Needleman-Wunsch Algorithm on a large phonetic dataset

Dominic Plein

October 19, 2024

1 Project proposal

This project is part of the GP-GPU Computing course at UGA by Christophe Picard. Here, we present a proposal for the final project.

The International Phonetic Alphabet (IPA) uses special symbols[1] to represent the sound of a spoken language. Especially in French, the pronunciation of words can be quite different from the written form, which is why the phonetic alphabet is useful. For new language learners and phonetic enthusiasts alike it could be interesting to find groups of words that sound familiar to a given word. Identifying such groups helps setting a word in a broader context and thus remembering it, and also fosters a joyful play with language.

In order to evaluate the closeness of two words, we need to **define a metric that quantifies the distance of two words**. We only consider their phonetic transcription, i. e. homophones like the French words "vert" (green) and "verre" (glass) have a distance of 0 and are considered the same (in the phonetic sense). For the distance metric, we will employ the **Needleman-Wunsch algorithm**, which calculates the global alignment of two sequences and was originally used in bioinformatics to compare DNA sequences. Here, the alphabet will instead consist of the phonetic symbols. Out of all possible alignments of two words (including gaps), the Needleman-Wunsch algorithm finds the one with the smallest distance, i. e. the alignment with the highest score. A good introduction to the algorithm can be found on the respective Wikipedia page.

To evaluate the feasibility of this project, we combined different datasets to create a new one consisting of **700,000 French words and their phonetic transcription** as well as their frequency in the French language[2]. We implemented a Rust program that calculates the distance of all pairs of words. This corresponds to **computing the weights for all edges of a fully-connected graph**, where nodes are the words (their phonetic transcriptions) and edges have weights corresponding to the distance between the words. For the full dataset, we thus find:

$$\text{Number of nodes} = 700,000 \tag{1}$$

$$\text{Number of edges} = \frac{1}{2} \cdot 700,000 \cdot 699,999 \tag{2}$$

$$= 244,999,650,000 \tag{3}$$

This presents a huge challenge both to computation and memory. Even if we only stored each edge weight with 7 bits, the edges alone would require $\approx 214\,\text{GB}$ of memory. With regards to computation, the algorithms lends itself to parallelization, as the distance calculation of two words is independent of other word pairs.

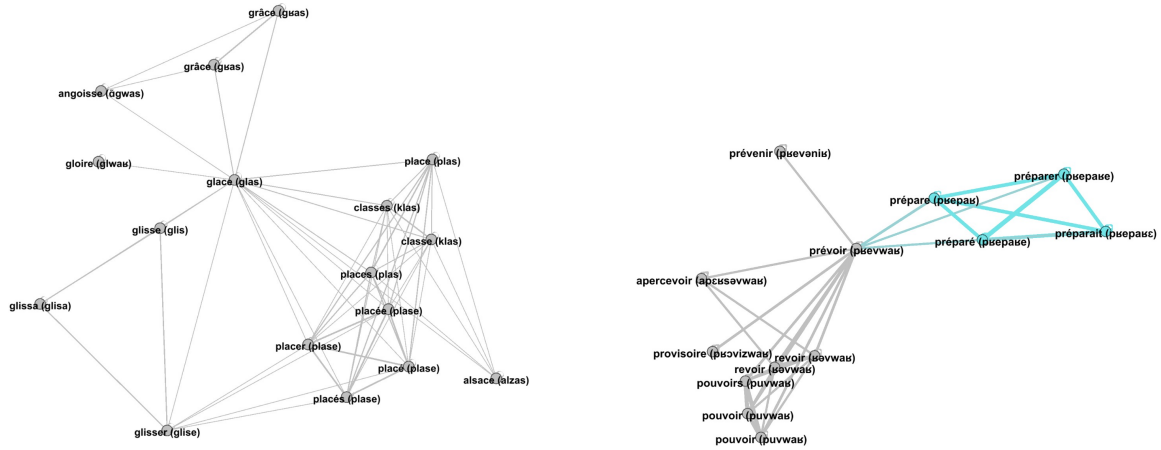
After a first Python implementation, we wrote a Rust program to calculate the distance with Needleman-Wunsch. It was parallelized on a 4-core consumer CPU using the Rayon library. Based on calculations on the first 20,000 words, we estimate that the implementation will take $\approx 3\,\text{h}$ to calculate the distances for all word pairs. This could be acceptable as it only has to be done once. After that, the lookup is in $\mathcal{O}(1)$ time.

