homophone or near-homophone (e.g., *dessert* for *desert*, or *piece* for *peace*).

Detecting non-word errors is generally done by marking any word that is not found in a dictionary. For example, the misspelling *graffe* above would not occur in a dictionary. Some early research (Peterson, 1986) had suggested that such spelling dictionaries would need to be kept small, because large dictionaries contain very rare words that resemble misspellings of other words. For example the rare words *wont* or *veery* are also common misspelling of *won't* and *very*. In practice, Damerau and Mays (1989) found that while some misspellings were hidden by real words in a larger dictionary, the larger dictionary proved more help than harm by avoiding marking rare words as errors. This is especially true with probabilistic spell-correction algorithms that can use word frequency as a factor. Thus modern spell-checking systems tend to be based on large dictionaries.

The finite-state morphological parsers described throughout this chapter provide a technology for implementing such large dictionaries. By giving a morphological parser for a word, an FST parser is inherently a word recognizer. Indeed, an FST morphological parser can be turned into an even more efficient FSA word recognizer by using the **projection** operation to extract the lower-side language graph. Such FST dictionaries also have the advantage of representing productive morphology like the English *-s* and *-ed* inflections. This is important for dealing with new legitimate combinations of stems and inflection . For example, a new stem can be easily added to the dictionary, and then all the inflected forms are easily recognized. This makes FST dictionaries especially powerful for spell-checking in morphologically rich languages where a single stem can have tens or hundreds of possible surface forms.[5]

FST dictionaries can thus help with non-word error detection. But how about error correction? Algorithms for isolated-word error correction operate by finding words which are the likely source of the errorful form. For example, correcting the spelling error *graffe* requires searching through all possible words like *giraffe*, *graf*, *craft*, *grail*, etc, to pick the most likely source. To choose among these potential sources we need a **distance metric** between the source and the surface error. Intuitively, *giraffe* is a more likely source than *grail* for *graffe*, because *giraffe* is closer in spelling to *graffe* than *grail* is to *graffe*. The most powerful way to capture this similarity intuition requires the use of probability theory and will be discussed in Ch. 4. The algorithm underlying this solution, however, is the non-probabilistic **minimum edit distance** algorithm that we introduce in the next section.

3.11 Minimum Edit Distance

DISTANCE

Deciding which of two words is closer to some third word in spelling is a special case of the general problem of **string distance**. The distance between two strings is a measure of how alike two strings are to each other.

[5] Early spelling error detectors for English, by contrast, simply allowed any word to have any suffix – thus Unix SPELL accepts bizarre prefixed words like *misclam* and *antiundoggingly* and suffixed words based on *the* like *thehood* and *theness*.

MINIMUM EDIT DISTANCE

Many important algorithms for finding string distance rely on some version of the **minimum edit distance** algorithm, named by Wagner and Fischer (1974) but independently discovered by many people; see the History section of Ch. 6 for a discussion of the history of these algorithms. The minimum edit distance between two strings is the minimum number of editing operations (insertion, deletion, substitution) needed to transform one string into another. For example the gap between the words *intention* and *execution* is five operations, shown in Fig. 3.23 as an **alignment** between the two strings. Given two sequences, an **alignment** is a correspondance between substrings of the two sequences. Thus I aligns with the empty string, N with E, T with X, and so on. Beneath the aligned strings is another representation; a series of symbols expressing an **operation list** for converting the top string into the bottom string; d for deletion, s for substitution, i for insertion.

ALIGNMENT

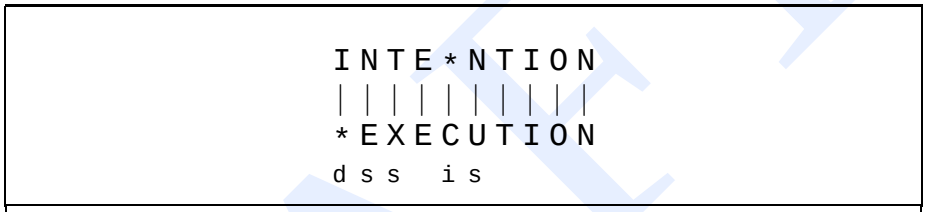

Figure 3.23 Representing the minimum edit distance between two strings as an **alignment**. The final row gives the operation list for converting the top string into the bottom string; d for deletion, s for substitution, i for insertion.

