San Jose State University

SJSU ScholarWorks

Spring 2015

Cryptanalysis of Classic Ciphers Using Hidden Markov Models

Rohit Vobbilisetty
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

 Part of the Information Security Commons

Cryptanalysis of Classic Ciphers Using Hidden Markov Models

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Rohit Vobbilisetty

May 2015

The Designated Project Committee Approves the Project Titled

Cryptanalysis of Classic Ciphers Using Hidden Markov Models

by

Rohit Vobbilisetty

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2015

Dr. Mark Stamp   Department of Computer Science

Dr. Richard M. Low   Department of Mathematics

Dr. Thomas Austin   Department of Computer Science

ABSTRACT

Cryptanalysis of Classic Ciphers Using Hidden Markov Models

by Rohit Vobbilisetty

Cryptanalysis is the study of identifying weaknesses in the implementation of cryptographic algorithms. This process would improve the complexity of such algorithms, making the system secure.

In this research, we apply Hidden Markov Models (HMMs) to classic cryptanalysis problems. We show that with sufficient ciphertext, an HMM can be used to break a simple substitution cipher. We also show that when limited ciphertext is available, using multiple random restarts for the HMM increases our chance of successful decryption.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

CHAPTER

APPENDIX

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

A Hidden Markov Model (HMM) includes a Markov process with hidden or unobserved states. That is, the states of the Markov process are not directly visible to the observer, although a series of observations (dependent on the states) are visible. The states are related to the observations by discrete probability distributions [23], also called emission probabilities.

The HMM training process can be viewed as a (discrete) hill-climb on the parameter space consisting of state transition probabilities and observation probabilities. Consequently, we are only assured of a local maximum. Also, in general, we only obtain the global maximum when we start sufficiently close– within its "attraction basin". Therefore, if we train multiple HMMs, each with a separate random initialization, we expect to obtain a stronger model, particularly in cases where the available training data is limited.

In this project, we propose to use HMM analysis to cryptanalyze various classic ciphers. We will quantify the effectiveness of multiple random restarts [2] with respect to the amount of data available, and compare it with the results obtained from the Jakobsens [10] algorithm. We also attempt to leverage the processing power of the Graphics Processing Unit [21] by implementing the Baum-Welch algorithm for Hidden Markov Models, and compare the performance with its equivalent single-threaded implementation on CPU.

Initially, we analyze the Simple Substitution Cipher using HMMs with multiple random restarts by varying the data size of ciphertext. This includes the Caesar

cipher and the generic case, where a symbol from the plaintext can map to one of the symbols from the ciphertext assuming there is a one-to-one relationship. We define accuracy as the number of symbols correctly deciphered by this method, and compare accuracies for increasing data sizes. We also compare accuracy with increasing data size, and increasing random restarts.

This paper is organized as follows. In Chapter 2, we provide background information on cryptography, cryptanalysis, various types of ciphers and Hidden Markov Models. Chapter 3 provides the detailed process of performing Hidden Markov Models with random restarts. Chapter 5 outlines the results obtained from our experiments. We finally conclude the report by providing the conclusion in Chapter 6, followed by future work and enhancements in Chapter 7.

CHAPTER 2

Background

In this section we will briefly discuss about Cryptography, Cryptanalysis and finally Simple and Homophonic substitution ciphers. We then cover the basics of Hidden Markov Models and describe the Baum-Welch [23] algorithm in detail. Lastly, we cover the basics of Graphics Processing Unit and the CUDA [16] (Compute Unified Device Architecture) programming language.

2.1 Cryptography

Cryptography is the art of securing communication in the presence of third parties. The main aspects are data confidentiality, data integrity, authentication and non-repudiation.

2.2 Cryptanalysis

Cryptanalysis is the study of analyzing cryptographic algorithms and other systems to uncover the hidden aspects of the systems. This includes identifying weaknesses in the design and implementation of cryptographic algorithms [24]. This process may involve executing many experiments with varying parameters, making it computationally intensive.

In cryptography, a substitution cipher is the method of substituting each unit of the plain text with its associated unit of cipher text, according to a defined system. The unit can consist of single or multiple characters, based on the complexity of the algorithm. The decryption is the exact reverse of the encryption method, where the cipher unit is substituted with its associated plain text unit. Such ciphers are similar

to transposition ciphers, except that the units within the plaintext are rearranged in a complex manner, keeping the units unchanged. In case of Substitution ciphers, the sequence of units is not altered.

There are several types of substitution ciphers. If the unit being substituted is a single letter, it is called a Simple Substitution Cipher. It implies that there is a one to one mapping between the plaintext symbol and the cipher text symbol.

In case of Homophonic Substitution Ciphers, multiple plain text symbols can map to the same ciphertext symbol and vice-versa, making it complex to decipher using frequency analysis. This is also called polyalphabetic substitution ciphers.

2.3   Simple Substitution Cipher

In case of Simple Substitution Cipher, a single plain text symbol is mapped to a single cipher text symbol. This is the simplest case of substitution ciphers, making it susceptible to cryptographic attacks.

In the case of English language, there are 26 distinct symbols that can map to 26 distinct cipher symbols. A total of 26! keys can be computed using this approach. The work factor for decipherment would be 26!, making this approach infeasible.

An improved approach to attack this cipher is using letter frequencies. Each letter in the English language has an associated frequency, making it easier to decipher the text. During decryption, matching the letter frequencies obtained from the cipher text, with the ones associated with the plain text would yield the decryption key. e.g. the letter 'E' in English has the highest letter frequency, which can be matched against the letter with the highest frequency from the cipher text [6, 13].

2.3.1 Caesar Cipher

This is also known as the shift cipher, and is one of the widely known techniques of encryption. In this method, each plain text symbol is replaced with a cipher symbol, which is a number of alphabetic positions behind or ahead. This cipher is similar to the ROT13 [13] system, where the shift is of size 13. The Caesar Cipher [24] is insecure and has no application in the field of security [6, 13].
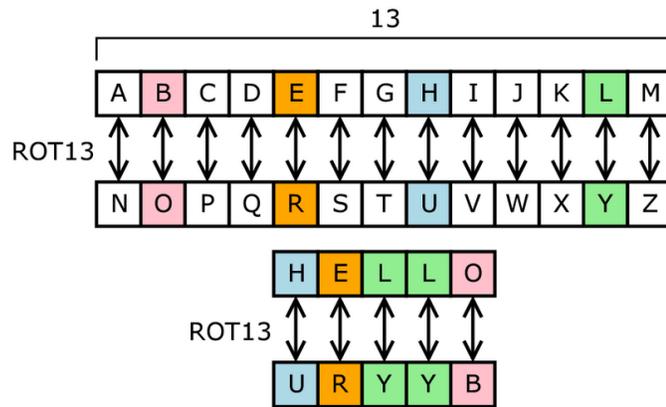
Figure 1: ROT 13 - A Caesar Cipher with Shift 13

.

Encryption of a letter x by a shift n, can be described mathematically as,

$$E_n(x) = (x + n) \mod 26 \tag{1}$$

Decryption is performed similarly,

$$D_n(x) = (x - n) \mod 26 \tag{2}$$

where, let A = 0, B = 2, C = 3, ..., Z = 25.

2.4 Homophonic Substitution Cipher

The homophonic substitution ciphers are comparatively harder to decipher using frequency analysis. In this case, a single plaintext symbol maps to multiple ciphertext

symbols [6]. This would ensure that the frequencies are flattened across all symbols, making it harder to decipher using frequency analysis. But, each cipher text symbol can represent only one plain text symbol.

Based on the Jakobsen algorithm [10], an efficient attack exists to determine the key for homophonic substitution cipher. This algorithm consists of three layers, outer layer, random key layer, and inner layer [28].

2.5 Jakobsen's Algorithm

This section would describe the Jakobsen's Fast attack on Simple Substitution ciphers. This is also applicable to Homophonic substitution ciphers.

2.5.1 Digram Frequencies

Digrams are sequences of 2 letters for a given language. In case of the English language, there are 26^2 possible digrams. The digram sequence 'th' has the highest occurrence and constitutes to 1.5 percent of the total digrams associated with the English language. Table A.11 lists the most common digrams for English language (Computed using 40,000 words). Table A.12 lists the digram frequency counts computed using "Brown Corpus" [25]. Ideally, these counts need to be normalized by dividing it by the number of characters.

Jakobsen [10] provides a fast attack on substitution ciphers, by using only digraph statistics. The entire process requires decrypting the cipher text only once. The following section lists the Jakobsen's Fast Attack [10] on Simple Substitution Ciphers.

2.5.2   Fast Attack on Simple Substitution Ciphers

We assume that the plain text consists of the English language, and the ciphertext would also consist of 26 symbols of the English language.

Jakobsen's algorithm [10] uses digram statistics to compute the score for the current key. The ciphertext is parsed only once to compute the digram distribution matrix for the plaintext. Subsequent scoring is done by modifying the initial putative key (used for decryption) and the associated digram distribution matrix. These two entities are compared with the expected digram statistics for English language for the purpose of scoring.

Determining the initial putative key

The initial putative key is determined by computing the monogram statistics for the given cipher text and comparing it with the statistics for English language.

The sequence E T A O I N S R H L D C U M F P G W Y B V K X J Q Z, indicates the letters in the descending order of their monogram frequencies for English Language. The most frequent alphabet in the ciphertext would correspond to the letter E. Remaining associations are calculated using the similar method.

Scoring a putative key

An efficient way of scoring the putative key, is to calculate the sum of the numerical differences of the digraph frequencies associated with the putative key and the ones for English language. Listed below is the formula for scoring the current putative key:

$$d(X, Y) = \sum_{i,j} |x_{ij} - y_{ij}|$$

Note that the matrices X and Y are of the same dimensions.

The preceding equation would yield the distance between the 2 matrices that represent digraph distributions. The smaller the distance, the more likely that the putative key would decrypt the ciphertext to yield a result that is closer to the actual plaintext. So, this distance can be treated as a score, where a score of 0 implies a perfect match.

We aim to achieve a better score by modifying the putative key, by swapping two elements at a time. Based on this swap, only the rows and columns starting and ending with these two elements need to be swapped within the digram distribution of the putative key.

The Jakobsen's Fast attack assumes the following:

1. Compute E, a matrix which contains the excepted Digram frequencies of English language.

2. Let C be the input ciphertext.

3. Let K be the initial putative decryption key that is computed using monogram frequency statistics of the ciphertext.

4. Let P represent the putative plaintext obtained by decrypting C using the putative key.

5. Let D represent the digram distribution matrix for P.

Algorithm 2.1 depicts a hill climb approach to attack the cipher. At each stage, the modified key is ignored if it does not result in an improved score (lower score).

Algorithm 2.1 Jakobsen's Fast Attack on substitution ciphers

1: Initialize E with expected digram frequencies
2: C = Input cipher text
3: K = Compute Initial Putative Key
4: P = Putative plaintext by decrypting C using K
5: D = Digram distribution matrix for P
6: score = $d(D, E)$
7: **for** $i = 1$ to $n - 1$ **do**
8:    **for** $j = 1$ to $n - i$ **do**
9:       $D' = D$
10:      swapRows(j, $j + i$)
11:      swapColumns(j, $j + i$)
12:      **if** d(D',E) < score **then**
13:         $D = D'$
14:         swapElements(j, $j + i$) {Swap elements of the putative key}
15:         score = d(D',E)
16:      **end if**
17:    **end for**
18: **end for**
19: **return** K

2.6 Hidden Markov Models

A Markov Chain or Markov Process [23] refers to the memoryless process of a stochastic process. A stochastic process possesses the Markov property if the conditional probability of the successive states depends only upon the present state and not the ones preceding it.

A Hidden Markov Model [23] or HMM is a statistical Markov Model in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states. In simpler Markov models (like a Markov chain), the state is directly visible to the observer, and therefore the state transition probabilities [23] are the only parameters. In a HMM, the internal state is not directly visible, but the output (dependent on the internal state) is visible. Each state has a probability distribution over the possible output tokens. Therefore, the sequence of tokens generated by an

HMM provides some information regarding the sequence of internal states.

2.6.1 Notation

The components of a Hidden Markov Model can be represented using the following notation:

$$T = \text{length of the observation sequence}$$

$$N = \text{number of states in the model}$$

$$M = \text{number of observation symbols}$$

$$Q = \{q_0, q_1, \ldots, q_{N-1}\} = \text{distinct states of the Markov process}$$

$$V = \{0, 1, \ldots, M-1\} = \text{set of possible observations}$$

$$A = \text{state transition probabilities}$$

$$B = \text{observation probability matrix}$$

$$\pi = \text{initial state distribution}$$

$$\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1}) = \text{observation sequence}$$

Figure 2 depicts the basic layout of a Hidden Markov Model. The internal states are labeled in 'x', which would be the plain text symbol. The observations are labeled in 'y', which represents a ciphertext symbol. The transition probabilities amongst internal states are labeled in 'a', which is an element of the A matrix. Finally, the observation probabilities are depicted using the label 'b' amongst the observation symbol and internal state.

2.6.2 HMM Problems

Formally, there are 3 problems [23] associated with the Hidden Markov Model.
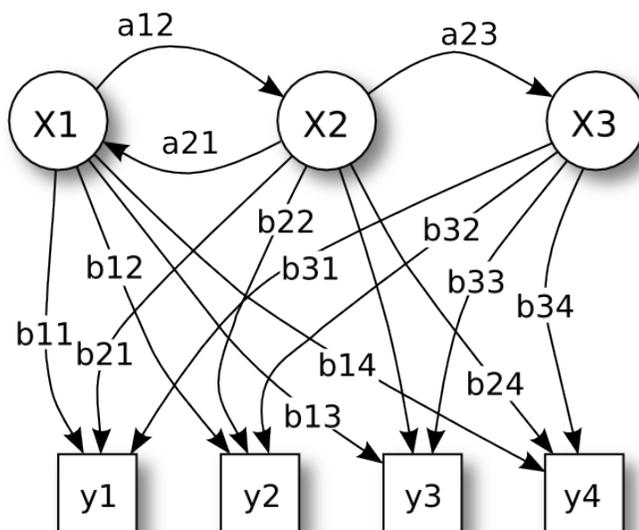
Figure 2: Hidden Markov Model

2.6.2.1 Problem 1

This deals with model evaluation. Given the model and a sequence of observations, we need to find the probability $P(\mathcal{O}|\lambda)$. The main idea is to find the likelihood of the sequence given the model.

2.6.2.2 Problem 2

Given the model and the sequence of observations, we need to find the optimal sequence for the underlying Markov process. The main idea is to uncover the hidden part of the Hidden Markov Model.

2.6.2.3 Problem 3

Given the observation sequence and the dimensions N and M, find the model lambda that maximizes the probability of the given sequence

2.6.3 HMM Scaling

This concept is required to avoid numerical underflow while calculating probabilities after each stage of the HMM algorithm. This is achieved by summing up the values of α_t at t and dividing each quantity by this summed quantity.

2.6.4 Baum-Welch Algorithm

The Baum-Welch algorithm is used to find unknown parameters of the Hidden Markov Model. Before describing the Baum-Welch algorithm, we would briefly explain the Forward algorithm.

2.6.4.1 Forward Algorithm

This algorithm is the solution to Problem 1, i.e. finding the probability that the given observation sequence was generated from the given model. The following equation depicts the probability:

$$\text{Probability} = P(\mathcal{O}|\lambda)$$

where $\mathcal{O}$ is the observation sequence and λ is the model.

This would imply that the model would have been trained on the given observation sequence in the past. The matrix B would contain the state transition probabilities that would conform with the given observation sequence to yield a higher probability.

The Forward Algorithm analyzes the probabilities of transitioning from one state to the next based on the given observations. The solution to the Forward Algorithm is to iteratively compute the state transition probabilities for each observation.

Let the newly computed state transition probabilities be stored in a new variable

called α, which is a 2 dimensional array of size $N \times T$. Here N is the number of states and T is the number of observations. At a given time t, α would contain the probability of producing the observation $\mathcal{O}$ for a given state $j-1$ and the probabilities of traversing through all states up till state $j-1$.

This algorithm is recursive, so we assume that the transition probabilities before state j have already been computed. The first step or the initialization step, are depicted within π, which is a 1 dimensional array of size $1 \times T$. It contains the probabilities of starting from the initial state. In other words this process would be computed in a top-down approach. The probability of producing the first observation symbol is given by:

$$P(\mathcal{O}_1) = \pi_i b_i(\mathcal{O}_1)$$

The algorithm terminates by summing up the values of α at time t=T. This gives the probability of traversing through all states and producing the entire observation sequence, for the given model. To summarize, the Forward Algorithm can be defined by the following recurrence(s):

The formula to calculate the scaling factor is given by:

$$\alpha_t(i) = \sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji} b_i(\mathcal{O}_t)$$

$t = $ observation at time t

$a = $ contains the state transition probabilities

$b = $ contains the emission probabilities

As T increases, the $\alpha_t(i)$ tends to 0. This will lead to a numerical underflow, which can be avoided by scaling the numbers. The scaling factor is calculated by summing

13

up the values of α at time t. Then each value of α for time t is divided by this scaling factor. The scaling factor is a 1 dimensional array with dimensions $1 \times T$.

