a matrix $H$ shown on the Table 2.2a. Our query vector is $p = v_1 = (8, 3)$ and we search for top-2 similar vectors to the query vector. We copy the matrix data and sort each column in descending order in Table 2.2b.
In this sorted matrix we store the sorted value of a coordinate itself and the index of the vector, which this coordinate value belongs to. Since the data is sorted by columns, the first row now contains coordinates corresponding to different data points. Next, we observe values and the respective point ids in the first row. In our case they are equal to 8 and 10 from the 1-st and the 0-th vectors respectively. We do a random access and get all the complementary coordinates of the points from the first row, i.e. we get vectorId: [(5, 10), (8, 3)]. Now, we calculate the similarity function for every point seen so far and store top-k pairs of the function values and vector ids in a heap of length k sorted by the function value, i.e. \[ pv^T_0 = 70 \] and \[ pv^T_1 = 73 \].

At this moment, in order to learn whether we need to proceed downwards over the sorted matrix we calculate the threshold value of the current row. We treat all coordinates of the current row as if they would belong to one point and compute inner product of the current row and \( p \):

\[
\text{threshold} := 8 \times 8 + 3 \times 10 = 94
\]

If the minimal value in the maintained heap is larger than the calculated threshold, we stop. Intuitively, the calculated threshold corresponds to the possible maximal value that was not seen yet. In this example this is not a case, since 70 < 94. We pick a column, suppose column 0. The next value in this column is (5, 0). There is no need to update the heap, because we have already seen this vector. The threshold gets a new value: \( \text{threshold} = 8 \times 5 + 3 \times 10 = 70 \). Since the threshold is smaller or equal than the minimal element in the heap, there is for sure no unseen point with a larger similarity value. At this step we stop the execution, without having to observe the whole matrix \( H \) and output \([(1, 73), (0, 70)]\).

**Algorithm 3** TA

```
1: input: \( p, H, k \)
2: \( q \leftarrow \) empty priority queue
3: for all column in \( H \) do
4: copy and sort(column)
5: end for
6: \( i \leftarrow 0 \)
7: while (\( i < \text{size(column)} \) and (not (calculated threshold < min(q)) or size(q) < \( k \))) do
8: pick a column \( c \) from sorted \( H \)
9: \( id \leftarrow \) id from \( i \)-th row and \( c \)-th column in sorted matrix \( H \)
10: \( d \leftarrow p, v^T_{id} \)
11: maintainTop_k(q, d, v_{id})
12: update threshold
13: \( i ++ \)
14: end while
15: return q
```

**Implementation** In our implementation we store the matrix as a vector of columns including sorted as well as unsorted data. We normalize the matrix vectors beforehand to accelerate the
2.3 Threshold Algorithm

cosine similarity calculation. We execute algorithm in two steps. In the first step a number of candidate points is chosen and the cosine similarity is calculated for each of them. In the second step a threshold value is calculated and compared to the last element in a maintained descending sorted set. The second step returns false if the last element from the set is larger than a calculated threshold. The algorithm repeats the first step while the second step returns true or the result set contains less elements than $k$.

Let’s consider the heuristic for choosing candidate points. For applying the heuristic we look whether the weight of the query vector for a certain column is positive, i.e. a corresponding coordinate value. If it is positive, then we traverse the corresponding column top-down and pick the candidate points from the column (since they are sorted in a descending order). Otherwise, if the coordinate of query vector is negative, we examine points bottom-up. The columns are not treated equally. We process that column first, where the value in a current row multiplied by the corresponding query vector coordinate yields the largest result. Naturally, the favored column may change during the execution. As experiments show, applying such a heuristic reduces a runtime of algorithm at least by a factor of two.

In order not to waste the resources, we maintain a set of already seen points. If we meet some point more than once during the execution, we do not need to calculate the cosine distance for it every time we see this point.

**Optimization** One possible way of accelerating TA is choosing a better way for the threshold calculation for normalized data. Instead of multiplying the query vector with the current row values of the opposite matrix, we could find such vector $x$ that maximizes the threshold value:

$$\max_x x^T q$$

$x$ is a subject to the following constraints:

$$\begin{cases} 
  x^T x = 1 \\
  x_i \leq u_i 
\end{cases}$$

where $u_i$ are current values in each column and comprises a vector $u$. This is a problem of convex optimization with quadratic and linear constraints. We write down the Lagrangian for the problem uniting all the constraints:

$$L(x, \mu, \nu) = -x^T q + \mu(x^T x - 1) + \nu^T (x - u)$$

where $\mu, \nu$ are Lagrangian multipliers. Note that $\nu$ is a vector and $\mu$ is a scalar. Instead of maximizing the objective function we negate and minimize it. We take the first derivative of the function with respect to $x$ and equate it to 0. We get:

$$\nabla_x L = -q + \mu x + \nu = 0$$

$$x = \frac{1}{\mu} (q - \nu)$$
We continue taking derivatives and excluding variables. In the end we get a non-trivial system of non-linear equations:

\[
\frac{\partial}{\partial \nu} \frac{1}{2\mu} (q^T \nu - 1) + \frac{\partial}{\partial \nu} \left( \frac{1}{4} \nu^T \nu + 1 \right) + u - \frac{q}{2\mu} + \frac{\mu \nu}{2} = 0
\]

where

\[
\mu = \sqrt{1 - q^T \nu \frac{1}{2} \nu^T \nu + 2}
\]

and

\[
\frac{\partial \mu}{\partial \nu} = \frac{1}{2\mu} - \frac{q(\frac{1}{2} \nu^T \nu + 2) - \nu(1 - q^T \nu)}{\frac{1}{2} \nu^T \nu + 2}
\]

This system of non-linear equations of the fifth degree doesn’t have an analytical solution in general form, so approximate methods come into play. Since this equation has to be solved on each step of the TA, the process of finding the solution, i.e. an extremum has to be extremely fast. If we would find such a vector \( x \), a threshold value would increase faster and TA would stop earlier. The solution to this problem can serve for a considerable speed-up of TA and can be explored in the future work.

Another possible performance improvement could be achieved by adding an additional artificial column that represents lengths of vectors sorted in descending order. However, in our work we do not use this heuristics and stick to the above sketched implementation.

TA’s performance is by far not the best one, however it is a good example of anytime algorithm. We discuss the performance of TA later in Chapter 4.

### 2.4 Locality-Sensitive Hashing

Locality-sensitive hashing (LSH) [AI08] is a technique that allows to quickly find similar points in a multidimensional space. In our scope we use this algorithm in order to find the candidate vectors for the \( k \) most similar patterns based on the factorization data. This approach belongs to probabilistic algorithms, i.e. it doesn’t guarantee to find the complete result, but outputs the correct result with a high probability. By tuning the parameters one can get an output as precise, as one wishes.

The main idea of LSH is to perform random projections of the data and thus map it to a lower-dimensional space [KG09]. By random projection we mean a projection of the data on a random vector or hyperplane. After a random projection all vectors are grouped in buckets. Next, the points that lie close-by after every random projection are grouped into the same buckets. Now, for finding similar points to a particular point we have to examine only those points that reside in the same bucket for all random projections.

There are different similarity functions and hash functions respectively [AI08]. For any hash function to be considered locality-sensitive, two requirements are to be met. The first is the fact that adjacent points stay close after the projection [SC08]. For any two points \( p \) and
2.4 Locality-Sensitive Hashing

Figure 2.1: Dividing space into two subspaces by a random hyperplane $h$

$q$ in $\mathbb{R}^d$, that lie close to each other (the distance is smaller than some $R_1$), there is a high probability $P_1$ that they stay close after projection and fall into the same bucket:

$$\text{distance}(p - q) \leq R_1 \Rightarrow P_H[h(p) = h(q)] \geq P_1$$

The second requirement is: for any two points $p$ and $q$ in $\mathbb{R}^d$ that lie far apart from each other, there is a low probability $P_2$, that these two points fall into the same bucket:

$$\text{distance}(p - q) \geq cR_1 = R_2 \Rightarrow P_H[h(p) = h(q)] \geq P_2$$

For LSH to work, $P_1$ has to be larger than $P_2$, if $R_1 > R_2$:

$$P_1 > P_2$$

distance$(p - q)$ can be the $L_2$ vector norm, cosine similarity or any other metric.

