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

Gábor Kövesdán

OPTIMIZING SOFTWARE WITH HEURISTIC ALGORITHMS

SUPERVISOR
Gábor Bányász

BUDAPEST, 2012

Table of Contents

1. Kivonat

A számítástudomány területének eredményeképp manapság már rengeteg problémára létezik hatékony algoritmus, valamint a hardveripar rohamos fejlődése is segít az egyre bonyolultabb számítások elvégzésében. Ennek ellenére a szoftver hatékonysága nem elhanyagolható és néha az általános megoldások nem bizonyulnak kielégítőnek a teljesítmény szempontjából. A heurisztikák használata jellemzően csak nevezetes problémákra terjedt el, pedig sok esetben javíthatunk a teljesítményen ilyen algoritmusok alkalmazásával. A heurisztikák lényege, hogy gyorsan szolgáltatnak eredményt és bár az így kapott eredmények nem pontosak, sok esetben kielégítőek. Ha pontos eredményekre van szükségünk, a heurisztikák néha akkor is segíthetnek jelentősen csökkenteni a számítási igényt. Jelen dolgozat a heurisztikák témakörét vizsgálja meg részletesen, majd egy esettanulmányon át mutatja be, hogyan használhatóak ki a bemutatott ötletek egy valós rendszerben. A bemutatott technikák elég általánosak, így sok felhasználási területen alkalmazhatók.

2. Abstract

The discipline of computer science has achieved that nowadays we have efficient algorithmic solutions to a great many of problems and the rapid development of the hardware industry also helps dealing with complex computations. Despite this, software efficiency remains a crucial point in software development and sometimes general solutions do not offer a satisfying performance. Although heuristic algorithms are typically only used with some particular problems, they may significantly improve performance in a wider range of problems. Heuristics more quickly give results to problems and although these results are not punctual, they are often satisfying. Even if punctual results are expected, heuristics can contribute to significantly decreasing the computational need. This article introduces the topic of using heuristics and provides a case study that presents how such techniques can be used in a real software systems.

3. Acknowledgments

I would like to say thanks to Xin Li, who helped me joining the FreeBSD Project, where I got involved with this topic and to Mike Haertel for sharing some thoughts on grep performance, which lead me to the ideas and solutions presented in this article. Thanks goes to Ville Laurikari for having developed TRE, an efficient and modern regular expression matcher, which also inspired the present work in several aspects. Last but not least, I thank Gábor Bányász, who accepted my inquiry and volunteered to supervise this article for the “TDK” students' scientific conference and I thank all the people, who are not explicitly mentioned here but helped my professional growth as a software developer.

4. Introduction

In 2011, I wrote an article about heuristic optimization of POSIX-compliant pattern matching, which covered some simple techniques on how to improve the performance of usual implementations of pattern matchers by using heuristics. This was based on my experiences with the development of BSD grep, a BSD-licensed implementation of the grep utility defined by the POSIX standard. I noticed that my implementation was trivially simple — it was almost nothing more than a wrapper around the regex implementation in the C library — so I expected it to be efficient but soon I had to realize that the GNU grep implementation performed significantly better. Looking at the code and having an email conversation with the original author, I relized that they used some heuristics to speed up pattern matching. This was my first encounter with the idea of heuristic optimization. I liked the idea but I felt that such optimizations did not really fit into the grep utility but that they should have been integrated into the main regular expression code so that utilities can use it and benefit from this performance boost transparently. I decided to create a regex library by enclosing such optimization logic into the library itself. I took the TRE library as a starting point and I started generalizing the heuristic algorithm since grep was special in some aspects. For example, it always searches for matches inside lines, or in other words, matches cannot overlap two lines. I started working on this project but I realized that it could not be done with the standard POSIX interface, so I used a slightly modified one — which was apparently already available in TRE — that takes the input lenght as well. I successfully implemented the objectives and I published the results in my previous work.

But the project did not stop there. I realized that it was still not possible to efficiently search for multiple patterns since the interface only accepted one pattern at a time and searching for more patterns required looping over the input text multiple times. I realized that working with heuristics was simple and I could easily generalize my solution to efficiently work with multiple patterns and I provided such an interface. At the same time, I decided to refactor my implementation to be an independent library depending on TRE and later I plan to make it more portable so that it can be used with any type of POSIX-compliant regex implementation. In this article, first

I present some theoretical background on heuristics and then I explain my pattern matcher optimization as a case study.