The formula to calculate the scaling factor is given by:

$$c_t = \frac{1}{\sum_{j=0}^{N-1} \alpha_t(j)}$$

Then scale each value on α_t by multiplying with the scaling factor c_t:

$$\alpha_t(i) = c_t \alpha_t(i) \tag{3}$$

According to equation 3, summing up all values at time t would result in the numerator being equal to the denominator:

$$\sum_{j=0}^{N-1} \alpha_{T-1}(j) = 1$$

With reference to (3),

$$\alpha_t(i) = c_0 c_1 c_2 ... c_t \alpha_t(i)$$

Which implies that:

$$\sum_{j=0}^{N-1} \alpha_{T-1}(j) = c_0 c_1 c_2 ... c_{T-1} \sum_{j=0}^{N-1} \alpha_{T-1}(j)$$

$$= c_0 c_1 c_2 ... c_{T-1} P(\mathcal{O}|\lambda)$$

After combining these results, gives the following result:

$$P(\mathcal{O}|\lambda) = \frac{1}{\prod_{j=0}^{T-1} c_j}$$

This implies that the reciprocal product of all scaling factors will yield the final probability of the observation sequence given the model. Finally to avoid underflow, we usually take the *log-likelihood* of the probability. This is given by:

$$\log P(\mathcal{O}|\lambda) = -\sum_{j=0}^{T-1} \log c_j$$

The Backward Algorithm analyzes and iteratively computes the transition probabilities in the reverse order. This method is similar to the Forward Algorithm, which starts from the first observation symbol and traverses all successive symbols. Similar to the Forward Algorithm, the same scaling factor is used for calculating $\beta_t(i)$ in case of the Backward Algorithm.

This is followed by computing Gamma γ and Di-gamma $\gamma(i,j)$ using α, β and matrices A, B. The final step is to re-estimate the values for π, A and B. The subsequent iteration starts again with the Forward Algorithm, Backward Algorithm, Compute Gamma, Di-Gamma and Re-estimation. The process repeats until the probability is better than the previous iteration. The subsequent section describes the entire process of the Baum-Welch Algorithm in detail with the help of pseudo-code.

2.6.4.2   Complete Pseudocode

The complete pseudo-code for HMM. includes alpha, beta pass, compute di-gamma. Re-estimate A, B, π and finally the probability. The Baum-Welch algorithm consists of the following step:

1. The $\alpha - pass$

2. The $\beta - pass$

3. Compute γ and $Di - \gamma$

4. Re-estimate entities π and B.

The following section depicts the pseudocode for Baum-Welch algorithm [23]:

Initialization

The entities π, A and B are initialized with random values, where each row sums to 1. In other words the entities should be row stochastic and should not contain uniform values.

$$\pi = \text{is of size } 1 \times N$$

$$A = \{a_{ij}\} \text{ is of size } N \times N$$

$$B = \{b_{ij}\} \text{ is of size } N \times M$$

$$\text{maxIters} = \text{Number of times to re-estimate parameters}$$

$$\text{iter} = 0$$

$$\text{existingLogProb} = \text{NULL}$$

Alpha pass

This stage deals with the Forward Algorithm, where the values for α are computed iteratively, starting from the first observation.

Algorithm 2.2 Compute $\alpha_0(i)$

$c_0 = 0$
for $i = 0$ to $N - 1$ **do**
$\quad \alpha_0(i) = \pi_i b_i(\mathcal{O}_0)$
$\quad c_0 = c_0 + \alpha_0(i)$
end for

The algorithm above computes the values for α_0, which is for $t = 0$. We also calculate the scaling factor c_0 by summing up all values for α at $t = 0$. Next, we scale the values for α that were calculated up till this point. As mentioned earlier, scaling is necessary to avoid numerical underflow. This is done by dividing each individual value by the scaling factor.

Algorithm 2.3 Scale $\alpha_0(i)$

$c_0 = \frac{1}{c_0}$
for $i = 0$ to $N - 1$ **do**
$\quad \alpha_0(i) = c_0 \alpha_0(i)$
end for

Next, we calculate the remaining values of α for $t = 1$ up till $t = T - 1$.

Algorithm 2.4 Compute remaining values for $\alpha_t(i)$

---

for $t = 0$ to $T - 1$ **do**
$\quad c_t = 0$
$\quad$ **for** $i = 0$ to $N - 1$ **do**
$\quad\quad \alpha_t(i) = 0$
$\quad\quad$ **for** $j = 0$ to $N - 1$ **do**
$\quad\quad\quad \alpha_t(i) = \alpha_t(i) + \alpha_{t-1}(j)a_{ji}$
$\quad\quad$ **end for**
$\quad\quad \alpha_t(i) = \alpha_t(i)b_i(\mathcal{O}_t)$
$\quad\quad c_t = c_t + \alpha_t(i)$
$\quad$ **end for**
end for

Repeat the process for scaling all values for α computed in the previous step.

Algorithm 2.5 Scale $\alpha_t(i)$

$c_t = \frac{1}{c_t}$
for $i = 0$ to $N - 1$ **do**
$\quad \alpha_t(i) = c_t \alpha_t(i)$
end for

Beta Pass

This section describes the beta pass or the Backward Algorithm. This is similar to the Forward Algorithm, but operates in the reverse order starting from the final observation up till the first observation. The scaling factors computed during the α-pass are re-used to scale the values contained within β. Initialize the values for β at $t = T - 1$ with the scaling values at c_{T-1}.

Algorithm 2.6 Scale $\beta_{T-1}(i)$

---
for $i = 0$ to $N - 1$ **do**
   $\beta_{T-1}(i) = c_{T-1}$
end for

Compute the remaining values for β starting from $t = T - 2$ up till $t = 0$.

Algorithm 2.7 Compute and scale remaining $\beta_t(i)$

---
for $t = T - 2$ to 0 **do**
  for $i = 0$ to $N - 1$ **do**
     $\beta_t(i) = 0$
    for $j = 0$ to $N - 1$ **do**
       $\beta_t(i) = \beta_t(i) + a_{ij}b_j(\mathcal{O}_{t+1})\beta_{t+1}(j)$
    end for
     $\beta_t(i) = c_t\beta_t(i)$ {Scale the values for β_t}
  end for
end for

Compute Gamma

The variables Gamma $\gamma(i)$, and Di-Gamma $\gamma(i,j)$ are additional parameters that are computed in order to help re-estimate the parameters π (the initial probabilities) and B (emission probabilities).

Algorithm 2.8 Compute γ
---

  for $t = 0$ to $T - 2$ **do**
    denom $= 0$
    for $i = 0$ to $N - 1$ **do**
      for $j = 0$ to $N - 1$ **do**
        denom $=$ denom $+ \alpha_t(i)a_{ij}b_j(\mathcal{O}_{t+1})\beta_{t+1}(j)$
      end for
    end for
    for $i = 0$ to $N - 1$ **do**
      $\gamma_t(i) = 0$
      for $j = 0$ to $N - 1$ **do**
        $\gamma_t(i,j) = (\alpha_t(i)a_{ij}b_j(\mathcal{O}_{t+1})\beta_{t+1}(j))/$denom
        $\gamma_t(i) = \gamma_t(i) + \gamma_t(i,j)$
      end for
    end for
  end for

Reestimate π, B

This section describes the process to re-estimate the values for pi and B. Since A contains the state transition probabilities, which in other words are the digraph frequency statistics for the English language, it should not be re-estimated. This would make sure that the digraph frequency statistics are preserved for the entire process.

Algorithm 2.9 Reestimate π
---

  for $i = 0$ to $N - 1$ **do**
    $\pi_i = \gamma_0(i)$
  end for

Algorithm 2.10 Reestimate B

---
for $i = 0$ to $N - 1$ **do**
  for $i = 0$ to $M - 1$ **do**
    numer $= 0$
    denom $= 0$
    for $t = 0$ to $T - 2$ **do**
      if $\mathcal{O}_t == j$ **then**
        numer $=$ numer $+ \gamma_t(i)$
      end if
      denom $=$ denom $+ \gamma_t(i)$
    end for
    $b_i(j) = \dfrac{\text{numer}}{\text{denom}}$
  end for
end for

Compute the probability

Finally, compute the probability by summing up the log values of the scaling factor, c:

Algorithm 2.11 Compute probability

---
logProb $= 0$
for $i = 0$ to $T - 1$ **do**
  logProb $=$ logProb $+ log(c_i)$
end for
logProb $= -$logProb

If the $logProb$ computed in the current iteration is greater than the value computed in the previous iteration, proceed to the next iteration by repeating from first step.

2.7 Graphics Processing Unit

A Graphics Processing Unit [21] or GPU is a specialized piece of hardware capable of rapidly accessing memory as well as manipulate it, for the purposes of displaying high quality graphics. Their highly parallel structure is best suited for purposes where

several blocks of memory need to be altered concurrently. This is not efficient in case of a CPU (traditional processor), since it has a limited number of cores.

Such specialized hardware interfaces with the motherboard through an expansion slot such as the PCI-Express interface, which is a high speed serial expansion bus standard capable of achieving bandwidth rates of 8 GB/s. Intra device communication can reach up to 120 GB/s.

There are several application of the GPU, such as in the field of Computer Vision, Computational Finance, Computer Aided Design, and many more. Popular programs like Adobe Suite of tools and MATLAB, utilize the GPU to accelerate computationally intensive tasks.

2.7.1 Overview
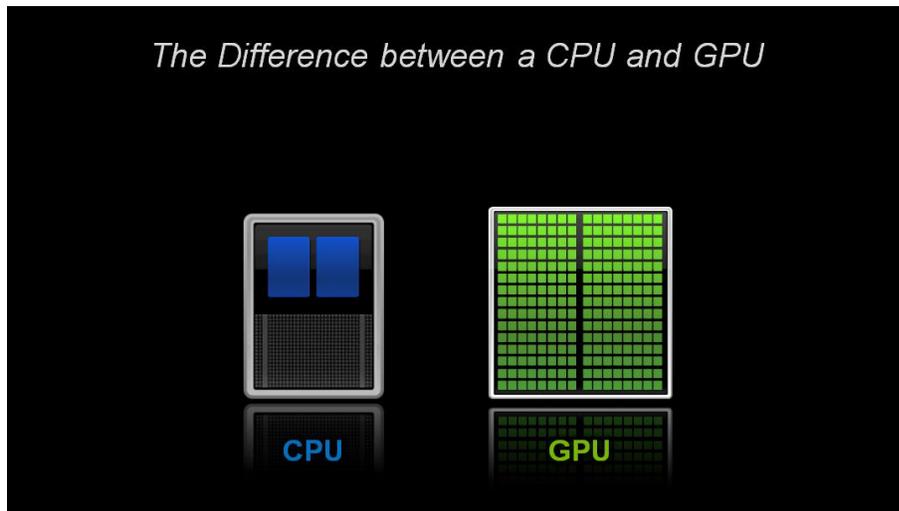
Figure 3: The difference between a CPU and GPU [11]

A traditional CPU contains few cores and can support few threads at a time. The GPU consists of hundreds or even thousands of cores, and can support thousands of threads simultaneously. This allows a program to speed up execution by $100\times$ on

a GPU, making it very efficient.
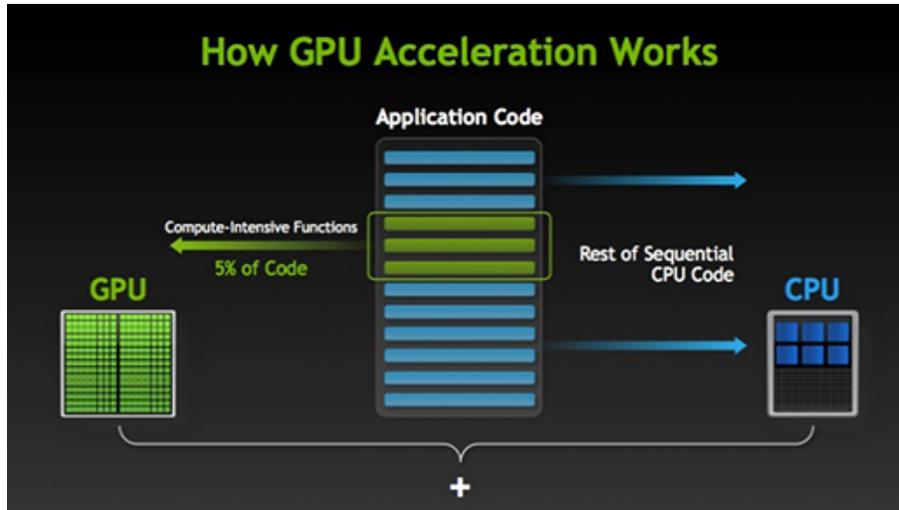
Figure 4: How does GPU Accelerated code work

A program intended to be executed on a GPU, consists of heterogeneous code [26], i.e. code that can execute on CPU and GPU. The computationally intensive code section would execute on the GPU and return the result back to the CPU, while the sequential code continues to execute on the CPU.

Figure 5 depicts a machine that has thousands of tubes, each tube capable of painting a single pixel independent of the other. All tubes paint their respective pixel simultaneously, acting similar to how several GPU processors execute certain instructions in parallel.

2.7.2 CUDA programming language

Also known as Compute Unified Device Architecture [16] is a parallel computing platform developed by NVIDIA, using which the GPU is suitable for general purpose
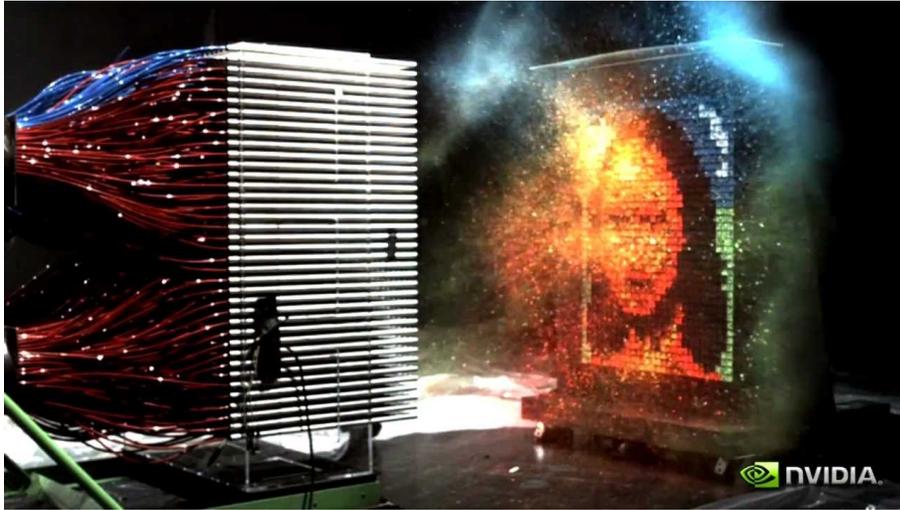
Figure 5: This image depicts how a GPU functions by painting a picture instantaneously

applications such as High Performance Computing. CUDA is generally an extension to existing programming languages like C, C++, Python and Fortran. Recently, OpenCL [20] has also become broadly supported, which is an open standard defined by the Khronos group and provides cross platform support.

The CUDA programming language enables developers to access the GPU instruction set directly to accelerate general purpose applications in the field of biological simulations, cryptography, and several other areas. GPUs are used for several purposes including graphics rendering as well as in game physics using the PhysX engine. The current stable release of CUDA is *v7.0*.

The basic flow for a CUDA program [30] is listed below:

- Allocate memory for the variables on the GPU.

- Implement the code and load to GPU. The GPU would use the memory allocated in Step 1.

- Once execution completes, copy the results back from the GPU to the CPU.

The special section of code executed on the GPU is called a kernel. CUDA allows the definition for a kernel on any supporting language.

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

The __global__ keyword, would enable the host code (CPU) to launch the *add* method on the *device* (GPU). The variables *a*, *b*, and *c* should point to the device (GPU) memory. All keywords highlighted in red belong to the CUDA Programming language.

A sample kernel launch sequence is listed below:

```
add<<< 1, 1 >>>( a, b, c );
```

The method name is followed by the Grid and Block dimensions enclosed within triple brackets, followed by the method arguments. A Grid consists of multiple *Blocks*, and a Block consists of multiple Threads. In the preceding sample, the Grid and Block dimensions are of size 1, resulting in a single launch of the kernel.

```
add<<< N, 1 >>>( a, b, c );
```

The preceding code segment would launch N instances of the kernel, with N Blocks and 1 Thread per Block. Since, the method 'add' runs in parallel, it is possible to speed up vector addition. Each instance can refer to its Block using the keyword ThreadIdx.x. The following sample uses ThreadIdx.x to access the designated element within the array.

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

2.7.2.1 cuBLAS

The cuBLAS (Compute Unified Basic Linear Algebra Subroutines) library [17] consists of routines that accelerate Vector-Vector, Matrix-Vector and Matrix-Matrix algebra. These routines are invoked from the host code and execute efficiently on the GPU. The library provides routines for single, double precision and complex inputs. This library is based on the original BLAS, consisting of similar low level routines, implemented by high level math programming frameworks like MATLAB and R.

cuBLAS Examples

We assume all device variables have been pre-allocated and cuBLAs has been initialized. All methods used in the examples are intended for inputs with double precision.

Vector DOT product

We assume two vectors a_d and b_d of size 10, which are allocated on the device. The result of the operation would be stored by $result\_d$. The following code sample depicts the dot product for two vectors.

```
int N = 10;
double *a_d;
double *b_d;
double *result_d;
```

```
cudaMalloc((void **)&a_d, sizeof(double) * N);
cudaMalloc((void **)&b_d, sizeof(double) * N);
cudaMalloc((void **)&result_d, sizeof(double));


cublasStatus_t status = cublasDdot(handle, N, a_d, 1, b_d, 1,
                                   &result_d);
```

Matrix-Vector Multiplication

The method cublasDgemv multiplies a matrix with a vector. In the following example, the matrix a_d is multiplied with the vector b_d, the resulting vector is stored within $result\_d$. The variables $scalar\_d$ and $zero\_d$ are initialized with values 1 and 0. All variables are allocated on the device.

```
int N = 10;
double *a_d;
double *b_d;
double *scalar_d;
double *zero_d;
double *result_d;


cudaMalloc((void **)&a_d, sizeof(double) * N * N);
cudaMalloc((void **)&b_d, sizeof(double) * N);
cudaMalloc((void **)&scalar_d, sizeof(double));
cudaMalloc((void **)&zero_d, sizeof(double));
cudaMalloc((void **)&result_d, sizeof(double) * N);
```

```
//Initialize scalar_d with 1 and zero_d with 0.
cudaMemset(scalar_d, 1, sizeof(double));
cudaMemset(zero_d, 0, sizeof(double));

cublasStatus_t status = cublasDgemv(handle, CUBLAS_OP_T,
                        N, N,
                        scalar_d,
                        a_d, N,
                        b_d, 1,
                        zero_d,
                        result_d, 1);
```

CHAPTER 3

Hidden Markov Models with Random Restarts

Experiments were performed on the CPU and the GPU having different hardware configurations. Each section describes each experiment in detail.

