A Method for Enhancing Search Using Transliteration of Mandarin Chinese

Vijay John
March 2006

1 Abstract

Although search engines are increasingly capable of answering queries, they do not yet look for all possible transliterations of the same word. This paper describes a new approach to enhance search engine performance on terms originally from Mandarin Chinese. We propose an algorithm that can expand searches to include relevant transliterations of Mandarin Chinese search terms in addition to the original *Hànyǔ Pīnyīn* (unaccented) search terms. The algorithm improves searches by extracting parts of search terms using a right-to-left search method. This algorithm, *Xiǎozhǐ*, also implements a program specifically designed for Mandarin Chinese search terms written in *Pīnyīn*. The program includes a list of transliteration replacement sets, within which the elements are possible transliterations of the same sound. The first element in each set is the unaccented *Pīnyīn* transliteration. *Xiǎozhǐ*, however, may be extended to other languages using a different list of sets.

2 Introduction

A large proportion of web pages are in English. When dealing with foreign terms, such pages generally contain transliterations of foreign terms. There is little standardization of transliteration, thus it is often difficult to find relevant results when search queries involve transliterated terms originally from non-Romanized languages. This paper describes a new approach that enhances search engine performance on terms originally from Mandarin Chinese.

The purpose of *Xiǎozhǐ* and associated computer programs proposed in this paper is, ultimately, to facilitate searches consisting of terms of Chinese origin using several transliterations of the same word and to produce as many proper results as possible.

Generally, the user of a search engine could transliterate words from any language in several different ways. However, the original search term may not necessarily be the same as the most commonly accepted transliteration of the word in Roman letters.[1]

An English-speaker, for example, might transcribe the word that is usually spelled *pot* as "pawt." In certain cases, Google® will search for the entered word and a few syntactic variations. However, it will not search for transliteration variations. A Google search of "pawt," for example, may produce results that use abbreviations, acronyms, etc. that are spelled with those four letters. Perhaps, it may also include slight variations such as "pawts" and "pawting." Only a few of these results may contain any references to *pot*.

In a language such as Mandarin, this problem is even more complicated. Mandarin's official script is either Simplified or Traditional *Hànzǐ* ("Chinese characters").[2] There are not one but several official and reputed transliteration systems that exist for the language, depending on the particular country. Other well-known entities (e.g. Yale University, Herbert Giles's *A Chinese-English Dictionary*) devise phonetic transliteration systems and thus introduce even more variations. In addition, of course, searchers from different cultural and social backgrounds often misspell words and transcribe words in very different ways.

Furthermore, even accepted transliteration systems for Mandarin Chinese may have significantly different spellings not only of single phonemes (as in English) but also of entire syllables that are between 1-4 Roman characters long. For example, Mandarin syllables that are spelled `zhun` in *Hànyǔ Pīnyīn* proper would be written as `chun` in Wade-Giles (a widely used transliteration system in Taiwan), `jwun` in Yale Transcription, and `juen` in Gwoyeu Romatzyh (formerly the official transliteration system in Taiwan).

Clearly, detailed tables of replacement patterns are needed to search for alternate transliterations to terms entered by a searcher. Replacement algorithms, in combination with other strategies such as eliminating unlikely search terms, can then be used to search effectively for alternate transliterations of search phrases. This paper explains one such algorithm (*Xiǎozhǐ*) that has the potential to improve search engines.

3 Previous Work

Many researchers[3] have written about various other aspects of Chinese transliteration, in particular:

1. the transliteration of English (or non-Chinese) names into Chinese characters,
2. the rendering of these names into *Hànyǔ Pīnyīn* (Mainland China's Romanization system), and
3. the transformation of Chinese names from *Pīnyīn* to the original *Hànzǐ*.

Still others have focused on similar transliteration problems in other languages, e.g. Arabic and Japanese. Their techniques are more similar to the algorithm described below than to *Xiǎozhǐ*. [Stalls and Knight] mention that:

(Arbabi et al., 1994) developed an algorithm at IBM for the automatic forward transliteration of Arabic personal names into the Roman alphabet. Using a hybrid neural network and knowledge-based system approach, this program first inserts the appropriate missing vowels into the Arabic name, then converts the name into a phonetic representation, and maps this representation into one or more possible Roman spellings of the name. The Roman spellings may also vary across languages (*Sharif* in English corresponds with *Chérife* in French).

