Generating all Possible Palindromes from Ngram Corpora

Alexandre Papadopoulos[1,2] and **Pierre Roy**[1] and **Jean-Charles Régin**[3] and **François Pachet**[1,2]
[1]SONY CSL, 6 rue Amyot, 75005 Paris
[2]Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France
[3]Université Nice-Sophia Antipolis, I3S UMR 6070, CNRS, France
alexandre.papadopoulos@lip6.fr, roy@csl.sony.fr, jcregin@gmail.com, pachetcsl@gmail.com

Abstract

We address the problem of generating all possible palindromes from a corpus of Ngrams. Palindromes are texts that read the same both ways. Short palindromes ("race car") usually carry precise, significant meanings. Long palindromes are often less meaningful, but even harder to generate. The palindrome generation problem has never been addressed, to our knowledge, from a strictly combinatorial point of view. The main difficulty is that generating palindromes require the simultaneous consideration of two inter-related levels in a sequence: the "character" and the "word" levels. Although the problem seems very combinatorial, we propose an elegant yet non-trivial graph structure that can be used to generate all possible palindromes from a given corpus of Ngrams, with a linear complexity. We illustrate our approach with short and long palindromes obtained from the Google Ngram corpus. We show how we can control the semantics, to some extent, by using arbitrary text corpora to bias the probabilities of certain sets of words. More generally this work addresses the issue of modelling human virtuosity from a combinatorial viewpoint, as a means to understand human creativity.

1 Introduction

Virtuosity is a device commonly used by artists to attract the attention of an audience. There is indeed a natural fascination for contemplating objects that are obviously difficult to create or design. It is therefore natural to study human virtuosity as its understanding may produce interesting insights on human creative processes. Virtuosity has so far been addressed from the physiological viewpoint (e.g. by Furuya and Altenmüller [2013]) but has not been addressed in Artificial Intelligence, i.e. from a problem solving viewpoint.

This paper addresses a specific form of virtuosity in text: *palindromes*, i.e. texts that read the same forward and backward, such as "*race car*", or "*A man, a plan, a canal: Panama*". Palindromes have always appealed to humans to the extent of being used in religious contexts. For instance, a famous Byzantine Greek palindrome (Νίψον ἀνομήματα, μὴ μόναν ὄψιν) appears above several religious fountains and churches. Palindromes are also found in many religious texts, from ancient Egyptian papyri to the Quran. For example, an amulet recently discovered in Paphos, Cyprus, bears the inscription of a palindrome written in Egyptian, containing a reference to both the Jewish God and Egyptian deities [Śliwa, 2013]: ΙΑΕW ΒΑΦΡΕΝΕΜ ΟΥΝΟΘΙΛΑΡΙ ΚΝΙΦΙΑΕΥΕ ΑΙΦΙΝΚΙΡΑΛ ΙΘΟΝΥΟΜΕ ΝΕΡΦΑΒW ΕΑΙ.

These ancient palindromes are usually short, highly significant sentences. Another, more recent trend in palindrome invention is to create *very long* palindromes, which increase the combinatorial complexity, usually at the expense of semantic consistency. Competitions are organised to get the longest palindromes that look as natural as possible. Particularly impressive long palindromes in that line were written in French by Georges Perec in 1969 (1,247 words, 5,556 letters) in the form of a short story. Later, in English, a full palindromic novel, *Satire: Veritas*, by David Stephens, was written in 1980 (58,795 letters), and an even longer, *Dr. Awkward & Olson in Oslo*, by Lawrence Levine, in 1986 (31,954 words). Those are, to our knowledge, the longest existing palindromes in English (or, indeed, in any language).

Two kinds of palindromes can be considered: short, highly significant palindromes such as "Madam, in Eden, I'm Adam", and very long palindromes which usually do not carry precise meanings but emphasise the combinatorial dimension of the process. In both cases, virtuosity is at stake.

From the modelling viewpoint, the main issue is that palindromes constrain both the character and word levels of a sequence. In a purely character-based generation, we have to ensure that the sequence can be segmented, using spaces, into valid words, a non-trivial combinatorial problem. With a word-based approach, we face a large combinatorial space: in practice, the Google Books Ngram Corpus [Michel *et al.*, 2011], which contains all Ngrams occurring in the set of English books digitised by Google from 1800 to 2012, contains 4.4M unique words and more than 400M unique 2-grams, i.e. contiguous sequences of two words occurring in a book.

