

CSCI 2300 — Data Structures and Algorithms

Project 1 — Spell Checking

Spring 2006

Due Date

The due date is **Friday, February 10, 2006 by 11:59:59pm**. See the syllabus for late policies and academic integrity policies. See below for submission guidelines.

Spell Checking

Spell checking is the process of verifying that a particular word is spelled properly according to some dictionary. Spell checkers are used in many applications, including word processors (such as Microsoft Word), electronic dictionaries, and optical character recognition (OCR) systems that need to turn images of printed text (or even handwriting) into coherent text.

Spell checking itself is trivial, requiring only a simple lookup in a dictionary. However, most applications of spell checking also require that the spell checker provide a list of potentially correct spellings (“near matches”) when the word was spelled improperly. For instance, if I type “spelng” into an online dictionary, it will provide suggestions of similar words that I may have meant to type, including “spelling”, “spoiling”, “sapling”, and “splendid”.

Your task is to implement a spell checker that determines if a given word is spelled correctly based on a dictionary lookup. When the word is not spelled correctly, it will provide a list of similar-*sounding* words based on your implementation of the Metaphone algorithm, and ordered based on their edit distance from the string that the user typed.

We could use only the edit distance to find near matches, but by first using the Metaphone algorithm we achieve better results because it can find the word you are looking for even if you have no idea how to spell it. Additionally, we can isolate the set of near matches more quickly using the Metaphone algorithm.

Overview

Your program will be run from the command line, and will take a rule file (`rules.txt`, to be described later), a dictionary file containing about 10,000 words (`dictionary.txt`, from the web page), and some words. It will check the spelling of each word. If the word is spelled properly, it will print “Word ‘[the word]’ is spelled correctly.” Otherwise, it will print the word and its pronunciation, followed by a list of like-sounding words ordered by the edit distance (shown in brackets). For example, if your program is named `spellcheck.exe`, here is a sample program run:

```
spellcheck.exe rules.txt dictionary.txt spelng supress save
```

```
Word ‘spelng’ not in dictionary. It sounds like ‘SPLN’.
```

```
Perhaps you meant:
```

```
spelling [1]
sapling [2]
spellings [2]
spoiling [2]
splendid [5]
```

```
Word ‘supress’ not in dictionary. It sounds like ‘SPRS’.
```

Perhaps you meant:
 suppress [1]
 spares [3]
 suppressed [3]
 suppresses [3]
 surprise [3]
 surprises [3]
 suppressing [4]
 surprised [4]
 spurious [5]
 surprising [5]
 surprisingly [7]

Word ‘save’ is spelled correctly.

Read on to learn about the Metaphone algorithm, which will be used to isolate the set of near matches; and an edit distance algorithm, which will be used to rank the results (notice that “spelling” comes before “spoiling”). Fig. 1 gives an overview of the spell-checking process.

Please write your program to mimic the output given exactly. We will provide more example output on the project web page.

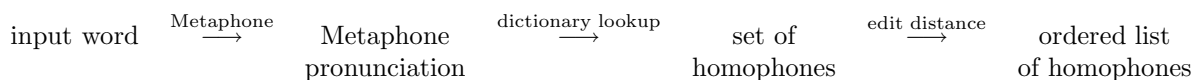


Figure 1: Overview: Sequence of Transformations for Spell Checking

The Metaphone Algorithm

The Metaphone algorithm, created by Lawrence Philips, takes a word and returns a very rough approximation of the sound of that word. The rough approximation eliminates the distinction between some characters; for instance, ‘S’ and ‘Z’ sound very similar, so the Metaphone algorithm maps them to the same sound (“s”). Some letters may make many different types of sounds: ‘C’, for instance, may make the “sh” sound (denoted by ‘X’) when it is part of “cia” (as in “social”), the “s” sound when it is following by ‘C’, ‘I’, or ‘Y’ (as in “since”), or even the “k” sound when it is by itself, or followed by a ‘K’ (as in “clack”).