We can also assign a particular cost or weight to each of these operations. The **Levenshtein** distance between two sequences is the simplest weighting factor in which each of the three operations has a cost of 1 (Levenshtein, 1966).[6] Thus the Levenshtein distance between *intention* and *execution* is 5. Levenshtein also proposed an alternate version of his metric in which each insertion or deletion has a cost of one, and substitutions are not allowed (equivalent to allowing substitution, but giving each substitution a cost of 2, since any substitution can be represented by one insertion and one deletion). Using this version, the Levenshtein distance between *intention* and *execution* is 8.

DYNAMIC PROGRAMMING

The minimum edit distance is computed by **dynamic programming**. Dynamic programming is the name for a class of algorithms, first introduced by Bellman (1957), that apply a table-driven method to solve problems by combining solutions to subproblems. This class of algorithms includes the most commonly-used algorithms in speech and language processing; besides minimum edit distance, these include the **Viterbi** and **forward** algorithms (Ch. 6), and the **CYK** and **Earley** algorithm (Ch. 13).

The intuition of a dynamic programming problem is that a large problem can be solved by properly combining the solutions to various subproblems. For example, consider the sequence or "path" of transformed words that comprise the minimum edit distance between the strings *intention* and *execution* shown in Fig. 3.24.

Imagine some string (perhaps it is *exention*) that is in this optimal path (whatever it is). The intuition of dynamic programming is that if *exention* is in the optimal

[6] We assume that the substitution of a letter for itself, e.g. substitution *t* for *t*, has zero cost.

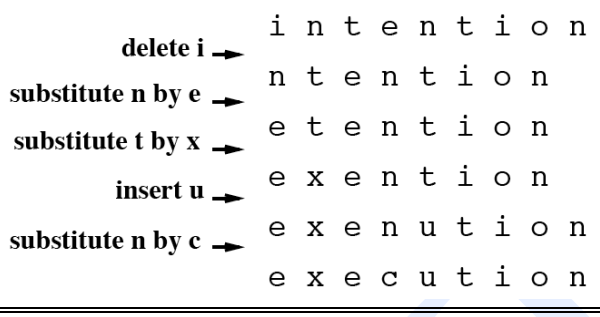

Figure 3.24 Operation list transforming *intention* to *execution* (after Kruskal 1983)

operation-list, then the optimal sequence must also include the optimal path from *intention* to *exention*. Why? If there were a shorter path from *intention* to *exention* then we could use it instead, resulting in a shorter overall path, and the optimal sequence wouldn't be optimal, thus leading to a contradiction.

Dynamic programming algorithms for sequence comparison work by creating a distance matrix with one column for each symbol in the target sequence and one row for each symbol in the source sequence (i.e., target along the bottom, source along the side). For minimum edit distance, this matrix is the *edit-distance* matrix. Each cell *edit-distance*[*i*,*j*] contains the distance between the first i characters of the target and the first j characters of the source. Each cell can be computed as a simple function of the surrounding cells; thus starting from the beginning of the matrix it is possible to fill in every entry. The value in each cell is computed by taking the minimum of the three possible paths through the matrix which arrive there:

$$distance[i,j] = \min \begin{cases} distance[i-1,j] + \text{ins-cost}(target_{i-1}) \\ distance[i-1,j-1] + \text{subst-cost}(source_{j-1}, target_{i-1}) \\ distance[i,j-1] + \text{del-cost}(source_{j-1})) \end{cases} \tag{3.5}$$

The algorithm itself is summarized in Fig. 3.25, while Fig. 3.26 shows the results of applying the algorithm to the distance between *intention* and *execution* assuming the version of Levenshtein distance in which the insertions and deletions each have a cost of 1 (ins-cost($\cdot$) = del-cost($\cdot$) = 1), and substitutions have a cost of 2 (except substitution of identical letters has zero cost).

Knowing the minimum edit distance is useful for algorithms like finding potential spelling error corrections. But the edit distance algorithm is important in another way; with a small change, it can also provide the minimum cost **alignment** between two strings. Aligning two strings is useful throughout speech and language processing. In speech recognition, minimum edit distance alignment is used to compute word error rate in speech recognition (Ch. 9). Alignment plays a role in machine translation, in which sentences in a parallel corpus (a corpus with a text in two languages) need to be matched up to each other.

