Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Gábor Kövesdán

HEURISTIC PATTERN MATCHING OF POSIX REGULAR EXPRESSIONS

SUPERVISOR
Gábor Bányász

BUDAPEST, 2011

Table of Contents

1. Kivonat

A POSIX reguláris kifejezések implementációi hagyományos esetben DFA vagy NFA automatát valósítanak meg. Az előbbi a nagy állapottér miatt bizonyos esetekben nagyon sok memóriát használhat fel, megvalósítása bonyolultabb, nagyobb terjedelmű. Az NFA megvalósítás egyszerűbb, memóriafelhasználása optimális, de teljesítményben elmarad a DFA implementációktól.

A reguláris kifejezések illesztése kiemelkedően fontos feladat, mert sok alkalmazás igényli azt. A FreeBSD saját grep programján történt fejlesztések világossá tették, hogy a gyenge teljesítmény bizonyos esetekben felhalmozódhat, és kritikussá válhat, ugyanakkor az extrém memóriahasználat sem megengedhető, főleg a beágyazott rendszerekben történő felhasználás miatt.

A GNU grep a C könyvtár implementációját megkerüli és heurisztikákat használva jobb teljesítményt ér el. Ezt az elvet érdemes lenne a reguláris kifejezések implementációjában megvalósítani. A GNU grep azonban egy független szoftverkomponens, és nincs befolyása a C könyvtár felett, csak így tudja elérni, hogy minden környezetben gyors legyen.

A dolgozat szerzője úgy döntött, hogy a FreeBSD operációs rendszer C könyvtárában létrehoz egy hatékony reguláris kifejezés implementációt, így minden ráépülő program profitál a hatékonyságából. Habár az ötlet részben már létezett a GNU grepben, a vizsgálatok alapján ez az első általános célú reguláris kifejezés implementáció, amely így működik. A dolgozat bemutat pár elméletileg megfontolandó módszert arra, hogy hogyan lehet heurisztikusan közelíteni a reguláris kifejezéseket, és ezáltal jobb teljesítményt elérni.

2. Abstract

Implementations of POSIX regular expressions usually use a DFA or NFA automaton. The former tends to have a big memory footprint because of its big state space and its implementation is more complex and less concise. The NFA implementation is simpler, has an optimal memory footprint but it does not perform so well as DFAs.

Pattern matching of regular expressions is a crucial task because a great many of applications need it. The development of FreeBSD's grep utility has shown that the poor performance can sum up in some situations, becoming critical. In contrast, an extreme memory usage cannot be allowed, especially because of FreeBSD's common usage as an embedded platform.

GNU grep uses some shortcuts against the C library implementation and achieves a better performance by using heuristics. This principle should be implemented in the regular expression matcher. However, GNU grep is an independent component and has no control over the C library, so it can only achieve the same efficiency in all environments in this way.

The author of this paper decided to create an efficient regular expression implementation for the C library of the FreeBSD operating system, so that all utilities that depend on it can benefit from its efficiency. Although the idea already existed partly in GNU grep, according to preliminary research, this is the first general purpose regular expression implementation that works in this way. This paper explains some potential techniques of how to approximate regular expressions with heuristics and achieve a better performance.

3. Introduction

This paper is a case study on replacing GNU grep and the old regex engine in FreeBSD and tries to summarize all of the relevant conclusions. First, some pieces of background information are provided that explain the importance of this project and also summarize the factors that had to be taken into account. After this introductionary information, a section describes literal pattern matching and its possible generalizations and extensions and then another section talks about possible heuristics that can be used to approximate regular expressions with literal or almost literal fragments that can be matched with a faster algorithm than the usual automaton. As we will see, the different approaches have different advantages and disadvantages so the concrete decisions may differ in different situations. This work focuses on the overall performance and tries to choose those methods that are the most efficient in general cases. After the explanation of the heuristic approaches, a short benchmark shows the current performance that has been achieved so far, as of 2011 October.

