trie

Jeroen F.J. Laros

Nov 05, 2022

CONTENTS:

This library provides a generic trie implementation for small alphabets, the structure of the leaf nodes can be specified by the user.

Apart from the basic operations, a generator is provided for easy iteration over all words stored in the trie and a number of functions for *approximate matching* are implemented.

Please see ReadTheDocs for the latest documentation.

CHAPTER

ONE

INSTALLATION

1.1 Latest release

Navigate to the latest release and either download the `.zip` or the `.tar.gz` file and unpack the downloaded archive.

1.2 From source

The source is hosted on GitHub, use the following command to install the latest development version.

```
git clone https://github.com/jfjlaros/trie.git
```

CHAPTER

TWO

USAGE

Include the header file to use the trie library.

```
#include "trie.tcc"
```

The library provides the `Trie` class, which takes two template arguments, the first of which determines the alphabet size, the second determines the type of the leaf.

```
Trie<4, Leaf> trie;
```

```
vector<uint8_t> word = {0, 1, 2, 3};
trie.add(word);
```

```
Node<4, Leaf>* node = trie.find(word);
```

```
trie.remove(word);
```

```
for (Result<Leaf> result: trie.walk()) {
  // result.leaf : Leaf node.
  // result.path : Word leading up to the leaf.
}
```

```
for (Result<Leaf> result: trie.hamming(word, 1)) {
  // result.leaf : Leaf node.
  // result.path : Word leading up to the leaf.
}
```

```
struct MyLeaf : Leaf {
  vector<size_t> lines;
}
```

```
size_t line = 0;
for (vector<uint8_t> word: words) {
  MyLeaf* leaf = trie.add(word);
  leaf->lines.push_back(line++);
}
```

CHAPTER

THREE

API DOCUMENTATION

3.1 Node

```
#include "node.tcc"
```

3.1.1 Class definition

template<uint8_t **alphabetSize**, class **T**>

class **Node**

Node.

Template Parameters

- **alphabetSize** – Size of the alphabet.
- **T** – *Leaf* type.

Public Functions

bool **isEmpty**() const

Check whether a node neither has any children, nor a leaf.

Returns

True is the node is empty, false otherwise.

Public Members

array<*Node**, *alphabetSize*> **child** = {}

Children.

T ***leaf** = {nullptr}

Leaf.

3.2 Leaf

3.2.1 Class definition

struct **Leaf**

Leaf.

Public Members

size_t **count** = {0}

Counter.

3.3 Trie

```
#include "trie.tcc"
```

3.3.1 Class definition

template<uint8_t **alphabetSize**, class **T**>

class **Trie**

Trie.

Template Parameters

- **alphabetSize** – Size of the alphabet.
- **T** – *Leaf* type.

Public Functions

T ***add**(vector<uint8_t> const&) const

Add a word.

Parameters
word – Word.

Returns
Leaf.

void **remove**(vector<uint8_t> const&) const

Remove a word.

Parameters
word – Word.

Node<*alphabetSize*, *T*> ***find**(vector<uint8_t> const&) const

Find a word.

Parameters
word – Word.

Returns
node if found, `nullptr` otherwise.

generator<Result<*T*>> **walk**() const

Traverse.

Returns
Traversal results.

generator<Result<*T*>> **hamming**(vector<uint8_t> const&, int const) const

Hamming.

Parameters

- **word** – Word.
- **distance** – Maximum distance.

Returns
Traversal results.

generator<Result<*T*>> **asymmetricHamming**(vector<uint8_t> const&, int const) const

Asymmetric Hamming.

Parameters

- **word** – Word.
- **distance** – Maximum distance.

Returns
Traversal results.

generator<Result<*T*>> **levenshtein**(vector<uint8_t> const&, int const) const

Levenshtein.

Parameters

- **word** – Word.
- **distance** – Maximum distance.

Returns
Traversal results.

generator<Result<*T*>> **asymmetricLevenshtein**(vector<uint8_t> const&, int const) const

Asymmetric Levenshtein.

Parameters

- **word** – Word.
- **distance** – Maximum distance.

Returns
Traversal results.

CHAPTER

FOUR

CONTRIBUTORS

- Jeroen F.J. Laros <jlaros@fixedpoint.nl> (Original author, maintainer)

Find out who contributed:

```
git shortlog -s -e
```

INDEX