Hashtable/dictionary/map lookup with regular expressions

Asked 16 years, 1 month ago Modified 8 years, 3 months ago Viewed 9k times

21

I'm trying to figure out if there's a reasonably efficient way to perform a lookup in a dictionary (or a hash, or a map, or whatever your favorite language calls it) where the keys are regular expressions and strings are looked up against the set of keys. For example (in Python syntax):

```
>>> regex_dict = { re.compile(r'foo.') : 12, re.compile(r'^FileN.*$') : 35 }
>>> regex_dict['food']
12
>>> regex_dict['foot in my mouth']
12
>>> regex_dict['FileNotFoundException: file.x does not exist']
35
```

(Obviously the above example won't work as written in Python, but that's the sort of thing I'd like to be able to do.)

I can think of a naive way to implement this, in which I iterate over all of the keys in the dictionary and try to match the passed in string against them, but then I lose the O(1) lookup time of a hash map and instead have O(n), where n is the number of keys in my dictionary. This is potentially a big deal, as I expect this dictionary to grow very large, and I will need to search it over and over again (actually I'll need to iterate over it for every line I read in a text file, and the files can be hundreds of megabytes in size).

Is there a way to accomplish this, without resorting to O(n) efficiency?

Alternatively, if you know of a way to accomplish this sort of a lookup in a database, that would be great, too.

(Any programming language is fine -- I'm using Python, but I'm more interested in the data structures and algorithms here.)

Someone pointed out that more than one match is possible, and that's absolutely correct. Ideally in this situation I'd like to return a list or tuple containing all of the matches. I'd settle for the first match, though.

I can't see O(1) being possible in that scenario; I'd settle for anything less than O(n), though. Also, the underlying data structure could be anything, but the basic behavior

I'd like is what I've written above: lookup a string, and return the value(s) that match the regular expression keys.

python regex dictionary hash

Share

Improve this question

Follow

edited Sep 1, 2016 at 15:01

Martijn Pieters

1.1m ● 319 ● 4.2k ● 3.4k

asked Nov 3, 2008 at 21:39

Jeff

15.9k ● 15 ● 53 ● 60

1 I'm guessing, but I think this is impossible because there are an infinite number of regular expressions that match your key, and an infinite number of keys that match your regular expression. – Paul Tomblin Nov 3, 2008 at 21:43

"this is impossible because there are an infinite number of regular expressions that match your key" -- it's not guessing. That's a consequence of the definition of a regular expression. – S.Lott Nov 3, 2008 at 22:27

@jeff Do you have an example implementation for the accepted solution? It will be helpful, Thanks! – 18bytes Jun 11, 2012 at 8:39

19 Answers

Sorted by: Highest score (default)

4

What you want to do is very similar to what is supported by xrdb. They only support a fairly minimal notion of globbing however.

Internally you can implement a larger family of regular languages than theirs by storing your regular expressions as a character trie.

- single characters just become trie nodes.
- .'s become wildcard insertions covering all children of the current trie node.
- *'s become back links in the trie to node at the start of the previous item.
- [a-z] ranges insert the same subsequent child nodes repeatedly under each of the characters in the range. With care, while inserts/updates may be somewhat expensive the search can be linear in the size of the string. With some placeholder stuff the common combinatorial explosion cases can be kept under control.
- (foo)|(bar) nodes become multiple insertions

This doesn't handle regexes that occur at arbitrary points in the string, but that can be modeled by wrapping your regex with .* on either side.

Perl has a couple of Text::Trie -like modules you can raid for ideas. (Heck I think I even wrote one of them way back when)

Share
Improve this answer
Follow

edited Nov 6, 2008 at 14:28

answered Nov 5, 2008 at 20:52

Edward Kmett
30k ● 7 ● 87 ● 107

any implementation available? – bill Jun 16, 2009 at 10:16

@bill there are a few on blogs, as well my question on Code Review. You can also look at what I ended up with; it's only a basic Trie but it might help you get going. – Adam Jun 2, 2013 at 12:00

4

This is not possible to do with a regular hash table in any language. You'll either have to iterate through the entire keyset, attempting to match the key to your regex, or use a different data structure.