LSH can be used for finding similar vectors using cosine similarity, but not directly to find large inner products to the query vector. In order to find an inner product of two vectors one has additionally to multiply the cosine similarity by the lengths of these vectors. For cosine similarity, a particular LSH function has to be used. This LSH function should preserve the angle between two vectors after the projection [Cha02]. For cosine similarity we pick a random vector of unit-length $r \in \mathbb{R}^d$ and define the hash function as:

$$h_r(p) = \begin{cases} 1 & : rp^T \geq 0 \\ 0 & : rp^T < 0 \end{cases}$$

This hash function $h_r(p)$ can also be seen as the one partitioning the space into two half-spaces by a randomly chosen hyperplane. In Figure 2.1, a random hyperplane $h$ unluckily separates adjacent points $B$ and $C$, but close points $A$ and $B$ reside in the same subspace. For any two vectors $p$ and $v \in \mathbb{R}^d$ holds: $Pr[h_r(p) = h_r(v)] = 1 - \frac{\angle(x,v)}{\pi}$. This equation means, that the probability of separating two vectors with a random hyperplane is directly proportional to the angle between them. Ravichandran et al. has shown [RPH05] that this LSH family preserves the angle between the vectors after a random projection.
We set \( k' \) as a number of random projections. After applying \( k' \) random projections the real data points are transformed into binary strings of length \( k' \) each. The most popular distance between any two binary strings is the so called Hamming distance and equals to the number of coordinates, where these two binary strings differ. Now, we have to group these binary strings in a number of buckets. For this purpose the second hash function is applied on top of the binary representations. We use a simple hash function, which returns a projection of the input point on a random coordinate, i.e. \( h'_i(p) = p_i \). Now, the probability of collision \( \Pr[h'_i(p) = h'_i(v)] \) is equal to the fraction of coordinates on which \( p \) and \( v \) agree. Therefore, this hash function family is locality-sensitive and we can apply it for finding similar vectors.

If the data is normalized, Euclidean distance can be transformed to cosine similarity [SDLT11]. Finding the maximal cosine similarity corresponds to finding the smallest Euclidean distance and vice versa. The Euclidean distance between unit-length vectors \( h_i \) and \( h_j \) can be expressed in terms of cosine between those two vectors:

\[
\|h_i - h_j\|^2 = \|h_i\|^2 + \|h_j\|^2 - 2\|h_i\|\|h_j\| \cos \alpha(h_i, h_j) = 2 - 2\cos \alpha(h_i, h_j)
\]

For Euclidean distance as metric another LSH function is used, in particular a projection on a random vector, i.e. a dot product with a random vector. The locality-sensitive hash function for Euclidean distance is defined as follows:

\[
h^{x,h}(p) = \left\lfloor \frac{xp^T + b}{w} \right\rfloor
\]

where \( p \) is a query vector, \( x \) is a random vector with coordinates, selected from a Gaussian distribution \( \mathcal{N}(0, 1) \), \( \lfloor \cdot \rfloor \) is a flooring operation, \( w \) is the width of buckets and \( b \) is a random variable, uniformly distributed on \((0, w]\). Due to dot product linearity the difference between hashed points has a distribution proportional to the difference between the original points.

In order to overcome the possibility of a certain projection to separate adjacent points we perform \( \text{ProjCount} \) different projections (i.e. \( \text{ProjCount} \) dot products) in parallel. Thus, the difference between \( P_1 \) and \( P_2 \) is enhanced and the probability for distant points to fall into the same quantization bin in all projections becomes \( (P_1/P_2)^{\text{ProjCount}} > (P_1/P_2) \). After applying projections every point in \( d \)-dimensional space becomes a point in \( \text{ProjCount} \)-dimensional space. In each of \( \text{ProjCount} \) dimensions we divide the space into buckets of the length \( w \). If two points get into the same bin in all \( \text{ProjCount} \) dimensions, they are candidates for being similar.

The choice of parameter \( \text{ProjCount} \) is not straightforward. If \( \text{ProjCount} \) is large, then there will be many projections, where even close neighbors land in different buckets. To eliminate this problem we keep \( \text{ProjCount} \) relatively small, but repeat the above procedure \( L \) times. Then it is unlikely that true neighbors will be unluckily separated in all \( L \) cases of applying \( \text{ProjCount} \) projections. Intuitively, for large values of \( \text{ProjCount} \) the probability of collision for distant, as well as for close points, decreases. These probabilities are \( P_1^{\text{ProjCount}} \) and \( P_2^{\text{ProjCount}} \) respectively. A good choice of \( \text{ProjCount} \) in our case was between 25 and 40. At the same time \( L \) should be large enough in order to ensure that close points get into the same
2.4 Locality-Sensitive Hashing

![Figure 2.2: Projection on a random vector](image)

**Algorithm 4** SimLSHHyp

1. **input:** $p$, $H$, $k$, $ProjCount$, $coordCount$, $L$
2. **result** ← empty set
3. **for all** $i$ in $[0, L-1]$ **do**
   4. initialize $hash_i(ProjCount)$
   5. initialize randomCoordinates($coordCount$)
   6. build $hashTable_i(H)$ using $hash_i(ProjCount)$, randomCoordinates($coordCount$)
   7. **result** ← **result** ∪ $hashTable_i[hash_i(p)]$
4. **end for**
5. **return** **result**
6. {Now we can compute distances using naive approach}

buckets at least once. Thus, by increasing $L$, we can find the true nearest neighbors with an arbitrary large probability.

Now, let’s consider the parameter $w$, the width of buckets. By increasing the width $w$ we enlarge the number of points falling into a bucket. This would force us to check more candidate points for similarity with a queried one within a single bucket. For instance, if we would double $w$ in Figure 2.2, all three points would reside in one bucket. From this point of view choosing $w$ is a trade-off between a large hash table with small entries and a small hash table with longer time to process candidates for neighbors.

All these parameters are empirically tuned during the experiments on particular data. In our experiments we compare two LSH functions, projection on a random vector and projection on a random hyperplane. One possible way of avoiding parameters tuning is building an LSH forest [BCG05].

**Implementation** We implemented two versions of LSH algorithm for solving Similarity top-$k$ problem using different locality sensitive hashing function families: SimLSHVector and SimLSHHyp.

A projection on a random hyperplane is used as LSH function in Algorithm 4. The hyperplanes are generated from normal distribution with mean 0.0 and standard deviation 1.0 using Mersenne Twister pseudo-random number generator [MN98]. The number of applied projections is constant for every hash table and is set by the parameter $ProjCount$. Final
hashed values are stored in bitset data structures. These bitsets are mapped to sets of corresponding pattern ids using a tree map data structure. Usually, a hash table is used instead [SC08]. We use a map, because in our case it leads to much lower memory usage and better performance. Regardless of this implementation issue, we still stick to the original terminology and call these map a "hash table" throughout this thesis. Such hash table is constructed \( L \) times using different random projections. \( L \) is given beforehand as an input parameter. For finding neighbors we traverse the buckets of vector \( p \) in all \( L \) hash tables. Finally, the naive approach is used to find top-\( k \) vectors among those in the result set. The point is that the result set comprises much less vectors than the original \( H \) and the run-time decreases drastically.

In our second implementation SimLSHVector we use projection on a random vector as LSH function. In SimLSHVector the following parameters are specified as input: \( ProjCount \) is the number of random vectors we project our data on, \( w \) is the width of buckets in hash tables and \( L \) is the number of hash tables to be constructed.