As a consequence of this dual view, it is difficult to apply standard general modelling techniques, such as mixed integer programming, constraint programming or boolean satisfiability solvers. If we fix the number of characters, the number of words remains unknown, and vice-versa. Furthermore, we do not know in advance the position of the centre of the palindrome. Moreover, the language of all palindromic words is not a regular language [Hopcroft *et al.*, 1979], and therefore, automata theory is of no help in finding a greedy algorithm for generating palindromes. Still, we propose a method that generates palindromes of any arbitrary length.

This paper addresses the palindrome generation problem from a combinatorial viewpoint. To our knowledge, this is the first algorithmic solution to a properly formalised combinatorial problem. We propose a method that provides an efficient way of generating all palindromes of arbitrary sizes from a given Ngram corpus. The method is based on a simple yet non-trivial graph structure, which somehow compiles away the inherent difficulty of handling both the character and word levels. As an additional benefit, this method can also be used in a constraint programming setting, enabling the use of extra constraints, such as cardinality constraints to impose semantic clusters.

2 Constrained Text Generation

Markov processes have often be used to automatically generate text. A Markov process is a random process with a limited memory: it produces a state with a probability depending only on the last state, or a fixed number of them. If we apply this to text generation, a state can represent a word, and such a process will generate sequences of words, or phrases. When the Markov model, i.e. the model defining the probability distribution for choosing the next word, is trained on a particular corpus, this process generates sequences imitating the statistical properties of the corpus. As it has been repeatedly observed, those sequences not only contain a semblance of meaning, but also imitate the style of the corpus [Jurafsky and Martin, 2009]. For short, we will say that such a phrase is *based* on the input corpus.

A Markov process can be modelled as a directed graph, encoding the dependency between the previous state and the next state. Then, a *random walk*, i.e. a walk in this graph where the probability for choosing each successor has been given by the Markov model, will correspond to a new phrase. Often, additional control constraints are desired, which cannot be added to the Markov model because they involve long-range dependencies. One approach to deal with control constraints is to generate a vast amount of sequences for little cost, and keep the satisfactory ones. However, this will clearly not be enough for constraints that are hard to satisfy, such as forming a palindrome. Alternatively, we can incorporate the control constraints into the stochastic process, by performing random walk on a more refined, adequately defined, graph. For example, in earlier work [Papadopoulos *et al.*, 2014], we defined a graph in which any walk produces a sequence of words that contains no subsequence belonging to the corpus, longer than a given threshold, in order to limit plagiarism.

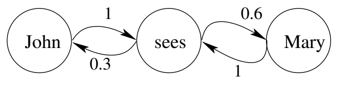
We propose a similar approach, which we first illustrate on the simple problem of generating sequences complying with a syntactic pattern.

2.1 A Motivating Example: Syntactic Patterns

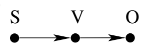
Suppose we want to generate sequences of words based on a given corpus, following a specific syntactic pattern, i.e. a sequence of part-of-speech elements (such as nouns, verbs, etc.). Given such a pattern, we can traverse the corpus graph, i.e. the graph corresponding to a Markov model on this corpus, and check that the pattern of the generated sequence matches the imposed syntactic pattern. Better, we can simultaneously walk the corpus graph and the syntactic pattern, ensuring both walks match.

An alternative method for doing this "simultaneous walk" consists in building a new graph, encoding a type of *conjunction* between the corpus graph and the syntactic pattern graph, in which a walk is a phrase based on the corpus that also satisfies the syntactic pattern.

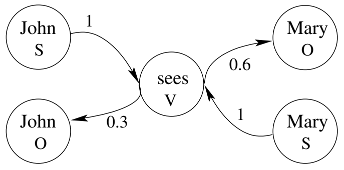
Example 1. Consider a simple corpus, composed of the three phrases "John sees Mary.", "Mary sees Mary.", "Mary sees John.". The graph of a Markov model estimated on this corpus is shown on Figure 1(a), along with the transition probabilities. Suppose we want to impose the pattern Subject Verb Object (SVO, for short). This defines the graph shown on Figure 1(b). We can build a graph in which vertices are tagged by their part of speech, shown on Figure 1(c). A walk in this graph following the SVO pattern will produce a valid sentence from the Markov model, with the right pattern.