In order to extend the edit distance algorithm to produce an alignment, we can start by visualizing an alignment as a path through the edit distance matrix. Fig. 3.27 shows

```
function MIN-EDIT-DISTANCE(target, source) returns min-distance

  n ← LENGTH(target)
  m ← LENGTH(source)
  Create a distance matrix distance[n+1,m+1]
  Initialize the zeroth row and column to be the distance from the empty string
    distance[0,0] = 0
    for each column i from 1 to n do
      distance[i,0] ← distance[i-1,0] + ins-cost(target[i])
    for each row j from 1 to m do
      distance[0,j] ← distance[0,j-1] + del-cost(source[j])
  for each column i from 1 to n do
    for each row j from 1 to m do
      distance[i,j] ← MIN( distance[i−1,j] + ins-cost(target_{i−1}),
                           distance[i−1,j−1] + subst-cost(source_{j−1}, target_{i−1}),
                           distance[i,j−1] + del-cost(source_{j−1}))
  return distance[n,m]
```

Figure 3.25 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms. The various costs can either be fixed (e.g. $\forall x, \text{ins-cost}(x) = 1$), or can be specific to the letter (to model the fact that some letters are more likely to be inserted than others). We assume that there is no cost for substituting a letter for itself (i.e. $\text{subst-cost}(x,x) = 0$).

n	*9*	8	9	10	11	12	11	10	9	8
o	*8*	7	8	9	10	11	10	9	8	9
i	*7*	6	7	8	9	10	9	8	9	10
t	*6*	5	6	7	8	9	8	9	10	11
n	*5*	4	5	6	7	8	9	10	11	10
e	*4*	3	4	5	6	7	8	9	10	9
t	*3*	4	5	6	7	8	7	8	9	8
n	*2*	3	4	5	6	7	8	7	8	7
i	*1*	2	3	4	5	6	7	6	7	8
#	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
	#	e	x	e	c	u	t	i	o	n

Figure 3.26 Computation of minimum edit distance between *intention* and *execution* via algorithm of Fig. 3.25, using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions. In italics are the initial values representing the distance from the empty string.

this path with the boldfaced cell. Each boldfaced cell represents an alignment of a pair of letters in the two strings. If two boldfaced cells occur in the same row, there will be an insertion in going from the source to the target; two boldfaced cells in the same column indicates a deletion.

Fig. 3.27 also shows the intuition of how to compute this alignment path. The com-

putation proceeds in two steps. In the first step, we augment the minimum edit distance algorithm to store backpointers in each cell. The backpointer from a cell points to the previous cell (or cells) that were extended from in entering the current cell. We've shown a schematic of these backpointers in Fig. 3.27, after a similar diagram in Gusfield (1997). Some cells have multiple backpointers, because the minimum extension could have come from multiple previous cells. In the second step, we perform a **backtrace**. In a backtrace, we start from the last cell (at the final row and column), and follow the pointers back through the dynamic programming matrix. Each complete path between the final cell and the initial cell is a minimum distance alignment. Exercise 3.12 asks you to modify the minimum edit distance algorithm to store the pointers and compute the backtrace to output an alignment.

BACKTRACE

n	9	↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙←↓ 12	↓ 11	↓ 10	↓ 9	↙ **8**
o	8	↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↓ 10	↓ 9	↙ **8**	← 9
i	7	↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9	↙ **8**	← 9	← 10
t	6	↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	↙ **8**	← 9	← 10	←↓ 11
n	5	↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ **8**	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙↓ 10
e	4	↙ 3	← 4	↙← **5**	← **6**	← 7	←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9
t	3	↙←↓ 4	↙←↓ **5**	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙ 7	←↓ 8	↙←↓ 9	↓ 8
n	2	↙←↓ **3**	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↓ 7	↙←↓ 8	↙ 7
i	**1**	↙←↓ 2	↙←↓ 3	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙ 6	← 7	← 8
#	**0**	1	2	3	4	5	6	7	8	9
	#	e	x	e	c	u	t	i	o	n

Figure 3.27 When entering a value in each cell, we mark which of the 3 neighboring cells we came from with up to three arrows. After the table is full we compute an **alignment** (minimum edit path) via a **backtrace**, starting at the **8** in the upper right corner and following the arrows. The sequence of boldfaced distances represents one possible minimum cost alignment between the two strings.

There are various publicly available packages to compute edit distance, including UNIX `diff`, and the NIST `sclite` program (NIST, 2005); Minimum edit distance can also be augmented in various ways. The Viterbi algorithm, for example, is an extension of minimum edit distance which uses probabilistic definitions of the operations. In this case instead of computing the "minimum edit distance" between two strings, we are interested in the "maximum probability alignment" of one string with another. The Viterbi algorithm is crucial in probabilistic tasks like speech recognition and part-of-speech tagging.

