NAME

PCRE2 - Perl-compatible regular expressions (revised API)

PCRE2 PERFORMANCE

Two aspects of performance are discussed below: memory usage and processing time. The way you express your pattern as a regular expression can affect both of them.

COMPILED PATTERN MEMORY USAGE

Patterns are compiled by PCRE2 into a reasonably efficient interpretive code, so that most simple patterns do not use much memory for storing the compiled version. However, there is one case where the memory usage of a compiled pattern can be unexpectedly large. If a parenthesized group has a quantifier with a minimum greater than 1 and/or a limited maximum, the whole group is repeated in the compiled code. For example, the pattern

```
(abc|def){2,4}
```

is compiled as if it were

```
(abc|def)(abc|def)((abc|def)(abc|def)?)?
```

(Technical aside: It is done this way so that backtrack points within each of the repetitions can be independently maintained.)

For regular expressions whose quantifiers use only small numbers, this is not usually a problem. However, if the numbers are large, and particularly if such repetitions are nested, the memory usage can become an embarrassment. For example, the very simple pattern

```
((ab){1,1000}c){1,3}
```

uses over 50KiB when compiled using the 8-bit library. When PCRE2 is compiled with its default internal pointer size of two bytes, the size limit on a compiled pattern is 65535 code units in the 8-bit and 16-bit libraries, and this is reached with the above pattern if the outer repetition is increased from 3 to 4. PCRE2 can be compiled to use larger internal pointers and thus handle larger compiled patterns, but it is better to try to rewrite your pattern to use less memory if you can.

One way of reducing the memory usage for such patterns is to make use of PCRE2's "subroutine" facility. Re-writing the above pattern as

```
((ab)(?2){0,999}c)(?1){0,2}
```

reduces the memory requirements to around 16KiB, and indeed it remains under 20KiB even with the outer repetition increased to 100. However, this kind of pattern is not always exactly equivalent, because any captures within subroutine calls are lost when the subroutine completes. If this is not a problem, this kind of rewriting will allow you to process patterns that PCRE2 cannot otherwise handle. The matching performance of the two different versions of the pattern are roughly the same. (This applies from release 10.30 - things were different in earlier releases.)

STACK AND HEAP USAGE AT RUN TIME

From release 10.30, the interpretive (non-JIT) version of **pcre2_match()** uses very little system stack at run time. In earlier releases recursive function calls could use a great deal of stack, and this could cause problems, but this usage has been eliminated. Backtracking positions are now explicitly remembered in memory frames controlled by the code. An initial 20KiB vector of frames is allocated on the system stack (enough for about 100 frames for small patterns), but if this is insufficient, heap memory is used. The amount of heap memory can be limited; if the limit is set to zero, only the initial stack vector is used. Rewriting

patterns to be time-efficient, as described below, may also reduce the memory requirements.

In contrast to **pcre2_match()**, **pcre2_dfa_match()** does use recursive function calls, but only for processing atomic groups, lookaround assertions, and recursion within the pattern. The original version of the code used to allocate quite large internal workspace vectors on the stack, which caused some problems for some patterns in environments with small stacks. From release 10.32 the code for **pcre2_dfa_match()** has been re-factored to use heap memory when necessary for internal workspace when recursing, though recursive function calls are still used.

The "match depth" parameter can be used to limit the depth of function recursion, and the "match heap" parameter to limit heap memory in **pcre2_dfa_match()**.

PROCESSING TIME

Certain items in regular expression patterns are processed more efficiently than others. It is more efficient to use a character class like [aeiou] than a set of single-character alternatives such as (a|e|i|o|u). In general, the simplest construction that provides the required behaviour is usually the most efficient. Jeffrey Friedl's book contains a lot of useful general discussion about optimizing regular expressions for efficient performance. This document contains a few observations about PCRE2.

Using Unicode character properties (the \p, \P, and \X escapes) is slow, because PCRE2 has to use a multi-stage table lookup whenever it needs a character's property. If you can find an alternative pattern that does not use character properties, it will probably be faster.

By default, the escape sequences \b, \d, \s, and \w, and the POSIX character classes such as [:alpha:] do not use Unicode properties, partly for backwards compatibility, and partly for performance reasons. However, you can set the PCRE2_UCP option or start the pattern with (*UCP) if you want Unicode character properties to be used. This can double the matching time for items such as \d, when matched with **pcre2_match()**; the performance loss is less with a DFA matching function, and in both cases there is not much difference for \b.

