An Eulerian Approach to N-Gram Table Conversion

Aaron Olson

Technical Report CS-2005-22

December 5, 2005

Faculty of Computer Science
6050 University Ave., Halifax, Nova Scotia, B3H 1W5, Canada

AN EULERIAN APPROACH TO *N*-GRAM TABLE CONVERSION

by
Aaron Olson

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF COMPUTER SCIENCE WITH HONOURS

AT

DALHOUSIE UNIVERSITY
HALIFAX, NOVA SCOTIA
AUGUST 16, 2004

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Computer Science for acceptance a thesis entitled "**An Eulerian Approach to n-gram Table Conversion**" by **Aaron Olson** in partial fulfillment of the requirements for the degree of **Bachelor of Computer Science with Honours**.

Dated: August 16, 2004

Supervisor: Vlado Kešelj

Reader: Pat Keast

DALHOUSIE UNIVERSITY

Date: **August 16, 2004**

Author: **Aaron Olson**

Title: **An Eulerian Approach to n-gram Table Conversion**

Faculty: **Computer Science**

Degree: **B.C.Sc.** Convocation: **October** Year: **2004**

Table of Contents

List of Tables

List of Figures

Abstract

In natural language processing, sequences of n characters or words known as *n-grams* are used for authorship attribution, text clustering, and other types of text analysis. Given a text corpus and a word n-gram table, we examine the problem of generating tables of character n-grams from tables of word n-grams and vice versa. We also present a framework for analysing this problem and develop techniques to solve some of the more common instances of the problem.

Given a table of word n-grams, we describe how to derive the equivalent character n-gram table. For character n-gram table to word n-gram table conversion, we develop a new technique based on Eulerian circuits in directed graphs. We test a proof-of-concept implementation of this technique and evaluate its effectiveness in guessing words from the original text. We also suggest ways in which this technique may be extended and improved for future work with n-grams in NLP and other areas of text processing.

Chapter 1

Introduction

1.1 History

Statistical methods for analysing language are not new. For centuries, code-breakers have used statistical methods for analysing the underlying structure in natural language in order to discern meaning. In particular, sequences of n characters or words known as *n-grams* have been used in frequency analysis and other methods of attacking cryptographic ciphers [6].

While modern cryptography has come to rely more on probability and number theory, n-gram analysis is now used in natural language processing (NLP). Tables of n-gram counts are collected from bodies of text (known as *corpora*) for use in tasks such as speech recognition, error-correction, and formal language theory. The two most common types of n-grams are word n-grams and character n-grams.

1.2 Motivation

In some instances it may be useful to have tables of both character n-grams and word n-grams, but only one table may be on hand. For example, in authorship attribution, distributions of word and character n-grams throughout the text can be used to develop statistical profiles of known authors. For an anonymous sample of text, n-gram distributions are collected and compared with known profiles. If the distributions happen to suit a particular author's profile, that author may be attributed with having written that text sample. However, if word n-gram analysis proves inconclusive, character n-gram analysis may be used instead. If character n-gram profiles are not available for certain authors, it would be useful to be able to generate a character n-gram profile from only the word n-gram profile.

In addition to helping with authorship attribution, n-gram table conversion can be

useful when studying languages without spaces (e.g. Japanese). Like English, words in Japanese may have one, two, three, or more characters. Unlike English however, extracting word n-grams from Japanese is not as straightforward as extracting groups of letters from in-between spaces or some other delimiting character. Humans who can read Japanese distinguish between different words based on their intuition and understanding of the language. Because computers cannot yet understand human language to a great extent, alternative ways of determining words in Japanese must be sought out.

In general, reconstructing long, meaningful strings from shorter ones could be useful not only in NLP, but other areas as well. Data mining and information retrieval, for instance, rely heavily on deriving meaning from large sets of relatively small data. Our DNA is constructed from only four nucleotide bases: adenine, thymine, guanine, and cytosine. These four bases are often represented by the letters A, T, G, and C. Reconstructing large sequences of these letters can help in molecular genetics and other areas of bioinformatics [10].

1.3 General Problem

n-gram tables list the number of occurrences of unique n-grams (for both characters and words). To help solve the aforementioned kinds of problems, we examine the general question of whether or not it is possible to generate the character n-gram table from *only the word n-gram table* for a particular text corpus, and vice versa. We present a framework for analysing the overall problem and develop a new technique using random Eulerian circuits in an attempt to solve some of the sub-problems within that framework.

1.4 Research Objectives

In order to evaluate our solutions, we apply them to the simple task of generating a table of single words from tables of character n-grams. We select our text corpora from famous works of English literature and compare our table of guessed words to tables of known words. Because we do not want to neglect any correct words, we test

our solution for completeness. We also evaluate how many of our guesses are either absent from the original corpus or simply not words at all. The fewer mistakes of either kind, the better. In order to evaluate our solution for speed, we measure the running time in seconds of each trial.

Chapter 2

Background

2.1 Related Work

In recent years, n-grams have gained a lot of popularity in NLP. They have been used for word prediction in both statistical [5] and grammatical [14] speech recognition systems. They have also been studied in conjunction with context-free grammars [12]. More recently, they have been successful in authorship attribution [9] and anti-virus techniques [8].

Similar research has also been done in bioinformatics. In certain molecular structures, nucleotide bases are organised into groups of three. These short sequences are associated with amino acids, which in turn are chained together to form proteins [10]. These structures resemble letters, words, and sentences in natural language. In particular, work done in protein sequence representation [4] and DNA fragment reassembly [11] demonstrate this resemblance.

In both of the aforementioned techniques from bioinformatics, graph theory plays a significant role, Eulerian circuits (i.e. circuits which use each edge exactly once) in particular. Eulerian circuits are understood quite well in graph theory [1]; what undergraduate math student has not encountered Euler's famous *Seven Bridges of Königsberg*? We take advantage of this knowledge and apply it to the general problem of n-gram table conversion. In particular, our research makes use of directed graphs and their Eulerian circuits.

2.2 Problem Framework

We identify various instances in which n-gram table conversion may be necessary:

1. Generating character n-gram tables from word n-grams
   (a) With known delimiters
   (b) With unknown delimiters
   (c) Without delimiters
2. Word n-grams from Character n-grams
   (a) With known delimiters
   (b) With unknown delimiters
   (c) Without delimiters

The length of the source n-grams is important in n-gram table conversion. Given source n-grams longer than the n-grams we wish to generate, generating the other n-gram table is straightforward. Unfortunately, we do not always have these large source n-grams.

Generally speaking, word n-grams will be made up of more characters than character n-grams. Thus, for most practical data sets, character n-gram to word n-gram table conversion will be more difficult than word n-gram to character n-gram table conversion. We present techniques for these conversions with "long" n-grams and "short" character n-grams.

Chapter 3

Formalisms

3.1 Conventions

In the algorithms that follow, we treat n-grams and text corpora as arrays. Array elements are referenced with the traditional square braces [ ] and integer indices beginning with 1 (not 0, as with some programming languages). Individual symbols within an n-gram are indexed with subscripts: ω_j refers to the j^{th} symbol in the n-gram and is equivalent to $\omega[j]$ (but is easier to read). To reference the length of an array, we append ".length" to the array name (e.g. for an array A of 10 elements, $A.length = 10$).

We use hash tables (or simply "hashes") to represent tables of n-grams. Hash elements are referenced with the familiar curly braces { } and keys of arbitrary length. We assume that all of our hashes have worst-cast insertion and search times of $O(n)$ where n is the number of items currently stored in the hash. Additionally, we assume that hash tables of integers are initialised to 0.

We also use a list data structure that behaves as a double-ended stack (LIFO: Last In, First Out). The familiar stack operations *push* and *pop* operate on one end of the list (the "front"), while *add* and *remove* operate on the other end (the "back"). Note that we can emulate a queue (a FIFO) by pushing elements onto the front and removing them from the back, or adding them to the back and popping them from the front.

The *character length* of a word n-gram is its length in characters.

