pyahocorasick Documentation

Wojciech Muła

Jan 18, 2019

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 occurrences at once in some input text. The library provides an *ahocorasick* Python module that you can use as a plain dict-like Trie or convert a Trie to an automaton for efficient Aho-Corasick search.

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 as tests and the pure Python automaton are dedicated to the Public Domain.

CHAPTER 1

Download and source code

You can fetch pyahocorasick from:

- GitHub https://github.com/WojciechMula/pyahocorasick/
- Pypi https://pypi.python.org/pypi/pyahocorasick/
- Conda-Forge https://github.com/conda-forge/pyahocorasick-feedstock/

CHAPTER 2

Quick start

This module is written in C. You need a C compiler installed to compile native CPython extensions. To install:

```
pip install pyahocorasick
```

Then create an Automaton:

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

You can use the Automaton class as a trie. Add some string keys 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))
```

Then check if some string exists in the trie:

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

And 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
```

Now convert the trie to an Aho-Corasick automaton to enable Aho-Corasick search:

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

Then search all occurrences of the keys (the needles) in an input string (our haystack).

Here we print the results and just check that they are correct. The *Automaton.iter()* method return the results as two-tuples of the *end index* where a trie key was found in the input string and the associated *value* for this key. Here we had stored as values a tuple with the original string and its trie insertion order:

```
>>> for end_index, (insert_order, original_value) in A.iter(haystack):
...     start_index = end_index - len(original_value) + 1
...     print((start_index, end_index, (insert_order, original_value)))
...     assert haystack[start_index:start_index + len(original_value)] == original_
↪value
...
(1, 2, (0, 'he'))
(1, 3, (1, 'her'))
(1, 4, (2, 'hers'))
(4, 6, (3, 'she'))
(5, 6, (0, 'he'))
```

You can also create an eventually large automaton ahead of time and *pickle* it to re-load later. Here we just pickle to a string. You would typically pickle to a file instead:

```
>>> import cPickle
>>> pickled = cPickle.dumps(A)
>>> B = cPickle.loads(pickled)
>>> B.get('he')
(0, 'he')
```

See also:

- FAQ and Who is using pyahocorasick? https://github.com/WojciechMula/pyahocorasick/wiki/FAQ#who-is-using-pyahocorasick

CHAPTER 3

Documentation

The full documentation including the API overview and reference is published on readthedocs.

CHAPTER 4

Overview

With an Aho-Corasick automaton you can efficiently search all occurrences 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 to reuse them over and over as an indexed structure for fast multi pattern string matching.

One of the advantages 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 applications where you search for several pre-defined "needles" in a 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 has a slightly different API.

4.1 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 and only value.

> **Warning:** When the library is built with unicode support on Python 3, 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.
>
> Unicode is **NOT supported** on Python 2 for now.

CHAPTER 5

Build and install from PyPi

To install for common operating systems, use pip. Pre-built wheels should be available on Pypi at some point in the future:

```
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 (aka. Visual Studio 2008). There have been reports that *pyahocorasick* does not build yet with MinGW. It may build with cygwin but this has not been tested. If you get this working with these platforms, please report in a ticket!

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

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