3.1 What is a Random Restart?

This subsection briefly describes the concept of Random Restart [22] with context to a Hidden Markov Model. We start by covering related work, then describe Hill Climbing and its disadvantages. Finally, we define random restarts and its advantage.

3.1.1 Hill Climbing and its Disadvantage

The hill climbing technique starts from an initial solution and iteratively tries to improve the score based on some function [4, 6]. During each iteration, the solution from the previous step is re-evaluated by modifying the input and subjecting it to the evaluation function. The new input is chosen only if the score improves, else the previous input is modified in a different manner. Figure 6 depicts the Hill Climbing algorithm.

3.1.2 Random Restarts and its Advantage

The hill climbing technique (Figure 6) requires the initial start point and the number of iterations. If the initial start point is not optimal, the solution obtained would correspond to a local maximum. Choosing a different initial start point might improve the score or obtain a higher local maximum, which may well be a global maximum.
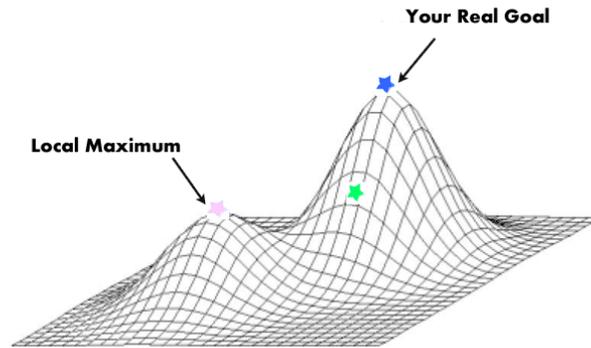
Figure 6: Hill Climbing algorithm, main goal is to attain Global Maximum [27].

The entities π and B [23] are initialized using random values, generated using a Pseudo Random Number Generator (PRNG). The PRNG is seeded using a random value, which if varied would imply a different start point for the training phase for the HMM. The seed for the Pseudo Random Number Generator is generated with the help of some entropy. e.g. the System Time is a good example exhibiting randomness. Due to this process, the algorithm achieves convergence faster resulting in the Global Maximum.

3.2 HMM with Random Restarts (CPU)

This section describes the entire process of executing Hidden Markov Model with multiple random restarts uptill the final stage of computing the accuracy. Subsequent subsections describe certain steps in detail.

3.2.1 Related Work

There has been considerable research being done with Hidden Markov Models. Berg-Kirkpatrick *et al.* [2] has attempted to crypt-analyze the Zodiac-304 and Zodiac-340 [5] ciphers with the help of HMM with random restarts. Nuhn *et al.* [14] provides an EM based training procedure for probabilistic substitution ciphers, providing high decipherment accuracies.

3.2.2 Algorithm Overview

The main steps to decipher text using Hidden Markov Model with random restarts are listed below:

- Train the Hidden Markov Model using English text to obtain digraph statistics from matrix A. Alternatively, the frequencies can be calculated externally and substituted within matrix A.

- Compute a random encryption key.

- Encrypt a sample dataset (English language) using the encryption key, to obtain the associated cipher text.

- Train the Hidden Markov Model using the cipher text obtained from the previous step. The matrix A would contain digraph frequencies obtained from the first step. Matrix A should not be modified during the re-estimation phase.

- Obtain the decryption key from matrix B, which contains the emission probabilities for each encrypted symbol (observation) and the plain text symbol. Choose the most probable internal state (plain text symbol) for the given observation (cipher-text symbol). Section 3.2.2.1 describes this step in detail.

- Decrypt the text using the decryption key obtained from the previous step.

- Compare the decrypted text with the original plain text to obtain the success rate or accuracy by using the formula:

$$\text{Success Rate or Accuracy} = \frac{\text{Number of letters that match}}{\text{Total number of letters}}$$

Section 3.2.2.2 describes this step in detail.

NOTE: We normalize scores or accuracy across multiple test cases. Each test-case deals with a randomly generated encryption key.
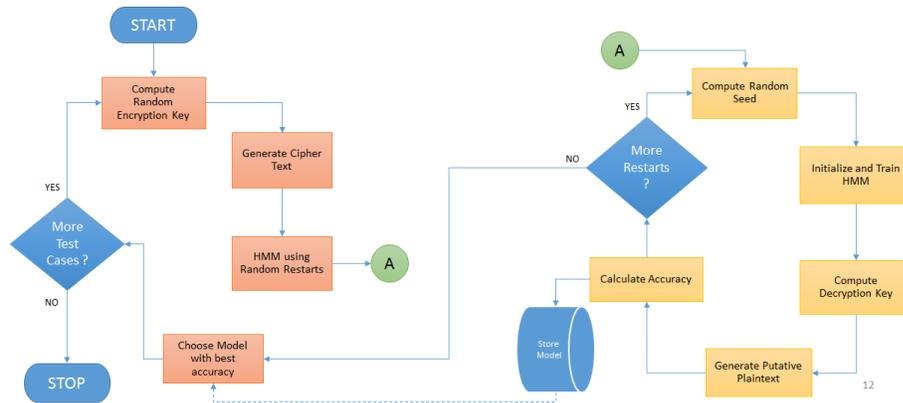
Figure 7: Flow Diagram for Cryptanalysis using HMM with Random Restarts

Moreover, we use the concept of random restarts associated with the HMM. In this case, the Pseudo Random Number Generator is seeded with a random value and is used to initialize the entities π and B. This could possibly improve the accuracy, resulting in the global maximum.

3.2.2.1   Computing Decryption Key

Once the model has been trained, the matrix B would contain the final emission probabilities associated with each state. The aim is to find the best emission probability associated with each observation, resulting in the mapped internal state.

The row headers represent the internal (hidden) states, while the column headers represent the observation. The process starts by choosing an observation symbol (column header), and then selecting the row with the highest probability, that associates with this column. In other words, we are finding the most probable internal state associated with the current observation symbol. This process is repeated for all symbols.

A simple example:

Let us consider an example to describe the process of computing the decryption key from the matrix B once the HMM model has been trained. Table 1 depicts the contents of the matrix B^T, containing the final emission probabilities after the training phase. We assume that the HMM consists of 6 internal states (depicted with alphabets a through f). We also assume that the observation sequence consists of 6 unique symbols (1 through 6). The objective is to find the most probable internal state for each observation symbol.

Table 1: Sample contents for the matrix B^T from a trained model.

	1	2	3	4	5	6
a	0.00000	0.00000	0.92113	0.00000	0.00000	0.00000
b	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000
c	0.00000	0.00000	0.00000	0.00000	1.00000	0.00000
d	0.00000	0.00000	0.00000	0.00000	0.00000	0.45910
e	0.97749	0.00000	0.00000	0.00000	0.00000	0.00000
f	0.00000	0.55477	0.00000	0.00000	0.00000	0.00000

For each observation symbol, select the highest probability associated with the column. This implies choosing the most probable internal state associated with this observation symbol.

Table 2: Most probable internal state chosen for each observation symbol.

	1	2	3	4	5	6
a	0.00000	0.00000	0.92113	0.00000	0.00000	0.00000
b	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000
c	0.00000	0.00000	0.00000	0.00000	1.00000	0.00000
d	0.00000	0.00000	0.00000	0.00000	0.00000	0.45910
e	0.97749	0.00000	0.00000	0.00000	0.00000	0.00000
f	0.00000	0.55477	0.00000	0.00000	0.00000	0.00000

Table 2, depicts the most probable internal state for each observation symbol. The probabilities are highlighted in red. An interesting fact is that the chosen probabilities form a perfect diagonal, which implies that the input text was encoded using the Caesar Cipher [24] with Shift 3. This example depicts a converged model. In a complex scenario, there would be many internal states and observation symbols. The model may not converge, unless the experiment was performed for a high number of iterations. In such cases every column might consist of multiple probabilities, and the highest probability is chosen.

3.2.2.2 Calculating Accuracy

The accuracy might vary based on a given ciphertext, so we average the accuracy across multiple test cases. For each test case, a ciphertext is generated using a random encryption key and a given plain text. The model is trained using the given ciphertext for certain iterations. The associated decryption key is computed by looking up the most probable internal state for the given observation symbol. The ciphertext is decrypted using this decryption key and the associated plaintext is matched using

the original plaintext to compute the percentage of symbols that matched. This process is continued for multiple test cases and the accuracy is averaged across all test cases.

3.3   Baum-Welch algorithm (GPU)

This section describes the previous work dealing with HMMs in the field of Speech Recognition and Pattern Recognition. This is followed by a brief overview of the optimized algorithm on the Graphics Processing Unit.

3.3.1   Related Work

There has been much research and development dealing with HMM within the GPU domain. Liu [12] provides an efficient method to implement the various algorithms (Forward, Viterbi and Baum-Welch) for the Hidden Markov Model on the GPU. The implementation is compatible with CUDA v2.0. Hymel [9] improved the implementation and used multiple HMM's for the purpose of pattern recognition. The efficiency is noticeable for large number of states and iterations. Yu *et al.* [29] achieved 9.2x and 7.9x speedup during the training and testing stages, when used as a Speech Recognition platform for real-time applications. The performance can be limited due to the hardware's memory bandwidth and availability of resources. Yu *et al.* [29] states that the GPU version outperforms a single threaded CPU version for internal states greater than 256 for the Forward Algorithm.

3.3.2   Algorithm Overview

The Baum-Welch algorithm for the GPU [9] would be similar to the CPU version. All steps dealing with Vector-Vector, Matrix-Matrix and Matrix-Vector algebra are

implemented using customized kernels and the cuBLAS library. This library provides methods that can perform matrix-vector and matrix-matrix algebra in a very efficient manner, by interleaving the operations across multiple processors of the GPU. The implementation is compatible with CUDA Toolkit v6.5.

One such example would be in case of the $\alpha - pass$ associated with the Baum-Welch algorithm. With reference to Algorithm 2.4, the inner loop consisting of j runs for $(N-1)$ times. With the help of the cuBLAS DOT product $cublasDDot()$, this can reduced to a single operation. The method $cublasDdot()$ computes the DOT product by distributing the processing across $(N-1)$ GPU threads. The runtime for the outer loop consisting of $\alpha_t(i)b_i(\mathcal{O}_t)$ is $O(N-1)$, which is also reduced to a single operation or $O(\log N)$ with the help of CUBLAS method $cublasDgemv()$. The overall runtime for Algorithm 2.4 reduces to $O(T \log N)$, the $\log N$ is due to the recursive property of the algorithm.

Algorithms 2.2 and 2.3 speed up with the help of $cublasDdot()$, reducing the runtime from $(N-1)$ steps to a single step. Similar operations have been used for speeding up the $\beta - pass$, $compute\ \gamma$ and the re-estimation phase.

CHAPTER 4

Experiments Setup

4.1 CPU based Experiments

The experiments intended to execute on the CPU were developed using Visual Studio 2013 Ultimate Edition, and compiled using gcc compiler on the remote server. Table 3 lists the configuration of the remote system.

Table 3: System Configuration (CPU)

CPU	Intel Xeon E5-2697 (12 Cores), 2.7 GHz
RAM	8 GB
HDD	1 TB
OS	Cent OS 6.6

4.2 GPU based Experiments

We briefly define some basic concepts before describing the actual experiment.

Amazon Web Services

Amazon Web Services is a division of Amazon Inc., specializing in cloud computing services (Platform as a Service) and web based storage (Storage as a Service) and available for enterprises and consumers.

The NVIDIA CUDA compiler

The NVIDIA CUDA Compiler (nvcc) [15] is specially designed to compile the source code containing heterogeneous code, i.e. a mix of code executing on the CPU and GPU. The nvcc compiler separates the CPU code and sends it to the gcc or g++ compiler, while the GPU code section is compiled by the nvcc compiler itself. The

linking stage includes the compiled GPU functions within the host binary. e.g. The GPU code may internally use the cuBLAS library, so the linker would include the CUBLAS compiled binaries into the host binary.

PUTTY

PUTTY is an open source terminal, allowing an end user to connect to remote systems. The software supports multiple protocols like SSH (with encryption), Telnet and SCP.

Table 4: AWS GPU Instance configuration [1]

CPU	Intel Xeon E5-2670 (8 Cores), 2.6 GHz
RAM	15 GB
HDD	30 GB
OS	Amazon Linux
GPU	NVIDIA GRID K520

Table 5: NVIDIA GRID K520 specifications [19]

Model	NVIDIA GRID K520
Total GPUs	2 GK104 GPUs
GPU Cores	3072 (1536/GPU)
GPU Core Clock	800 MHz
Memory	8 GB (4GB/GPU)

This experiment was developed and executed on an Amazon Web Services GPU instance (code named 'g2.2xlarge'), which is attached with a NVIDA GRID K520 GPU [19]. This was achieved by connecting to the remote system using PUTTY. Moreover, the code was also developed locally using Visual Studio with the NSight plugin (bundled with the CUDA toolkit v6.5). The program was compiled using the nvcc [15] compiler on the remote machine, supplied with additional arguments to

indicate that the source uses cuBLAS library routines. Sample compilation sequence:

$$nvcc - I\text{<Path to CUDA samples>} - lcublas \ \text{< Program Name >} .cu$$

Tables 4 and 5 depict the system specification for an AWS GPU Instance and NVIDIA GRID K520 GPU.

CHAPTER 5

Results

Experiments were performed against the text encoded using Simple and Homophonic Substitution Ciphers. This section lists the results for Substitution ciphers using Jakobsen's algorithm and HMM with random restarts.

5.1 Jakobsen's Fast Attack

In this experiment we implemented Jakobsen's Fast Attack [10] over a cipher text obtained by encrypting a random selection of the "Brown Corpus" [25]. We perform this experiment multiple times, each time we use a random encryption key to encrypt the text selected from the Brown Corpus. Figure 8 depicts the results obtained by using random selections of encrypted text extracted from 'Alice In Wonderland', using a random encryption key. The expected digraph frequencies 'E', are calculated using the Brown Corpus. The accuracy based on data is always higher than the one based on the key.
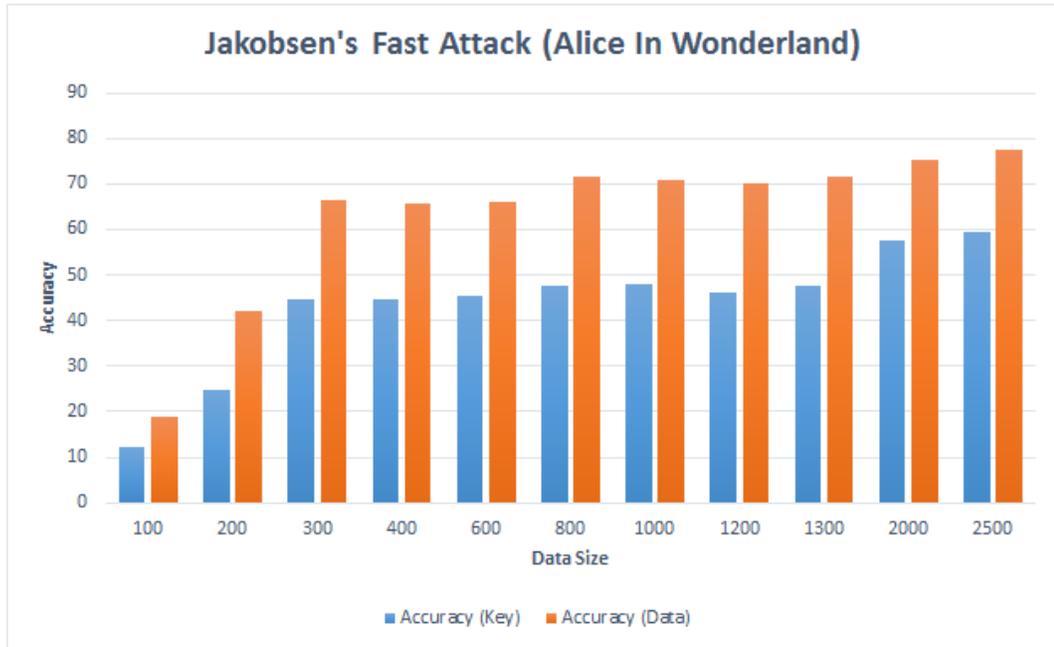
Figure 8: Results for Jakobsen's Fast Attack (Accuracy vs Data Size)

5.2   HMM with Random Restarts (CPU)

This section describes the results dealing with Caesar Cipher and Simple Substitution Cipher.

5.2.1   Caesar Cipher

In this experiment, we encrypt some sample text using Caesar cipher with Shift 3. Then we train the HMM over the encrypted text to obtain the trained model. We decrypt the cypher text using the decryption key, computed from matrix B and compute the accuracy.

```
In cryptography, a Caesar cipher, also known as a Caesar's cipher,
the shift cipher, Caesar's code or Caesar shift, is one of the
simplest and most widely known encryption techniques.  It is a type of
substitution cipher in which each letter in the plaintext is replaced
```

by a letter some fixed number of positions down the alphabet. For
example, with a shift of 3, A would be replaced by D, B would become
E, and so on. The method is named after Julius Caesar, who used it
to communicate with his generals.,The encryption step performed by a
Caesar cipher is often incorporated as part of more complex schemes,
such as the VigenÃre cipher, and still has modern application in the
ROT13 system. As with all single alphabet substitution ciphers, the
Caesar cipher is easily broken and in practice offers essentially no
communication security.

The sample text listed above, is encrypted with Caesar Cipher having Shift 3. We
ignore all characters except alphabets.