You should choose a data structure that is appropriate to the problem you're trying to solve. If you have to match against any arbitrary regular expression, I don't know of a good solution. If the class of regular expressions you'll be using is more restrictive, you might be able to use a data structure such as a trie or suffix tree.

Share Improve this answer Follow

answered Nov 3, 2008 at 21:44

Adam Rosenfield
399k ● 101 ● 522 ● 597

4

In the general case, what you need is a lexer generator. It takes a bunch of regular expressions and compiles them into a recognizer. "lex" will work if you are using C. I have never used a lexer generator in Python, but there seem to be a few to choose from. Google shows PLY, PyGgy and PyLexer.

If the regular expressions all resemble each other in some way, then you may be able to take some shortcuts. We would need to know more about the ultimate problem that you are trying to solve in order to come up with any suggestions. Can you share some sample regular expressions and some sample data?

Also, how many regular expressions are you dealing with here? Are you sure that the naive approach *won't* work? As Rob Pike once said, "Fancy algorithms are slow when n is small, and n is usually small." Unless you have thousands of regular expressions, and thousands of things to match against them, and this is an interactive application where a user is waiting for you, you may be best off just doing it the easy way and looping through the regular expressions.

Share Improve this answer Follow

answered Nov 3, 2008 at 21:53

Andru Luvisi

25.3k ● 6 ● 55 ● 66

> We anticipate having thousands of regexes soon. In all cases we have to match those regexes repeatedly, typically thousands of times per user operation. It may be okay to go with the naive solution and then rewrite the algorithm when performance degrades, as this does not need to run interactively. – Jeff Nov 3, 2008 at 22:08

4

This is definitely possible, as long as you're using 'real' regular expressions. A textbook regular expression is something that can be recognized by a deterministic finite state machine, which primarily means you can't have back-references in there.

There's a property of regular languages that "the union of two regular languages is regular", meaning that you can recognize an arbitrary number of regular expressions at once with a single state machine. The state machine runs in O(1) time with respect to the number of expressions (it runs in O(n) time with respect to the length of the input string, but hash tables do too).

Once the state machine completes you'll know which expressions matched, and from there it's easy to look up values in O(1) time.

Share Improve this answer Follow

answered Nov 4, 2008 at 6:30

Trevor Strohman

> I vaguely recall reading something about this in Higher Order Perl but can't find location at the moment. Anyone else remember? – Michael Carman Nov 4, 2008 at 17:08

3

What happens if you have a dictionary such as

```
regex_dict = { re.compile("foo.*"): 5, re.compile("f.*"): 6 }
```

In this case `regex_dict["food"]` could legitimately return either 5 or 6.

Even ignoring that problem, there's probably no way to do this efficiently with the regex module. Instead, what you'd need is an internal directed graph or tree structure.

Share Improve this answer Follow

answered Nov 3, 2008 at 21:46

Eli Courtwright

193k ● 68 ● 218 ● 257

What about the following:

```
class redict(dict):
def __init__(self, d):
    dict.__init__(self, d)

def __getitem__(self, regex):
    r = re.compile(regex)
    mkeys = filter(r.match, self.keys())
    for i in mkeys:
        yield dict.__getitem__(self, i)
```

It's basically a subclass of the dict type in Python. With this you can supply a regular expression as a key, and the values of all keys that match this regex are returned in an iterable fashion using yield.