5. Using Heuristics in Problem Solving

5.1. Approximation with Heuristics

First, to talk about heuristics, we should have an idea of what they are. Most probably, everyone have heard this word quite some times since we use a great variety of heuristics in real life but we do not normally give a formal definition of them. A heuristic is considered to be a quick but less precise way of processing a calculation. We can consider it an approximation that gives us a good estimation but not an exact result. For example, measuring distance by considering one step as one meter is a usual heuristic, which is obviously not punctual but definitely faster than an exact measurement with proper tools. Sometimes this punctuality is not enough but in some cases it is. I will give here a formal definition so that I can more precisely talk about heuristics.

Let us consider a decision problem, which is solved by an $f(x)$ function. I talk about decision problems here but other problems can easily be rephrased to be decision problems, for example, instead of `what is the maximum value of` $g(x)$`?` we can ask `whether` $g(x) \geq n$ `?` , which is, in turn, a decision problem. From now on, I will use the $c(f, x)$ notation to detonate the cost (either time or space cost) of the $f(x)$ function on input x. Since we are talking about decision problems, I will also use the notation used in automata theory and the language accepted by the $f(x)$ function will be referred as L_f . The context I am basing this chapter on is that we have a decision problem and we need to decide for a series of inputs whether they belong to the corresponding language or not. What I will try to optimize is the overall performance that the user perceives after running the algorithm on a whole series of inputs. However, it may happen that for some individual cases the cost is higher but I will mostly concentrate on the average performance.

I define a heuristic with the following criterion:

Equation 1. Definition of full heuristic

$$\forall x \in \Sigma^* : c(h, x) \leq c(f, x)$$
$$\forall z \in \Sigma^* : z \in L_f \Longrightarrow z \in L_h$$

This means that the heuristic accepts at least those inputs that the generic algorithm does but it may accept more and its cost is always less. The accuracy of the heuristic can be observed from two different viewpoints. The first and more intuitive concept of accuracy somehow measures the difference between inputs that are accepted by the original algorithm and those that are accepted by the heuristic. For example, if we consider the inputs to be strings, it can be the maximum Hamming-distance between the two mentioned sets of inputs. The second concept of accuracy considers the two original and the heuristic algorithms languages and tries to describe the relation between the accepted inputs of them. The less additional inputs the heuristic accepts the more punctual it is since the closer it is to the original problem. With finite input sets we can define a **accuracy ratio** that quantifies this characteristic but with infinite input sets there is no good quantification.

Equation 2. Definition of accuracy ratio

$$\frac{|L_f|}{|L_h|}$$

As it can be seen, I called the heuristics that satisfy this definition full heuristics, since their cost is always lower than the cost of the original algorithm regardless of what the actual input is. Sometimes it may be enough if the heuristic has only lower cost in some particular cases. The following definition describes this type of heuristics.

Equation 3. Definition of partial heuristic

$$\exists\, x \in \Sigma^* : c(h, x) \leq c(f, x)$$
$$\forall\, z \in \Sigma^* : z \in L_f \Longrightarrow z \in L_h$$

Considering a case when we have some statistics of the input and there is a partial heuristic available that approximates the problem more efficiently for inputs that satisfy the criterion A but less efficiently for the rest, we can calculate the average processing cost with the given input statistics. If it is lower than the cost of the generic algorithm, we may consider using this heuristic.

Note

As a side note, it is worth to add that the above definitions and the whole article focus on time costs but we may as well define storage-

based heuristics and use the very same techniques to optimize the storage consumption. This may be really useful if the storage resources are limited, for example, in embedded systems.

As we have seen, sometimes we can find partial heuristics for a subset of the possible inputs. It is also possible that we can find several different partial heuristics that cover different subsets of the set of inputs. In this case, it is also possible to introduce a **heuristic selector function** that examines the input and selects the possible heuristic to use.

