DAWG Documentation

Release 0.6

Mikhail Korobov

December 24, 2016

Contents

This package provides DAWG(DAFSA)-based dictionary-like read-only objects for Python (2.x and 3.x).

String data in a DAWG may take 200x less memory than in a standard Python dict and the raw lookup speed is comparable; it also provides fast advanced methods like prefix search.

Based on dawgdic C++ library.

CHAPTER 1

License

Wrapper code is licensed under MIT License. Bundled dawgdic C++ library is licensed under BSD license. Bundled libb64 is Public Domain.

CHAPTER 2

Installation

From PyPI:

```
pip install DAWG
```

CHAPTER 3

Usage

There are several DAWG classes in this package:

- `dawg.DAWG` - basic DAWG wrapper; it can store unicode keys and do exact lookups;
- `dawg.CompletionDAWG` - `dawg.DAWG` subclass that supports key completion and prefix lookups (but requires more memory);
- `dawg.BytesDAWG` - `dawg.CompletionDAWG` subclass that maps unicode keys to lists of `bytes` objects.
- `dawg.RecordDAWG` - `dawg.BytesDAWG` subclass that maps unicode keys to lists of data tuples. All tuples must be of the same format (the data is packed using python `struct` module).
- `dawg.IntDAWG` - `dawg.DAWG` subclass that maps unicode keys to integer values.
- `dawg.IntCompletionDAWG` - `dawg.CompletionDAWG` subclass that maps unicode keys to integer values.

3.1 DAWG and CompletionDAWG

`DAWG` and `CompletionDAWG` are useful when you need fast & memory efficient simple string storage. These classes does not support assigning values to keys.

`DAWG` and `CompletionDAWG` constructors accept an iterable with keys:

```
>>> import dawg
>>> words = [u'foo', u'bar', u'foobar', u'foö', u'bör']
>>> base_dawg = dawg.DAWG(words)
>>> completion_dawg = dawg.CompletionDAWG(words)
```

It is then possible to check if the key is in a DAWG:

```
>>> u'foo' in base_dawg
True
>>> u'baz' in completion_dawg
False
```

It is possible to find all keys that starts with a given prefix in a `CompletionDAWG`:

```
>>> completion_dawg.keys(u'foo')
>>> [u'foo', u'foobar']
```

to test whether some key begins with a given prefix:

```
>>> completion_dawg.has_keys_with_prefix(u'foo')
>>> True
```

and to find all prefixes of a given key:

```
>>> base_dawg.prefixes(u'foobarz')
[u'foo', u'foobar']
```

Iterator versions are also available:

```
>>> for key in completion_dawg.iterkeys(u'foo'):
...     print(key)
foo
foobar
>>> for prefix in base_dawg.iterprefixes(u'foobarz'):
...     print(prefix)
foo
foobar
```

It is possible to find all keys similar to a given key (using a one-way char translation table):

```
>>> replaces = dawg.DAWG.compile_replaces({u'o': u'ö'})
>>> base_dawg.similar_keys(u'foo', replaces)
[u'foo', u'foö']
>>> base_dawg.similar_keys(u'foö', replaces)
[u'foö']
>>> base_dawg.similar_keys(u'bor', replaces)
[u'bör']
```

3.2 BytesDAWG

`BytesDAWG` is a `CompletionDAWG` subclass that can store binary data for each key.

`BytesDAWG` constructor accepts an iterable with `(unicode_key, bytes_value)` tuples:

```
>>> data = [(u'key1', b'value1'), (u'key2', b'value2'), (u'key1', b'value3')]
>>> bytes_dawg = dawg.BytesDAWG(data)
```

There can be duplicate keys; all unique values are stored in this case:

```
>>> bytes_dawg[u'key1']
[b'value1, b'value3']
```

For unique keys a list with a single value is returned for consistency:

```
>>> bytes_dawg[u'key2']
[b'value2']
```

`KeyError` is raised for missing keys; use `get` method if you need a default value instead:

```
>>> bytes_dawg.get(u'foo', None)
None
```

`BytesDAWG` support `keys`, `items`, `iterkeys` and `iteritems` methods (they all accept optional key prefix). There is also support for `similar_keys`, `similar_items` and `similar_item_values` methods.