With this you can do the following:

```
>>> keys = ["a", "b", "c", "ab", "ce", "de"]
>>> vals = range(0,len(keys))
>>> red = redict(zip(keys, vals))
>>> for i in red[r"^.e$"]:
...     print i
...
5
4
>>>
```

Share Improve this answer Follow

answered May 3, 2009 at 1:18

Florian Nigsch

> Functionally, this is fine, but performance-wise, it's still O(n) because filter() is O(n) (well, actually it's worse than O(n) because we have to match the regular expression against each key, which has a non-constant cost, but I assume that will be part of any solution). I'd like to search the keys in a better-than-O(n) way, if possible. Others suggested data structures such as tries that might make this possible. – Jeff May 4, 2009 at 19:44

Here's an efficient way to do it by combining the keys into a single compiled regexp, and so not requiring any looping over key patterns. It abuses the `lastindex` to find out which key matched. (It's a shame regexp libraries don't let you tag the terminal state of the DFA that a regexp is compiled to, or this would be less of a hack.)

The expression is compiled once, and will produce a fast matcher that doesn't have to search sequentially. Common prefixes are compiled together in the DFA, so each

character in the key is matched once, not many times, unlike some of the other suggested solutions. You're effectively compiling a mini lexer for your keyspace.

This map isn't extensible (can't define new keys) without recompiling the regexp, but it can be handy for some situations.

```
# Regular expression map
# Abuses match.lastindex to figure out which key was matched
# (i.e. to emulate extracting the terminal state of the DFA of the regexp
engine)
# Mostly for amusement.
# Richard Brooksby, Ravenbrook Limited, 2013-06-01

import re

class ReMap(object):

    def __init__(self, items):
        if not items:
            items = [(r'epsilon^', None)] # Match nothing
        key_patterns = []
        self.lookup = {}
        index = 1
        for key, value in items:
            # Ensure there are no capturing parens in the key, because
            # that would mess up match.lastindex
            key_patterns.append('(' + re.sub(r'\((?!\?:)', '(?:', key) + ')')
            self.lookup[index] = value
            index += 1
        self.keys_re = re.compile('|'.join(key_patterns))

    def __getitem__(self, key):
        m = self.keys_re.match(key)
        if m:
            return self.lookup[m.lastindex]
        raise KeyError(key)

if __name__ == '__main__':
    remap = ReMap([(r'foo.', 12), (r'FileN.*', 35)])
    print remap['food']
    print remap['foot in my mouth']
    print remap['FileNotFoundException: file.x does not exist']
```

Share

Improve this answer

Follow

edited Jun 29, 2013 at 8:10

answered Jun 1, 2013 at 18:20

rptb1

480 ● 4 ● 11

2

There is a Perl module that does just this Tie::Hash::Regex.

```
use Tie::Hash::Regex;
my %h;

tie %h, 'Tie::Hash::Regex';
```

```
$h{key}   = 'value';
$h{key2}  = 'another value';
$h{stuff} = 'something else';

print $h{key};  # prints 'value'
print $h{2};    # prints 'another value'
print $h{'^s'}; # prints 'something else'

print tied(%h)->FETCH(k); # prints 'value' and 'another value'

delete $h{k};   # deletes $h{key} and $h{key2};
```

Share Improve this answer Follow

answered Nov 4, 2008 at 4:31

Brad Gilbert

34.1k ● 11 ● 79 ● 130

1 Yes, I know about this module, and the described behavior is exactly what I want, but I peeked at the source code for this and it's really just iterating over the keys for each lookup. So it's really just an O(n) solution, albeit a convenient one. – Jeff Nov 4, 2008 at 16:21

2

@rptb1 you don't have to avoid capturing groups, because you can use re.groups to count them. Like this:

```
# Regular expression map
# Abuses match.lastindex to figure out which key was matched
# (i.e. to emulate extracting the terminal state of the DFA of the regexp
engine)
# Mostly for amusement.
# Richard Brooksby, Ravenbrook Limited, 2013-06-01

import re

class ReMap(object):
    def __init__(self, items):
        if not items:
            items = [(r'epsilon^', None)] # Match nothing
        self.re = re.compile('|'.join('('+k+')' for (k,v) in items))
        self.lookup = {}
        index = 1
        for key, value in items:
            self.lookup[index] = value
            index += re.compile(key).groups + 1

    def __getitem__(self, key):
        m = self.re.match(key)
        if m:
            return self.lookup[m.lastindex]
        raise KeyError(key)

def test():
    remap = ReMap([(r'foo.', 12),
                   (r'.*([0-9]+)', 99),
                   (r'FileN.*', 35),
                   ])
```

```
    print remap['food']
    print remap['foot in my mouth']
    print remap['FileNotFoundException: file.x does not exist']
    print remap['there were 99 trombones']
    print remap['food costs $18']
    print remap['bar']

if __name__ == '__main__':
    test()
```

Sadly very few RE engines actually compile the regexps down to machine code, although it's not especially hard to do. I suspect there's an order of magnitude performance improvement waiting for someone to make a really good RE JIT library.