The subject explained here supposes some basic knowledge in automata theory and programming in POSIX environment. For those, who do not feel confident in these areas, it is recommended to read the short glossary at the end of the paper that contains the most important acronyms and terms. If necessary, the referred bibliography also explains about the subject.

4. Background

In the FreeBSD Project, it has always been an objective — although a lower priority one — to get rid of GNU software. It has various reasons. First of all, GNU software is licensed under the GPL license, which does not suit very well the BSD philosophy. Although it guarantees the software itself and any of its derivative works to remain open source, it is not actually free in the terms of how the BSD community interprets freedom because it limits the commercial use and companies cannot build their products on these pieces of software and keep their business secret at the same time. This does not motivate companies to use these pieces of software and participate in their development. Another approach is to use a more permissive license that allows commercial use without the obligation of contributing anything back. Nevertheless, a great many of companies that use such software decide to contribute back. In such a way, they get the support of the community: highly experienced developers can review the code, users can try it out and report back bugs, and so on, which can lead to a better product for the company. This is definitely a mutual interest.

The replacement of GNU software is not just about the license. Instead of strictly conforming to standards and having a clear policy on API changes, GNU software tends to be very authoritative. The developers often just take non-standard solutions and they do not seem to have a strict policy against breaking the API or breaking compatibility between two version. BSD developers are more conservative and cautious in this question but depending on GNU software prohibits them to follow a stricter policy.

Sometimes there are also other considerations but these two are most probably those of highest importance. In 2008, the author of this paper made efforts to replace the GNU grep program in FreeBSD with a BSD-licensed version. The grep program is a small command-line utility that is used to look for matching lines in files for a specified regular expression. [1] During the development, it was discovered that GNU grep is huge but extremely fast because it uses some internal optimizations to heuristically approximate the possible matching fragments of the text and only calls the regex engine [2] on such fragments. What

it actually does is taking the longest literal part of the pattern or patterns and approximating the matches with either the Boyer-Moore or the Commentz-Walter algorithm, depending on if we have one or multiple patterns. This means, the major part of the text is processed with an alternative algorithm that is faster than the automaton. This strategy significantly cuts down the processing time. It was also seen that GNU grep sometimes consumes too much memory because it uses a DFA implementation and it is not just a theoretical threat. A developer of a company that uses FreeBSD in embedded systems reported such cases.

These facts lead to the following conclusions:

- It would be more beneficial to implement various strategies in the regex engine instead of the grep utility because the code would be more cohesive, keeping the utilities small and clean. Furthermore, all utilities that use regex pattern matching could benefit from it. However, it is not possible with a totally POSIX-conformant API as explained later becasue it uses NUL-terminated strings and not byte-counted buffers, which requires reading the input text character at a time. Besides, GNU grep has to include the optimizations if it wants to remain fast on all platforms because it has no control over the libc implementation that is present on the system. In contrast, a new regex library could integrate these ideas providing an alternative interface for byte-counted buffers.

- Performance is crucial because in large searches the drawback can sum up. It suggests a fast DFA implementation but it is very complex and can take too much memory. An NFA implementation is simpler and more memory-friendly but slower.

- If the major part of the search is done with a faster algorithm and the automaton is rarely called to verify the heuristically approximated match, the performance drawback of NFAs shall become moderate, while still keeping the low memory consumption, so a heuristic NFA matcher seems to be a good trade-off.

In this phase of the development, it was decided to get a new and efficient regex engine first. The currently used implementation is that old one written by Henry Spencer. The author himself stated that this implementation was not efficient and the code is not maintained any more. The following requirements were pinned up for the future regex engine of FreeBSD:

- Must be written in C/C++, preferably C because it will be part of libc.
- Must have a good license.
- Must be efficient, yet should use NFA instead of DFA or somehow limit the memory usage.
- Must have a clear and easily maintainable code.
- Must have good I18N-capabilites. Should accept wide string input, not just single- and multi-byte strings.
- Must be POSIX-conformant.

