BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Rule-Based Password Generation

Karel Jiránek*

Abstract
This paper describes the password generation based on rules. Generator described in this paper uses knowledge from training phase with a password set to reduce time to find a correct password. The paper addresses the design of the password generator for Fitcrack tool. Fitcrack is a password recovery solution which can recover passwords from encrypted files. The main aim of this work is to extend Fitcrack tool with another password generator. When the password being cracked the generator produce password, which is hashed and compared with hash from the encrypted file.

Keywords: Password generation, probability grammar, Fitcrack, cryptography, OpenCL

Supplementary Material: *N/A*

*xjiran00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The security in information technologies is an aplenty discussed topic. The goal of the efforts is to keep our data safe. One way to achieve that is to use strong passwords for our accounts and data. On the other side, complicated passwords and modern cryptography methods mean inconvenience for security law-enforcement agencies during the investigation process. To restore a lost password, we can use several different approaches.

The main problem of the currently used solution and password generators is time required to find – to generate the correct password. It is caused by the absence of any rules during the creation process. For instance, brute force generator generates passwords in alphabetical order changing a letter or set of letters with next from alphabet. From mentioned above the prime aspect of the measurement and evaluation of the result is time needed to find the correct password. We can also compare password per second ratio. It is obvious that simple methods without any rules or ordering will perform better.

There are many solutions allowing users to restore forgotten password. One of them is Fitcrack tool developed at Faculty of Information Technology. Beyond Fitcrack there are other tools for recovering passwords. For example Hashcat – open source tool with big support all around the world.

Our solution gives the goal to reduce time to find the correct password. Also, we want to extend Fitcrack tool with another password generator. To achieve our goal probabilistic grammar will be used to reduce the time and searched pool of passwords.

2. Related work

Beside probabilistic approach, there are many other methods to recover lost password. Easiest and straight forward way is brute force attack. This method is very easy to implement and performs well in password per second ratio. However as the complexity of character set and length of password grows so as the number of passwords. With secure password generated from the character set containing digit and special characters brute force method becomes ineffective – the time

needed to crack a single password can easily exceed dozen of years [1].

Key patterns bring completely different view at password recovery. Generation of a password is based on the easily memorable key push sequence rather than on probability of characters. A key pattern is an order of pushed buttons at keyboard situated into a shape. People use geometrical shapes to remember the password. Among most favorite one belong vertical or horizontal line such as *qwerty* and *asdfg*. Key pattern method requires sophisticated training rules to generate key pattern rules. Also, the additional training set is needed. Results of experiments carried out at Florida State University shows the key pattern method does not bring significant improvement to the password cracking process [2].

As another method we can use Markov string to reduce the time to find the correct password. The Markov chain (string) describes a stochastic process where the probability of next state (next generated letter in the password) depends only on the current state. Nevertheless Markov chain method is more self-sufficient then key pattern method or rule-based password generation – no training password set is needed – method is satisfied with plain language dictionary [3, 4].

The design of our generator is based on rule-based password generation described in Password Cracking Using Probabilistic Context-Free Grammars [5]. The method fragments password in training set by type and then in time of generation reassemble the password but in the probabilistic order. More about grammar in the subsection 3.1. This work varies in the way of storing fragments of the password and in the way of generating the password.

The main difference is that the probability of the fragment is computed during the training phase but when the fragments are sorted the value is omitted (not saved to the file). In the related work mentioned above fragments are stored with this value. The biggest advantage of our design is that the generator does not have to recompute the total probability of a currently generated password because fragments are sorted in the time of training.

Not storing the probability value of fragment can cause wrong generation order of passwords. In other words, password with lower probability might be generated before password with the higher one. The generator has no chance to compute the total probability of password without explicitly assigned probability to each fragment. The minor disorder is overwhelmed with massive generation parallelism on GPU.

3. Design of generator

3.1 Used grammar

During the generator process, we will use the probabilistic context-free grammars.

Grammar is ordered quaternion in order G = (N, T, P, S), where N is a finite set of non-terminal symbols, T is a finite set of terminal symbols, P is a finite set of rewrite rules and finally S start symbol from N [6].