Share Improve this answer Follow

answered Jun 1, 2013 at 23:32

Nick Barnes
191 ● 8

I think it's probably better to eliminate the capturing parens when building the regexp, for two reasons: 1. the matcher doens't have to store useless info, and 2. the set of keys into the dict remains dense, and I'm hoping that's optimised. – rptb1 Jun 3, 2013 at 20:41

@rptb1: "I'm hoping that's optimised" — dictionaries in Python have no special treatment of dense sets of small integer keys. See `dictobject.c` . – Gareth Rees Oct 9, 2013 at 11:08

1

As other respondents have pointed out, it's not possible to do this with a hash table in constant time.

One approximation that might help is to use a technique called "n-grams". Create an inverted index from n-character chunks of a word to the entire word. When given a pattern, split it into n-character chunks, and use the index to compute a scored list of matching words.

Even if you can't accept an approximation, in most cases this would still provide an accurate filtering mechanism so that you don't have to apply the regex to every key.

Share Improve this answer Follow

answered Nov 3, 2008 at 21:54

erickson
269k ● 59 ● 401 ● 497

1

A special case of this problem came up in the 70s AI languages oriented around deductive databases. The keys in these databases could be patterns with variables -- like regular expressions without the * or | operators. They tended to use fancy extensions of trie structures for indexes. See krep*.lisp in Norvig's Paradigms of AI Programming for the general idea.

Share Improve this answer Follow

answered Nov 4, 2008 at 5:02

Darius Bacon

15.1k ● 6 ● 56 ● 53

1

If you have a small set of possible inputs, you can cache the matches as they appear in a second dict and get O(1) for the cached values.

If the set of possible inputs is too big to cache but not infinite, either, you can just keep the last N matches in the cache (check Google for "LRU maps" - least recently used).

If you can't do this, you can try to chop down the number of regexps you have to try by checking a prefix or somesuch.

Share Improve this answer Follow

answered Nov 4, 2008 at 12:39

Aaron Digulla

328k ● 110 ● 622 ● 834

1

I created this exact data structure for a project once. I implemented it naively, as you suggested. I did make two immensely helpful optimizations, which may or may not be feasible for you, depending on the size of your data:

- Memoizing the hash lookups
- Pre-seeding the the memoization table (not sure what to call this... warming up the cache?)

To avoid the problem of multiple keys matching the input, I gave each regex key a priority and the highest priority was used.

Share Improve this answer Follow

answered Nov 6, 2008 at 5:55

mccutchen

0

The fundamental assumption is flawed, I think. you can't map hashes to regular expressions.

Share Improve this answer Follow

answered Nov 3, 2008 at 21:44

Jimmy

91.3k ● 18 ● 121 ● 139

You can, at least in Python. It's not very useful, though (for me at least), as they'll only match the same regex object. – Jeff Nov 3, 2008 at 22:11

0

I don't think it's even theoretically possible. What happens if someone passes in a string that matches more than 1 regular expression.

For example, what would happen if someone did:

```
>>> regex_dict['FileNfoo']
```

How can something like that possibly be O(1)?

Share Improve this answer Follow

answered Nov 3, 2008 at 21:44

Moe

29.6k ● 10 ● 53 ● 69

0

It *may* be possible to get the regex compiler to do most of the work for you by concatenating the search expressions into one big regexp, separated by "|". A clever regex compiler might search for commonalities in the alternatives in such a case, and devise a more efficient search strategy than simply checking each one in turn. But I have no idea whether there are compilers which will do that.