Random vectors for projections are generated using normal distribution with mean 0.0 and standard deviation 1.0 using Mersenne Twister pseudo-random number generator. After having projected a vector from \( H \) on \( ProjCount \) random vectors we get \( ProjCount \) real numbers. For the sake of fast comparison of hashed vectors and not storing \( ProjCount \) real numbers for each of the initial vectors we use another hash function on top of \( ProjCount \) real numbers. This hash function has a form of \( \sum_{i=0}^{ProjCount} a_i \cdot r_i \), where \( r_i \) are the \( ProjCount \) real values we get after projections and \( a_i \) are integer coefficients uniformly distributed over the range \([0, 255]\). To decrease the number of collisions we use two such hash functions in parallel. The vectors with the same sum values (bucket ids) get into the same buckets and are considered to be neighbors. As in SimLSHHyp we collect all the neighbors of the query vector \( p \) from all hash tables in in the result set. Then the naive algorithm is used to find and output top-\( k \) elements within the result set. As in the previous implementation, in order to keep the memory consumption low we use a tree map data structure that maps bucket ids to sets of patterns ids.

The main disadvantage of LSH-based solutions is its large number of parameters to be set beforehand. However, we will show that for Similarity top-\( k \) problem SimLSHHyp and SimLSHVector are the best solutions. We compare SimLSHHyp and SimLSHVector with other algorithms in Section 4.2.
3 Large Entries in a Matrix Product

3.1 Problem Definition

Finding large inner products, i.e. large results in the matrix product fast is a challenging problem. By large we mean all results larger than some predefined threshold. A naive multiplication of two big matrices is not an option because of possibly large matrix dimensionalities. In Chapter 2 we have shown how one can search for the large inner products of a certain query vector within a single matrix. Large inner products in that case corresponded to the most similar vectors. In this chapter we demonstrate how one can find large inner products not for a single query vector but for a large number of vectors. Note that every entry $v_{ij}$ of the resulting multiplication matrix $V$ is calculated as an inner product of the vector-row $w_i$ from the matrix $W$ and the vector-column $h_j$ from the matrix $H$.

In the subject-object-pattern setting we discussed in Section 2.1 of the previous chapter, large entries in the resulting product matrix $V$ would refer to those facts that are true with a high probability. The larger the value of entry is, the more confident we are about a certain fact.

The naive matrix multiplication of an $(n \times r)$ and an $(m \times r)$ matrices has a runtime complexity of $O(nrm)$. In practice, $r$ is relatively small and can be considered to be constant, but it still results in quadratic runtime cost with respect to the dimensionalities of matrices. Figure 3.1 shows the results of our experiment for $x = m = n$ and $r = 200$ respectively on a laptop with a 1.3 GHz processor and 4 GB of main memory. The $X$-axis shows the parameter $x$, while the $Y$-axis displays the runtime of matrix multiplication measured in milliseconds. Due to the shown dependence a multiplication of matrices $(10^6 \times 200) \ast (200 \times 10^6)$ would take approximately 127 days. Such a runtime is not acceptable for the systems, where the response time is crucial.

We formalize two problems of finding large inner products within a matrix product result.

**Threshold matrix multiplication problem:** Given two matrices, an $(n \times r)$ matrix of patterns $H$ and an $(m \times r)$ matrix of subject-object pairs $W$ and $\theta \in \mathbb{R}$, find all vectors pairs $(w_i, h_j) \in W \times H$ such that $w_i h_j^T > \theta$.

**Top-k matrix multiplication problem:** Given two matrices, an $(n \times r)$ matrix of patterns $H$ and an $(m \times r)$ matrix of subject-object pairs $W$ and $\theta \in \mathbb{R}$, find all vectors pairs $(w_i, h_j) \in W \times H$ such that $(w_i, h_j) \in \text{top}_k(w_i h_j^T)$.

We mainly focus on the Threshold matrix multiplication problem, i.e. finding all entries larger than a certain threshold $\theta$. 

15
3.2 Naive Matrix Multiplication

The most straightforward way to find large entries in the matrix multiplication results is to calculate inner product of every row of the first matrix \( W \) and every column of the right matrix \( H \) and return the values larger than a certain predefined threshold \( \theta \).

For the sake of simplicity we denote naive algorithm from now on by NAÏVE. We briefly sketch the NAÏVE algorithm for matrix multiplication in Algorithm 5.

**Algorithm 5 NAÏVE**

1: input: \( W, H, \theta \)
2: for each row \( i \) in \( W \) do
3:   for each column \( j \) in \( H \) do
4:     result \( \leftarrow w_i h_j^T \)
5:     if result \( > \theta \) then
6:       output (result, \( i, j \))
7:   end if
8: end for
9: end for

**Implementation** As in the one-sided version, we store matrices as vectors of vectors. In NAÏVE, as in all other algorithms, we implement the output as a sorted set.

As argued before, NAÏVE takes unacceptably much time. Nevertheless, it should be mentioned that the naive way has also its advantages over other possible approaches. NAÏVE is easy to implement and can be parallelized without much effort. For instance, \( W \) can be partitioned
3.3 SPEEDY Solution

into several pieces, where each thread is assigned to one single partition. Then each of the
threads multiplies his partition with the $H$, i.e. executes the naive algorithm on its partition.
However, even such parallelization can result in too long runtime.

Another important advantage of NAÏVE is the fact that NAÏVE doesn’t require any parameters
to be set beforehand and works equally good for different data distributions. We will see
further in this chapter that this is not the case for other algorithms.

### 3.3 SPEEDY Solution

As argued before, the naive solution is not a good choice if one cares about execution time.
However, one can try to improve it by exploiting some data characteristics.

If one would examine the length distributions of vectors from both matrices, one would see in
our case a very skewed length distributions over all vectors from both matrices. On the length
histogram in Figure 3.2a one can see that there is a relatively small amount of vectors of a
large length in the matrix $W$. Please, keep in mind that the vertical $Y$-axis has a logarithmic
scale. Every pile shows a number of vectors having the length between two adjacent points
on the $X$-axis. One can see from Figure 3.2a that there are only approximately 40 vectors
with length larger than 2 in the subject-object matrix $W$.

The length histogram of the right matrix $H$ shown in Figure 3.2b looks different, but the
distribution is still rather skewed. The lengths of vectors in the right matrix $H$ are in average
larger than those from the left matrix $W$. Still, the number of vectors of length larger than 2.5
is smaller than 1000 with the overall number of vectors greater than 700,000. Our advanced
algorithm SPEEDY is designed for matrices having skewed distribution and is sketched in
Algorithm 6.

Exploiting the fact that the inner product of any two vectors $a$ and $b$ is related to the angle
between these two vectors, we can accelerate the NAÏVE approach. Consider the following

![Figure 3.2: Vector lengths distribution](image)

(a) Matrix W  
(b) Matrix H
Algorithm 6 SPEEDY

1: input: \( W, H, \theta \)
2: \( W' \leftarrow \text{sortLength}_{>}(W) \)
3: \( H' \leftarrow \text{sortLength}_{>}(H) \)
4: for each \( w_i \) in \( W' \) do
5: \( \text{if } ||W'[w_i]|| * ||H'[0]|| < \theta \text{ then break} \)
6: \( \text{id}_w \leftarrow W'[w_i].\text{id} \)
7: for each \( h_j \) in \( H' \) do
8: \( \text{if } ||W'[w_i]|| * ||H'[h_j]|| < \theta \text{ then break} \)
9: \( \text{id}_h \leftarrow H'[h_j].\text{id} \)
10: \( \text{if } w_{id_w} h_{id_h}^T > \theta \text{ then output } (w_{id_w}, h_{id_h}, id_w, id_h) \)
11: \end for
12: end for

equivalence:

\[
ab^T = ||a|| ||b|| \cos \alpha
\]

Since the cosine values are defined in the range of \([-1, 1]\), we have that \( ab^T \leq ||a|| ||b|| \). Therefore, for the search for vector pairs with an inner product larger than \( \theta \) we can skip all those vectors having the product of lengths smaller than \( \theta \): \( ||a|| ||b|| < \theta \). These filtered out vectors can’t contribute to the result set anyway, since the maximal cosine value can be at most 1 and even in that case the inner product would be for sure bellow the threshold \( \theta \).