3.2 N-grams

3.2.1 Definitions

n-grams are sequences of words or characters. More formally,

Definition 3.2.1 *Given a set Σ of symbols and a natural number N, an* ***n*-gram** *is a sequence*

$$\boldsymbol{\omega = \omega_1\omega_2 \ldots \omega_{N-1}\omega_N} \tag{3.1}$$

where $\boldsymbol{\omega_j \in \Sigma}$ for $\boldsymbol{1 \leq j \leq N}$, $\boldsymbol{|\omega| = N}$. The words unigrams*,* bigrams*, and* trigrams *are often used to refer to 1-grams, 2-grams, and 3-grams respectively.*

Symbols in Σ are usually *characters* (i.e. single letters) or *words*. *Character* n-grams are sequences of N characters (e.g. $a\text{␣}fa$ is an example of a character 4-gram). When collecting character n-grams, the "space" character ␣ is considered a valid symbol ($\text{␣} \in \Sigma$). Similarly, sequences of N words are known as *word* n-grams (e.g. $a\text{␣}fast\text{␣}German\text{␣}car$).

The *character length* of an n-gram is the number of characters needed to encode the n-gram. For character n-grams of length N, the character length is exactly N. The character length of word n-grams, however, varies with the size of each word. $A\text{␣}fast\text{␣}German\text{␣}car$ has a character length of 17.

3.2.2 N-gram Collection from Text Corpora

Tables of n-gram counts are generated from text corpora using the *sliding window* method. Consider collecting character 4-grams from the phrase *a fast German car*:

```
[a fa]st German car
a[ fas]t German car
a [fast] German car
a f[ast ]German car
a fa[st G]erman car
⋮
a fast German[ car]
```

At each step, the characters inside the box constitute the character 4-gram. Word n-grams are collected in a similar way:

```
[a fast] German car
a [fast German] car
a fast [German car]
```

Our convention will be to treat all letters as uppercase and all punctuation marks as $\sqcup$. We also treat contiguous sections of whitespace as $\sqcup$. We assume that our text corpora have been pre-processed to reflect these conventions.

As we collect n-grams, we count the number of instances of each one in the corpus. We also mark the first and last n-grams as such. Tables of n-grams have entries for all but the last $n-1$ symbols. Given a text corpus L, the following algorithm returns a hash table T of integers having n-grams as keys:

COLLECT N-GRAMS(L)
1 **for** $i \leftarrow 1$ **to** $T.length - N + 1$
2 **do** $current\_ngram \leftarrow L[i..i + N - 1]$
3 $\quad T\{current\_ngram\} \leftarrow T\{current\_ngram\} + 1$
4 **return** T

The character 4-gram table corresponding to the corpus *"a fast German car"* is as follows:

The order of the table entries is not important. Generally, they are sorted according to their counts, but our example is too small to have repeated n-grams.

Note that the length of our corpus L must be greater than the length of our n-grams; we cannot collect an n-gram with length N from fewer than N symbols. Additionally, two different corpora might produce the same n-gram tables, though

4-gram	Count	
␣CAR	1	*last*
␣FAS	1	
␣GER	1	
A␣FA	1	*first*
AN␣C	1	
AST␣	1	
ERMA	1	
FAST	1	
GERM	1	
MAN␣	1	
N␣CA	1	
RMAN	1	
ST␣G	1	
T␣GE	1	

Table 3.1: Character 4-grams for *a fast German car*

in practice this rarely happens. Finally, given a table of n-grams we can generate the equivalent table of $(n-1)$-grams of the same type. Recall that n-grams having length N are collected from all but the last $N-1$ symbols. $(n-1)$-grams, then, are collected from all but the last $(N-1)-1 = N-2$ symbols. To generate a table of $(n-1)$-grams, we can just copy each entry in the table of n-grams, omitting the last symbol in each entry and adding counts for duplicate entries (for example, *fast* and *fash* would both become *fas*). This produces a table of almost all of the $(n-1)$-grams. The only remaining $(n-1)$-gram, the last one, can be collected from the last n-gram in the original table, starting with the second symbol.

3.3 Graph Theory

Let Γ be a directed graph (also known as a *digraph*). Let $V(\Gamma)$ denote the vertex set of Γ and $E(\Gamma)$ denote the edge set. For two vertices $u, v \in V(\Gamma)$, we denote the edge from u to v by $(u, v) \in E(\Gamma)$. We say that Γ is *Eulerian* if and only if for every vertex v of Γ, the number of edges having v as head (the *indegree*) and the number

Trigram	Count	
␣CA	1	
␣FA	1	
␣GE	1	
A␣F	1	*first*
AN␣	1	
AST	1	
CAR	1	*last*
ERM	1	
FAS	1	
GER	1	
MAN	1	
N␣C	1	
RMA	1	
ST␣	1	
T␣G	1	

Table 3.2: Character trigrams for *a fast German car*

of edges having v as tail (the *outdegree*) are the same (indegree(v) = outdegree(v)). Eulerian digraphs contain *Eulerian Circuits* — circuits which use each edge in the graph exactly once. That is, the condition indegree(v) = outdegree(v) for all $v \in V(\Gamma)$ is sufficient to guarantee the existence of an Eulerian circuit [13].

A *spanning arborescence* of Γ is a spanning tree of Γ in which all edges point towards (or away from) a root vertex r. In any Eulerian circuit C of Γ, the subgraph obtained by collecting at each vertex $v \in V(\Gamma), v \neq r$ the first edge in C having v as head forms a spanning arborescence [13]. This spanning arborescence is known as the *residual arborescence* of C [1].

Note: we have only briefly covered the necessary topics from graph theory. For a more formal and complete exposition, see Tutte, [13].

3.4 Precision and Recall

Given a list L of words, we measure how accurate our guesses have been and how much of L we have managed to guess (known as *true positives*). Words that we guess to be true but are not in L are known as *false positives*, while words in L that we fail

to guess are known as *false negatives*. We denote the number of true positives by tp, the number of false positives by fp, and number of false negatives by fn. Precision and recall, then, are defined as follows:

$$\textbf{precision} = \frac{tp}{tp + fp} \tag{3.2}$$

$$\textbf{recall} = \frac{tp}{tp + fn} \tag{3.3}$$

For our purposes, precision is a measure of how many of our guessed words are correct. Recall is a measure of how many correct words out of L we managed to find.

Chapter 4

Methodology

4.1 General Overview

For word n-gram table to character n-gram table conversion, our approach is to collect character n-grams from the first word of each word n-gram except for the last n-gram (in which we collect character n-grams from the entire word n-gram). For character n-gram table to word n-gram table conversion, we build an Eulerian graph from the character n-gram table which associates potential words with cycles in the graph, starting and ending with $\sqcup$. We then follow these cycles and compare the words associated with them to words collected from the original corpus.

4.2 Assumptions

We make some assumptions about our source n-grams before attempting the conversion:

1. **The first n-gram and last n-gram are known.** We need to know which n-grams occur first and last in the original corpus for both types of n-gram conversion. When converting word n-grams to character n-grams, we need to know the last word n-gram so that we do not miss the last few character n-grams contained in that word n-gram. For character n-gram to word n-gram conversion, we build an Eulerian graph. We need the first and last character n-grams in order to ensure that our graph is indeed Eulerian.

2. **The table is complete.** By "complete" we mean that each n-gram in the original corpus is accounted for in the table.

3. **The language has word delimiters.** We use word delimiters in character n-grams to determine where our word guesses begin and end.

4. **We do not have access to the original text.** If we had access to the original text, we would just be able to generate the required n-grams from it instead of from another table of n-grams.

5. **We do not know anything else about the language.** Additional heuristics such as dictionaries and grammars could be used to verify our results. However, we are interested in what is theoretically possible using only the n-grams themselves. In a production environment, other language aids would certainly be used.

