Text Generator
Brendan Geisler
Alyssa Krauss
Peter Rowlands
Abstract
This project serves to produce random but coherent text based on a sample input size. Random text
generators can be used in chatbots, poetry, and songwriting. Using a matrix and Markov Chain
algorithms, our text generator can produce random coherent text by accepting an input file that contains
sample text.

Keywords
Markov Chain: an algorithm that produces the next item in a sequence based on the probability of
items that have come next in similar sequences.
state: the words that are input into our Markov Chain algorithm which will determine the next word
order: the number of words in a state
random coherent text: text that makes sense grammatically and is not an exact reproduction of the
sample text.
compressed matrix: a matrix that is not completely populated

Introduction
The use of Markov Chains with a matrix data structure will provide an efficient way to generate random
coherent text.

Data Structure and Building a Markov Chain
The first task of our text generator is to build the Markov Chain. Our Markov Chain consists of a
compressed matrix and two maps. One map contains an index and every unique state found in the
sample text (called the states map.) The other map contains an index and every unique word found in
the sample text (called the words map.) The number of words in each state is defined by the private
data member 'order' which is specified by the user. Typically, the order is around three or four. After
building the states and the words maps from the input file, the matrix must be filled. A cell in the
matrix gets incremented if the word from the word map has occurred directly after the state from the
states map. Consider the following example shown in Figure 1:

Order 2 Markov Chain
Input text: I like food. I like drink but I like food more.
RED: the words map, where the words are keys
YELLOW: the states map, where the states are keys
GREEN: the compressed matrix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
<td>like</td>
<td>food</td>
<td>drink</td>
<td>but</td>
<td>more</td>
</tr>
<tr>
<td>1</td>
<td>like food</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>like drink</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>drink but</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>but i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>food more</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Notice that in the input text, the word “food” comes after the state “i like” twice, and the word “drink” comes after the state “i like” once. Thus, the value of the matrix at (i like, food) is 2 and (i like, drink) is 1. Also notice that most of the entries in the table are NULL. The use of a compressed matrix here eliminates storing zeros and saves on memory costs. Naturally, the input file would consist of many more sentences.

Selecting the Next Word from a Constructed Markov Chain
Choosing the next word when generating text is randomly chosen, but weighted by the number of occurrences of that word following a state. Looking at Figure 1, if the input state is “i like” there is a 2/3 chance that “food” will be the output word, and a 1/3 chance that “drink” will be the output word. The new word would then be pushed to the back of the state, and the first word would be popped. Suppose the first state inputted is “i like.” If the next word that is produced is “food” then the new state becomes “like food.” Run it again on “like food” and the next word produced would be “more.” The algorithm would continue in this fashion until it reaches the end of a sequence. Then, a new first state is randomly chosen and the algorithm repeats. Words are generated until the desired number of words are output. The next word function is shown in appendix A.

Results
Using the text generate to get good results requires some tweaking. First, it is important to choose a “good” sample input. This means the sample should be very large and relatively diverse. Novels, song lyrics, and movie scripts (without the stage direction) all are good options because they tend to have many colloquialisms and will produce coherent random text. Sample inputs that have many outlandish states in it will pull exact sentences from the text because there is only one word to follow a specific state. The only downfall of choosing a large input, is that it may take a while to build the Markov Chain. After choosing a good sample input, selecting the order of the chain is important. Choose too large, and again the generated text will be exact sentences from the sample. Choose too small and the sentences will not make sense. Generally, for large sample inputs (i.e. a novel) an order of three or four works best. For smaller inputs, a small order will be more successful.

Future Work
The next steps in developing our text generator would be to turn it into a full fledged chatbot. The chatbot would make intelligent responses to a query or comment from a user. Accomplishing this would require several steps. First, the first state logic would have to be adapted so it chose a key state from the query. This would involve ignoring common words and focusing on unique words related to the question. Second, we would have to add logic that would recognize questions and return a statement. Finally, we would have to compile a huge sample input file of conversations (most likely from movie scripts) which would create random responses to the same queries.

Conclusions
Overall, the text generator was a success. With a wisely chosen sample text and order, the generated text was random and coherent.
Appendix A.
The nextword function

//Next word takes the input of a state, and outputs a word
string Markov::nextWord(string state) {
    int state_row = 0;
    map<string, int>::iterator it;
    state_row = states[state];
    double total_hits = sum(state_row); //this will be the range of the random number

    // generate random number
    int rand_num = rand() % (int)total_hits;
    int word_col = 0;
    int run_sum = 0;
    Matrix::OneD::iterator rowit = rows(table)[state_row].begin();

    //cout<<total_hits<<endl;
    while (rowit != rows(table)[state_row].end()) {
        run_sum += *rowit;
        if(run_sum > rand_num) {
            word_col = rowit.index();
            break;
        }
        rowit++;
    }
    map<string, int>::iterator it1;
    it1 = words.begin();

    while (it1->second != word_col) {
        it1++;
    }
    return it1->first;
}
Biography

Brendan “Dr. Jones” Geisler
Brendan is a former employee of the United States Postal Service. His favorite meal from the dining hall is beef tips and mushrooms, preferably without mushrooms. He is the most interesting man alive. Brendan doesn't always drink beer, but when he does, he opts for Dos Equis.

Alyssa “Alygator” Krauss
Alyssa is a Virginian Scorpio who aspires to be the richest woman alive. She enjoys reading books on rainy days and going on walks to nowhere, followed by trolley rides back home. Alyssa's favorite courtroom TV show is Judge Judy with Peoples' Court at a close second.

Peter “Short Round” Rowlands
Peter likes to surround himself with all things band, which has led to his media nickname “band-man.” Peter also “no touch nothing.”