

Алгоритм определения повторных пользовательских отправок на основе расстояния Хэмминга в системе обучения программированию

Фатеенков Данила Витальевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Болтовский Гавриил Александрович

Приамурский государственный университет им. Шолом-Алейхема

Студент

Аннотация

В данной статье рассматривается процесс реализации, внедрения и тестирования алгоритма поиска повторных отправок решений к задачам в процессе изучения учебных дисциплин в информационной системе организации обучения программированию ITProgEdu, работающей на базе Приамурского государственного университета имени Шолом-Алейхема. В статье представлено подробное описание алгоритма нахождения расстояния Хэмминга, процессы токенизации и векторизации строк, а также принцип работы алгоритма SimHash.

Ключевые слова: PHP, алгоритм, строковый алгоритм, SimHash

An algorithm for detecting repeated user dispatches based on Hamming distance in a programming learning system

Fateenkov Danila Vitalievich

Sholom-Aleichem Priamursky State University

Student

Boltovsky Gavriil Alexandrovich

Sholom-Aleichem Priamursky State University

Student

Abstract

This paper considers the process of implementation, introduction and testing of the algorithm of searching for repeated dispatches of solutions to problems in the process of studying academic disciplines in the information system of the organization of programming training ITProgEdu, working on the basis of the Sholom-Aleichem Priamursky State University. The article presents a detailed description of the algorithm for finding the Hamming distance, the processes of tokenization and vectorization of strings, as well as the principle of operation of the SimHash algorithm.

Keywords: PHP, algorithm, string algorithm, SimHash

1. Введение

1.1 Актуальность

С ростом объёмов обрабатываемой информации в системах различного вида появляется и необходимость эффективных и не затратных по времени и памяти алгоритмов сравнения больших текстовых данных. В настоящее время существует множество различных алгоритмов работы со строками, такие как Z-строки, алгоритм Рабина-Карпа и другие. Часть этих алгоритмов работает с хэшами строк. И если возникает необходимость сравнения нескольких текстовых документов, то такие алгоритмы хоть и эффективны, но не позволят точно сравнить входные данные, из-за чего они в контексте поставленной задачи будут неэффективными.

Среди алгоритмов сравнения хэшей можно выделить SimHash. SimHash остается актуальным в эпоху big data благодаря своей эффективности в обработке и сравнении больших объемов текстовой информации. Этот алгоритм преобразует документы в компактные бинарные хеш-коды, сохраняя при этом семантическую близость: похожие документы получают хеши с малым числом различий. Это особенно востребовано в задачах веб-поиска, обнаружения дубликатов, плагиата или рекомендационных систем, где требуется быстро анализировать миллиарды документов.

1.2 Обзор исследований

И.М. Яхонтова и Н.М. Нилова провели исследование на тему математических методов сопоставления текстовых строк и разработке методики анализа и загрузки данных в информационную систему предприятия [1]. Была разработана методика анализа и загрузки данных при помощи методик нечеткого сравнения строк.

Н.М. Куляшова рассмотрела проблему распознавания почтовых адресов из строки произвольного формата [2]. В статье обоснована необходимость создания программного обеспечения для автоматизации распознавания адресных структур из строки произвольного формата.

Э.В. Шаповалова и И.В. Свищева рассмотрели алгоритмы, используемые для сопоставления строк, один из них - алгоритм Рабина-Карпа [3]. В статье описан принцип работы и асимптотика алгоритма.

М. Гао описал алгоритм на языке программирования Python, который может быть использован для разделения категорий новостей, кластеризации новостей и упрощения чтения [4]. В статье используются также модули K-Medodis и Hamming.

А.В. Миронов в своей научной работе предложил подход Matching Table для поиска четких и нечетких дубликатов сущностей, обладающих атрибутивным составом, который обеспечивает высокую скорость поиска дублирующихся записей [5]. Также в работе анализируются алгоритмы поиска нечетких дубликатов, которые могут использоваться в предложенном подходе.

1.3 Цель исследования

Цель – реализовать и внедрить алгоритм нахождения ранее отправленных пользователем решений, которые будут с точностью в 100% совпадать с новым.

2 Материалы и методы

Для реализации поставленной задачи используется язык программирования PHP. Для хранения данных об отправках используется СУБД MySQL.