4.3 Character n-grams from Word n-grams.

Generating character n-gram tables from word n-gram tables is relatively straightforward. In our word n-gram table where each n-gram has length N, there is an entry which begins with each word in our corpus (except for the last $N-1$). Thus, collecting character n-grams from the first word in each word n-gram is equivalent to collecting them from all but the last $N-1$ words in the corpus. For the last $n-1$ words, we collect character n-grams from all n words until we have collected the last few characters. For example, consider collecting character 4-grams from a table of word bigrams of the phrase *a fast German car*:

Word bigram	Count	
FAST␣GERMAN	1	
A␣FAST	1	*first*
GERMAN␣CAR	1	*last*

Table 4.1: Word bigrams for *a fast German car*

The order of the n-grams is unimportant; we present them here with no particular order. In all bigrams except for the last one (GERMAN␣CAR), we collect character 4-grams up until the first ␣. From the last bigram, we collect character 4-grams from the entire bigram:

Word bigram	Count	Character 4-gram	Count	
FAST␣GERMAN	1	FAST	1	
		AST␣	1	
		ST␣G	1	
		T␣GE	1	
		␣GER	1	
A␣FAST	1	A␣FA	1	*first*
		␣FAS	1	
GERMAN␣CAR	1	GERM	1	
		ERMA	1	
		RMAN	1	
		MAN␣	1	
		AN␣C	1	
		N␣CA	1	
		␣CAR	1	*last*

Table 4.2: Character 4-grams from word bigrams

Aside from the ordering, the resulting character 4-gram table is identical to Table 3.1.

For the following algorithm, we are collecting character n-grams of length N from a table of word n-grams. The input is T, a hash table of integers with word n-grams as keys. The output is c, a hash table of integers with character n-grams as keys. Note that each word N-gram is assumed to have enough characters.

CHARS-FROM-WORDS(T)

1 **for each** $wgram$ **in** T
2 **do if** $wgram$ *is the last one*
3 **then for** $i \leftarrow 1$ **to** $wgram.length - N + 1$
4 **do** $c\{wgram[i..i+N-1]\} \leftarrow c\{wgram[i..i+N-1]\} + T\{wgram\}$
5 **else for** $i \leftarrow 1$ **to** $wgram_1.length + 1$
6 **do** $c\{wgram[i..i+N-1]\} \leftarrow c\{wgram[i..i+N-1]\} + T\{wgram\}$
7 **return** c

4.4 Word n-grams from Character n-grams

When generating word n-grams from character n-grams, the first task is to try to discover possible words (word unigrams). Our technique is summarised in the following steps:

1. Collect and classify n-grams
2. Build an Eulerian graph
3. Generate a spanning arborescence T rooted at the delimiter (␣)
4. Find an Eulerian circuit C having T as its arborescence
5. Traverse C, collecting words as cycles which begin and end with the delimiter

To illustrate the process, we will convert a table of character 4-grams from the phrase *abcd ebcdf* to word unigrams. While *abcd ebcdf* is not a realistic phrase, it a suitable example. Table 4.3 lists the character 4-grams and their counts.

Character 4-gram	Count	
␣EBC	1	
ABCD	1	*first*
BCD␣	1	
BCDF	1	*last*
CD␣E	1	
D␣EB	1	
EBCD	1	

Table 4.3: Character 4-grams from *abcd ebcdf*

The word unigrams we want are ABCD and EBCDF.

4.4.1 N-gram classification

In order to generate a word n-gram table from a character n-gram table, we need to take a closer look at the character n-grams. Some character n-grams will have no delimiters in them at all. We call these *fragments*. Some will only have one delimiter at the beginning or at the end; we call these *prefixes* and *suffixes* respectively. Some will have exactly two delimiters, one at the beginning and one at the end. We call these *explicit* n-grams. Other n-grams will have delimiters in other places, and may have more than one. For the initial word unigram discovery, we only need to consider the fragments, prefixes, suffixes, and explicit n-grams.

Classification	General Form
fragments	$\omega_1\omega_2\ldots\omega_{n-1}\omega_n$
prefixes	$\sqcup\omega_2\ldots\omega_{n-1}\omega_n$
suffixes	$\omega_1\omega_2\ldots\omega_{n-1}\sqcup$
explicit words	$\sqcup\omega_2\ldots\omega_{n-1}\sqcup$

Table 4.4: A character n-gram classification scheme

Given character n-grams of length N, we can read word unigrams directly from the explicit n-grams. Each of these words will be $N-2$ characters long. Because we can also generate a table of n-grams of length $N-1$ from n-grams of length N, we can continue reading word unigrams directly from the explicit n-grams. We can then simply read our word unigrams from the explicit word n-grams just generated.

With this scheme, we classify our example character 4-grams as in Table 4.5.

Because both words are longer than 2 characters, we have no explicit words to read. We also ignore CD$\sqcup$E and D$\sqcup$EB as indicated above.

Character 4-gram	Count	Type	
␣EBC	1	*prefix*	
ABCD	1	*fragment*	*first*
BCD␣	1	*suffix*	
BCDF	1	*fragment*	*last*
CD␣E	1		
D␣EB	1		
EBCD	1	*fragment*	

Table 4.5: Example 4-grams classified

4.4.2 Building directed graphs from n-grams

For words having character length $N-1$ and greater, we build a directed Eulerian graph in such a way that each word in the original text corresponds to one cycle in the graph, beginning and ending with the delimiter.

We construct our graph by using a simple mapping of fragment, prefix, and suffix n-grams to edges in the graph. Conceptually, edges in the graph correspond to pairs of $(n-1)$-grams which overlap. Assume our n-grams have length N. For a given fragment n-gram ω, the corresponding edge is $(\omega_{1..N-1}, \omega_{2..N})$. Prefix n-grams map to $(␣, \omega_{2..N})$ and suffix n-grams map to $(\omega_{1..N-1}, ␣)$. If the *first* and *last* n-grams are fragment n-grams, we add $(␣, \omega_{1..N-1})$ for the *first* n-gram and $(\omega_{2..N}, ␣)$ for the *last* n-gram. This simulates whitespace at the beginning and end of our corpus if it does not already exist. Figure 4.1 shows examples for the character 4-grams ␣FAS, FAST, and AST␣.

$$␣ \longrightarrow FAS \qquad FAS \longrightarrow AST \qquad AST \longrightarrow ␣$$

Figure 4.1: Directed edges made from prefixes, fragments, and suffixes

For each instance of these edges, we implicitly add the vertices $\omega_{1..N-1}$ and $\omega_{2..N}$ if they do not already exist. Note that the graph contains multiple duplicate edges.

Due to its construction, the multiplicity of these edges guarantees that we will always generate the same number of words as in the original text. Specifically, there is one edge coming from and going to the delimiter vertex for each word in our corpus.

Because different corpora can produce the same n-gram tables, they can also produce the same graph structure. This can occur when two or more words in the original corpus share a common substring of length $\geq n - 1$. We refer to this situation as a *word collision*. Figure 4.2 shows an example subgraph which illustrates the word collision caused by the common substring "ldi" shared by "soldier" and "balding":

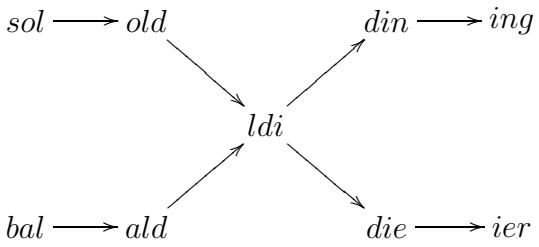


Figure 4.2: Word collision between **soldier** and **balding**

In such situations, there is no way to know which possibility is correct without some type of additional information (such as a dictionary of valid words).

Continuing with our example, we build a graph from Table 4.5:

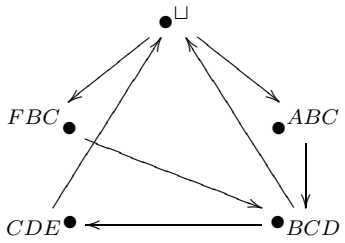


Figure 4.3: An example graph built from character 4-grams

4.4.3 Spanning arborescence using Wilson's Algorithm

While several methods of generating spanning arborescences exist [2], we chose Wilson's Algorithm because it provides a completely random arborescence in the sense that any possible spanning arborescence can be generated. This allows for the widest range of possible words to discover. Although Wilson's Algorithm generates a converging spanning arborescence by default, we can simply reverse each edge to form a diverging arborescence. Alternately, the random walk in Wilson's could be a backwards walk. For our example, shown in Figure 4.4, we use a diverging arborescence so that its relationship to Eulerian circuits is more clear.