Equation 4. Definition of heuristic selector function

$$s: X \longrightarrow H$$
$$h: X \longrightarrow Y$$
$$h(x) = \begin{cases} h_1(x) \text{ if } c_1 \\ h_2(x) \text{ if } c_2 \\ \cdots \\ h_n(x) \text{ if } c_2 \end{cases}$$

When choosing such a solution, it should also be taken into account that the heuristic selector function adds an additional cost to the processing so it is only profitable if the overall performance is still better than using a generic algorithm. This may usually be true for three-phase algorithms. These have a compilation phase, in which they process some parameters and construct some internal structures used for the processing; an execution phase, where the actual processing occurs; and a clean up phase where the internal data structures are cleaned up and reserved resources are freed. Examples for such are the standard POSIX way of regular expression matching, which will be discussed in the case study and standard POSIX character conversion. In this kind of applications the compilation phase is only run once and the calculated internal structures are reused over and over on a set of inputs. As a consequence, extra steps in the compilation phase are negligible, while the steps done in the execution phase should be kept minimal.

Let us check the cost of using a heuristic selector function. Considering a single x_i input that is processed with a h_i heuristic, the total cost of this individual execution can be calculated as follows:

Equation 5. Cost of a single execution with heuristic selector function

$$c(s, x_i) + c(h_i, x_i)$$

From this, it is easy to see that the average cost will be:

Equation 6. Average cost with heuristic selector function

$$\sum_{x_I \in X} \frac{c(s, x_i) + c(h_i, x_i)}{|X|}$$

So far, we have seen some basic methods of using heuristics to speed up our computations. If the accuracy of the heuristics is acceptable, the solution is already done and the processing time has been reduced by changing to heuristics at a cost of loosing some accuracy. In turn, if exact results are needed, these techniques cannot simply be relied on but fortunately, there may be other solution.

5.2. Exact Calculations with Heuristics

In the above introduction, the definition of heuristics was carefully chosen to facilitate the next technique that is described here. Remember that heuristics were defined on decision problems in such a way that they accept at least the same inputs as the generic algorithm and they may accept some additional ones as well. In the approximative approach, it was more important at how extent each false positive differed from an actually accepted input but in this technique it will be more important how often the heuristic gives a false positive.

The technique consists in preselecting inputs with a fast heuristic and only using the generic algorithm if the preselection was successful. In this way, the expensive algorithm will only be executed on those inputs that could not be evaluated at the preselection phase.

The preselection heuristic can be of two types:

- A **positive preselection** can quickly accept some inputs but for some of them it gives a `cannot decide` answer. Only these inputs need to be passed to the generic algorithm.

- A **negative preselection** can quickly reject some inputs but for some of them it gives a `cannot decide` answer. In this case, these have to be further processed.

It may also be worth to create a chain of preselections if there are several efficient ones available that handle different cases but it is important to take into account the efficiency considerations explained below. The practical applicability of this technique depends on two main factors:

1. The heuristic should be chosen in such a way that it do not return too many false positives so that it can quickly disqualify a significant number of inputs without having to frequently run the expensive algorithm.

2. The statistical occurrence of accepted inputs has to be low enough. Consider the extreme case of only receiving accepted inputs in which case both the heuristic preselection and the generic algorithm will be run for all inputs. In this case the preselection just means an additional cost since using only the generic algorithm would have resulted in lower execution time.

Let us call the set of accepted inputs Y . In this case, the cost of processing an x_i input that belongs to Y can be calculated in the following way:

Equation 7. Cost of an accepted input

$$c(h, x_i) + c(f, x_i)$$

For an x_j input that does not belong to Y , we can calculate the following cost:

Equation 8. Cost of an non-accepted input

$$c(f, x_j)$$

Using this calculations, the average cost is:

Equation 9. Average cost

$$\frac{\sum_{x_i \in Y} c(h, x_i) + c(f, x_i) + \sum_{x_j \notin Y} c(f, x_j)}{|X|}$$

If the above calculated average cost is lower than the total cost of using only the generic algorithm, the first criterion is satisfied but we still have to take into account the statistical distribution of accepted inputs.

5.3. Creating Heuristics

We have seen before how heuristics can be used for problem solving but it has not been discussed yet how heuristic algorithms can be "found out". We will see some systematic methods to construct heuristics for our problems.

5.3.1. Using Necessary or Sufficient Conditions

If we can phrase an easily verifiable sufficient condition for accepted inputs, it may be a good heuristic, especially if it can quickly disqualify some inputs. Or it is possible to reverse the logic and if there is a necessary condition for non-accepted inputs, those inputs that do not meet this condition, cannot be non-accepted so they must be accepted.