There was an implementation that suited all of the requirements and it was TRE, which is an open source, portable and general-purpose regex library. Besides, it has an alternative interface for wide strings and also accepts byte-counted buffers, not just NUL-terminated strings. TRE was designed with efficiency considerations in mind and its memory consumption is also predictable. [5] The plan was to integrate it to FreeBSD's libc and extend it with the heuristic capabilities. The heuristic capabilities will be integrated into TRE but they will form a new layer on top of the implementation, just like in GNU grep. They will try to solve the matching or narrowing down the context and only call the automaton if necessary.

5. Literal Matcher for TRE

Literal matching is the first important step to optimize the regular expression engine. It will be used directly for literal patterns and it is the basis of heuristically approximating non-literal patterns. Literal matching is not a new problem. There are numerous existing algorithms but they are very general and need adaptation to be used in a regular expression matcher. There are a couple of problems that are not present in purely literal matching:

- Both pattern and input text can be provided as single- or multi-byte or wide string.
- Matching may be case-insensitive.
- Sometimes it is important where the match begins and ends, sometimes we only care about the fact if there is a match or not.
- The dot character and the beginning of line and end of line anchors are relatively easy to handle. It would be a good idea to adapt the algorithm to handle these characters, as well.

The Turbo Boyer-Moore and the Quick Search algorithms will be used here as a starting point. These are well-known algorithms and there are various detailed descriptions and sample implementations online [6] [7] [8], so they are not detailed here.

These algorithms work for cases when the pattern (that is, a shorter text) is specified and processed first and the input (a longer text) can vary in future matches. In the generalizations of the algorithms, one important consideration is to keep the number of the matching phase steps low and possibly trade them off for extra steps in the processing phase. The pattern is short and only processed once, while the input text is longer and there tend to be more matching tries (that is, usually scanning the whole text for matches), so it is usually negligible if new steps are introduced in the processing phase but extra steps in the matching phase can sum up.

The adaptation to the dot character and anchors and the reverse matching are ideas that were implemented by Sean C. Farley <scf@FreeBSD.org> for freegrep. The rest is introduced by the author.

It is also worth to note here that when multiple patterns are searched, the input text has to be scanned multiple times because the algorithms mentioned above only deal with one pattern. There are algorithms that can handle multiple patterns at the same time and find matches by reading the input text only once. One such algorithm is the Commentz-Walter algorithm that is used by GNU grep. What GNU grep does is constructing the longest literal fragment for each regular expression and finding possibly matching lines with this algorithm. However, this algorithm is not applicable here because:

- The grep utility is line-based (as if the `REG_NEWLINE` flag were used), while general regex matching is not. If the longest literal fragment is not a prefix of the pattern, the input text has to be scanned from the beginning of the text or beginning of the line of our match is line-based. The former is definitely impractical but the second shall be efficient in practice.
- The POSIX API is limiting because `regcomp` only takes one regular expression, so it is not possible to process all the patterns together by using the standard interface.

5.1. Handling Different Kinds of Input

Storing the pattern in one of the forms would cause that the input supplied in the other form would have to be converted at matching time. As described before, it is not really desired because it will significantly decrease the performance. It is a better idea to reserve some extra space and store the pattern in both forms. The POSIX standard says that matching regular expressions is based on the byte sequence representation of the character and not on the actual meaning of it, so the common algorithms can be used on `char *` input regardless if it contains a single-byte or a multi-byte character. However, the byte-length and the number of characters is not necessarily the same, so the shift values have to be calculated for the both forms but it is done in processing time so the overhead is negligible. Patterns are usually not long so this approach does not take too much memory and computational time, while the performance gain is huge.

Nevertheless, using the algorithm for wide strings raises a problem. When working with `char *` strings, the bad character shift values are usually stored in and looked up from an array of integer values that is indexed with the byte value of the character. In all systems that support Unicode, `wchar_t` is defined to be a 4-byte integer type because the whole Unicode character set does not fit in a more narrow type. If we only use one byte for shift value that still means 2^32 bytes, which is more than 4 GB, so it is practically impossible. Fortunately, the number of distinct shift values is equal to the number of distinct characters in the pattern, and the rest of the characters have the same shift value. This suggests storing a default value and using a hash table for the distinct values. If the hash table is implemented properly, it adds only a small overhead compared to the traditional solution, and still, the pattern was also stored and processed in `char *` forms, so this overhead only applies to wide string input.