Encryption Key (Shift = 3)

ABCDEFGHIJKLMNOPQRSTUVWXYZ

||||||||||||||||||||||||||

DEFGHIJKLMNOPQRSTUVWXYZABC

The associated encrypted text is listed below:

LQFUBSWRJUDSKBDFDHVDUFLSKHUDOVRNQRZQDVDFDHVDUVFLSKHUWKHVKLIWFLSKHUFDHV

DUVFRGHRUFDHVDUVKLIWLVRQHRIWKHVLPSOHVWDQGPRVWZLGHOBNQRZQHQFUBSWLRQWHFK

QLTXHVLWLVDWBSHRIVXEVWLWXWLRQFLSKHULQZKLFKHDFKOHWWHULQWKHSODLQWHAWLVUH

SODFHGEBDOHWWHUVRPHILAHGQXPEHURISRVLWLRQVGRZQWKHDOSKDEHWIRUHADPSOHZLWK

DVKLIWRIDZRXOGEHUHSODFHGEBGEZRXOGEHFRPHHDQGVRRQWKHPHWKRGLVQDPHGDIWHUMX

OLXVFDHVDUZKRXVHGLWWRFRPPXQLFDWHZLWKKLVJHQHUDOVWKHHQFUBSWLRQVWHSSHUIRU

PHGEBDFDHVDUFLSKHULVRIWHQLQFRUSRUDWHGDVSDUWRIPRUHFRPSOHAVFKHPHVVXFKDVW

KHYLJHQUHFLSKHUDQGVWLOOKDVPRGHUQDSSOLFDWLRQLQWKHURWVBVWHPDVZLWKDOOVLQJ

OHDOSKDEHWVXEVWLWXWLRQFLSKHUVWKHFDHVDUFLSKHULVHDVLOBEURNHQDQGLQSUDFWLF

41

```
HRIIHUVHVVHQWLDOOBQRFRPPXQLFDWLRQVHFXULWB
```

The model is trained over the English language, resulting in matrix A containing the digraph frequencies. The HMM is now trained using the following inputs:

- The encrypted text as described above

- The matrix A from the previous training phase

- Entities π and B are initialized with random values using a Pseudo Random Number Generator.

Note that the matrix A is not modified during the training phase. Matrix B contains the emission probabilities, and depicts the association between the observation symbol (Encrypted symbol) and the internal state (English symbol). The decryption key is computed from matrix B after training the model.

We list the parameters for best case of the experiment (using random restarts):

Data Size: 671

HMM iterations: 1400

The resulting decryption key obtained from matrix B is:

<div align="center">

ABCDEFGHIJKLMNOPQRSTUVWXYZ

| | | | | | | | | | | | | | | | | | | | | | | | | |

XY?KBCDEFVHIJFLMNOPZRSTUVW

</div>

Success Rate (Key): 0.807692

The encrypted data is decrypted using the decryption key, resulting in the following text:

```
INCRYPTOVRKPHYKCKESKRCIPHERKLSOFNOWNKSKCKESKRSCIPHERTHESHIFTCIPHERCKES
```

```
KRSCODEORCKESKRSHIFTISONEOFTHESIMPLESTKNDMOSTWIDELYFNOWNENCRYPTIONTECH
NIZUESITISKTYPEOFSUBSTITUTIONCIPHERINWHICHEKCHLETTERINTHEPLKINTEXTISRE
PLKCEDBYKLETTERSOMEFIXEDNUMBEROFPOSITIONSDOWNTHEKLPHKBETFOREXKMPLEWITH
KSHIFTOFKWOULDBEREPLKCEDBYDBWOULDBECOMEEKNDSOONTHEMETHODISNKMEDKFTERJU
LIUSCKESKRWHOUSEDITTOCOMMUNICKTEWITHHHISVENERKLSTHEENCRYPTIONSTEPPERFOR
MEDBYKCKESKRCIPHERISOFTENINCORPORKTEDKSPKRTOFMORECOMPLEXSCHEMESSUCHKST
HEVIVENRECIPHERKNDSTILLHKSMODERNKPPLICKTIONINTHEROTSYSTEMKSWITHKLLSINV
LEKLPHKBETSUBSTITUTIONCIPHERSTHECKESKRCIPHERISEKSILYBROFENKNDINPRKCTIC
EOFFERSESSENTIKLLYNOCOMMUNICKTIONSECURITY
```

Success Rate (Data): 0.907601

It is evident that the success rate based on the data is higher as compared to the one based on the key. Increasing the number of iterations of the re-estimation algorithm, improves the score.

5.2.2 Simple Substitution Cipher

In the generic case of a Simple Substitution cipher, each English alphabet is associated with only one cipher symbol (one-to-one association). The approach is similar to the Caesar cipher.

Similar to the previous case, the HMM is initially trained using English language, resulting in the matrix A containing the digraph frequencies. Next, the encryption key is randomly generated and used to encrypt the given data set.

The HMM is trained using the matrices A, B and π, where the matrix A was obtained from the previous step. The matrices B and π are initialized with random

values, where the seed value is provided to the Programmable Random Number Generator (PRNG). The matrix A is not modified during the training phase, preserving the digraph frequency statistics for the entire process.

After the training phase, the decryption key is obtained from the matrix B. Varying the seed varies the start point. This technique is useful to improve the decryption process, in case the data set is of limited size.

Multiple experiments were performed for 50, 100 and 200 HMM iterations. For each set of experiments the data size and the number of random restarts are varied and the associated accuracy is calculated. Each subsequent subsection will also include the actual accuracies obtained (in percentage of text correctly deciphered), along with a line chart Success Rate vs Data Size and a 3-dimensional graph (Surface Plot) for Success Rate vs Data Size vs Random Restarts.

5.2.2.1   50 Iterations

This section lists the experiments and the associated result, performed with 50
iterations of HMM re-estimation.

Table 6: Experiment performed on the Brown Corpus [25] for 50 iterations

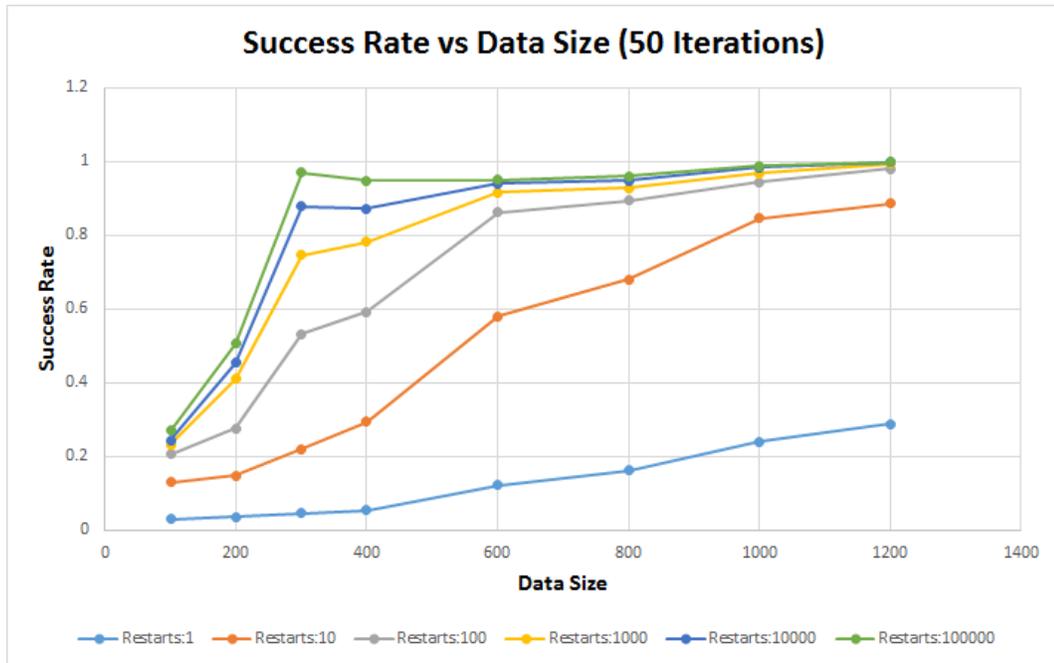| Restarts	Data Size							
	100	**200**	**300**	**400**	**600**	**800**	**1000**	**1200**
1	2.9421	3.5056	4.6116	5.3426	12.1128	16.1381	24.006	28.7762
10	12.9864	14.6582	21.8856	29.4231	57.884	67.9847	84.4543	88.4877
100	20.576	27.563	53.1393	59.1042	86.1152	89.389	94.4393	97.9659
1000	23.17	40.87	74.5833	78.1325	91.6517	92.7263	96.786	99.3358
10000	24.3	45.35	87.8333	87.075	93.95	94.825	98.28	99.75
100000	27	50.5	97	94.75	95	95.875	98.7	99.75

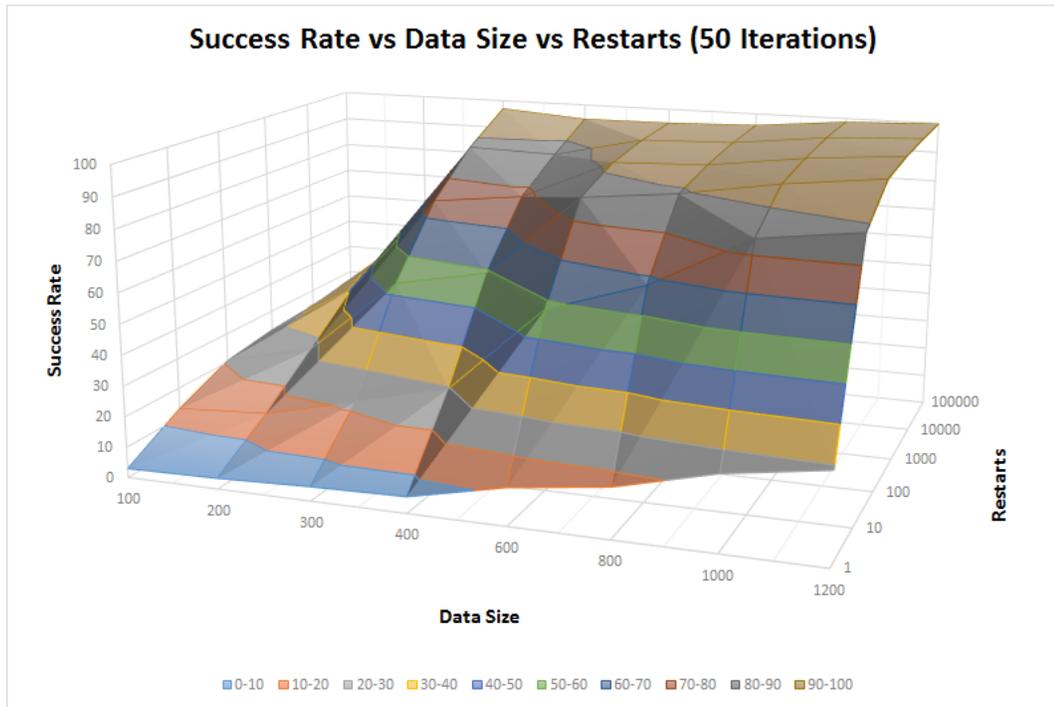Figure 9: Success Rate vs Data Size (50 Iterations)

Figure 10: Success Rate vs Data Size vs Restarts (50 Iterations)
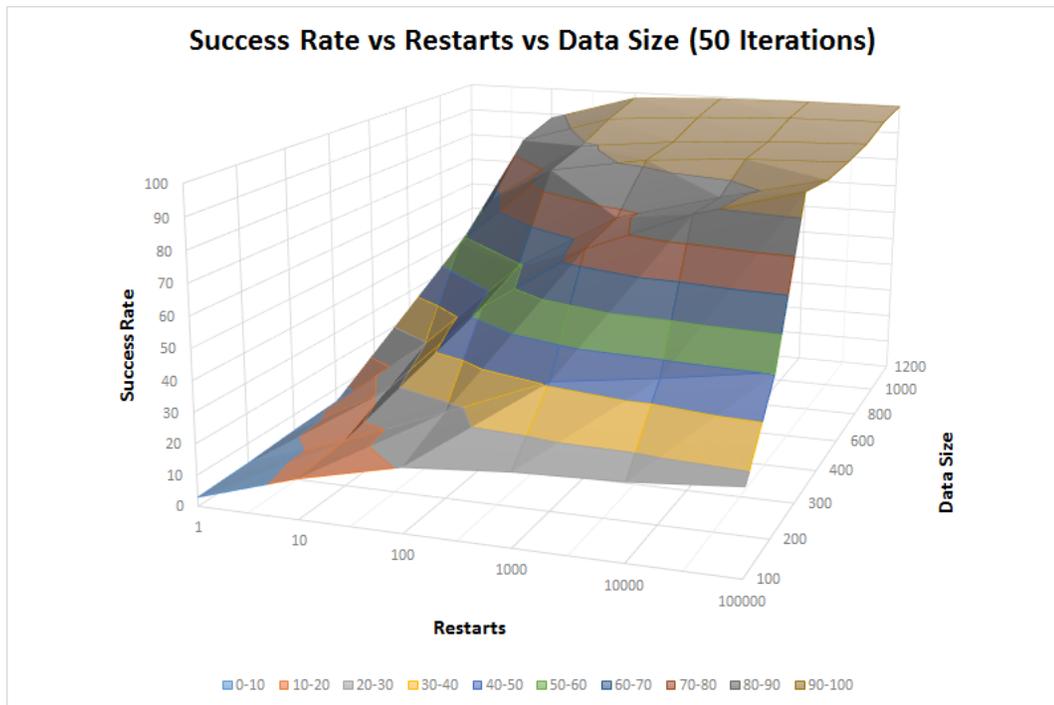

Figure 11: Success Rate vs Restarts vs Data Size (50 Iterations)

5.2.2.2 100 Iterations

This section lists the experiments and the associated result, performed with 100 iterations of HMM re-estimation.

Table 7: Experiment performed on the Brown Corpus [25] for 100 iterations

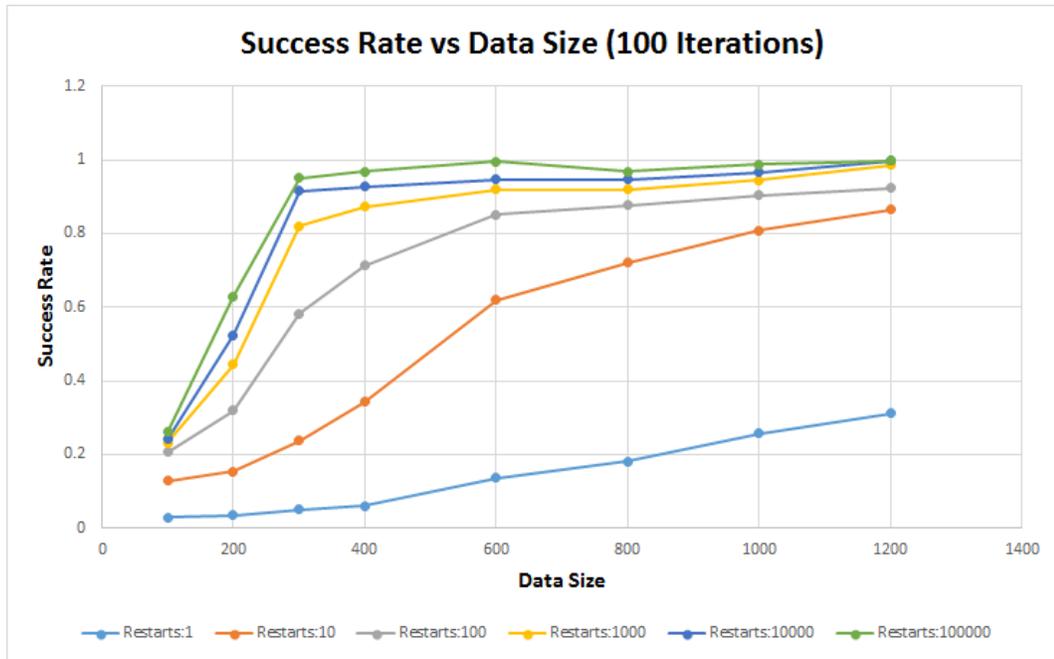| Restarts	Data Size							
	100	200	300	400	600	800	1000	1200
1	2.8597	3.4694	5.0943	6.0071	13.6352	18.1152	25.5932	31.0575
10	12.747	15.3403	23.6893	34.2184	61.8933	72.0557	80.7479	86.4106
100	20.575	31.8775	58.2497	71.216	85.0243	87.6932	90.4134	92.317
1000	23.14	44.37	81.9433	87.345	91.8717	92.0125	94.551	98.6067
10000	24.2	52.45	91.5667	92.625	94.65	94.6125	96.67	99.75
100000	26	63	95	96.75	99.5	96.875	98.7	99.75

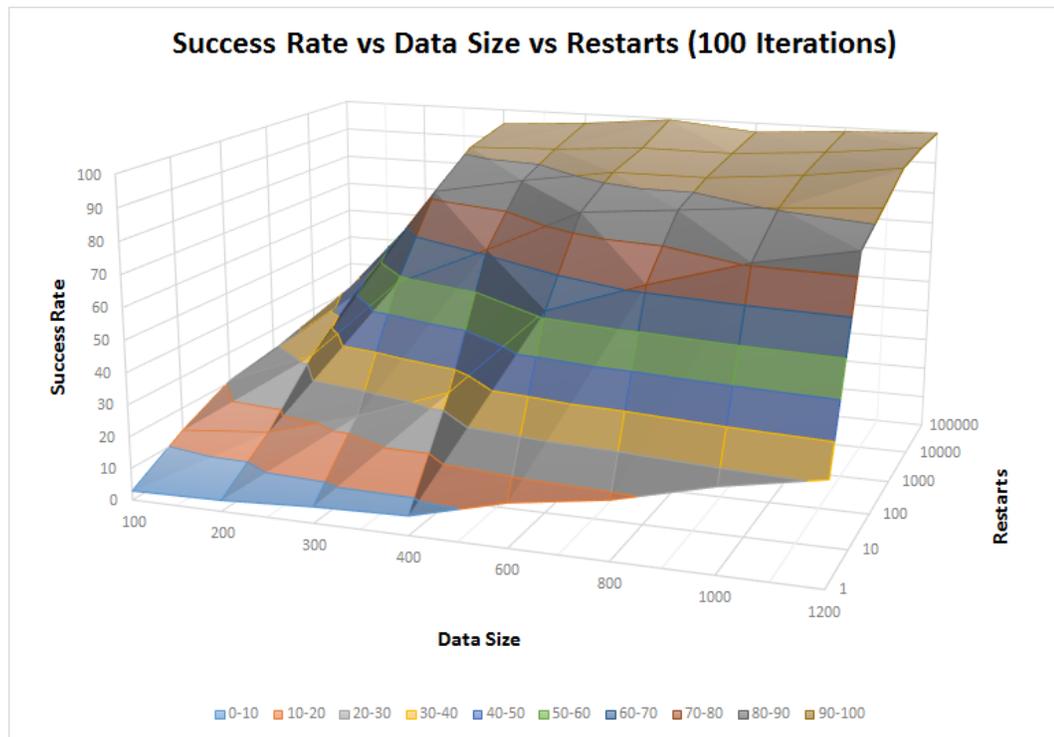Figure 12: Success Rate vs Data Size (100 Iterations)

Figure 13: Success Rate vs Data Size vs Restarts (100 Iterations)

5.2.2.3 200 Iterations

This section lists the experiments and the associated result, performed with 200 iterations of HMM re-estimation.