In our case, N includes substitute characters for all characters in the password. Set of terminal symbols T contains passwords issued to check for correctness. The rewrite rules P define how to replace a symbol in the password with the substitute character. The initial symbol depends on the first character in the password. Grammar is called probabilistic when each rewrite rule has assigned a certain probability [7]. In our case, it means we have to assemble a set of rewrite rules according to their total number of appearance in the training set. From now further we will call the rewrite rules pre-terminal strings.

The grammar of our generator is based on Matthew Weir's work [5]. First of all, we must define the set of non-terminals. All alphabet characters in the password (lower case and upper case) will be substituted with L, digits with D and special characters with S. Table 1 sums up our grammar.

Type	Pre-term. symbol	Symbols
Letter	L	abcdefghijklmnopqrstuvwxyz
Digit	D	1234567890
Special character	S	@$%&(){}+-.

Table 1. Table shows type of symbol, char to substitute symbol (pre-terminal character) with and symbol to substitute.

Any uninterrupted sequences of the symbols of the same type are shortened to one substitute letter. The length (number) of the shortened sequence is appended to the proper substitute letter. For instance, the password *myAngles* is rewritten to *LLLLLLLL* and shortened to *L8* – eight letter template (pre-terminal string).

3.2 Training phase

In the training phase, the generator (more precisely the training script) trains on a set of training passwords. The pre-terminal strings are generated in this phase. In one iteration the exactly one password is rewritten to the pre-terminal string. Basically the rewriting is the replacing every character from a password with the substitute symbol – the table 1 presents the substitute symbols (pre-terminal symbols) and types of character to be replaced with substitute symbol. Beside pre-terminal string, it is necessary to store the fragments