(a) Corpus graph



(b) SVO pattern graph



(c) Corpus graph with pattern SVO

Figure 1: The graphs for the example corpus

2.2 Conjunction of Graphs

The tensor product of graphs is an interesting operation for combining graphs and expressing some common properties. It was introduced by Principia Mathematica [Whitehead and Russell, 1912] and is equivalent to the Kronecker product of the adjacency matrices of the graphs [Weichsel, 1962].

Definition 1. The tensor product $G_1 \times G_2$ of graphs G_1 and G_2 is a graph such that the vertex set of $G_1 \times G_2$ is the Cartesian product $V(G_1) \times V(G_2)$; and any two vertices (u_1, u_2) and (v_1, v_2) are adjacent in $G_1 \times G_2$ if and only if u_1 is adjacent with v_1 and u_2 is adjacent with v_2.

We propose a natural extension for labeled graph, which we name *graph conjunction*, where we only associate vertices that have compatible labels, according to a given relation.

Definition 2. The conjunction graph $G_1 \otimes_r G_2$ of two labeled graphs (G_1, l_1) and (G_2, l_2), associated with a Boolean binary relation r on labels, is a labeled graph defined as follows:

1. the vertex set of $G_1 \otimes_r G_2$ is the set of vertices (u_1, u_2) such that $u_1 \in V(G_1)$, $u_2 \in V(G_2)$ *and* $r(l_1(u_1), l_2(u_2))$ is true; vertex (u_1, u_2) is labeled $(l_1(u_1), l_2(u_2))$;
2. there is an edge between (u_1, u_2) and (v_1, v_2) in $G_1 \otimes_r G_2$ if and only if $(u_1, v_1) \in E(G_1)$ and $(u_2, v_2) \in E(G_2)$.

Example 2. The graph of Figure 1(c) is actually the conjunction of the graphs of the two other graphs, where vertices *John* and *Mary* are compatible with either S or O (they are either the subject or the object of a sentence in the given corpus), and vertex *sees* is compatible with vertex V.

3 Application to Palindromes

We now present our approach for generating palindromes. It consists in computing the conjunction of carefully defined graphs representing a corpus.

3.1 Definitions

A palindrome is a phrase such that the character sequence obtained after removing white space, punctuation, ignoring case, is symmetrical. We give an alternative, less intuitive definition, but which is the direct inspiration for the technique we propose. A sequence of characters forms a palindromic phrase if and only if:

1. it can be segmented, by inserting white spaces, into a sequence of valid words, when read from left to right (the forward way);
2. it can also be segmented into a sequence of words, when read from right to left (the backward way);
3. at least one forward segmentation and one backward segmentation produces the same sequence of words.

Note that the third point is not redundant: for example, the sequence of characters `IMADAM` forms a phrase in both ways: we can read "*I'm Adam.*" from left to right, and "*Mad am I.*" from right to left. However, this is not a palindrome because those two phrases are not the same. Additionally, when the third condition holds, the characters are pairwise equal around a central symmetry. For example, the characters `RACECAR` can be read as "*Race car.*" in both directions, and is therefore a palindrome. This means that the word "*Race*" appears at the beginning of the character sequence, and also, in reverse, at the end of the character sequence, which implies the first letter is equal to the last letter, the second to the second last, etc.

Definition 3 (Palindrome Generation Problem)**.** Given a corpus, the palindrome generation problem is the problem of generating the palindrome phrases based on this corpus.

Introducing a corpus has two benefits. The first, as we mentioned earlier, is that it allows us to constraint the generation of palindromes in a way that improves the meaningfulness of the palindromes. This implies a second, combinatorial, benefit. By drastically reducing the number of words that can follow a given word, we greatly reduce the search space at each step of the text generation process. For a dictionary-based approach, where any word can follow any other word, the size of this space is equal to the square of the dictionary size. For a corpus-based approach, this reduces to a size which is linearly bounded by the size of corpus.