Table 8: Experiment performed on the Brown Corpus [25] for 200 iterations

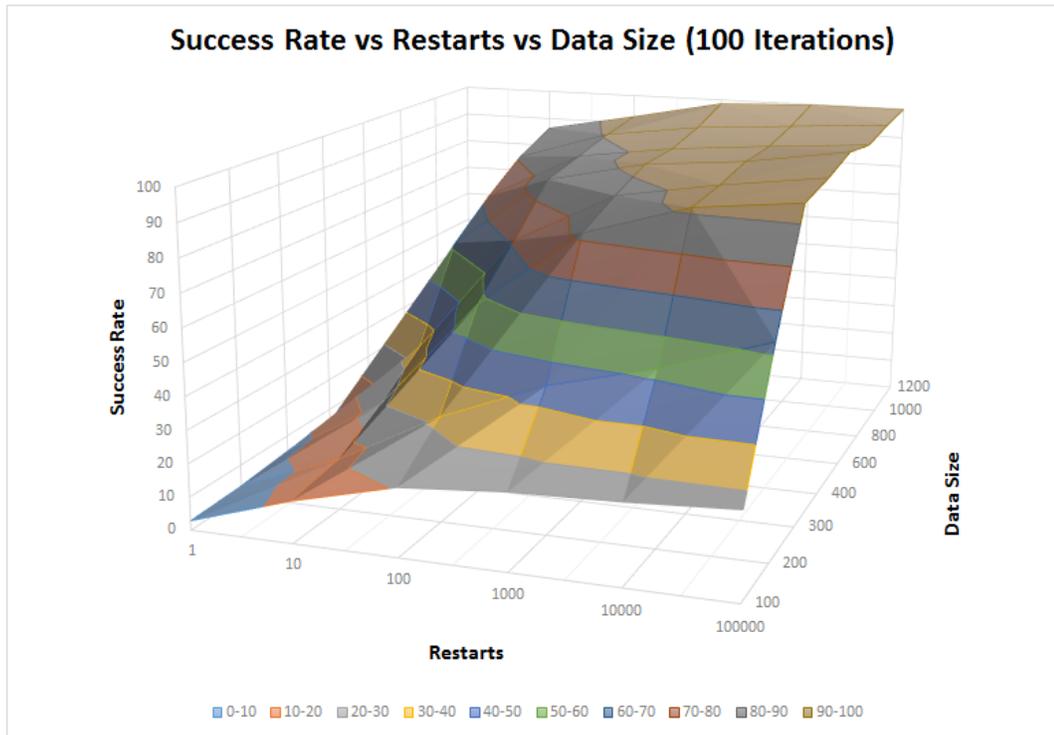| Restarts	Data Size							
	100	200	300	400	600	800	1000	1200
1	2.8019	3.5619	5.2741	6.7028	14.4213	18.6835	28.201	33.2846
10	12.6197	16.2178	24.7003	38.2516	62.9263	71.8961	84.2349	86.8467
100	20.355	34.669	61.7643	77.99	83.3952	84.9175	90.3245	91.8125
1000	23.56	48.21	83.8533	90.5475	89.2917	89.65	94.905	98.07
10000	27	60.1	90.5333	93.575	94.8833	93.1	97.83	99.7417
100000	37	69	94.3333	94.5	99.5	94.375	99.7	99.75

Figure 14: Success Rate vs Restarts vs Data Size (100 Iterations)
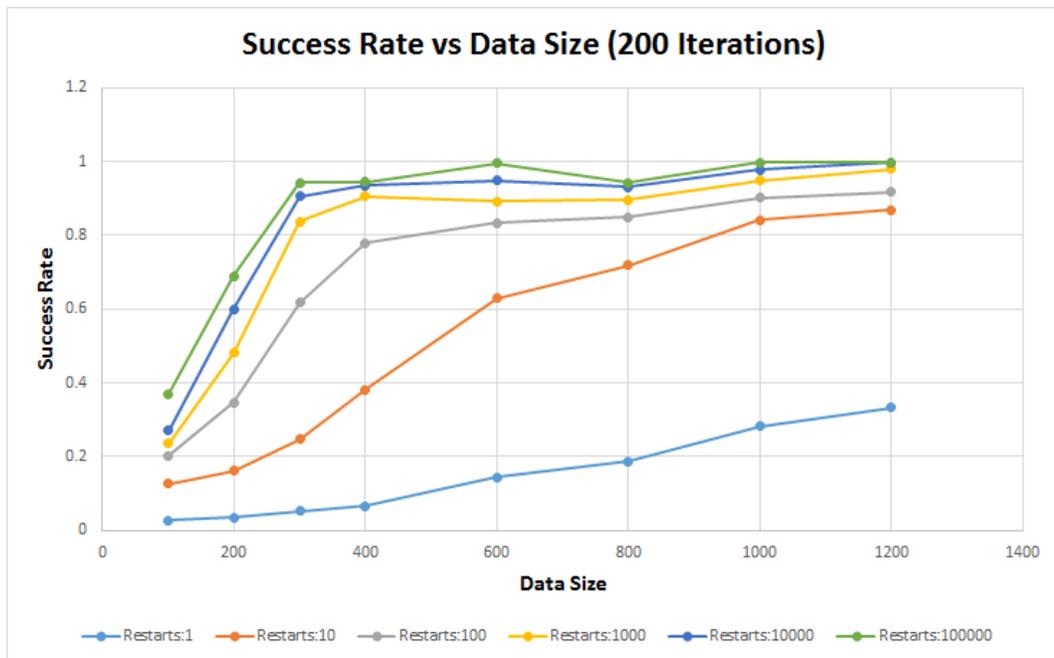

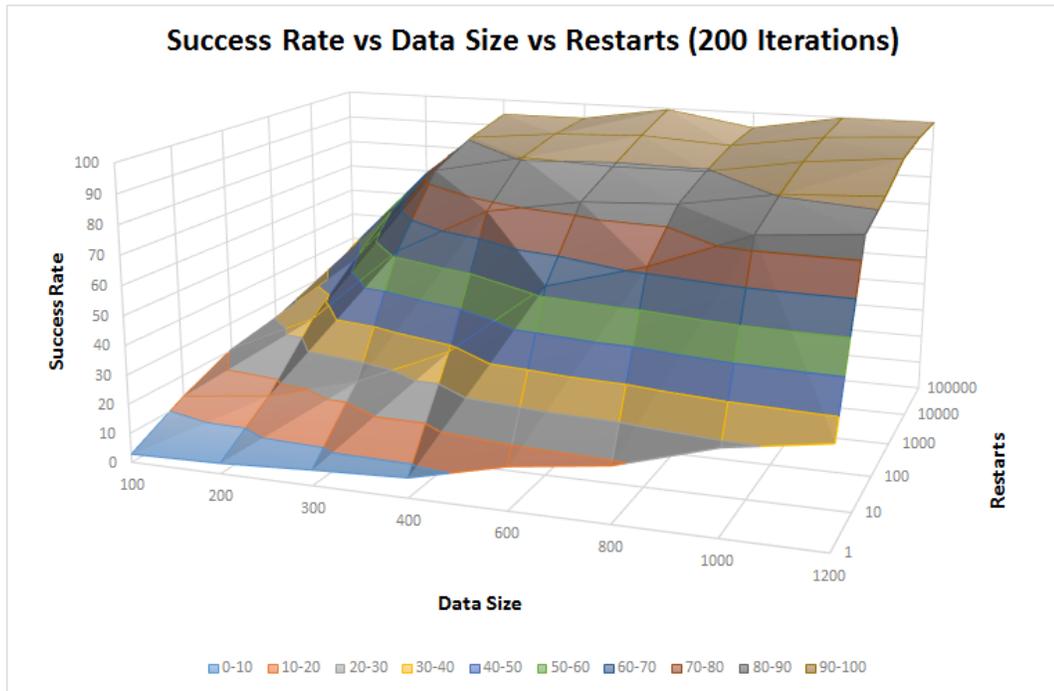Figure 15: Success Rate vs Data Size (200 Iterations)

Figure 16: Success Rate vs Data Size vs Restarts (200 Iterations)
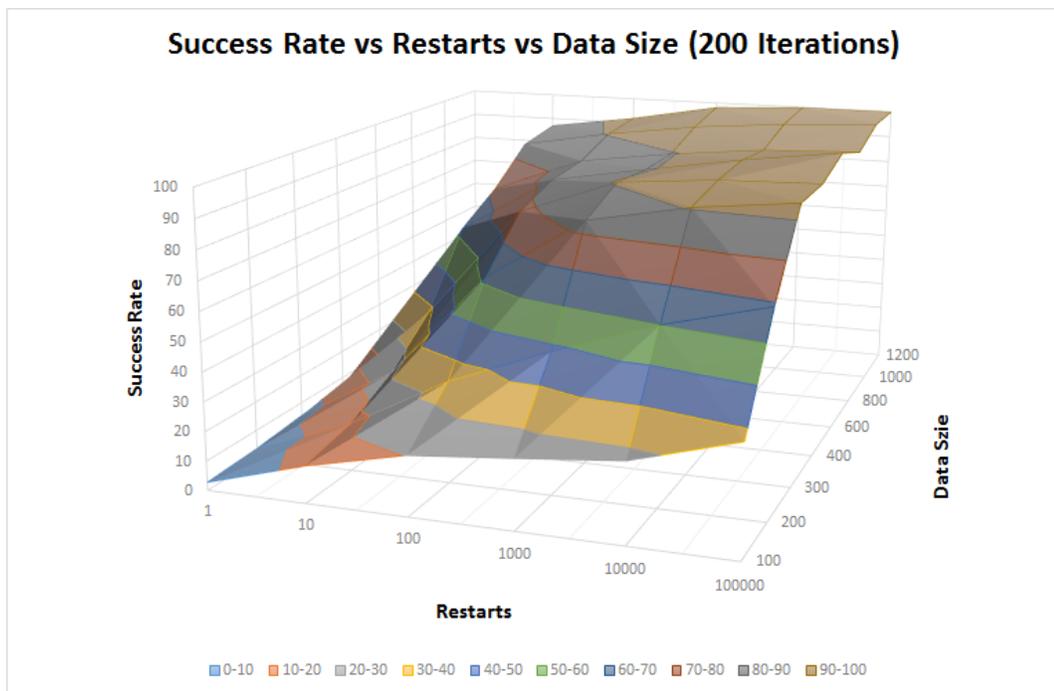

Figure 17: Success Rate vs Restarts vs Data Size (200 Iterations)

5.2.2.4   500 Iterations

This section lists the experiments and the associated result, performed with 200 iterations of HMM re-estimation.

Table 9: Experiment performed on the Brown Corpus [25] for 500 iterations

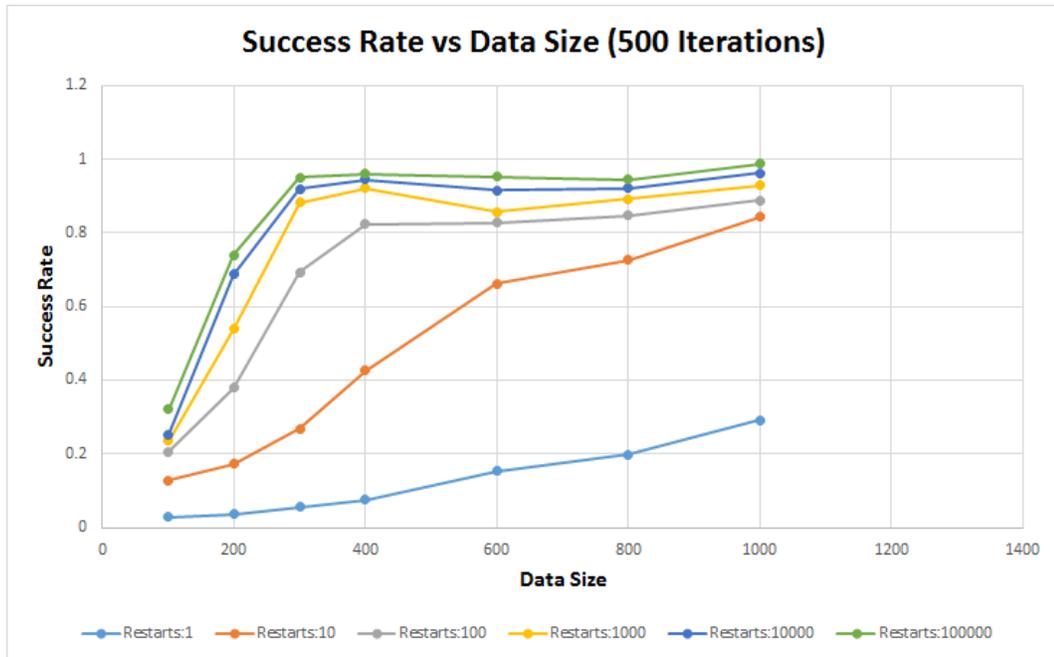| Restarts	Data Size							
	100	200	300	400	600	800	1000	1200
1	2.8299	3.6747	5.5458	7.5169	15.2426	19.7124	29.1063	35.26
10	12.759	17.2218	26.8621	42.5173	66.1846	72.7052	84.237	86.475
100	20.645	37.9955	69.2907	82.4093	82.7338	84.6935	88.7054	92.395
1000	23.42	53.93	88.18	92.19	85.6933	89.1312	92.972	95.8
10000	25.3	68.75	92	94.35	91.6	92.1625	96.18	98.15
100000	32	74	95	96	95.1667	94.5	98.7	99.75

Figure 18: Success Rate vs Data Size (500 Iterations)

The score does improve with increasing data size, since more information can be obtained from a larger dataset. Keeping the data size constant, the score improves
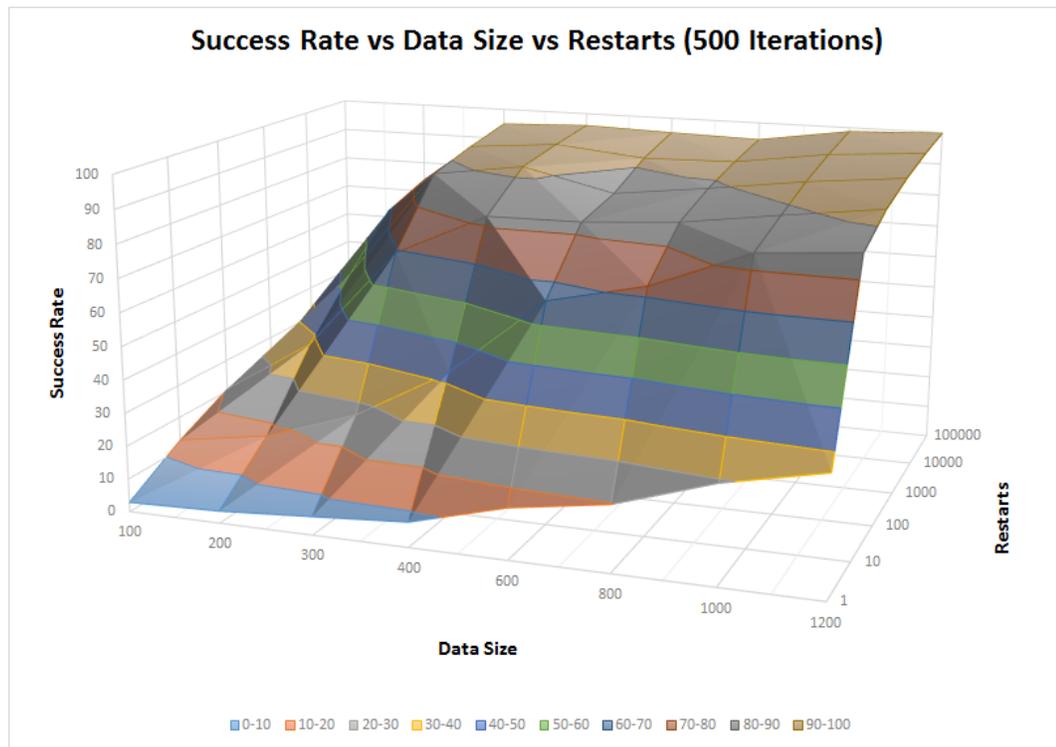
Figure 19: Success Rate vs Data Size vs Restarts (500 Iterations)

with a higher degree by increasing the number of random restarts. Each restart involves a random start point, which might lead to the global maximum.