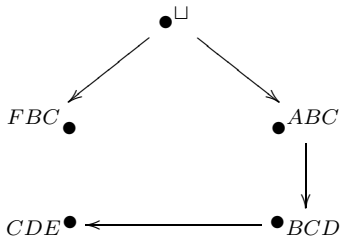


Figure 4.4: An example spanning arborescence

Given a digraph Γ and a vertex r of Γ, Wilson's Algorithm returns a random spanning arborescence T having r as root [2]. Starting with the root vertex, it works by choosing random unvisited vertices and tracing random paths (branches) to the current tree, terminating when all vertices have been visited. It makes use of the *loop erasure* of a path P. Informally, $loop\_erasure(P)$ is the path P with cycles removed (see Gulwani [2]).

```
WILSON-TREE(Γ, r)
1  T ← r
2  unvisited_vertices ← V(Γ) − {r}
3  while unvisited_vertices ≠ φ
4  do v ← a random unvisited vertex
5      P ← random walk from v to current_tree
6      current_tree ← current_tree + loop_erasure(P)
7  return current_tree
```

4.4.4 Eulerian Circuit Construction

Now that we have used Wilson's algorithm to generate our spanning arborescence T of Γ, we now generate an Eulerian circuit C in Γ having T as its residual arborescence. Recall from Section 3.3 that a residual arborescence of an Eulerian circuit is the collection of the *first edges* that visit each vertex in that circuit. Therefore, to find an Eulerian circuit which corresponds to T, we trace edges backwards from r (the root of the tree), ensuring that each branch of the tree is the *last edge* we traverse from each vertex. Because no branch enters the root, the initial edge choice is completely random. When we reverse this random walk, we will have and Eulerian circuit having T as its spanning arborescence [13].

The following algorithm performs this process traversal. Inputs are Γ our graph, T our spanning arborescence, and r the root vertex of T. Recall that for our purposes, r is the delimiter vertex $\sqcup$. I is a hash table of lists of edges $(a, v) \in E(\Gamma), a \in V(\Gamma)$ where vertices $v \in V(\Gamma)$ are keys. The subroutine "shuffle" randomises the order of the elements in its argument (usually a list).

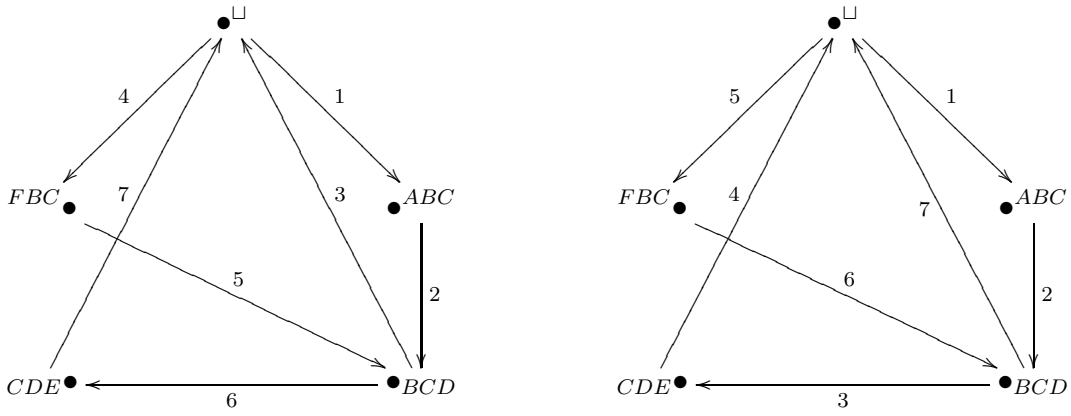
```
BUILD-EULER-CIRCUIT(Γ, T, r)
 1  I{r} ← list of edges (a, r) where a ∈ V − {r}
 2  shuffle(I{r})
 3  for each v ∈ V(Γ) − {r}
 4  do I{v} ← list of edges (a, v) where a ∈ V − {v}
 5       I{v} ← I{v} − {e} where e ∈ E(T)
 6       shuffle(I{v})
 7       I{v}.add(e)
 8
 9  C ← { }
10  current_vertex ← r
11  while I not empty
12  do (a, v) ← I{current_vertex}.pop()
13       current_vertex ← a
14       C ← C.push(current_vertex)
15  return C
```

The return value C is a list of edges which, when reversed (i.e. by pop()-ing them), form an Eulerian Circuit starting and ending with r.

Figure 4.5 shows two example Eulerian circuits: one in which $(CDE, \sqcup)$ is chosen first, the other with $(BCD, \sqcup)$ chosen first. Edges are labeled as they would occur in the final circuit traversal.

4.4.5 Circuit Traversal

Now that we have our Eulerian circuit C, we can traverse the edges and generate word unigrams as we visit r, where $r = \sqcup$. Let W be a hash of integers with word unigrams (strings of various lengths) as keys.



Circuit A *Circuit B*

Figure 4.5: Two example Eulerian circuits

```
GENERATE-WORD-NGRAM-TABLE(C)
 1  current_word ← NIL
 2  while C ≠ φ
 3  do (u, v) ← C.pop()
 4      switch
 5          case u = ⊔ :
 6                  current_word ← v
 7
 8          case v = ⊔ :
 9                  W{current_word} ← W{current_word} + 1
10                  current_word ← NIL
11          case default :
12                  current_word ← current_word + v[v.length]
13  return W
```

Our return value W will be a hash table of possible word n-grams which we can compare to our initial table. Potential *first* and *last* n-grams can be determined from the source character n-gram table.

Our two example circuits A and B give us different unigram guesses. As we traverse circuit A we get the circuit shown in Figure 4.6:

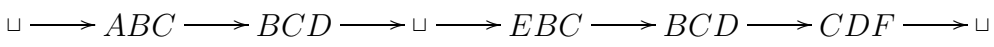


Figure 4.6: Example circuit A

It is easy to see that this circuit generates the word unigrams ABCD and EBCDF, as shown in Table 4.6.

Word unigram	Count
ABCD	1
EBCDF	1

Table 4.6: Word unigrams from circuit A

Circuit B however, shown in Figure 4.7, generates another possibility.

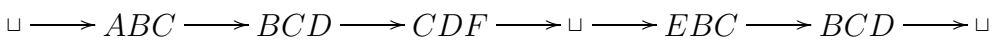


Figure 4.7: Example circuit B

From this circuit, our word unigrams would be ABCDF and EBCD, as shown in Table 4.7.

Word unigram	Count
EBCD	1
ABCDF	1

Table 4.7: Word unigrams from circuit B

Because of the word collision surrounding BCD, the original corpus is ambiguous. If character 5-grams were to be used, the graph simplifies considerably:

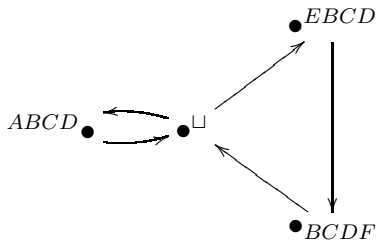


Figure 4.8: Example graph built from character 5-grams

With this graph, there is no ambiguity; the only possible word unigrams are ABCD and EBCDF.

Chapter 5

Experiments

5.1 Experiment Setup

We tested a Perl implementation of our technique with the following corpora from English literature:

- *Wuthering Heights*, by Emily Brontë
- *Tarzan*, by Edgar Rice Burroughs
- *The Warlord of Mars*, by Edgar Rice Burroughs
- *Alice's Adventures in Wonderland*, by Lewis Carroll
- *Through the Looking Glass*, by Lewis Carroll
- *Fanny Hill*, by John Cleland
- *A Tale of Two Cities*, by Charles Dickens
- *A Christmas Carol*, by Charles Dickens
- *King Solomon's Mines*, by H. Rider Haggard
- *The Legend of Sleepy Hollow*, by Washington Irving
- Act III of *Hamlet*, by William Shakespeare
- *Tom Sawyer*, by Mark Twain

From each corpus we generated a table of word unigrams and tables of character n-grams of length $N, 3 \leq N \leq 14$ using the Perl module `Text::NGrams` [7]. We then used our technique with the tables of character n-grams to generate tables of

word unigram "guesses." The root vertex chosen was the delimiter, ␣. Precision and recall measurements were calculated for each trial for both unique words and all words (i.e. with word counts taken into consideration). We also calculated precision and recall for the overall technique and compared it to results generated only by the Eulerian circuit. Finally, we measure the average running time in seconds by using the `time` command found on UNIX systems; we took our measurements from the `user` field.