They then extend this previous algorithm in order to create a program transliterating the Arabic versions of the Western names back into Roman script. Their program deals with an aspect unique to languages using the Hebrew and Arabic scripts: most texts in Arabic

(as well as in Hebrew, Persian, Yiddish, Urdu, Pashtu, and many other languages) substantially neglect to write vowels to represent their spoken equivalents. [Stalls and Knight] also mention a method by Knight and Graehl that probabilistically constructs alternate terms, finally picking optimal search terms through graph algorithms.

Two papers have used similar algorithms in order to solve such problems in Japanese and Chinese, i.e. to use accepted transliterations of names common among those who speak the particular language and convert the Romanizations into *kanji* and *hànzǐ*, respectively. Yet a third paper [Mettler] uses a syllable-by-syllable transliteration method in order to transliterate foreign names from Roman letters into *katakana*, a writing system normally used in order to write foreign words in Japanese (as well as short sounds like *pii* ("beep") and, occasionally, other short particles such as *naa*).

The algorithm introduced in this paper is what we may call a "right-to-left" algorithm. This means that a string is scanned from right to left in order to find transliterated phonemes. Most of the algorithms for transliteration process strings from left to right.

[Kuo] describes how to find transliterated pairs, i.e. terms in languages other than Mandarin and their transliterations into traditional *hànzǐ*, from the Web. The target of the transliteration is Chinese. It mentions various transliteration systems (Wade-Giles, *Tōngyòng Pīnyīn*, and *Hànyǔ Pīnyīn*). Terms are segmented from left to right. [Wan] introduces another left-to-right algorithm focused on transliterating English terms, especially English names, into Chinese. [Gao] creates Chinese terms out of English terms, such as names. It tries to find matching Chinese characters for English phonemes.

4 Initial Program

Initially, we decided that the most convenient method for transliterating Mandarin Chinese terms into Roman script would be to create a program that includes a long list of *transliterated sets*. A transliterated set is a set of strings. For example, `{zhun, chun, jwen, juen}` is a transliterated set. The sets were created according to the initials and finals in the *Pīnyīn* alphabet.

New sets have also been created that include other sounds, of four letters maximum, that are not formally listed in the alphabet but may have unusual or special transliteration concerns. For example: `zhun` appears to be considered a syllable rather than an "initial" or "final." (Examples of "initials" and "finals" in *Pīnyīn* include *c, ch, zhi, uang*). These new sets have been created in an effort to simplify the process of transliteration for the computer program in a relevant manner.

The program was originally designed to operate in the following manner: it was assumed that the original search term and the first element in each set were written in *Pīnyīn* without any tones. The other elements in each set reflected both relatively accepted systems of transliteration and various cultural backgrounds; "iao" is also written as "yau" (Yale Transcription) and "iaor" (to acknowledge the "retroflexed r" in Beijing dialect, i.e. the tendency of Mandarin-speakers from Beijing to add "r" to the end of many words, such as *diànyǐngr* for "movie theater"). All sets may be altered at any time if desired;

thus, the program can be perpetually improved as more investigation reveals more facts about transliteration.

This computer program was written in Java. In a Command window, the searcher could type in a term in *Pīnyīn* and not bother to use tone marks. The program would, in turn, search Google for all results of the original term and of a modified term. However, a major flaw was soon found with this approach: the program would make several unnecessary changes before searching for the second term. The term `ying` was, according to the list, to be transliterated only as `ying` and `yingr`. Instead, the program searched in the following manner:

1. Searched for original term "ying,"
2. Changed "ying" to "yink" to "yenk" to "yenk'" to "yemk'" to "yermk'" to "yarmk'," and
3. Searched for "yarmk'."