```
pip install .
```

If compilation succeeds, the module is ready to use.

CHAPTER 6

Support

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

CHAPTER 7

Contributing

You can submit contributions through GitHub pull requests.

CHAPTER 8

Authors

The initial author and maintainer is Wojciech Muła. Philippe Ombredanne, the current co-owner, rewrote documentation, setup CI servers and did a whole lot of work to make this module better accessible to end users.

Alphabetic list of authors:

- **Andrew Grigorev**
- **Bogdan**
- **David Woakes**
- **Edward Betts**
- **Frankie Robertson**
- **Frederik Petersen**
- **gladtosee**
- **INADA Naoki**
- **Jan Fan**
- **Pastafarianist**
- **Philippe Ombredanne**
- **Renat Nasyrov**
- **Sylvain Zimmer**
- **Xiaopeng Xu**

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 9

License

This 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 and test code.

Full text of license is available in LICENSE file.

CHAPTER 10

Other Aho-Corasick 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:

- py_aho_corasick by Jan
- Written in pure Python.
- Poor performance.
- ahocorapy by abusix
- Written in pure Python.
- Better performance than py-aho-corasick.
- Using pypy, ahocorapy's search performance is only slightly worse than pyahocorasick's.
- Performs additional suffix shortcutting (more setup overhead, less search overhead for suffix lookups).
- Includes visualization tool for resulting automaton (using pygraphviz).
- MIT-licensed, 100% test coverage, tested on all major python versions (+ pypy)
- 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
- Written in C.

- seems unmaintained (last update in 2005).
- GPL-licensed.

CHAPTER 11

API Overview

This is a quick tour of the API for the C **ahocorasick** module. See the full API doc for more details. The pure Python module has a slightly different interface.

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

11.1 Module constants

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

11.2 Automaton class

Note: `Automaton` instances are pickle-able meaning that you can create ahead of time an eventually large automaton then save it to disk and re-load it later to reuse it over and over as a persistent multi-string search index. Internally, Automaton implements the `__reduce__() magic method`.

`Automaton([value_type], [key_type])`

> Create a new empty Automaton optionally passing a *value_type* to indicate what is the type of associated values (default to any Python object type). It can be one of `ahocorasick.STORE_ANY`, `ahocorasick.STORE_INTS` or `ahocorasick.STORE_LENGTH`. In the last case the length of

the key will be stored in the automaton. The optional argument *key_type* can be `ahocorasick.KEY_STRING` or `ahocorasick.KEY_SEQUENCE`. In the latter case keys will be tuples of integers. The size of integer depends on the version and platform Python is running on, but for versions of Python >= 3.3, it is guaranteed to be 32-bits.

11.3 Automaton Trie methods

The Automaton class has the following main trie-like methods:

`add_word(key, [value]) => bool` Add a `key` string to the dict-like trie and associate this key with a `value`.

`remove_word(key) => bool` Remove a `key` string from the dict-like trie.

`pop(key) => value` Remove a `key` string from the dict-like trie and return the associated `value`.

`exists(key) => bool` or `key in ...` Return True if the key is present in the trie.

`match(key) => bool` Return True if there is a prefix (or key) equal to `key` present in the trie.

11.4 Automaton Dictionary-like methods

A pyahocorasick Automaton trie behaves more or less like a Python dictionary and implements a subset of dict-like methods. Some of them are:

`get(key[, default])` Return the value associated with the `key` string. Similar to *dict.get()*.

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

`values([prefix, [wildcard, [how]]]) => yield object` Return an iterator on values associated with each keys.

`items([prefix, [wildcard, [how]]]) => yield tuple (string, object)` Return an iterator on tuples of (key, value).

11.4.1 Wildcard search

The 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 (.) in 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?"
```

11.5 Aho-Corasick methods

The Automaton class has the following main Aho-Corasick methods:

`make_automaton()` Finalize and create the Aho-Corasick automaton.

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.

11.5.1 AutomatonSearchIter class

Instances of this class are returned by the `iter` method of an `Automaton`. This iterator can be manipulated through its *set()* method.

set(string, [reset]) => None Set a new string to search eventually keeping the current Automaton state to continue searching for the next chunk of a string.

For example:

```
>>> it = A.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 string_set:
...     for index, value in A.iter(string)
...         print(index, '=>', value)
```

does the same job as:

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

11.6 Automaton Attributes

The Automaton class has the following attributes:

kind [readonly] Return the state of the `Automaton` instance.

store [readonly] Return the type of values stored in the Automaton as specified at creation.

11.7 Saving and loading automaton

There is support for two method of saving and loading an automaton:

- the standard `pickle` protocol,
- custom `save` and `load` methods.

While pickling is more convenient to use, it has quite high memory requirements. The `save`/`load` method try to overcome this problem.

Warning: Neither format of pickle nor save are safe. Although there are a few sanity checks, they are not sufficient to detect all possible input errors.

11.7.1 Pickle

```
import ahocorasick
import pickle

# build automaton

A = ahocorasick.Automaton()
# ... A.add_data, A.make_automaton

# save current state
with open(path, 'wb') as f:
    pickle.dump(A, f)

# load saved state
with open(path, 'rb') as f:
    B = pickle.load(f)
```

11.7.2 Save/load methods

```
import ahocorasick
import pickle

# build automaton

A = ahocorasick.Automaton()
# ... A.add_data, A.make_automaton

# save current state
A.save(path, pickle.dumps)

# load saved state
B = ahocorasick.load(path, pickle.loads)
```

Automaton method `save` requires `path` to the file which will store data. If the automaton type is `STORE_ANY`, i.e. values associated with words are any python objects, then `save` requires also another argument, a callable. The callable serializes python object into bytes; in the example above we use standard pickle `dumps` function.