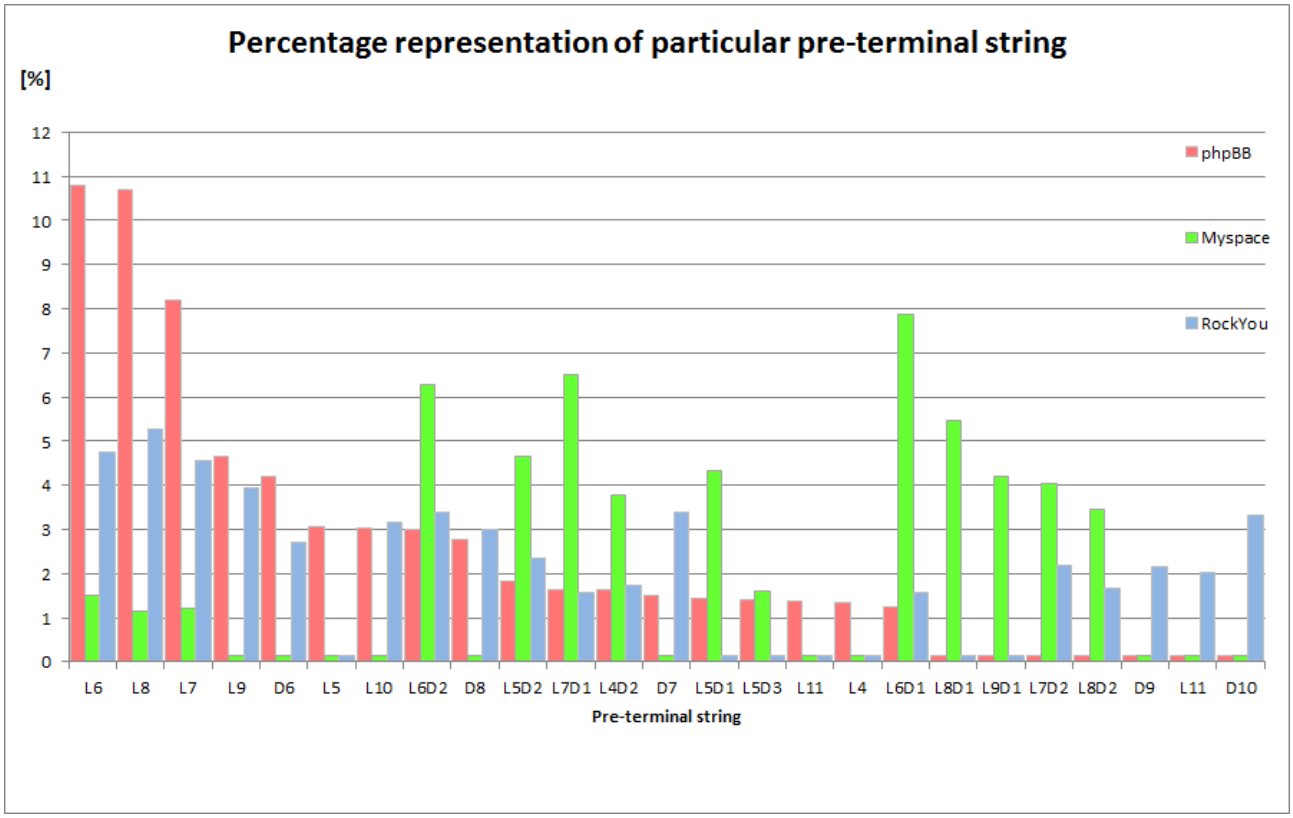

Figure 1. Percentage representation of particular pre-terminal string in sets phpBB, Myspace and RockYou.

from the password. The fragment is a continuous sequence containing the pre-terminal symbols of the same type. For example, the password *p@sswod11* is segmented into four fragments: *p*, *@*, *ssword* and *11*. Each fragment is saved and moved higher in the fragment file (fragment probability is increased) – the fragment on the top of the file has highest probability.

For training purposes, leaked password sets accessible on the Internet[1] were used. Our training script was tested on Myspace set (≈ 37 K passwords), phpBB set (≈ 185 K passwords) and on RockYou set (≈ 15 M passwords).

In the figure 1 we can see the percentage representation of the particular pre-terminal string across the sets. For clarity, only the structures which gained 2% probability are present in the graph.

3.3 Generator core

The first step of the generation is taking the first template (pre-terminal string) from the prepared dictionary. The dictionary with templates was created in the training phase as a result of rewriting passwords. Next step is to insert fragments of proper type and length into the template.

For the generation process, it is essential to define auxiliary index (pivot). The pivot points to the pre-terminal part which is currently changed with a fragment from the dictionary. Other template sections are static at that moment. Initial point for pivot for each template is first part e.g. in the pre-terminal string *L5S2D1*, the first part is *L5*. The pivot is moved to the next part in two cases. First, if fragment dictionary is exhausted – there are no other fragment in left or if there is still at least one part which has not been replaced yet. The detailed move of pivot and generation of the password can be seen in the algorithm 1.

Algorithm 1 Password generation algorithm

```
1:  for each pre-term from preTerm.dic do
2:      pivot ← 1
3:      while pivot != 0 do
4:          for each fragment indexed by pivot do
5:              pre-term[pivot] ← fragment
6:              if all pre-terms replaced then
7:                  send password to crackers
8:              else
9:                  pivot ← pivot + 1
10:             end if
11:         end for
12:         pivot ← pivot − 1
13:         pre-term[pivot] ← pre-term part
14:     end while
15: end for
```

[1] https://wiki.skullsecurity.org/index.php?title=Passwords.

4. Implementation

Both Fitcrack and the probabilistic generator is implemented in C/C++ programming language. OpenCL ISO C language was used to implement kernels for GPU. The script which generates mandatory dictionaries for cracking is written as a script in Python.

4.1 Parsing script

The script which is used in training phase was written in Python 3 and takes single argument. The argument must be file name or path to the file with a training set of passwords. Script presupposes that passwords are separated with at least one symbol of newline.

After creation and verification of file, the script reads input line by line in main loop. Each line is read symbol by symbol. When a symbol of a different type is encountered, length (number of symbols read so far) is appended to substitute symbol. This pair – substitute letter + length is added to previously substituted part of template. E.g. from password *password11!* we get pre-terminal string *L8D2S1* – eight letters followed with two digits and one special character. The fragment from substituted part of the string is stored separately. When the same fragment is encountered, its probability is increased.

When the all password from file are parsed, the created dictionaries (sets of pre-terminal string and fragments) are sorted by fragment/template probability. The number of occurrences (probability) is not needed after sorting thus after sorting is finished this number is omitted. Finally, these dictionaries are saved to the files.