However, we also want to adopt a **custom scoring matrix** for the Needleman-Wunsch algorithm that takes into account phonetic symbol similarity itself. For example, replacing a b by a p should have a lower penalty than replacing a b by an m. Defining such a matrix a priori based on random word pairs where only one sound is changed (minimal pairs) has turned out to be difficult. A better approach could be to calculate all distances with a given default scoring (1 on the diagonal, -1 on off-diagonals), then use the results to get the neighbors of a word. With this, we get a better understanding of what words are considered similar by the algorithm and can adjust the scoring matrix accordingly. This iterative process requires a fast implementation of the Needleman-Wunsch algorithm to reduce the turnaround time.

Future works could discuss how clustering algorithms like the Louvain method can be parallelized in a way that also allows only reading parts of the dataset at a time. By doing so, we could cluster words based on their phonetic similarity and thus find *groups of words* that sound similar (compared to only finding the neighbors of a word).

[1] See for example the French list here.

[2] In the final report, we will provide more details on the dataset and their sources.



(a) Neighbors of “glace”

(b) Neighbors of “prévoir”

Figure 1: Visualization of neighbors based on phonetic distance. Edge weights encode the closeness of two words. Colors are artifacts of a previous visualization and can be ignored.

Based on the first 10,000 most frequently used words, we calculated the distance between every pair and visualized the graph using Gephi’s ForceAtlas2 algorithm. The neighbors of the word “glace” and “prévoir” are shown in Figure 1. For bigger graphs than that, Gephi is not able to handle the amount of data anymore.

Having translated the problem into a graph structure also allows us to use graph algorithms to discover interesting properties. As an example, Gephi implements the *shortest path algorithm*: users can click on two words and the shortest path between them is calculated and shown in the graph. Beforehand, we filtered the graph to only include the most strong edges. With this, we can find chains like the following (read them aloud to hear the phonetic similarity):

- trottoir → entrevoir → devoir → voire → voile → val → valait → fallait → falaise
- falaise → fallait → palais → passais → dépassait → dépendait → répondait → répond → raison → maison
- confusion → conclusion → exclusion → explosion → exposition → explications → respiration → précipitation → présentation → présenta → présente → présence → présidence → résidence → résistance → existence

Glossary

IPA International Phonetic Alphabet. 1

Algorithm 1.1: Needleman-Wunsch

Input: $A = \{ A_0, \ldots, A_{\texttt{len}(A)-1} \}$, $B = \{ B_0, \ldots, B_{\texttt{len}(B)-1} \}$,
similarity: similarityScoreFunc, p: GapPenalty
Output: score

```
1  Function calculateScore():
2      Init scoreMatrix with dimensions (len(A) + 1) × (len(B) + 1)
3      for i ∈ {0, ..., len(A)} do
4          scoreMatrix[i][0] ← p · i
5      for j ∈ {0, ..., len(B)} do
6          scoreMatrix[0][j] ← p · j
7      for i ∈ {1, ..., len(A)} do
8          for j ∈ {1, ..., len(B)} do
9              cost ← similarity(A_i, B_j)
10             matchScore ← scoreMatrix[i − 1][j − 1] + cost
11             deleteScore ← scoreMatrix[i − 1][j] + p
12             insertScore ← scoreMatrix[i][j − 1] + p
13             scoreMatrix[i][j] ← max(matchScore, deleteScore, insertScore)
14     return scoreMatrix[len(A)][len(B)]
```