5.2.3 Homophonic Substitution Cipher

We perform few more experiments for a simple case of the homophonic substitution cipher. The entire process of the experiment is similar to the one performed for the Simple Substitution Cipher. The only difference is the process for encrypting the plain text during the initial part of the experiment. We assume 26 plain text symbols and 29 ciphertext symbols, which implies that atleast one plain text symbol can be substituted with one or more ciphertext symbol.

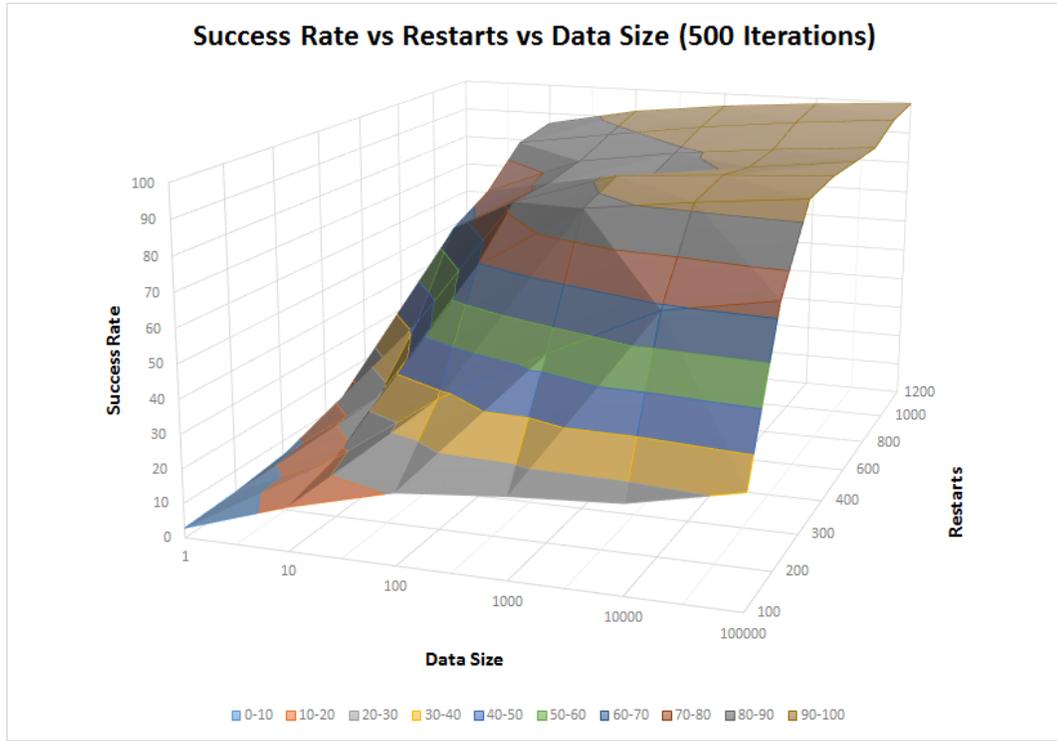Figure 20: Success Rate vs Restarts vs Data Size (500 Iterations)

5.2.3.1 100 Iterations

This section lists the experiments and the associated result, performed with 100 iterations of HMM re-estimation.

Table 10: Experiment performed on the Brown Corpus [25] for 100 iterations (Homophonic substitution cipher)

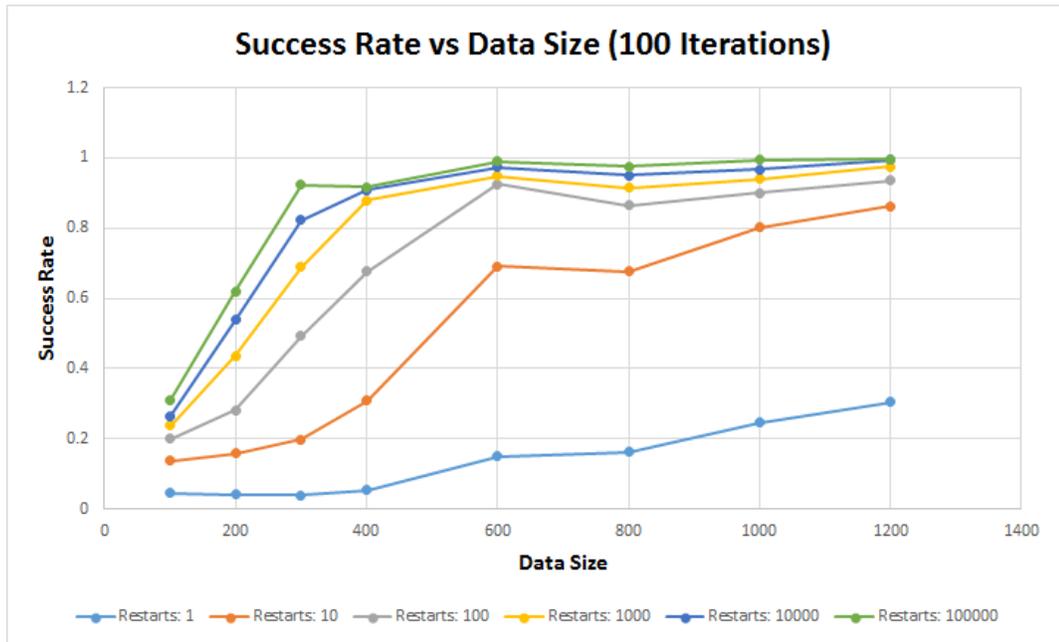| Restarts	Data Size							
	100	200	300	400	600	800	1000	1200
1	4.6078	4.0925	3.957	5.2821	15.0108	16.3368	24.6675	30.5536
10	13.7166	15.7832	19.7143	30.8389	69.1877	67.65	80.1744	86.3479
100	20.004	28.093	49.382	67.6798	92.452	86.4894	90.0111	93.6927
1000	23.78	43.62	68.8367	87.8625	94.795	91.5588	94.082	97.64
10000	26.5	53.95	82.2	90.95	97.3333	94.9625	96.68	99.4083
100000	31	62	92.3333	91.75	99	97.5	99.5	99.75

Figure 21: Success Rate vs Data Size (100 Iterations)

Figure 22: Success Rate vs Data Size vs Restarts (100 Iterations)
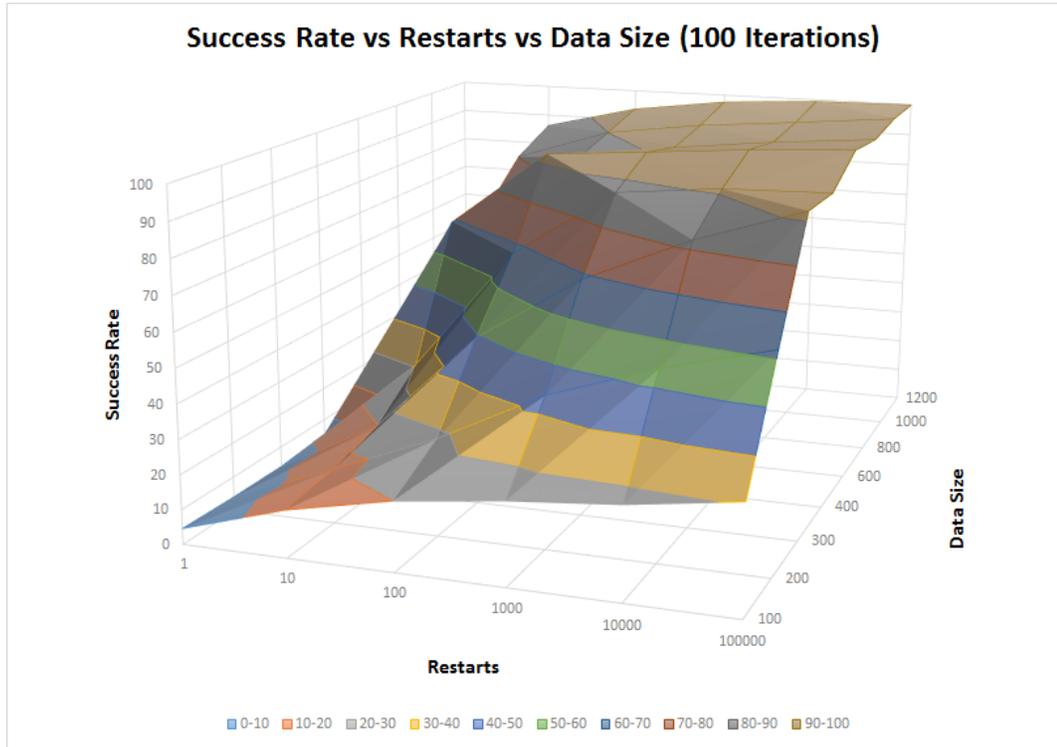
Figure 23: Success Rate vs Restarts vs Data Size (100 Iterations)

5.2.4 Real World Case

In an actual scenario, only the ciphertext is available and we assume the values for N and M i.e. the number of plaintext and ciphertext symbols. There would be no actual plaintext available to evaluate this technique, or in other words calculate the accuracy. One interesting parameter to consider is the probability $P(\mathcal{O}|\lambda)$ once the model has been trained.

We perform few experiments for a fixed data size and HMM iterations over a random selection of text encrypted using Simple Substitution cipher. This experiments is performed for 1000 random restarts. Then we analyze the relationship between the probability $P(\mathcal{O}|\lambda)$ and the Success Rate to identify the trend between the two parameters. Remaining results can be found in Section A.2 of the Appendix.
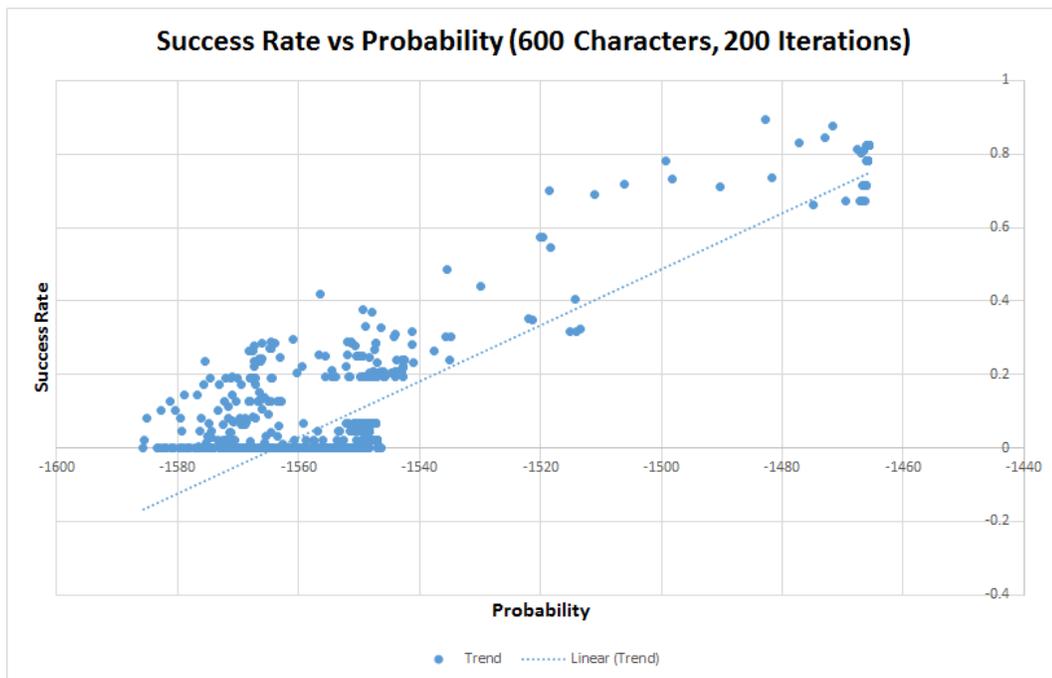
Figure 24: Success Rate vs Probability (600 Characters, 200 Iterations)

It is evident from the scatter plot and the trend line that there is a linear relationship between the Success Rate and probability $P(\mathcal{O}|\lambda)$, which might help to choose the model that would yield the highest accuracy.

5.3 Baum-Welch Algorithm (GPU)

In this section we compare the results obtained from GPU version with CPU version of Baum-Welch algorithm.

5.3.1 Performance Analysis (GPU vs CPU)

The GPU version of the program involves lots of memory transfers between the CPU and the GPU. The performance degrades for lower number of iterations, while it improves for a higher number of iterations and length of observation sequence. This is evident from the fact that the computation is distributed among several processors

of the GPU, and noticeable for large computations.

We performed basic performance tests for the CPU and GPU version of the program and compute the execution ratio. The dataset used is "Brown Corpus" [25], with varying size of ciphertext and varying HMM iterations.

$$\text{Execution Ratio} = \frac{\text{Time taken on GPU}}{\text{Time taken on CPU}}$$
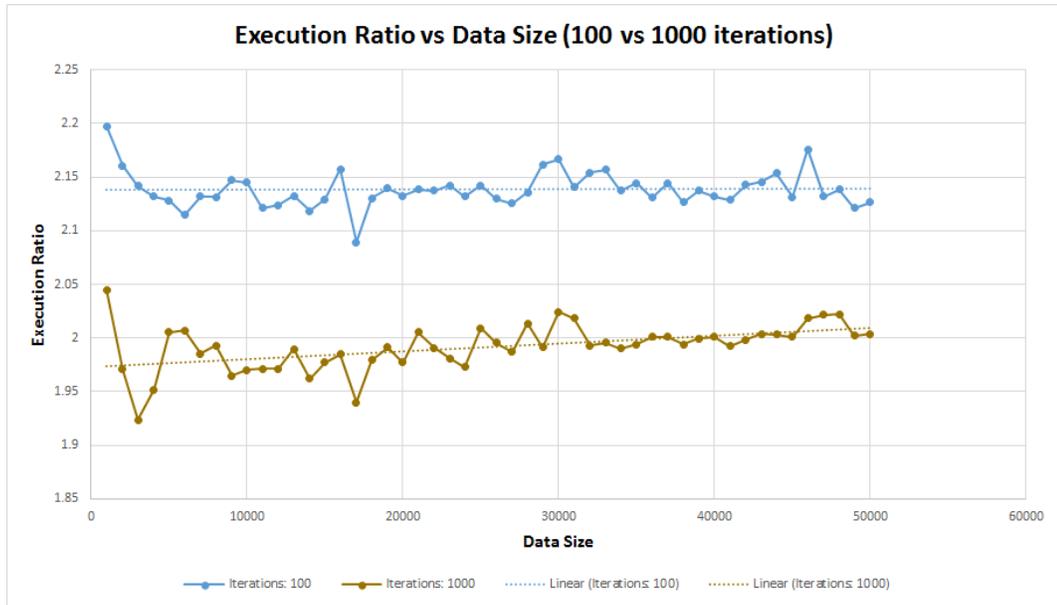
Figure 25: Execution Ratio vs Data Size (100 vs 1000 Iterations)

Figure 25 compares the Execution Ratio against Data Size for the Baum-Welch algorithm. There is a clear distinction between the Trend Lines (Linear) for each figure, and we conclude that the performance improves for a larger input and iterations.

Figure A.29 compares the Execution Ratio against Data Size for multiple HMM iterations. The ratio drops for a higher number of HMM iterations, depicting that the processing is distributed among multiple thread of the GPU. Section A.3 lists additional results.

5.4   Challenges

The concept of Hidden Markov Models with random restarts proves to be very efficient in case of decipherment problems, even in cases where the ciphertext available is of limited size. The seed for the Pseudo Random Number Generator was chosen randomly using several sources of randomness or entropy, a good example would be the System Time. Increasing the number of random restarts, significantly improves the accuracy where compared with the results by increasing the HMM iterations. This experiment in itself is computationally intensive, and we try to perform this experiment on a Graphics Processing Unit. To improve efficiency of the GPU experiment, the implementation was optimized to efficiently distribute the processing among multiple processors (or threads) within the GPU.

CHAPTER 6

Conclusion

In case of the Jakobsen's Fast Attack, the accuracy improves with increasing data size. With limited data available, the accuracy is very low due to insufficiency of language statistics for the input ciphertext. With increasing data size, the digraph statistics is closer to the English language statistics.

We performed several experiments for HMM with random restarts, by extracting and encrypting random sections of the "Brown Corpus" [25]. Each set of experiments deals with a specific value for HMM iterations and varying data size and random restarts. The results obtained from these experiments are impressive and the accuracy improves significantly for increasing random restarts. This process is computationally intensive, and we implemented the solution on a NVIDIA GPU. This experimental setup is slower when compared with the single threaded CPU version, limited by the GPU hardware's kernel allocation and the availability of resources. To conclude, this technique is useful even for a limited amount of ciphertext.

CHAPTER 7

Enhancements and Future Work

The existing experiment for Hidden Markov Models is successful for Simple Substitution cipher, and should be extended to Homophonic Substitution ciphers like the Purple cipher [8].

The experiment using Hidden Markov Models with random restarts is computationally intensive, and needs to be enhanced for faster execution. The existing solution on the GPU should allow the GPU to initialize the matrices of the HMM. The cuRAND [18] library (implemented on CUDA) can be used to generate random seeds and initialize the matrices of HMM with random values. Multiple instances of the Baum-Welch algorithm can be executed on the GPU, to simulate the concept of HMM with random restarts. This technique would be very efficient, as the GPU consists of thousands of cores and can operate in SIMD (Single Instruction Multiple Data) mode.

Finally, we can also utilize the Hadoop ecosystem [3] to spawn multiple instances of the Hidden Markov Model, each instance seeds a separate Pseudo Random Number Generator with a random value, simulating the concept of HMM with random restarts. The underlying architecture would consist of several nodes to support a higher number of HMM instances executing simultaneously. Each phase of the HMM can be reimplemented using Map-Reduce [3] technique for faster execution.