3 Результаты и обсуждения

Алгоритм поиска и сравнения решений основан на SimHash, а также расстоянии Хэмминга. Основной задачей в рамках статьи можно считать необходимость реализации функционала поиска решения, которое по структуре будет полностью совпадать с ранее отправленным этим же пользователем к этой же задаче. Стандартное сравнение не подойдет, так как может отличаться формат данных (разные регистры для одних и тех же слов, разное количество пробелов и отступов), поэтому для решения такой задачи нужно задействовать метод SimHash.

SimHash – это алгоритм, который используется для поиска схожих документов или текстов. В отличие от обычного хэширования (например, SHA-256), где даже небольшое изменение в исходном тексте приводит к совершенно другому хэшу, SimHash генерирует хэш таким образом, что документы с похожим содержанием будут иметь близкие (по Хэммингу) хэши.

Данный алгоритм можно представить в виде последовательности следующих шагов:

1. Первым шагом является разбиение текста на токены. Также важно нормализовать текст, удалив знаки препинания, приведя все символы к нижнему регистру и т.д.

2. После разбиения текста на токены, на основе полученного массива данных из первого шага формируется массив шинглов для дальнейших вычислений.

Шинглы – это последовательности слов (например, триграммы). Использование шинглов помогает учитывать порядок слов в тексте, что может быть полезно для более точного сравнения документов.

3. Каждому шинглу нужно присвоить вес. В простейшем случае можно использовать частоту встречаемости шингла в документе. Для более сложных задач рекомендуется использовать подход TF-IDF (Term Frequency-Inverse Document Frequency).

4. Для каждого шингла создается вектор, представляющий его влияние на конечный хэш. Обычно используется фиксированная длина вектора (например, 64 бита), где каждый бит зависит от веса шингла.

5. На основе вектора создается сам SimHash. Если значение вектора в позиции i больше или равно нулю, соответствующий бит в хэше будет равен 1, иначе – 0.

Результат работы такого алгоритма – хэш, который отражает содержание документа таким образом, что схожие документы имеют близкие хэши по расстоянию Хэмминга.

Расстояние Хэмминга (англ. Hamming distance) – это метрика, используемая для измерения разницы между двумя строками одинаковой длины. Оно показывает количество позиций, в которых соответствующие символы двух строк различаются. Расстояние Хэмминга широко применяется в различных областях информационных технологий (например, в биоинформатике и криптографии).

Расстояние Хэмминга между двумя строками s_1 и s_2 одинаковой длины определяется как количество позиций, в которых символы этих строк различаются:

$$D_H(s_1, s_2) = \sum_{i=0}^{n-1} \delta(s_1[i], s_2[i])$$

Где n – длина строк, а $\delta(a, b)$ – функция, равная 1, если $a=b$, и 0, если это условие не выполняется.

Стоит рассмотреть краткий пример: в результате работы функции SimHash были созданы 2 хэша в виде двоичной последовательности: 01001011 и 01011010. Значения различаются на 4 и 8 позициях, соответственно расстояние Хэмминга будет равно 2.

Таким образом, если расстояние Хэмминга между двумя строковыми данными равно нулю, то можно сделать. Что эти строки совпадают.

Данный алгоритм должен быть реализован в информационной системе обучения программированию ITProgEdu, которая работает на базе Приамурского государственного университета с 2023-го года и используется для обучения студентов на направлениях высшего образования. Необходимость реализации такого алгоритма обусловлена тем, что в системе студенты часто переотправляют решения, которые были проверены системой ранее. И для исключения нагрузки на такие отправки было принято решение внедрения подобного алгоритма в систему, чтобы пользователи не могли отправить один и тот же код несколько раз.

Алгоритм будет искать совпадения только для пользователя, который отправил решение к задаче и только в рамках той задачи, к которой был отправлен код программы.

На первом этапе работы алгоритма создаются токены текста. Для этого из текста удаляются знаки препинания и всё содержимое переводится в нижний регистр, а затем текст делится на слова и массив слов возвращается в качестве выходного значения. Для удаления всех символов кроме букв и цифр задействовано регулярное выражение вида « $[/\wedge p\{L\}p\{N\}s]/u$ ». В данном случае $[\wedge p\{L\}p\{N\}s]$ означает "любой символ, который не является буквой ($p\{L\}$), цифрой ($p\{N\}$) или пробелом (s)". Отрицание достигается с помощью

символа `^`, который ставится в начале квадратных скобок. Флаг `u` (Unicode) указывает, что регулярное выражение должно интерпретировать строку как последовательность кодовых точек UTF-16.