5.2. Adapting the algorithm to case-insensitive match

The conventional solution for this problem is using single-case pattern and single-case input. However, this means that the whole input text would have to be converted to single-case, which adds a lot of extra steps at the matching phase. It is a better solution to store the bad character shift values for both cases and calculate the good suffix shift values on a single-case pattern because it trades off the matching phase steps for some extra processing time. In this way, only those characters have to be converted at matching phase that are actually compared. If we have big shift values, it may be more efficient but in worse cases it may actually be much worse. Probably, it can be decided by analyzing the pattern, which strategy to use.

5.3. Handling Dots and Anchors

Both basic and extended regular expressions define beginning of line (^) and end of line ($) anchors. It is actually very simple to check these and they do not require running the automaton. What they actually need is a check at the borders of the match; whether they are the first or last character or whether they are preceded or followed by a newline character.

The dot character, which matches any single character is also very easy to implement in the Quick Search algorithm. Actually, it does not need any other adaptation than modifying the comparing code to always return `true` for dot

and limiting the maximum bad character shift, based on the last dot. It has to be checked, which is more efficient: using the Quick Search algorithm with a smaller maximum shift and avoiding the automaton totally or using a purely literal heuristic (maybe longest literal fragment or prefix) and then call the automaton.

In fact, there are more considerations than those described above because the `REG_NEWLINE` processing time flag and the `REG_NOTBOL` and `REG_NOTEOL` matching time constants influence the actual behavior of the anchors and the dot character but it is straightforward to add these checks to the code.

5.4. Reverse Matching

If matching the dot character is implemented as described in the previous section, the last dot can significantly limit the maximum shift value. If reversing the pattern the last dot is earlier, it would result in a higher maximum shift and the inverse of the Quick Search algorithm could be used, that is, using the reversed pattern and starting to match from the end of the text. This would result in better performance, however `regexec` must return the first match through its *`pmatch`* parameter, unless the pattern was prepared with the `REG_NOSUB` flag, so it can only be used when the pattern was compiled with this flag.

6. Using Heuristics

This section describes the kinds of heuristics that have been considered for the optimization. It also explains some of the factors that influence efficiency and applicability. As we will see, there is no best solution and to find one that is good enough, the techniques have to be tried out to check, which ones perform well in practice.

6.1. Longest Literal Fragment Heuristics

This is the technique that GNU grep uses and it consists of taking the longest literal fragment of the pattern and using that one to locate possibly matching fragments. The heuristic itself is not that accurate (compared to the array of heuristics, for example) but the idea is to be able to scan the text with big shift values. Furthermore, the longer the literal fragment is, its statistical occurrence will be lower in a random text. Although a prefix heuristic can exactly tell where the possible match starts, it may also give more false positives if it is shorter than the longest literal fragment.

The most important problem with this heuristic is that it cannot be applied in any general case. The grep utility is line-based, that is, there cannot be any newline characters inside the matching text (just as if `REG_NEWLINE` were used). So if we find a match for the heuristic, what we should do is isolating the matching line and pass it to the automaton. With general regex matching, newlines can also occur inside the matching text, so even if we found a match for the heuristic, we still would have to scan all the previous text. This means that this technique is not really applicable in general regex matching, except when the pattern is compiled with REG_NEWLINE, which mimics the grep behavior.

6.2. Prefix Heuristics

This is the simplest form of using a heuristic. The idea is simply taking the longest literal prefix of the pattern, for example, using “int” instead of “int[13][26]_t”. The prefix is quite simple to construct, all that has to be done is to read the pattern from the beginning and stop at the first unescaped special character.

The problem with this heuristic is that if there is an early false positive in a long text and then a long fragment without matches, then a big part of the text will be read by the automaton. If the pattern represents a fixed-length text — that is, there are no repetition characters or collating elements that consists of of multiple characters — it may be possible to limit the context, where the automaton is called.