5.2 Experiment Results

Note: For each experiment, we performed three trials and calculated the standard deviation σ. However, in all experiments, $\sigma \leq 10^{-5}$. While we do not show it in our data plots, it is interesting to note that the technique is so consistent.

5.2.1 Unique Words

The overall technique performs better in precision and recall measurements as the size of the n-grams increases. The precision and recall measurements are shown in Figure 5.1 for the overall technique for unique words.

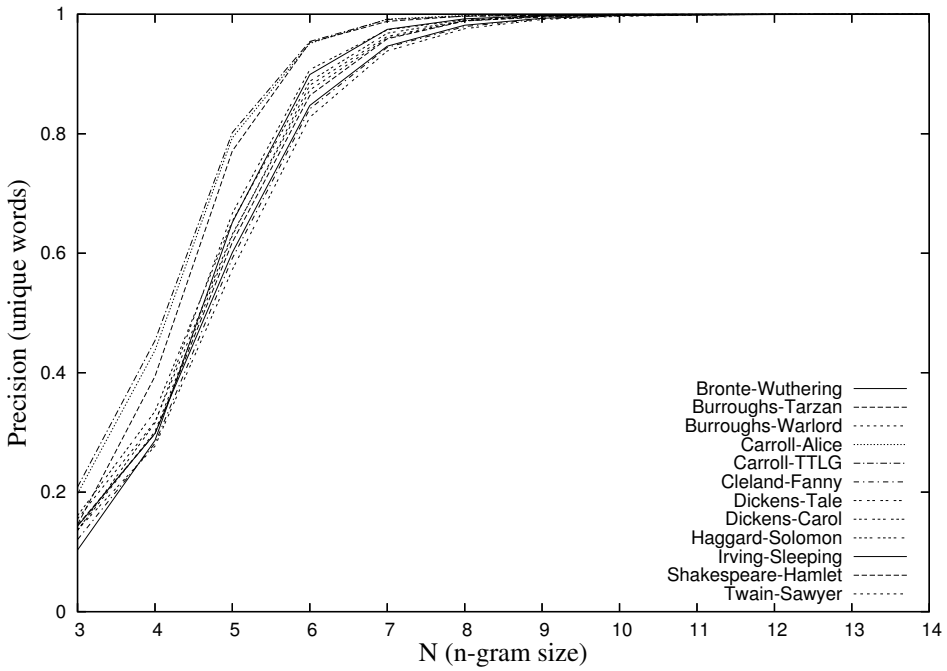


Figure 5.1: n-gram size vs. Precision (unique words)

The most dramatic increases are between $n = 4$ and $n = 6$. For texts with lower average word length (see Appendix A), most notably *Through the Looking Glass* and *Alice's Adventures in Wonderland*, the technique performs substantially better. The recall graph more accurately reflects these differences in word length, as shown in

Figure 5.2:

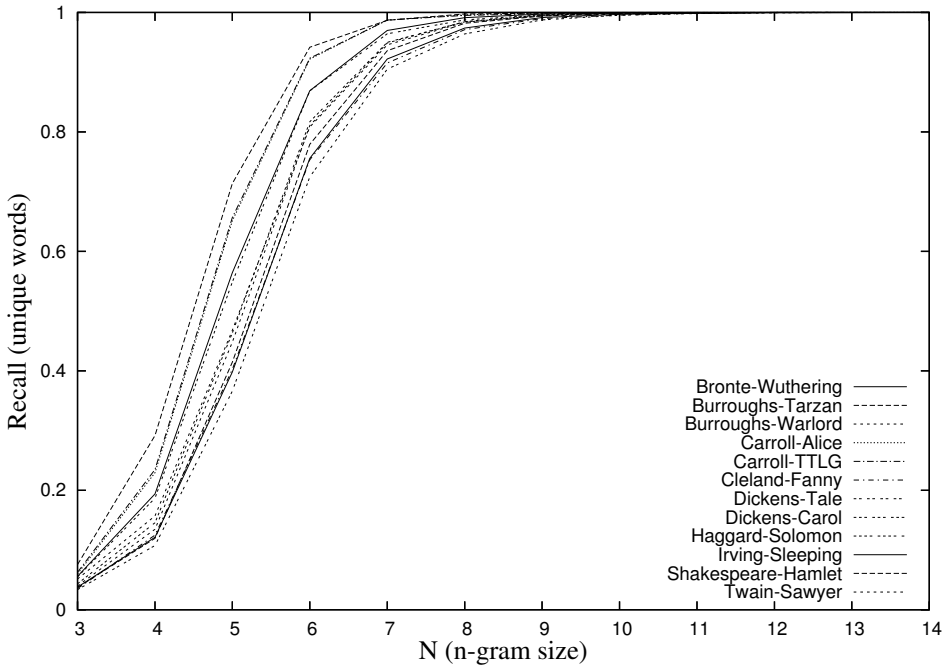


Figure 5.2: n-gram size vs. Recall (unique words)

5.2.2 All Words

Our technique performs better in precision and recall measurements when all words are considered, rather than just unique words. Figure 5.3 shows this increased performance:

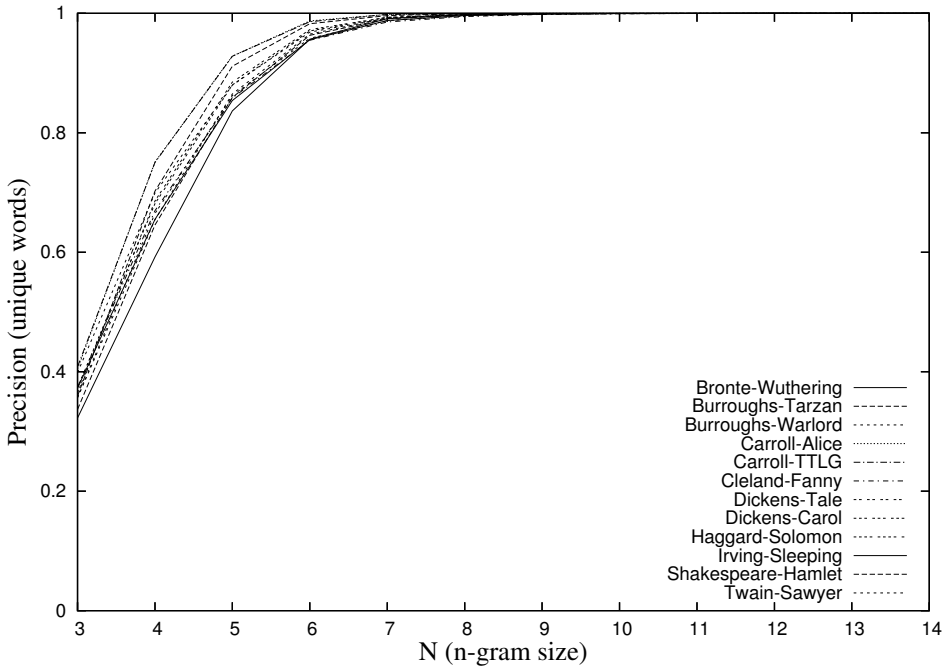


Figure 5.3: n-gram size vs. Precision/Recall (all words)

This change is due to the fact that the average word length is lower when we consider all words in the corpus (see Appendix A). We also note that with word counts taken into consideration, precision and recall are equal. This is due to the fact that our algorithm attempts to reconstruct every word in the original text instead of guessing the distribution. Each failed attempt becomes both a false positive as well as a false negative, thus precision = recall.