Thus, one can speedup the naive algorithm by multiplying the matrices starting from vectors with the largest lengths and proceed in descending order of the vector length. We stop as soon as the product of the vector lengths becomes smaller than \( \theta \). Another heuristic we used allows to further decrease the number of vectors we need to process. We compute an inner product between each vector from the matrices we process with the longest vector from another matrix. As soon as this inner product becomes smaller than the threshold \( \theta \) we can stop, because there are no more candidates left to contribute to the result.

Implementation  

The vectors length information for every matrix is precomputed and stored in an additional vector. These vectors are sorted by the length value in descending order. We access this length vector while checking the stopping condition of heuristics described in Section 3.3.

Our experiments in Section 4.3 indicate that exploiting data features can be very beneficial. SPEEDY improves the execution time by several orders of magnitude over the naive approach, if the length distribution of vectors is skewed. However, it is useless to apply SPEEDY on normalized data, where all vectors have the length of 1.
3.4 Solution based on the Threshold Algorithm

Algorithm 7 THRESH
1: input: W, H, \( \theta \)
2: \( q \leftarrow \) empty queue
3: sort(columns) in \( W, H \)
4: \( i \leftarrow 0 \)
5: initialize(leftThreshold)
6: initialize(rightThreshold)
7: leftIds \leftarrow \) ids from first row of \( W \)
8: rightIds \leftarrow \) ids from first row of \( H \)
9: \( q \leftarrow \) NaiveMultiply(leftIds, rightIds, \( \theta \))
10: initState(state\_H, H)
11: initState(state\_W, W)
12: while (\( \exists \) nextId in \( W \) and \( \exists \) nextId in \( H \)) do
13: \( \text{left\_query} \leftarrow \) nextId(W, state\_W)
14: \( q \leftarrow q \cup \) ThresholdAlgorithm(left\_query, \( H, \theta \))
15: update(leftThreshold, state\_W)
16: if leftThreshold < \( \theta \) then
17: break
18: end if
19: \( \text{right\_query} \leftarrow \) nextId(H, state\_H)
20: \( q \leftarrow q \cup \) ThresholdAlgorithm(right\_query, \( W, \theta \))
21: update(rightThreshold, state\_H)
22: if rightThreshold < \( \theta \) then
23: break
24: end if
25: end while
26: return \( q \)

3.4 Solution based on the Threshold Algorithm

In Section 2.3 we have shown the threshold algorithm for finding the most similar vectors to a given one in a matrix. Now we apply this algorithm for finding large inner products resulting from the multiplication of two matrices \( W \) and \( H \). As before, by large inner products we mean all pairs of vectors ids with inner products larger than a certain threshold \( \theta \). For the purpose of simplicity and compactness we will from now on denote the threshold algorithm by THRESH.

In Algorithm 7 we sketch the modified threshold algorithm for the fast Threshold matrix multiplication problem. As in the one-sided version we discussed in Section 2.3 we sort all columns in both matrices in descending order. We store these sorted as well as unsorted matrices in the main memory to avoid data access latency. The function initialize() in line 5 initializes the threshold values for both matrices by calculating the inner product of the first rows of the sorted matrices. Initially, these thresholds are equal for both matrices. Further, we always calculate a threshold value by multiplying the corresponding boundary values, i.e. current positions from both matrices. In line 9 \( q \) receives the values larger than
\( \theta \) resulting from the multiplication of the vectors from the both first rows. For that we use function \texttt{NaiveMultiply()} which implements the NAÏVE algorithm taking as input two sets of ids from the right and the left matrix respectively. \texttt{NaiveMultiply()} outputs only those pairs of vectors that have inner products larger than \( \theta \). The function \texttt{initState()} in line 10 initializes the variables storing the current state of the algorithm for choosing the next candidate vector. This process of choosing is implemented in function \texttt{nextId()} and is called in line 12 of Algorithm 7. The function \texttt{nextId()} returns the id of the next candidate vector for a given matrix. The next candidate vector is picked from a column using the same heuristic we applied in TA. However, in this case instead of using query pattern coordinates, we calculate inner product of the current rows from the sorted matrices. This inner product of the current rows corresponds to the current threshold value. In Algorithm 7 the threshold value is calculated using the function \texttt{update()}. We choose the column having the largest product of the current rows values. Intuitively, the chosen column corresponds to the column in a matrix, where the inner product grows the most. This largest product is the largest summand we use to calculate the current threshold value. We pick the next id from a chosen in this way column of the \( W \) and treat it as a query vector in the one-sided threshold algorithm in line 14 for searching in the \( H \) and vice versa. THRESH processes both matrices in such steps in a round-robin manner and merges the results with the output queue \( q \). After each step we update the threshold value using the function \texttt{update()} in line 15 and stop if the calculated threshold falls bellow \( \theta \). In the worst case these steps are repeated for all ids in both matrices.

**Implementation** In the implementation of \texttt{state\( H \)} and \texttt{state\( W \)} in line 10 of Algorithm 7 a max-heap is maintained. The heap is updated after every step of the algorithm. The largest value of the heap is replaced by a new value calculated based on the next chosen vector. Then, the heap is rebuilt, since this new value is not always the largest one in the updated heap. If the new value updated in the heap is not the maximal value in the heap, the column to traverse for a new candidate changes.

The threshold is calculated as the sum of all elements in the heap. For performance reasons we do not recalculate the whole sum at each step of THRESH, but subtract the old value from and add a new one to the previous threshold value.

The one-sided threshold algorithm in line 14 of Algorithm 7 is a slightly modified version of the one-sided threshold algorithm discussed in Section 2.3. The modified algorithm takes as input a vector from one matrix and treats it as a query vector within the other matrix. In the previous version of the one-sided threshold algorithm a query vector was assumed to come from the queried matrix. Each instance of the two one-sided threshold algorithms maintains a separate set of already seen vectors ids from another matrix.

In Section 4.3 we present the results of conducted experiments using THRESH and compare its performance with other algorithms.
3.5 Optimized Locality sensitive hashing-based matrix multiplication Algorithm (OLGA)

**Algorithm 8 OLGA**

1. **input**: \( W, H, \theta, \text{ProjCount}, w, L \)
2. \( q \leftarrow \) empty queue
3. **for all** \( i \) in \([0, L-1]\) **do**
   4. initialize \( \text{hash}_i(\text{ProjCount}) \)
   5. \( \text{hashTable}_W^i \leftarrow \) empty map
   6. \( \text{hashTable}_H^i \leftarrow \) empty map
   7. **for all** \( w_i \) in \( W \) **do**
      8. \( \text{bucket}_i = \text{hash}_i(w_i, w) \)
      9. \( \text{hashTable}_W^i[\text{bucket}_i].\text{add}(w_i) \)
   10. **end for**
   11. **for all** \( h_j \) in \( H \) **do**
      12. \( \text{bucket}_i = \text{hash}_i(h_j, w) \)
      13. \( \text{hashTable}_H^i[\text{bucket}_i].\text{add}(h_j) \)
   14. **end for**
   15. **for all** \( \text{bucket}_i \) in \( \text{keys}(\text{hashTable}_W^i) \) **do**
      16. **if** \( \text{hashTable}_H^i\text{.contains}(\text{bucket}_i) \) **then**
      17. \( q \leftarrow q \cup \text{NaiveMultiply}(\text{hashTable}_W^i[\text{bucket}_i], \text{hashTable}_H^i[\text{bucket}_i], \theta) \)
      18. **end if**
   19. **end for**
20. **end for**
21. **return** \( q \)

3.5 Optimized Locality sensitive hashing-based matrix multiplication Algorithm (OLGA)

The general idea of the LSH-based matrix multiplication algorithm is very similar to the hash join of two tables in a database. Both matrices are split into buckets using the same locality-sensitive hash function. Then, only the vectors in the buckets having the same hash values on both sides are multiplied using the NAÏVE approach. In this section we explain OLGA in detail.