As a result, Google ultimately searched only for results containing `ying` and those concerning the Yarmuk River that flows through Turkey and Iraq. The program searched in this manner because of the following reasons:

1. *g* is transliterated as "k" in Wade-Giles;
2. *k* as used in Mainland China is supposed to be transliterated as "k'" in Wade-Giles;
3. *i* may be transliterated as *e*, considering the similarity between the two sounds in many Mandarin words (for example, *sè* (color) and *sì* (four), both pronounced (formally, at least) as [sɪ] with a falling tone);
4. *n* might be confused with the letter *m* due to their strong similarity, proximity on the keyboard, nasal quality, etc.,
5. *e* might be written in Beijing dialect as *er*,
6. *er* might be confused with *ar*, since Mandarin-speakers often pronounce certain words written as *er* so that they seem more likely to be transcribed (i.e. transliterated) in *Pīnyīn* as *ar* (for example, the number "two," which is written *èr* but normally spoken as [a:r] with a falling tone).

In summary, the problem was the tendency to search for only two terms in addition to a significant lack of proper organization dictating the necessary transformations of the original search term. Clearly, the program needed to be restructured so that it would at least search somewhat sensibly instead of merely making as many changes as possible to the search term, starting at the end, transliterating letter by letter right-to-left, and reinitiating the process.

5 Proposed Algorithm

The problem would have been relatively less grave, I found, if the program had used the original *Pīnyīn* search term to search the list for individual sets rather than individual letters. The program was changing each result letter by letter. Furthermore, the

conventional left-to-right scanning technique used in the program resulted in poor matches.

I found that the following algorithm improves the replacement process:

1. Before searching for any terms, see whether the search term is included in the list of sets.
2. If not, subtract one letter from the *end* of the term. For example, if `ding` is not available, search for `din` in the list.
3. Continue step #2 until what remains of the term is found to be the first element in one of the sets. In the case of our last example (`ding`), that would mean that after `din` (which is not listed), look for `di` (also not listed) and, finally, `d` (which is listed).
4. Next, look for the rest of the term that has been discarded in the list. So, after finding `d` in `ding`, look for `ing` (which is listed).
5. If necessary, continue step #4, storing terms that are found and performing step #4 with what remains. Stop when the part that remains is an available *Pīnyīn* transcription.
6. Find all listed transformations of the parts of the term that are found. (For `ding`, one set shows that "d" may be changed to "d, ch, t," while another reveals that "ing" may be changed to "ing, ingr, ino, im, in.")
7. List all possible combinations of these transformations ("ding, dingr...ching, chingr...ting, tingr, tino...").
8. Search for all combinations using Google using a simple concatenation.
9. Eliminate the combinations that do not produce a significant number of results.[4]

This approach is compatible even with most multisyllable words, although the program does not yet include a system indicating the division of syllables. For example, when the word `pengren` (*pēngrèn* in Mandarin Chinese means "cooking" but is composed of two syllables with distinct meanings: *pēng-rèn*) is entered into the system, the results should be determined by looking first for "pengren," then "pengr," etc. until it finds the true components: `p, eng, r, en`. Replacements can then be done by replacing each of these components by another string in its set. The sets in this example are:

```
...
{"p", "ph", "bh", "p'", "bp", "pch"}...
{"r", "j", "zh", "l"}...
{"eng", "engr", "uu", "u"}...
{"en", "in", "enr", "on"}...
```

Here `p` could be replaced with `ph`, `bh`, etc. Similarly, `r` could be replaced with `j`, `zh`, etc. Resultantly, several search patterns can be obtained from the original search term `pengren` (e.g. `phuujin`).

6 Adding Transliteration Knowledge

The method described here depends on knowing the relationships between different transliteration schemes such as Wade-Giles, Yale transcription, etc. This information is stored in simple tables in the current version. The program simply uses tables of replacement patterns within the algorithm described in the last section.

The information about alternate transliterations can be simply given in an array as above. A match on a term inside a particular sub-array, such as `{"p", "ph", "bh", "p'", "bp", "pch"}`, means that alternate transliterations within the same array may be used to replace that particular matched part.

Wade-Giles and Yale transliterations have been included into the program mainly based on a table provided in Janey Chen's *A Practical English-Chinese Pronouncing Dictionary* [Chen]. These tables consist of all syllables (without tones) that exist in Mandarin Chinese. They are organized alphabetically using the Yale syllabary.