Для перевода букв в нижний регистр была использована функция `«mb_strtolower»`, позволяющая в качестве опционального аргумента указать кодировку, в которой определена строка.

Последний шаг работы функции: разделение строки на массив слов с помощью регулярного выражения, в котором указан пробельный символ. Для разбиения строки на слова использована функция `«preg_split»`:

```
function tokenize($text) {  
    $text = mb_strtolower($text, 'UTF-8');  
    $text = preg_replace('/[^\p{L}\p{N}\s]/u', '', $text);  
    $tokens = preg_split('/\s+/', $text, -1,  
PREG_SPLIT_NO_EMPTY);  
    return $tokens;  
}
```

`Array ( [0] => это [1] => пример [2] => текста [3] => для [4] => проверки )`

Рисунок 1. Разделение тестовой строки на токены с предварительным форматированием

Второй этап работы алгоритма: формирование шинглов. Функция создания шинглов принимает на вход массив токенов (слова после обработки с помощью функции токенизации) и размер шингла `n`. Она возвращает массив шинглов длиной `n`.

В функции объявлен соответствующий массив `shingles`, в котором будут сохранены все шинглы исходной строки после токенизации.

После объявления массива алгоритм в цикле проходит по всем возможным начальным позициям для шинглов. Оператор `«count($tokens) - $n»` гарантирует, что выход за пределы массива токенов при создании шингла длиной `n` исключен. Внутри цикла производится извлечение содержимого из массива токенов, начиная с индекса `i` и длиной `n` с помощью функции `«array_slice()»`. Полученный срез сохраняется в строку с помощью функции `«implode()»` и полученный результат добавляется в результирующий массив.

Размер шингла `n` влияет на точность и чувствительность алгоритма. Большие значения `n` могут привести к более точному сравнению документов, но также увеличат вероятность того, что два документа будут считаться различными даже при небольших изменениях. Малые значения `n` могут привести к большому количеству совпадений между несвязанными документами. В качестве стандартного значения `n` равен 3:

```
function getShingles($tokens, $n = 3) {  
    $shingles = [];  
    for ($i = 0; $i <= count($tokens) - $n; $i++) {
```

```
        $shingles[] = implode(' ', array_slice($tokens, $i,  
$n));  
    }  
    return $shingles;  
}
```

Array ([0] => это пример текста [1] => пример текста для [2] => текста для проверки)

Рисунок 2. Массив шинглов для тестовой строки, сформированный с помощью соответствующей функции

После формирования соответствующего массива нужно вычислить веса. Чем больше вес шингла, тем больший вклад он вносит в создание вектора документа и, следовательно, в генерацию конечного SimHash.

В простейшем случае вес может быть равен количеству раз, которое это значение встречается в документе. Это называется Term Frequency (TF). В контексте поставленной задачи такой реализации достаточно для работы алгоритма SimHash, хотя в долгосрочной перспективе рекомендуется применять более продвинутые алгоритмы.

В коде программы была определена функция для расчётов `calculateWeights`, которая в качестве результата возвращает массив целых чисел – весов каждого шингла.

Для каждого шингла в массиве `shingles` нужно проверить – существует ли он уже в массиве `weights`. Если нет, то он должен быть добавлен с начальным весом 0. А если такое значение уже есть в массиве, то его вес нужно увеличить на 1. На выходе получается ассоциативный массив, где ключами являются шинглы, а значениями для ключей – веса. Итоговый вариант функции вычисления весов следующий:

```
function calculateWeights($shingles) {  
    $weights = [];  
    foreach ($shingles as $shingle) {  
        if (!isset($weights[$shingle])) {  
            $weights[$shingle] = 0;  
        }  
        $weights[$shingle]++;  
    }  
    return $weights;  
}
```

Array ([это пример текста] => 1 [пример текста для] => 1 [текста для проверки] => 1)

Рисунок 3. Результат работы функции `calculateWeights` для тестовой строки

Предпоследний этап формирования хэша: векторизация полученных значений. Векторизация заключается в том, что каждый шингл "вкладывается" в определенные позиции вектора, и его вес влияет на значения этих позиций.