5.3.2. Relaxing Criteria

If the accepted inputs are such that there is a set of criteria that they have to meet (or we can rephrase it as a necessary and sufficient set of conditions), it may be possible to leave out less important and less easily verifiable criteria from the requirements and this may speed up the calculations. In case exact results are needed, we can first use those criteria that more quickly and extensively disqualify non-accepted inputs and only run the rest of the checks later. In other words, if the algorithm is composed of a set of conditional checks, putting first those conditions that most probably fail will quickly disqualify numerous inputs and the later checks will not even have to be run. This cuts down the number of total processing steps executed. The order of these checks really do matter in terms of efficiency.

5.3.3. Randomized Algorithms

There are randomized algorithms that use random choices of parameters and they find a good witness with high probability for a decision problem but there is no 100% probability that they ever find a solution, the probability just approximates 1. Such algorithms can also be used with a step-bounded execution to quickly obtain a positive and negative answer and the expensive algorithm only needs to be executed in case the step-bounded execution of the randomized algorithm did not provide a good witness.

6. A Case Study with Heuristics

This section presents a case study on the concepts explained in the previous section from the subject of pattern matching. Pattern matching is clearly a decision problem and if the matching is used as a search, it involves several consecutive executions since matches may occur at any position of the input text. Therefore, this is an expensive use case of pattern matching. The other typical use case of pattern matching is validation but in this case the whole input has to match the whole pattern so there are no repetitive executions.

6.1. Background

As explained in the Introduction, this work has grown out from the development work on BSD grep, which is a POSIX-compliant command-line utility. Its initial version was obtained from the OpenBSD Project and I worked on improving it and extending its functionality for the FreeBSD Project. At FreeBSD, we wanted to replace the GNU implementation, which not only has a less attractive license for the BSD community but also follows some principles that we are not really committed to. The implementation I worked on became feature-complete to substitute the GNU version in FreeBSD and its code was so simple that I could not really see any further opportunities to improve the performance but I had to realize that the performance was far from satisfying. I had to investigate the reasons and the original author of GNU grep also helped me by sharing his experiences. The main reasons was that GNU grep used its own heuristics to limit the examined cases to some possibly matching contexts and only passed those fragments to the POSIX regular expression functions. I did not like the idea of implementing those in BSD grep since I thought they should have better gone to the regular expression library code to keep the utility programs smaller and cleaner and hide the complexity in the programming library but unfortunately, I had to realize that implementing such shortcuts was not possible with the standard POSIX interface so I decided to create a higher-level regular expression library that could form an intermediate layer between the standard regular expression library and the utility programs. In a previous work [1] I reported about some early results but it still lacked the multiple pattern interfaces and the single pattern solutions did not reach their final form either. Here, I will not talk about the single pattern cases with the same details

but I will summarize the most important considerations and will give an efficient algorithm.

Apart from the problems explained above, it had to be taken into account that the grep utility works with lines so matches cannot overlap two lines. This also simplifies the problem and there are more efficient and simpler ways of optimization in this case. Nevertheless, in some more general cases it is also possible to speed up matching. GNU grep only solved the single and multiple pattern problems of the line-oriented case but in a slightly different manner than I am doing and applying heuristics to general cases is my own addition. Also, I want to clerly state here that neither any exact solution nor any code has been directly copied from GNU grep. I only took the basic idea of the solution and I reimplemented it in the way I found the most appropriate for such a higher-level regular expression library that I decided to develop. This also permits that I license the result of my work under the BSD license.

6.2. How to implement the grep utility

The grep utility takes one or more regular expressions and reads either a file or the standard input and returns matching lines from the input. It shall return all the matching lines and in the order of occurrence if any of the specified regular expressions match any fragment of the line. There are also options, which colorize the matching part and that enumerate lines. Regular expression matching is greedy by definition, that is, always takes the longest possible matching part. This has to be taken into account when highlighting matches. Also, it can also add line numbering to the output.