In the two-sided matrix multiplication version of LSH-based algorithm OLGA sketched in **Algorithm 8** we use the same \( \text{hash}() \) function as in the SimLSHVector. OLGA uses a projection on a random vector as a locality-sensitive hash function. We will discuss the performance of the algorithms later in **Section 4.3** in more detail.

We normalize vectors from both matrices upfront, in order to find entries in normalized \( W \) and \( H \) with large cosine similarities. Both matrices are hashed using the same hash function in the way we discussed in **Section 2.4**. As in the one-sided version, vectors residing in the same bucket are assumed to be similar. We go through all the buckets from the left hash table and search for a corresponding bucket with the same hash value in the right hash table. If such corresponding bucket is found, the NAIVE matrix multiplication is executed for the sets of vectors contained in these right and left buckets. The results of multiplication larger
than $\theta$ are added to the resulting queue $q$. This procedure is repeated $L$ times on $L$ hash tables in order to account for possible hash function collisions.

As in the one-sided version of the LSH-based algorithm, there are three parameters to be set for OLGA. The choice of $ProjCount$, $w$ and $L$ is very important and affects the runtime of the algorithm as well as the correctness of the delivered results tremendously. Unfortunately, opposed to the one-sided version, multiple experiments for finding the best parameters for the both-sided version of OLGA are very time-consuming. However, we came up with an idea, how one can tune parameters without much overhead in time.

We focus on tuning the parameters $L$ and $w$. As described in Section 2.4, $L$ is a number of hash tables (i.e. repetitions) and $w$ is a length of each bucket in a hash table. We keep $ProjCount$ (the dimensionality of our vectors after projection) constant.

Algorithm 9 OLGARepeat

```plaintext
1: input: $W$, $H$, $\theta$, $ProjCount$, $w$, $maxL$
2: $q \leftarrow$ empty queue
3: for all $i$ in $[0, maxL)$ do
4:   initialize hash$_i(ProjCount)$
5:   build hashTable$_i^W$
6:   build hashTable$_i^H$
7: end for
8: for all $i = 0; i < maxL; i + +$ do
9:   for all bucket _id$_W$ in keys(hashTable$_i^W$) do
10:      if hashTable$_i^H$.contains(bucket _id$_W$) then
11:         $q \leftarrow q \cup$ NaiveMultiply(hashTable$_i^W$[bucket _id$_W$], hashTable$_i^H$[bucket _id$_W$], $\theta$)
12:     output $q$
13: end if
14: end for
15: return $q$
```

Let us consider the way we tune $L$ while keeping all the remaining parameters unchanged in algorithm that we call OLGARepeat. A given maximal number $maxL$ of hash tables is constructed in line 9 of Algorithm 9. We examine the number of hash tables starting from 1 till $maxL$ in line 8 step-by-step, processing one pair of hash tables after another and reporting the cumulative results after each step. In this way we can find such a value of $L$ that after processing $L$ hash tables it is not worth continuing and processing the next one. We stop processing hash tables if there is no considerable gain in result set any more. OLGARepeat is an anytime algorithm and can be stopped at any time of the execution.

Next, we show the way we tune the width of buckets, the parameter $w$. By increasing the width of buckets step by step one can find a value of $w$ which is optimal for OLGA’s performance/accuracy ratio. Good performance/accuracy ratio means that only few unnecessary vector multiplications are done, but the number of the results in comparison with NAÏVE is still high. We call this algorithm OLGAWidt.
3.5 Optimized Locality sensitive hashing-based matrix multiplication Algorithm (OLGA)

Algorithm 10 OLGAWidthOrRepeat

1: input: $W$, $H$, $\theta$, $ProjCount$, $minW$, $maxL$, $\epsilon$, $maxExp$
2: $q$ ← empty queue
3: initLSHAlgorithm($W$, $H$, $ProjCount$, $minW$, $maxL$)
4: init $qWidth$
5: init $qRepeat$
6: $deltaWidth$ ← $size(qWidth)$
7: $deltaRepeat$ ← $size(qRepeat)$
8: $q$ ← $qWidth$ $\cup$ $qRepeat$
9: while ($deltaWidth$ $> \epsilon$ and $deltaRepeat$ $> \epsilon$) do
10: oldSize ← $size(q)$
11: if ($deltaRepeat$ $\geq$ $deltaWidth$) then
12: examine next repetition, update $qRepeat$
13: $q$ ← $q$ $\cup$ $qRepeat$
14: $deltaRepeat$ ← $size(q)$ $-$ $size(qRepeat)$
15: end if
16: if ($deltaRepeat$ $<$ $deltaWidth$) then
17: merge buckets, update $qWidth$
18: $q$ ← $q$ $\cup$ $qWidth$
19: $deltaWidth$ ← $size(q)$ $-$ $size(qWidth)$
20: end if
21: end while
22: return $q$

Changing the width of buckets once they have been built requires the rebuilding of hash tables and takes a lot of time. To avoid that, one can examine not only single buckets from both hash tables, but also their corresponding neighboring buckets. Observing additionally an adjacent bucket would correspond to making the bucket twice as large as the initial one. Certainly, one has to increase the width of buckets for both matrices respectively. Now, if one takes three adjacent buckets in addition to the initial bucket, the width of the resulting bucket becomes four times larger than it was before. In such a way the width of buckets can be easily doubled until a single bucket spans the whole hash table.

Furthermore, one can combine two approaches presented above into one and on each step either examine a new hash table or double the length of buckets in one of the hash tables. The choice what exactly to do can be done randomly or using some heuristics. We call this algorithm OLGAWidthOrRepeat.

We make a decision whether to double the width or to examine the next hash table depending on the contribution to the result set the last time this particular choice was taken. In this sense OLGAWidthOrRepeat is a greedy approach. We initialize OLGA algorithm using the $ProjCount$ parameter, the minimal bucket width $minW$ and the maximal number of repetitions $maxL$ in line 3 of Algorithm 10. We use an input parameter $\epsilon$ for comparing with the contribution of the previous step to the resulting set. When a resulting set grows less than by $\epsilon$ elements, we stop. In order to make the choice we calculate and store the previous gain of the resulting set for both of the possible choices in variables $deltaRepeat$
3 Large Entries in a Matrix Product

and \( \text{deltaWidth} \). Depending on these variables a particular action is taken in line 11. For instance, if \( \text{deltaRepeat} \) is larger than \( \text{deltaWidth} \), the next pair of hash tables is processed. After processing the next pair of hash tables the variable \( \text{deltaRepeat} \) is updated. In order to initialize the variables \( \text{deltaRepeat} \) and \( \text{deltaWidth} \) we perform both actions once at the beginning in line 4. Please, note that the length of buckets is doubled in one randomly chosen hash table, not in all.

**OLGA for non-normalized data**  As stated in Section 2.4 OLGA can’t be used directly to find large entries from a matrix multiplication result using the original data, only using normalized data. However, OLGA can complement other approaches in order to find large entries from a matrix product. One way of accelerating SPEEDY is to make the threshold \( \theta \) as large as possible. We will discuss the behavior of SPEEDY with respect to different \( \theta \) later in Section 4.3. Intuitively, SPEEDY stops earlier with the larger values of threshold. We came up with an approach how to speedup SPEEDY using OLGA. The general idea is first to run OLGA on normalized data and output all vectors with an inner product larger a certain constant \( \delta \). Suppose, the parameters of OLGA have been tuned in such a way that OLGA outputs all the pairs of vectors with cosine similarity larger than \( \delta \). Next, we run SPEEDY on non-normalized data but with the new threshold value \( \theta' = \frac{\theta}{\delta} \). This new value is potentially larger than \( \theta \). As already mentioned, a bigger threshold \( \theta' \) has an effect that SPEEDY stops earlier. We accelerate SPEEDY using OLGA and not vice versa, because larger \( \theta \) in SPEEDY has much larger impact on a runtime than in OLGA. We call this algorithm OLGA+SPEEDY.