3.12 Human Morphological Processing

In this section we briefly survey psycholinguistic studies on how multi-morphemic words are represented in the minds of speakers of English. For example, consider the word *walk* and its inflected forms *walks*, and *walked*. Are all three in the human lexicon? Or merely *walk* along with *-ed* and *-s*? How about the word *happy* and its derived

forms *happily* and *happiness*? We can imagine two ends of a theoretical spectrum of representations. The **full listing** hypothesis proposes that all words of a language are listed in the mental lexicon without any internal morphological structure. On this view, morphological structure is simply an epiphenomenon, and *walk*, *walks*, *walked*, *happy*, and *happily* are all separately listed in the lexicon. This hypothesis is certainly untenable for morphologically complex languages like Turkish. The **minimum redundancy** hypothesis suggests that only the constituent morphemes are represented in the lexicon, and when processing *walks*, (whether for reading, listening, or talking) we must access both morphemes (*walk* and *-s*) and combine them.

FULL LISTING

MINIMUM REDUNDANCY

Some of the earliest evidence that the human lexicon represents at least some morphological structure comes from **speech errors**, also called **slips of the tongue**. In conversational speech, speakers often mix up the order of the words or sounds:

if you break it it'll drop

In slips of the tongue collected by Fromkin and Ratner (1998) and Garrett (1975), inflectional and derivational affixes can appear separately from their stems. The ability of these affixes to be produced separately from their stem suggests that the mental lexicon contains some representation of morphological structure.

it's not only us who have screw looses (for "screws loose")
words of rule formation (for "rules of word formation")
easy enoughly (for "easily enough")

More recent experimental evidence suggests that neither the full listing nor the minimum redundancy hypotheses may be completely true. Instead, it's possible that some but not all morphological relationships are mentally represented. Stanners et al. (1979), for example, found that some derived forms (*happiness*, *happily*) seem to be stored separately from their stem (*happy*), but that regularly inflected forms (*pouring*) are not distinct in the lexicon from their stems (*pour*). They did this by using a repetition priming experiment. In short, repetition priming takes advantage of the fact that a word is recognized faster if it has been seen before (if it is **primed**). They found that *lifting* primed *lift*, and *burned* primed *burn*, but for example *selective* didn't prime *select*. Marslen-Wilson et al. (1994) found that *spoken* derived words can prime their stems, but only if the meaning of the derived form is closely related to the stem. For example *government* primes *govern*, but *department* does not prime *depart*. Marslen-Wilson et al. (1994) represent a model compatible with their own findings as follows:

PRIMED

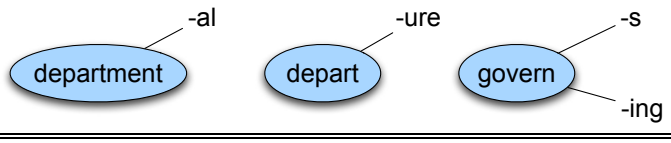

Figure 3.28 Marslen-Wilson et al. (1994) result: Derived words are linked to their stems only if semantically related.

In summary, these early results suggest that (at least) productive morphology like inflection does play an online role in the human lexicon. More recent studies have

MORPHOLOGICAL FAMILY SIZE

shown effects of non-inflectional morphological structure on word reading time as well, such as the **morphological family size**. The morphological family size of a word is the number of other multimorphemic words and compounds in which it appears; the family for *fear*, for example, includes *fearful, fearfully, fearfulness, fearless, fearlessly, fearlessness, fearsome,* and *godfearing* (according to the CELEX database), for a total size of 9. Baayen and colleagues (Baayen et al., 1997; De Jong et al., 2002; Moscoso del Prado Martín et al., 2004) have shown that words with a larger morphological family size are recognized faster. Recent work has further shown that word recognition speed is effected by the total amount of **information** (or **entropy**) contained by the morphological paradigm (Moscoso del Prado Martín et al., 2004); entropy will be introduced in the next chapter.

3.13 SUMMARY

This chapter introduced **morphology**, the arena of language processing dealing with the subparts of words, and the **finite-state transducer**, the computational device that is important for morphology but will also play a role in many other tasks in later chapters. We also introduced **stemming**, **word and sentence tokenization**, and **spelling error detection**.