From an implementer's viewpoint, this behavior means that the whole input text has to be scanned for matches and all of the matching lines in order have to be printed out. It is practically implemented by looping over the lines since if the first pattern does not match, another can still match the line. However, as we will see, it is not the most efficient implementation since it requires processing the input line by line, which means searching newline characters, which in turn requires processing the text character at a time to locate each line break. If we know the length of the input text — so we do not have to look for the ending NUL byte — and we can ensure not loosing any matches in this way, we could shift more characters at a time

and thus cutting down the processing steps. This suggests a different processing approach. Let us consider the single-pattern case first and let us suppose that we have and efficient algorithm that sometimes shifts more characters. We can just find the first match and then look for the line boundaries. In this way we only have to do character by character processing at a very narrow context, the preceding lines that did not match were processed with the efficient algorithm that shifted more characters at a time. This algorithm will be obtained by using the ideas exposed in the previous section. However, when using the line numbering functionality and some similar features, this optimization is not possible and we have to fall back to the original approach.

Now let us consider the case when multiple patterns are specified. It is possible to loop over all of the patterns but the first match cannot be just immediately processed since it is possible that another pattern gives an earlier match and the input text has to be processed sequentially. This suggests that a more advanced algorithm would be practical here that can handle check for all the patterns by reading the input text only once.

With the above considerations in mind, I will present below the algorithm solutions for both the single- and the multiple-pattern cases. The general cases will also be explained, where the processing is not line-oriented.

6.3. The Single-Pattern Case

There are several subcases in the single-pattern case but basically what will give us a performance boost is using a literal text matching algorithm, like the Quick Search Algorithm or the Boyer-Moore Algorithm. These algorithms can shift several characters in the input text and the maximum shift is directly proportional to the pattern length so the longer the pattern is, the more efficient its processing will be.

Most regular expressions contain some literal fragments and for many of them we can phrase a necessary condition that the line can only match the pattern if it contains the literal text fragment. With this in mind it is only necessary to run the full regular expression routine on lines that match these literal fragments. I have to emphasize that it will be efficient only if the statistical data of the input text is such that not all or too many lines match the pattern. This “too many” depends on

several parameters, most of which are changing on each run so it is hard to estimate but one extreme case is when all lines are matching. In this case both the literal and the full algorithm is run on all lines, so the literal algorithm means just an extra processing step. But as the efficiency of GNU grep shows, in practice, this method really does reduce execution time.

As explained before, the POSIX standard defines a three-phase processing API for regular expressions, which has a compilation, an execution and a cleanup phase. The extraction of the literal fragments requires a simple parser that can identify literal parts and this processing can be done in the compilation phase, which is only run once for a pattern. In turn, the matching phase is run numerous times so the overhead of this processing step is negligible. The literal matching algorithms also require some data structures and this task also fits nicely to this compilation phase.

As for the literal fragments that are actually used, there are several possible but slightly different approaches. If the matching is line-oriented or the pattern does not have any part that can match the newline character, we can simply look for the longest literal fragments, which will give the maximum possible shift value and thus will be the fastest one in going through the input text. However, this is only applicable with these conditions case, in which we know that the earliest possible point of the match is the beginning of the line and the latest possible point is the end of the line so we can clearly determine the potential context that need to be passed to the full matcher. In turn, in the general case, if the pattern may catch a newline, matches can overlap lines so even if we find an occurrence of the longest literal fragment in the last line, it is possible that the match starts at the very first character. Nevertheless, if the pattern starts with a literal string, it can be used to find at least a possibly matching starting position. This is required for the general case; for patterns that do not start with a literal fragment, there is no remedy than using the full algorithm. Once we have such a prefix fragment, we can take some intermediate fragments and a suffix fragment if such exist. It may also be the case that only the prefix fragment exists but it is at least enough to quickly traverse the text until we find a possible starting point of a match. A suffix fragment may also be very useful since it can limit the context where the full algorithm is called. If the occurrence of the initial fragment is a false positive, the full algorithm will continue running until it finds a match on its own or until the input is finished. In turn, if

we look for a suffix fragment, we can limit the context where the full algorithm is run. And similarly, we can use some intermediate fragments as well, as many as we want. Probably, most of the patterns used in practice are simple and do not contain too much isolated literal fragments but I implemented this algorithm in such a way that it also takes the longest intermediate literal fragment if available.

