Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Информационная безопасность» (ИУ8)

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

СЕМЕСТРОВЫЙ ПРОЕКТ

Алгоритм проверки орфографии с помощью БК-дерева

Вариант 22

Руководитель проекта: _____ / Чесноков В.О.
(подпись, дата)

Разработчик проекта: _____ / Кустов И.А.
(подпись, дата)

Москва, 2018

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Деревья BK-Trees или Burkhard-Keller - это древовидная структура данных, разработанная для быстрого поиска близкого совпадения со строкой, например, как используется средством проверки орфографии, или при выполнении «нечеткого» поиска термина. Цель состоит в том, чтобы находилось «seek» и «peek», если ищется «aeek». БК-деревья упрощают решение проблемы, у которой нет очевидного решения, кроме поиска грубой силой.

БК-деревья были впервые предложены Буркхардом и Келлером в 1973 году в своей работе «Some approaches to best match file searching». Единственная копия этого документа находится в архиве ACM, который является только подпиской. Дополнительные детали представлены в документе «Fast Approximate String Matching in a Dictionary».

ФОРМАЛЬНАЯ ПОСТАНОВКА ЗАДАЧИ

Необходимо написать программу, реализующую исправление орфографических ошибок на основе исходного текста и словаря языка, на котором написан текст. Программа должна заносить словарь в BK-дерево, по которому потом происходит поиск слов. Если слово найдено, оно перезаписывается в новый файл. В случае, если слово не найдено точным поиском, происходит неточный поиск, в процессе которого мы получаем список похожих слов, из которого ищется наиболее похожее на исходное. Все встречаемые символы перезаписываются в новый файл без изменений.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Основная информация об алгоритме

Деревья Буркхарда Келлера являются метрическими деревьями, алгоритмы построения таких деревьев основаны на свойстве метрики отвечать неравенству треугольника. Это свойство позволяет метрикам образовывать метрические пространства произвольной размерности. На основании этого свойства можно построить структуру данных, позволяющую осуществлять поиск в таком метрическом пространстве, которой и являются деревья Буркхарда Келлера. Для построения дерева выбирается любая строка из набора строк и устанавливается корнем дерева, а каждая следующая строка добавляется относительно расстояния от корня дерева. У любого узла могут быть потомки только с различным расстоянием до этого узла. Если потомок с таким расстоянием уже есть, то строка сравнивается с этим потомком и становится его «ребенком».

Например, пусть дан набор слов: «Корень», «Голень», «Корпус», «Колдун», «Камень», «Король», «Корова», «Костюм». Необходимо построить BK-дерево. Для этого нужно взять любое слово, например, «Корень», и сделать его корневым узлом дерева. Расстояние Хэмминга между «Корень» и «Голень» равно 2, поэтому этот узел добавится с пометкой 2 на дуге. Аналогично и для слов «Корпус» и «Колдун». Эти слова также добавляются непосредственно к корню, т. к. расстояние отличается от 2. Расстояние Хэмминга между словом «Камень» и корнем дерева равно 2. Узел с таким расстоянием уже есть («Голень»), поэтому добавление слова «Камень» происходит к слову «Голень». Аналогично добавляются и остальные слова (рисунок 1).

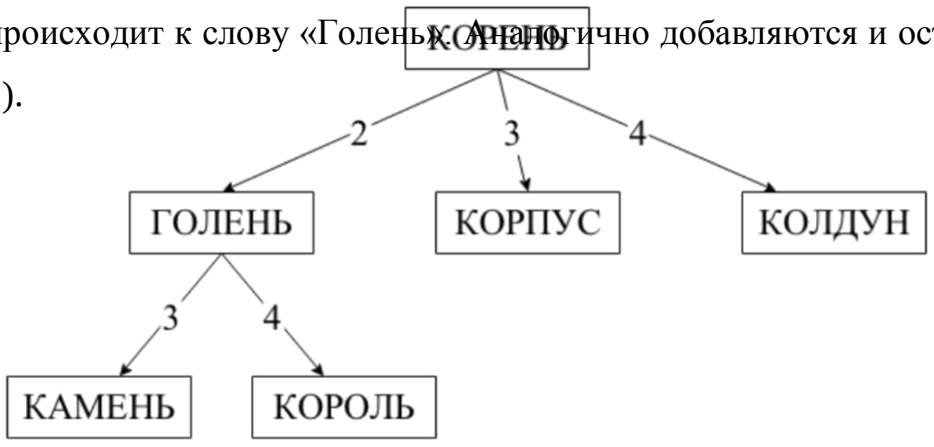

Рисунок 1. BK-tree

Расстояние между строками

При выборе алгоритма сравнения строк, рассматривались 3 алгоритма:

- Расстояние Левенштейна
- Расстояние Дамерау-Левенштейна
- Расстояние Хемминга
-

Расстояние Левенштейна

Расстояние Левенштейна между двумя строками в теории информации и компьютерной лингвистике — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна и его обобщения активно применяется:

- для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи).
- для сравнения текстовых файлов утилитой diff и ей подобными. Здесь роль «символов» играют строки, а роль «строк» — файлы.