4.2 Generator

The CPU generator is available in the stand-alone version for generating password dictionaries but also as the integrated feature of Fitcrack. The core and logic of these generators is same. Both versions are implemented in C/C++ to reach desired performance. Before generation, it is necessary to load dictionaries into RAM. Schema of the generation is shown in the picture 2. A template is exhausted when all possible combinations of fragments were inserted into the template.

5. Experiments

5.1 Dictionaries

In this section, experiment with the stand-alone versions of the password generator are described. We decided not to experiment with integrated generator in Fitcrack because the most time consuming part of the cracking is consumed by verification of the password. It is hard to measure the performance of the generator and performance of cracker separately.

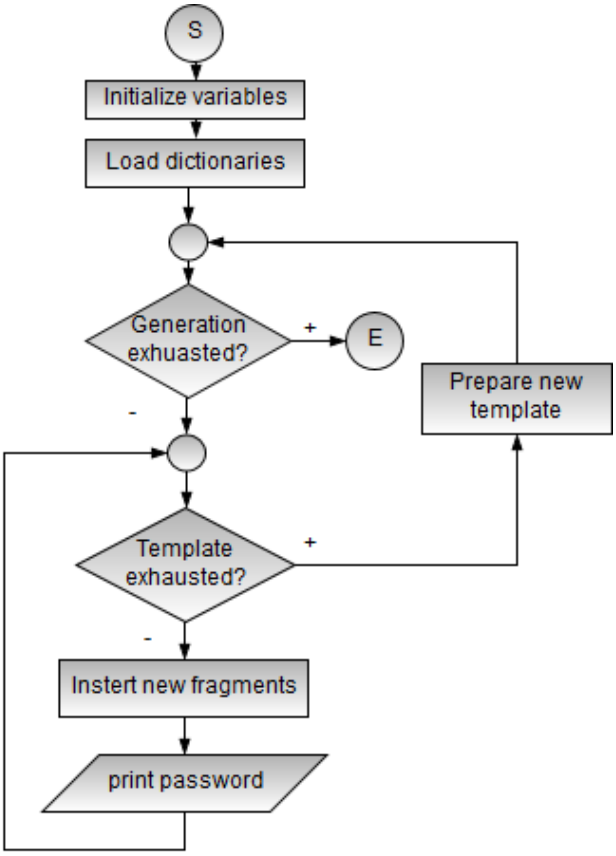

Figure 2. Core logic of CPU generator. Movement of pivot is omitted for simplicity.

Dictionaries	Number of passwords	Set type
Myspace	37 000	Leaked
phpBB	185 000	Leaked
RockYou	15 000 000	Leaked
Pwgen	60 000	Generated
Memorable	60 000	Generated

Table 2. Table showing base password set, number of password in set and how set was obtained.

For experimental purposes, passwords sets displayed in the table 2 were used. First three can be obtained can be obtained from skullsecurity.org[2]. Also two artificial sets were generated for experimental purposes. Passwords in the artificial sets are between 3 and 11 letters long. *Pwgen* set was created with *pwgen*, GPL'ed password generator viable through Ubuntu packages. Passwords in this set are pseudo-random. Last used set *Memorable* was created with password generator accessible on github.com[3]. Passwords in this set should be easily remembered by humans, so we called it *memorable*.

[2] https://wiki.skullsecurity.org/index.php?title=Passwords
[3] https://github.com/bermi/password-generator

We also have to define dictionaries used as input of cracking. In the experiments described further, we will use pre-generated subsets of actual dictionaries because it is not possible to store whole dictionaries. Dictionaries used in experiments were generated from outputs (also dictionaries) created in training phase. For experimental purposes maximal length of generated dictionary was set to 15 million passwords.

5.2 Experiment 1

In this experiment, 1000 random passwords are chosen from leaked password sets (sets *memorable* and *pwgen* are artificially generated). Then these passwords are used an input of the experiment. During the experiment, we want to gain the information whether passwords chosen from leaked sets are present in the generated dictionary. Values displayed in result table are calculated as the total number of the passwords (1000) divided by the number of the number of passwords hit – the number of found passwords. In other words, result value represents how many randomly chosen passwords from leaked dictionary (in percents) was found in generated dictionary.