Here's a summary of the main points we covered about these ideas:

- **Morphological parsing** is the process of finding the constituent **morphemes** in a word (e.g., `cat +N +PL` for *cats*).
- English mainly uses **prefixes** and **suffixes** to express **inflectional** and **derivational** morphology.
- English **inflectional** morphology is relatively simple and includes person and number agreement (*-s*) and tense markings (*-ed* and *-ing*).
- English **derivational** morphology is more complex and includes suffixes like *-ation*, *-ness*, *-able* as well as prefixes like *co-* and *re-*.
- Many constraints on the English **morphotactics** (allowable morpheme sequences) can be represented by finite automata.
- **Finite-state transducers** are an extension of finite-state automata that can generate output symbols.
- Important operations for FSTs include **composition**, **projection**, and **intersection**.
- **Finite-state morphology** and **two-level morphology** are applications of finite-state transducers to morphological representation and parsing.
- **Spelling rules** can be implemented as transducers.
- There are automatic transducer-compilers that can produce a transducer for any simple rewrite rule.
- The lexicon and spelling rules can be combined by **composing** and **intersecting** various transducers.
- The **Porter algorithm** is a simple and efficient way to do **stemming**, stripping off affixes. It is not as accurate as a transducer model that includes a lexicon,

but may be preferable for applications like **information retrieval** in which exact morphological structure is not needed.

- **Word tokenization** can be done by simple regular expressions substitutions or by transducers.
- **Spelling error detection** is normally done by finding words which are not in a dictionary; an FST dictionary can be useful for this.
- The **minimum edit distance** between two strings is the minimum number of operations it takes to edit one into the other. Minimum edit distance can be computed by **dynamic programming**, which also results in an **alignment** of the two strings.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Despite the close mathematical similarity of finite-state transducers to finite-state automata, the two models grew out of somewhat different traditions. Ch. 2 described how the finite automaton grew out of Turing's (1936) model of algorithmic computation, and McCulloch and Pitts finite-state-like models of the neuron. The influence of the Turing machine on the transducer was somewhat more indirect. Huffman (1954) proposed what was essentially a state-transition table to model the behavior of sequential circuits, based on the work of Shannon (1938) on an algebraic model of relay circuits. Based on Turing and Shannon's work, and unaware of Huffman's work, Moore (1956) introduced the term **finite automaton** for a machine with a finite number of states with an alphabet of input symbols and an alphabet of output symbols. Mealy (1955) extended and synthesized the work of Moore and Huffman.

The finite automata in Moore's original paper, and the extension by Mealy differed in an important way. In a Mealy machine, the input/output symbols are associated with the transitions between states. In a Moore machine, the input/output symbols are associated with the state. The two types of transducers are equivalent; any Moore machine can be converted into an equivalent Mealy machine and vice versa. Further early work on finite-state transducers, sequential transducers, and so on, was conducted by Salomaa (1973), Schützenberger (1977).

Early algorithms for morphological parsing used either the **bottom-up** or **top-down** methods that we will discuss when we turn to parsing in Ch. 13. An early bottom-up **affix-stripping** approach as Packard's (1973) parser for ancient Greek which iteratively stripped prefixes and suffixes off the input word, making note of them, and then looked up the remainder in a lexicon. It returned any root that was compatible with the stripped-off affixes. AMPLE (A Morphological Parser for Linguistic Exploration) (Weber and Mann, 1981; Weber et al., 1988; Hankamer and Black, 1991) is another early bottom-up morphological parser. Hankamer's (1986) keCi is a an early top-down *generate-and-test* or *analysis-by-synthesis* morphological parser for Turkish which is guided by a finite-state representation of Turkish morphemes. The program begins with a morpheme that might match the left edge of the word, and applies every possible phonological rule to it, checking each result against the input. If one of the outputs

succeeds, the program then follows the finite-state morphotactics to the next morpheme and tries to continue matching the input.

The idea of modeling spelling rules as finite-state transducers is really based on Johnson's (1972) early idea that phonological rules (to be discussed in Ch. 7) have finite-state properties. Johnson's insight unfortunately did not attract the attention of the community, and was independently discovered by Ronald Kaplan and Martin Kay, first in an unpublished talk (Kaplan and Kay, 1981) and then finally in print (Kaplan and Kay, 1994) (see page **??** for a discussion of multiple independent discoveries). Kaplan and Kay's work was followed up and most fully worked out by Koskenniemi (1983), who described finite-state morphological rules for Finnish. Karttunen (1983) built a program called KIMMO based on Koskenniemi's models. Antworth (1990) gives many details of two-level morphology and its application to English. Besides Koskenniemi's work on Finnish and that of Antworth (1990) on English, two-level or other finite-state models of morphology have been worked out for many languages, such as Turkish (Oflazer, 1993) and Arabic (Beesley, 1996). Barton et al. (1987) bring up some computational complexity problems with two-level models, which are responded to by Koskenniemi and Church (1988). Readers with further interest in finite-state morphology should turn to Beesley and Karttunen (2003). Readers with further interest in computational models of Arabic and Semitic morphology should see Smrž (1998), Kiraz (2001), Habash et al. (2005).