С в биоинформатике для сравнения генов, хромосом и белков.

С точки зрения приложений определение расстояния между словами или текстовыми полями по Левенштейну обладает следующими недостатками:

- При перестановке местами слов или частей слов

получаются сравнительно большие расстояния;

- Расстояния между совершенно разными короткими словами оказываются небольшими, в то время как расстояния между очень похожими длинными словами оказываются значительными.

Пусть S1 и S2 — две строки (M и N соответственно) над некоторым алфавитом, тогда редакционное расстояние (расстояние Левенштейна) d(S1, S2) можно подсчитать по следующей рекуррентной формуле d(S1, S2) = D(M, N), где

$$D(i,j) = \begin{cases} 0, & i=0,\ j=0 \\ i, & j=0,\ i>0 \\ j, & i=0,\ j>0 \\ \min\{ & \\ \quad D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & j>0,\ i>0 \\ \quad D(i-1,j-1)+\mathrm{m}(S_1[i],S_2[j]) & \\ \} & \end{cases},$$

Здесь шаг по i символизирует удаление (D) из первой строки, по j — вставку (I) в первую строку, а шаг по обоим индексам символизирует замену символа или отсутствие изменений.

Очевидно, справедливы следующие утверждения:

- $\mathrm{d}(S_1,S_2) \geqslant \big||S_1|-|S_2|\big|$
- $\mathrm{d}(S_1,S_2) \leqslant \max\big(|S_1|,|S_2|\big)$
- $\mathrm{d}(S_1,S_2)=0 \Leftrightarrow S_1=S_2$

Расстояние Дамерау-Левенштейна

Расстояние Дамерау — Левенштейна является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Формула:

$$d_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \\ d_{a,b}(i-2,j-2)+1 \end{cases} & \text{if } i,j > 1 \text{ and } a_i = b_{j-1} \text{ and } a_{i-1} = b_j \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise,} \end{cases}$$

Расстояние Хемминга

Число позиций, в которых соответствующие символы двух слов одинаковой длины различны.

Выбор расстояния

Расстояние Дамерау-Левенштейна рассматривает большее количество возможных ошибок, что повышает точность поиска.

Алгоритм Вагрена-Фишера

Искомое расстояние формируется через вспомогательную функцию D(M,N), находящую редакционное расстояние для подстрок $S_1[0..M]$ и $S_2[0..N]$. Тогда полное редакционное расстояние будет равно расстоянию для подстрок полной длины: $d(S_1,S_2) = D_{S1},S_2(M,N)$.

Самоочевидным фактом, является то, что:

D(0,0) = 0.

Действительно, пустые строки и так совпадают. Также, ясны значения для:

D(i, 0) = i;

D(0, j) = j.

Действительно, любая строка может получиться из пустой, добавлением нужного количества нужных символов, любые другие операции будут только

увеличивать оценку. В общем случае чуть сложнее:

D(i,j) = D(i-1,j-1), если S_1[i] = S_2[j], иначе

D(i,j) = min(D(i-1,j), D(i,j-1), D(i-1,j-1)) + 1.

В данном случае, мы выбираем, что выгоднее: удалить символ (D(i-1,j)), добавить (D(i,j-1)), или заменить (D(i-1,j-1)).

Нетрудно понять, что алгоритму получения оценки не требуется памяти больше чем два столбца, текущий (D(i,*)) и предыдущий (D(i-1,*)). Однако в полном объеме матрица нужна для восстановления редакционного предписания. Начиная из правого нижнего угла матрицы (M,N) мы идем в левый верхний, на каждом шаге ища минимальное из четырех значений:

- удаление символа
- добавление символа
- смена местами соседних символов
- замена одного символа на другой

Помимо данного алгоритма существует алгоритм Хиршберга. Он заключается в разбиении матрицы на 2 части и их отдельный подсчет. Это позволяет распараллелить задачу поиска расстояния, однако на словах маленькой длины это не дает практического выигрыша во времени. По этой причине в программе используется алгоритм Вагнера-Фишера.