When a pattern begins with .* not in atomic parentheses, nor in parentheses that are the subject of a backreference, and the PCRE2_DOTALL option is set, the pattern is implicitly anchored by PCRE2, since it can match only at the start of a subject string. If the pattern has multiple top-level branches, they must all be anchorable. The optimization can be disabled by the PCRE2_NO_DOTSTAR_ANCHOR option, and is automatically disabled if the pattern contains (*PRUNE) or (*SKIP).

If PCRE2_DOTALL is not set, PCRE2 cannot make this optimization, because the dot metacharacter does not then match a newline, and if the subject string contains newlines, the pattern may match from the character immediately following one of them instead of from the very start. For example, the pattern

```
.*second
```

matches the subject "first\nand second" (where \n stands for a newline character), with the match starting at the seventh character. In order to do this, PCRE2 has to retry the match starting after every newline in the subject.

If you are using such a pattern with subject strings that do not contain newlines, the best performance is obtained by setting PCRE2_DOTALL, or starting the pattern with ^.* or ^.*? to indicate explicit anchoring. That saves PCRE2 from having to scan along the subject looking for a newline to restart at.

Beware of patterns that contain nested indefinite repeats. These can take a long time to run when applied to a string that does not match. Consider the pattern fragment

```
^(a+)*
```

This can match "aaaa" in 16 different ways, and this number increases very rapidly as the string gets longer. (The * repeat can match 0, 1, 2, 3, or 4 times, and for each of those cases other than 0 or 4, the + repeats can match different numbers of times.) When the remainder of the pattern is such that the entire match is going to fail, PCRE2 has in principle to try every possible variation, and this can take an extremely long

time, even for relatively short strings.

An optimization catches some of the more simple cases such as

```
(a+)*b
```

where a literal character follows. Before embarking on the standard matching procedure, PCRE2 checks that there is a "b" later in the subject string, and if there is not, it fails the match immediately. However, when there is no following literal this optimization cannot be used. You can see the difference by comparing the behaviour of

```
(a+)*\d
```

with the pattern above. The former gives a failure almost instantly when applied to a whole line of "a" characters, whereas the latter takes an appreciable time with strings longer than about 20 characters.

In many cases, the solution to this kind of performance issue is to use an atomic group or a possessive quantifier. This can often reduce memory requirements as well. As another example, consider this pattern:

```
([^<]|<(?!inet))+
```

It matches from wherever it starts until it encounters "<inet" or the end of the data, and is the kind of pattern that might be used when processing an XML file. Each iteration of the outer parentheses matches either one character that is not "<" or a "<" that is not followed by "inet". However, each time a parenthesis is processed, a backtracking position is passed, so this formulation uses a memory frame for each matched character. For a long string, a lot of memory is required. Consider now this rewritten pattern, which matches exactly the same strings:

```
([^<]++|<(?!inet))+
```

This runs much faster, because sequences of characters that do not contain "<" are "swallowed" in one item inside the parentheses, and a possessive quantifier is used to stop any backtracking into the runs of non-"<" characters. This version also uses a lot less memory because entry to a new set of parentheses happens only when a "<" character that is not followed by "inet" is encountered (and we assume this is relatively rare).

This example shows that one way of optimizing performance when matching long subject strings is to write repeated parenthesized subpatterns to match more than one character whenever possible.

SETTING RESOURCE LIMITS

You can set limits on the amount of processing that takes place when matching, and on the amount of heap memory that is used. The default values of the limits are very large, and unlikely ever to operate. They can be changed when PCRE2 is built, and they can also be set when **pcre2_match()** or **pcre2_dfa_match()** is called. For details of these interfaces, see the **pcre2build** documentation and the section entitled "The match context" in the **pcre2api** documentation.

The **pcre2test** test program has a modifier called "find_limits" which, if applied to a subject line, causes it to find the smallest limits that allow a pattern to match. This is done by repeatedly matching with different limits.

AUTHOR

Philip Hazel
University Computing Service
Cambridge, England.

REVISION

Last updated: 03 February 2019