The result of the experiment can be seen in the table 3. In our experiment, pseudo dictionary named *Brute force* was involved. We consider this set as a dictionary, but we also presumed that brute force would find password sooner or later so we can say that 100 % random passwords will be found. We can notice that some number are followed by the letter *hits*, which represents the number of hits if percentage value of hits is lower than one.

5.3 Experiment 2

The second experiment is very similar to the first one. However in this experiment, time to crack a password is measured. The passwords are taken one by one from the generated dictionary (15 000 000 passwords). Taking a password from a generated dictionary simulates generating process. The advantage of this approach is that with pre-generated dictionaries we do not have to wait until a password is generated. For each password from the generated dictionary, experimental tool check if the password is present in 1000 randomly selected passwords (*1000myspace*, *1000phpBB*, *1000RockYou* etc.). For each match, we noted down the position of the password in the generated dictionary, i.e. its sequence number. After processing the entire 15-milion dictionary, we calculated the average number of iterations required to find a password (the average password sequence number). The lower the better. For computing the cracking time, we needed a reference password per second ratio. We chose speed 22,938,300,000 password per second, which is an average cracking speed while attacking PDF 1.7 (Adobe Acrobat 9)[4]. The average time to crack the password was calculated as a multiplication between the average iteration number to get the correct password (the number of passwords generated before the correct one) and time to crack (verify + generate) one password (1/speed per second).

The situation is complicated with pseudo brute force dictionary. The number of the iteration must be estimated in a different manner, because brute-force generator is very noneffective. It would be unbearably time consuming to wait until the password is found. For the estimation we used special estimation program, algorithm is not discussed in this paper. It is also necessary to mention that every password was trimmed before submitting to the estimation script – all uppercase were changed to lower case and all special characters were released from the password. The trimming was carried out to satisfy brute-force character base (*abcdefghijklmnopqrstuvwxyz0123456789*) – the characters which are not included in the base, but present in submitted password, would cause an error during estimation.

The results are presented in the figure 4. The excellent result are marked with green (time $<=$ day) and bad with red color (time $>$ day). Some cells are hatched because the number of the found passwords during the experiment was lower than 1 % (10 passwords).

5.4 Experiment 3

In the experiment number 3 grammar base attack (rule base attack) and dictionary attack are compared. For this experiment we chose 200 random password from the *faithwriters* leaked password set. The full *faithwriters* set can be obtained also from the *skullsecurity.org*. The experiment has two part. In the first part, we tried to crack 200 password with the leaked password sets – *Myspace* and *phpBB*. In other words we try to find a password from the 200 selected passwords in the leaked dictionary. In the second part we repeated the procedure but instead of leaked password sets, the dictionaries containing 4 billions passwords generated by our generator were used. The result can be seen in figure 5.

5.5 Conclusion of experiments

From the tables 3 and 4 we can say dictionaries created from leaked sets (real passwords) perform very well against the same type of sets (passwords created

[4]Cracking on pc configuration:
https://sagitta.pw/hardware/gpu-compute-nodes/brutalis/

		Random passwords from leaked sets				
		Myspace1000	phpBB1000	RockYou1000	Memorable1000	Pwgen1000
Generated dictionary	Brute force	100 %	100 %	100 %	100 %	100 %
	Myspace	63 %	70 %	40 %	0 hits	0 hits
	RockYou	17.8 %	31.8 %	46.4 %	9 hits	0 hits
	phpBB	10 %	54 %	12.5 %	0 hits	0 hits
	Memorable	1 hits	1 hits	0 hits	70 %	0 hits
	Pwgen	0 hits	0 hits	0 hits	0 hits	8 %

Figure 3. Output from experiment focused on the percentage of hit 1000 random passwords in generated dictionary. The excellent result are marked with green, average with blue and bad with red color.

			Random passwords from leaked sets				
			Myspace1000	phpBB1000	RockYou1000	Memorable1000	Pwgen1000
Generated dictionary	Brute force	average crack time	35744741 years	1366711 years	96721065 years	20 days	21 days
	Myspace		13 ms	240 ms	19 ms	3 ms	∞
	RockYou		24 ms	14 ms	24 ms	460 µs	∞
	phpBB		36 ms	15 ms	40 ms	830 ns	∞
	Memorable		1.9 µs	2.1 µs	∞	1.6 µs	∞
	Pwgen		630 µs	∞	∞	∞	6.6 ms

Figure 4. Output from experiment focused on time (iteration sequence number) reduction. The excellent result are marked with green (time $<=$ day) and bad with red color (time $>$ day). Some cells are hatched because the number of the found passwords during the experiment was lower than 1 % (10 passwords).