Module method `load` also requires `path` to file that has data previously saved. Because at the moment of loading data we don't know what is the store attribute of automaton, the second argument - a callable - is required. The callable must convert back given bytes object into python value, that will be stored in automaton. Similarly, standard `pickle.loads` function can be passed.

11.8 Other Automaton methods

The Automaton class has a few other interesting methods:

dump() => (list of nodes, list of edges, list of fail links) Return a three-tuple of lists describing the Automaton as a graph of (nodes, edges, failure links). The source repository and source package also contains the `dump2dot.py` script that converts `dump()` results to a graphviz dot format for convenient visualization of the trie and Automaton data structure.

get_stats() => dict Return a dictionary containing Automaton statistics. Note that 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. Also available by calling sys.getsizeof(automaton instance).

CHAPTER 12

Examples

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

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

>>> # test that these key words exists in the trie all right
>>> '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
>>> A.remove_word('he')
True
>>> A.remove_word('he')
False
>>> A.pop('she')
(3, 'she')
>>> 'she' in A
False

>>> # convert the trie in an Aho-Corasick automaton
>>> A = ahocorasick.Automaton()
>>> for index, word in enumerate('he her hers she'.split()):
```

(continues on next page)

(continued from previous page)

```
...   A.add_word(word, (index, word))
>>> A.make_automaton()

>>> # then find all occurrences of the stored 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'))
```

12.1 Example of the keys method behavior

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

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

>>> # Search some prefix
>>> list(A.keys('cat'))
['cat', 'catastropha']

>>> # Search with a wildcard: here '?' is used as a wildcard. You can use any␣
↪character you like.
>>> 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 13

API Reference

13.1 Automaton(value_type=ahocorasick.STORE_ANY, [key_type])

Create a new empty Automaton. Both `value_type` and `key_type` are optional.

`value_type` is one of these constants:

- ahocorasick.STORE_ANY [*default*] : The associated value can be any Python object.
- ahocorasick.STORE_LENGTH : The length of an added string key is automatically used as the associated value stored in the trie for that key.
- ahocorasick.STORE_INTS : The associated value must be a 32-bit integer.

`key_type` defines the type of data that can be stored in an automaton; it is one of these constants and defines type of data might be stored:

- ahocorasick.KEY_STRING [*default*] : string
- ahocorasick.KEY_SEQUENCE : sequences of integers; The size of integer depends the version and platform Python, but for versions of Python >= 3.3, it is guaranteed to be 32-bits.

13.1.1 Examples

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
>>> A
<ahocorasick.Automaton object at 0x7f1da1bdf7f0>
>>> B = ahocorasick.Automaton(ahocorasick.STORE_ANY)
>>> B
<ahocorasick.Automaton object at 0x7f1da1bdf6c0>
>>> C = ahocorasick.Automaton(ahocorasick.STORE_INTS, ahocorasick.KEY_STRING)
>>> C
<ahocorasick.Automaton object at 0x7f1da1527f10>
```

13.2 add_word(key, [value]) -> boolean

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. The value associated with an existing word is replaced.

The value is either mandatory or optional:

- 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_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.
- 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.

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).

13.2.1 Examples

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
>>> A.add_word("pyahocorasick")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: A value object is required as second argument.
>>> A.add_word("pyahocorasick", (42, 'text'))
True
>>> A.get("pyhocorasick")
(42, 'text')
>>> A.add_word("pyahocorasick", 12)
False
>>> A.get("pyhocorasick")
12
```

```
>>> import ahocorasick
>>> B = ahocorasick.Automaton(ahocorasick.STORE_INTS)
>>> B.add_word("cat")
True
>>> B.get()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> B.get("cat")
1
>>> B.add_word("dog")
True
>>> B.get("dog")
2
>>> B.add_word("tree", 42)
True
>>> B.get("tree")
42
```

(continues on next page)

(continued from previous page)

```
>>> B.add_word("cat", 43)
False
>>> B.get("cat")
43
```

13.3 exists(key) -> boolean

Return True if the `key` is present in the trie. Same as using the 'in' keyword.

13.3.1 Examples

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
>>> A.add_word("cat", 1)
True
>>> A.exists("cat")
True
>>> A.exists("dog")
False
>>> 'elephant' in A
False
>>> 'cat' in A
True
```

13.4 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.