Share Improve this answer Follow

answered Nov 4, 2008 at 0:13

Alex Coventry

70.7k ● 5 ● 39 ● 40

0

It really depends on what these regexes look like. If you don't have a lot regexes that will match almost anything like '`.*`' or '`\d+`', and instead you have regexes that *contains* mostly words and phrases or any fixed patterns longer than 4 characters (e.g.'`a*b*c`' in `^\d+a\*b\*c:\s+\w+`) , as in your examples. You can do this common trick that scales well to millions of regexes:

Build a inverted index for the regexes (rabin-karp-hash('fixed pattern') -> list of regexes containing 'fixed pattern'). Then at matching time, using Rabin-Karp hashing to compute sliding hashes and look up the inverted index, advancing one character at a time. You now have O(1) look-up for inverted-index non-matches and a reasonable O(k) time for matches, k is the average length of the lists of regexes in the inverted index. k can be quite small (less than 10) for many applications. The quality (false positive means bigger k, false negative means missed matches) of the inverted index depends on how well the indexer understands the regex syntax. If the regexes are

generated by human experts, they can provide hints for contained fixed patterns as well.

Share Improve this answer Follow

answered Nov 4, 2008 at 1:57

ididak
5,858 ● 1 ● 22 ● 21

0

Ok, I have a very similar requirements, I have a lot of lines of different syntax, basically remark lines and lines with some codes for to use in a process of smart-card format, also, descriptor lines of keys and secret codes, in every case, I think that the "model" pattern/action is the beast approach for to recognize and to process a lot of lines.
I'm using `C++/CLI` for to develop my assembly named `LanguageProcessor.dll`, the core of this library is a lex_rule class that basically contains :

- a Regex member
- an event member

The constructor loads the regex string and call the necessary codes for to build the event on the fly using `DynamicMethod`, `Emit` and `Reflexion`... also into the assembly exists other class like meta and object that constructs ans instantiates the objects by the simple names of the publisher and the receiver class, receiver class provides the action handlers for each rule matched.

Late, I have a class named `fasterlex_engine` that build a Dictionary`<Regex, action_delegate>` that load the definitions from an array for to run.

The project is in advanced point but I'm still building, today. I will try to enhance the performance of running surrounding the sequential access to every pair foreach line input, thru using some mechanism of lookup the dictionary directly using the regexp like:

```
map_rule[gcnew Regex("[a-zA-Z]")];
```

Here, some of segments of my code:

```
public ref class lex_rule: ILexRule
{
private:
    Exception           ^m_exception;
    Regex               ^m_pattern;

    //BACKSTORAGE delegates, esto me lo aprendi asiendo la huella.net de m*e*da
JEJE
    yy_lexical_action   ^m_yy_lexical_action;
```

```
    yy_user_action      ^m_yy_user_action;

public:
    virtual property    String ^short_id;
private:
    void init(String ^_short_id, String ^well_formed_regex);
public:

    lex_rule();
    lex_rule(String ^_short_id,String ^well_formed_regex);
    virtual event    yy_lexical_action ^YY_RULE_MATCHED
    {
        virtual void add(yy_lexical_action ^_delegateHandle)
        {
            if(nullptr==m_yy_lexical_action)
                m_yy_lexical_action=_delegateHandle;
        }
        virtual void remove(yy_lexical_action ^)
        {
            m_yy_lexical_action=nullptr;
        }

        virtual long raise(String ^id_rule, String ^input_string, String
^match_string, int index)
        {
            long lReturn=-1L;
            if(m_yy_lexical_action)
                lReturn=m_yy_lexical_action(id_rule,input_string, match_string,
index);
            return lReturn;
        }
    }
};
```

Now the fasterlex_engine class that execute a lot of pattern/action pair:

```
public ref class fasterlex_engine
{
private:
    Dictionary<String^,ILexRule^> ^m_map_rules;
public:
    fasterlex_engine();
    fasterlex_engine(array<String ^,2>^defs);
    Dictionary<String ^,Exception ^> ^load_definitions(array<String ^,2>
^defs);
    void run();
};
```