6.3. Fixed-length Heuristics

It is usually possible to use a dot instead of a bracket expression because it still represents one character. One exception is when collating elements or equivalence classes are used because they break this rule. In general, this kind of heuristic can always be used as long as the pattern only matches a string from a given length, so it is referred here as fixed-length heuristic. In this case, the maximum shift of the literal matcher may be lower than with a prefix heuristic but it may avoid calling the automaton in more cases. For example, when the pattern is “int[13][26]_t” the prefix heuristic would be “int”, which would give a false positive for “int ret;” and the automaton would have to be called, while “int.._t” would avoid the automaton using a maximum shift of 2 instead of 3. So it is not definitively clear, which way is better. The efficiency of these two kinds of heuristics is a function of the concrete pattern, the statistical characteristics of the input string and the effective cost of calling the underlying automaton. In contrast to the prefix heuristic, a false positive does not degrade the performance.

6.4. Heuristic Arrays

Even when matches can have variable-length, it is possible to give a more accurate heuristic, which will be referred here as a heuristic array. It is actually not a heuristic but an array of fixed-length heuristics that are constructed from fixed-length fragments of the pattern. For example, an input fragment that matches “fooo*ba[rz]z+abc” will also match the following patterns in this order: “foo”, “ba.z”, “abc”. In other cases, the we may have a heuristic array, where the end of the match is open because it ends with a repretition character. In such cases, a false positive may degrade the performance but the statistical occurrence of a false positive is lower with a heuristic array than with a simple prefix heuristic.

The consideration that we had about fixed-length patterns, also applies here: “ba.z” has a maximum shift of 1, while “ba” would give 2.

Besides, there is another factor to consider. If the statistical characteristics of the input text are such that a matching prefix is usually followed by a whole match then the intermediate part is checked by both the literal algorithm and the automaton. In such cases, a simple prefix heuristic could result in a better performance.

It may also be a good idea to drop short intermediate heuristics that have a low maximum shift and only rely on the longer fragments.

7. Other Considerations

Having an efficient regular expression engine does not guarantee that all of the utilities that link to it will be in fact efficient. There are some considerations and guidelines that we will have to take into account when writing utilities that use regular expressions. These are quite inter-related issues and require some extensions over the standard POSIX API.

7.1. Use Byte-Counted Buffers

The POSIX API functions use NUL-terminated strings and not byte-counted buffers. This requires reading the input text one character at a time to find the end of the input and thus ruins the literal algorithms that can shift various characters at once. Fortunately, TRE introduced an alternative interface, providing the `regncomp` and `regnexec` functions that take the length of the pattern and the input text respectively. This does not just eliminate the need to read all the input characters but allows using NUL characters in the patterns and the text. Although it is not standard-conformant any more, the difference is very small and thus portability can easily conserved with preprocessor macros.

Although TRE's solution is definitely more elegant, it is worth to note that Henry Spencer's implementation that became like a “de facto” standard in BSD distributions, also offered a non-standard solution to specify where the pattern ends. This was the `REG_PEND` flag. And the another BSD-specific extension, the `REG_STARTEND` flag also allowed calculating the length of the input text.

7.2. Using REG_NEWLINE

When the utility is line-based, that is, a match cannot overlap multiple lines, it is usually redundant to split the input text to lines and call `regexec` for all of the lines. Splitting the lines requires reading the input characters one-by-one to find the newline characters, just like in the case of NULs, as described above. It is more practical to process the pattern with the `REG_NEWLINE` flag and let the regular expression library quickly skip lines by shifting with the proper amount. It is only possible if we can use byte-counted buffers through some extensions described above.

One exception from this shortcut is when we want to count lines for some reason, for example, with the `-n` option of grep. In this case, it is actually necessary to read the input characters one-by-one.

7.3. Avoid Copies

When files are read or lines actually have to be separated, it is a common error to copy data into an internal buffer. All copies should be avoided and the raw buffer should be passed to `regexec` directly. It can only be done if the raw data is either NUL-terminated or there is any way to tell the matcher how long the input is, for example using the non-standard `regnexec` or the `REG_STARTEND` flag. When reading from files, the easiest way is to use the `mmap` [3] system call and pass the mapped memory directly to the matcher.

