HTWK

Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Automatic dictionary generation for machine stenography

Bachelors thesis

in partial fulfillment of the requirements for the

Bachelor of Science

in the degree of Computer Science
of the faculty Informatik und Medien

Leipzig, 2023-12-12

This is a cleaned up & extended version of the original thesis with all names redacted to serve as documentation for this project!

Abstract

Machine stenography, which allows much faster speeds than achievable with traditional typing, usually utilizes manually written dictionaries translating chord sequences to text. This requires both significant manual labour and is error-prone.

In an attempt to address this issue, this thesis evaluates the potential usage of a configurable optimization based approach for automatically generating a machine stenography dictionary from a pronunciation dictionary and word frequency data.

In doing so, a cost-based model was developed to describe what "good" entries look like, which was then implemented and evaluated in comparison to the Plover theory [1].

The immediate results are mixed, especially due to ambisyllabic consonants, but also because the Plover dictionary includes a significant number of miss-strokes, irregular translations and briefs, all of which this thesis does not attempt to replicate. However, by extending the approach to better handle different possible syllable boundaries, word forms as well as prefixes and suffixes, as well as a better comparison point to better guide the optimization of user configurations, it may result in dictionaries comparable to those that are manually compiled, allowing for easier iteration on a stenography theory.

Contents

1. Introduction

In the realms of traditional typing and writing, extensive practice and dedication can lead to impressive sustained speed and accuracy. For instance, the leaderboards on monkeytype.com [2], a popular online typing test[1], include sustained typing speed[2] records of above 200 words per minute[3] achieved by 103 people. However, most people are well below 100 words per minute, based on the 136M keystrokes dataset [3][4].

In scenarios demanding real-time transcription of speech, such as for instance court proceedings in the United States of America (USA), the typing speeds attained by most people do not match average speaking speeds, considering that the average number of words per minute in conversation exceeds 200 [4].

One possible solution is transcribing using writing systems such as machine stenography or written shorthand. Both utilize abbreviations, as well as optimized symbols in the case of written shorthand or an optimized keyboard layout in the case of machine stenography. In exchange for its steep learning curve, it allows for significantly improved writing[5] speeds [5]. For instance, stenographers are required to maintain at least 180 words per minute to register as professional reporters in the USA [6].

Before modern computerized theories, shorthand had to be manually translated back into traditional writing (in this context referred to as "longhand"). Now, with machine stenography, that translation process can happen automatically in real-time, using a dictionary that maps inputs to actual words.

These dictionaries, while manually assembled for the most part, sometimes take an existing dictionary and modify it to fit an altered theory. This manual approach, however, suffers from avoidable inconsistencies through human error, particularly in the case of modifying existing dictionaries. Additionally, the manual effort involved in changing dictionaries that often contain more than 100,000 entries[6] makes the

[1] As of Dec 11, 379 million tests were completed on the site [2] ("about" section).

[2] in the $60s$ category

[3] here defined as five characters per word

[4] Based on `metadata_participants.txt`, only about 1.37% of participants had an average speed of more than 100wpm, though participants are self-selected and higher typing speeds likely don't correspond to sustained typing speeds as example sentences were at most 70 characters long.

[5] The accepted terminology is "writing" as opposed to "typing" in the case of machine stenography.

[6] The Plover theory has 147424 entries in `main.json` as of Dec 2023 [1].

process of experimenting with different alterations of a theory unrealistic at worst or daunting at best.

This thesis proposes a software system that could be capable of generating a dictionary comparable to existing dictionaries, from a pronunciation dictionary, highly flexible user-defined rules, and configurable weights for both the language and stenography theory. The proposed system allows much quicker iteration on stenography theories, as well as more consistent dictionaries.

Additionally, the user may supply their own used word frequencies to tune the results towards their own language usage. While all popular machine stenography software supports adding custom translations quickly and efficiently, doing so still takes up time and interrupts the process of writing.

While this thesis focuses on a specific American English stenography theory, the Plover theory, it is entirely plausible to use the same approach for other stenography theories, including other languages as all language-specific information is fully configurable.

2. Theoretical background

2.1. Linguistics

It should be noted that this thesis presumes General American (GA) (as opposed to Received Pronunciation (RP)), since the comparison point of the Plover theory [7] is also based on GA.

2.1.1. Phonetic transcription

This thesis uses the International Phonetic Alphabet (IPA) for phonetic transcriptions, which assigns letters (adapted mainly from the Latin and Greek alphabet) to sounds and diacritics as well as suprasegmental letters that modify those sounds, aiming to be able to represent all sounds found in human speech in all spoken languages. The resulting groups of letters with any number of diacritics or suprasegmentals modifying that sound are called "segments". Transcriptions using IPA are notated between slashes in this thesis[7]. For example, the word "linguistics" can be transcribed with IPA as such: /lɪŋˈgwɪ.stɪks/[8][8], in which /ɪ/ describes the specific "i" sound. The two suprasegmentals /ˈ/ and /./ in this transcription represent the syllable boundaries.

It should be noted that a single word can have different pronunciations, even outside of those listed in (pronunciation) dictionaries, so all transcriptions found in this thesis merely correspond to *one* common pronunciation.

Outside of obvious consonants (/pbtdkgmnfvszhlw/), the notation in this thesis[9] is:

- /ŋ/, as in "lu**ng**" /lʌŋ/
- /θ/, as in "pa**th**" /pæθ/
- /ð/, as in "**th**is" /ðɪs/
- /ʃ/, as in "**sh**oe" /ʃuː/
- /ʒ/, as in "fu**si**on" /ˈfjuː.ʒn̩/
- /j/, as in "**y**es" /jɛs/
- /ɹ/, as in "**r**ule" /ɹuːl/
- /t͡ʃ/, as in "**ch**eck" /t͡ʃek/[10]

[7]This thesis only includes phonemic transcriptions, phonetic transcriptions are commonly notated in brackets.

[8]Notated with altered stress markers, since [8] consistently notates ambisyllabic consonants (described in subsection 2.1.2) as belonging to the previous syllable

[9]specifically the sets of IPA symbols and diacritics required to transcribe GA

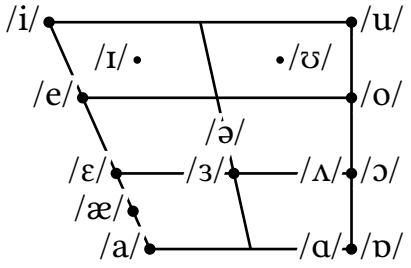

figure 2.1: The IPA vowel chart limited to all vowels used in English

- /d͡ʒ/, as in "**j**am" /d͡ʒæm/[11]
- /ˈ/ marks primary stress, /ˌ/ marks secondary stress, and /./ marks a syllable boundary before an unstressed syllable
- the diacritic in /l̩/ (as opposed to /l/) denotes a syllabic consonant, which is a consonant that takes the place of a vowel or diphthong in a syllable, as in "medd**le**" /ˈmɛd.l̩/[12]
- Vowels as described in the vowel chart in figure 2.1
  - Glide between two vowels such as in the word "eye" /aɪ/, called diphthongs, are notated by notating those two vowels next to each other
  - the horizontal axis encodes the tongue position ("front", "central" & "back")
    - compare /æ/ as in "h**a**t" /hæt/ to /ɑ/ as in "f**a**ther" /ˈfɑːðɚ/
  - the vertical axis encodes mouth closure ("close", "close-mid", "open-mid", "open")
    - compare /i/ as in "fl**ee**ce" /fliːs/ to /e/
  - left to the dot means "unrounded", right means "rounded"
    - compare /ʌ/ as in "c**u**p" /kʌp/ to /ɔ/ as in "h**o**rse" /hɔːɹs/
- /ɚ/ and /ɛˑ/ are "R-colored vowels", as found in words like "word" /wɚːd/
- /ː/ elongates a vowel

(word examples with GA IPA transcriptions from [8])

2.1.2. Syllables

Syllables are made up of the onset, nucleus and coda. The onset and coda are clusters of consonants at the beginning and end of the syllable respectively, separated by the nucleus. The nucleus is a vowel, diphthong or syllabic consonant. As an example, in the word "life" /laɪf/ [8], the onset is /l/, the nucleus is /aɪ/ and the coda is /f/.

A syllabic consonant can also be seen as /ə/ followed by that consonant, for example in the word "meddle" /ˈmɛd.əl/ when pronounced very slowly, instead of /ˈmɛd.l̩/ [8] (in fact, [8] notates syllabic consonants as an optional /ə/ before the consonant instead of the IPA diacritic).

Syllable boundaries in English and many other languages can usually be found using the "maximum onset principle", as described in [9] (principle 2 in theories of the syllable). It describes the principle of splitting syllables from each other in such

[10] with added tie on /tʃ/
[11] with added tie on /dʒ/
[12] with moved syllable markers

a way that the onset of each syllable is as long as possible while remaining a valid possible onset in that language[13]. As an example, for the verb "ab.stract" /æbˈstɹækt/ [8]: /bstɹ/ isn't a valid onset, but /stɹ/, /tɹ/, /ɹ/ as well as the empty onset are. Of those, /stɹ/ is the longest, so the syllable boundary is found between the /b/ and /s/.

It should be noted, however, that syllable boundaries and their patterns are somewhat controversial. For example, the /n/ in the word "dinner" /ˈdɪnɚ/ would be part of the second syllable based on the maximum onset principle, however [8] notates it as /ˈdɪn.ɚ/ instead, and it can in fact be argued that it belongs to either syllable, or possibly both. A consonant that is part of both the coda of the previous and the onset of the next syllable can be called "ambisyllabic", as described in [9] (principle 4 in theories of the syllable).

2.1.3. ARPABET

Another transcription notation specific to the English language is ARPABET, which assigns one or two letters to each phoneme (a set of sounds that are able to distinguish between words, for example /t/ and /d/ are different phonemes as they can distinguish between the words "to" and "do"). This thesis uses the Carnegie Mellon University (CMU)'s CMU Pronouncing Dictionary (CMUdict) [10], which contains the following two letter codes with the corresponding phonemes as translated into IPA in [11]:

AA //ɑ//, //ɒ//	AE //æ//	AH //ʌ//	AO //ɔ//	AW //aʊ//	AY //aɪ//
B //b//	CH //t͡ʃ//	D //d//	DH //ð//	EH //ɛ//	ER //ɜˠ//
EY //eɪ//	F //f//	G //g//	HH //h//	IH //ɪ//	IY //i//
JH //d͡ʒ//	K //k//	L //l//	M //m//	N //n//	NG //ŋ//
OW //oʊ//	OY //ɔɪ//	P //p//	R //ɹ//	S //s//	SH //ʃ//
T //t//	TH //θ//	UH //ʊ//	UW //u//	V //v//	W //w//
Y //j//	Z //z//	ZH //ʒ//			

Two-letter codes are separated by spaces, and vowels may also have a "0" indicating no stress, a "1" indicating primary stress or a "2" indicating secondary stress immediately following them, so the word "father" is transcribed as "F AA1 DH ER0" in [10], corresponding to (with the suprasegmentals moved to the syllable boundaries) /ˈfɑ.ðɜˠ/.

2.2. English machine stenography

English machine stenography usually involves a "stenotype" as a keyboard, or one of the available hobbyist machines, though even a traditional keyboard is possible as

[13]Valid onsets can be observed easily in single syllable words. For example, /stɹ/ is a valid onset as it is found in "**str**ain" /stɹeɪn/ [8].

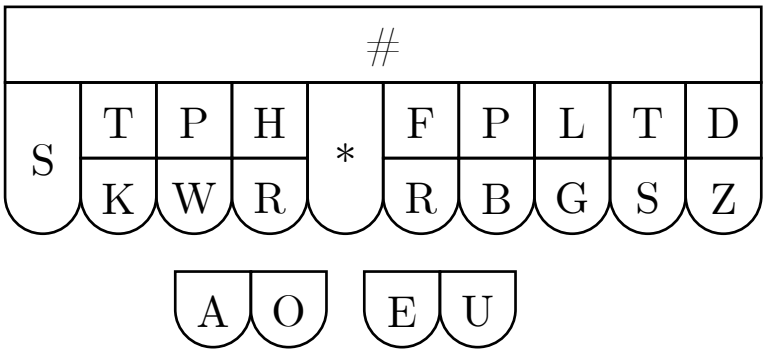

figure 2.2: Layout of a stenotype [12]

well[14]. Professional stenotypes are specialized keyboards or typewriters (and often a combination of the two) with much fewer keys, that don't write character by character, but "chord by chord" [12] ("How steno works"). A chord on a stenotype is the combination of all the keys pressed on a stenotype until all keys are let go. Stenotypes also tend to require significantly less force to actuate to ease writing using combinations of potentially many keys at once[15].

2.2.1. Theories

Most English machine stenography theories use the same basic layout (see figure 2.2).