The Metaphone algorithm maps every sound in a word down to one of the following sounds: *, B, X, S, K, J, T, F, H, L, M, N, P, R, @, W, Y. The ‘*’ represents the sound of a vowel at the beginning of the word, ‘X’ represents the “sh” sound, and ‘@’ represents the “th” sound. All of the other consonant sounds in the list sound like the consonant. Vowels not at the beginning of the word are considered silent.

See Fig. 2(a) for some examples of strings generated by the Metaphone algorithm for some common words. We (arbitrarily) limit the length of the Metaphone output strings to 4 characters, which has been found to produce better results in many cases than using longer strings.

Implementing Metaphone

To implement the Metaphone algorithm, you will use a table-driven algorithm to transform the input string into its Metaphone pronunciation. Fig. 2(b) contains several of the transformation rules used in the Metaphone algorithm. The complete set of rules will be provided on the project web page in the file `rules.txt`. Each of the rules contains a *pattern* that should be matched to the input word. Patterns are first matched at the first character of the string. When the *first* matching pattern is found (the order of the rules is significant), it first consumes part of the input string and then the pronunciation for that pattern is added to the

	Word	Metaphone	#	Pattern	Pronunciation
			1	A(HR)^	*
	spelling	SPLN	2	C(EIY)–	S
	sapling	SPLN	3	E	-
	since	SNS	4	I	-
	social	SXL	5	LL–	-
	clack	KLK	6	MB\$	M
	truth	TR@	7	N	N
	after	*FT	8	SC(EIY)–	S
	fatter	FT	9	S	S
(a)	Metaphones for example words.		10	TH	@

(b) Sample Metaphone transformation rules.
The complete set is given in `rules.txt`.

Figure 2: Metaphone pronunciations and transformation.

Metaphone output string. We then start with the next character in the word that has not been consumed, and try to match all patterns, applying the first one we find. Patterns have the following scheme:

- One or more letters in its prefix. For instance, the prefix of rule #9 is “S” and the prefix of rule #5 is “LL”. The prefix must match the current place in the string for the pattern to match.
- *Optional*: a parenthesized set of letters, such as “(EIY)” in rule #8. If the current place in the word matches any one of the characters between the parentheses, then we continue processing the pattern. For instance, rule #8 could match “sce”, “sci”, or “scy”.
- *Optional*: some number of “–” signs. Typically, when we match a full pattern, all of the letters in the word that we used to match the pattern are “consumed”, and will not be examined again. For instance, if we first match rule #10 for the word “that”, the next character we examine will be the “a” (not the “h”). However, for each – sign that occurs, we consume one less character from the word. When we process the word “llama” with pattern #5, it will match the “LL” but only consume the first “L”: we will then apply the appropriate transformation based on the word “lama”.
- *Optional*: the ‘^’ (caret) character, which signals that this pattern may only match at the beginning of the word. For instance, rule #1 can be applied for the word “art” but not for the word “afar”.
- *Optional*: the ‘\$’ character, which signals that this pattern may only match at the end of the word. For instance, rule #6 can be applied for the word “dumb” but not for the word “wombat”.

The pronunciation for each rule is either some series of pronunciation characters or is the single character ‘_’, meaning “silent”. If the pronunciation for a rule is silent, nothing is appended to the Metaphone pronunciation string when that rule is applied. For instance, rule #4 makes all “I”’s silent.

Example: To transform the word “science” to its Metaphone pronunciation, we take the following steps:

1. Set the Metaphone output word to the empty string.
2. Start at the first ‘s’: the first rule that matches is #8, so we consume “sc” (but not the ‘i’) from the input word and add ‘S’ to the Metaphone output word.
3. We are now at the ‘i’, and the first rule that applies is #4. We consume the ‘i’ and leave the output word unchanged.
4. We are now at the first ‘e’, and the first rule that applies is #3. We consume the ‘e’ and leave the output word unchanged.