3.3 RecordDAWG

`RecordDAWG` is a `BytesDAWG` subclass that automatically packs & unpacks the binary data from/to Python objects using `struct` module from the standard library.

First, you have to define a format of the data. Consult Python docs (http://docs.python.org/library/struct.html#format-strings) for the format string specification.

For example, let's store 3 short unsigned numbers (in a Big-Endian byte order) as values:

```
>>> format = ">HHH"
```

`RecordDAWG` constructor accepts an iterable with `(unicode_key, value_tuple)`. Let's create such iterable using `zip` function:

```
>>> keys = [u'foo', u'bar', u'foobar', u'foo']
>>> values = [(1, 2, 3), (2, 1, 0), (3, 3, 3), (2, 1, 5)]
>>> data = zip(keys, values)
>>> record_dawg = RecordDAWG(format, data)
```

As with `BytesDAWG`, there can be several values for the same key:

```
>>> record_dawg['foo']
[(1, 2, 3), (2, 1, 5)]
>>> record_dawg['foobar']
[(3, 3, 3)]
```

3.4 BytesDAWG and RecordDAWG implementation details

`BytesDAWG` and `RecordDAWG` stores data at the end of the keys:

```
<utf8-encoded key><separator><base64-encoded data>
```

Data is encoded to base64 because dawgdic C++ library doesn't allow zero bytes in keys (it uses null-terminated strings) and such keys are very likely in binary data.

In DAWG versions prior to 0.5 `<separator>` was `chr(255)` byte. It was chosen because keys are stored as UTF8-encoded strings and `chr(255)` is guaranteed not to appear in valid UTF8, so the end of text part of the key is not ambiguous.

But `chr(255)` was proven to be problematic: it changes the order of the keys. Keys are naturally returned in lexicographical order by DAWG. But if `chr(255)` appears at the end of each text part of a key then the visible order would change. Imagine `'foo'` key with some payload and `'foobar'` key with some payload. `'foo'` key would be greater than `'foobar'` key: values compared would be `'foo<sep>'` and `'foobar<sep>'` and `ord(<sep>)==255` is greater than `ord(<any other character>)`.

So now the default `<separator>` is chr(1). This is the lowest allowed character and so it preserves the alphabetical order.

It is not strictly correct to use chr(1) as a separator because chr(1) is a valid UTF8 character. But I think in practice this won't be an issue: such control character is very unlikely in text keys, and binary keys are not supported anyway because dawgdic doesn't support keys containing chr(0).

If you can't guarantee chr(1) is not a part of keys, lexicographical order is not important to you or there is a need to read a `BytesDAWG`/`RecordDAWG` created by DAWG < 0.5 then pass `payload_separator` argument to the constructor:

```
>>> BytesDAWG(payload_separator=b'\xff').load('old.dawg')
```

The storage scheme has one more implication: values of `BytesDAWG` and `RecordDAWG` are also sorted lexicographically.

For `RecordDAWG` there is a gotcha: in order to have meaningful ordering of numeric values store them in big-endian format:

```
>>> data = [('foo', (3, 2, 256)), ('foo', (3, 2, 1)), ('foo', (3, 2, 3))]
>>> d = RecordDAWG("3H", data)
>>> d.items()
[(u'foo', (3, 2, 256)), (u'foo', (3, 2, 1)), (u'foo', (3, 2, 3))]

>>> d2 = RecordDAWG(">3H", data)
>>> d2.items()
[(u'foo', (3, 2, 1)), (u'foo', (3, 2, 3)), (u'foo', (3, 2, 256))]
```

3.5 IntDAWG and IntCompletionDAWG

`IntDAWG` is a `{unicode -> int}` mapping. It is possible to use `RecordDAWG` for this, but `IntDAWG` is natively supported by dawgdic C++ library and so `__getitem__` is much faster.

Unlike `BytesDAWG` and `RecordDAWG`, `IntDAWG` doesn't support having several values for the same key.