The main comparison point for this thesis is going to be the Plover theory, as it is freely distributed by the Open Steno Project [7]. As such, all stenography examples will use that theory.

A chord in English machine stenography theories usually stands in for a single syllable or a group of syllables (vowels in unstressed syllables tend to be dropped whenever possible, especially if the word stays fully recognizable) [12] ("How steno works"). Each chord is made up of smaller sets of keys that each encode different sounds or orthographical features of the desired output, in which the physical order on the keyboard[16] and order of those features in the output usually coincide [12] ("Chorded keyboard"). If they do not coincide because a given feature is moved in a chord, that chord is said to have an "inversion" [12] ("Inversion"). Any (non-zero) number of chords can then map to a single translation, which ordinarily is a single word[17].

[14]However, a traditional keyboard with stagger makes pressing combinations of keys such as [TK] harder, and a keyboard without n-key-rollover reduces the potential speed since not all keys will be detected if sufficiently many are pressed at the same time.

[15]In fact, on most lever machines adding an extra key to a chord does not require any significant pressure, as the main required force is used to depress a bar that runs under all keys of the stenotype.

[16]following the order [STKPWHRAOEUFRPBLGTSDZ], the star key and number bar do not constitute an inversion

[17]They can however stand for multiple words as well, as seen in the Magnum theory for instance. In fact, the Guinnes World Record for speed writing held by Mark Kislingbury, the creator of the Magnum theory, averages only 0.8 strokes per word (or 1.25 words per stroke) [13].

As an example, using the Plover theory, the word “steno” can potentially be written with two chords, the left S and T key together with the E key, and then the left T, P and H keys, which in combination can encode the /n/ sound, together with the O and E keys for the /oʊ/ diphthong. This would in short form be written as [STE/TPHOE] (see subsection 2.2.2). A shorter one chord variant can drop the /oʊ/ sound resulting in [STEPB] where [PB] corresponds to /n/. An example involving an inversion is the word “liter”, which can be written as [HREURT], in which [HR] corresponds to /l/ and [EU] to /iː/.

The precise mappings from key combinations to sounds or orthographical features, and based on that, the mapping from chords to actual words is how different “stenography theories” differ, with different choices having different advantages and disadvantages.

For example, the Magnum theory compresses much more heavily at the cost of requiring much more effort in both learning and usage, while the Phoenix theory much more strictly holds to phonetic rules but as such tends to require more chords [12] (“Theories and dictionaries”).

A contributing factor to some theories being harder to learn is the prevalence of possible ambiguities between different desired outputs that would use the same chord or sequence of chords if the other didn’t exist. Examples in the Plover theory include, beyond homophones, word pairs like “life” and the adjective “live” (both matching [HRAOEUF]) in which [F] matches both /f/ and /v/, “debate” and “gate” (both matching [TKPWAEUT]) in which [TK] for /d/ and [PW] for /b/ result in the same output as [TKPW] for /g/.

Plover as a theory that does also significantly depends on spelling for vowels [14] (lesson 3A-4), does however allow a lot of homophones to be easily distinguished. Outside of these, the star key is often added (or, more rarely, removed) to resolve the ambiguity between two words [12] (“*, D, and Z”), and in cases where that is not possible, longer sequences of chords are used.

In general, a dictionary is better:

1. the less chords are required to write a given word, and
2. the easier it is to “guess” which chords correspond to a given word. To that end, a dictionary may also include many different possible sequences of chords that map to the same word.

Additionally, some dictionaries include entries for “misstrokes”, which are chords that are similar to the actual intended ones, to require less manual correction when the stenographer makes a mistake.

2.2.2. Formatting of chords

Chords are formatted by listing each key that is pressed in the order [STKPWHRAO*EUFRPBLGTSDZ]. To distinguish keys between the left and right side that have the same label, if neither a vowel key nor the star key is included, a

dash is inserted instead, so the left and right T keys together would be written as [T-T]. The left key alone can be notated [T], and the right one [-T].

Additionally, if the number bar is pressed down, the keys are named [12K3W4R50*EU6R7B8G9SDZ], and if no keys with a number replacement are pressed a [#] is inserted at the start of the chord instead.

However, the proposed software does not yet handle the number bar by doing that replacement (though it is easily extensible to implement it correctly), and instead just emits the [#] instead, as the number bar is not used significantly in the Plover theory outside of numeric outputs, which this thesis does not attempt to produce.

3. Approach

This thesis attempts to assign costs to different "undesirable" qualities (such as a word requiring more chords, see section 3.6) while attempting to extend its output using user defined partial translations or "rules" (see section 3.5). It then tries to minimize those weights for each word, generating a list of candidates. Weights are all in $\mathbb{R}_0^+$ (the non-negative real numbers, $\mathbb{R}_0^+ = \mathbb{R} \cap [0, \infty)$) or $\overline{\mathbb{R}}_0^+$ (the non-negative real numbers with an infinite element ∞, $\overline{\mathbb{R}}_0^+ = \mathbb{R}_0^+ \cap \{\infty\}$, with $\infty + x = x + \infty = \infty$ as well as $\infty > x$ for all $x \in \mathbb{R}_0^+$)[18].

To be able to apply both orthographic and phonetic rules, it needs to match chunks of spelling with their respective pronunciations. Furthermore, to be able to preferably split words by syllables, it needs to be able to find syllable boundaries (see section 3.2).

The resulting candidates are then used to try and find the best possible dictionary with the least overall weight.

3.1. Patterns

Users can define patterns to match parts of a word for different purposes (see subsection 3.2.1 for mapping the pronunciation to the used spelling, subsection 3.2.2 for splitting syllables and section 3.5 for the user-defined rules that describe how parts of chords are generated). These patterns follow the basic syntax of regular expressions, excluding repetition operators and adding extra syntax for matching on a split alphabet, as described below.

In general, a pattern p, is defined on one or two alphabets A (the primary alphabet, usually IPA segments) and B (the secondary alphabet, used for orthography) with $A \cap B = \emptyset$, defining a language $L(p)$ on $((A \cup B) \times \{m, c\})^*$, where m denotes a matched item and c a consumed item. In case it is defined on one alphabet, B is defined to be the empty set $\emptyset$, and in the case of an overlap between A and B the two can simply be made distinct for the following definitions without loss of generality. The semantic difference between matched (m) and consumed (c) items only matters for user-defined rules and as such is described in the section defining them (section section 3.5) instead. Items may be written with a_k as a short form of (a, k).

[18] An infinite weight can be interpreted as a given circumstance being forbidden from occurring.

The definition of $L(p)$ as well as the syntax for $P_{A,B}$, where $P_{A,B}$ is the set of all patterns of A and B, is, listed in by precedence in the order of decreasing binding strength:

$$\text{Empty word:} \quad L(\varepsilon) = \{\varepsilon\} \tag{3.1}$$

$$\text{Single items:} \quad L(a) = \{a_c\} \qquad \forall a \in A$$

$$L(\#b) = \{b_c\} \qquad \forall b \in B \tag{3.2}$$

$$\text{Item classes:} \quad L([w]) = \{a_c \mid a \in w\} \qquad \forall w \in A^+$$

$$L(\#[w]) = \{b_c \mid b \in w\} \qquad \forall w \in B^+ \tag{3.3}$$

$$\text{Any item:} \quad L(.) = \{a_c \mid a \in A\}$$

$$L(\#.) = \{b_c \mid b \in B\} \tag{3.4}$$

$$\text{Parentheses:} \quad L((p)) = L(p) \qquad \forall p \in P_{A,B} \tag{3.5}$$

$$\text{Secondary pattern:} \quad L(\#(p)) = L(p) \qquad \forall p \in P_{B,\emptyset} \tag{3.6}$$

$$\text{Don't consume}^{19}\text{:} \quad L((?:p)) = L(p) \text{ with } c\text{s switched for } m\text{s} \qquad \forall p \in P_{A,B} \tag{3.7}$$

$$\text{Optionality:} \quad L(p?) = L(p) \cup \{\varepsilon\} \qquad \forall p \in P_{A,B} \tag{3.8}$$

$$\text{Concatenation}^{20}\text{:} \quad L(pq) = L(p) \circ L(q) \qquad \forall p, q \in P_{A,B} \tag{3.9}$$

$$\text{Alternatives:} \quad L(p \mid q) = L(p) \cup L(q) \qquad \forall p, q \in P_{A,B} \tag{3.10}$$

As an example, $a|\#b(?:c)?$ corresponds to the language $\{a_c, b_c, b_c c_m\}$ as an alternative between a and $\#b(?:c)?$, which is a concatenation between $\#b$ and an optional non-consuming c.

For simplicity we define the following sets:

$$I = A \cup B \tag{3.11}$$

$$I' = I \times \{\underbrace{s}_{\text{denotes a skipped item}}, m, c\} \tag{3.12}$$

$$P = (A \cup B) \times \{m, c\} \tag{3.13}$$

It is applied on an input, which contains a matching tag $\{s, m, c\}$ per input item, by returning a weight defined by a cost function $w_p : I'^* \to \overline{\mathbb{R}}_0^+$.

Similarly to the Levenshtein distance (which is defined as the minimum number of insertions, deletions and replacements of characters to turn one string into another), it selects the minimum possible weight based on three cost functions $d : I \times P \to \overline{\mathbb{R}}_0^+$ (the cost for matching single items between a pattern item and input item), $s_i : I \to \overline{\mathbb{R}}_0^+$ (the cost of skipping an input item, or inserting from the perspective of the pattern) and $s_p : P \to \overline{\mathbb{R}}_0^+$ (the cost of skipping a pattern item, or deleting it):

$d'(\varepsilon, \varepsilon) = 0$

[19]The syntax is inspired by non-capturing parentheses in Python's regular expressions (the `re` module in the standard library) [15].

[20]Concatenation here means concatenating over the alphabet $(A \cup B) \times \{m, c\}$. As an example, $a_c b_c \circ d_c = a_c b_c c_m d_c$ with $A = \{a, b, c, d\}$.

$$
\begin{aligned}
d'(a_s, \varepsilon) &= s_i(a) && \forall a \in I \\
d'(\varepsilon, a) &= s_p(a) && \forall a \in P \\
d'(a_s, b) &= \infty && \forall a \in I, b \in P \\
d'(a_m, b) &= d(a_m, b) && \forall a \in I, b \in P \\
d'(a_c, b) &= d(a_c, b) && \forall a \in I, b \in P \\
d'(w, v) &= \min\{d'(w_l, v_l) + d'(w_r, v_r) \,|\, w_l \circ w_r = w, v_l \circ v_r = v\} && \forall w \in I'^*, v \in P^* \qquad (3.14) \\
w_{p(w)} &= \min\{d(w, v) \,|\, v \in L(p)\} && \forall w \in I'^* \qquad (3.15)
\end{aligned}
$$

($d' : I'^* \times I^* \to \overline{\mathbb{R}}_0^+$ in the equations above is a helper function)

In fact, it is precisely the minimum Levenshtein distance ($w_{p(w)} = \min\{\text{lev}(w, v) \,|\, v \in L(p)\}$), $\text{lev}(w, v)$ being the Levenshtein distance between w and v, for the case in which $d(i, p, k) = s_i(i) = s_p(p, k) = 1$.

3.1.1. Kinds of patterns

This thesis uses three kinds of patterns:

- **IPA patterns** (for subsection 3.2.1): $P_{\text{IPA}} = P_{\text{IPA},\emptyset}$.
  These operate on the alphabet of segments transcribed with IPA. As such, for example /aː/ is counted as a single element.
  Diphthongs are counted as multiple segments.
- **Spelling patterns** (for subsection 3.2.1): $P_{\text{Char}} = P_{\text{Char},\emptyset}$.
  These operate on the alphabet of Unicode codepoints.
- **Mixed patterns** (for subsection 3.2.2 and section 3.5): $P_{\text{IPA,Char}}$.
  These operate on both the IPA transcription and spelling simultaneously.

Unless otherwise specified, s_i and s_p are defined as configurable constants for each alphabet. For mixed inputs their returned cost depends on whether the item is a pronunciation or spelling item, with each being configurable separately.

IPA comparisons are more involved, and described in section 3.7.

For spelling comparisons, unless otherwise specified, d returns a configurable constant weight if the items do not match and 0 otherwise.

To apply mixed patterns, the minimum weight of the pattern on all concatenations of all possible interleavings of substring pairs is used. As an example, for $(//\text{f}//, \text{f})$, $(/\text{juː}/, \text{ew})$ (the orthography/spelling mapping for "few" /fjuː/ [8]) the possible orderings are $\{/\text{f}/\#\text{f}, \#\text{f}/\text{f}/\} \circ \{/\text{juː}/\#(\text{ew})\}, /\text{j}/\#\text{e}/\text{uː}/\#\text{w}, /\text{j}/\#(\text{ew})/\text{uː}/, \#\text{e}/\text{juː}/\#\text{w}, ..., \#(\text{ew})/\text{juː}/\}$. Additionally, d returns ∞ if the alphabet of the pattern and input items are not the same, and on spelling items it may be scaled, depending on the user's configuration, by the ratio of the pronunciation length and spelling length.