The specific information in some parts of the table has been included in separate sets. Thus, all information pertaining to the unaccented *Pīnyīn* syllable *quan* has been included under *set* `quan`, i.e. the set beginning with `quan`. Generally, however, information pertaining to the transliteration of one syllable has been decomposed into elements of the sets created in the program. For example, transliterations for *Pīnyīn* syllable *bo* include *po* (Wade-Giles) and *bwo* (Yale). For this reason, `p` has been entered as an element in *set* `b`, and `wo` has been entered as an element in *set* `o`.

Information concerning other accepted transliteration systems (e.g. Gwoyeu Romatzyh) is in accordance with various online sources. [5] In addition to these systems, we have also included some unofficial transcriptions that could be, or have been, used in transliterating Mandarin Chinese terms. For instance, Mandarin-speakers may pronounce *Pīnyīn ch, sh, zh, chi, shi, zhi* as *c, s, z, ci, si, zi*. Therefore, some may enter a misspelled query, e.g. with *c* instead of *ch*.

Since the Beijing accent of Mandarin often includes the letter *r* at the end of each syllable, the addition of *r* at the end of a set's first element has been reflected where possible (e.g. *er, ianr, iar* have been included in sets `e` and `ia`).

Each Chinese dialect has its own pronounciation of a Mandarin character. Also, many other languages have borrowed vocabulary from Chinese dialects and adjusted the pronounciation to suit the particularities of their own language. Transliterations based on such pronounciations of Mandarin words have been extensively included in the implementation of *Xiǎozhǐ*.

For example, under *set* `shi`, the element "shek" has been included because the Mandarin syllable *shi* could be transliterated as *shek*, based on its Cantonese equivalent. (In particular, the name that is written in *Pīnyīn* as *Jiǎng Jièshí* is commonly transliterated as "Chiang Kai-shek" in English texts. The transliteration "Kai-shek" is based on the Cantonese pronunciation of Mandarin *Jièshí*.) Similarly, "you" has been included in *set*

`iao`. The Mandarin word for "cuisine," which Japanese has borrowed from a Chinese dialect, is spelled in *Pīnyīn* as *liàolǐ*. This word is pronounced in Japanese as [rjo:ri] and often transliterated as *ryouri* in Roman script.

The implementation of *Xiǎozhǐ* is given in Appendix A.

7 Further Research

Naturally, there are still a great number of important problems; this is merely a first small step toward proper transliteration of Chinese in search engines. Evidently, searching for "dino," "dim," "din," "chino," etc. would be unlikely to help the searcher find anything related to *Pīnyīn* "ding," even if they are common results. (The situation is similar for any search term, including the other example `pengren`).

Sometimes, particularly in cases of multisyllabic entries, it is important for transliteration purposes to recognize what constitutes a syllable in the language in question. Consider, for example, the Mandarin word written in *Pīnyīn* as *zhúnián*, which means "year after year." One of the sets in the list begins with the element `zhun`. However, searching for `zhun` and `ian` is inappropriate here, although that is what the program is bound to do in its present state. In reality, the word *zhúnián* is pronounced *zhú-nián*.

Indubitably, there are various other significant flaws to be corrected in the transliterating procedure as it now remains. However, these do not appear to devalue the algorithm by any means. In fact, the algorithm is merely the foundation for many more improvements.

The transliteration knowledge described in Section 6 is only an introduction of the sources and methods used to construct transliteration tables. The construction of more extensive transliteration tables is a separate topic that needs further investigation.

It is possible to generalize the algorithm to other non-English languages. A different set of component replacements would be needed for each language. The right-to-left replacement method can also have other applications such as stemming and matching. We hope to consider these later.

8 Conclusion

This algorithm proposes the use of a program that transliterates individual Mandarin Chinese words in as many different ways as possible. The program first searches for an entire term in a list of sets containing *Pīnyīn* sounds followed by other possible transliterations of the same sounds. If the entire term is not included in the list, the program subtracts one letter at a time from the end of the term and searches the list again. After finding a part of the term that is included in the list, the program repeats the process with the part of the term for which it has not yet found a corresponding set. Finally, the program uses the sets to create all possible combinations, searches, and keeps those combinations that are most common. This could help search engines improve their facilities for those searching for any information regarding a Chinese word, phrase, city name, etc.