`IntDAWG` constructor accepts an iterable with (unicode_key, integer_value) tuples:

```
>>> data = [ (u'foo', 1), (u'bar', 2) ]
>>> int_dawg = dawg.IntDAWG(data)
```

It is then possible to get a value from the IntDAWG:

```
>>> int_dawg[u'foo']
1
```

`IntCompletionDAWG` supports all `IntDAWG` and `CompletionDAWG` methods, plus `.items()` and `.iteritems()`.

3.6 Persistence

All DAWGs support saving/loading and pickling/unpickling.

Write DAWG to a stream:

```
>>> with open('words.dawg', 'wb') as f:
...     d.write(f)
```

Save DAWG to a file:

```
>>> d.save('words.dawg')
```

Load DAWG from a file:

```
>>> d = dawg.DAWG()
>>> d.load('words.dawg')
```

Warning: Reading DAWGs from streams and unpickling are currently using 3x memory compared to loading DAWGs using `load` method; please avoid them until the issue is fixed.

Read DAWG from a stream:

```
>>> d = dawg.RecordDAWG(format_string)
>>> with open('words.record-dawg', 'rb') as f:
...     d.read(f)
```

DAWG objects are picklable:

```
>>> import pickle
>>> data = pickle.dumps(d)
>>> d2 = pickle.loads(data)
```

CHAPTER 4

Benchmarks

For a list of 3000000 (3 million) Russian words memory consumption with different data structures (under Python 2.7):

- dict(unicode words -> word lengths): about 600M
- list(unicode words) : about 300M
- `marisa_trie.RecordTrie` : 11M
- `marisa_trie.Trie`: 7M
- `dawg.DAWG`: 2M
- `dawg.CompletionDAWG`: 3M
- `dawg.IntDAWG`: 2.7M
- `dawg.RecordDAWG`: 4M

Note: Lengths of words were not stored as values in `dawg.DAWG`, `dawg.CompletionDAWG` and `marisa_trie.Trie` because they don't support this.

Note: marisa-trie is often more memory efficient than DAWG (depending on data); it can also handle larger datasets and provides memory-mapped IO, so don't dismiss marisa-trie based on this README file. It is still several times slower than DAWG though.

Benchmark results (100k unicode words, integer values (lengths of the words), Python 3.3, macbook air i5 1.8 Ghz):

```
dict __getitem__ (hits)           7.300M ops/sec
DAWG __getitem__ (hits)           not supported
BytesDAWG __getitem__ (hits)      1.230M ops/sec
RecordDAWG __getitem__ (hits)     0.792M ops/sec
IntDAWG __getitem__ (hits)        4.217M ops/sec
dict get() (hits)                 3.775M ops/sec
DAWG get() (hits)                 not supported
BytesDAWG get() (hits)            1.027M ops/sec
RecordDAWG get() (hits)           0.733M ops/sec
IntDAWG get() (hits)              3.162M ops/sec
dict get() (misses)               4.533M ops/sec
DAWG get() (misses)               not supported
BytesDAWG get() (misses)          3.545M ops/sec
RecordDAWG get() (misses)         3.485M ops/sec
```

```
IntDAWG get() (misses)           3.928M ops/sec

dict __contains__ (hits)         7.090M ops/sec
DAWG __contains__ (hits)         4.685M ops/sec
BytesDAWG __contains__ (hits)    3.885M ops/sec
RecordDAWG __contains__ (hits)   3.898M ops/sec
IntDAWG __contains__ (hits)      4.612M ops/sec

dict __contains__ (misses)       5.617M ops/sec
DAWG __contains__ (misses)       6.204M ops/sec
BytesDAWG __contains__ (misses)  6.026M ops/sec
RecordDAWG __contains__ (misses) 6.007M ops/sec
IntDAWG __contains__ (misses)    6.180M ops/sec

DAWG.similar_keys  (no replaces) 0.492M ops/sec
DAWG.similar_keys  (l33t)        0.413M ops/sec

dict items()                     55.032 ops/sec
DAWG items()                     not supported
BytesDAWG items()                14.826 ops/sec
RecordDAWG items()               9.436 ops/sec
IntDAWG items()                  not supported

dict keys()                      200.788 ops/sec
DAWG keys()                      not supported
BytesDAWG keys()                 20.657 ops/sec
RecordDAWG keys()                20.873 ops/sec
IntDAWG keys()                   not supported

DAWG.prefixes (hits)             1.552M ops/sec
DAWG.prefixes (mixed)            4.342M ops/sec
DAWG.prefixes (misses)           4.094M ops/sec
DAWG.iterprefixes (hits)         0.391M ops/sec
DAWG.iterprefixes (mixed)        0.476M ops/sec
DAWG.iterprefixes (misses)       0.498M ops/sec

RecordDAWG.keys(prefix="xxx"), avg_len(res)==415             5.562K ops/sec
RecordDAWG.keys(prefix="xxxxx"), avg_len(res)==17            104.011K ops/sec
RecordDAWG.keys(prefix="xxxxxxxx"), avg_len(res)==3          318.129K ops/sec
RecordDAWG.keys(prefix="xxxxx..xx"), avg_len(res)==1.4       462.238K ops/sec
RecordDAWG.keys(prefix="xxx"), NON_EXISTING                  4292.625K ops/sec
```