3.2. Language settings

Language settings are required for orthography matching and syllable splitting, but can also define a set of IPA substitutions in case the IPA input is used. This is useful to ensure more regular inputs and allow potentially easier rules.

The included language definition for English for example includes substitutions to turn /n̩/, /m̩/, /l̩/ and /ɹ̩/ into /ən/, /əm/, /əl/ and /əɹ/ respectively, as this is more conducive to how most stenography theories handle vowels (the Plover theory has no notion of a syllabic consonant for instance). For the same reason, R-colored vowels are replaced by their respective uncolored vowels followed by /ɹ/ (so /ɜˠ/ turns into /ɜɹ/ and /ɚ/ turns into /əɹ/).

3.2.1. Matching spelling and pronunciation

To match spelling and pronunciation, the language settings include a (finite) set of pairs of patterns $P \subset P_{\text{Char}} \times P_{\text{IPA}}$ (see section 3.1 and subsection 3.1.1), with every pair consisting of one pattern for some specific spelling and one for its corresponding pronunciation(s).

To find the most likely associations between spelling and pronunciation substrings, the software generates a pair of equal length partitions of the spelling and pronunciation such that the sum of weights over each pair of substrings in those partitions is minimal. With d, s_i and s_p all being not configurable[21] and returning 1 on mismatches and skipped items, the weight of a pair of substrings (s, p) is defined as:

$$w(s,p) = a + \min\left\{ w_{p_s}(s) + w_{p_p}(p) \,\middle|\, (p_s, p_p) \in P \right\} \tag{3.16}$$

In this definition, a is an additional configurable weight which is added for each pair of substrings to encourage more accurate bigger pairings if they exist.

An example of such a partition is, in the word "tongues" /tʌŋs/ [8][22],

//t//	//ʌ//	//ŋ//	//s//
t	o	ngue	s

, with a weight of $4a$.

3.2.2. Syllable splitting

To find syllable boundaries to be able to split words by their syllables, the language settings include a list of mixed patterns which are unified into a single mixed pattern p using alternatives (see equation equation 3.10) that defines all valid onsets in the given language.

These are applied between nuclei which are identified as any positive number of syllabic IPA segments to find the longest possible onset to satisfy the maximum onset principle (as described in subsection 2.1.2), unless the input already includes a syllable marker between them[23].

[21] except for the IPA power, see section 3.7

[22] with the /s/ added for the plural

[23] Or the ARPABET to IPA adapter generates them, see section 3.8.

To identify the longest possible onset, the onset o with a preceding coda c with the minimum weight $w_p(o) + a \cdot |c|$ is selected, where a is a configurable weight added for each coda element to encourage longer onsets.

However, it has to be noted that this approach cannot detect

- Ambisyllabic consonants,
- Syllable boundaries between nuclei that are not separated by any consonants, or
- Syllable boundaries that do not obey the maximum onset principle for other reasons, such as some compound words in which the syllable boundary is placed between each word instead.

3.3. Stenography layout definition

The stenography layout is defined by the combination of the following:

- The **order of all keys** in formatted chords, so this includes keys that don't contribute to the steno order as well (each key is assumed to have a label that is exactly one Unicode codepoint long)
- The **key before which a hyphen is added** if it is required[24]
- All **unordered keys**, defined as the set of keys that do not contribute to the steno order
- All **"disambiguator keys"**, which are keys that may be used to resolve ambiguities between different desired outputs (for the Plover theory specifically the star key, see "*, D, and Z" in [12])

3.4. Syllables and chords

A given word is split into multiple chords with the following configurable weights in $\overline{\mathbb{R}}_0^+$:

- **Joining:** The cost of joining two syllables including the of nuclei of both involved syllables (this is useful for rules like the one for /ʃn̩/ as seen in words like "fashion" /ˈfæ.ʃn̩/ [8][25]).
- **Joining while dropping a nucleus:** The cost of joining two syllables, dropping the nucleus of one of the two involved syllables (for example turning the transcription for "partial", /ˈpɑːɹ.ʃəl/ [8][26], into /ˈpɑːɹʃl/[27]). It should be noted, however, that the weight adjustments discussed later still refer to the stress of remaining segments as seen originally. These weights can be configured based on the stress of the syllable with the dropped nucleus, and whether it is the left or right syllable.
- **Splitting between syllables:** The cost of starting a new chord at the next syllable (letting a chord and syllable boundary coincide). All unmatched segments in the previous chord contribute to the weight as ignored input elements.

[24] This is necessary to be able to parse and render chords correctly.

[25] with altered stress markers

[26] with altered stress markers and /ɹ/ instead of /r/

[27] Note the absence of a syllable boundary and diacritic which would make the /l/ syllabic, it simply "extends the coda" of the syllable.

- **Splitting inside a syllable:** The cost of starting a new chord without changing the syllables currently in consideration. This can be useful for some consonant clusters for disambiguation purposes or for avoiding overlaps (for example "scalp" /skælp/ [8] could be written [SKAL/-P] to not conflict with "skam" /skæm/ [8] which is written [SKAPL] with [PL] for /m/). The initial rule group is optionally configurable as well, which makes it possible to prefer [SKAL/-P] over [SKAL/P], as the former is the accepted longer chord sequence in the Plover theory since the /p/ is still part of the coda.
- **Syllable lookahead:** An extra weight attached to any consumed item in syllables *after* the ones currently in consideration, this is ordinarily not desired.
- **Fully dropped syllables:** The cost of completely ignoring a syllable. The weights can be configured based on the stress of the dropped syllable.
- **Splitting at a nucleus:** The cost for splitting at the nucleus of a syllable instead of a syllable boundary, useful for words like "debatably" /dɪˈbeɪ.tə.bli/ [8], which could be written [TKPWAEUT/PWHREU] ([TK] for /d/, [PW] for /b/, [AEU] for /eɪ/, [EU] for /i/). The weights can be configured based on the stress of the syllable that is split in half.

Ending a syllable also always adds the weight for any segment that wasn't consumed by any rule.

3.5. Rules

The mapping of phonemic and orthographic features to keys or sets of keys is handled by a set of rules. These are grouped into rule groups, which are ordered and may potentially be configured to allow at most one rule they contain to be applied per chord without a penalty (see rule group violations in section 3.6).

In the case of the Plover theory, these groups are the groups for the onset ([STKPWHR]), the nucleus ([AOEU]) and the coda ([FRPBLGTSDZ]), with the nucleus group only allowing one rule to be applied as all combinations of vowel keys have their own, separate, meaning.

Each rule consists of the following configuration:

- A mixed **pattern** $p \in P_{\text{IPA,Char}}$
- A primary **output** as a set of keys
- Optionally, an **alternative output** or a list of alternative outputs (this is useful for some patterns like the one for /st/ in the coda, which can explicitly map to both [*S] and, alternatively, [FT])
- Optionally, one or more **"unless-rules"** (described in subsection 3.5.1)
- Optionally, a list of **"disallowed outputs"**, for which the rules producing them are not allowed after the application of this rule (not currently used, but can be used to disallow unintuitive combinations of rules)
- Optionally, an **additional weight** in $\mathbb{R}_0^+$ to be added whenever this rule is applied (if the weight is only supposed to be used if there is no other alternative, such as

the one for /zeɪʃən/ (and /tɪʃən/), which maps to [-GS], even though [-GS] ordinarily stands in for the shorter /ʃən/. In this example, this allows shorter chords to be generated that are still intuitive if [-F], which can stand for /z/, is already used or would require an inversion)

- Optionally, whether or not the rule is **"unordered"** (in respect to steno order, by default rules aren't unordered), which disables the addition of the order violation weight described in section 3.6 (this is especially useful for some suffixes such as [-G] /ɪŋg/, which ordinarily do not need to follow the steno order rules due to how common they are)
- Optionally, whether or not the rule is allowed to consume segments in syllables not currently in consideration, which disables the addition of the **syllable lookahead** cost described in section 3.4 (this is used for the [AE] rule for words like "fancy", in which it matches both the current and next vowel)

Applying a rule involves matching the pattern from any position after or at the current position, to any possible end position in the input, while marking consumed items as such. Following that, the output of this rule (or any alternative) is added to the current accumulated chord. The weight of a rule in a given context is the minimum over all possible end positions of the sum of required weight for that match itself as well as any other accumulated weights.

The next rule then begins matching after the first matched item.

3.5.1. Unless-rules

A rule may have any number of "unless-rules" attached. These consist of a mixed pattern, an importance in $\texttt{overline}\{\mathbb{R}\}_0^+$ that defaults to ∞ and a threshold in $\mathbb{R}_0^+$ that defaults to 0.5.

An unless-rule matches the input if and only if its pattern matches with a minimum weight greater than the threshold starting at the first matched item, without marking anything as consumed or adding additional weights outside of the pattern itself. If an unless-rule matches the input, the cost of applying the original rule is increased by the importance.

Unless-rules are important for rules, such as those that shouldn't apply for certain spellings, as other rules are expected to be applied instead. An example is the rule that maps /ə/ to [E] **unless** the spelling contains "a", "i", "o" or "u" at the current position, in which case other vowel key combinations fitting the orthography should be used instead.

3.6. Extra weights and weight adjustments

The theory configuration allows the definition of the following extra weights in $\texttt{overline}\{\mathbb{R}\}_0^+$, each being applied for every occurrence:

- **Output overlap:** The cost of two rules producing overlapping key combinations, based on the keys in the produced output as well as actual shared keys, as this

is generally seen as unintuitive. So [KWR] (/j/) and [P] (/p/) are counted as overlapping, as [K] is before [P] and [R] is after [P] in the steno order.

- **Order violations:** Roughly the cost of inversions, with potentially different weights depending on the number of violations of the steno order. Their number is counted as the number of rules that produce outputs that are ordered before the output that is ordered last in all previously applied rules.
- **Rule group violations:** The cost of applying a rule in a group before the last applied group (preventing rules in the onset group to be used for the coda), or in the same one if it is a group allowing at most one rule (preventing multiple rules in the same group to be used if it shouldn't allow that).
- **Skip group cost:** The cost of not applying any rule in a group before applying a rule after that group. This is useful to prefer earlier groups.
- **Double match:** The cost of matching and consuming the same input item in multiple rules.
- **Rule addition:** The cost of applying any rule in $\mathbb{R}_0^+$. This can be used to encourage "longer" rules.
- **Alternative usage:** The cost of using an alternative output of a rule in $\mathbb{R}_0^+$.
- **Disambiguator cost:** The cost of using a disambiguator key in $\mathbb{R}_0^+$.
- **Anti-disambiguator cost:** The cost of removing a disambiguator key in $\mathbb{R}_0^+$.

The weights for ignored or inaccurate matches of IPA segments can additionally be configured to be multiplied by an adjustment factor based on

- The stress of the syllable, and
- Whether the segment is part of the onset, nucleus or coda of the syllable

This allows stressed syllables to be translated more closely if trade-offs have to be made.

3.7. IPA handling

The cost d (the syllabic aware normalized weighted feature difference to some power p, unless changes aren't permitted) for matching two IPA segments is defined using the following:

- A configurable boolean c, which denotes whether or not **changes are allowed** at all, defaulting to true
- A configurable number $p \in \mathbb{R} \cap (0, 1]$, the **configurable power**, which is used to make differences between phonemes more pronounced (with smaller powers creating bigger increases in weight)
- $f(s)$ returns the feature vector of an IPA segment s using PanPhon [16]
- $d_w(v_1, v_2)$ returns

the normalized (d_w is scaled to return at most 1) weighted distance between two feature vectors as defined in PanPhon [16]

- x_s is true if the segment s is syllabic, and false otherwise

Using these, its definition is:

$$d(a,b,k) = \begin{cases} 0 & \text{, if } a = b \\ \infty & \text{, if } \neg c \wedge a \neq b \\ 1 & \text{, if } c \wedge \neg(a_s \leftrightarrow b_s) \\ d_w(f(a), f(b))^p & \text{, if } c \wedge a \neq b \wedge (a_s \leftrightarrow b_s) \end{cases} \tag{3.17}$$

This allows comparing IPA with some fault tolerance, especially if unexpected diacritics are used. It also allows rules to be used that have a sufficiently similar sound but don't match exactly, at some penalty. However, the actual normalized differences are very small for some pairs of phonemes that are easily differentiable. As such, the configurable power is used to allow small differences to still result in a significant weight (as d_w is in $[0,1]$, $d_w(a,b)^p \geq d_w(a,b)$ if $p \in (0,1]$).

3.8. Pronunciation dictionary

The attempt at recreating the Plover theory, the proposed software solution uses CMUdict [10] as its pronunciation dictionary, since it is very big, freely available for offline use and internally very consistent. CMUdict uses the (two letter) ARPABET transcription codes, which then have to be mapped into IPA to work with the rest of the software.