A number of practical implementations of sentence segmentation were available by the 1990s. Summaries of sentence segmentation history and various algorithms can be found in Palmer (2000), Grefenstette (1999), and Mikheev (2003). Word segmentation has been studied especially in Japanese and Chinese. While the max-match algorithm we describe is very commonly used as a baseline, or when a simple but accurate algorithm is required, more recent algorithms rely on stochastic and machine learning algorithms; see for example such algorithms as Sproat et al. (1996), Xue and Shen (2003), and Tseng et al. (2005).

Gusfield (1997) is an excellent book covering everything you could want to know about string distance, minimum edit distance, and related areas.

Students interested in further details of the fundamental mathematics of automata theory should see Hopcroft and Ullman (1979) or Lewis and Papadimitriou (1988). Roche and Schabes (1997) is the definitive mathematical introduction to finite-state transducers for language applications, and together with Mohri (1997) and Mohri (2000) give many useful algorithms such as those for transducer minimization and determinization.

The CELEX dictionary is an extremely useful database for morphological analysis, containing full morphological parses of a large lexicon of English, German, and Dutch (Baayen et al., 1995).

Roark and Sproat (2007) is a general introduction to computational issues in morphology and syntax. Sproat (1993) is an older general introduction to computational morphology.

EXERCISES

3.1 Give examples of each of the noun and verb classes in Fig. 3.6, and find some exceptions to the rules.

3.2 Extend the transducer in Fig. 3.17 to deal with `sh` and `ch`.

3.3 Write a transducer(s) for the K insertion spelling rule in English.

3.4 Write a transducer(s) for the consonant doubling spelling rule in English.

3.5 The Soundex algorithm (Odell and Russell, 1922; Knuth, 1973) is a method commonly used in libraries and older Census records for representing people's names. It has the advantage that versions of the names that are slightly misspelled or otherwise modified (common, for example, in hand-written census records) will still have the same representation as correctly-spelled names. (e.g., Jurafsky, Jarofsky, Jarovsky, and Jarovski all map to J612).

a. Keep the first letter of the name, and drop all occurrences of non-initial a, e, h, i, o, u, w, y

b. Replace the remaining letters with the following numbers:

b, f, p, v → 1
c, g, j, k, q, s, x, z → 2
d, t → 3
l → 4
m, n → 5
r → 6

c. Replace any sequences of identical numbers , only if they derive from two or more letters that were *adjacent* in the original name, with a single number (i.e., 666 → 6).

d. Convert to the form `Letter Digit Digit Digit` by dropping digits past the third (if necessary) or padding with trailing zeros (if necessary).

The exercise: write a FST to implement the Soundex algorithm.

3.6 Implement one of the steps of the Porter Stemmer as a transducer.

3.7 Write the algorithm for parsing a finite-state transducer, using the pseudo-code introduced in Chapter 2. You should do this by modifying the algorithm ND-RECOGNIZE in Fig. **??** in Chapter 2.

3.8 Write a program that takes a word and, using an on-line dictionary, computes possible anagrams of the word, each of which is a legal word.

3.9 In Fig. 3.17, why is there a *z, s, x* arc from q_5 to q_1?

3.10 Computing minimum edit distances by hand, figure out whether *drive* is closer to *brief* or to *divers*, and what the edit distance is. You may use any version of *distance* that you like.

3.11 Now implement a minimum edit distance algorithm and use your hand-computed results to check your code.

3.12 Augment the minimum edit distance algorithm to output an alignment; you will need to store pointers and add a stage to compute the backtrace.

Antworth, E. L. (1990). *PC-KIMMO: A Two-level Processor for Morphological Analysis*. Summer Institute of Linguistics, Dallas, TX.

Baayen, R. H., Piepenbrock, R., and Gulikers, L. (1995). *The CELEX Lexical Database (Release 2) [CD-ROM]*. Linguistic Data Consortium, University of Pennsylvania [Distributor], Philadelphia, PA.

Baayen, R. H., Lieber, R., and Schreuder, R. (1997). The morphological complexity of simplex nouns. *Linguistics*, *35*(5), 861–877.

Barton, Jr., G. E., Berwick, R. C., and Ristad, E. S. (1987). *Computational Complexity and Natural Language*. MIT Press.

Bauer, L. (1983). *English word-formation*. Cambridge University Press.