Please take this benchmark results with a grain of salt; this is a very simple benchmark on a single data set.

CHAPTER 5

Current limitations

- `IntDAWG` is currently a subclass of `DAWG` and so it doesn't support `keys()` and `items()` methods;
- `read()` method reads the whole stream (DAWG must be the last or the only item in a stream if it is read with `read()` method) - pickling doesn't have this limitation;
- DAWGs loaded with `read()` and unpickled DAWGs uses 3x-4x memory compared to DAWGs loaded with `load()` method;
- there are `keys()` and `items()` methods but no `values()` method;
- iterator versions of methods are not always implemented;
- `BytesDAWG` and `RecordDAWG` has a limitation: values larger than 8KB are unsupported;
- the maximum number of DAWG units is limited: number of DAWG units (and thus transitions - but not elements) should be less than 2^29; this mean that it may be impossible to build an especially huge DAWG (you may split your data into several DAWGs or try marisa-trie in this case).

Contributions are welcome!

CHAPTER 6

Contributing

Development happens at github: https://github.com/pytries/DAWG

Issue tracker: https://github.com/pytries/DAWG/issues

Feel free to submit ideas, bugs or pull requests.

If you found a bug in a C++ part please report it to the original bug tracker.

6.1 How is source code organized

There are 4 folders in repository:

- `bench` - benchmarks & benchmark data;
- `lib` - original unmodified dawgdic C++ library and a customized version of libb64 library. They are bundled for easier distribution; if something is have to be fixed in these libraries consider fixing it in the original repositories;
- `src` - wrapper code; `src/dawg.pyx` is a wrapper implementation; `src/*.pxd` files are Cython headers for corresponding C++ headers; `src/*.cpp` files are the pre-built extension code and shouldn't be modified directly (they should be updated via `update_cpp.sh` script).
- `tests` - the test suite.

6.2 Running tests and benchmarks

Make sure tox is installed and run

```
$ tox
```

from the source checkout. Tests should pass under python 2.6, 2.7, 3.2, 3.3 and 3.4.

In order to run benchmarks, type

```
$ tox -c bench.ini
```

6.3 Authors & Contributors

- Mikhail Korobov <kmike84@gmail.com>;
- Dan Blanchard;

- Jakub Wilk;
- Alex Moiseenko;
- Matt Hickford;
- Ikuya Yamada.

This module uses dawgdic C++ library by Susumu Yata & contributors.

base64 decoder is a modified version of libb64 (original author is Chris Venter).

CHAPTER 7

Changes

7.1 0.7.8 (2015-04-18)

- extra type annotations are added to make the code a bit faster;
- mercurial mirror at bitbucket is dropped;
- wrapper is rebuilt with Cython 0.22.