This is done by replacing all codes with the appropriate IPA transcription, with three exceptions, since CMUdict does not contain the following symbols, as seen in the listed phoneme set [10]:

- **"AH0"** (unstressed /ʌ/) is interpreted as /ə/, as /ə/ is not present in the phoneme list CMUdict. It is consistently transcribed as "AH0" instead, as seen in "above" /əˈbʌv/ [8][28] which is notated "AH0 B AH1 V" in CMUdict version 0.7b.
- **"ER"** (/ɝ/) is interpreted as /ɜɹ/, as IPA substitutions aren't applied for ARPABET inputs[29].
- **"ER0"** (unstressed /ɜextrhoticity/ or /ɜɹ/ with the above substitution) is interpreted as /əɹ/, as /ɚ/ is not present in the phoneme list CMUdict either[30]. It is consistently transcribed as "ER0" instead, for example in "mother" /ˈmʌ.ðɚ/ [8][31] which is notated "M AH1 DH ER0" in CMUdict version 0.7b.

ARPABET transcriptions do not include syllable boundaries, but only use one symbol per valid nucleus (with syllabic consonants being notated as /ʌ/ followed by the consonant instead). As such, there is always a syllable boundary between two ARPABET vowel symbols following one another.

3.9. Generating a full dictionary

To generate a full dictionary, all (or at least as many as possible) words should have at least one translation with the lowest possible weight. To make translations for

[28] both pronunciation and the word itself as an example from /ə/ in the phonetics help page

[29] This is unlikely to pose any significant problems, as ARPABET is specific to English.

[30] It is not interpreted as /ɚ/ for the same reason that "ER1" and "ER2" aren't interpreted as /ɝ/.

[31] with both pronunciation (with altered stress markers) and the word itself as an example from /ɚ/ in the phonetics help page

more common words take precedence, the user may supply their own word frequency data, which multiplies each weight by the word count plus 1, or by 1 if the word count isn't specified.

Additionally, for each unmapped chord sequence for which a word association exists, the lowest weight word is selected. This allows different candidates for chord sequences of a given word to be in the dictionary, reducing the mental effort otherwise required to find exactly the right translation.

To increase efficiency for very long words where a prefix of the chord sequence may suffice, the mapping using that prefix is added as well, as long as it doesn't conflict with any earlier translation or a sequence prefix of a lower weight.

4. Implementation

The implementation is available on GitHub: https://github.com/42triangles/plover-dictgen

4.1. Using A* to find optimal solutions

The specific class of optimization problems which consists of weights for "steps" can be solved by interpreting the problem as a directed weighted graph $G = (V, E)$, where V is the set of all possible states, $c : V \to 2^V$ is the function returning all possible states following a given state and the weights $d : E \to \mathbb{R}_0^+$ correspond to the cost function between states allowed in c:

$$E = \{(v, v') \mid v \in V, v' \in c(v)\} \tag{4.1}$$

To make weights in $\texttt{overline}\{\mathbb{R}\}_0^+$ possible, one can define c' to follow the above definition as follows:

$$c'(s) = \left\{s' \,\middle|\, s' \in c(s), d(s, s') \in \mathbb{R}_0^+\right\} \tag{4.2}$$

To then find the lowest possible cost, Dijkstra's algorithm can be utilized from some initial state i to find any accepting state in some set F.

To increase performance, in some cases it is possible to define some heuristic $h : V \to \mathbb{R}_0^+$ that returns the minimum possible cost required to reach an accepting state in F from any given state. This heuristic can then be used in the A* algorithm, which will still result in optimal results.

If h is a partial function instead, A* can still be utilized by applying the following modifications, where H is the set of all states for which h is defined:

$$V' = \left\{x \,\middle|\, x \in \mathbb{R}_0^+ \times V, x \text{ is reachable from } i' \text{ with } c'\right\} \tag{4.3}$$

$$c'(x, v) = \left\{\begin{cases}(h(v'), v') & \text{, if } v' \in H \\ (\max\{x - d(v, v'), 0\}, v') & \text{, otherwise}\end{cases} \,\middle|\, v' \in c(v)\right\} \tag{4.4}$$

$$d'(x_l, v_l, x_r, v_r) = d(v_l, v_r) \tag{4.5}$$

$$h'(x, v) = x \tag{4.6}$$

$$i' = \left(\begin{cases}h(i) & \text{, if } i \in H \\ 0 & \text{, otherwise}\end{cases}, i\right) \tag{4.7}$$

$$F' = \left\{(w, v) \,\middle|\, w \in \mathbb{R}_0^+, v \in F\right\} \tag{4.8}$$

h' is a valid heuristic (as in, a heuristic that never returns a higher cost than the minimum that is required) if h is:

- Initially, if $i \in H$, the heuristic value is taken from $h'(i)$ which is known to be valid.
- Initially, if $i \notin H$, the heuristic value of zero is always valid as the graph does not include negative edge weights.
- $v' \in H$, in which case the heuristic value is replaced with another valid one.
- $v' \notin H$: Since (x, v) was reachable from i', x is a valid heuristic for v. If $x - d(v, v')$ wasn't a valid heuristic for v' there would be a contradiction (where d can also notate the length of the shortest path between a node and a set of nodes, returning ∞ if they aren't connected):

$$x - d(v, v') > d(v', F) \text{ since } x - d(v, v') \text{ isn't a valid heuristic} \tag{4.9}$$

$$d(v, F) - d(v, v') > d(v', F) \quad \begin{array}{l}\text{since } d(v, F) \geq x \text{ since } x \text{ is a valid} \\ \text{heuristics for } v\end{array} \tag{4.10}$$

$$d(v, v') + d(v', F) < d(v, F) \quad \textit{this violates the triangle inequality} \tag{4.11}$$

4.1.1. Implementation in Rust using coroutines

To make the implementation of the actual application logic as easy as possible, the implementation uses copyable[32] coroutines in Rust. A coroutine with input I (the `trait Coroutine` is generic over that type), yield type Y (`type Coroutine::Yield`) and return type R (`type Coroutine::Return`) is a set of states S equipped with a resuming function $r : S \times I \to (\{\text{yield}\} \times S \times Y) \cup (\{\text{return}\} \times R)$ (`fn Coroutine::resume(self: Pin<&mut Self>, arg: I)`). A coroutine is copyable if any state $s \in S$ can be used multiple times with potentially different inputs.

With $Y = \left\{ y \,\middle|\, y \in 2^{\{I \times \mathbb{R}_0^+ \times (\{\text{noheuristic}\} \cup \mathbb{R}_0^+)\}}, |\{i \,|(i, w, h) \in y\}| = |y| \right\}$ (that is, the coroutine may yield sets of inputs, weights and optional heuristics with unique inputs), and some initial state and input $s_i \in S, i_i \in I$, we can define the requirements for A* as defined above, with a similar heuristic as the one described for partial heuristics, as long as the returned heuristics by the coroutine are correct:

$$V = (\{\text{r}\} \times S \times I \times \mathbb{R}_0^+) \cup (\{\text{d}\} \times R) \tag{4.12}$$

$$h_c(h, w, h') = \begin{cases} h' & \text{, if } h' \in \mathbb{R}_0^+ \\ \max\{h - w, 0\} & \text{, if } h' = \text{noheuristic} \end{cases} \tag{4.13}$$

$$c'(s, b, h) = \begin{cases} \{(\text{r}, s', b', h_c(h, w, h')) \,|(b', w, h') \in y\} & \text{, if } r(s, b) = (\text{yield}, s', y) \\ \{(\text{d}, r)\} & \text{, if } r(s, b) = (\text{return}, r) \end{cases} \tag{4.14}$$

$$c(x) = \begin{cases} c'(s, b, h) & \text{, if } x = (\text{r}, s, b, h) \\ \emptyset & \text{, if } x \in F \end{cases} \tag{4.15}$$

[32] meaning they implement `Clone` (`Copy` also exists and has stricter requirements, but "cloneable" is Rust specific terminology and the distinction is not useful for this thesis)

$$d'(s,b,s',b') = \begin{cases} w \text{ , if } r(s,b) = (\text{yield}, s', y), & \underbrace{(b', w, h') \in y}_{\substack{b' \text{ is unique in } y \\ \rightarrow \text{ there is exactly one such element}}} \\ 0 \text{ , if } r(s,b) = (\text{return}, r) \end{cases} \tag{4.16}$$

$$d(x,y) = \begin{cases} d'(s,b,s',b') \text{ , if } x = (\text{r}, s, b, h), y = (\text{r}, s', b', h') \\ 0 \text{ , if } y \in F \end{cases} \tag{4.17}$$

$$h(x) = \begin{cases} h \text{ , if } x = (\text{r}, s, b, h) \\ 0 \text{ , if } x \in F \end{cases} \tag{4.18}$$

$$i = (\text{r}, s_i, i_i, 0) \tag{4.19}$$

$$F = \{\text{d}\} \times R \tag{4.20}$$

The actual types in the implementation are defined as follows (for some weight type `W` $\subseteq \mathbb{R}_0^+$ and deduplication token `D: Hash`:

Rust
```rust
// The yield type
#[derive(Clone, Default)]
pub enum Instruction<W, D> {
    #[default]
    Fail,
    Push(W),
    Branch,
    Heuristic(W),
    Dedup(D),
}

#[derive(Clone, Copy, PartialEq, Eq, Default)]
pub enum BranchTaken {
    #[default]
    Left,
    Right,
}

// The input type
#[derive(Clone, Copy, Default)]
pub struct State<W> {
    // the minimum weight to reach the current state as identified
    // through deduplication, this is necessary for the chord
    // generation implementation
    pub weight: W,
    pub branch: BranchTaken,
}
```

The input to the coroutine is a `State<W>`, and the output is a `Instruction<W, D>`, where

- `Instruction::Fail` corresponds to an empty continuation (which may also be used instead of `Instruction::Push` to add a weight of ∞),
- `Instruction::Push(w)` corresponds to adding a cost of w without branching,
- `Instruction::Branch` corresponds to a binary branch,
- `Instruction::Heuristic(w)` corresponds to replacing the current heuristic, and
- `Instruction::Dedup(d)` is used to identify "equal states", as coroutines in Rust cannot be directly compared. Assuming `d` is uniquely identifying the inner state

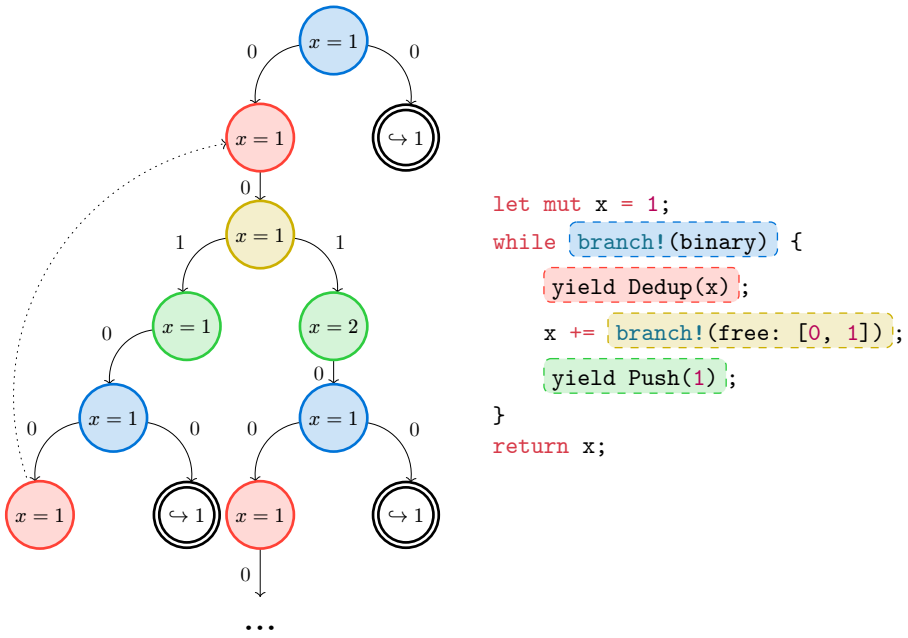

figure 4.1: A simple example for a coroutine representing a graph that can be traversed by A* (or Dijkstra since it never sets any heuristics)

of the coroutine, this reduces unnecessary recomputation. As an example, without this feature, finding the Levenshtein distance[33] between two strings of lengths n and m wouldn't be $O(nm)$, but exponential.

Using that basic setup and some macros that make branching more intuitive, the A* runtime can solve very readable representations of optimization problems as seen in figure figure 4.1.