5. We are at the 'n': apply rule #7 by consuming the 'n' and adding 'N' to the output word.
6. We are at the second 'c': the first rule that matches is rule #2: consume the 'c' (but not the 'e') and add 'S' to the output word.
7. We are at the second 'e': rule #3 matches, so consume the 'e' and leave the output word unchanged.
8. We have reached the end of the input word. The Metaphone pronunciation word is "SNS".

You should never have a case where no pattern matches. Also, you can ignore any errors in the input, and assume that no word contains any punctuation characters or numbers: only characters a-z (lower or upper case) will be used to make words.

Using Metaphone to find homophones

Once you have implemented the Metaphone algorithm, you can determine the set of words in a dictionary that "sound like" a particular word (i.e., homophones) just by calculating the Metaphone pronunciations of every word in the dictionary. Those with the same metaphone pronunciation as the given word sound like the word, and should be returned as results.

Edit distance: ranking homophones

Once you have determined the set of homophones for the word, you need to determine the order in which those results should be presented. Since the user probably mistyped the word, we want results that are close from a typing perspective. For instance, "spelling" is closer to "speling" than is "sapling", so we would like to present "spelling" as the first alternative if the user types the word "speling". To do this, we will compute the *edit distance* between the typed word and each of the real words that sound like the typed word. The edit distance is a measure of how many steps it takes to transform one string into another, so closer matches will have smaller edit distances. When two homophones have the same edit distance from the word typed, order them alphabetically.

The edit distance between string S_1 and S_2 is the minimum number of symbol deletions and replacements required to transform S_1 and S_2 into a common string.

Example 1

Given:

$S_1 = agttgtagct$ $S_2 = agtgctact$

To transform S_1 and S_2 into a common string we can apply three deletions.

1. delete $S_1[3]$ which is a t
2. delete $S_1[7]$ which is a g
3. delete $S_2[4]$ which is a c

The result is the string $agtgtagct$. The absolute minimum number of deletions/replacements required to transform S_1 and S_2 into a common string is three. See for yourself: Try to transform the strings using only two deletions/replacements. You will see that it is impossible.

Consider the following naive approach to computing the edit distance between two strings:

1. Given two strings, try deleting each symbol of the two strings (one at a time), or replacing each symbol with a different symbol, and see if the two strings are ever equal.
2. If the strings with a single deletion or replacement are never equal, try deleting/replacing every possible combination of two symbols and see if the two strings are ever equal.

- To compute a new cell, three neighbors must already be computed (see Figure 3(b)): (1) the left neighbor, (2) the top neighbor, and (3) the diagonal top-left neighbor.

Recall that $T[i][j]$ is the edit distance between $X[0, \dots, j-1]$ and $Y[0, \dots, i-1]$. If the two symbols that correspond to $T[i][j]$ are equal, then no deletion or replacement is needed. Thus, the new edit distance is equal to the edit distance between $X[0, \dots, j-2]$ and $Y[0, \dots, i-2]$, which was previously computed and stored in $T[i-1][j-1]$.

If the two symbols are not equal, then a deletion or replacement must be performed. To perform a replacement, we take the edit distance from the diagonal top-left neighbor and increase it by one. To delete, we must examine the edit distance stored in the top and left neighbors. If the top neighbor has a smaller edit distance, then the symbol along the left should be deleted. If the left neighbor has a smaller edit distance, then the symbol along the top should be deleted. This decision ensures that we choose the proper deletions to minimize the edit distance.

Computing $T[i][j]$

- Each cell in the table corresponds to two symbols: the symbol along the top row and the symbol along the left column.
- If the two symbols are equal, we copy the diagonal top-left value to the new cell.
- If the two symbols are not equal, we examine the top cell, diagonal top-left cell, and the left cell. We choose the cell with the smallest value and copy that value, add one to it, and write the incremented value into the new cell.

Figure 4 shows the entire computed table. The purpose of the entire table is to arrive at the value in the bottom right corner, which represents the edit distance between the two strings.