3.2 Algorithms

We now present the main algorithms for building palindrome phrases based on a corpus. Our algorithm consists in defining two corpus graphs, one corresponding in a forward traversal of words, and from a word to its possible successors in the corpus, and another for the reverse traversal of words, and from a word to its predecessors in the corpus. The palindrome graph is the conjunction of those two graphs, where vertices are compatible if they agree on their character.

More precisely, from the transition model and from the letters of all the words, we define a graph with a finer granularity. For convenience we denote by $|w|$ the number of letters contained in the word w. We define the so-called forward letter-corpus graph, or simply *forward graph*, as follows:

Definition 4 (Forward Graph)**.** Given a corpus, and transition model M on this corpus, the forward graph $G_f = (V_f, A_f, l_f)$ is a labelled directed graph, such that:

1. The vertex set V_f corresponds all possible positions in each word of the corpus. Formally, for each pair (w, p), where w is a word of the corpus, and p a position in the word (counting from 1 to $|w|$), we define vertex n_{wp}, labelled $l_f(n_{wp}) = (w, p)$.
2. The edge set is defined from the model transitions and from the succession of letters in a word. More precisely,
   - for each word w of the corpus and for each position $p, 1 \leq p < |w|$, there is an arc from n_{wp} to $n_{wp'}$ with $p' = p + 1$;
   - for each non-zero probability transition from w to w', there is an arc from vertex $n_{w|w|}$, corresponding to the last position of the word w, to vertex $n_{w'1}$, corresponding to the first position of the word w'.

Example 3. Suppose we have the corpus with the phrases "Madam Im mad.", "Im Adam.". The full forward graph G_f for this corpus is shown on Figure 2. Note that vertices represent positions in a word, and not characters: character *m* appears on five vertices (since there are five occurrences of *m* in the corpus).

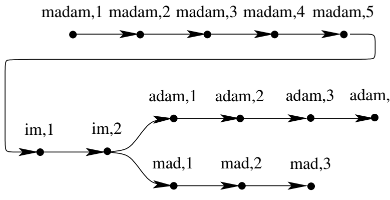


Figure 2: The forward graph for Example 3

Similarly, we define the backward graph, which represents the corpus when read in the reverse direction. It is simply defined by inverting the direction of each arc of the forward graph.

Definition 5 (Backward Graph)**.** Given a corpus, and transition model M on this corpus, the backward graph $G_b =$

(V_b, A_b, l_b) is the graph with the same vertices as the forward graph G_f, with the same labels, and with all arcs inverted.

Algorithm 1 builds the forward graph G_f and the backward graph G_b.

Algorithm 1: Forward and backward graph creation

```
Data: Word-based transition model $M$
Result: $G_f = (V_f, A_f, l_f)$ the *forward graph* based on
        $M$, $G_b = (V_b, A_b, l_b)$ the *backward graph* based
        on $M$
$V_f \leftarrow \emptyset$, $A_f \leftarrow \emptyset$
for $w \in \mathit{words}(M)$ do
    Add vertex $v$ to $V_f$
    $l_f(v) \leftarrow (w, 1)$
    for $i \leftarrow 2, \ldots, |w|$ do
        Add vertex $v'$ to $V_f$
        $l_f(v') \leftarrow (w, i)$
        Add arc to $A_f$ from $v$ to $v'$
        $v \leftarrow v'$
for $w \in \mathit{words}(M)$ do
    for $w' \in \mathit{words}(M) \mid M(w \rightarrow w') > 0$ do
        Add arc to $A_f$ from $v_{w|w|}$ to $v_{w'1}$
$V_b \leftarrow V_f$
$A_b \leftarrow \{(v_2, v_1) \mid (v_1, v_2) \in A_f\}$
$l_b = l_f$
```

Our goal is to produce phrases that are a succession of the same, valid, words in both directions. In order to achieve this, we propose to combine the forward graph G_f and the backward graph G_b. In other words, we define the conjunction graph $G_f \otimes_r G_b$, where the relation r matches vertices that agree on a character. More precisely, two vertices n_{wp} and $n_{w'p'}$ are compatible iff $w[p] = w'[p']$, where $w[p]$ denotes the character of word w at position p. The resulting vertex in the conjunction graph is labelled with both (w, p) and (w', p'). We call *palindrome graph* this conjunction graph.