A more involved example is calculating the Levenshtein distance (in $O(nm)$ time, the Levenshtein distance is returned by the runtime as the total cost):

Rust
```rust
fn levenshtein<'a>(
    mut a: &'a [u8],
    mut b: &'a [u8]
) -> impl Explorer<usize, (usize, usize)> + 'a {
    move |_| {
        while let (&[l, ref ra @ ..], &[r, ref rb @ ..]) = (a, b) {
            yield Instruction::Dedup((a.len(), b.len()));
            yield Instruction::Heuristic(a.len().abs_diff(b.len()));

            (a, b) = branch!(weighted: [
                (1, (ra, b)),
                (usize::from(l != r), (ra, rb)),
                (1, (a, rb)),
            ]);
        }

        yield Instruction::Push(a.len() + b.len());
    }
}
```

[33] An example follows later in this section.

4.1.2. Other possible APIs for A*

It has to be noted that copyable coroutines, or features that are similar or can emulate the same behaviour, are a rare feature in popular programming languages. For example, while both Python and C++ have coroutines (in the case of Python they are called "generators"), it is impossible to create an independent copy of a generator.

Languages that can benefit from the simplified syntax are, as a result, mostly constrained to Rust and functional programming languages with `do` notation such as Haskell or Idris (as coroutines are describable as a monad or monad transformer).

An alternative user-facing API for the underlying A* approach for optimization problems is the use of combinators, which can even allow for implicit deduplication based on a comparable state[34], for example the `levenshtein` example above could potentially be expressed like so in Python:

Python
```python
def levenshtein(a, b):
    return const((a, b))\
        .then_loop(
            lambda pair:
                continue_ if pair[0] and pair[1] else break_with(pair),
            id()
                .add_heuristic(
                    lambda pair: abs(len(pair[0]) - len(pair[1]))
                )
                .then_branch(lambda pair: [
                    weighted(1, (pair[0][1:], pair[1])),
                    weighted(
                        int(pair[0] != pair[1]),
                        (pair[0][1:], pair[1][1:])
                    ),
                    weighted(1, (pair[0], pair[1][1:])),
                ])
        )\
        .then_push(lambda pair: len(pair[0]) + len(pair[1]))\
        .start_with((a, b))
```

4.2. Pattern implementation

The pattern parsing and evaluation (which is built in terms of Dijkstra as described in section 4.1) is implemented in `src/matcher.rs`. It compiles a pattern into a sequence of "instructions". These are either

- the instruction to match an input item with an optional pattern item (to allow matching any item with ., see the difference between equations equation 3.3 and equation 3.4), or
- the instruction to skip some number of items, optionally with branching (if it branches it continues at both the next instruction and the instruction skipping

[34]Monadic implementations can't really benefit from this as they have to implement the bind operation, which "hide" their state in the function. However, applicative functors still provide some syntactical benefit with applicative do while retaining comparable states.

the given number of instructions) to allow for alternatives and item classes (see equation equation 3.10).

Its application to an input is defined in `Matcher::explore_on` by taking steps through instructions, parameterized over

- the input (which is defined by an initial state, and a function returning possible next states together with input items),
- possibly an alternative initial state in the input,
- the initial "user data", which may be used to collect information about matches for instance,
- shared data for the next two parameters to allow the constraints by copyable coroutines to be met without having to copy data into both closures,
- the closure describing how the user data may be altered and what weight a given (partial) match has (so a match containing *at least one* of the pattern item type and the input item type), and
- the closure describing how to end the matching process after there are no more instructions to process, by returning a weight and output based on the remaining input and current user data.

This allows for a shared implementation for section 4.4 and section 4.5 regardless of the semantics of what a pattern is applied on and how it is applied, and independent of the amount and kind of data that should be recorded about those matches.

It deduplicates based on its instruction pointer into the instruction sequence, the input state and user data.

4.3. Input preparation

Depending on the transcription type (ARPABET or IPA), a word undergoes one of the following two procedures (with a very basic port of the PanPhon library in Python [16] to Rust):

	for IPA:	**for ARPABET:**
1.	Apply configured substitutions	-
2.	Parse IPA into feature vectors based on data from [16][35]	Parse ARPABET (see section 3.8) into feature vectors based on [16]
3.	Extract existing syllable boundaries	-
4.	Create the mapping between spelling and pronunciation (see section 4.4)	
5.	Split syllable where syllable boundaries are not yet known (see section 4.4)	
6.	-	Retroactively applying the known stress information to the found syllable boundaries

This is implemented in `src/system.rs`, `src/ipa.rs` and `src/arpabet.rs`.

[35]The parsing logic also relies on [16], by collecting as many initial diacritics as possible, then collecting the main letter or combination of letters, and then collecting diacritics following them.

4.4. Orthography mapping and syllable splitting

Orthography mapping is implemented by matching pairs of orthography patterns step by step using the approach described in subsection 4.1.1. It also includes a heuristic based on the known maximum length of matches of patterns, by calculating how many patterns are at least required to match everything, and multiplying that by the weight of adding a new match.

Syllable splitting is implemented by creating subsequences for every possible onset of the orthography mapped input, between clusters of syllabic items separated by at least one syllabic-item. These are then checked against the weights as defined subsection 3.2.2 using the Dijkstra approach[36].

For ARPABET syllable boundaries between vowels are known due to its structure. For example "hiatus" /haɪˈeɪ.təs/ [8] is transcribed "HH AY0 EY1 T AH0 S" [10], which implies a syllable boundary between "AY0" (unstressed /aɪ/) and "EY1" (stressed /eɪ/). However in the case of IPA, there is no attempt made to detect syllable boundaries without consonants, though extending the software by adding a pattern for possible nuclei is possible.

The implementations can be found in `src/language.rs` in `Language::map_ortho` and `Language::split_syllables` respectively.

4.5. Generating chords

Generating chords poses a challenge due to the sheer number of possible combinations, especially for longer words. The number of ways in which syllables can be combined to chords is already exponential (just counting which ones are combined results in $2^{\{n-1\}}$ possibilities for n syllables, as each syllable boundary may or may not be a chord boundary). This is even true if there is a maximum number of syllables per chord (as chords have a maximum number of user-defined rules that can be applied), as long as it allows more than one syllable, as having a boundary at every second boundary still leaves half the boundaries independent of each other.

As is, the proposed software does not attempt to solve this exponential growth, as it is in practice acceptable with the word lengths found in [10]: To iterate on the theory, one can potentially limit the word list to ignore longer words and add them back in later, and building the final dictionary still takes less than an hour on the consumer machine described in appendix B.

There is another, more pronounced source of exponential runtimes related to chord splits however: The precise translation for a chord does not influence any chords that follow it (except for consumed items outside of its syllable range), but with the obvious approach of collecting chords and using their aggregate in deduplication tokens, future choices cannot be deduplicated. The alternative of not

[36]It does not include a heuristic as patterns do not implement heuristics themselves, and the inputs for those patterns are very small either way.

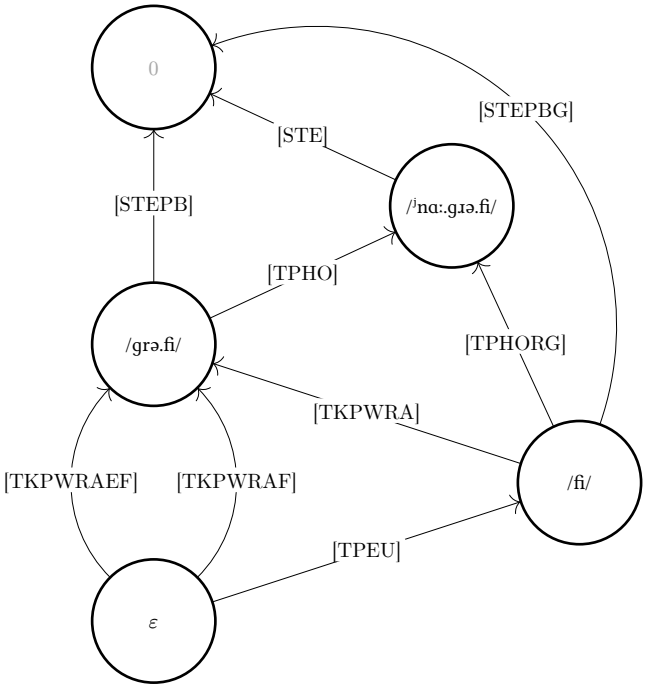

figure 4.2: A possible graph for the word "stenography" /stəˈnɑː.g*rə.fi/ [8], 0 signifies no older split exists, and ε denotes the end of the word. This graph does not include the costs that are associated with each edge.

including earlier chords in those tokens stops chords that are discovered later during the A* traversal to ever be returned. As such, it is not acceptable as a result.

The current implementation solves this by utilizing side effects in the coroutine to implement its own deduplication for every translation split (by potentially yielding `Instruction::Fail` to remove a state from consideration since it is a duplicate). It records possible chords based on the relevant parts of translation state for the given split (found in `SplitInfo`). An entry for a possible chord also includes the current weight and a reference to the last split, if it wasn't the first chord. As such, it records a directed acyclic multi-edge graph of splits with chords as edges, as seen in figure figure 4.2.

This graph can then be traversed (in reverse order of the resulting chords) to find all possible combinations of chords without having to recompute them in the A* traversal, by maintaining a priority queue of known split states together with accumulated chords. To actually find them in the correct order, no accumulated chord sequence may be returned from the iterator until the A* traversal has also reached the weight of that item, as otherwise other lower weight items may still exist.

This part of the implementation does not utilize any heuristics, as finding a good heuristic is difficult. It is possible to create a heuristic by assigning an arbitrary cost to considering the next syllable, and using the known total weight spent per syllable joining, however it does not increase performance and makes a correct

implementation of the final iterator much more difficult, as edges from split infos may not necessarily be discovered in the order of their actual weight anymore.

Additionally, due to performance considerations it selects only the first possible solution for matching patterns, which does not always result in optimal chords as there might be multiple ways of matching a single pattern against the input[37].

The implementations can be found in `src/system.rs`:

- `WordContext` implements the chord alternative bookkeeping, as well as methods that do not change any state.
- `TranslationState` implements the state that is not specific to the current chord, such as which input items were consumed.
- `ChordState` implements the state specific to assembling a single chord.
- `PossibleTranslationsIterator` implements the final iterator that returns full chord sequences including disambiguator usage.
- `System::possible_translations` implements the coroutine (which contains the control flow for creating chords) and returns an instance of the iterator.

4.6. Generating a full dictionary

Generating a full dictionary can be achieved with the following steps:

1. All words are translated into up to 128 possible chord sequences each using the `translate-all.py` script, piping the output (stdout) into a file, with the following arguments:
   - the `translate-single-arpa` binary (which can be built using `cargo build --release`)
   - the system configuration (for example `english-plover.toml`)
   - a UTF8 encoded version of CMUdict [10][38]
   - optionally an upper bound for memory usage (the supported suffixes are `K`, `M` and `G`)[39]
2. Word frequency data is collected (see subsection 4.6.1).
3. The `generate-dictionary` binary is invoked, piping the output (stdout) into the file for the final dictionary, with the following arguments:
   - the collected chord sequence translations from step 1
   - the compiled word frequency information from step 2 (an empty file also works, in that case no word frequency adjustments will take place)

[37] This is the reason for the duplicated [SPWR] (//ɪntəɹ/, /əntɘ*ɹ/, each of those with an additional unspecified consumed item) rule, as the final optional item doesn't add any extra weight, resulting in two solutions with the same weight if the input doesn't end in /te*r/. Of those, only one is ever selected, even though the other might be more applicable given the input word.

[38] It can be converted using `iconv -f windows-1252 cmudict-0.7b -t utf8 -o cmudict-0.7b-utf`.

[39] This actual memory usage may significantly overshoot that suggested upper bound, as the implementation tries to estimate the maximum number of A* nodes it can allow in the queue, and uses that as the actual limiting factor instead.

- any number of JSON files containing chord sequences that shouldn't be included in the dictionary, such as sequences controlling formatting or existing briefs (each file has to be either an array of strings or an object, in which case its keys are used)

The result is a JSON dictionary compatible with Plover [7].

`translate-all.py` is a Python script that filters out CMUdict entries for symbols and converts the remaining words into lowercase[40], invoking the translation binary for each word and recording the results. It also attempts to parallelize the translation process by running as many instances as there are threads (usually twice the number of cores).

`generate-dictionary` uses the following (approximation, it is not guaranteed to result in the lowest possible weight for all entries) algorithm to create a dictionary (implemented in `src/dictionary.rs`):

1. The weights of all candidates are adjusted by the word frequency as described in section 3.9.
2. All words are sorted by their word count (ascending)[41].
3. The algorithm attempts to insert the best candidate for a word (without immediately replacing another entry).
4. If this does not succeed, and the weight for displacing the entry in the dictionary to its next candidate is less than the weight of displacing the current entry, the two are swapped.
5. If it didn't succeed, regardless of swapping, steps 3 through 5 are repeated until it either succeeds or there are no more possible candidates.
6. Steps 3 through 5 are repeated until all words are processed.
7. Filler entries are inserted for free entries, always taking the lowest weight word for each chord sequence.
8. Prefix entries are generated for free entries (that weren't filled originally or as filler), always taking the lowest weight word for each chord sequence prefix.

