rust-fst Documentation

Release 0.1.2

Johannes Baiter

Jan 21, 2018

Contents

Python bindings for burntsushi's fst crate (rustdocs) for FST-backed sets and maps.

For reasons why you might want to consider using it, see BurntSushi's great article on "Index[ing] 1,600,000,000 Keys with Automata and Rust".

If you want to know more about performance characteristics, memory usage and about the implementation details, please head over to the documentation for the Rust crate.

tl;dr:

- Work with larger-than-memory sets
- Perform fuzzy search using Levenshtein automata

CHAPTER 1

Installation

- You will need:
  - Python >= 3.3, Python or PyPy >= 2.7 with development headers installed
  - Rust nightly (install via rustup)
- Clone the repository. Installation with `pip install git+...` does not work currently
- Run `rustup override add nightly` to add an override for rustup to use the nightly channel for the repository
- Run `python setup.py bdist_wheel` to generate a wheel
- Install the wheel with `pip install dist/rust_fst-0.1-py3-none-any.whl`

CHAPTER 2

Status

The package exposes almost all functionality of the `fst` crate, except for:

- Combining the results of slicing, `search` and `search_re` with set operations
- Using raw transducers

CHAPTER 3

Examples

```
from rust_fst import Map, Set

# Building a set in memory
keys = ["fa", "fo", "fob", "focus", "foo", "food", "foul"]
s = Set.from_iter(keys)

# Fuzzy searches on the set
matches = list(s.search(term="foo", max_dist=1))
assert matches == ["fo", "fob", "foo", "food"]

# Searching with a regular expression
matches = list(s.search_re(r'f\w{2}'))
assert matches == ["fob", "foo"]

# Store map on disk, requiring only constant memory for querying
items = [("bruce", 1), ("clarence", 2), ("stevie", 3)]
m = Map.from_iter(items, path="/tmp/map.fst")

# Find all items whose key is greater or equal (in lexicographical sense) to
# 'clarence'
matches = dict(m['clarence':])
assert matches == {'clarence': 2, 'stevie': 3}

# Create a map from a file input, using generators/yield
# The input file must be sorted on the first column, and look roughly like
#   keyA 123
#   keyB 456
def file_iterator(fpath):
  with open(fpath, 'rt') as fp:
    for line in fp:
      key, value = line.strip().split()
      yield key, int(value)
m = Map.from_iter( file_iterator('/your/input/file/'), '/your/mmapped/output.fst')

# re-open a file you built previously with from_iter()
```

```
m = Map(path='/path/to/existing.fst')
```

CHAPTER 4

API Reference

class rust_fst.**Set**(*path*, *_pointer=None*)

An immutable ordered string set backed by a finite state transducer.

The set can either be constructed in memory or on disk. For large datasets it is recommended to store it on disk, since memory usage will be constant due to the file being memory-mapped.

To build a set, use the *from_iter()* classmethod and pass it an iterator and (optionally) a path where the set should be stored. If the latter is missing, the set will be built in memory.

The interface follows the built-in *set* type, with a few additions:

- Range queries with slicing syntax (i.e. *myset['c':'f']* will return an iterator over all items in the set that start with 'c', 'd' or 'e')
- Performing fuzzy searches on the set bounded by Levenshtein edit distance
- Performing a search with a regular expression

A few caveats must be kept in mind:

- Once constructed, a Set can never be modified.
- Sets must be built with iterators of lexicographically sorted unicode strings

static build(**args*, ***kwds*)

Context manager to build a new set.

Call `insert()` on the returned builder object to insert new items into the set. Keep in mind that insertion must happen in lexicographical order, otherwise an exception will be thrown.

Parameters **path** – Path to build set in, or *None* if set should be built in memory

Returns `SetBuilder`

classmethod from_iter(*it*, *path=None*)

Build a new set from an iterator.

Keep in mind that the iterator must return unicode strings in lexicographical order, otherwise an exception will be thrown.