LIST OF REFERENCES

[1] Amazon EC2 instances, Amazon Web Services, 2015
   `http://aws.amazon.com/ec2/instance-types/`

[2] T. Berg–Kirkpatrick and D. Klein, Decipherment with a Million Random
   Restarts, *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2013

[3] M. Bhandarkar, MapReduce programming with apache Hadoop, *Parallel and Distributed Processing (IPDPS), 2010 IEEE International Symposium*, 1–1, 2010

[4] J. Borghoff, L. R. Knudsen and K. Matusiewicz, Hill climbing algorithms and
   trivium, *Selected Areas in Cryptography*, 57–73, 2011

[5] T. Dao, Analysis of the Zodiac 340–cipher, Doctoral dissertation, Department
   of Computer Science, San Jose State University, 2007
   `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.6265&`
   `rep=rep1&type=pdf`

[6] A. Dhavare, R. M. Low and M. Stamp, Efficient cryptanalysis of homophonic
   substitution ciphers, *Cryptologia*, 37(3):250–281, 2013

[7] Digraph Frequencies (based on a sample of 40,000 words)
   `http://www.math.cornell.edu/~mec/2003-2004/cryptography/subs/`
   `digraphs.html`

[8] W. Freeman, G. Sullivan and F. Weierud, Purple Revealed: Simulation and
   Computer–Aided Cryptanalysis of Angooki Taipu B, *Cryptologia*, 29:193–232,
   2005

[9] S. R. Hymel, Massively Parallel Hidden Markov Models for Wireless Applications, Doctoral dissertation, Virginia Polytechnic Institute and State University,
   2011
   `http://scholar.lib.vt.edu/theses/available/etd-12082011-204951/`
   `unrestricted/Hymel_SR_T_2011.pdf`

[10] T. Jakobsen, A Fast Method for the Cryptanalysis of Substitution Ciphers,
   Cryptologia, 19(3):265–274, 1995

[11] K. Krewell, What's the Difference Between a CPU and a GPU, 2009
   `http://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-`
   `between-a-cpu-and-a-gpu`

[12] C. Liu, cuHMM: a CUDA Implementation of Hidden Markov Model Training
and Classification, Johns Hopkins University, 2009
`https://liuchuan.org/pub/cuHMM.pdf`

[13] I. V. S. Manoj, Cryptography and Steganography, *International Journal of Computer Applications*, 1(12):975–8887, 2010

[14] M. Nuhn and H. Ney, EM Decipherment for Large Vocabularies, *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, 2014

[15] NVIDIA CUDA Compiler Driver, NVIDIA corporation, 2015
`http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc`

[16] NVIDIA CUDA (Compute Unified Device Architecture) Programming Guide,
NVIDIA corporation, 2015
`http://docs.nvidia.com/cuda/cuda-c-programming-guide`

[17] NVIDIA CUDA cuBLAs Library, NVIDIA corporation, 2015
`http://docs.nvidia.com/cuda/cublas`

[18] NVIDIA CUDA cuRand Library, NVIDIA corporation, 2010
`http://docs.nvidia.com/cuda/curand`

[19] NVIDIA GRID GPU Specifications and Features, NVIDIA corporation, 2015
`http://www.nvidia.com/object/cloud-gaming-gpu-boards.html`

[20] OpenCL, The open standard for parallel programming of heterogeneous systems,
Khronos group, 2015
`https://www.khronos.org/opencl`

[21] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips,
GPU Computing, *Proceedings of the IEEE*, 96(5):879–899, 2008

[22] S. M. Siddiqi, G. J. Gordon and A. W. Moore, Fast State Discovery for HMM
Model Selection and Learning, *In International Conference on Artificial Intelligence and Statistics*, 492–499, 2007

[23] M. Stamp, A Revealing Introduction to Hidden Markov Models, 2012
`http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf`

[24] M. Stamp and R. M. Low, Applied Cryptanalysis: Breaking Ciphers in the Real
World, John Wiley and Sons, 1–24, , 2007
`http://www.wiley.com/WileyCDA/WileyTitle/productCd-047011486X.html`

[25] The Brown Corpus of Standard American English, available for download at,
`http://www.cs.toronto.edu/~gpenn/csc401/a1res.html`

[26] What is GPU Computing, NVIDIA corporation, 2015
http://www.nvidia.com/object/what-is-gpu-computing.html

[27] T. Williams, Review of A* and Hill Climbing Algorithms, 2013
http://trevoirwilliams.com/review-of-a-hill-climbing-algorithms

[28] J. Yi, Cryptanalysis of Homophonic Substitution–Transposition Cipher, Master's
report, Department of Computer Science, San Jose State University, 2014
http://scholarworks.sjsu.edu/etd_projects/357

[29] L. Yu, Y. Ukidave and D. Kaeli, GPU–accelerated HMM for Speech Recognition, *In Heterogeneous and Unconventional Cluster Architectures and Applications Workshop, IEEE*, 2014

[30] C. Zeller, CUDA C/C++ Basics, NVIDIA corporation, 2011
http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf

Additional Experiment Results

A.1 Digraph frequencies

Table A.11: Digram statistics for 40,000 words of English Language [7]

Digraph	Count	Frequency
th	5532	1.52
he	4657	1.28
in	3429	0.94
er	3420	0.94
an	3005	0.82
re	2465	0.68
nd	2281	0.63
at	2155	0.59
on	2086	0.57
nt	2058	0.56
ha	2040	0.56
es	2033	0.56
st	2009	0.55
en	2005	0.55
ed	1942	0.53
to	1904	0.52
it	1822	0.5
ou	1820	0.5
ea	1720	0.47
hi	1690	0.46
is	1660	0.46
or	1556	0.43
ti	1231	0.34
as	1211	0.33
te	985	0.27
et	704	0.19
ng	668	0.18
of	569	0.16
al	341	0.09
de	332	0.09
se	300	0.08
le	298	0.08
sa	215	0.06
si	186	0.05
ar	157	0.04
ve	148	0.04
ra	137	0.04
ld	64	0.02
ur	60	0.02

Table A.12: Digram frequency counts for English language (Brown Corpus)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	2442	10184	19546	17816	826	5038	9086	2170	14296	906	4663	38956	13142	73763	1287	9254	228	41185	38908	56592	4443	8477	4535	796	9953	641
B	7168	1050	437	308	22347	322	133	373	4372	685	45	8970	420	186	8723	314	10	4405	1923	1835	8189	245	511	0	5785	2
C	20212	458	3038	379	22950	459	276	22035	9447	323	6318	5786	404	245	28518	564	170	5620	1485	15170	4773	111	427	0	1209	22
D	17454	7455	4069	4743	30313	5510	4430	6842	23950	2601	1285	4447	4228	4708	13739	4098	243	5988	11180	17346	5902	1097	5519	2	3006	27
E	46521	10162	28160	54267	22653	16667	11183	11434	17745	4874	3294	26151	21874	57685	15937	16956	2160	82335	62160	36543	3685	11301	17399	6922	7820	233
F	11912	1598	2707	1394	9945	7274	1370	2922	13165	909	290	3612	2117	1043	20480	2091	60	9002	3285	18039	4073	365	2016	3	851	17
G	11854	2085	1898	1248	15114	2346	2345	12759	8796	993	418	3460	1834	3278	9179	1909	81	8596	4376	9099	3410	250	2463	3	861	9
H	42173	1379	1782	961	122755	1465	925	2053	34380	623	289	1248	1765	1736	21653	1415	74	4193	2668	11508	3255	209	1900	0	1952	24
I	9234	3380	24549	13166	13311	7388	10245	696	442	126	2419	17446	12929	88829	25697	3429	451	11992	41885	41646	464	9538	905	743	21	2347
J	3327	825	955	580	2175	905	223	521	2000	148	47	347	602	335	3894	796	29	669	1264	3269	2673	148	1108	6	50	6
K	2788	731	538	368	10644	789	434	1354	5150	213	248	928	512	2637	1664	470	19	389	2757	2221	278	67	1159	2	592	1
L	22601	2699	2436	12829	32976	4080	1546	2176	25076	1017	1426	24810	2366	1104	16789	2698	85	1642	7428	7777	5067	1349	2235	1	17561	27
M	22979	4297	772	456	30052	928	589	1123	13199	449	196	622	3906	848	14077	8314	23	1740	4465	3491	4844	91	1106	2	2322	5
N	23376	4067	17545	50950	30605	5906	40663	6139	18434	2730	2987	4609	3737	5232	23142	3239	293	2198	22515	56125	3660	2039	5005	101	4914	134
O	6488	6617	7801	8779	2995	40863	4528	3543	4748	811	3550	14581	22505	62025	11429	10051	128	47511	13932	22633	36553	7318	15052	503	1924	201
P	12957	598	421	303	17518	561	286	4008	6103	180	122	9790	1077	310	13258	5596	21	15548	2801	5025	3813	43	847	0	533	4
Q	10	4	2	0	2	5	0	3	13	7	0	13	1	5	5	4	0	2	2	8	4962	1	7	0	0	0
R	30518	3487	6907	8950	68986	4012	5268	4218	28115	1414	4239	5456	8275	7536	30725	4335	141	5685	19926	21737	5227	2611	3533	21	9122	51
S	28100	5914	10758	3716	35928	7180	4412	19604	28083	3827	2953	5577	6522	5018	25213	11377	587	2988	21025	55617	11720	707	8645	28	2321	38
T	28783	4420	4992	2654	45309	4323	2662	139354	51051	2017	1078	6823	4275	2511	48267	3300	148	16478	17481	21490	9490	401	8961	3	8482	167
U	4716	3385	6525	3532	4976	825	5527	334	3929	60	230	13123	4855	15706	484	5559	11	18753	17159	16704	62	171	384	176	237	117
V	4316	36	25	17	30507	14	8	14	8991	14	9	9	26	10	2382	17	1	41	103	28	96	9	37	0	226	1
W	19464	464	434	484	14397	377	235	15285	15459	178	149	787	526	3737	9805	425	14	1448	1658	1483	194	59	539	0	544	4
X	1030	56	986	40	670	101	41	182	1136	52	17	42	69	25	231	2466	7	68	104	1625	107	21	128	7	115	1
Y	7881	3382	3093	2239	6144	3185	2225	3221	4681	1740	664	2092	2903	1831	10258	3037	73	2006	7342	7880	568	299	3700	5	369	64
Z	828	25	25	3	2023	17	25	22	517	5	17	116	26	2	233	9	0	13	26	29	32	10	28	0	112	347

65
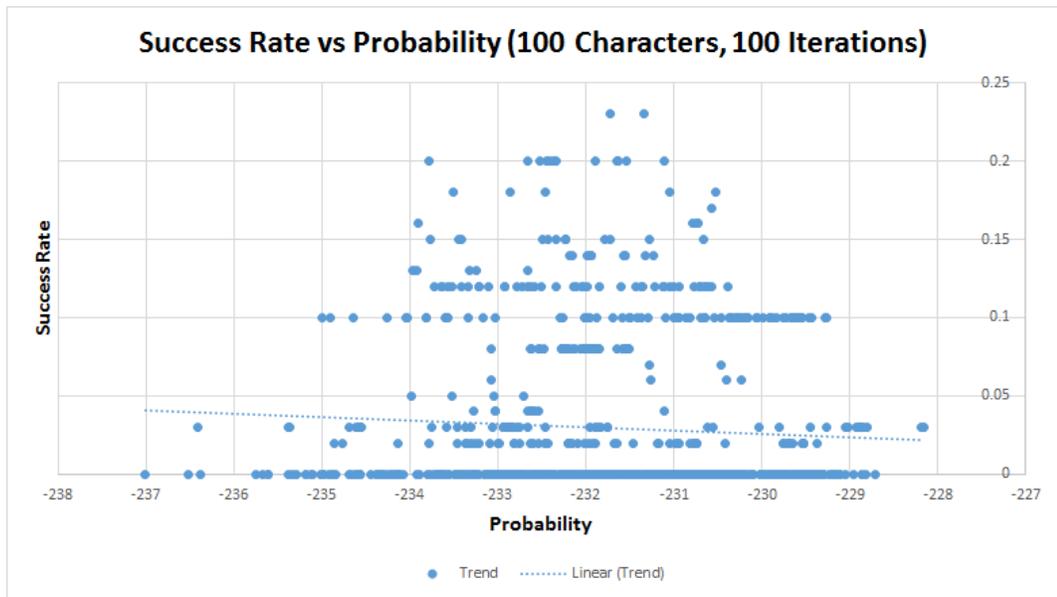
A.2   Success Rate vs Model Probability

Figure A.26: Success Rate vs Probability (100 Characters, 100 Iterations)
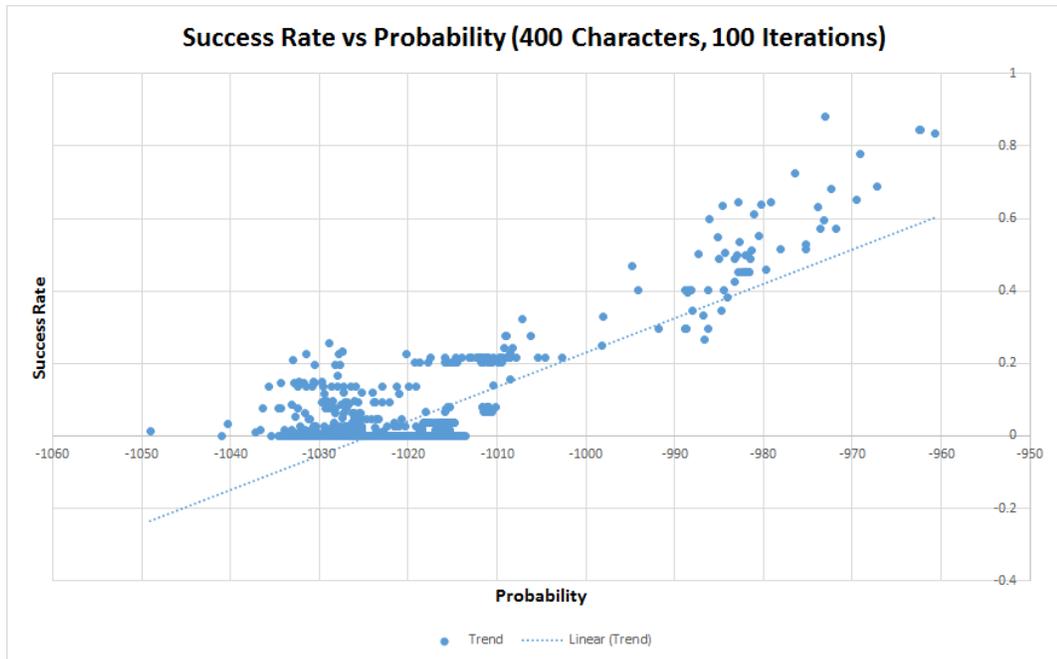

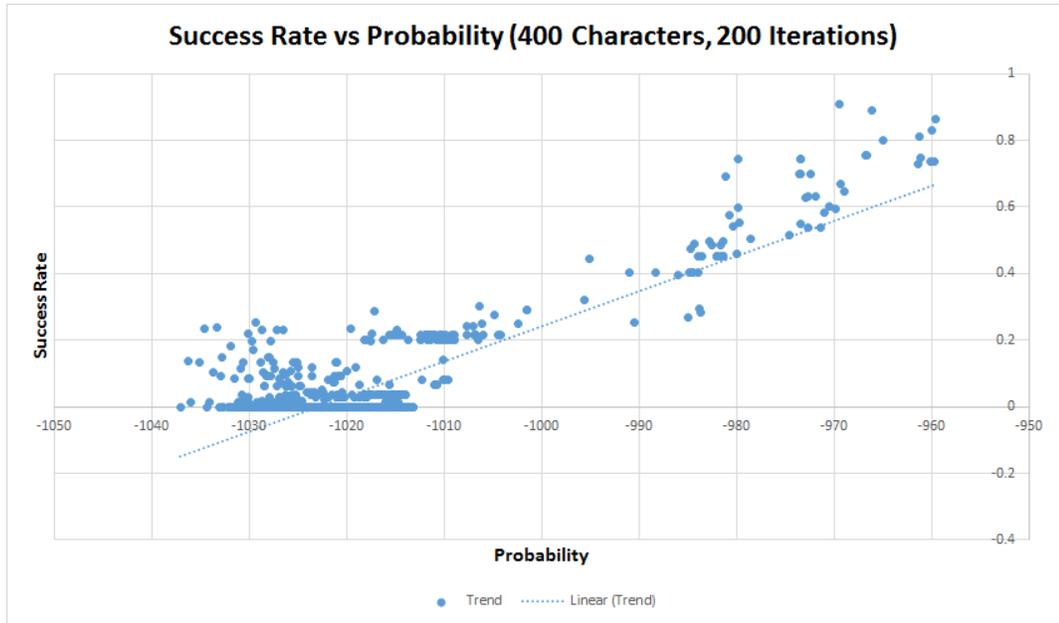Figure A.27: Success Rate vs Probability (400 Characters, 100 Iterations)