Beesley, K. R. (1996). Arabic finite-state morphological analysis and generation. In *COLING-96*, Copenhagen, pp. 89–94.

Beesley, K. R. and Karttunen, L. (2003). *Finite-State Morphology*. CSLI Publications, Stanford University.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Chomsky, N. and Halle, M. (1968). *The Sound Pattern of English*. Harper and Row.

Damerau, F. J. and Mays, E. (1989). An examination of undetected typing errors. *Information Processing and Management*, *25*(6), 659–664.

De Jong, N. H., Feldman, L. B., Schreuder, R., Pastizzo, M., and Baayen, R. H. (2002). The processing and representation of Dutch and English compounds: Peripheral morphological, and central orthographic effects. *Brain and Language*, *81*, 555–567.

Fromkin, V. and Ratner, N. B. (1998). Speech production. In Gleason, J. B. and Ratner, N. B. (Eds.), *Psycholinguistics*. Harcourt Brace, Fort Worth, TX.

Garrett, M. F. (1975). The analysis of sentence production. In Bower, G. H. (Ed.), *The Psychology of Learning and Motivation*, Vol. 9. Academic.

Grefenstette, G. (1999). Tokenization. In van Halteren, H. (Ed.), *Syntactic Wordclass Tagging*. Kluwer.

Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press.

Habash, N., Rambow, O., and Kiraz, G. A. (2005). Morphological analysis and generation for arabic dialects. In *ACL Workshop on Computational Approaches to Semitic Languages*, pp. 17–24.

Hankamer, J. (1986). Finite state morphology and left to right phonology. In *Proceedings of the Fifth West Coast Conference on Formal Linguistics*, pp. 29–34.

Hankamer, J. and Black, H. A. (1991). Current approaches to computational morphology. Unpublished manuscript.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Huffman, D. A. (1954). The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, *3*, 161–191. Continued in Volume 4.

Johnson, C. D. (1972). *Formal Aspects of Phonological Description*. Mouton, The Hague. Monographs on Linguistic Analysis No. 3.

Kaplan, R. M. and Kay, M. (1981). Phonological rules and finite-state transducers. Paper presented at the Annual meeting of the Linguistics Society of America. New York.

Kaplan, R. M. and Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics*, *20*(3), 331–378.

Karttunen, L., Chanod, J., Grefenstette, G., and Schiller, A. (1996). Regular expressions for language engineering. *Natural Language Engineering*, *2*(4), 305–238.

Karttunen, L. (1983). KIMMO: A general morphological processor. In *Texas Linguistics Forum 22*, pp. 165–186.

Kiraz, G. A. (2001). *Computational Nonlinear Morphology with Emphasis on Semitic Languages*. Cambridge University Press.

Knuth, D. E. (1973). *Sorting and Searching: The Art of Computer Programming Volume 3*. Addison-Wesley, Reading, MA.

Koskenniemi, K. (1983). Two-level morphology: A general computational model of word-form recognition and production. Tech. rep. Publication No. 11, Department of General Linguistics, University of Helsinki.

Koskenniemi, K. and Church, K. W. (1988). Complexity, two-level morphology, and Finnish. In *COLING-88*, Budapest, pp. 335–339.

Krovetz, R. (1993). Viewing morphology as an inference process. In *SIGIR-93*, pp. 191–202. ACM.

Kruskal, J. B. (1983). An overview of sequence comparison. In Sankoff, D. and Kruskal, J. B. (Eds.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pp. 1–44. Addison-Wesley, Reading, MA.

Kukich, K. (1992). Techniques for automatically correcting words in text. *ACM Computing Surveys*, *24*(4), 377–439.

Lerner, A. J. (1978). *The Street Where I Live*. Da Capo Press, New York.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, *10*(8), 707–710. Original in *Doklady Akademii Nauk SSSR* 163(4): 845–848 (1965).

Lewis, H. and Papadimitriou, C. (1988). *Elements of the Theory of Computation*. Prentice-Hall. Second edition.

Marslen-Wilson, W., Tyler, L. K., Waksler, R., and Older, L. (1994). Morphology and meaning in the English mental lexicon. *Psychological Review*, *101*(1), 3–33.

McCawley, J. D. (1978). Where you can shove infixes. In Bell, A. and Hooper, J. B. (Eds.), *Syllables and Segments*, pp. 213–221. North-Holland, Amsterdam.

Mealy, G. H. (1955). A method for synthesizing sequential circuits. *Bell System Technical Journal*, *34*(5), 1045–1079.