AND FOR TO DECORATE THIS TOPIC..some code of my cpp file:

this code creates a constructor invoker by parameter sign

```
inline Exception ^object::builder(ConstructorInfo ^target, array<Type^> ^args)
{
try
{
    DynamicMethod ^dm=gcnew DynamicMethod(
```

```
        "dyna_method_by_totem_motorist",
        Object::typeid,
        args,
        target->DeclaringType);
    ILGenerator ^il=dm->GetILGenerator();
    il->Emit(OpCodes::Ldarg_0);
    il->Emit(OpCodes::Call,Object::typeid->GetConstructor(Type::EmptyTypes));
//invoca a constructor base
    il->Emit(OpCodes::Ldarg_0);
    il->Emit(OpCodes::Ldarg_1);
    il->Emit(OpCodes::Newobj, target); //NewObj crea el objeto e invoca al
constructor definido en target
    il->Emit(OpCodes::Ret);
    method_handler=(method_invoker ^) dm-
>CreateDelegate(method_invoker::typeid);
}
catch (Exception ^e)
{
    return  e;
}
return nullptr;
```

}

This code attach an any handler function (static or not) for to deal with a callback raised by matching of a input string

```
Delegate ^connection_point::hook(String ^receiver_namespace,String
^receiver_class_name, String ^handler_name)
{
Delegate ^d=nullptr;
if(connection_point::waitfor_hook<=m_state) // si es 0,1,2 o mas => intenta
hookear
{
    try
    {
        Type ^tmp=meta::_class(receiver_namespace+"."+receiver_class_name);
        m_handler=tmp->GetMethod(handler_name);
        m_receiver_object=Activator::CreateInstance(tmp,false);

        d=m_handler->IsStatic?
            Delegate::CreateDelegate(m_tdelegate,m_handler):
            Delegate::CreateDelegate(m_tdelegate,m_receiver_object,m_handler);

        m_add_handler=m_connection_point->GetAddMethod();
        array<Object^> ^add_handler_args={d};
        m_add_handler->Invoke(m_publisher_object, add_handler_args);
        ++m_state;
        m_exception_flag=false;
    }
    catch(Exception ^e)
    {
        m_exception_flag=true;
        throw gcnew Exception(e->ToString()) ;
    }
}
return d;
}
```

finally the code that call the lexer engine:

```
array<String ^,2> ^defs=gcnew array<String^,2>  {/*   shortID    pattern
namespc    clase           fun*/
                                                    {"LETRAS",  "[A-Za-z]+"
,"prueba",  "manejador",    "procesa_directriz"},
                                                    {"INTS",    "[0-9]+"
,"prueba",  "manejador",    "procesa_comentario"},
                                                    {"REM",     "--[^\\n]*"
,"prueba",  "manejador",    "nullptr"}
                                                }; //[3,5]

//USO EL IDENTIFICADOR ESPECIAL "nullptr" para que el sistema asigne el proceso
del evento a un default que realice nada
fasterlex_engine ^lex=gcnew fasterlex_engine();
Dictionary<String ^,Exception ^> ^map_error_list=lex->load_definitions(defs);
lex->run();
```

Share

Improve this answer

Follow

edited Dec 16, 2011 at 8:47

JMax

26.6k ● 12 ● 73 ● 89

answered Apr 29, 2011 at 17:50

TOTEM_MOTORIST

1 ● 1

0

The problem has nothing to do with regular expressions - you'd have the same problem with a dictionary with keys as functions of lambdas. So the problem you face is figuring is there a way of classifying your functions to figure which will return true or not and that isn't a search problem because f(x) is not known in general before hand.

Distributed programming or caching answer sets assuming there are common values of x may help.

-- DM

Share Improve this answer Follow

answered Apr 17, 2012 at 11:51

DangerMouse

724 ● 8 ● 20