9 Notes

[1]Note that, unlike a *standardized* transliteration, "the most commonly accepted transliteration" does not necessarily have to be accepted in all countries. For example, in the case of Mandarin, *Hànyǔ Pīnyīn* is the official transliteration system of the People's Republic of China, which is the most populated country in the world. Therefore, we will consider *Hànyǔ Pīnyīn* to be "the most commonly accepted transliteration." However, Taiwan officially uses *Tōngyòng Pīnyīn*, a system not yet implemented in *Xiǎozhǐ*.

[2]Simplified characters are officially used in the People's Republic of China and in Singapore whereas traditional characters are the official script in the Republic of China (Taiwan). Other languages that use *hànzǐ* to a slightly limited extent, such as Japanese (which calls them *kanji*), Korean (which uses the term *hanja*), and other Chinese languages (Cantonese, Wú, Mín languages, Gan, etc.) sometimes use a combination of both. However, they tend to adopt traditional characters, because in such areas, *hànzǐ* has often been used for hundreds of years. (The simplified characters have only been in existence since the 20th century.)

[3]More specifically: Wu et al., 2005; Wei, 2004; Kuo and Yan, 2004; Meng et al., 2000; Wan and Verspoor.

[4]The program determines how many results are "significant" by comparing the number of results for all combinations. For example, if "ding" yields five billion results and "tino" yields only two results, "tino" has not produced a "significant" number of results. Therefore, "tino" is eliminated.

[5]For example, you could find some of this information by searching for "Gwoyeu Romatzyh" on Google.

10 References

[Chen] Chen, Janey. A Practical English-Chinese Pronouncing Dictionary. Boston: Tuttle, 1991.

[Gao] Gao Wei, Phoneme-Based Statistical Transliteration of Foreign Names for OOV Problem, Master's Thesis, The Chinese University of Hong Kong, June 2004.

[Kuo] Jin-Shea Kuo and Ying-Kuei Yan, "Generating Paired Transliterated-Cognates Using Multiple Pronunciation Characteristics from Web Corpora," PACLIC 18, December 8th-10th, 2004, Waseda University, Tokyo, pp. 275-282.

[Meng] Helen Meng, Berlin Chen, Erika Grams, Sanjeev Khudanpur, Wai-Kit Lo, Gina-Anne Levow, Douglas Oard, Patrick Schone, Karen Tang, Hsin-Min Wang, and Jian Qiang Wang Mandarin-English Information (MEI): Investigating Translingual Speech Retrieval, University of Pennsylvania, October 2000.

[Mettler] Matt Mettler, TRW Japanese Fast Data Finder, TRW Systems Development Division, Redondo Beach, California.

[Stalls and Knight] BG Stalls, K Knight, Translating Names and Technical Terms in Arabic Text, COLING/ACL Workshop on Computational Approaches to Semitic Languages, Montreal, Quebéc, 1998

[Wan] Stephen Wan and Cornelia Maria Verspoor, "Automatic English-Chinese Name Transliteration for Development of Multilingual Resources," Microsoft Research Institute, Macquarie University, Sydney, Australia.

[Wu] Jian-Cheng Wu, Tracy Lin, and Jason S. Chang, "Learning Source-Target Surface Patterns for Web-Based Terminology Translation," Proceedings of the ACL Interactive Poster and Demonstration Sessions, pp. 37-40, Ann Arbor, June 2005.

11 Appendix A

There are three aspects to implementing the *Xiǎozhǐ*, algorithm. The first part is creating a list of transliteration sets. The second part is implementing the algorithm from section 5 and the third part is utilizing an Application Programming Interface (API) from Google.

11.1 The rule sets

The following rule sets were created from a variety of sources as described in section 6. This information is stored in a table of patterns. The algorithm scans strings from right to left, to find parts that match (for now) the first term in each row. Currently the algorithm only tries to replace *Hànyǔ Pīnyīn* terms, but this can be expanded to include other terms. After finding these matching locations, the algorithm replaces each found pattern in the search term with all the alternates available from the list in the row that starts with the pattern.