The included generated dictionary excludes all of `commands.json` from Plover. The used word frequency data is discussed in subsection 4.6.1.

4.6.1. Word frequency data

Word frequency data is expected to contain one word (that does not contain a space), followed by a space, followed by the number of occurrences, per line (empty lines are ignored). These may be supplied from any number of sources.

The repository also includes the script `collect-wordcounts.py`, which counts the occurrences of words filtered by the word list in CMUdict [10]. Its first argument is a UTF8 version of CMUdict, and every remaining argument is a text file to count occurrences of words in.

[40] It also removes diacritics for the invocation of the translation binary.

[41] This way more common words are more straight forward in their chord placement as they get added last.

The generated dictionary uses the word frequencies as found in the 1M 2016 Wikipedia corpus from [17], compiled into the expected format with:

```Shell
sed '/ /d;s/^[^\t]*\t/;s/\t/ /' eng_wikipedia_2016_1M-words.txt >freq
```

4.7. Example configuration

The provided language definition for English is adapted from [18] for the valid onsets and [19] for the spelling to sound correspondences, but modified based on experimentation with examples from both Wiktionary transcriptions [20] and transcriptions in CMUdict [10].

The provided theory definition for the Plover theory is adapted from [12] and [14], together with personal experience of the author [18] was also used to double check that vowels were correct, though experimentation using transcriptions from both [20] and CMUdict [10] was also instrumental.

5. Evaluation

5.1. Basic limitations

As is, deliberate choices were made in trying to keep the software simple to keep the focus on estimating the viability of this approach:

- The conceptualisation of syllable boundaries in the current implementation is very basic. Ambisyllabic consonants aren't handled for instance, resulting only in possible chord sequences like [TKEU/TPHER] (/dɪ/, /nəɹ/) for "dinner" instead of possible sequences like [TKEUPB/TPHER] (/dɪn/, /nəɹ/). The syllable boundaries found in the Plover theory tend to default to the latter or moving the consonant to the preceding syllable. For example, the dictionary contains [TKEUPB/ER] (/dɪn/, /əɹ/), [TKEUPB/TPHER] (/dɪn/, /nəɹ/), [TKAOERPB] (/diːɹn/) and [TKEURPB] (/dɪɹn/), but not [TKEU/TPHER] (/dɪ/, /nəɹ/) [1].
- Chord sequence candidates are generated completely independently for each word, and their insertion into the dictionary does not take common prefixes or suffixes into account. However, affixes tend to have reasonably stable patterns in most stenography theories, while the proposed software does not attempt to make them more likely. A possible solution would be adding the expected chords or parts of chords to the system definition file and using unless-rules, but a better solution is likely to explicitly handle affixes when generating chords.[42]
  - Additional to the above is the problem that CMUdict [10] contains different forms of the same word (for example it contains both "write" and "writing" separately), which results in potentially inconsistent translations between them. Plover can automatically apply a lot of suffixes correctly in most cases, including /ɪŋ/, /əd/, reducing the benefit of including different word forms unless the word is irregular in pronunciation or spelling.
- All entries are always lower-case. A possible solution for correct capitalisation would be relying on the word frequency list or some other source, however, as is, the software takes its words from CMUdict [10], which does not make any case distinctions (in fact, all words are upper-case).
- As is, the dictionary generation does not explicitly create so-called "briefs" (which are single chords that stand for very common or long words).
- While in some cases one entry per known pronunciation makes sense (which is how the software operates), this can lead to conflicts between a word and itself, or

[42]Some prefixes and suffixes have been added, though they are very limited.

result in conflicts where dropping a less common pronunciation of a word would be preferable, where it instead tries new translations.
- It requires a lot of memory, in fact the word "supercalifragilisticexpialidocious" failed due to running out of memory in the generated dictionary.
- Not all entries may be included due to the approach for building the final dictionary. With the provided language and system definitions, CMUdict 0.7b [10] and the word frequencies from [17], the words "aah" and "uhh" are missing[43]. If word frequency data is omitted, no word is missing, but the results (especially for common single syllable words) are much worse.
- The current approach relies on filling the dictionary with all chord sequences that were found, which improves usability due to multiple alternative chords being present, but also leads to significantly bigger file (209MiB) than the Plover dictionary [1] (4.1MiB, about 2% of the size) that takes about 19s to load in Plover [7] on the development machine (see appendix appendix B).

5.2. Qualitative evaluation of example translations

To evaluate example translations in the final dictionary, the dictionary without filler and prefixes was generated and formatted with the command line tool `jq`. Then 10 random entries were extracted with `shuf -n 10 final-dictionary-core.json` to be examined more closely, together with the 10 most common words in the used word frequency list [17] and the 10 most common multisyllabic words. The resulting list can be found in appendix appendix A.

Outside of words that either aren't in the Plover dictionary at all or are only included as briefs, the only examples (using word frequency data) that don't contain all of Plovers [1] canonical entries (which are entries that are neither irregular, briefs nor miss-strokes) are:
- "was", which doesn't include [WUZ] /wʌz/ and [WAS] /wɑz/ (with an orthographic [-S]), but does include [WAZ] /wɑz/. The final dictionary contains no filler entries for "was".
- "also", which doesn't include the two chord version [AL/SO] /æl.sɑ/ or [Al/SOE] /æl.soʊ/ (both of which are not phonetically accurate), but does include [AULS] /oʊls/ which the Plover dictionary maps to "always" instead. The generated dictionary uses [AULZ] /ɔlz/[44] and [AUL/WAEUZ] /ˈɔl.weɪz/ for "always".
- "other", which uses [*URT] (/ø*rθ/) instead of [O/THER] /ʌ.ðəɹ/. While [O/THER] is part of the candidates, in the filled dictionary it is displaced by the filler "auther" and "rother" instead.
- "citation" uses a single chord version that conflicts with "station", and disambiguates "station" instead. The Plover translations include a [KR] rule for

[43] "aah"s ("AA1") final conflict was "ahh" ("AA1") and "uhh"s ("AH1") was "ahh" ("AA1").
[44] the /ɔ/ is exaggerated to distinguish it from /ɑ/ which is also possible for [A]

orthographic “c”s, and entries with the /t/ in different syllables (/saɪt.eɪʃən/ and /saɪt.teɪʃən/ are both not present in the generated dictionary).

- “only” doesn’t include the orthographic entry [OPBL], but does include the phonetic Plover entry [OEPBL] /oʊnl/.
- “many” doesn’t include the [PHAEPB] /mæni/ translation, and instead includes [PH*EPB] /men/, [PHAEPB] instead maps to “mean”, and [PHA*EPB] to “meany”.
- “between” includes [PWAOE/TWAOEPB] /biː.twiːn/ (and also includes [PW/TWAOEPB] /b.twiːn/), whereas Plover uses [PWE/TWAOEPB] /bə.twɪːn/, which is present in the filled dictionary however. Plover also includes the phonetically inaccurate [TWEPB] /twen/, as well as the short versions [TPWAOEPB] /tbɪːn/ (which maps to “bettina” and “bottini” with and without [*]), and [TWAOEPB] /twɪːn/ (which maps to “tween” and “betweens”).
- “over” shares [OEFR] /oʊvr/, but doesn’t include [A*UFR] in the generated dictionary.
- “during” shares [TKURG] /dʊɹɪŋ/ and [TKAOURG] /duːɹɪŋ/, but includes [TKREUPBG] /dɹɪŋ/ instead of [TKAOUR/-G] /duːɹ.ɪŋ/ (which maps to “doering” with and without the dismabiguator), [TKAOURPBG] /duːɹŋ/ (which maps to “dering” and “doering”), [TKUR/-G] /dʊɹ.ɪŋ/ (which maps to “durgan” and “demurring”) and [TKURPBG] /dʊɹŋ/ (mapping to “durning” and “dunker” instead).
- “rereading” shares its translation with an implicit entry [RAOE/RAEGD], but doesn’t include any of the explicit entries due to the [RE] prefix not being explicitly listed in the configuration and the “ea” orthography feature disabling the [AOE] rule with unless-rules in favour of the [AE] rule.
- “contain” includes [TKAEUPB] /tkeɪn/, but not any of the Plover entries [KAUPB/TAEUPB] /kɔn.teɪn/[45] (which is missing), [KOPB/TAEUPB] /kɔn.teɪn/ (which maps to “container” and “containers” depending on the star key) and [TAEUPB] (which instead maps to “attain” and “retain”).
- “embezzler” contains shorter translations, but the filled dictionary contains the explicit [-EPL/PW-EZ/HR-ER].
- “midwesterner” includes [PHEU/TKW*EPBS] instead of [PHEUD/W*ES/TPHER], which is not present in the final dictionary.

Additionally, the following five (randomly selected) entries share no generated chord candidates with the Plover dictionary:

- “abu” only maps to [AB/SKWRU] in the Plover dictionary
- “indiscriminate” maps to:
  - [EUPB/TKEUS/SKREUPL/KR*EUPL/TPHA-T], [EUPB/TKEUS/KRAEUPL/TPHAT], [EUPB/TKEUS/KREUPL/TPHAEUT] and have their /s/ always attached to (at least) its preceding syllable, violating the maximum onset principle,

[45] the /ɔ/ is similarly exaggerated as for “also”

  - [EUPBD/SKREUPL/TPHAPBT] is irregular as there is no third /n/ in the word,
  - [EUPBDZ/KREUPL/TPHAEUT] does not match any pronunciation in CMUdict [10] with /eɪ/ in its final chord,
- "duchy" only maps to [TKUFP/KWREU]
- "spontaneously" maps to [SPA*EUPBLS] and [SPAEUPBLS] (which are briefs) as well as the normal entry [SPOPB/TAEUPB/SHREU]. And while this entry isn't generated, the similar [SPOPB/TAEUPB/SHRAOE] is (differing between [EU] /ɪ/ and [AOE] /iː/).
- "grumpy" uses [TKPWRUFRP/PEU], which has its /p/ attached to both syllables, and [TKPWRUPL/PEU], which ends in /ɪ/ instead of /iː/. The final dictionary doesn't contain the similar [TKPWRUPL/PEU] though, as it instead maps to "grumpier".

As seen in the examples above, the main differences are caused by the following problems

- Ambisyllabic consonants are only ever assigned to the next syllable, as per the maximum onset principle.
- Affixes or word forms often result in other chord sequences since they either
  - displace other entries that fit the translation better even though they can be generated using prefix and suffix entries, or
  - don't fit the phonetic rules.
- [SKWR] textipa{J} or [KWR] textipa{j} are (inconsistently) used in the Plover dictionary to join a chord to the previous one if there are no initial consonants,
- Inconsistencies in the Plover dictionary also cause differences, though these are expected.

5.3. Briefs

This approach is able to automatically generate reasonable brief candidates for a given word.

`english-plover-brief.toml` is a configuration file based on the original system definition, but with the weights changed, to force a single chord and make ignoring items cheaper, and without the restriction on rules in the nucleus.

For example, for the word "systematic" using the CMUdict transcription "S IH2 S T AH0 M AE1 T IH0 K" [10], it generates the following candidates (excluding dismabiguator usage):

- [STPHAEUBG] (/stmæ.ɪk/) at weight 0.019
- [STPHAT] (/stmæt/) at weight 0.02
- [SEUFPLT] (/sɪsmt/) at weight 0.02
- [STPHABG] (/stmæk/) at weight 0.02
- [STPHABGT] (/stmækt/) at weight 0.021

Due to the optimization of only taking the first possible match of a pattern, solutions like [STPHEUBG] (/stmɪk/) and [STPHEUBGT] (/stmɪkt/) are not found since the pattern for /ɪ/ matches the first /ɪ/ at "s**y**stematic". It also has no incentive to drop the /t/, as it always fits in the steno order, even though for example [SPHAT] (/smæt/) might be more intuitive to some users as the /t/ occurs in an unstressed syllable.

As a comparison point, the briefs offered in the Plover dictionary [1] are [SPH-BG] (/sm/, /k/) and [SPHA*EBG] (/smæ.ik/). However, briefs are a matter of personal preference, since they have to be easy to memorize or guess for the person using them.

5.4. Comparison with the original Plover dictionary

5.4.1. Similarity between JSON dictionaries

A measure of the similarity can be calculated with the `count-similarities.py` script.