		Column index j											
		0	1	2	3	4	5	6	7	8	9	10	
		a g t a c g t c a t											
Row index i	0	0	1	2	3	4	5	6	7	8	9	10	
	1	g	1	1	1	2	3	4	5	6	7	8	9
	2	t	2	2	2	1	2	3	4	5	6	7	8
	3	a	3	2	3	2	1	2	3	4	5	6	7
	4	t	4	3	3	3	2	2	3	3	4	5	6
	5	c	5	4	4	4	3	2	3	4	3	4	5
	6	g	6	5	4	5	4	3	2	3	4	4	5
	7	t	7	6	5	4	5	4	3	2	3	4	4
	8	a	8	7	6	5	4	5	4	3	3	3	4
	9	t	9	8	7	6	5	5	5	4	4	4	3

Figure 4: The computed dynamic programming table. The edit distance between the two strings is 3.

Dynamic programming saves an incredible amount of computation by recording the edit distance between the prefixes of the strings, rather than recompute them as needed. The naive algorithm we first described implicitly recomputes this information over and over again.

The problem with dynamic programming is that the table requires a lot of memory. Given two input strings of length n and m , the size of the table would be $(n+1)(m+1)$. For example, if you were given two input strings that were each one million symbols long, the table would contain 10^{12} cells, which is too large to store in memory.

However, storing the entire table is not necessary. In fact, we can compute the final edit distance by storing only two rows of the table. If we align a string of length n along the top of the table, the amount of

memory required is $2(n+1)$. This is described as a linear memory algorithm because the amount of memory is linearly proportional to the length of the input strings, which is a very nice feature.¹

Grading criteria

- Compilation (30 points): Your program must compile using VC++ .NET or g++ (version 2.95 or higher; check with `g++ --version`). If you comment out most of your code or if you do not attempt to implement the algorithms you will not receive this credit.
- Metaphone algorithm: 20 points.
- Spell checker: 15 points. This includes reading in the dictionary, spell-checking words, and presenting the results.
- Edit distance algorithm: 15 points.
- Design and Documentation (20 points): Your algorithms must be well-designed and well-structured, and your code well-commented. Make sure you implement an efficient solution to the problem. For example, you should not search the entire dictionary to find all homophones. You are encouraged (but not required) to use STL containers and algorithms in your code.

Notes and suggestions

- When you write code to read in `rules.txt`, make absolutely sure your program has read everything properly *before* passing the resulting data structure to the Metaphone algorithm. You don't want to spend time debugging the algorithm when it's just your input reading causing problems.
- Write **and test** the Metaphone algorithm implementation and the edit distance implementation separately. Writing and testing in isolating makes the whole program easier to debug.
- You may want to write the Metaphone algorithm first, then the spell checker (without sorting the homophones), and finally implement the edit distance algorithm to rank the results.
- Start early!

Submission Guidelines

Your submission must include only your source code (`.h` and `.cpp` suffix) and a brief report as a `readme.txt` file. Your submission should NOT include any files with a `.exe`, `.dsp`, `.dsw`, `.ncb`, or `.opt` suffix. Every file should have your name in a comment line at the top. Your `readme.txt` file should have a brief description of your program design, the breakdown of the files, which compiler you used (VC++ .NET will be taken as the default), a summary of what you think works and fails in your program.

Exact details of the web-based submission procedure you must follow will be posted on the project web page. Briefly, you should submit a single zip, tar, or gzip file containing your source code files and `readme.txt` file. You can submit your project multiple times; only the most recent project submission will be graded. The most recent project submission will also be used to compute late days, if any.

¹You are not required to implement this optimization.

Final Warning

A project that does not follow the submission guidelines will receive a 10 point deduction. Proper submission is entirely **your responsibility**. Contact your TA if you have any doubts whatsoever about your submission. Do **NOT** submit your project via email. Be sure to keep copies of your files, and **do not change them after submitting**. Don't even copy them. It is a good idea to keep your submission files in a private (protected) directory on RCS to have a valid timestamp. After grades are posted, you have exactly one week to resolve all problems. After that week is up, all grades are final.