Parameters

- **it** (*iterator over unicode strings*) – Iterator to build set with
- **path** – Path to build set in, or *None* if set should be built in memory

Returns The finished set

Return type *Set*

__init__(*path*, *_pointer=None*)
Load a set from a given file.

Parameters **path** – Path to set on disk

__contains__(*val*)
Check if the set contains the value.

__iter__()
Get an iterator over all keys in the set in lexicographical order.

__len__()
Get the number of keys in the set.

__getitem__(*s*)
Get an iterator over a range of set contents.

Start and stop indices of the slice must be unicode strings.

Important: Slicing follows the semantics for numerical indices, i.e. the *stop* value is **exclusive**. For example, given the set *s = Set.from_iter(["bar", "baz", "foo", "moo"])*, *s['b': 'f']* will only return *"bar"* and *"baz"*.

Parameters **s** (slice) – A slice that specifies the range of the set to retrieve

union(**others*)
Get an iterator over the keys in the union of this set and others.

Parameters **others** – List of *Set* objects

Returns Iterator over all keys in all sets in lexicographical order

intersection(**others*)

Get an iterator over the keys in the intersection of this set and others.

Parameters **others** – List of *Set* objects

Returns Iterator over all keys that exists in all of the passed sets in lexicographical order

difference(**others*)

Get an iterator over the keys in the difference of this set and others.

Parameters **others** – List of *Set* objects

Returns Iterator over all keys that exists in this set, but in none of the other sets, in lexicographical order

symmetric_difference(**others*)

Get an iterator over the keys in the symmetric difference of this set and others.

Parameters **others** – List of *Set* objects

Returns Iterator over all keys that exists in only one of the sets in lexicographical order

issubset(*other*)
Check if this set is a subset of another set.

Parameters **other** (*Set*) – Another set

Return type bool

issuperset(*other*)
Check if this set is a superset of another set.

Parameters **other** (*Set*) – Another set

Return type bool

isdisjoint(*other*)
Check if this set is disjoint to another set.

Parameters **other** (*Set*) – Another set

Return type bool

search_re(*pattern*)
Search the set with a regular expression.

Note that the regular expression syntax is not Python's, but the one supported by the *regex* Rust crate, which is almost identical to the engine of the RE2 engine.

For a documentation of the syntax, see: http://doc.rust-lang.org/regex/regex/index.html#syntax

Due to limitations of the underlying FST, only a subset of this syntax is supported. Most notably absent are:

- Lazy quantifiers (`r'*?'`, `r'+?'`)
- Word boundaries (`r'\b'`)
- Other zero-width assertions (`r'^'`, `r'$'`)

For background on these limitations, consult the documentation of the Rust crate: http://burntsushi.net/rustdoc/fst/struct.Regex.html

Parameters **pattern** – A regular expression

Returns An iterator over all matching keys in the set

Return type `KeyStreamIterator`

search(*term*, *max_dist*)
Search the set with a Levenshtein automaton.

Parameters

- **term** – The search term
- **max_dist** – The maximum edit distance for search results

Returns Iterator over matching values in the set

Return type `KeyStreamIterator`

class rust_fst.**Map**(*path=None*, *_pointer=None*)

An immutable map of unicode keys to unsigned integer values backed by a finite state transducer.

The map can either be constructed in memory or on disk. For large datasets it is recommended to store it on disk, since memory usage will be constant due to the file being memory-mapped.

To build a map, use the *from_iter()* classmethod and pass it an iterator and (optionally) a path where the map should be stored. If the latter is missing, the map will be built in memory.

In addition to querying the map for single keys, the following operations are supported:

- Range queries with slicing syntax (i.e. *myset['c':'f']* will return an iterator over all items in the map whose keys start with 'c', 'd' or 'e')
- Performing fuzzy searches on the map keys bounded by Levenshtein edit distance
- Performing a search on the map keys with a regular expression
- Performing set operations on multiple maps, e.g. to find different values for common keys

A few caveats must be kept in mind:

- Once constructed, a Map can never be modified.
- Maps must be built with iterators of lexicographically sorted (str/unicode, int) tuples, where the integer value must be positive.

static build(**args*, ***kwds*)
Context manager to build a new map.