There is one more trick, which can be used to further limit the context on which the full algorithm is run. Sometimes patterns contain non-literal parts but we can make sure the length of the text fragment matched by these parts is constant. In these cases the total length of possible matches is constant and can be calculated. This implies that the context of the full algorithm can further be limited and it is not even necessary to deal with line boundaries or suffix fragments, just a sufficiently wide context has to be chosen for the full algorithm. This strategy is also applicable in the general case since it makes no difference whether this context contains a newline character or not.

Summarizing the above description, here is a description of how a single pattern is processed:

Procedure 1. Processing a Single Pattern

1. Compilation phase.
    a. Identify literal fragments.
    b. Determine length of matches if possible.
    c. Based on Step 1.a, Step 1.b and compilation parameters, determine the strategy to use.
    d. Run compilation steps for the full algorithm.
2. Execution phase.
    - If match was compiled as line oriented.
        a. Look for longest literal fragment, halt if not found.
        b. If match length is known, calculate context, if not, look for line boundary.
        c. Run full algorithm on the determined context. If matches, return match, if does not match continue at Step 2.1.a.
    - If match was not compiled as line oriented.
        - If prefix fragment is available.

a. Look for the prefix literal fragment, halt if not found.

b.
- If match length is known calculate context using it.
- If end fragment is available, look for it and determine the context with it.
- Otherwise, the context lasts until the end of the input.

c. Run full algorithm on the determined context. If matches, return match, if does not match continue with Step 2.2.1.a.

- If prefix fragment is not available, run the full algorithm on the whole input.

3. Clean up phase.

a. Clean up the literal matcher's structures.

b. Clean up the full matcher's structures.

The following example demonstrates how a single pattern is processed:

Example 1. Processing of a Pattern

Let us consider the `int[13][62]_t` pattern, which we can use to find declarations of `int16_t` and `int32_t` typed variables in a C source code. The pattern has two literal fragments: `int` and `_t`. Apart form this, it is known that the total length of a match is surely 7 characters, since the two literal fragments cover 3 and 2 characters and the bracket expressions cover one character each. The longest literal fragment is `int` so it will be used for the line-oriented case. Apparently, the pattern cannot catch a newline so the same approach can also be used in the general case. After finding the `int` string, the only thing to do is to properly mark the context, which starts at the first character `i` and covers 7 characters. If the pattern were `int.*_t`, it would require searching for line boundaries in the line-oriented case and using the `int` prefix fragment and the `_t` suffix fragment in the general case since `.*` can match newlines.

The algorithms have been completely covered above but I have to make one last side note about the interface. The POSIX standard defines the the execution phase call in the following way:

```
int regexec(const regex_t *preg,
      const char *string, size_t nmatch, regmatch_t
      pmatch[], int eflags);
```

As mentioned before, this interface is limiting since it passes the input text as a NUL-terminated string, which requires character at a time reading and does not allow leveraging literal matching algorithms that shift more character at a time so I needed a different interface in my library. I took TRE's approach, which defined the following API:

```
int tre_regnexec(const regex_t *preg,
      const char *string, size_t len, size_t nmatch,
      regmatch_t pmatch[], int eflags);
```

Apart from the new argument, I also used an implementation-specific prefix and wrapped the `regex_t` structure into an `regfast_t` one, which carries extra information that is needed for the heuristic searches:

```
int regfast_regnexec(const regfast_t *preg, const char *str,
      size_t len, size_t nmatch, regfast_match_t pmatch[],
      int eflags);
```

TRE also defines a similar interface for the compilation phase, which I adopted but in fact, it is not necessary since patterns have to be parsed entirely and they are practically short and the compilation phase is only run once, so this can be just considered a convenience feature.

6.4. The Multiple-Pattern Case

The multiple-pattern case is somewhat more complex. The basic idea to use here is that there are several literal matching algorithms that take multiple patterns and find the first occurrence of any of them by just reading the text once. One such example is the Commentz-Walter algorithm, which is an automaton-based approach and is used by GNU grep. Another example is the Wu-Manber algorithm, which is a more recent algorithm and is a generalization of the aforementioned Boyer-Moore algorithm for multiple patterns. I chose to use this algorithm, since it is easy to implement and seems very efficient. The complexity comes in selecting the sets of literal fragments to look for and how to deal with the context once such a match is found.