5.2.3 Unique Words vs. All Words

For comparison, Figure 5.4 displays the average precision and recall over all corpora for both unique words and all words:

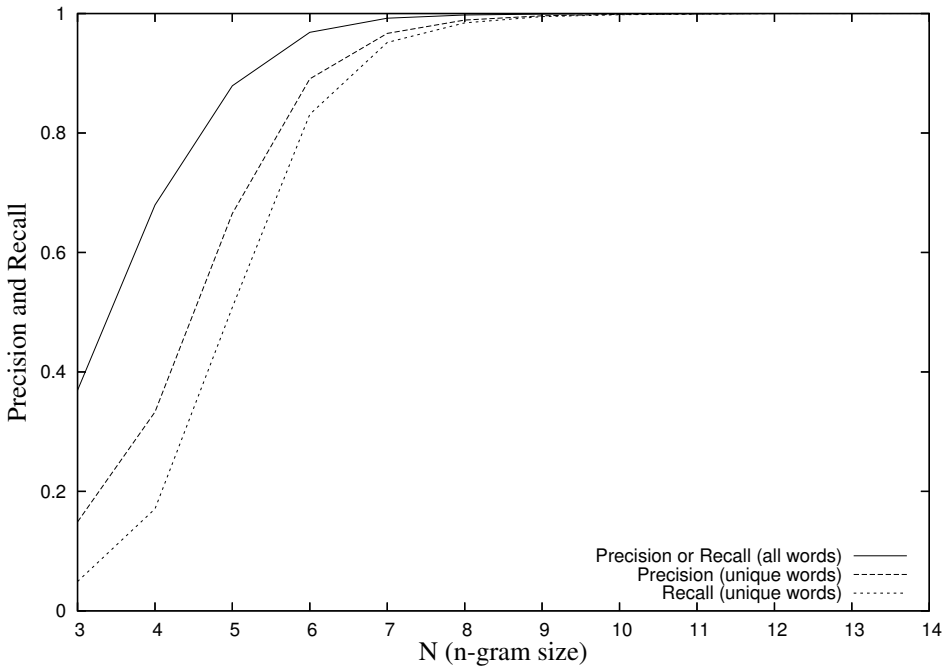


Figure 5.4: Precision and Recall for unique words and all words

While performance for all words is better overall, when the average word length is close to the source character n-gram length, precision and recall are high for unique words. For our corpora, average overall word length for unique words is 6.97 letters per word. In our graph, precision and recall are both 0.95 (95%) when $n = 7$. For all words however, with an average word length of 4.47 letters per word, precision and recall are only 0.8 (80%). This suggests that our technique might be more suitable for raw word discovery than for overall table recovery.

5.2.4 Overall Technique vs. Eulerian Circuit

For n-grams of length N, because the overall technique is guaranteed to be correct for words having character length $\leq N-2$, we compare average precision and recall for only those words generated by the Eulerian circuit (i.e. for words having length $\geq N-1$).

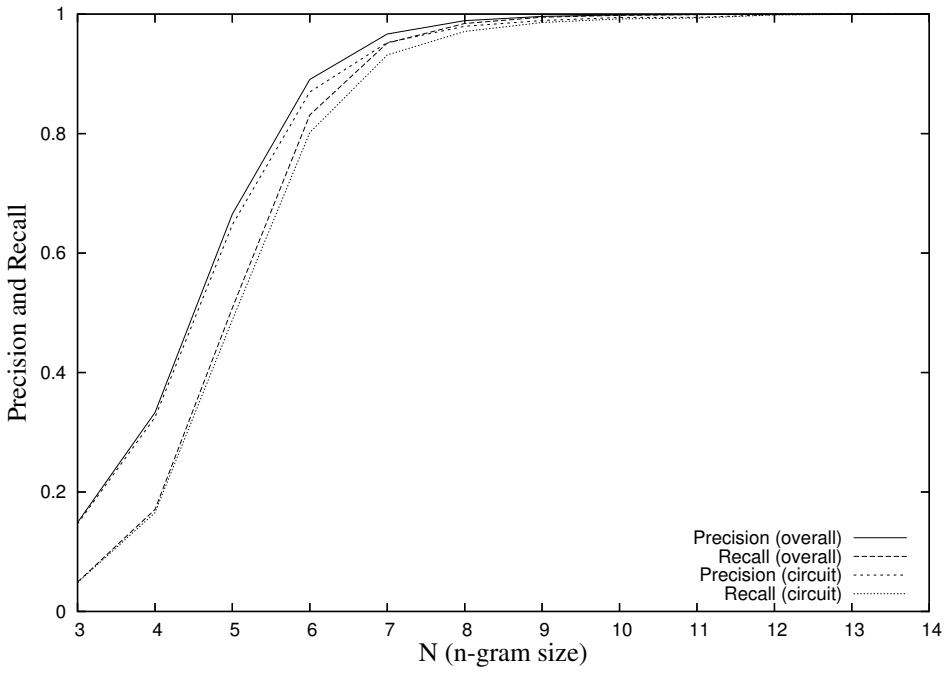


Figure 5.5: Overall vs. Eulerian Circuit

As shown in Figure 5.5, the Eulerian circuit part of our technique performs quite well (the bottom two curves), and traces a curve much like that of the technique overall. This is unsurprising as all words of a certain length ($\leq n-2$) are guaranteed to be found, and thus the shape of the curve overall is determined by the performance of circuit part.

5.2.5 Running Time

The running time for our algorithm increases sharply in roughly the same range in which the precision and recall count increase:

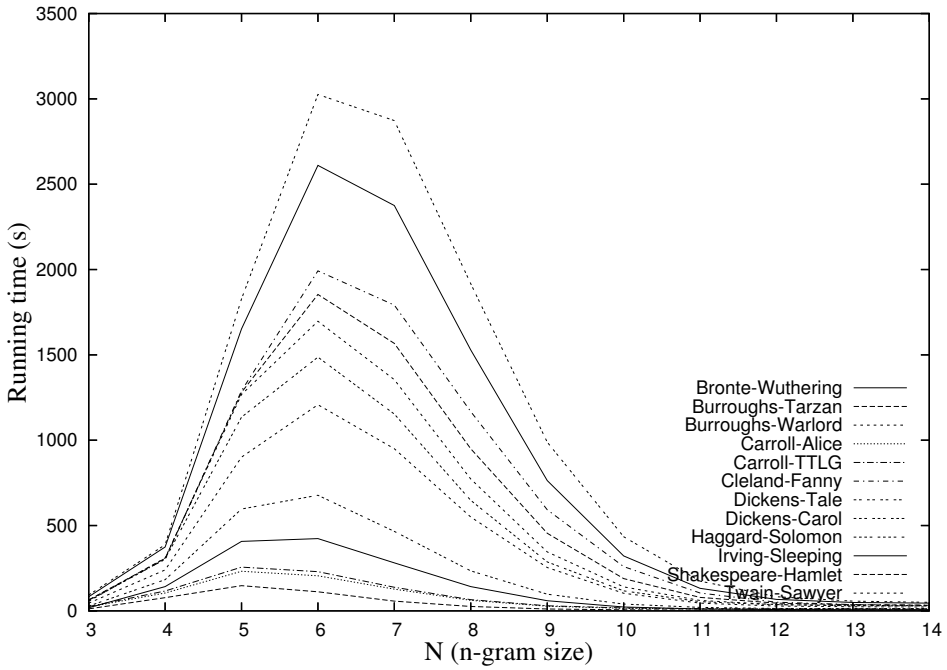


Figure 5.6: n-gram size vs. running time

As n increases, the number of potential n-grams increases exponentially, increasing the size and complexity (i.e. number of potential circuits) of the graph. As a result, the running time increases. However, as the n-gram size increases, the number of actual n-grams collected from the text decreases. Thus the running time goes back down. The running time generally peaks at $n = 6$.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We have presented a framework for the problem of converting tables of one type of n-gram to another. Within that framework, we present techniques for solving the most common real-world scenario of having "long" word n-grams and "short" character n-grams. Not only does our technique recover words from the initial text, but it performs particularly well at retrieving the frequency of those words.

The strengths of our technique lie in its consistency and its completeness. In effect, the Eulerian circuit traversal explores a wide range of possible ways in which the source n-grams overlap.

In general, Eulerian circuits and graph theory seem to be promising areas for future research with statistical and computational linguistics, and with future work could lead to very successful word discovery techniques.

6.2 Future Work

6.2.1 Improving Word Discovery

This technique relies on the fact that the graph is Eulerian, i.e. that the original text is composed of words separated by spaces, which can be translated to cycles in the graph. If applied to other data sets (incomplete tables, for instance), there would have to be an Eulerian circuit in the graph. This can be verified by checking the indegree and outdegree of every vertex, adding edges if necessary to ensure an Eulerian circuit. As well, certain heuristics about the language could be applied when generating the graph. If a cycle is too long to be a potential word, it can be rejected and the edges replaced. A dictionary might also be used, which should guarantee that all words will eventually be found.