After obtaining the results of OLGA and SPEEDY we merge the results into one set and output it. While merging the result sets one needs to take into account the fact that inner product values of OLGA were calculated on normalized data. This requires an additional multiplication of the OLGA’s result values by lengths of two corresponding vectors in order to get the correct inner product values.

**Optimization**  Contrary to the one-sided version of LSH, OLGA initializes the hash tables carefully. For performance reasons we decide to keep only those pairs of hash tables (right and left ones) having a cost of buckets multiplication in a certain range \([a, b]\). By the cost of a hash table pair we mean a sum of products of buckets costs, namely the cardinalities of corresponding buckets from both hash tables. We go through the left hash table and find corresponding buckets in the right hash table. The bucket cardinalities of corresponding buckets are multiplied and summed over the whole hash table pair. We observed that a very small cost of the hash tables leads to no result output. However, a pair of hash tables with a large cost is very time-consuming to process, though such a pair outputs many results. We found empirically the values of \([a, b]\) for our data, however they may have to be changed under other conditions. If the overall cost of a hash table pair is not in the predefined range \([a, b]\), we discard these hash tables and generate a new pair. We repeat the process until we get \( L \) hash table pairs, where each of them satisfies the cost condition.

Another way of accelerating OLGA is to use SPEEDY instead of NAIVE for multiplying vectors between two corresponding buckets. In order to use SPEEDY every bucket is sorted in advance...
by vector length in descending order. We call this algorithm OLGA++.

**Implementation**  OLGA expects normalized data as input. We normalize vectors from both matrices before the initialization of hash tables. Unless otherwise stated, we refer in this subsection to all versions of this algorithm as OLGA.

In contrary to the algorithm described by Slaney et al. [SC08] the variable `hashTable` in line 5 of Algorithm 8 is an instance of `std::map`. As in the one-sided version, we use a map data structure instead of a hash table, because of the much smaller main memory footprint. Our analysis has shown that usually only a few buckets in a hash table structure contain non-empty set of vectors. The exhaustive main memory usage for storing empty buckets was a reason for out-of-memory error during the execution. This unacceptable situation forced us to use a map structure. We use a red-black tree-based implementation for our map data structure. The insertion and search operations take $O(\log n)$ each for this data structure. However, for small values of $n$ these operations on a map are only slightly slower than on a hash table.

For efficiency reasons we maintain and look-up a set of already multiplied pairs of vectors while execution in all versions of OLGA. We limit the size of this set to a constant to prevent excessive memory consumption.

Contrary to THRESH, OLGA can be easily parallelized by distributing the pairs of hash tables among multiple threads. The parallelization of THRESH is not so straightforward as of OLGA or NAÏVE. However, a tremendous disadvantage of OLGA over NAÏVE, SPEEDY and THRESH is its complicated process of choosing the appropriate parameters. Tuning of these input parameters is tricky and may take much time.

In Section 4.3 we present our experimental results using different versions of OLGA. We also show the performance overhead of OLGA comparing to NAÏVE and other approaches.
4 Experimental Results

4.1 Experimental Setting

For our experiments we use subject-object-pattern triples extracted from the New York Times text. The matrix \( V \) representing a subject-object-pattern co-occurrence was factorized into matrices \( W \) and \( H \). Matrix \( W \) represents latent factors of subject-object pairs. Matrix \( H \) refers to latent factors of patterns. We use two different datasets of \( W \) and \( H \). The dataset NYT-SVD was received by factorization of original matrix \( V \) using singular value decomposition. Another dataset NYT-NMF was received applying non-negative matrix factorization on \( V \) with the condition that all latent factors in both \( W \) and \( H \) are non-negative. The matrix \( W \) \((771,611 \times 200)\) consists of 771,611 subject-object pairs disambiguated against Yago [SKW07]. The matrix \( H \) \((132,209 \times 200)\) consists of 132,209 patterns.

Unless otherwise stated, in our experiments for finding large results in \( WH^T \) we use sample data, 10,000 vectors from each matrix. We choose this sample data uniformly at random from every of matrices \( W \) and \( H \). The data chosen from dataset NYT-SVD is called NYT-SVD-10k and data chosen from dataset NYT-NMF is called NYT-NMF-10k. If this data is preliminarily normalized we call it NYT-SVD-10k-norm and NYT-NMF-10k-norm respectively. We use the original matrices in our final experiments. We perform our experiments using sample data on a laptop with an 1.3 GHz Intel Core2 Duo processor and 4 GB of main memory. We conduct the experiments on the original data using a machine with a 2.4 GHz Intel Xeon E5530 and 48 GB of main memory.

4.2 Experimental Results for Similar Pattern Search

In this section we compare all algorithms designed for efficient similar pattern search discussed in Chapter 2.

Threshold Algorithm-based Solution  
In Figure 4.1 we compare the execution times measured in milliseconds for different versions of TA. Note, that all Y-axes have logarithmic scales. We compare execution times of NaiveThresh and NaiveTopk with the TAtthresh and TAttopk run on the NYT-SVD as well as NYT-NMF datasets. We are looking for the similar vectors to the vector “yesterday” from \( H \) in Figure 4.1a and Figure 4.1b. In Figure 4.1c and Figure 4.1d we search for similar patterns to the vector “wife of”. We have picked these two patterns because of their different properties. While “yesterday” has many close vectors, i.e. vectors with a large similarity, “wife of” lies far apart from the other vectors.
4 Experimental Results

Our experiments suggest that TA is uncompetitive and in most of the cases performs worse than NaiveThresh and NaiveTopk. The reason is that TA makes many steps in order to ensure that all possible vectors that can contribute to the result are found. TA also has to maintain larger and more complicated structures in the main memory while running. The only case, where TA outperforms NaiveThresh and NaiveTopk, is its execution on NYT-NMF data for a “good” vector “yesterday”. TA finds almost all the results after a few steps, but proceeds further to ensure there are no more candidate vectors for the result set. TAThresh shows the same performance as TAtopK for the pattern “wife of”, because they both make too many steps during execution. The reason for this performance gap between NYT-SVD and NYT-NMF data lies in the fact that TA requires a skewed distribution of the data. NYT-NMF data has many entries close to 0 because of the non-negative condition and therefore, a skewed distribution of the vector coordinates values.

In order to justify this assumption we generated random matrices $(132{,}209 \times 200)$ using Dirichlet distribution, parametrized by different values of $\alpha$. The concentration parameter $\alpha \in [0,1]$ regulates the skewness of the generated data. The larger the value of $\alpha$, the more evenly is the resulting distribution (the generated data tends towards the uniform distribution).

In Figure 4.2 we show the execution time of TAtopK and NaiveTopk with respect to different values of parameter $\alpha$ for a random vector. One can see that NaiveTopk, as expected, doesn’t depend on the data distribution at all. However, TA outperforms NaiveTopk only with
4.2 Experimental Results for Similar Pattern Search

Figure 4.2: Performance of TAtopK and NaiveTopk on skewed data

\( \alpha < 0.05 \). The same trend we observed for NaiveThresh and TAThresh.

To conclude, we don’t recommend using threshold algorithm-based solutions for searching for similar vectors, unless one finds a better way of calculating the threshold value such that it decreases fast and the algorithm stops earlier. However, TA has a very good anytime algorithm behavior and provides most of the results already after a few steps.