Call `insert()` on the returned builder object to insert new items into the mapp. Keep in mind that insertion must happen in lexicographical order, otherwise an exception will be thrown.

Parameters **path** – Path to build mapp in, or *None* if set should be built in memory

Returns `MapBuilder`

classmethod from_iter(*it*, *path=None*)
Build a new map from an iterator.

Keep in mind that the iterator must return lexicographically sorted (key, value) pairs, where the keys are unicode strings and the values unsigned integers.

Parameters

- **it** (*iterator over (str/unicode, int) pairs, where int >= 0*) – Iterator to build map with
- **path** – Path to build map in, or *None* if set should be built in memory

Returns The finished map

Return type *Map*

__init__(*path=None*, *_pointer=None*)
Load a map from a given file.

Parameters **path** – Path to map on disk

__getitem__(*key*)
Get the value for a key or a range of (key, value) pairs.

If the key is a slice object (e.g. *mymap['a':'f']*) an iterator over all matching items in the map will be returned.

Important: Slicing follows the semantics for numerical indices, i.e. the *stop* value is **exclusive**. For example, *mymap['a':'c']* will return items whose key begins with 'a' or 'b', but **not** 'c'.

Parameters **key** – The key to retrieve the value for or a range of unicode strings

Returns The value or an iterator over matching items

keys()
Get an iterator over all keys in the map.

values()
Get an iterator over all values in the map.

items()
Get an iterator over all (key, value) pairs in the map.

search_re(*pattern*)
Search the map with a regular expression.

Note that the regular expression syntax is not Python's, but the one supported by the *regex* Rust crate, which is almost identical to the engine of the RE2 engine.

For a documentation of the syntax, see: http://doc.rust-lang.org/regex/regex/index.html#syntax

Due to limitations of the underlying FST, only a subset of this syntax is supported. Most notably absent are:

- Lazy quantifiers (r'*?', r'+?')
- Word boundaries (r'')
- Other zero-width assertions (r'^', r'$')

For background on these limitations, consult the documentation of the Rust crate: http://burntsushi.net/rustdoc/fst/struct.Regex.html

Parameters **pattern** – A regular expression

Returns An iterator over all items with matching keys in the set

Return type `MapItemStreamIterator`

search(*term*, *max_dist*)
Search the map with a Levenshtein automaton.

Parameters

- **term** – The search term
- **max_dist** – The maximum edit distance for search results

Returns Matching (key, value) items in the map

Return type `MapItemStreamIterator`

union(**others*)
Get an iterator over the items in the union of this map and others.

The iterator will return pairs of *(key, [IndexedValue])*, where the latter is a list of different values for the key in the different maps, represented as a tuple of the map index and the value in the map.

Parameters **others** – List of *Map* objects

Returns Iterator over all items in all maps in lexicographical order

intersection(**others*)

Get an iterator over the items in the intersection of this map and others.

The iterator will return pairs of *(key, [IndexedValue])*, where the latter is a list of different values for the key in the different maps, represented as a tuple of the map index and the value in the map.

Parameters **others** – List of *Map* objects

Returns Iterator over all items whose key exists in all of the passed maps in lexicographical order

difference(**others*)

Get an iterator over the items in the difference of this map and others.

The iterator will return pairs of *(key, [IndexedValue])*, where the latter is a list of different values for the key in the different maps, represented as a tuple of the map index and the value in the map.

Parameters **others** – List of *Map* objects

Returns Iterator over all items whose key exists in this map, but in none of the other maps, in lexicographical order

symmetric_difference(**others*)

Get an iterator over the items in the symmetric difference of this map and others.

The iterator will return pairs of *(key, [IndexedValue])*, where the latter is a list of different values for the key in the different maps, represented as a tuple of the map index and the value in the map.

Parameters **others** – List of *Map* objects

Returns Iterator over all items whose key exists in only one of the maps in lexicographical order

Index

Symbols

B

D

F

I

K

M

S

U

V