1. ya, ja, ia, eea, yea, yeah, yar, iar, jar, eear, year
2. yi, i, yir, ir, yat, yat-
3. ia, ya, iu, eea, yea, yeah, yar, iar, jar, eear, year, a
4. z, ts, dz, j, zh
5. ci, tz'u, tsz, tzu, ts'u, ts'uh, tsu, tsuh
6. si, szu, sz, se
7. zi, tzu, dz, ja, ji
8. ju, chu, jyu, zhu, jü, jew, jur, jyur, jür, jewr
9. qu, ch'u, chyu, chu, chue, chü, chee, chi, ch'ue, ch'ü, ch'u, ch'ee, ch'i, qur, chyur, chur, chuer, chür
10. xu, hsü, hsu, syu, xur, syur
11. a, o, u, ar, ah, aa
12. b, p, bp, bh, pp
13. c, ts, ts'
14. d, t, ch
15. e, o, ih, i, uh, u, a, er, ir, ø, ö, oe, ör, oer, ør, yr
16. f, ph, h, ff
17. g, k, j
18. h, h', kh, r, k, x, ch

19. i, ih, r, e, uh, u, a, er, ir, ø, ö, oe, ör, oer, ør, yr, y
20. j, ch, t, ch', t', k,c, dz, cs, qu
21. k, k', c
22. l, n, r
23. m, n, mh
24. n, m, l
25. o, wo, a, or, wo, aw, å
26. p, ph, bh, p', bp, pch
27. q, ts, ch, ch', c, chh, kv, sh
28. r, j, zh, l
29. t, d, t'
30. u, yu, ur, yur, o, oe
31. wu, u, wur, ur, vu
32. ü, u, yu, ür, ur, yur, ue, uer
33. x, hs, sh, s, sy, sj, h, k, g, hsz
34. ong, ung, ongr, ungr, oeng, oung, unas
35. jun, zhun, chun, jwun, jyun, junr, zhunr, jwunr, jyunr
36. zhun, chun, jwun, zhunr, jwunr
37. qun, ch'un, chyun, qunr, chyunr
38. xun, hsun, syun, xunr, syunr
39. yu, yü, yur
40. ün, ünr, uen, uenr
41. yun, yün, van, vân, yunr
42. jie, kai-, kai, kaj
43. hong, hung, ang, hongr
44. lang, ookami, oukami, okami, langr, lar
45. ing, ingr, ino, im, in
46. ang, angr, ou
47. eng, engr, uu, u
48. juan, chuan, jywan, zhuan, juanr, jywanr, zhuanr, juar, zhuar
49. quan, ch'uan, chywan, chuan, chiuan, quanr, chywanr, chiuanr, quar, chiuar
50. xuan, hsuan, sywan, xuanr, sywanr, xuar
51. jue, chueh, jywe, juer, jywer
52. que, ch'ueh, chueh, chywe, quer, chywer
53. xue, hsueh, sywe, xuer, sywer
54. zh, ch, j, d, cs
55. sh, s, si, x, sch
56. ch, c, chh, ch', cch, chch
57. zhi, chih, jr, der, dir, sa
58. chi, chih, chr
59. shi, shih, shr, shy, shek, sek, chek, shin
60. ri, jih, r, ni, zh
61. ian, ien, yan, ianr, iar, en
62. yan, yen, yanr, yar, jen, jün
63. uan, wan, uanr, wanr

64. wan, wanr, van
65. un, wun, uen, un-, unr, wunr, uenr, yunr
66. ai, air, a'i, a-i, ei, eir, aj, ay
67. ei, eir, e, er, ee
68. ui, uei, wei, uir, weir, way, wayr, ware, wair, wear
69. an, anr, anu
70. en, enr, on
71. in, inr, on
72. ie, ieh, ye, ier, yer
73. ye, yeh, yer
74. iu, you, iou, iur, your, iour
75. you, yu, iou, your, iour
76. ao, au, ow, aor, o, aur
77. ua, wa, uar, war
78. wa, war, ua, uar
79. ou, our, ov
80. ue, uer, ak
81. wei, weir, we, way, wayr, ware, wair, wear
82. uo, o, wo, uor, wor
83. wo, wor, ga, a
84. er, erh, el, ar
85. iao, yau, iaor, yaur, yo, you
86. yao, yau, yaor, yaur
87. iong, iung, yung, iongr, yungr
88. yong, yung, yongr, yungr
89. uang, wang, uangr, wangr
90. wang, wangr, ou
91. üan, üanr
92. yuan, ywan, yuanr, ywanr, yuar, ywar
93. iang, ang, yang, iangr, yangr, an
94. yang, yangr
95. yin, in, yinr, inr
96. ying, yingr
97. üe, üeh, ywe, ue, ueh, üer, ywer, uer
98. yue, yueh, ywe, yuer, ywer
99. uai, wai, uair, wire
100. wai, wair, wire