**LSH-based Solution** LSH-based solutions SimLSHVector and SimLSHHyp yield a breakthrough in run-time over the naive approach. Depending on the input parameters SimLSHVector and SimLSHHyp perform up to a factor of 200 faster than NaiveThresh and NaiveTopk.

However, a big challenge is to choose the input parameters for SimLSHVector and SimLSHHyp in a way to achieve the minimal execution time. LSH-based solutions belong to the class of Monte Carlo algorithms. The correctness of the result can be improved by running LSH-based algorithms multiple times. Therefore, the choice of input parameters influences both runtime and correctness of the result. We measure the correctness of SimLSHVector and SimLSHHyp by comparing their results to the result calculated by the naive approach. In the case of Similarity top-\( k \) problem, a recall is defined as a percentage of matching results of SimLSHVector or SimLSHHyp with the correct results. In Similarity threshold problem case we define precision as a percentage of matching results of SimLSHVector or SimLSHHyp with the results of NaiveThresh. A recall is in this case 100\%, since all returned by SimLSHVector or SimLSHHyp similarities are larger than \( \theta \). A purpose of parameter tuning is to find such a combination of input parameters for SimLSHVector or SimLSHHyp that minimizes the execution time for some fixed recall or precision.

In Figure 4.3 we show the recall/precision and the execution time for different combination of input parameters for SimLSHVector, SimLSHHyp and the naive approach. We used the matrix \( \mathbf{H} \) from the NYT-NMF-norm data set. The X-axis displays an execution time in milliseconds and has a logarithmic scale. The X-axis shows a precision or recall with respect to NaiveTopk and NaiveThresh results respectively. It should be mentioned that in the case of
4 Experimental Results

LSH-based algorithms we preliminarily normalize the input data. Therefore, NYT-SVD-norm dataset has the same behavior as NYT-NMF-norm, since after the normalization of vectors the distribution of NYT-NMF-norm data becomes uniform. Figure 4.3a and Figure 4.3c as well as Figure 4.3b and Figure 4.3d show that a query vector \( p \) has an effect on input parameters. For different patterns we observe different recall or precision ratio using the same combination of input parameters. The reason for this difference is that “wife of” has only two close vectors with \( \theta > 0.7 \). Blue points in all four figures show the execution time of NaiveTopk and NaiveThresh (with 100% precision). Clearly, one would prefer such a combination of parameters that correspond to the lowest points on the diagrams. Such points have the smallest possible execution time with a certain precision or recall.

LSH-based algorithms SimLSHVector or SimLSHHyp show an excellent performance in solving Similarity threshold problem as well as Similarity top-k problem. Therefore, we would recommend SimLSHVector or SimLSHHyp as a solution for searching similar vectors.

The disadvantage of SimLSHVector or SimLSHHyp is the expensive process of tuning the input parameters. In our settings for NYT-NMF-norm data they are \( L = 30, w = 3.5, ProjCount = 25 \) for SimLSHVector and \( L = 18, ProjCount = 60, CoordCount = 35 \). With these parameters, 75% of precision/recall was achieved.
4.3 Experiments for Large Entries Search

In this section we compare all algorithms previously discussed in Chapter 3 that are designed for solving Threshold matrix multiplication problem. If not otherwise stated, in order to save time we conducted our experiments not on the original matrices but on 10,000 random vectors uniformly chosen from each of \( W \) and \( H \). This corresponds to the product matrix \( V \) with \( 10^8 \) entries.

**SPEEDY**  
SPEEDY takes advantage of skewed length distributions in \( W \) and \( H \) and outperforms NAÏVE up to a factor of 4,000. In Figure 4.4 we compare the execution times of NAÏVE and SPEEDY for different values of \( \theta \). We show the performance on two different datasets - NYT-SVD-10k and NYT-NMF-10k. \( \theta \) is chosen such that the result set cardinality is 100, 1,000, 10,000 or 100,000 entries. The \( Y \)-axis displays the run-time measured in milliseconds and has a logarithmic scale. As expected, the execution time of SPEEDY grows as \( \theta \) decreases and the output set grows. Obviously, decreasing \( \theta \) would increase the number of candidate vectors contributing to the result set. If \( \theta \) gets too small SPEEDY will have to check all the vectors, and will not have any advantage over NAÏVE. If one looks for a relatively small percentage of the result product matrix, then NAÏVE is the best choice.

A great strength of SPEEDY is that it doesn’t require any input parameters set and consumes a little of main memory. SPEEDY is data-dependent and performs the best on data with a skewed length distribution in both multiplied matrices. SPEEDY can also handle any other data distribution, but in the case of a uniform distribution its runtime converges to the runtime of NAÏVE. Fortunately, our data has a skewed lengths distribution and therefore SPEEDY is the choice of the author.

**THRESH**  
Our experiments in Section 4.2 indicate that TA is not competitive in searching for similar patterns. Contrary to TA, the performance of THRESH compared to NAÏVE is much better. Moreover, THRESH turned out to have a very beneficial behavior as anytime algorithm. Figure 4.5 shows an output size at every step of THRESH. The \( X \)-axis displays the
execution time and the $Y$-axis is the output size of an algorithm after a particular amount of time has passed. For comparison the output of NAIVE is shown, after every $n$ steps. Since NAIVE processes vectors in their original order, the output grows linearly. Contrary to NAIVE, THRESH finds all the results after a few steps. THRESH processes the data further in order to ensure there is no candidates for the result set. Thus, one can pause THRESH after the few first steps and get the result by an order of magnitude faster than using NAIVE. One can stop the execution of THRESH if the result set grows less than for a certain predefined threshold $\delta$.

However, the situation changes when $\theta$ grows. Figure 4.6 shows the results of experiments for THRESH conducted for different data sets and different $\theta$. The threshold $\theta$ is chosen in a way the output result consists of 1000 entries in Figure 4.6a and Figure 4.6b and 100000 entries in Figure 4.6c and Figure 4.6d. As in the previous Figure 4.5, the $X$-axis displays the execution time and the $Y$-axis shows the result size of an algorithm at a particular point of time. Figure 4.6 shows in the case of the output set cardinality 100000 THRESH performs inefficiently and finds all the result entries a way to late comparing to NAIVE. Note that like the one-sided version NaiveThresh, THRESH performs much better on NYT-NMF-10k with a more skewed distribution of the vector coordinates as on NYT-SVD-10k. Thus, the performance of THRESH is data-dependent. Still, THRESH is not suitable for a search with a small $\theta$, i.e. a big set cardinality of the result.

To conclude, we would recommend using THRESH if one expects a small output set and the data has a skewed distribution. Also, one has to stop the execution of THRESH once the result set does not grow for more than a considerable amount of elements.

**OLGA** We compare OLGA, OLGA++ and NAIVE for different data sets and different $\theta$ in Figure 4.7. $\theta$ is chosen such that the result consists of 100, 1000, 10000 or 100000 entries. The $Y$-axis displays the execution times measured in milliseconds for different values of $\theta$. Note that $Y$-axis has a logarithmic scale. We used NYT-SVD-10k-norm dataset in Figure 4.7a and NYT-NMF-10k-norm dataset in Figure 4.7b. The LSH-based solution OLGA doesn’t outperform the NAIVE approach for threshold matrix multiplication problem in a way it did for the Similarity threshold and Similarity top-$k$ problems. OLGA performs the best.
4.3 Experiments for Large Entries Search

(a) NYT-NMF-10k, output size 1 000

(b) NYT-SVD-10k, output size 1 000

(c) NYT-NMF-10k, output size 100 000

(d) NYT-SVD-10k, output size 100 000

Figure 4.6: THRESH for different \( \theta \) and data sets

for small result set cardinality. In this cases, OLGA outperforms NAIVE by a factor of 10. In comparison to SPEEDY, OLGA shows a more stable behavior for increasing output set cardinalities. OLGA++ improves the OLGA very slightly. The reason for the comparatively moderate performance of OLGA, in the author’s opinion, is that though OLGA multiplies only a few vectors from corresponding buckets, the total number of bucket pairs is very large. Also, to achieve a precision of at least 75% one has to repeat the process many times.

