Ветка 4t1p_SuffixTree

Ветка создана для разработки библиотеки `stringalgo` - функционала для работы со строками при помощи суффиксного дерева.

1. **Директория 'suffix_tree_internals_no style'.** Содержит файлы с исходным кодом библиотеки, к которому еще не был применен google-style. Самая первая версия файлов библиотеки.
   - **Поддиректория 'tested_UVA_10679'.** Содержит ссылку на задачу в тестирующей системе, в решении которой можно использовать суффиксное дерево; файлы с исходным кодом решения этой задачи, а также подтверждение прохождения им тестов.
2. **Директория 'suffix_tree_google_style'.** Содержит файлы с исходным кодом библиотеки, к которому в той или иной степени применен google-style.
   - **Поддиректория 'google-style_no-naming'.** К коду применены все правила google-style кроме правил наименований и форматирования. Вторая версия файлов библиотеки.
   - **Поддиректория 'google-style_complete'.** К коду применены все правила google-style. Третья и последняя версия файлов библиотеки.
3. **Директория 'libstringalgo'.** Содержит архивы последних версий файлов библиотеки, которые доступны пользователю для скачивания, встраивания и использования.

Библиотека `stringalgo`

Библиотека представляет из себя реализацию суффиксного дерева, пригодную для прикладного использования, а также набор функций для работы со строками.

В файле `suffix_tree.h` объявлен класс `SuffixTree`, который представляет из себя реализацию суффиксного дерева. В файле `find_occurrences.h` объявлена функция `FindAllOccurrences`, которая находит все вхождения строки-образца в строку-текст.

Описание функционала

Класс `SuffixTree`

Объект класса `SuffixTree` может быть сконструирован 3-мя способами:

1. Конструктор по умолчанию. Создает суффиксное дерево для пустой строки:

   ```
   SuffixTree()
   ```

2. Конструктор копирования. Полностью копирует суффиксное дерево `suffix_tree`:

   ```
   SuffixTree(const SuffixTree& suffix_tree)
   ```

3. Конструктор, принимающий строку в качестве аргумента. Строит суффиксное дерево по строке `sample`, добавляя в конец символ конца строки (символ, отличный от любого, содержащегося в `sample`):

   ```
   SuffixTree(const std::string& sample)
   ```

Далее считаем, что суффиксное дерево построено по строке `sample`.

Структура `Link`

Структура `SuffixTree::Link` представляет собой ориентированное ребро суффиксного дерева, которое задается следующим образом:

- `target_node_index` - вершина суффиксного дерева, в которую ведет это ребро;
- `sample_start_index`, `sample_end_index` - индексы, задающие подстроку строки `sample`, являющуюся меткой ребра (см. Бор). Индекс `sample_start_index` - индекс первого символа подстроки, а индекс `sample_end_index` - индекс последнего символа подстроки, увеличенный на 1. Таким образом, метка ребра определяется как `sample [ sample_start_index, sample_end_index )`.

В частном случае, если `sample_start_index == -1`, то структура представляет из себя суффиксную ссылку дерева.

Структура `DFSChooseNextNeighbourResult`

Объекты структуры `DFSChooseNextNeighbourResult` возвращаются методом `ChooseNextNeighbour` класса `TravelVisitor`, объект которого передается в качестве аргумента методу `SuffixTree::DepthFirstSearchTraversal`, для выбора следующего ребра в обходе суффиксного дерева. Структура состоит из двух полей:

- `use_suffix_link` - равно `true`, если следующее ребро, по которому следует совершить переход в обходе дерева - это суффиксная ссылка.
- `chosen_neighbour` - `const_interator` класса `std::map<char, Link>` на исходящее из активной вершины ребро, по которому следует перейти.

Метод `AppendSample`

Прототип:

```
void AppendSample(const std::string& append_sample)
```

Описание: дописывает к строке `sample` строку `append_sample` и достраивает суффиксное дерево, учитывая добавленные символы. Этот метод следует использовать только для деревьев, построенных с помощью конструктора по умолчанию.

Сложность: O(длина append_sample * log(размер нового алфавита)).

Метод `GetLinkIterator`

Прототип:

```
LinkMapConstIterator GetLinkIterator(int node_index, char letter) const
```

Описание: возвращает `const_interator` класса `std::map<char, Link>` на исходящее из вершины `node_index` ребро, метка которого начинается с символа `letter`. Функцию следует использовать в методе `ChooseNextNeighbour` класса `TraversalVisitor`, объект которого передается в качестве аргумента методу `SuffixTree::DepthFirstSearchTraversal`, для выбора следующего ребра в обходе суффиксного дерева.

Сложность: O(log(размер алфавита)).

Метод `IsLeaf`

Прототип:

```
bool IsLeaf(int node_index) const
```

Описание: проверяет, является ли вершина `node_index` листом, т.е. имеет ли она исходящие ребра.

Сложность: O(1).

Метод `sample`

Прототип:

```
const std::string& sample() const
```

Описание: возвращает константную ссылку на строку `sample`, по которой построено суффиксное дерево.

Сложность: O(1).

Метод `non_existing_char`

Прототип:

```
char non_existing_char() const
```

Описание: возвращает символ, который не встречается в строке `sample` и используется суффиксным деревом как символ конца слова.

Сложность: O(1).

Метод `Size`

Прототип:

```
size_t Size() const
```

Описание: возвращает количество вершин в суффиксном дереве.

Сложность: O(1).