Intuitively, a vertex of the palindrome graph corresponds to a position in a word of the forward corpus graph *and* a position in another word of the backward corpus graph. A path in the palindrome graph thus corresponds to a simultaneous advancement in the forward graph and in the backward graph, always staying on vertices supporting the same character. Advancing in the forward graph corresponds to building words from left to right, while advancing in the backward graph corresponds to building words in reverse, from right to left. Therefore, advancing in the palindrome graph corresponds to building a palindrome from its extremities towards its centre. In other words, there is a mapping between the palindromes of a given corpus, and paths of the palindrome graph.

Example 4. The palindrome graph of the corpus in Example 3 is shown on Figure 3.

Let us formalise this relationship. Let $G_p = (V_p, A_p) = G_f \otimes_r G_b$ be the palindrome graph. The *initial* vertices of G_p are the vertices that simultaneously start a word in the forward corpus graph and finish a word in the backward corpus graph:

$$V_I = \{v \in V_p \mid l_f(v) = (w, 1) \text{ and } l_b(v) = (w', |w'|)\}$$

For terminal vertices, we need to distinguish between odd-length and even-length palindromes (in terms of number of characters). For odd-length palindromes, the terminal vertices of G_p are the vertices that correspond to the same word at the same position both in the forward and the backward corpus graph. The character at this position is the centre of the palindrome:

$$V_T = \{v \in V_p \mid l_f(v) = l_b(v)\}$$

For even-length palindromes, it is slightly more complicated, since the centre of the palindrome does not fall on a character, but between two characters. As a consequence, the definition of terminal vertices involves a property on edges:

$$V_T = \{v \in V_p \mid (v, v') \in A_p \text{ and } l_f(v) = l_b(v')\}$$

Algorithm 2: Odd Palindrome Graph

```
Data: $G_f$, $G_b$, forward and backward graphs
Result: $G_p$ the palindrome graph of $G_f$ and $G_b$
$V_I = \{(v_f, v_b) \in V_f \times V_b \mid l_f(v_f) = (w, 1)$
                          and $l_b(v_b) = (w', |w'|)\}$
$Q \leftarrow \mathit{EmptyQueue}$
for $(v_f, v_b) \in V_I$ do
    if ConsistentVertex$(v_f, v_b)$ then
        Add vertex $(v_f, v_b)$ to $V_p$
        $Q.\mathit{enqueue}((v_f, v_b))$
        $l_p(v_f, v_b) \leftarrow (l_f(v_f), l_b(v_b))$
// Build graph
while $\neg Q.\mathit{IsEmpty}$ do
    $(v_f, v_b) \leftarrow Q.\mathit{dequeue}$
    for $(v_f, v'_f) \in A_f$ do
        for $(v_b, v'_b) \in A_b$ do
            if ConsistentVertex$(v'_f, v'_b)$ then
                if $(v'_f, v'_b) \notin V_p$ then
                    Add vertex $(v'_f, v'_b)$ to $V_p$
                    $Q.\mathit{enqueue}((v'_f, v'_b))$
                    $l_p(v'_f, v'_b) \leftarrow (l_f(v'_f), l_b(v'_b))$
                Add arc to $A_p$ from $(v_f, v_b)$ to $(v'_f, v'_b)$
// Remove vertices not reaching $V_T$
$\mathit{Trim}(G_p)$
return $G_p = (V_p, A_p, l_p)$

function ConsistentVertex$(v_f, v_b)$
    $(w_f, i_f) \leftarrow l_f(v_f)$
    $(w_b, i_b) \leftarrow l_b(v_b)$
    $\mathit{word}_f \leftarrow w_f[i_f \ldots |w_f|]$
    $\mathit{word}_b \leftarrow -w_b[1 \ldots i_b]$ // Reversed string
    return $\mathit{word}_f \subseteq \mathit{word}_b \vee \mathit{word}_b \subseteq \mathit{word}_f$
```