Our technique uses both a random spanning arborescence and a random Eulerian circuit in the graph. The spanning arborescence need not be completely random; any spanning arborescence will give rise to many Eulerian circuits in the graph [13]. Generating a spanning arborescence that does not trace any "bad" words will at least slightly improve precision and recall measurements. There is more potential for improvement with the Eulerian circuit and it may be possible to generate the circuit in such a way as to decrease the number of non-word cycles.

The technique can also be applied to tables of n-gram probabilities (rather than discrete counts). In this case, random cycles could be generated until the distribution of words resembles that found in the original text.

6.2.2 Bigrams, Trigrams, and More

Certain types of n-grams are not used in the word discovery algorithm; instead they are used to discover how the words fit together. This is much more difficult, as one n-gram is only likely to contain one delimiter, telling us about a word bigram. n-grams with more delimiters are not nearly as common (though they become more so as n increases).

The Eulerian circuit technique could work quite well with incomplete tables of n-grams. If the graph generated from a given n-gram table is not Eulerian (i.e. for at least one vertex, the indegree and outdegree are different), it suffices to add enough edges from the delimiter vertex to and from these vertices as required. It may also be possible to connect these vertices to each other if they are "close enough," which could be determined with distance metrics.

Our technique grew conceptually from the fact that Eulerian circuits decompose into cycles, where each cycle corresponds to a possible word from the original text. An Eulerian graph could be constructed using the entire character n-gram table. An Eulerian path may then be traversed from the first character n-gram to the last character n-gram, which could possibly generate word bigrams, trigrams, and n-grams for higher values of n.

The Eulerian circuit approach could also be applied to other forms of sequential data, such as phonemes or syllables in NLP. Incomplete copies of ancient texts might

also benefit from this kind of approach. Outside of the domain of natural language, our technique could be applied to music, where notes and phrases correspond to characters and words. Amino acids and protein sequences in bioinformatics also bear a strong resemblance to characters and words. Based on the success of our technique in discovering new potential words from word fragments, Eulerian circuit-based techniques may have a lot to offer.

Bibliography

[1] Graham Brightwell and Peter Winkler. Note on counting Eulerian circuits. Technical report, The London School of Economics and Political Science (LSE-CDAM), 1994. *http://www.cdam.lse.ac.uk/Reports/Files/cdam-2004-12.pdf*.

[2] Sourav Chatterji and Sumit Gulwani. Generating random spanning trees. *http://www.cs.berkeley.edu/~gulwani/random_spanning_trees.ps*.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction fo Algorithms*. The MIT Press, 2nd edition, 2002.

[4] Bailin Hao, Huimin Xie, and Shuyu Zhang. Compositional representation of protein sequences and the number of Eulerian loops. *http://arXiv.org/abs/physics/0103028*.

[5] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2000.

[6] David Kahn. *The Codebreakers*. Scribner, revised edition, 1996.

[7] Vlado Kešelj. Text::Ngrams.pl. *http://vlado.cs.dal.ca/~vlado/srcperl/Ngrams/*.

[8] Vlado Kešelj, Tony Abou-Assaleh, Nick Cercone, and Ray Sweidan. N-gram-based detection of new malicious code. In *COMPSAC 2004 Workshop and Fast Abstract Proceedings*, September 2004.

[9] Vlado Kešelj, Fuchun Peng, Nick Cercone, and Calvin Thomas. N-gram-based author profiles for authorship attribution. In *Proceedings of the Conference Pacific Association for Computational Linguistics, PACLING'03*, pages 255–264. Dalhousie University, Halifax, NS, Canada, August 2003.

[10] David L Nelson and Michael M Cox. *Principles of Biochemistry*. Worth Publishers, third edition, 2000.

[11] Pavel Pevzner, Haixu Tang, and Michael Waterman. An Eulerian path approach to DNA fragment assembly. In *Proc Natl Acad Sci USA*, 2001.

[12] Andreas Stolcke and Jonathan Segal. Precise n-gram probabilities from stochastic context-free grammars. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics (ACL-94)*, 1994.

[13] William Tutte. *Graph Theory*, volume 21 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley Publishing Company, 1984.

[14] Gertjan van Noord, Gosse Bouma, Rob Koeling, and Mark-Jan Nederhof. Robust grammatical analysis for spoken dialogue systems. *Journal of Natural Language Engineering*, 1999.

[15] Eric W. Weisstein. de Bruijn graph. From *MathWorld* – A Wolfram Web Resource. *http://mathworld.wolfram.com/deBruijnGraph.html*.

Appendix A

Average Word Length Tables

A.1 Average Unique Word Length

Corpus	Word count	Average word length (in letters)
Hamlet, Act III	2688	6.00
Through The Looking Glass	5733	6.39
Alice's Adventures in Wonderland	5305	6.51
A Christmas Carol	6748	6.72
The Legend of Sleepy Hollow	3892	6.82
King Solomon's Mines	12196	6.93
The Warlord Of Mars	8859	7.06
Tom Sawyer	12532	7.07
Tarzan	12306	7.19
Wuthering Heights	17715	7.46
Fanny Hill	12574	7.56
A Tale Of Two Cities	19142	7.57

Table A.1: Average unique word length

Average word length overall: 6.97

A.2 Average Overall Word Length

Corpus	Word count	Average word length (in letters)
Through The Looking Glass	30012	4.31
King Solomon's Mines	79675	4.35
Alice's Adventures in Wonderland	26450	4.38
The Warlord Of Mars	57167	4.42
Hamlet, Act III	7643	4.44
Tom Sawyer	71620	4.44
Fanny Hill	84759	4.48
A Christmas Carol	28345	4.49
Wuthering Heights	116051	4.54
A Tale Of Two Cities	135934	4.54
Tarzan	85539	4.57
The Legend of Sleepy Hollow	11811	4.68

Table A.2: Average word length (for all words)