11.2 Algorithm Implementation

The *Xiǎozhǐ* algorithm in section 5 is implemented in Java. It uses a table containing the list of sets of replacements.

The first part of the algorithm is to find the longest initial part of a search string according to the pattern in the set of replacements. This function finds the table row that starts with a string that is part of the search term at a specific location.

```
Function LongestInitial part of a String word from position start in that word
  Let all be the part of the word from start
  Drop letters from the end of all until we find an initial segment in the table
  When we find a table starting with the leftover part of all
    Return the row in the table corresponding to this part.
```

We use a decomposition function to get the terms in the table. In the current implementation we look only for the patterns at the first location of each row. We can instead look for patterns at any location in each row.

```
Function Decompose: to decompose a word
  Let Parts be a collection to hold possible decompositions
  Let len be the length of the string word
  Let cut be the place where we will cut off a decomposed part
  While the cut location is less than len
    Let top be the value from the LongestInitial function
      when applied to the word starting at the cut location
    If top cannot be found, quit this loop
    Let tops be the starting pattern in that row
    Increase cut by the length of tops
    Add top to the list Parts
```

Once we find the rows corresponding to the patterns we want to replace, we can create all possible alternate search terms by replacing these patterns with the alternate patterns in each row of the table. This is done with a simple combination of terms, using the Parts list from the Decompose function. A function getterms to combine the parts is shown in the program code later in this appendix.

11.3 Google API

The *Xiǎozhǐ* algorithm decomposes a search term and creates a list of alternate search terms. To see whether any of these search terms yield results, we use Google to search using the alternate search terms. This is done through an API available from Google. The Google Web API provides a way to access Google's search functions through Java programs. .

The web location http://www.google.com/apis/ contains information about the API. This API requires the use of a key that is specific to each user. Users have to obtain the key from Google. There are some limitations to the API (such as returning only 10 results per call, and allowing only 1000 calls per day) but these are not a serious limitation to research work using this API.

The following references explain the API.

1. Google Web API Reference http://www.google.com/apis/reference.html
2. API FAQ http://www.google.com/apis/api_faq.html
3. Web API discussion group http://groups.google.com/group/google.public.web-apis
4. Java API reference tree file:///c:/googleapi/javadoc/index.html

(Note that the actual location of 4 is the location where you have installed the Web API.)

The Google API is used in implementing a search function for the *Xiǎozhǐ* algorithm. This implementation is as follows:

```
import com.google.soap.search.*;
import java.io.*;
import java.util.*;

public class search {

  static GoogleSearch gsearch;

  // note: the clientKey should be changed to a valid user key
  static String clientKey = "3gFfnZ9QFABJ59F0QmVBnJb12UAQQIHQa";
  static String directive = "search";
  static final int cutoff = 10000;

  public search () {
    gsearch = new GoogleSearch();
    gsearch.setKey(clientKey);
  }

  public Vector best (Vector terms) {
    Vector best = new Vector ();
    for (int i=0; i<terms.size() && i<20; i++) {
      String term = (String)terms.elementAt (i);
      gsearch.setQueryString (term);
      try {
        GoogleSearchResult r = gsearch.doSearch();
        int count = r.getEstimatedTotalResultsCount();
        System.out.println (""+i+":"+term+" "+count);
        if (count > cutoff) best.add (term);
      }
      catch (Exception e) {
        System.out.println ("Error searching for "+term);
      }
    }
    return best;
  }
};
```

