pyahocorasick Documentation

Release 1.1.0

Wojciech Muła

July 29, 2016

Contents

pyahocorasick is a fast and memory efficient library for exact or approximate multi-pattern string search meaning that you can find multiple key strings occurences at once in some input text. It is implemented in C and tested on Python 2.7 and 3.4+. It works on Linux, Mac and Windows. The *license* is BSD-3-clause. Some utilities, such tests and some Python are dedicated to the Public Domain.

CHAPTER 1

Download

You can fetch **pyahocorasick** from GitHub or Pypi.

CHAPTER 2

Doc

http://pyahocorasick.readthedocs.io/en/

CHAPTER 3

Quick start

Install:

```
pip install pyahocorasick
```

Create an Automaton:

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
```

Add some strings and their associated value to this trie. Here we associate a tuple of (insertion index, original string) as a value to each key string we add to the trie:

```
>>> for idx, key in enumerate('he her hers she'.split()):
...     A.add_word(key, (idx, key))
```

Check if some string exists in the trie:

```
>>> 'he' in A
True
>>> 'HER' in A
False
```

Play with the `get()` dict-like method:

```
>>> A.get('he')
(0, 'he')
>>> A.get('she')
(3, 'she')
>>> A.get('cat', '<not exists>')
'<not exists>'
>>> A.get('dog')
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
KeyError
>>>
```

Convert the trie to an Aho-Corasick automaton:

```
>>> A.make_automaton()
```

Then search all occurrences of the keys (the needles) in an input string (our haystack). Print the results and check that they are correct:

```
>>> haystack = '_hershe_'
>>> for end_index, (insert_order, original_value) in A.iter(haystack):
...     print((end_index, (insert_order, original_value)))
...     start_index = end_index - len(original_value)
...     assert haystack[start_index:end_index] == original_value
(2, (0, 'he'))
(3, (1, 'her'))
(4, (2, 'hers'))
(6, (3, 'she'))
(6, (0, 'he'))
```

See also:

- *API overview* for more options and the API documentation.
- *More Examples* for more examples.
- *Build and install* for more details on installation.
- *Tests* to run unit tests.
- *Support* for help and bugs.
- and *Authors* and *License* .

Contents

CHAPTER 4

Introduction

With an **Aho-Corasick automaton** you can efficiently search all occurences of multiple strings (the needles) in an input string (the haystack) making a single pass over the input string. With pyahocorasick you can eventually build large automatons and pickle them and reuse them over and over as an index structure for fast multi pattern string matching.

One of the advantage of an Aho-Corasick automaton is that the typical worst-case and best-case **runtimes** are about the same and depends primarily on the size of the input string and secondarily on the number of matches returned. While this may not be the fastest string search algorithm in all cases, it can search for multiple strings at once and its runtime guarantees make it rather unique. Because pyahocorasick is based on a Trie, it stores redundant keys prefixes only once using memory efficiently.

A drawback is that it needs to be constructed and "finalized" ahead of time before you can search strings. In several applicatiosn where you search several pre-defined "needles" in variable "haystacks" this is actually an advantage.

Aho-Corasick automatons are commonly used for fast multi-pattern matching in intrusion detection systems (such as snort), anti-viruses and many other applications that need fast matching against a pre-defined set of string keys.

Internally an Aho-Corasick automaton is typically based on a Trie with extra data for failure links and an implementation of the Aho-Corasick search procedure.

Behind the scenes the **pyahocorasick** Python library implements these two data structures: a Trie and an Aho-Corasick string matching automaton. Both are exposed through the *Automaton* class.

In addition to Trie-like and Aho-Corasick methods and data structures, **pyahocorasick** also implements dict-like methods: The pyahocorasick **Automaton** is a **Trie** a dict-like structure indexed by string keys each associated with a value object. You can use this to retrieve an associated value in a time proportional to a string key length.

pyahocorasick is available in two flavors:

- a CPython **C-based extension**, compatible with Python 2 and 3.
- a simpler pure Python module, compatible with Python 2 and 3. This is only available in the source repository (not on Pypi) under the py/ directory and it has a slightly different API.

CHAPTER 5

Some background about pyahocorasick internals

- I wrote this article about different trie representations — These are experiments I made while creating this module.

CHAPTER 6

Other Aho-Corarisk implementations for Python you can consider

While **pyahocorasick** tries to be the finest and fastest Aho Corasick library for Python you may consider these other libraries:

- noaho by Jeff Donner — Written in C. Does not return overlapping matches. Does not compile on Windows (July 2016). No support for the pickle protocol.
- acora by Stefan Behnel — Written in Cython. Large automaton may take a long time to build (July 2016) No support for a dict-like protocol to associate a value to a string key.
- ahocorasick by Danny Yoo — seems unmaintained (last update in 2005) and is GPL-licensed. Written in C.

CHAPTER 7

API overview

This is the API for the C **ahocorasick** module. The pure Python module has a slightly different interface.

7.1 Module

The module `ahocorasick` contains a few constants and the main `Automaton` class.

7.1.1 Constants

- `ahocorasick.unicode` — see *Unicode and bytes*
- `ahocorasick.STORE_ANY`, `ahocorasick.STORE_INTS`, `ahocorasick.STORE_LENGTH` — see *Constructor*
- `ahocorasick.EMPTY`, `ahocorasick.TRIE`, `ahocorasick.AHOCORASICK` — see *Attributes*
- `ahocorasick.MATCH_EXACT_LENGTH`, `ahocorasick.MATCH_AT_MOST_PREFIX`, `ahocorasick.MATCH_AT_LEAST_PREFIX` — see description of the *keys* method

7.1.2 Automaton class

Note: `Automaton` instances are pickable (It implements the `__reduce__() magic method`).

Constructor

Automaton(value_type) Create a new empty Automaton. value_type is optional and one of these constants:

`ahocorasick.STORE_ANY` Any Python object can be stored as a value associated to a string key (default).

`ahocorasick.STORE_LENGTH` The length of the a string key is automatically added to the trie as the associated value for a string key.

`ahocorasick.STORE_INTS` A 32-bit integer is used for the associated values.

Trie methods

The Automaton class has the following trie methods:

add_word(key, [value]) => bool Add a `key` string to the dict-like trie and associate this key with a `value`. `value` is optional or mandatory depending how the Automaton instance was created. Return True if the `word` key is inserted and did not exists in the trie or False otherwise.

If the Automaton was created without argument (the default) as `Automaton()` or with `Automaton(ahocorasik.STORE_ANY)` then the `value` is required and can be any Python object.

If the Automaton was created with `Automaton(ahocorasik.STORE_LENGTH)` then associating a `value` is not allowed — `len(word)` is saved automatically as a value instead.

If the Automaton was created with `Automaton(ahocorasik.STORE_INTS)` then the `value``is optional. If provided it must be an integer, otherwise it defaults to ``len(automaton)` which is therefore the order index in which keys are added to the trie.

Calling ‘‘add_word‘‘ invalidates all iterators only if the new key did not exist in the trie so far (i.e. the method returned True).

clear() => None Remove all keys from the trie.

This method invalidates all iterators.

exists(key) => bool or key in ... Return True if the key is present in the trie. Same as using the ‘in’ keyword.

match(key) => bool Return True if there is a prefix (or key) equal to `key` present in the trie. For example if the key ‘example’ has been added to the trie, then calling `match('e')`, `match('ex')`, ..., `match('exampl')`, or `match('example')` all return True. But `exists()` is True only when calling `exists('example')`

longest_prefix(string) => integer Return the length of the longest prefix of string that exists in the trie.

Dictionary-like methods

A pyahocorasick trie behaves more or less like a Python dictionary and implements a subset of dict-like methods.

get(key[, default]) Return the value associated with the `key` string. Raise a `KeyError` exception if the key is not in the trie and no default is provided. Return the optional `default` value if provided and the key is not in the trie.

keys([prefix, [wildcard, [how]]]) => yield strings Return an iterator on keys.

If the optional `prefix` string is provided, then only keys starting with this prefix are yielded.

If the optional `wildcard` is provided as a single character string, then the `prefix` is treated as a simple pattern using this `wildcard` as a wildcard.

The optional `how` argument is used to control how strings are matched using one of these possible values:

ahocorasick.MATCH_EXACT_LENGTH [default] Yield matches that have the same exact length as the prefix length.

ahocorasick.MATCH_AT_LEAST_PREFIX Yield matches that have a length greater or equal to the prefix length.

ahocorasick.MATCH_AT_MOST_PREFIX Yield matches that have a length lesser or equal to the prefix length.

See *Example 2* and the section below.

values([prefix, [wildcard, [how]]]) => yield object Return an iterator on values associated with each keys. Keys are are matched optionally to the prefix using the same logic and arguments as in the `keys` method.

items([prefix, [wildcard, [how]]]) => yield tuple (string, object) Return an iterator on tuples of (key, value). Keys are are matched optionally to the prefix using the same logic and arguments as in the `keys` method.

len() Return the number of distinct keys added to the trie.

Wildcards

Methods `keys`, `values` and `items` can be called with an optional **wildcard**. A wildcard character is equivalent to a question mark used in glob patterns (?) or a dot from regular expressions (.). You can use any character you like as a wildcard.

Note that it is not possible to escape a wildcard to match it exactly — You need instead to select another wildcard character, not present in the provided prefix. For example:

```
automaton.keys("hi?", "?")  # would match "him", "his"
automaton.keys("XX?", "X")  # would match "me?", "he?" or "it?"
```

Aho-Corasick methods

make_automaton() Finalize and create the Aho-Corasick automaton based on the keys already added to the trie. This does not require additional memory. After successful the `Automaton.kind` attribute is set to `ahocorasick.AHOCORASICK`.

This method invalidates all iterators.

iter(string, [start, [end]]) Perform the Aho-Corasick search procedure using the provided input `string`. Return an iterator of tuples (end_index, value) for keys found in string where:

- `end_index` is the end index in the input string where a trie key string was found.
- `value` is the value associated with the found key string.

The start and end optional arguments can be used to limit the search to an input string slice as in string[start:end].

find_all(string, callback, [start, [end]]) Perform the Aho-Corasick search procedure using the provided input `string` and iterate over the matching tuples (end_index, value) for keys found in `string`. Invoke the `callback` callable with each matching tuple. The `callback` callable must accept two positional arguments:

- `end_index` is the end index in the input string where a trie key string was found.
- `value` is the value associated with the found key string.

The `start` and `end` optional arguments can be used to limit the search to a `string` slice as in `string[start:end]`.

Note that the `find_all` method is equivalent to:

```
    def find_all(self, string, callback):
        for end_index, value in self.iter(string):
            callback(end_index, value)
```

Attributes

kind [readonly] Return the state of the `Automaton` instance. This is read only and is maintained internally. Note that some methods are not available when automaton kind is `ahocorasick.EMPTY` or

ahocorasick.TRIE. They will raise an exception if called when not available. Testing this property before calling these methods may be a better (faster or more elegant) than a try/except block but you can use both approaches.

Possible kind values are:

ahocorasick.EMPTY The trie is empty.

ahocorasick.TRIE Some words have been added but the Automaton has not been constructed yet: methods related to Aho-Corasick such as find_all or iter will not work.

ahocorasick.AHOCORASICK The Aho-Corasick automaton has been constructed; all methods are available.

store [readonly] Return the type of values stored in the Automaton as specified when creating the object. By default ahocorasick.STORE_ANY``is used, thus any Python object is accepted as value. When ``ahocorasick.STORE_INTS or ahocorasick.STORE_LENGTH is used then values are 32-bit integers and do not use additional memory. See the add_word documentation for details.

Other methods

dump() => (list of nodes, list of edges, list of fail links) Returns a three-tuple of lists describing the Automaton as a graph of (nodes, edges, failure links):

- nodes: each item is a pair (node id, end of word marker)
- edges: each item is a triple (node id, label char, child node id)
- failure links: each item is a pair (source node id, node if connected by fail node)

For each of these the node id is a unique number and a label is a single byte.

The source repository and source package also contains the dump2dot.py script that converts dump() results to a graphviz dot format.

get_stats() => dict Return a dictionary containing some Automaton statistics:

- nodes_count — total number of nodes
- words_count — same as len(automaton)
- longest_word — length of the longest word
- links_count — number of edges
- sizeof_node — size of single node in bytes
- total_size — total size of trie in bytes (about nodes_count * size_of node + links_count * size of pointer). The real size occupied by the data structure could be larger because of internal memory fragmentation that can occur in a memory manager.

__sizeof__() => int Return the approximate size in bytes occupied by the Automaton instance in memory excluding the size of associated objects when the Automaton is created with Automaton() or ‘‘Automaton(ahocorasick.STORE_ANY). Also available by calling sys.getsizeof(automaton instance).

7.1.3 AutomatonSearchIter class

This class is not available directly but instances of AutomatonSearchIter` are returned by the ``iter method of an Automaton. This iterator has the following methods:

set(string, [reset]) => None Set a new string to search. When the `reset` argument is `False` (default), then the Aho-Corasick procedure is continued and the internal state of the Automaton and index are not reset. This allow to search for large strings in multiple chunks. For example:

```
it = automaton.iter(b"")
while True:
    buffer = receive(server_address, 4096)
    if not buffer:
        break

    it.set(buffer)
    for index, value in it:
        print(index, '=>', value)
```

When `reset` is `True` then processing is restarted. For example this code:

```
for string in set:
    for index, value in automaton.iter(string)
        print(index, '=>', value)
```

does the same job as:

```
it = automaton.iter(b"")
for string in set:
    it.set(it, True)
    for index, value in it:
        print(index, '=>', value)
```

7.1.4 More Examples

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()

# add some words to trie
>>> for index, word in enumerate("he her hers she".split()):
...   A.add_word(word, (index, word))

# test is word exists in set
>>> "he" in A
True
>>> "HER" in A
False
>>> A.get("he")
(0, 'he')
>>> A.get("she")
(3, 'she')
>>> A.get("cat", "<not exists>")
'<not exists>'
>>> A.get("dog")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError
>>>

# convert the trie in an Aho-Corasick automaton
A.make_automaton()

# then find all occurrences of keys in a string
```

```
for item in A.iter("_hershe_"):
...  print(item)
...
(2, (0, 'he'))
(3, (1, 'her'))
(4, (2, 'hers'))
(6, (3, 'she'))
(6, (0, 'he'))
```

7.1.5 Example of the keys method behaviour

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()

# add some words to trie
>>> for index, word in enumerate("cat catastropha rat rate bat".split()):
...   A.add_word(word, (index, word))

# prefix
>>> list(A.keys("cat"))
["cat", "catastropha"]

# pattern
>>> list(A.keys("?at", "?", ahocorasick.MATCH_EXACT_LENGTH))
["bat", "cat", "rat"]

>>> list(A.keys("?at?", "?", ahocorasick.MATCH_AT_MOST_PREFIX))
["bat", "cat", "rat", "rate"]

>>> list(A.keys("?at?", "?", ahocorasick.MATCH_AT_LEAST_PREFIX))
["rate"]
```

CHAPTER 8

Build and install

To install for common operating systems use pip. Pre-built wheels should be available on Pypi:

```
pip install pyahocorasick
```

To build from sources you need to have a C compiler installed and configured which should be standard on Linux and easy to get on MacOSX.

On Windows and Python 2.7 you need the Microsoft Visual C++ Compiler for Python 2.7 (or Visual Studio 2008). There have been reports that *pyahocorasick* does not build with MinGW. It may build with cygwin. If you get this working with these platforms, please report!

To build from sources, clone the git repository or download and extract the source archive.

Install *setuptools* and then run (in a *virtualenv* of course!):

```
pip install .
```

If compilation succeeds, the module is ready to use.

8.1 Unicode and bytes

The type of strings accepted and returned by `Automaton` methods are either **unicode** or **bytes**, depending on a compile time settings (preprocessor definition of `AHOCORASICK_UNICODE` as set in *setup.py*).

The `Automaton.unicode` attributes can tell you how the library was built. On Python 3, unicode is the default. On Python 2, bytes is the default.

Warning: When the library is built with unicode support, an Automaton will store 2 or 4 bytes per letter, depending on your Python installation. When built for bytes, only one byte per letter is needed.

CHAPTER 9

Tests

The source repository contains several tests. To run them use:

```
make test
```

CHAPTER 10

Support

Support is available through the GitHub issue tracker to report bugs or ask questions.

CHAPTER 11

Contributing

You can submit contributions through GitHub pull requests.

CHAPTER 12

Authors

The main author: Wojciech Muła, wojciech_mula@poczta.onet.pl

This library would not be possible without help of many people, who contributed in various ways. They created pull requests, reported bugs (as GitHub issues or via direct messages), proposed fixes, or spent their valuable time on testing. Thank you.

CHAPTER 13

License

Library is licensed under very liberal BSD-3-Clause license. Some portions of the code are dedicated to the public domain such as the pure Python automaton.

Full text of license is available in LICENSE file.

CHAPTER 14

API Reference

CHAPTER 15

Indices and tables

- genindex
- modindex
- search