For the generated dictionary, it results in the following values:

metric	value	explanation
word count	$40,454$[46]	the number of words shared between both dictionaries[47]. This is the comparison point for all non-scaled values unless otherwise specified.
word count, scaled	$18,694,251$	the sum of the word frequencies for each shared word. This is the comparison point for all scaled values unless otherwise specified.
fully covered	$6,248$ $\approx 15.44\%$	number of words / sum of frequencies of words for which all chord sequences from Plover [1] are present in the generated dictionary
fully covered, scaled	$1,258,367$ $\approx 6.73\%$	
partially covered	$10,566$ $\approx 26.12\%$	number of words / sum of frequencies of words for which some chord sequences from Plover are present
partially covered, scaled	$7,723,815$ $\approx 41.32\%$	
not covered	$23,640$ $\approx 58.44\%$	number of words / sum of frequencies of words for which both dictionaries share no chord sequences
not covered, scaled	$9,712,069$ $\approx 51.95\%$	

[46] This is about 32.2% of non-symbol CMUdict entries, and about 59.3% of Plover entries (removing entries that contain an opening curly brace, as those are likely formatting instructions).

[47] The value of this depends mainly on CMUdict [10], and on the entries that couldn't find any free entries (see section 5.1).

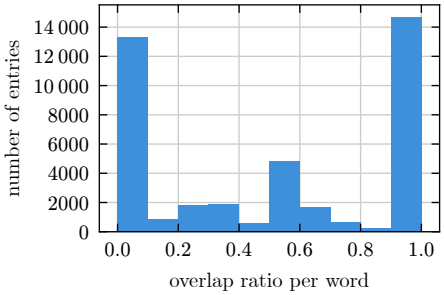

figure 5.1: Histogram of the ratio of included chord sequences per word

metric	value	explanation
translation count	102,682	Number of translations in the Plover dictionary for which words exist in the generated dictionary
translation count, scaled	77,606,062	The translation count scaled by word frequencies
covered translations	20,706 ≈ 20.17%	Number of translations that are shared between both dictionaries, percentage compares to the translation count
covered translations, scaled	10,156,788 ≈ 13.09%	Number of shared translations scaled by word frequencies

As can be seen, the similarity to the Plover dictionary is low, even when correcting for word frequencies.

5.4.2. Overlap between candidates and Plover translations

The lack of covering is caused by the translations into chord sequences, as can be seen by comparing the generated chord candidates with the Plover dictionary directly: As calculated in the Jupyter Notebook, 32.57% of words shared between CMUdict [10] and the Plover dictionary [1] don't share any chord sequences.

In general, the ratio of included chord sequences per word tends to be low, both in general (see figure 5.1) and grouped by the total number of translations in the Plover dictionary [1] (see figure 5.2).

5.5. Performance

The system specifications of the machine everything was executed on can be found in appendix appendix B.

Translating all entries took 31min 39.94s, consuming up to 23.2GiB according to GNU `time` (version 1.9, [21]). According to measurements done with `hyperfine` (version 1.18.0, [22]), creating the final dictionary takes $10.295s \pm 0.115s$ (over 20 runs, the range is [10.147s, 10.499s]).

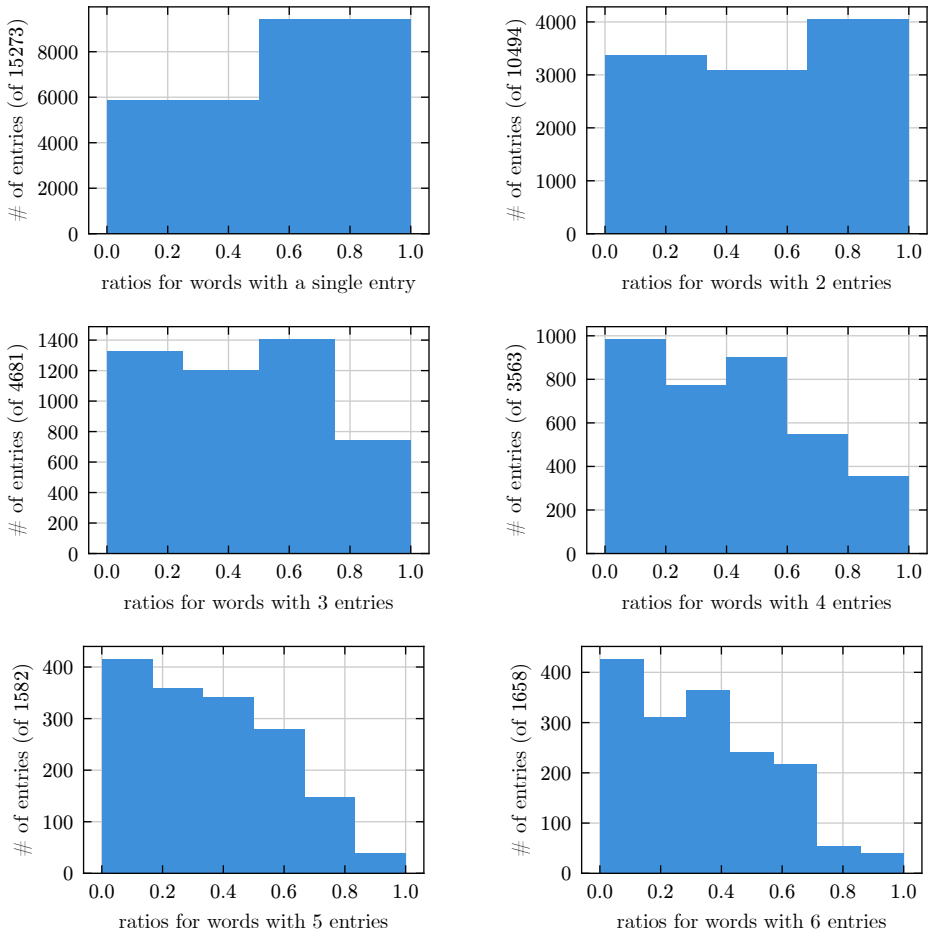

figure 5.2: Histograms of the ratio of included chord sequences per word, per translations in the Plover dictionary [1]

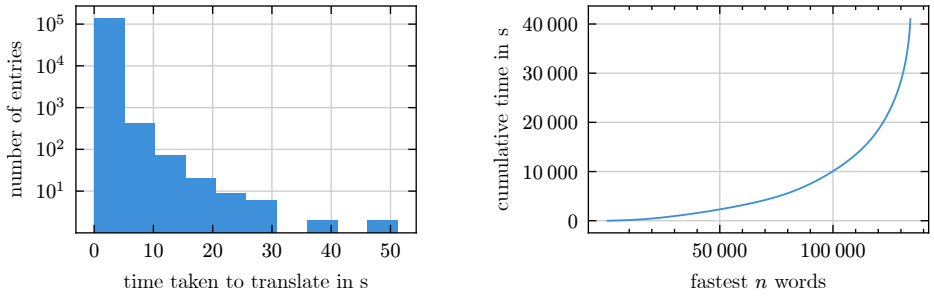

figure 5.3: Histogram of the time spent per word (using a log scale for the number of entries) and cumulative time spent, excluding the entry for "antidisestablishmentarianism" which took 647.05s

93.72% of chord candidates were produced in a second or less, with 48.08% of the total runtime spent on the remaining entries. As seen in figure figure 5.3, very short translations make up an overwhelming majority.

As seen in figure 5.4, as expected, the time spent on a word grows exponentially in respect to the length of that word.

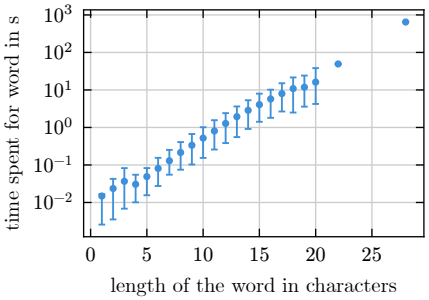

figure 5.4: Average time spent on words plotted against the length of these words[48] with a log scale, with error bars showing the 10^{th} and 90^{th} percentiles

[48] Different pronunciations count as separate words for this purpose

6. Conclusion

The results as found in chapter 5 are mixed, but promising especially if ambisyllabic consonants get handled similarly to how they do in the existing Plover dictionary [1]. Including the possible improvements to syllabification, there are a number of possible improvements that can be made, including those addressing the other basic limitations explained in section 5.1.

The amount of steps, limitations of the configuration format (TOML) and general user experience, including improving performance and giving feedback about conflicts, is a potential hindrance for actually making the proposed software accessible to stenographers who might not necessarily be comfortable with the amount of technical knowledge currently required.

In terms of maintainability, it may potentially make sense to either revise the chord generation implementation fundamentally, or to develop a safe (in terms of correctness) abstractions for "list element deduplication", as the current approach is error prone due to the way it breaks the assumptions of the A* runtime and depends on the order of the graph traversal. It may even be reasonable to reduce complexity by creating a more explicit algorithm for generating chords that could as a result be more intuitive as well, as the pattern implementation doesn't depend on its caller to do its own A* graph traversal and is still usable, allowing the usage of mixed patterns.

Additional possible improvements include allowing weight adjustments based on the first onset and the last coda, especially for brief generation, and in general more differentiated weights where additional context could apply. Another example is adjusting chord splitting weights based on word frequency, this could allow less frequent words to default to longer but easier to guess chord sequences.

Brief generation, while already usable, could be even better by reintroducing some branching based on the pattern matching, for example based on a weight threshold. Expanding on its ability to create briefs by automatically splitting the dictionary into a brief only and a normal part (with some overlap between them), based on word frequency and maybe length, could also prove to be a fruitful endeavour.

Further research is necessary to prove the optimization based formulation to be capable of creating high quality dictionaries compared to existing ones, or to disprove the same, though a better comparison point than the existing dictionary in

terms of comparing entries is needed, as a good number of entries that don't exist in the generated dictionary are miss-stroke entries, irregular entries or briefs.

A. Sample of generated dictionary entries

word	**occurrences**	**chord sequence with word frequency adjustments**	**chord sequence without word frequency adjustments**	**chord sequence in the Plover dictionary [1]**
"the" common	1,270,286	[THU], [THE], [THAOE]	[*T], [E], [TH/-E]	[-T]$^{\text{brief}}$ and 4 miss-stroke entries
"of" common	714,031	[OF], [UF]	[O/*F], [O*F]	[*F]$^{\text{brief}}$, [-F]$^{\text{brief}}$
"and" common	576,297	[TPH-D], [APBD]	[APB/-D], [TPH-D]	[APBD], [SKP]$^{\text{brief}}$, [STK]$^{\text{brief}}$, [STKP]$^{\text{brief}}$ and 16 miss-stroke entries
"to" common	518,163	[TO], [TEU], [TAOU]	[T*], [O], [-T/AOU]	[TO], [O]$^{\text{brief}}$, [TPO]$^{\text{miss-stroke}}$
"in" common	443,758	[EUPB], [-PB]	[*PB], [HL]	[TPH]$^{\text{brief}}$
"a" common	415,063	[A], [AEU]	[TH], [HEU]	[AEU] and 4 miss-stroke entries
"is" common	231,607	[*EUZ], [EUZ]	[*EUZ], [EU/-Z]	[S]$^{\text{brief}}$
"was" common	194,442	[WAZ], [WAUZ]	[WAZ], [WAS]	[WAS], [WAZ], [WUZ], [-FS]$^{\text{brief}}$ and 4 miss-stroke entries
"that" common	166,493	[THA*T], [THAT]	[THAT], [THA*T]	[THA]$^{\text{brief}}$ and 7 miss-stroke entries
"for" common	164,107	[TPOR], [-FR], [TPR-R]	[TPRO], [TPO/-R], [TPO/*R]	[TPO*R], [TPOR] and 4 miss-stroke entries
"also" multisyllabic	53,054	[AULS]	[A*ULS]	[AL/SO], [AL/SOE], [HR-S]$^{\text{brief}}$

word	**occurrences**	**chord sequence with word frequency adjustments**	**chord sequence without word frequency adjustments**	**chord sequence in the Plover dictionary [1]**
"other" multisyllabic	39,967	[*URT]	[O/THER]	[O/THER], [OER][brief], [OT/ER][irregular]
"citation" multisyllabic	32,087	[STAEUGS]	[STA*EUGS]	[KRAOEU/TAEUGS], [KRAOEUT/AEUGS], [SAOEU/TAEUGS], [SAOEUT/AEUGS], [SAOEUT/TAEUGS], [KRAOEUFT/AEUGS][miss-stroke]
"only" multisyllabic	30,434	[OEPBL]	[O*EPBL]	[OEPBL], [OPBL], [HOEPBL][irregular]
"many" multisyllabic	25,796	[PHEPB]	[PHE/TPHAOE]	[PHAEPB]
"after" multisyllabic	23,606	[AFRT]	[AF/TER]	[AF][brief], [AFT][brief]
"between" multisyllabic	23,512	[PWAOE/TWAOEPB], [PW/TWAOEPB]	[PWAOE/TWAOEPB], [PW/TWAOEPB]	[PWE/TWAOEPB], [TPWAOEPB], [TWAOEPB], [TWEPB] and 5 miss-stroke entries
"over" multisyllabic	22,657	[OEFR]	[O*EFR]	[A*UFR], [OEFR]
"about" multisyllabic	22,029	[PWOUT]	[PWOU/UT]	[PW][brief] and 2 miss-stroke entries
"during" multisyllabic	18,162	[TKREUPBG], [TKAOURG], [TKURG]	[TKUPBG], [TKREUPBG], [TKURG]	[TKAOUR/-G], [TKAOURG], [TKAOURPBG], [TKUR/-G], [TKURG], [TKURPBG] and 3 miss-stroke entries