11.4 Implementation

The program is implemented as three Java classes – `search` as above, `decomp` implementing the algorithm and patterns and `translit` to call search and decomp. The table of patterns is truncated in the listing below. The classes decomp and translit are shown below.

```
import java.io.*;
import java.util.*;

public class decomp {

  static String patterns [] [] = {
```

```
  {"a", "o", "u", "ar", "ah", "aa"},
  {"b", "p", "bp", "bh", "pp"},
  {"c", "ts", "ts'"},
  {"d", "t", "ch"},
      // etc ...
  {"wai", "wair", "wire"}
};

String Term;
Vector Parts;
public Vector Terms;

public decomp () {
}

public void decompose (String term) {
  Term = term;
  Parts = new Vector ();
  // look for longest initial and later syllables
  int len = term.length ();
  int cut = 0;
  while (cut < len) {
    int top = longestinitial (term, cut);
    if (top == -1) break;
    String tops = patterns [top][0];
    cut = cut + tops.length ();
    Parts.add (new Integer (top));
  }
}

int longestinitial (String word, int start) {
  String all = word.substring (start);
  // drop letters from the end until we find an initial segment
  // in the table
  int len = all.length ();
  for (int j=len; j>0; j--) {
    String part = all.substring (0, j);
    for (int i=0; i<patterns.length; i++) {
      if (part.equals (patterns [i][0]))
        return i;
    }
  }
  return -1;
}


public String show () {
  if (Parts == null) return "";
  StringBuffer sb = new StringBuffer ();
  for (int i=0; i<Parts.size (); i++) {
    Integer Top = (Integer)Parts.elementAt (i);
    int top = Top.intValue ();
    String tops = patterns [top][0];
    sb.append (tops);
    sb.append (" {");
    for (int k=0; k<patterns [top].length; k++) {
      sb.append (patterns [top][k]);
```

```
        sb.append (" ");
      }
      sb.append ("}\r\n");
    }
    String s = new String (sb);
    return s;
  }

  public Vector getterms () {
  if (Parts == null) return null;
  int n = Parts.size ();
    if (n == 0) return null;
    if (n == 1) {
      Terms = new Vector ();
      Terms.add (Term);
      return Terms;
    }

    // first get the elements in the first array
    Vector temp = new Vector ();
    Integer I = (Integer)Parts.elementAt (0);
    int top = I.intValue ();
    for (int j=0; j<patterns[top].length; j++)
      temp.addElement (patterns [top][j]);

    // at each step build a new vector combining all the prior strings
    // with the strings at this level
    for (int i=1; i<n; i++) {
      Vector now = new Vector ();
      I = (Integer)Parts.elementAt (i);
      top = I.intValue ();
      for (int j=0; j<patterns[top].length; j++)
        now.addElement (patterns [top][j]);
      // now combine with temp to form next
      Vector next = new Vector ();
      int an = temp.size ();
      int bn = now.size ();
      for (int a=0; a<an; a++) {
        String as = (String)temp.elementAt (a);
        for (int b=0; b<bn; b++) {
          String bs = (String)now.elementAt (b);
          String result = new String (as+bs);
          next.addElement (result);
        }
      }
      temp = next;
    }
    Terms = temp;
    return Terms;
  }
};
```

The translit program calls decomp and search to show results from substitutions using the algorithm.

```
import java.util.*;
```

```
public class translit {

  public static void main (String args []) {
    if (args.length > 0) {
      decomp d = new decomp ();
      d.decompose (args [0]);
      System.out.println ("Decomposition:");
      System.out.println (d.show ());
      d.getterms ();
      System.out.println ("Search Terms:");
      System.out.println (d.Terms.toString ());
      search gs = new search ();
      Vector best = gs.best (d.Terms);
      System.out.println ("Best search terms:");
      System.out.println (best.toString ());
    }
  }
};
```

The following steps will create a working program

1. Obtain a Google API key and substitute that key in the `search` class.
2. Create Java files `decomp.java`, `search.java` and `translit.java` containing the respective classes.
3. Edit `decomp.java` to include all the replacement patterns as shown at the start of this appendix.
4. Compile the three Java programs.
5. Test the compiled program by running "`java translit zhuangren`"