On Figure 4.8 we show the result size after each step of OLGAWidthOrRepeat. We start the algorithm with a minimal width \( w = 2.0 \) and one hash table pair. The \( X \)-axis shows the cumulative run-time in milliseconds of all previous steps during the execution. The \( Y \)-axis shows the size of a result set after each step of the algorithm. The experiment was run on NYT-SVD-10k-norm dataset with \( \theta \) corresponding to the output of 1000 entries. Though Figure 4.8 indicates non-efficiency of OLGA comparing to NAIVE it reveals an internal behavior of OLGA. Figure 4.8 indicates that doubling the minimal width of buckets doesn’t yield any significant overhead. On the other hand, to achieve a better precision of the output result one has to check more repetitions.

Next, we consider OLGA+SPEEDY applied to non-normalized data. Figure 4.9 displays the execution time for different approaches for four values of \( \theta \). The values of \( \theta \) correspond to the cases, when the result set consists of 100, 1000, 10 000 or 100 000 entries. Note that \( Y \)-axis has a logarithmic scale and shows the run-time of a particular algorithm in milliseconds. OLGA+SPEEDY outperforms NAIVE but doesn’t beat SPEEDY in the run-time. The main
Experimental Results

Figure 4.7: OLGA, OLGA++ and NAIVE for NYT-SVD-10k-norm and NYT-NMF-10k-norm

reason for that is a comparatively large run-time of OLGA. OLGA+SPEEDY performs better on NYT-NMF-10k data and in the best case outperforms NAIVE by a factor of 4. However, the difference in run-times shrinks when $\theta$ decreases.

OLGA performs by a factor of 10 better than NAIVE. OLGA is beneficial if only a small percentage of the matrix product has to be returned, i.e. $\theta$ is very large. OLGA can be easily parallelized and take advantage of multi-threading. However, OLGA needs a time-consuming process of hash tables initialization. In comparison to other algorithms, OLGA has a relatively high memory consumption. OLGA requires an expensive process of parameters tuning, which has to be repeated if the data size or data distribution changes.

Figure 4.10 shows the performance of the previously discussed algorithms run on the original matrices $W$ ($771\,611 \times 200$) and $H$ ($132\,209 \times 200$) using NYT-NMF data set. On Figure 4.10a we compare NAIVE, SPEEDY and THRESH when the output set comprises 2000 entries. Note, that the Y-axis has a logarithmic scale and displays the execution time in milliseconds. NAIVE takes more than 35 hours to complete the execution. SPEEDY outperforms NAIVE by a factor of 27000 and outputs the result set after 4617 ms. THRESH takes 289902 ms and beats NAIVE by a factor of 500.

Figure 4.10b illustrates the performance of NAIVE, OLGA++ and THRESH in the same context.
4.4 Summary

Figure 4.9: OLGA+SPEEDY, SPEEDY and NAÍVE for NYT-SVD-10k and NYT-NMF-10k

setting as Figure 4.10a, but the data is normalized upfront. The X-axis shows an execution time, measured in milliseconds and has a logarithmic scale. The Y-axis displays the output size in percentages. The output set comprises 300,000 entries. THRESH performs in this setting by a factor of 175 faster than NAÍVE. OLGA++ outputs 6% of the output set and requires for that only 4% of the NAÍVE execution time, which is better than NAÍVE’s performance. In our experiments on the NYT-NMF dataset we used the parameters tuned for the NYT-NMF-10k dataset. Apparently, OLGA++ needs retuning the input parameters for the original input data.

4.4 Summary

In this thesis we conducted multiple experiments using sample data as well as the entire dataset. The full dataset consists of the subject-object matrix \( W \) (771,611 \( \times \) 200) and pattern matrix \( H \) (132,209 \( \times \) 200). Sample data consists of 10,000 vectors randomly chosen from each of these matrices. We used the data extracted from the New Your Times using two different factorization approaches: singular value decomposition and non-negative matrix factorization.

Figure 4.10: Performance of algorithms on the original data (NYT-NMF)
For the matrix multiplication problem we tuned all input parameters on sample data and applied them on the full dataset.

The best choice for efficient similar pattern search is the LSH-based approach. Our experiments indicate that SimLSHHyp and SimLSHVector outperform the naive approach by a factor of 200. The threshold algorithm-based approach TA performs better than the naive approach if the matrices have a skewed distribution of vectors coordinates.

The best algorithm to search fast for large entries in a matrix product is SPEEDY. The experiments on the entire dataset indicate that SPEEDY outperforms NAIVE by a factor of 27 000, decreasing the runtime from 36 hours to 4 617 ms. SPEEDY takes advantage of the skewed vector lengths distribution in both matrices. The threshold algorithm-based approach THRESH outperforms NAIVE by a factor of 500. Our experiments suggest that THRESH has a very good behavior as anytime algorithm and outputs the whole result set after a few steps. The LSH-based approach OLGA performs better than NAIVE, but needs expensive input parameters tuning.
5 Conclusion

Results The main challenge we address in this thesis is the fast search for large entries in a matrix product. In order to solve this problem we designed three new algorithms: SPEEDY, THRESH and OLGA. All three algorithms outperform NAÏVE. Every of this algorithms works best if input data has particular properties. In this section we point out which algorithm suits which data the best.

In our opinion, the best choice is the SPEEDY approach. Our experiments show that this algorithm outperforms NAÏVE by a factor of up to 27 000 on the original data (771 611 subject-object pairs and 132 209 patterns). SPEEDY does not need any input parameters. The weakness of this approach is that SPEEDY requires a skewed distribution of input data and is, therefore, data dependent.

THRESH is also fairly effective and outperforms NAÏVE by a factor of up to 500 on the original data. Moreover, our experiments show that THRESH has an extremely good behavior as an anytime algorithm. THRESH can be applied to any data and does not need any parametrization.

OLGA performs better than NAÏVE by a factor of 10. However, our experiments suggest an uncompetitive performance of the LSH-based OLGA with respect to SPEEDY and THRESH. Opposite to SPEEDY and THRESH, OLGA outputs only a small portion of the full result set. The most probable reason for its comparatively moderate performance is a large number of hash tables that one has to process in order to achieve an acceptable precision. The performance of OLGA is extremely dependent on the input parameters. These parameters are hard to predict for different data or data sizes. We tuned the parameters for 10 000 sample vectors from original matrices. The experiments suggest that these parameters are not optimal for the multiplication of original matrices. The reason we didn’t tune input parameters on the original matrices is a very time-consuming process of parameter tuning. All LSH-based approaches expect normalized data as input, which is not always the case.

Another challenge we addressed in this work is the similar pattern search. Our experiments indicate that the best solution to this problem is the LSH-based approach. We implemented two versions of this approach, namely SimLSHVector and SimLSHHyp. These algorithms outperform the naive algorithm by a factor of up to 200 on our original data (132 209 patterns). Opposite to the LSH-based approach, the solution based on threshold algorithm TA is barely faster than the naive algorithm. We recommend to use SimLSHVector to quickly search for similar patterns.
Future Work  A promising direction to speed-up the threshold algorithm-based approaches is to improve the threshold calculation as discussed in Section 2.3. A better way to calculate a threshold value would cause THRESH to stop earlier and would considerably improve its performance. An alternative approach for optimization of THRESH could be using an artificial sorted column containing vector lengths. The algorithm would first examine the vectors that have large lengths.

Even more interesting would be to take a closer look and explore a joint space of input parameters for OLGA. A well-tuned combination of these parameters has a great potential to decrease OLGA’s run-time. This process of parameter tuning should be automated, if possible.

An absolutely sure way to get a linear performance gain for all discussed algorithms is to use multi-threading. Both SPEEDY and OLGA are comparatively easy to parallelize. However, it is not obvious how to parallelize THRESH.
Bibliography