word	**occurrences**	**chord sequence with word frequency adjustments**	**chord sequence without word frequency adjustments**	**chord sequence in the Plover dictionary [1]**
“rereading” random	2	[RAOE/ RAEGD]	[RAOE/ RAEGD]	[RAOE/RAOED/ -G], [RE/RAED/ -G] explicitly, [RE/ RAEGD] and [RAOE/REAGD] among others implicitly[49]
“left-brace” random	0	[HREFT/ PWRAEUS]	[HREFT/ PWRAEUS]	-
“contain” random	2303	[TKAEUPB]	[KOPB/ TAEUPB]	[KAUPB/ TAEUPB], [KOPB/ TAEUPB], [TAEUPB] and 2 miss-stroke entries
“heichelbech” random	0	[HAOEUBG/ PWHR/-BG]	[HAOEUBG/ PWHR-BG]	-
“embezzler” random	0	[EPH/ PWEFRL], [PH/ PWERLZ]	[PH/ PWEFRL], [PH/ PWERLZ]	[EPL/PWEZ/ HRER] explicitly, [EPL/PWEZ/-L/ ER] and [EPL/ PWEZ/*L/ER] implicitly[50]
“vogue” random	33	[SROEG]	[SROEG]	[SROEG], [SROEPBG]miss-stroke
“trebilcock” random	0	[TPWEUL/ KOBG]	[TPWEUL/ KOBG]	-
“midwest-erner” random	0	[PHEU/ TKW*EPBS]	[PHEU/ TKWEFRNT]	[PHEUD/W*ES/ TPHER] explicitly, more implicitly

[49] [RE] as a full chord is emits a prefix that can attach to any word, and [-G], both as a full chord and key (in which case it is said to be “folded in”) can attach an /ɪŋ/ ending to any word, so “read”, “reading” and “reread” entries can all be built up to “rereading”.

[50] through the [ER] suffix on “embezzle”

word	**occurrences**	**chord sequence with word frequency adjustments**	**chord sequence without word frequency adjustments**	**chord sequence in the Plover dictionary [1]**
“pulverized” random	10	[PUFL/ RAOEUDZ]	[P*UFL/ RAOEUDZ]	only implictly with [PUFL/AOEUDZ], [PUL/SRE/ RAOEUDZ], [PUL/ SRER/AOEUDZ], [PUFL/RAOEUZ/ -D], [PUL/SRE/ RAOEUZ/-D], [PUL/SRER/ AOEUZ/-D]
“muzzy” random	0	[PH*UZ]	[PHUZ]	-

B. Used PC to benchmark and test the software

Operating system: Arch Linux, kernel version 6.6.2-arch1-1

Used persistent storage: ext4 on a 898.5GiB partition on a "Samsung SSD 970 EVO 1TB"

Memory:

- 48GiB of DDR4
  - 2 × 8GiB 3600MHz "F4-3600C18-8GVK"
  - 2 × 16GiB 3600MHz "3600 C20 Series"
- 32GiB of swap on zram with zstd
- 32GiB swap on the same persistent storage as the main partition

Processor (output of `lscpu`, with manual line wrapping for printing):

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         48 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  24
  On-line CPU(s) list:   0-23
Vendor ID:               AuthenticAMD
  Model name:            AMD Ryzen 9 5900X 12-Core Processor
    CPU family:          25
    Model:               33
    Thread(s) per core:  2
    Core(s) per socket:  12
    Socket(s):           1
    Stepping:            0
    Frequency boost:     enabled
    CPU(s) scaling MHz:  55%
    CPU max MHz:         4950.1948
    CPU min MHz:         2200.0000
    BogoMIPS:            7403.67
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic
                         sep mtrr pge mca cmov pat pse3 6 clflush
                         mmx fxsr sse sse2 ht syscall nx mmxext
                         fxsr_opt pdpe1gb rdts cp lm constant_tsc
                         rep_good nopl nonstop_tsc cpuid
                         extd_apicid aperfmp erf rapl pni
                         pclmulqdq monitor ssse3 fma cx16 sse4_1
```

```
                        sse4_2 x2apic mov be popcnt aes xsave
                        avx f16c rdrand lahf_lm cmp_legacy svm
                        extapic cr8 _legacy abm sse4a
                        misalignsse 3dnowprefetch osvw ibs
                        skinit wdt tce to poext perfctr_core
                        perfctr_nb bpext perfctr_llc mwaitx cpb
                        cat_l3 cdp_ l3 hw_pstate ssbd mba ibrs
                        ibpb stibp vmmcall fsgsbase bmi1 avx2
                        smep bmi2 erms invpcid cqm rdt_a rdseed
                        adx smap clflushopt clwb sha_ni xsa
                        veopt xsavec xgetbv1 xsaves cqm_llc
                        cqm_occup_llc cqm_mbm_total cqm_mb
                        m_local user_shstk clzero irperf
                        xsaveerptr rdpru wbnoinvd arat npt lb rv
                        svm_lock nrip_save tsc_scale vmcb_clean
                        flushbyasid decodeassists p ausefilter
                        pfthreshold avic v_vmsave_vmload vgif
                        v_spec_ctrl umip pku ospke vaes
                        vpclmulqdq rdpid overflow_recov succor
                        smca fsrm debug_swap
Virtualization features:
  Virtualization:       AMD-V
Caches (sum of all):
  L1d:                  384 KiB (12 instances)
  L1i:                  384 KiB (12 instances)
  L2:                   6 MiB (12 instances)
  L3:                   64 MiB (2 instances)
NUMA:
  NUMA node(s):         1
  NUMA node0 CPU(s):    0-23
Vulnerabilities:
  Gather data sampling: Not affected
  Itlb multihit:        Not affected
  L1tf:                 Not affected
  Mds:                  Not affected
  Meltdown:             Not affected
  Mmio stale data:      Not affected
  Retbleed:             Not affected
  Spec rstack overflow: Vulnerable: Safe RET, no microcode
  Spec store bypass:    Mitigation; Speculative Store Bypass
                        disabled via prctl
  Spectre v1:           Mitigation; usercopy/swapgs barriers and
                        __user pointer sanitization
  Spectre v2:           Mitigation; Retpolines, IBPB
                        conditional, IBRS_FW, STIBP always-on,
                        RS B filling, PBRSB-eIBRS Not affected
  Srbds:                Not affected
  Tsx async abort:      Not affected
```

C. The commands used to generate the dictionary

```shell
# Building the Rust code
cargo build --release
# Converting cmudict to UTF8
iconv -f windows-1252 cmudict-0.7b -t utf8 -o cmudict-0.7b-utf8
# Translating all entrie
python3 translate-all.py \
    ./target/release/translate-single-arpa \
    english-plover.toml \
    cmudict-0.7b-utf8 >translated
# Preparing the word frequency data
sed '/ /d;s/^[^\t]*\t//;s/\t/ /' \
    eng_wikipedia_2016_1M-words.txt > wordfreq
# Generate the dictionary (assuming the Plover GitHub repository
# is cloned at ../plover)
./target/release/generate-dictionary \
    translated \
    wordfreq \
    ../plover/plover/assets/commands.json >dictionary.json
```

Optionally, tools like `jq` can be used to prettify the resulting JSON file.

To generate only the "core" dictionary (so the dictionary without filler), the two `for` loops in `src/dictionary.rs` after `let mut filler = HashMap::<...>::default();` have to be commented out.

Glossary

ambisyllabic:
an ambisyllabic consonant belongs to both the onset and the coda of the previous syllable (see subsection 2.1.2)

brief:
a single chord for a very common or very long word, which is expected to be memorized

CMU:
Carnegie Mellon University

CMUdict:
CMU Pronouncing Dictionary

coda:
the set of consonants in a syllable after the nucleus (see subsection 2.1.2)

diphthong:
a glide between vowels (see subsection 2.1.1)

GA - General American:
the "standard" American accent

IPA:
International Phonetic Alphabet

maximum onset principle:
the principle of syllable boundaries maximising the valid onset (see subsection 2.1.2)

nucleus:
a vowel, diphthong or syllabic consonant (see subsection 2.1.2)

onset:
the set of consonants in a syllable before the nucleus (see subsection 2.1.2)

RP - Received Pronunciation:
the "standard" British accent

syllabic:
a phonemic segment is syllabic if it is part of the nucleus of a syllable. Some consonants such as //l// can be syllabic (notated as //l//, see subsection 2.1.1)

USA:
United States of America

Bibliography

[1] Open Steno Project, "Plover main dictionary." Accessed: Dec. 09, 2023. [Online]. Available: https://github.com/openstenoproject/plover/blob/main/plover/assets/main.json

[2] Monkeytype, "Monkeytype." Accessed: Dec. 09, 2023. [Online]. Available: https://monkeytype.com/

[3] V. Dhakal, A. Feit, P. O. Kristensson, and A. Oulasvirta, "Observations on Typing from 136 Million Keystrokes," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*, ACM, 2018. doi: https://doi.org/10.1145/3173574.3174220.

[4] S. TAUROZA and D. ALLISON, "Speech Rates in British English," *Applied Linguistics*, vol. 11, no. 1, pp. 90–105, 1990, doi: 10.1093/applin/11.1.90.

[5] M. Knight, "CART, Court, and Captioning," 2010, Accessed: Dec. 09, 2023. [Online]. Available: https://www.openstenoproject.org/plover/

[6] T. Association for Court Reports and Captioners, "Registered Professional Reporter (RPR)." Accessed: Dec. 09, 2023. [Online]. Available: https://www.ncra.org/certification/NCRA-Certifications/registered-professional-reporter

[7] Open Steno Project, "Plover." Accessed: Dec. 09, 2023. [Online]. Available: https://www.openstenoproject.org/plover/

[8] Cambridge University Press & Assessment, "Cambridge Online Dictionary, English Pronunciation." Accessed: Dec. 10, 2023. [Online]. Available: https://dictionary.cambridge.org/

[9] D. Fallows, "Experimental evidence for English syllabification and syllable structure," *Journal of Linguistics*, vol. 17, no. 2, pp. 309–317, 1981, doi: 10.1017/S0022226700007027.

[10] Carnegie Mellon Speech Group, "CMU Pronouncing Dictionary (version 0.7b)." Accessed: Dec. 09, 2023. [Online]. Available: http://www.speech.cs.cmu.edu/cgi-bin/cmudict

[11] Wikipedia, The Free Encyclopedia, "ARPABET." Accessed: Dec. 09, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=ARPABET&oldid=1103994253

[12] T. Morin, "Art of Chording." Accessed: Dec. 09, 2023. [Online]. Available: https://www.artofchording.com/

[13] M. Kislingbury, "Magnum Steno." Accessed: Dec. 09, 2023. [Online]. Available: https://www.magnumsteno.com/

[14] Z. Brown, "Learn Plover!." Accessed: Dec. 09, 2023. [Online]. Available: https://www.openstenoproject.org/learn-plover/home.html

[15] Python Software Foundation, "\texttt{re} - Regular expression operations." 2023. Accessed: Dec. 09, 2023. [Online]. Available: https://docs.python.org/3/library/re.html

[16] D. R. Mortensen, P. Littell, A. Bharadwaj, K. Goyal, C. Dyer, and L. S. Levin, "PanPhon: A Resource for Mapping IPA Segments to Articulatory Feature Vectors," in *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, ACL, 2016, pp. 3475–3484.

[17] D. Goldhahn, T. Eckart, and U. Quasthoff, "Building large monolingual dictionaries at the leipzig corpora collection: From 100 to 200 languages.," in *Proceedings of the 8th International Language Resources and Evaluation (LREC'12)*, 2012, pp. 31–43.

[18] Wikipedia, The Free Encyclopedia, "English phonology." Accessed: Dec. 10, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=English_phonology&oldid=1188974201

[19] Wikipedia, The Free Encyclopedia, "English orthography." Accessed: Dec. 10, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=English_orthography&oldid=1188331084

[20] Wikimedia Foundation, "Wiktionary, the free dictionary." Accessed: Dec. 10, 2023. [Online]. Available: https://en.wiktionary.org/wiki/Wiktionary:Main_Page

[21] D. Keppel, D. MacKenzie, and A. Gordon, "GNU Time version 1.9." Accessed: Dec. 11, 2023. [Online]. Available: https://www.gnu.org/software/time/

[22] D. Peter, "hyperfine." [Online]. Available: https://github.com/sharkdp/hyperfine