ВЫБОР СТРУКТУР ДАННЫХ

В программе использовались 3 структуры данных: map, queue.

Для каждого узла необходимо хранить список пар «расстояние между словами», «указатель на узел слова». Так как список мог состоять из нескольких элементов, необходимо было совершать быстрый поиск по расстоянию между словами. Map хорошо подходит для решения данной проблемы.

При неточном поиске необходимо получить список похожих слов и далее отсеивать их по очереди. Для этого подходят queue и stack. Было решено использовать queue, из-за его свойства FIFO.

ТРЕБОВАНИЯ К ПРОЕКТУ

- Программа должна быть платформонезависимой, не иметь зависимостей от нестандартных библиотек и выполнена в виде консольного приложения.
- Программа принимает входные данные в виде одного или нескольких текстовых файлов и записывает результат работы в текстовый файл. Имена входных и выходных файлов задаются через аргументы командной строки. Программа не интерактивная, пользовательский ввод не предусмотрен.
- Программный код должен быть достойного качества, отформатирован выполнен в едином стиле.
- Реализация алгоритма должна быть инкапсулирована.
- Программа должна быть покрыта тестами. Тесты должны содержать проверку корректности всех основных реализованных алгоритмов. Каждый тест представляет собой два текстовых файла с одинаковым именем, но разным расширением (например, 001.dat и 001.ans) в формате входных и выходных данных соответственно.

ИНСТРУКЦИЯ

Программа содержит CMakeList файл, поэтому собрать можно с помощью cmake, выгрузив все необходимые файлы с репозитория.

При запуске программы необходимо передать аргументами 3 переменные. 1 – Путь до файла, который надо проверить. 2 – Путь до файла, в котором лежит словарь для данного текста. 3 – имя файла, в который будет сохраняться исправленный текст.

ПЛАН РЕШЕНИЯ ЗАДАЧИ

1. Реализация алгоритма Вагнера-Фишера для расстояния Дамерау-Левенштейна;
2. Написание структуры данных, в которой будет храниться словарь;
3. Реализация функции поиска слова с определенным количеством возможного количества ошибок;
4. Реализация функции считывания словаря из текстового файла;
5. Реализация функций считывания из текстового файла данных, поиска слов в тексте файла и записи результата в выходной текстовый файл.

ЛОГИКА ПРОГРАММЫ

Программа состоит из одного класса BK_tree, реализующего дерево хранения словаря, и ряда вспомогательных функций(correct_file, correct_string), направленных на парсинг входного файла и создание выходного.

Программа, получив 3 необходимых параметра, первым делом инициализирует дерево словаря. Утверждается, что в словаре повторений нет, поэтому проверки на это не осуществляется. Слово пробегает по дереву вниз, пока не будет найдено его место. После инициализации, открываются файл чтения и записи. Парсится файл чтения таким образом, чтобы исправлялись слова, а символы перезаписывались без изменений. Каждое слово ищется в словаре через функцию find_with_mistakes, которая сначала ищет слово без исправлений, а если не находит, то рекурсивно ищет по веткам. Кроме того, когда найдены несколько слов, мы ищем минимальное по кол-ву ошибок и одновременно, наиболее близкое по размеру к проверяемому.

Проверенные строки записывают в выходной файл, после записи последней проверенной строки, программа прекращает работу.

ФОРМАТ ВХОДНЫХ И ВЫХОДНЫХ ДАННЫХ

Программа принимает на вход 2 файла и имя будущего файла, в который будет сохранен результат. Первый файл представляет из себя текст, который необходимо отредактировать. Второй файл – библиотека, в которой каждое слово записано с новой строки. После редактирования текста, он записывается в файл с именем, который указывается третьим аргументом. Файлы на входе и выходе имеют формат .txt.