8. Benchmarks and Conclusions

8.1. The Test Program

The test program used for the benchmarks is a little grep-alike program. It uses `mmap` to map the files without any copies and then traverses the content by calling `regexec` with the `REG_STARTEND` flag. These decisions were to reduce the overhead to the possible minimum. It only has options to recursively scan a directory, to use BRE, ERE or literal match, to work in case-independent mode or to use `REG_NEWLINE`. These are all important tunables for the benchmark. The program does not really implement any error-checking features. If one file cannot be opened, it will be just skipped. At this point, there may be bugs in the heuristic code but the code generally works as expected. It is important to make sure the proper behavior because if the new matcher code does not work correctly, the comparison makes absolutely no sense.

8.2. The Test Environment

The test environmet was FreeBSD 10-CURRENT from October 2011. Being a development version, it was compiled with malloc debugging options that add some overhead but what we are interested in is the relative performance of the pure and the heuristic regular expression code. The tests were run on three different implementations: the current one in libc developed by Henry Spencer, TRE without heuristic features and TRE with heuristic features. The kernel has been recompiled to remove some kernel debuging options and superfluous device drivers and the system was started in single user mode to disable background programs that take CPU time from the running tests.

8.3. The Results

Some vague comparisons showed that generally the most efficient solution is to always shift as much as possible, that is, purely literal heuristics are better than those containing dots and the patterns that contain dots can be matched faster if it is approximated with a purely literal heuristic than using the generalized Quick Search algorithm. However, that may still be a useful generalization for less feature-rich search languages that support a joker character.

It can also be seen that in some cases the current libc implementation is still faster. The reason is probably that it is a DFA implementation but if we take a look at the corresponding results of the heuristic matcher, we can see that the difference is usually quite small and it scales quite well as the length of the pattern grows.

In general, the results are pretty positive. The performance growth is higher than 40 % in all tested cases and in some cases it is more than 80 %.

The following table summarizes the results for the three cases and the performance growth between the original TRE code and the optimized one. All measurements were taken five times after some ignored tries because the first runs tend to be slower because of caching. The average of the five results was taken. The test data was the whole `/usr/include` directory of the system.

As it can be seen, the patterns have been chosen to test the specific literal matching and heuristic features. There are fixed patterns with distinct letters and also periodic ones. Then we have fixed length patterns and such that are not fixed-length but can be approximated with a heuristic array.

Table 1. Test Restults

Pattern	**libc-regex**	**TRE**	**TRE-heur**	**Performance growth**
a	1.29 sec	1.2 sec	0.14 sec	88.33 %
abc	0.24 sec	1.51 sec	0.57 sec	62.25 %
abcdefgh	0.15 sec	1.51 sec	0.31 sec	79.47 %
abcdefghijklmnopqrstuv	0.14 sec	1.51 sec	0.21 sec	86.09 %
abcdabcd	0.15 sec	1.51 sec	0.31 sec	79.47 %
abbabbbaaba	0.15 sec	1.51 sec	0.26 sec	82.78 %
unsigned.*int.*;	4.88 sec	2.05 sec	1.17 sec	42.93 %
unsigned.*	2.92 sec	1.65 sec	0.83 sec	49.70 %
unsigned..	0.22 sec	1.45 sec	0.27 sec	81.38 %

9. Conclusion

The results have shown clearly that it is possible to increase the performance of automaton-based regular expression engines by using heuristics. These ideas may be generalized and used with other formal languages. It was also clear that the greedy approach was the efficient one, that is, using always the best maximum shift possible. However, this paper only reflects the current status of this research. For example, the longest fragment heuristic could also be implemented exclusively for cases, when the `REG_NEWLINE` flag is used, or although dots are not used in the heuristics, the length of the matching fragment could be calculated for those patterns that match a fixed-length text and this could be used to further limit the use of the automaton to a more narrow context in some cases. It would also be useful to check the method presented for `REG_ICASE` and he traditional single-case approach for patterns of different length. There may be other little tricks and ideas to try out and it is also worth to use a profiler to find the pieces of code that are called most often so that the most critical bottlenecks can be found. Hopefully, these ideas will be checked soon as part of this project.