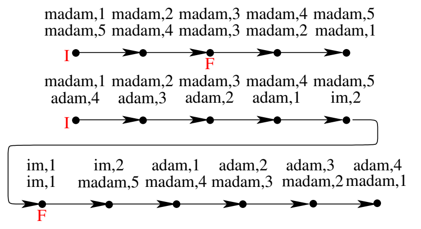

madam,1 madam,2 madam,3 madam,4 madam,5
madam,5 madam,4 madam,3 madam,2 madam,1
I
F
madam,1 madam,2 madam,3 madam,4 madam,5
adam,4 adam,3 adam,2 adam,1 im,2
I
im,1 im,2 adam,1 adam,2 adam,3 adam,4
im,1 madam,5 madam,4 madam,3 madam,2 madam,1
F

Figure 3: The palindrome graph for Example 3

Property 1. *Every path in the palindrome graph from a vertex in V_I to a vertex in V_T corresponds to a palindrome, and all palindromes correspond to such a path.*

Intuitively, when traversing a path from V_I to V_T, we generate the first half of a palindrome by reading the labels coming from the forward graph. Then, by going back to the beginning of the path, we generate the second half of the palindrome by reading the labels coming from the backward graph.

Example 5. In the graph at Figure 3, there are two initial vertices, marked with a red *I*, and two terminal vertices, marked with a red *F*. There are two paths from an *I* vertex to a *T* vertex, corresponding to the palindromes "*Madam.*" and "*Madam Im Adam.*".

Note that a path from a vertex in V_I to a vertex in V_T continues to a path that contains the same palindrome (in reverse). However, we do not need to continue past the vertex in V_T. Not only would this be redundant, but it could actually be wrong: a vertex in V_T could have several successors, only one of which starts a path that generates the same palindrome as the one that reached V_T.

3.3 Building the Palindrome Graph

Building the palindrome graph involves more than a straightforward implementation of the conjunction operator, since we need to limit the explosion in size of the conjunction graph, which could grow as the product of the sizes of the input graphs. We define pruning rules to significantly reduce the size of the conjunction graph and make this approach tractable in practice. From a computational point of view, we avoid considering the whole cartesian product of the set of vertices of the input graphs. Furthermore, we remove all vertices that are part of no valid path, i.e. a path starting in V_I and ending in V_T. Clearly, a vertex that is not in V_T and that has no successor can be removed. Removals are repeated until any vertex is in V_T or has a successor. We call such a graph trimmed. In order to limit memory and running times, we can define pruning rules, to prevent the algorithm from considering vertices that will necessarily lead to parts of the graph that will be eventually trimmed. Algorithm 2 builds the palindrome graph by implementing the conjunction operator following those ideas. It builds the conjunction using a breadth-first search, and then trims unnecessary parts of the graph. The breadth-first search queue is initialised with vertices in V_I that are compatible. Note that compatibility checking is performed using the `ConsistentVertex()` function. This checks that not only do the labels agree on their character, but that the rest of words agree too. For example a vertex with forward label (*adamant*,2) is compatible with vertex with backward label (*mad*,3), since the forward substring starting at index 2 ("damant") is compatible with the reversed backward substring ending at index 3 ("dam"): one is a substring of the other. If this condition does not hold for a vertex, this algorithm does not need to expand on this vertex, since with will necessarily result in a subgraph that will be trimmed. Finally, the algorithm trims the graph by doing a reverse breadth-first search starting from vertices that do not satisfy the accepting condition and have no successor.

3.4 Adding Syntactic Patterns

Thanks to the declarative nature of our approach, we can easily constrain the syntactic patterns of generated palindromes. We need to modify the graph shown on Figure 1(b), by adding a loop from every vertex to itself. By applying the conjunction between this modified pattern graph and the forward graph, we obtain a graph representing all phrases based on the corpus that satisfy the syntactic pattern. Then, we compute the palindrome graph as before, using the unmodified backward graph.

4 Evaluation

We applied our algorithm on several text corpora, and tried to discover interesting palindromes. To improve the entertaining value of palindromes, we tried to generate palindromes alluding to a certain topic. To achieve this, we chose corpora with a very distinctive topic. In order to counterbalance the drastic effect of the palindrome constraint, we enriched the Markov model with 2-grams from the Google Ngram Corpus, with a frequency of occurrence higher than 50,000 (2.4M in total). However, we bias the resulting model to favour corpus 2-grams over Google 2-grams by multiplying corresponding probabilities by a very high factor and renormalise.