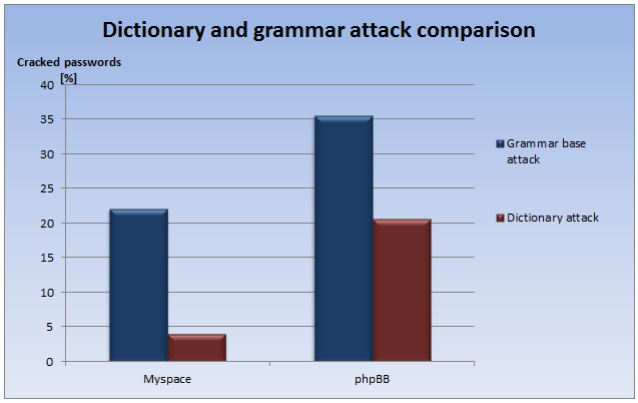

Figure 5. Dictionary and grammar attack comparison.

by humans vs. passwords created by humans, artificially created with one method vs. artificially created with the same method). We can see that dictionaries based on human passwords are almost useless against machine generated passwords. The excellent result are marked with green and bad with red color. From figure 5 we can say that our generator managed to find approximately 18-35 % more passwords than the dictionary attack.

6. Conclusion

In this paper, we discussed aspects of rule-based password generation. Other approaches such as *key patterns* and *Markov chain* are mentioned in 2. In the section 5 we experimented with various sets and dictionaries.

Training script and the password generator gives the solid performance (hit ratio) in the generation process. The performance of the password generator based on the rules rise and falls on the used dictionary. Several performance outputs from the experiments are showed in the section 5. Most of all our rule-based generator incredibly reduce time – the number of iteration needed – to find proper password when compared with brute force attack. The average saving is approximately $3.0 * 10^{24}$ iterations (reduction from a few thousand years to a seconds) compared with brute-force attack. It is necessary to say that the hit ratio of our generator – number of found passwords – was lower than brute-force's.

We would like to point out that the generator and training script are only means for cracking – a additional training set/s must be used. Generator is useless without any previously used training set/s.

The amount of generated passwords is currently not limited and depends purely on the grammar used. For more computationally-complex cracking, the number of generated passwords may be too high to be processed in a meaningful time. Thus, in the future work, we want to let the user define a limit called the threshold. The threshold defines the minimum probability of a rewrite rule to be used. Rewrite rules with probability lower than the threshold are ignored which reduces the number of generated passwords.

Acknowledgements

I would like to thank my supervisor Ing. Radek Hranický for help, willingness and advice during work at this paper.

References

[1] Radek Hranický, Petr Matoušek, Ondřej Ryšavý, and Vladimír Veselý. Experimental evaluation of password recovery in encrypted documents. In *Proceedings of ICISSP 2016*, pages 299–306. SciTePress - Science and Technology Publications, 2016.

[2] S. Houshmand, S. Aggarwal, and R. Flood. Next gen pcfg password cracking. *IEEE Transactions on Information Forensics and Security*, 10(8):1776–1791, Aug 2015.

[3] Peter Gazdík. *Využití heuristik při obnově hesel pomocí GPU* [online]. Vysoké učení technické v Brně. Fakulta informačních technologií, 2016 [cit. 2017-04-08]. Dostupné z: http://hdl.handle.net/11012/62088. Bakalářská práce. Vysoké učení technické v Brně. Fakulta informačních technologií. Ústav informačních systémů. Vedoucí práce Radek Hranický .

[4] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[5] Mathew Weir, Sudhir Aggarwall, Breno de Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *2009 30th IEEE Symposium on Security and Privacy*, pages 391–405, May 2009.

[6] Alexander Meduna. *Automata and Languages: Theory and Applications*. Springer Verlag, London, GB, 2000.

[7] Christopher D. Manning and Hinrich Schiitze. *Foundations of statistical natural language processing*. Mass.: MIT Press, Cambridge, 1999.