Метод `DepthFirstSearchTraversal`

Прототип:

```
template <typename TraversalVisitor> void DepthFirstSearchTraversal(TraversalVisitor* visitor) const
```

Описание: выполняет обход суффиксного дерева, вызывая методы класса `TraversalVisitor` при различных событиях. При обходе суффиксного дерева определены следующие события:

- начало обхода,
- посещение некой вершины в первый раз (переход в вершину от ее родителя),
- возвращение в вершину из потомка,
- выбор ребра для следующего перехода,
- переход по ребру к потомку,
- последний выход из вершины, т.е. выход из вершины без последующего в нее возвращения.

`TraversalVisitor` - класс, у которого должны быть определены следующие методы:

1. `void InitVisitor()`. Вызывается перед началом обхода.

2. `void DiscoverNode(const SuffixTree::Link& in_link)`. Вызывается при посещении некоторой вершины `node` в первый раз (т.е. при переходе от ее родителя `parent`). `in_link` - ребро, по которому перешли в вершину `node` из ее родителя `parent`.

3. `void ReturnToNode(const SuffixTree::Link& return_link, const SuffixTree::Link& in_link)`. Вызывается при возвращении в некоторую вершину `node` из потомка `son`. `return_link` - ребро из вершины `node` в потомка `son`. `in_link` - ребро из родителя вершины `node` в вершину `node`. Если метод вызван для ребра, являющегося суффиксной ссылкой, то аргумент `return_link` невалиден и не должен использоваться.

4. `void ExamineEdge(const SuffixTree::Link& link)`. Вызывается перед переходом по ребру от родителя к потомку. `link` - ребро, по которому совершается переход.

5. `DFSChooseNextNeighbourResult ChooseNextNeighbour(int active_node, const LinkMapConstIterator& link_map_begin_it, const LinkMapConstIterator& link_map_next_letter_it, const LinkMapConstIterator& link_map_end_it, int suffix_link)`. Вызывается при выборе ребра, по которому следует перейти из текущей вершины `active_node`. Пусть `old_link` - это последнее ребро, по которому был совершен переход из вершины `active_node` в потомка `active_node`. Если ранее таких переходов из вершины `active_node` сделано не было, то `old_link` не определено. `active_node` - текущая вершина в обходе суффиксного дерева. `link_map_begin_it` - `const_interator` класса `std::map<char, Link>` на исходящее из вершины `active_node` ребро, метка которого лексикографически меньше меток всех исходящих из вершины `active_node` ребер. `link_map_next_letter_it` - `const_interator` класса `std::map<char, Link>` на исходящее из вершины `active_node` ребро:

    - которое совпадает с `link_map_begin_it`, если посещение вершины `active_node` произошло в первый раз;
    - которое совпадает с `link_map_end_it`, если `old_link` определено и его метка является лексикографически наибольшей среди меток всех исходящих из `active_node` ребер (лексикографически следующего ребра из этой вершины не существует).
    - метка которого лексикографически следует за меткой `old_link` среди меток всех исходящих из `active_node` ребер, в ином случае. (Говорим, что строка `s1` лексикографически следует за строкой `s2` среди строк множества `S`, когда строка `s2` находится сразу после строки `s1` в массиве всех строк из `S`, отсортированном лексикографически).

6. `void FinishNode(const SuffixTree::Link& in_link)`. Вызывается перед последним выходом из вершины `node`. `in_link` - ребро из родителя вершины `node` в вершину `node`.

Сложность: зависит от поведения метода `TraversalVisitor::ChooseNextNeighbour`. При реализации этого метода для обычного обхода графа - O(длина sample).

Функция `FindAllOccurrences`

Прототип:

```
std::vector<int> FindAllOccurrences(const SuffixTree& suffix_tree, const std::string& search_string);
```

Описание: находит все вхождения строки `search_string` в строке `suffix_tree.sample()`, по которой построено суффиксное дерево `suffix_tree`. При этом строка, по которой построено `suffix_tree`, должна завершаться символом конца строки. Возвращает вектор индексов строки `suffix_tree.sample()`, задающих начало вхождений `search_string` в нее.

Встраивание

Для пользователей Linux:

1. Скачать архив 'libstringalgo_linux.zip' из директории 'libstringalgo' или по ссылке.

2. В терминале перейти в директорию, в которую был скачан архив, разархивировать файл 'libstringalgo_linux.zip':

    ```
    $ unzip libstringalgo_linux.zip
    ```

3. Во всех файлах проекта, где предполагается использование функционала библиотеки, подключить соответствующие header-файлы библиотеки:

    ```
    #include "suffix_tree.h"
    #include "find_occurrences.h"
    ```

4. Подключить библиотеку в IDE:

    - *Для Qt Creator.* В окне проекта ПКМ по названию проекта -> Add Library... -> External Library -> В поле 'Library file' вписать путь до файла 'libstringalgo.a' из архива, в поле 'Include path' вписать путь до директории, в которой лежит этот файл. Также добавить в файл проекта (.pro) строку:

        ```
        CONFIG += c++11
        ```

    - *Для других IDE.* Компилировать файлы проекта из терминала:

        ```
        $ g++ -std=c++11 -I ${PATH_TO_LIBRARY}/libstringalgo/ ${FILENAME}.cpp -L ${PATH_TO_LIBRARY}/libstringalgo/ -l stringalgo
        ```

Для пользователей Windows:

1. Скачать архив 'libstringalgo_win.zip' из директории 'libstringalgo' или по ссылке.

2. Разархивировать файл 'libstringalgo_win.zip' в директорию того проекта, где предполагается использование библиотеки.

3. В настройках IDE включить поддержку C++11. Добавить файлы библиотеки в список файлов проекта. Во всех файлах проекта, где предполагается использование функционала библиотеки, подключить соответствующие

header-файлы библиотеки:

```
#include "libstringalgo/suffix_tree.h"
#include "libstringalgo/find_occurrences.h"
```