The line-oriented case or the general case when no patterns can match a newline is quite straightforward. The longest literal fragment of each pattern has to be extracted and all of them have to be processed together with the Wu-Manber algorithm. Once there is a match, the context is determined (either by the match

length or by the line boundaries) and the full algorithm is run for the potentially matching pattern. This is very much the same as the single-pattern case, just the literal algorithm is different.

One special case of the above scenario is when all of the patterns are completely literal patterns. In this case they have to just directly be compiled with the Wu-Manber algorithm and there is no need to call the full algorithm once a match found.

A slightly more complicated case is when the match is not line-oriented and some of the patterns can catch a newline. In this case, if all of the patterns start with a literal fragment, at least a potential starting position can be located and the full algorithm can be run from that point. Alternatively, it would be possible to start using a single literal matching algorithm from here but it may return a farther match as well and other pattern may match earlier so the different subcases become too complex to deal with so the efficiency of such solution is not trivially good. This is something that definitely needs a try to be evaluated since the actual costs are not practically possible to calculate so for now, I have not implemented such a complex logic but I just simply use the full algorithm from there.

Before going to the general case, I want to mention a trivial special case. The algorithm is just given an array of patterns but practically it should also accept a one-length array, in which case the Wu-Manber algorithm makes no sense and the processing should just fall back to the single matcher implementation. The implementation can hide all this logic and make this processing transparent to the user.

If no one from the above strategies can be used, there is not much to do. It is not possible to look for the patterns with one scan neither it is possible to stop at the first match since we always have to return the first match and we cannot assure that a later pattern do not return an earlier match so it is necessary to process the text as many times as the number of the patterns but at least we can encapsulate this logic into the library and users do not have to deal with the loops, they can just use the higher level API, which will hide the complexity and will always return the first match and tell which pattern matched. Nevertheless, some patterns may be such that at least the individual match can be done with the strategies described in The Single Pattern Case. One more future idea is that with a different API the

next match of each pattern can also be cached and at the second run it is enough to look for the next match of the pattern for which the first result was returned.

The following steps summarize the implementation described above:

Procedure 2. Processing Multiple Patterns

1. Compilation phase.
    a. Identify literal fragments in each pattern.
    b. Determine length of matches for each pattern if possible.
    c. Based on Step 1.a, Step 1.b and compilation parameters, determine the strategy to use.
    d. Run compilation of all patterns with the full algorithm.
2. Execution phase.
    - If match was compiled as line-oriented.
        a. Look for longest literal fragments, halt if not found.
        b. If match length is known for the corresponding pattern, calculate context, if not, look for line boundary.
        c. Run full algorithm on the determined context for the corresponding pattern. If matches, return match, if does not match continue at Step 2.1.a.
    - If match was not compiled as line-oriented.
        - If prefix fragment is available for all patterns.
            a. Look for the prefix literal fragments, halt if not found.
            b.
                - If match length is known for the corresponding pattern, calculate context using it.
                - Otherwise, the context lasts until the end of the input.
            c. Run full algorithm on the determined context. If matches, return match, if does not match continue with Step 2.2.1.a.
        - If prefix fragments are not available, run the full algorithm on the whole input for each pattern.
3. Clean up phase.
    a. Clean up the literal matcher's structures.
    b. Clean up the full matcher's structures.

The following example demonstrates how a single pattern is processed:

Example 2. Processing of Multiple Patterns

Let us consider the `int[13][62]_t` and the `long` patterns. The first pattern has two literal fragments: `int` and `_t`. Apart form this, it is known that the total length of a match is surely 7 characters, since the two literal fragments cover 3 and 2 characters and the bracket expressions cover one character each. The second pattern is literal and thus cannot match a newline character. So both in the literal case and in the general case the longest literal fragments — which are `int` and `long` respectively — will be used together to find a possibly matching line and then the length information is used to isolate the possibly matching context. If the first pattern were `int.*_t`, the two literal patterns to look for would be the same but it would require searching for line boundaries in the line-oriented case and in general case it could only be used to find the start of a possibly match.