7.2 0.7.7 (2014-11-19)

- `DAWG.b_prefixes` method for avoiding utf8 encoding/decoding (thanks Ikuya Yamada);
- wrapper is rebuilt with Cython 0.21.1.

7.3 0.7.6 (2014-08-10)

- Wrapper is rebuilt with Cython 0.20.2 to fix some issues.

7.4 0.7.5 (2014-06-05)

- Switched to setuptools;
- some wheels are uploaded to pypi.

7.5 0.7.4 (2014-05-29)

- Fixed a bug in DAWG building: input should be sorted according to its binary representation.

7.6 0.7.3 (2014-05-29)

- Wrapper is rebuilt with Cython 0.21dev;
- Python 3.4 compatibility is verified.

7.7 0.7.2 (2013-10-03)

- `has_keys_with_prefix(prefix)` method (thanks Matt Hickford)

7.8 0.7.1 (2013-05-25)

- Extension is rebuilt with Cython 0.19.1;
- fixed segfault that happened on lookup from incorrectly loaded DAWG (thanks Alex Moiseenko).

7.9 0.7 (2013-04-05)

- IntCompletionDAWG

7.10 0.6.1 (2013-03-23)

- Installation issues in environments with LC_ALL=C are fixed;
- PyPy is officially unsupported now (use DAWG-Python with PyPy).

7.11 0.6 (2013-03-22)

- many thread-safety bugs are fixed (at the cost of slowing library down).

7.12 0.5.5 (2013-02-19)

- fix installation under PyPy (note: DAWG is slow under PyPy and may have bugs).

7.13 0.5.4 (2013-02-14)

- small tweaks for docstrings;
- the extension is rebuilt using Cython 0.18.

7.14 0.5.3 (2013-01-03)

- small improvements to `.compile_replaces` method;
- benchmarks for `.similar_items` method;
- the extension is rebuilt with Cython pre-0.18; this made `.prefixes` and `.iterprefixes` methods faster (up to 6x in some cases).

7.15 0.5.2 (2013-01-02)

- tests are included in source distribution;
- benchmark results in README was nonrepresentative because of my broken (slow) Python 3.2 install;
- installation is fixed under Python 3.x with `LC_ALL=C` (thanks Jakub Wilk).

7.16 0.5.1 (2012-10-11)

- better error reporting while building DAWGs;
- `__contains__` is fixed for keys with zero bytes;
- `dawg.Error` exception class;
- building of `BytesDAWG` and `RecordDAWG` fails instead of producing incorrect results if some of the keys has unsupported characters.

7.17 0.5 (2012-10-08)

The storage scheme of `BytesDAWG` and `RecordDAWG` is changed in this release in order to provide the alphabetical ordering of items.

This is a backwards-incompatible release. In order to read `BytesDAWG` or `RecordDAWG` created with previous versions of DAWG use `payload_separator` constructor argument:

```
>>> BytesDAWG(payload_separator=b'\xff').load('old.dawg')
```

7.18 0.4.1 (2012-10-01)

- Segfaults with empty DAWGs are fixed by updating dawgdic to latest svn.

7.19 0.4 (2012-09-26)

- `iterkeys`, `iteritems` and `iterprefixes` methods (thanks Dan Blanchard).

7.20 0.3.2 (2012-09-24)

- `prefixes` method for finding all prefixes of a given key.

7.21 0.3.1 (2012-09-20)

- bundled dawgdic C++ library is updated to the latest version.

7.22 0.3 (2012-09-13)

- `similar_keys`, `similar_items` and `similar_item_values` methods for more permissive lookups (they may be useful e.g. for umlaut handling);
- `load` method returns self;
- Python 3.3 support.

7.23 0.2 (2012-09-08)

Greatly improved memory usage for DAWGs loaded with `load` method.

There is currently a bug somewhere in a wrapper so DAWGs loaded with `read()` method or unpickled DAWGs uses 3x-4x memory compared to DAWGs loaded with `load()` method. `load()` is fixed in this release but other methods are not.

7.24 0.1 (2012-09-08)

Initial release.