13.4.1 Example

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
>>> A.add_word("cat", 42)
True
>>> A.get("cat")
42
>>> A.get("dog")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError
>>> A.get("dog", "good dog")
'good dog'
```

13.5 longest_prefix(string) => integer

Return the length of the longest prefix of string that exists in the trie.

13.5.1 Examples

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
>>> A.add_word("he", True)
True
>>> A.add_word("her", True)
True
>>> A.add_word("hers", True)
True
>>> A.longest_prefix("she")
0
>>> A.longest_prefix("herself")
4
```

13.6 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 calls to match('e'), match('ex'), . . . , match('exampl') or match('example') all return True. But exists() is True only when calling exists('example').

13.6.1 Examples

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
>>> A.add_word("example", True)
True
>>> A.match("e")
True
>>> A.match("ex")
True
>>> A.match("exa")
True
>>> A.match("exam")
True
>>> A.match("examp")
True
>>> A.match("exampl")
True
>>> A.match("example")
True
>>> A.match("examples")
False
>>> A.match("python")
False
```

13.7 len() -> integer

Return the number of distinct keys added to the trie.

13.7.1 Examples

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
>>> len(A)
0
>>> A.add_word("python", 1)
True
>>> len(A)
1
>>> A.add_word("elephant", True)
True
>>> len(A)
2
```

13.8 remove_word(word) -> bool

Remove given word from a trie. Return True if words was found, False otherwise.

13.8.1 Examples

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
>>> A.add_word("cat", 1)
True
>>> A.add_word("dog", 2)
True
>>> A.remove_word("cat")
True
>>> A.remove_word("cat")
False
>>> A.remove_word("dog")
True
>>> A.remove_word("dog")
False
>>>
```

13.9 pop(word)

Remove given word from a trie and return associated values. Raise a `KeyError` if the word was not found.

13.9.1 Examples

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
>>> A.add_word("cat", 1)
True
>>> A.add_word("dog", 2)
True
>>> A.pop("elephant")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError
>>> A.pop("cat")
1
>>> A.pop("dog")
2
>>> A.pop("cat")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError
```

13.10 clear()

Remove all keys from the trie. This method invalidates all iterators.

13.10.1 Examples

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
>>> A.add_word("cat", 1)
True
>>> A.add_word("dog", 2)
True
>>> A.add_word("elephant", 3)
True
>>> len(A)
3
>>> A.clear()
>>> len(A)
0
```

13.11 keys([prefix, [wildcard, [how]]])

Return an iterator on keys. If the optional `prefix` string is provided, only yield keys starting with this prefix.

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

13.12 items([prefix, [wildcard, [how]]])

Return an iterator on tuples of (key, value). Keys are matched optionally to the prefix using the same logic and arguments as in the keys() method.

13.13 values([prefix, [wildcard, [how]]])

Return an iterator on values associated with each keys. Keys are matched optionally to the prefix using the same logic and arguments as in the keys() method.

13.14 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 creation the `Automaton.kind` attribute is set to `ahocorasick.AHOCORASICK`.

13.15 iter(string, [start, [end]], ignore_white_space=False)

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]`.

The `ignore_white_space` optional arguments can be used to ignore white spaces from input string.

13.16 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 for 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 an input string slice as in string[start:end].

Equivalent to a loop on iter() calling a callable at each iteration.

13.17 __reduce__()

Return pickle-able data for this automaton instance.

13.18 save(path, serializer)

Save content of automaton in an on-disc file.

`Serializer` is a callable object that is used when automaton store type is `STORE_ANY`. This method converts a python object into bytes; it can be `pickle.dumps`.

13.19 load(path, deserializer) => Automaton

Load automaton previously stored on disc using `save` method.

`Deserializer` is a callable object which converts bytes back into python object; it can be `pickle.loads`.

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).

13.20 get_stats() -> dict

Return a dictionary containing Automaton statistics.

- *nodes_count* - total number of nodes
- *words_count* - number of distinct words (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).

13.20.1 Examples

```
>>> import ahocorasick
>>> A = ahocorasick.Automaton()
>>> A.add_word("he", None)
True
>>> A.add_word("her", None)
True
>>> A.add_word("hers", None)
True
>>> A.get_stats()
{'nodes_count': 5, 'words_count': 3, 'longest_word': 4, 'links_count': 4, 'sizeof_node
↪': 40, 'total_size': 232}
```

13.21 dump()

Return 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 number.

13.22 set(string, reset=False)

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 end index of the string being searched are not reset. This allow to search for large strings in multiple smaller chunks.