And all the algorithms described above need a proper interface, which takes several patterns. I added the following two interfaces, one of which uses NUL-terminated strings and the other lets the caller specifying pattern lengths in case byte-counted buffers are used. (Although, as mentioned, this does not have any advantage in performance, since patterns have to processed character by character anyway.) Again, I added an implementation-specific prefix and the `regex_t` structure is also wrapped into `mregex_t`.

```
int regfast_mregcomp(mregex_t *preg, size_t nr,
      const char **regex, int cflags);

int regfast_mregncomp(mregex_t *preg, size_t nr,
      const char **regex, size_t *n, int cflags);
```

The execution phase call follows the same conventions and also takes the input length:

```
int regfast_mregnexec(const mregex_t *preg,
      const char *str, size_t len, size_t nmatch,
      regfast_match_t pmatch[], int eflags);
```

6.5. Some Implementation Notes

The whole library is actually a higher-level layer on top of the concentional POSIX-standard regular expression library. Nevertheless, there are some internal layers in the code, as well. For example, the Boyer-Moore or the Wu-Manber

implementations are mostly self-contained, independently usable parts, although they use the `regfast_match_t` (wrapper of the standard `regmatch_t` structure) type to report match positions. Apart from this, there is a set of other functions that build on top of this and wire together the heuristic shortcut and the invocation of the full algorithm. However, the multiple-pattern part also uses the single-pattern parts in some cases. This could have been done with strict layering but that would add too much communication overhead, which is not acceptable in an optimization project like this on so the internal architecture is probably not straightforward at the moment but I tried to mitigate this by a clear organization of functions. The interface functions are defined in `interface.c` and these call internal functions defined in `compile.c` and `match.c`. These latter two hide the complexity of initializing the heuristic search and actually executing it. The file `heuristic.c` only holds the implementation of extracting literal fragments, this is called during the compilation phase. As mentioned, the literal matching algorithms are mostly independent and are defined in `boyer-moore.c` and in `wu-manber.c`. The `hashtable.c` is also self-contained and it contains a very simple and general-purpose hashtable implementation that is used by the literal matching algorithms. And the last `convert.c` file is very simple, it just converts between normal and wide character strings.

Another challenge during the implementation was that inspired by TRE I also decided to add interface for `wchar_t *` wide character strings.

6.6. Results

I published some results in my earlier work [1] with the single-pattern heuristics, which speak for themselves. I have to add those results can still be improved with common optimization techniques, like unrolling loops and profiling the execution and trying to cut down the time-consuming parts. This is still to do. As for the multiple-pattern cases, unfortunately, as of writing this, they are not yet mature enough for a performance comparison with other solutions since the library is still under active development. Hopefully, results will be published soon at the project homepage: `https://code.google.com/p/libregfast/`

7. Conclusion

This article have explained some theoretical background about heuristic algorithms and then demonstrated it in practice. As it can be seen, the theory behind heuristics is not really complex but in practice, the actual costs are often impossible to determine and putting the pieces together may not be so trivial. The regular expression case study is full of such challenges but the results of GNU grep show that the ideas do work in practice and it is worth dedicating time to analyze the problem and think about possible heuristics that can cut down the execution time. As also mentioned, one solution may be more efficient with some series of input and less efficient with other kinds of inputs. An important conclusion is that statistical data of the input is very important in optimization. Generally fast algorithms are hard to create and handling numerous different cases is sometimes complex and may give additional overhead but usually what determines the success of a product is the performance perceived by real users on real input data so in my opinion, optimizing to common use cases is acceptable even if it has a negative impact on less frequent, strange cases. But only time will tell if my regular expression implementation becomes widespread and successful.

As for the future development, I plan to first completely finish this implementation and let it become mature before experimenting with other ideas. My future plans include experimenting with slightly alternative approaches and doing benchmarks to determine that which approaches work best in practice. Besides, I have considered a different API, that would return all the matches at once and in this way its internal implementation would not have to sequentially find matches and it may as well be threaded. Experience with other pieces of software have shown that threading may also be a good way of optimization but it needs careful planning. I consider a threaded regular expression matcher an interesting project and I may be working on it if my time permits.

Bibliography

[1] *Heuristic Pattern Matching of POSIX Regular Expressions*. Gábor Kövesdán. https://tdk.bme.hu/VIK/DownloadPaper/POSIX-regularis-kifejezesek1.

[2] *IEEE Std 1003.1-2008*. The Open Group Base Specifications Issue 7. 9. Regular Expressions. http://pubs.opengroup.org/onlinepubs/9699919799/.

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

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

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

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