In order to randomly generate palindromes, we associate a probability to each starting vertex, and to each edge. The probability of $v \in V_I$ is equal to the prior probability of the forward word w, where $l_f(v) = (w, 0)$. The forward probability of an edge depends on whether this edge falls between a word, or links the end of a word to the start of another one. It is equal, respectively, to 1 or to the transition probability. The backward probability of an arc from u to v is similarly defined, except it takes into account the transition from the word of v to the word of u. The probability for using an edge is defined simply by the product of these two probabilities.

For example, we generated several palindromes using the King James Bible (the English language translation of the Old and New Testament from 1611), which has 868,866 words, 18,295 unique words and 180,651 2-grams. The vast majority of palindromes thus generated had a clear allusion to religion. From a combinatorial point of view, our approach has proved to be very tractable. It takes less than 3 seconds to build the palindrome graph, which has 2,459 vertices, and we can generate 10,000 palindromes in less than a second. It is very easy to generate an arbitrarily long palindrome. For example, we generated a palindrome with 84,541 words, 200,001 characters, longer, to our knowledge, than any palindrome, even a computer generated one. This palindrome starts with “Did I say, lo, he behold I, a sin, a man, are holy” and ends with “I say, lo, he ran a man. I said, lo, he be holy. As I did.”. Although it mimics the corpus well, this excessively long palindrome has very little meaning. A higher level of meaning can be achieved on very short palindromes, such as: “*Evil on an olive.*”, “*To lay a lot.*”, “*Born a man, rob.*”, “*Till i kill it.*”, “*God all I had, I hid: a hill a dog.*”, “*God, a sin: a man is a dog.*”, “*Sworn in us at a sun in rows.*” etc., or slightly longer but less meaningful, such as: “*Evil to no God. I say: do by my body as I do. Go not live.*”.

We also generated palindromes on another corpus, composed of ten books on “military art” downloaded from Project Gutenberg[1]. This corpus has 1.4M words, 50,793 unique words and 443,846 2-grams. The palindrome graph has 294,518 vertices, and is generated in 17s. Here again, the military theme is well evident, in palindromes such as: “*Drawn in war, died. I set a gate side, I, drawn in ward.*”, “*Army by a war, a front, so lost nor far away, by Mr. A.*”, “*Army be I do to go to die by Mr. A.*”,“*Never a way. By a war even.*”. Other interesting palindromes were found: “*Tied a minimum in. I made it.*”, “*Did I?, said I as I did.*”, “*To no demand, a bad name do not.*”, “*Name not one man.*”, etc. Again, longer palindromes tend to be less meaningful: “*Main action was in, or if one, most so late war. On no it is open or prone position, nor a wet, a lost, so men of iron. I saw, no it can, I am.*”.

To illustrate the flexibility of our method, which is not limited to generating palindromes, we generated ambigrams, i.e. sentences that have a different meaning when read backwards. To achieve this, we computed the forward and backward graph on two different corpora, and obtained sentences such as “*Dogs in oil.*” (in reverse “*Lion is god.*”), or “*Sun is alive.*” (in reverse “*Evil as in us.*”), which have a religious meaning when read in reverse. The forward and backward graphs can even be based on corpora in a different language, resulting in sentences such as “*Él a Roma se dedica.*” (in reverse “*Acide de sa morale.*”) or “*Es un otro mal.*” (in reverse “*La mort on use.*”), which can be read in Spanish from left to right, and French from right to left.

5 Discussion

We are aware of very few attempts at automatic palindrome generation. Norvig [2002] builds palindromes on the pattern of “A man, a plan, a canal: Panama”, by growing it using unique words from a dictionary. This is an ad hoc algorithm that uses two complex interleaved searches. It offers no backtracking, or indeed termination, guarantee, and it is hard to add further control. Interestingly, it can be seen as a depth-first search of our palindrome graph.