Figure A.28: Success Rate vs Probability (400 Characters, 200 Iterations)
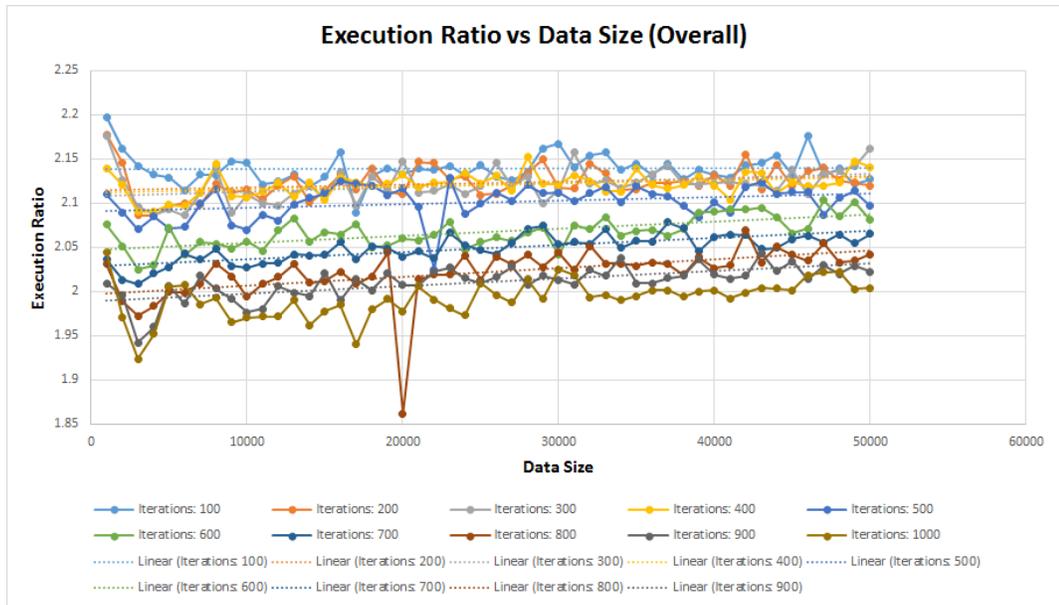
A.3 Additional results (GPU vs CPU)
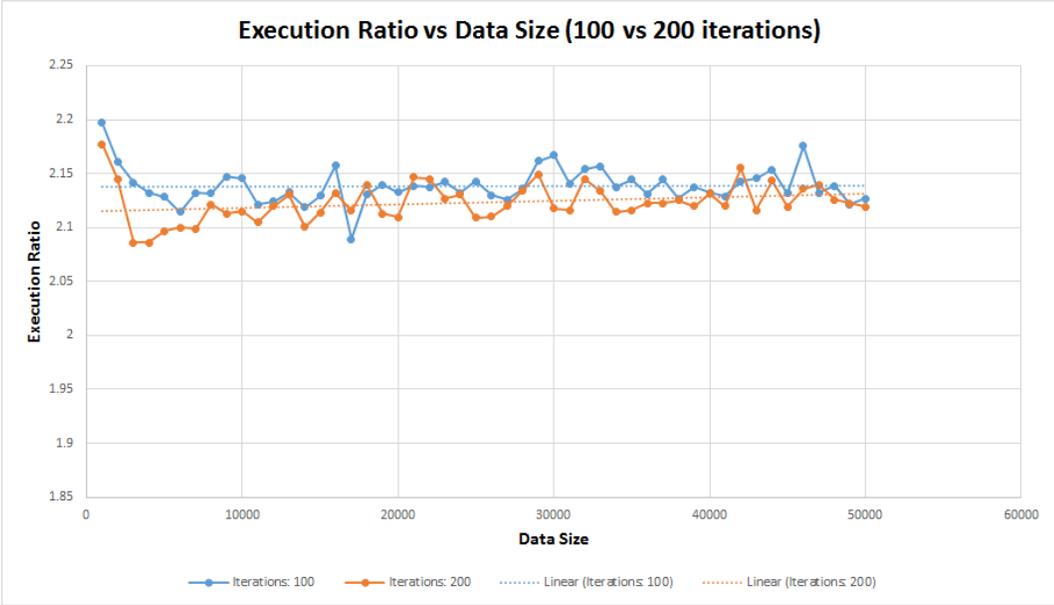

Figure A.29: Execution Ratio vs Data Size (Overall)

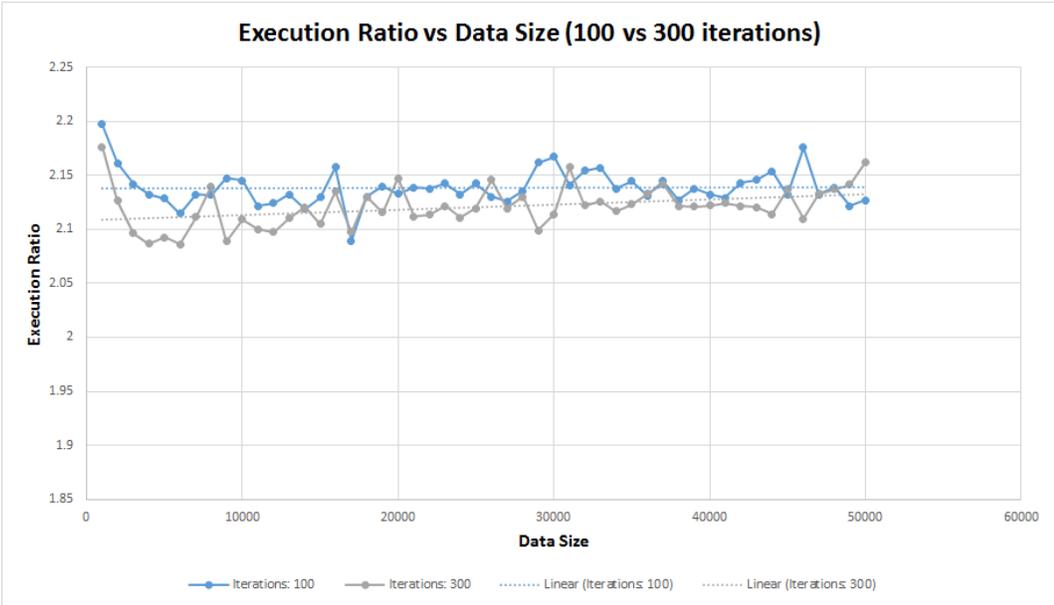Figure A.30: Execution Ratio vs Data Size (100 vs 200 Iterations)
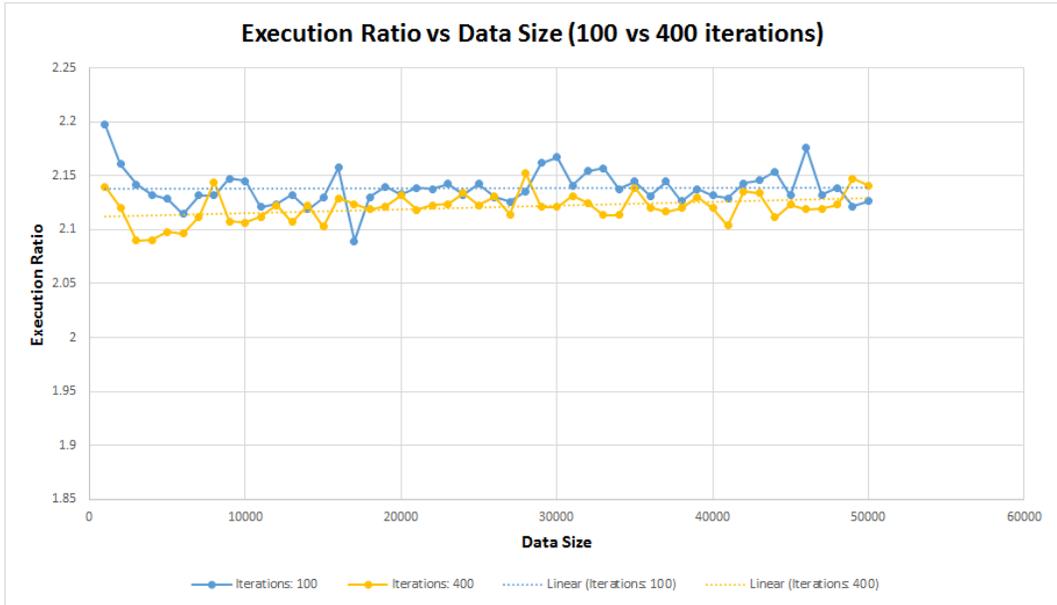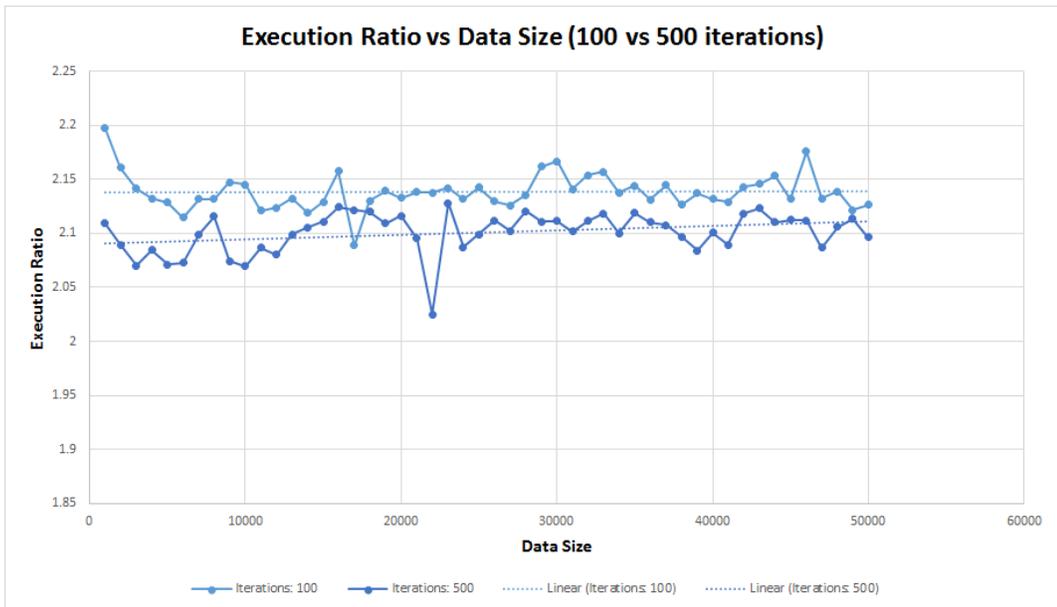

Figure A.31: Execution Ratio vs Data Size (100 vs 300 Iterations)

Figure A.32: Execution Ratio vs Data Size (100 vs 400 Iterations)

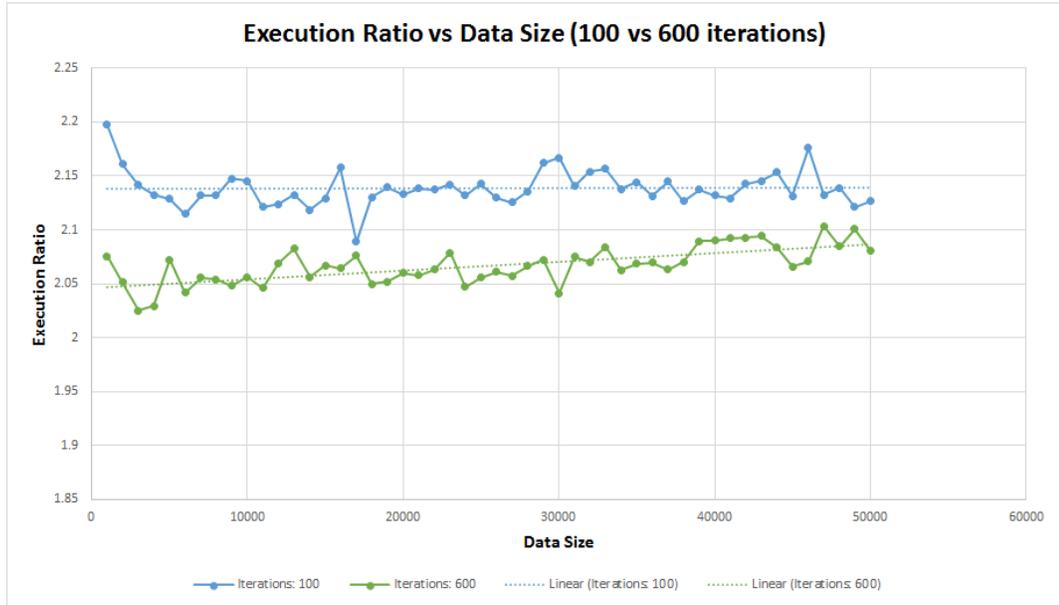Figure A.33: Execution Ratio vs Data Size (100 vs 500 Iterations)

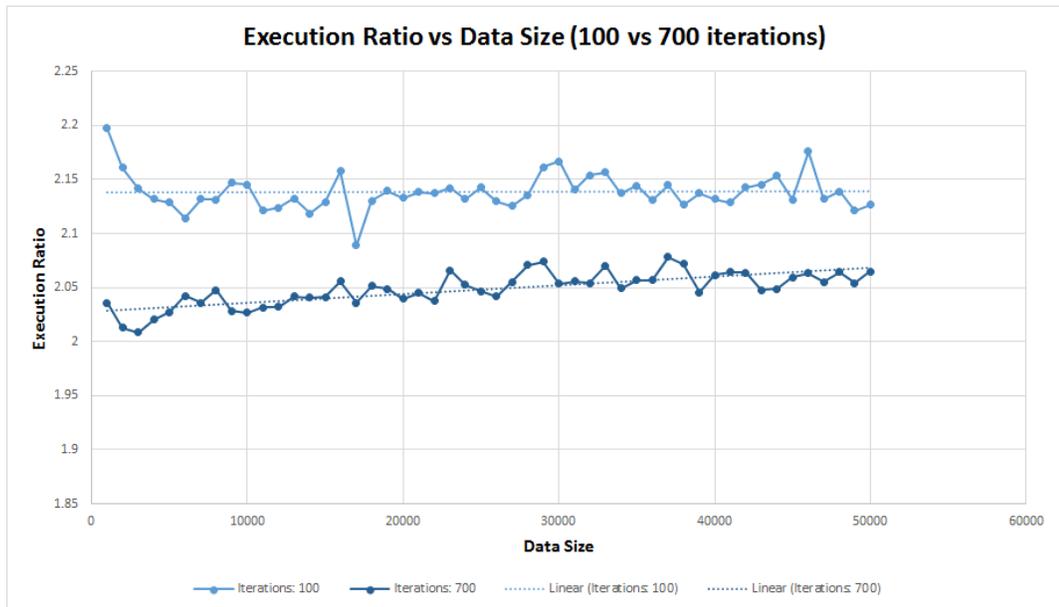Figure A.34: Execution Ratio vs Data Size (100 vs 600 Iterations)

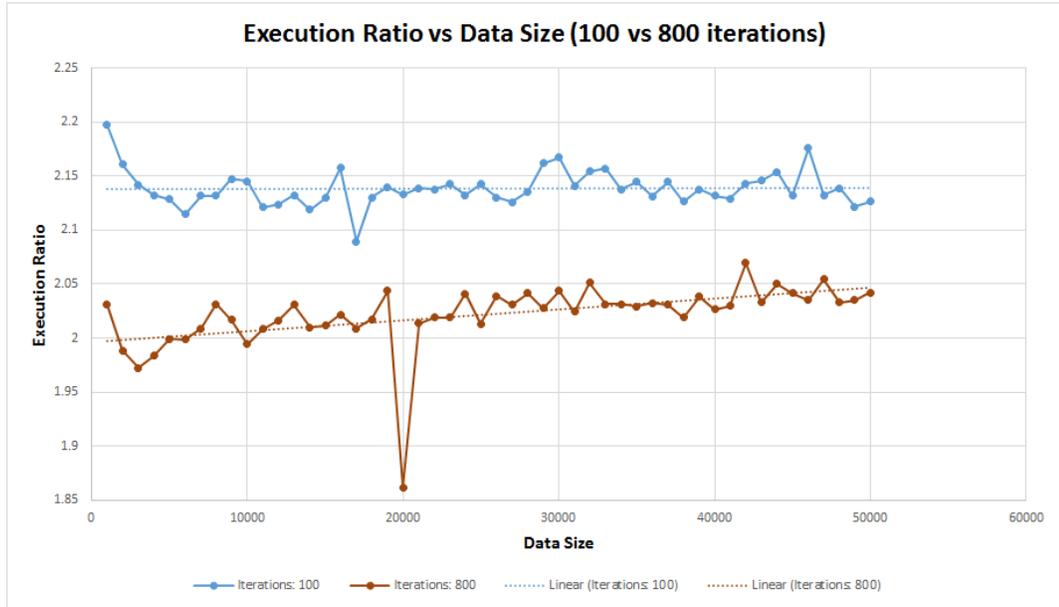Figure A.35: Execution Ratio vs Data Size (100 vs 700 Iterations)

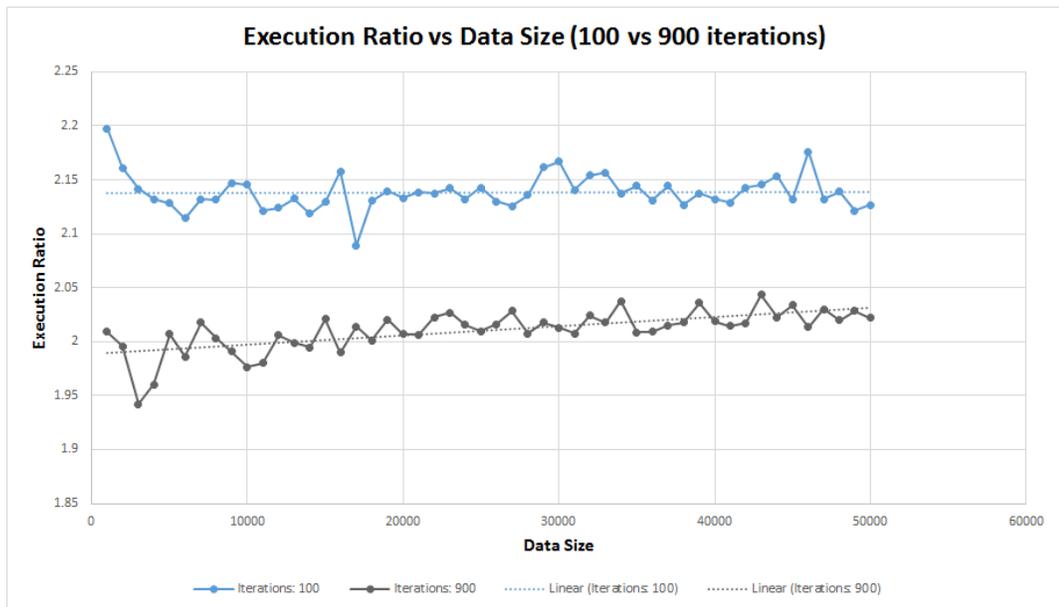Figure A.36: Execution Ratio vs Data Size (100 vs 800 Iterations)

Figure A.37: Execution Ratio vs Data Size (100 vs 900 Iterations)