СЛОЖНОТЬ И ИСПОЛЬЗОВАНИЕ ПАМЯТИ АЛГОРИТМАМИ

n- колво слов в дереве.

k – длина добавляемого слова.

damerau_levenshtein

Сложность O(k^2). Сложность квадратная, так как необходимо заполнить всю квадратную матрицу для нахождения результата

Память O(k), так как в алгоритме используется матрица 3 на k, в 3 столбца перезаписываются по mod3

add

Функция добавляет слово не в двоичное дерево, следовательно, сложность логарифмическая. Основания логарифма – среднее кол-во детей у узла, а это средняя длина слова в данном языке -1 (слово не может быть пустым). Например, в английском языке средняя длина слова 5. Следовательно логарифм будет по основанию 4. Так же на каждом шагу выполняется поиск расстояния между словами. Так как k = 5 = const для определенного алфавита, то k можно пренебречь. Следовательно, сложность O($\log_4 n$)

Память O(k), так как используется только константная память + память на поиск расстояния между словами, которая уничтожается после выхода из функции его поиска.

find

Сложность O($\log_4 n$) и память O(k) высчитывается аналогично функции добавления элемента в дерево, так как функции сами по себе очень похожи.

init

Функция инициализации дерева словарем, который передан вторым аргументом в программе. Сложность O($n\log_4 n$), так как для каждого слова из n вызывается функция add.

Память O(1), так как хранится лишь строка и открытый файл

find_in_branch

Рекурсивная функция. Расхождение зависит от количества допустимых ошибок. В нашем случае по умолчанию установлена 1 ошибка. Следовательно, на каждом узле, он вызовет 3 рекурсии + функцию расстояния слов O($3^{\log_4 n}$).
С памятью аналогично, только при поиске расстояния память удаляется. Следовательно O($3^{\log_4 n}$).

corrected_word

Функция вызывает find_in_branch, а затем проверяет линейно очередь, которую вернула функция, так как эта линейная сложность крайне мала, то сложность O($3^{\log_4 n}$). Аналогично с памятью. Использование памяти O($3^{\log_4 n}$).

find_with_mistakes

Функция всегда вызывает find и, если слово написано с ошибкой, то corrected_word. Следовательно, сложность O($3^{\log_4 n}$), в лучшем случае O($\log_4 n$), если слово находится в словаре. Используемая память O($3^{\log_4 n}$), в лучшем случае O(k).

correct_string

Функция парсит строку, следовательно, работает за O(m), где m – колво символов в строке. Однако, для каждого найденного слова вызывается функция find_with_mistakes. Так как в среднем слово 5 символов, то m/5 вызовов функции. O($\frac{m}{5} * 3^{\log_4 n}$)= O($m3^{\log_4 n}$). Используемая память O($3^{\log_4 n}$)

correct_file

Функция считывает строки из файла и вызывает для них correct_string, а затем результат записывается. Всего символов в файле q. Итого получается, что сложность функции O($q3^{\log_4 n}$). Используемая память O($3^{\log_4 n}$).

Итоговая сложность программы состоит из сложности инициализации дерева и сложности корректировки файла. O($n\log_4 n$)+ O($q3^{\log_4 n}$) = O($q3^{\log_4 n}$).
Итоговая используемая память аналогично равна O(1)+ O($3^{\log_4 n}$)= O($3^{\log_4 n}$).

P.S. При поиске с ошибками алгоритм взял с официального документа тех, кто разработал данный алгоритм

ОПИСАНИЕ ТЕСТОВ

В тестах 2 файла были для чтения и один на выход, который имеет расширение .ans.

В первом тесте продемонстрирована работа на английском словаре в 2000 слов. Программа корректно считала словарь, пропарсила текст и исправила его, записав в новый файл.

Во втором тесте были добавлены знаки препинания и другие сепаративные знаки. Задача усложнялась за счет того, что необходимо все сепаративные знаки определить и оставить без изменений. К тому же имелось слово, которого нет в словаре и слово с буквой в верхнем регистре. Все знаки сепараторы были перенесены в новый файл без изменений, найденные слова исправлены, слово с верхним регистром исправлено на нижний и произведен успешный поиск с исправлением слова, а слово, которое не было найдено, осталось без изменений.

В третьем тесте был использован русский словарь, чтобы показать, что программа направлена не на определенный язык, а может работать с любым, словарь которого занесен в программу. Ошибок в тесте не обнаружено.

В четвертом тесте был проверен файл, полностью состоящий из знаков сепараторов. Так как при парсинге, поиск начинается со знаков сепараторов, то программа завершила работу, просто перезаписав содержимое файла в новый.

В пятом тесте была проверка на пустой файл вместо словаря. На этот случай в функции find_with_mistakes имеется проверка на пустое дерево. Поэтому в новый файл был перезаписан исходный без изменений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Some approaches to best match file searching – Burkhard, Keller. 1973
2. https://ru.wikipedia.org/wiki/Расстояние_Левенштейна
3. https://ru.wikipedia.org/wiki/Расстояние_Дамерау_—_Левенштейна