Average word length overall: 4.47

Appendix B

Perl Code

```
#!/usr/bin/perl

#########################################################################
#
#       graphtest.pl
#
#       (c) 2004 Aaron Olson
#
#       This perl script is an ad-hoc implementation of the technique
#       for n-gram table conversion described in the author's honours
#       thesis. It is neither efficient nor bug-free, but is provided
#       for curious readers.
#
#       The script was developed as the author explored the problem
#       and as such is very conceptually laid out.
#
#       Certain empirical measurements were taken (see the chapter
#       entitled "Experiements"). Most of these accurately reflect
#       the effectiveness of the technique, but the increased running
#       time measurements are exaggerated by the inefficient
#       implementation given here.
#
#########################################################################
```

```
use strict;

use warnings;
use diagnostics;

my $delimiter = "_";

my $edge_delim = "-->";

my $ngramfile = shift @ARGV;

if(!defined($ngramfile)) {
        die "Usage: $0 filename\n";
}




#print STDERR "Collecting ngrams...\n";
my %ngrams = &collect_ngrams({ filename => $ngramfile, delimiter => $delimiter });

#print STDERR "Building graph...\n";
my %graph = &build_graph({ ngrams => \%ngrams, delimiter => $delimiter });

#print STDERR "Building tree...\n";
my %tree = &wilson_tree({ graph => \%graph, starting_vertex => $delimiter });
# my %tree = &arborescence({ graph => \%graph, starting_vertex => $delimiter });

#print STDERR "Building path...\n";
my @epath = &euler_path({ graph => \%graph, arborescence => \%tree });
```

```
#print STDERR "Collecting words...\n";
my %results = &collect_words({ path => \@epath, ngrams => \%ngrams });

foreach (keys %results) {
        print "$_\t$results{$_}\n";
}




# from the Perl FAQ
sub fisher_yates_shuffle {
        my $deck = shift;  # $deck is a reference to an array
        my $i = @$deck;
        while ($i--) {
                my $j = int rand ($i+1);
                @$deck[$i,$j] = @$deck[$j,$i];
        }
}


# returns an array of the successors of 'vertex'
sub successors {

        my $args = $_[0];
        my $g = $args->{graph} || die "graph => required";
        my $v = $args->{vertex} || die "vertex => required";
        my @succ;

        if(defined($g->{vertices}{$v})) {
                foreach my $edge (keys %{ $g->{edges} }) {
                        if ($edge =~ /^$v$edge_delim(\w+)$/) {
```

```
                                push @succ, $1;
                        }
                }
        }

        return @succ;
}


# returns an array of edges having 'vertex' as head (including multiple edges)
sub in_edges {

        my $args = $_[0];
        my $g = $args->{graph} || die "graph => required";
        my $v = $args->{vertex} || die "vertex => required";
        my @edges;

        my $i;

        if(defined($g->{vertices}{$v})) {
                foreach my $edge (keys %{ $g->{edges} }) {
                        if ($edge =~ /^(\w+)$edge_delim$v$/) {
                                for($i = 0; $i < $g->{edges}{$edge}; $i++) {
                                        push @edges, $edge;
                                }
                        }
                }
        }

        return @edges;
}
```

```
sub collect_ngrams {

        my $args = $_[0];
        my $filename = $args->{filename} || die "filename => required";
        my $delim = $args->{delimiter} || die "delimiter => required";

        my %ng;
        my $temp = "";

        open (FILE, $filename) or die "Can't open $filename: $!";

                while (<FILE>)
                {
                        my ($ngram, $freq) = split " ";

                        if ($ngram =~ /^[A-Z]+$/) {
                                $ng{fragments}{$ngram} += $freq;
                        } elsif ($ngram =~ /^$delim[A-Z]+$/) {
                                $ng{prefixes}{$ngram} += $freq;
                        } elsif ($ngram =~ /^[A-Z]+$delim$/) {
                                $ng{suffixes}{$ngram} += $freq;
                        } elsif ($ngram =~ /^$delim([A-Z]+)$delim$/) {
                                $ng{words}{$1} += $freq;
                        } else {
#                               print "ERROR: $ngram\n";
                        }
                }

        close(FILE);

        return %ng;
}
```

```
sub build_graph {

        my $args = $_[0];
        my $ng = $args->{ngrams} || die "ngrams => required";
        my $delim = $args->{delimiter} || die "delimiter => required";

        my $head;
        my $tail;

        my %g;

        $g{vertices}{$delim} = 1;       # add the delimiter node

        foreach (keys %{ $ng->{prefixes} })     # add the prefixes
        {
                $head = substr($_, 1, length($_) - 1);

                if (!defined($g{vertices}{$head})) {
                        $g{vertices}{$head} = 1;
                }

                $g{edges}{$delim . $edge_delim . $head} = $ng->{prefixes}{$_};
        }

        foreach (keys %{ $ng->{fragments} })    # add the word fragments
        {
                $head = substr($_, 1, length($_) - 1);
                $tail = substr($_, 0, length($_) - 1);

                if (!defined($g{vertices}{$head})) {
                        $g{vertices}{$head} = 1;
                }
```

```
                if (!defined($g{vertices}{$tail})) {
                        $g{vertices}{$tail} = 1;
                }

                $g{edges}{$tail . $edge_delim . $head} = $ng->{fragments}{$_};
        }

        foreach (keys %{ $ng->{suffixes} })     # add the suffixes
        {
                $tail = substr($_, 0, length($_) - 1);

                if (!defined($g{vertices}{$tail})) {
                        $g{vertices}{$tail} = 1;
                }

                $g{edges}{$tail . $edge_delim . $delim} = $ng->{suffixes}{$_};
        }

        return %g;
}


# returns a hash where each key corresponds to an edge in the arborescence
# a '1' is stored in each entry (as an integer)
sub arborescence {

        my $args = $_[0];
        my $g = $args->{graph} || die "graph => required";
        my $svert = $args->{starting_vertex} || die "starting_vertex => required";

        my %arbor = ( root => $svert );
        my %unvisited;
```

```
        my @visited;

        foreach (keys %{ $g->{vertices} }) {
                $unvisited{$_} = 1;
        }

        push @visited, $svert;

        delete $unvisited{$svert};

        while (%unvisited) {
                my $current = shift @visited || die %unvisited;
                foreach (&successors({ graph => $g, vertex => $current })) {
                        if($unvisited{$_}) {
                                $arbor{$current . $edge_delim . $_} = 1;
                                push @visited, $_;
                                delete $unvisited{$_};
                        }
                }
        }

        # yay! we visited all vertices

        return %arbor;
}


# Wilson's Algorithm for generating a random spanning arborescence
sub wilson_tree {

        my $args = $_[0];
        my $g = $args->{graph} || die "graph => required";
        my $svert = $args->{starting_vertex} || die "starting_vertex => required";
```

```
my %arbor = ( root => $svert );

my %unvisited;
my %visited;

my $dart;
my $dummy;      # placeholder

foreach (keys %{ $g->{vertices} }) {
        $unvisited{$_} = 1;
}

delete $unvisited{$svert};
$visited{$svert} = 1;

while (%unvisited) {
        my @unvisited = keys %unvisited;
        &fisher_yates_shuffle(\@unvisited);

        my $vertex = shift @unvisited;

        # generate random walk
        my @walk = ();
        do {
                my @indarts = &in_edges({ graph => $g, vertex => $vertex });
                &fisher_yates_shuffle(\@indarts);
                $dart = shift @indarts;
                push @walk, $dart;
                ($vertex) = split /$edge_delim/, $dart;
        } while (!defined($visited{$vertex}));

        # store indices of the vertices in @walk
```

```
                my %indices = ();
                my $i = 0;
                foreach (@walk) {
                        ($dummy, $vertex) = split /$edge_delim/, $_;
                        push @{ $indices{$vertex} }, $i++;
                }

                # generate loop-erasure of @walk; store it in @path
                my @path = ();
                $i = 0;
                while($i < @walk) {
                        $dart = $walk[$i];
                        push @path, $dart;
                        ($dummy, $vertex) = split /$edge_delim/, $dart;
                        my $last = pop @{$indices{$vertex}};
                        if($i < $last) {
                                $i = $last;
                        }
                        $i++;
                }

                foreach (@path) {
                        $arbor{$_} = 1;
                        ($dummy, $vertex) = split /$edge_delim/, $_;
                        delete $unvisited{$vertex};
                        $visited{$vertex} = 1;
                } # foreach

        } # while

        return %arbor;
}
```

```
sub euler_path {

        my $args = $_[0];
        my $g = $args->{graph} || die "graph => required";
        my $arbor = $args->{arborescence} || die "arborescence => required";

        my @indarts;
        my $dart;
        my $branch;

        my %queues;
        my $pathcount = 0;

        my @path;

        foreach (keys %{ $g->{vertices} }) {
                @indarts = &in_edges({ graph => $g, vertex => $_ });
                $pathcount += @indarts;

                while(@indarts) {
                        $dart = shift @indarts;
                        if (defined($arbor->{$dart})) {
                                push @{ $queues{$_} }, $dart;           #add to the back
                        } else {
                                unshift @{ $queues{$_} }, $dart;        #add to the fron
                        }
                }

                # randomise order, ensuring the arbor branch comes last
                $dart = pop @{ $queues{$_} };
                my $array_ref = \@{ $queues{$_} };
                &fisher_yates_shuffle( $array_ref );
```

```
                push @{ $queues{$_} }, $dart;
        }

        my $vertex = $arbor->{root};

        while ($pathcount) {
                $pathcount--;
                $dart = shift @{ $queues{$vertex} };
                push @path, $dart;
                ($vertex) = split /$edge_delim/, $dart;
#               $pathcount--;
        }

        return @path;
}


sub collect_words {

        my $args = $_[0];
        my $path = $args->{path} || die "path => required";
        my $ng = $args->{ngrams} || die "ngrams => required";

        my %words;
        my $word = "";

        while (@$path) {
                my $dart = pop @$path;
                my ($tail, $head) = split /$edge_delim/, $dart;
                if($tail eq $delimiter) {
                        $word .= $head;
                } elsif ($head eq $delimiter) {
                        $words{$word}++;
```

```
                        $word = "";
                } else {
                        $word .= substr($head, length($head) - 1, 1);
                }
        }

        foreach(keys %{$ng->{words}}) {
                $words{$_} = $ng->{words}{$_};
        }

        return %words;
}
```