Bibliography

[1] *IEEE Std 1003.1-2008*. The Open Group Base Specifications Issue 7. 9. Regular Expressions. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09.

[2] *IEEE Std 1003.1-2008*. The Open Group Base Specifications Issue 7. regcomp, regerror, regexec, regfree - regular expression matching. http://pubs.opengroup.org/onlinepubs/9699919799/functions/regcomp.html.

[3] *IEEE Std 1003.1-2008*. The Open Group Base Specifications Issue 7. mmap - map pages of memory. http://pubs.opengroup.org/onlinepubs/9699919799/functions/mmap.html.

[4] *Introduction to Automata Theory, Languages, and Computation*. 2nd edition. J.E. Hopcroft. R. Motwani. J.D. Ullman. ISBN 0-201-44124-1.

[5] *TRE - The free and portable approximate regex matching library*. About. Ville Laurikari. http://laurikari.net/tre/about/.

[6] *A Fast String Searching Algorithm*. Robert S. Boyer. J. Strother Moore. http://www.cs.utexas.edu/~moore/publications/fstrpos.pdf.

[7] *A Very Fast Substring Search Algorithm*. Daniel M. Sunday. http://delivery.acm.org/10.1145/80000/79184/p132-sunday.pdf?ip=152.66.152.135&acc=ACTIVE%20SERVICE&CFID=65026445&CFTOKEN=46914591&__acm__=1319712540_a83791abbc9e261a492dd469f2e0963a.

[8] *Exact String Matching Algorithms*. Christian Charras. Thierry Lecroq. http://igm.univ-mlv.fr/~lecroq/string/string.pdf.

[9] *Efficient submatch addressing for regular expressions.*. Master's thesis. Ville Laurikari. http://laurikari.net/ville/regex-submatch.pdf.

[10] *freebsd-current Mailing List*. Why is GNU grep fast?. http://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html.

Glossary

A

Application Programming Interface (API)
: The set of public functions of a subsystem that allow external components to use the provided functionality.

Automaton
: Model of a machine that has states and transitions. The transitions take input characters and change the state of the machine based on the input character.

B

Berkeley Software Distribution (BSD)
: A UNIX-derivative operating system that was developed at the University of California, Berkeley. Today, there are different operating systems that emerged from BSD, like FreeBSD. They are not considered to be UNIX-derivatives any more because the original UNIX code has been replaced.

BSD license (BSDL)
: Free software license that originates from the BSD system but is commonly used nowadays. It permits commercial use and closed source derivative products.

D

Deterministic Finite Automaton (DFA)
: An automaton that works deterministically, that is, receiving a specific input character in a specific state can have at most one transition to another specific state. Transitions on empty input are not allowed.

F

FreeBSD
: An open source operating system that emerged from BSD.

G

GNU General Public License (GPL)
: GNU's free software license. Requires that derivative works remain under the GPL license. Commercial use is not directly limited but modifications and derivative works must be open source, which is not always beneficial for commercial vendors. Because of this property, this license is sometimes called “viral” among developers that do not agree with this limitation.

GNU's Not Unix (GNU)
: A free software project that develops an open source operating system and related pieces of programs. They use the GPL license.

grep
: A command-line utility defined by POSIX that scans files for lines that match a specified regular expression.

I

Internalization (I18N)
: The process of making a software work properly with several input/output languages, taking into account the specific characteristics and features of each one.

N

Non-deterministic Finita Automaton (NFA)
: An automaton that allows several transitions for the same state and input character combination, or even transition on empty input. Thus, an NFA can be in several states at the same time.

P

POSIX
: A standard that pretends to ensure the portability of software.

R

regular expression (regex)
: A formal language that can be used for pattern matching, which in turn is used for software features like advanced search or input validation.