Mikheev, A. (2003). Text segmentation. In Mitkov, R. (Ed.), *Oxford Handbook of Computational Linguistics*. Oxford University Press, Oxford.

Mohri, M. (1996). On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, *2*(1), 61–80.

Mohri, M. (1997). Finite-state transducers in language and speech processing. *Computational Linguistics*, *23*(2), 269–312.

Mohri, M. (2000). Minimization algorithms for sequential transducers. *Theoretical Computer Science*, *234*, 177–201.

Moore, E. F. (1956). Gedanken-experiments on sequential machines. In Shannon, C. and McCarthy, J. (Eds.), *Automata Studies*, pp. 129–153. Princeton University Press, Princeton, NJ.

Moscoso del Prado Martín, F., Bertram, R., Häikiö, T., Schreuder, R., and Baayen, R. H. (2004). Morphological family size in a morphologically rich language: The case of finnish compared to dutch and hebrew. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *30*, 1271–1278.

NIST (2005). Speech recognition scoring toolkit (sctk) version 2.1. Available at http://www.nist.gov/speech/tools/.

Odell, M. K. and Russell, R. C. (1918/1922). U.S. Patents 1261167 (1918), 1435663 (1922)†. Cited in Knuth (1973).

Oflazer, K. (1993). Two-level description of Turkish morphology. In *Proceedings, Sixth Conference of the European Chapter of the ACL*.

Packard, D. W. (1973). Computer-assisted morphological analysis of ancient Greek. In Zampolli, A. and Calzolari, N. (Eds.), *Computational and Mathematical Linguistics: Proceedings of the International Conference on Computational Linguistics*, Pisa, pp. 343–355. Leo S. Olschki.

Palmer, D. D. (2000). Tokenisation and sentence segmentation. In Dale, R., Somers, H. L., and Moisl, H. (Eds.), *Handbook of Natural Language Processing*. Marcel Dekker.

Peterson, J. L. (1986). A note on undetected typing errors. *Communications of the ACM*, *29*(7), 633–637.

Porter, M. F. (1980). An algorithm for suffix stripping. *Program*, *14*(3), 130–127.

Quirk, R., Greenbaum, S., Leech, G., and Svartvik, J. (1985). *A Comprehensive Grammar of the English Language*. Longman, London.

Roark, B. and Sproat, R. (2007). *Computational Approaches to Morphology and Syntax*. Oxford University Press.

Roche, E. and Schabes, Y. (1997). Introduction. In Roche, E. and Schabes, Y. (Eds.), *Finite-State Language Processing*, pp. 1–65. MIT Press.

Salomaa, A. (1973). *Formal Languages*. Academic.

Schützenberger, M. P. (1977). Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, *4*, 47–57.

Seuss, D. (1960). *One Fish Two Fish Red Fish Blue Fish*. Random House, New York.

Shannon, C. E. (1938). A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, *57*, 713–723.

Smrž, O. (1998). *Functional Arabic Morphology*. Ph.D. thesis, Charles University in Prague.

Sproat, R. (1993). *Morphology and Computation*. MIT Press.

Sproat, R., Shih, C., Gale, W. A., and Chang, N. (1996). A stochastic finite-state word-segmentation algorithm for Chinese. *Computational Linguistics*, *22*(3), 377–404.

Stanners, R. F., Neiser, J., Hernon, W. P., and Hall, R. (1979). Memory representation for morphologically related words. *Journal of Verbal Learning and Verbal Behavior*, *18*, 399–412.

Tseng, H., Chang, P., Andrew, G., Jurafsky, D., and Manning, C. D. (2005). Conditional random field word segmenter. In *Proceedings of the Fourth SIGHAN Workshop on Chinese Language Processing*.

Veblen, T. (1899). *Theory of the Leisure Class*. Macmillan Company, New York.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the Association for Computing Machinery*, *21*, 168–173.

Weber, D. J., Black, H. A., and McConnel, S. R. (1988). AMPLE: A tool for exploring morphology. Tech. rep. Occasional Publications in Academic Computing No. 12, Summer Institute of Linguistics, Dallas.

Weber, D. J. and Mann, W. C. (1981). Prospects for computer-assisted dialect adaptation. *American Journal of Computational Linguistics*, *7*, 165–177. Abridged from Summer Institute of Linguistics *Notes on Linguistics* Special Publication 1, 1979.

Xue, N. and Shen, L. (2003). Chinese word segmentation as lmr tagging. In *Proceedings of the 2nd SIGHAN Workshop on Chinese Language Processing*, Sapporo, Japan.