We have fully solved the problem of generating palindromes as a combinatorial problem. However, the quality of the generated palindromes is on average low as some palindromes barely carry any meaning or lack beauty. This now opens a new problem, the problem of generating *good* palindromes, which should have both meaning and beauty.

Based on our observations, we believe that the beauty of a palindrome may be assessed, at least partially, by objective features. For instance, part of the magic of a palindrome comes from the ingenuity in the way a particular word, especially a long one, appears in reverse order. When the backward copy comprises several shorter words, the palindromic nature is somewhat hidden as in “*Had I a stone, he not said ah!*”, where “*stone*” appears inverted in “*he not said*”. On the contrary, symmetrical palindromes, such as “*refer, poor troop, refer!*”, are obvious, and appear as a mere trick that anyone could have discovered by chance.

We suggest a brief list of features that could capture the quality (correctness and beauty) of a palindrome. Concerning the correctness, the average Markov order of the palindrome, seen as a sequence of words, in the corpus, should be a good indication of the syntactical coherence. One could also check that the part-of-speech (PoS) pattern of the palindrome is acceptable. A sensible measure of the beauty of a palindrome could be given by computing a score with respect to three features: L, the length of the longest word; R, the proportion of repeated words, S, the proportion of words appearing in both directions (such as “*part*”,“*trap*”), where L is to be maximised and R and S minimised. This measure can be refined by taking into account the thematic consistency of the palindrome by, e.g., counting the proportion of words that belong to a specific semantic cluster.

This could be implemented directly within the algorithm, as filters or additional constraints. Alternatively, since our algorithm is very fast, it could be embedded in an evolutionary search procedure, using the above features and evaluation methods, in order to maximise the quality of the output.

6 Conclusion

We have introduced the problem of generating palindromes from a given corpus of Ngrams, and formulate it as a satisfaction problem. We propose a solution based on the introduction of a specific graph structure that combines several crucial properties needed to generate palindromes. This structure yields an efficient algorithm for generating all possible palindromes of any size. This algorithm is notably simple, since we managed to displace the complexity of search procedures to the representational richness of the conjunction graph.

We have applied our solution to actual corpora, and we have exhibited several palindromes pertaining to specific topics. We have also generated palindromes longer than known palindromes produced manually. Our algorithm solves the combinatorial dimension of palindrome generation. By doing so, it also opens a new problem: how to define what is a “nice” or “interesting” palindrome. Thanks to its flexibility and efficiency, we believe our algorithm can be used for testing various aesthetic theories, by incorporating this aspect into the algorithm, or by embedding the algorithm in another optimisation procedure. This fascinating problem, however, is probably not of a combinatorial nature.

Acknowledgements

This research is conducted within the Flow Machines project which received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 291156.

[1]`https://www.gutenberg.org`

References

[Furuya and Altenmüller, 2013] Shinichi Furuya and Eckart Altenmüller. Flexibility of movement organization in piano performance. *Frontiers in human neuroscience*, 7, 2013.

[Hopcroft *et al.*, 1979] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*, volume 3. Addison-wesley Reading, MA, 1979.

[Jurafsky and Martin, 2009] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

[Michel *et al.*, 2011] J. Michel, Y. Shen, A. Aiden, A. Veres, M. Gray, J. Pickett, D. Hoiberg, D. Clancy, P. Norvig, J. Orwant, and et al. Quantitative Analysis of Culture Using Millions of Digitized Books. *Science*, 331(6014):176–182, 2011.

[Norvig, 2002] Peter Norvig. World's longest palindrome sentence. `http://norvig.com/palindrome.html`, 2002. [Online; accessed 9-February-2015].

[Papadopoulos *et al.*, 2014] Alexandre Papadopoulos, Pierre Roy, and François Pachet. Avoiding Plagiarism in Markov Sequence Generation. In Carla E. Brodley and Peter Stone, editors, *AAAI*. AAAI Press, 2014.

[Weichsel, 1962] Weichsel. *Proc. American Mathematical Society*, 13:47–52, 1962.

[Whitehead and Russell, 1912] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. 1912.

[Śliwa, 2013] Joachim Śliwa. Magical amulet from Paphos with the ιαεω- palindrome. *Studies in Ancient Art and Civilization*, 17:293–301, 2013.