В функции производится обход всех созданных шинглов: для каждого шингла вычисляется его хэш и используется для обновления соответствующих позиций вектора. Обновление вектора производится по следующему принципу: если бит в хэше равен 1, то нужно увеличить значение соответствующей позиции вектора на вес шингла. Если бит равен 0, то нужно наоборот уменьшить значение этой позиции на вес.

Для вычисления хэша была выбрана встроенная в PHP функция «hash». В данном случае используется алгоритм crc32b, который возвращает 32-битный хэш. Параметр true указывает, что результат должен быть возвращен в виде двоичных данных, а не строки шестнадцатеричных символов.

Сформированный хэш затем должен быть преобразован в двоичную строку. Для этого написана следующая часть алгоритма:

```
$bits = str_split(str_pad(decbin(unpack('N', $hash)[1]), 32, '0', STR_PAD_LEFT));
```

Задействованы следующие функции:

1. «unpack('N', \$hash)[1]»: преобразует двоичные данные хэша в целое число. Параметр 'N' указывает на то, что ожидается 32-битное беззнаковое целое число в сетевом порядке байтов.

2. «decbin(...)»: преобразует полученное предыдущей функцией целое число в двоичную строку.

3. «str_pad(..., 32, '0', STR_PAD_LEFT)»: заполняет двоичную строку до длины 32 лидирующими нулями слева, чтобы строка всегда состояла из 32-х символов.

4. «str_split(...)»: разбивает двоичную строку на массив символов (каждый элемент массива — один бит).

После преобразования хэша в двоичную строку в цикле обновляется вектор по описанному выше принципу. В итоге была сформирована следующая функция в программном коде (входные аргумента: длина хэша и массив весов шинглов):

```
function createVector($weights, $hashSize = 64) {
    $vector = array_fill(0, $hashSize, 0);

    foreach ($weights as $shingle => $weight) {
        $hash = hash('crc32b', $shingle, true);
        $bits = str_split(str_pad(decbin(unpack('N',
$hash)[1]), 32, '0', STR_PAD_LEFT));

        for ($i = 0; $i < count($bits); $i++) {
            if ($bits[$i] === '1') {
                $vector[$i] += $weight;
            } else {
                $vector[$i] -= $weight;
            }
        }
    }
}
```

```

    }

    return $vector;

}

Array ( [0] => -1 [1] => 1 [2] => -1 [3] => 1 [4] => -1 [5] => 1 [6] => -1 [7] => 3 [8] => 1 [9] => 3 [10] => -1 [11] => -1 [12] => -3 [13] => 1 [14] => 1 [15] => 1 [16] => 1 [17] => -3 [18] => 1 [19] => 1 [20] => -1 [21] => 1 [22] => 1 [23] => -3 [24] => -1 [25] => 1 [26] => -1 [27] => -1 [28] => 1 [29] => -1 [30] => 3 [31] => 1 [32] => 0 [33] => 0 [34] => 0 [35] => 0 [36] => 0 [37] => 0 [38] => 0 [39] => 0 [40] => 0 [41] => 0 [42] => 0 [43] => 0 [44] => 0 [45] => 0 [46] => 0 [47] => 0 [48] => 0 [49] => 0 [50] => 0 [51] => 0 [52] => 0 [53] => 0 [54] => 0 [55] => 0 [56] => 0 [57] => 0 [58] => 0 [59] => 0 [60] => 0 [61] => 0 [62] => 0 [63] => 0 )

```

Рисунок 4. Результат векторизации тестовой строки

Полученный массив осталось преобразовать в двоичную строку, которая и будет являться результатом работы SimHash. Если обратить внимание на работу функции векторизации, то можно увидеть отрицательные значения на некоторых позициях. Очевидно, что в двоичной строке быть двоичных значений не может, поэтому их нужно привести к 0 на этапе формирования итоговой строки.

Функция `simHash`, отвечающая за формирование итоговой строки, работает достаточно просто: создаётся новая переменная (пустая), в которую записываются посимвольно значения из вектора хэша: если значение меньше нуля, то оно преобразуется в 0, иначе в 1. Функция `simHash` выглядит следующим образом:

```
function simHash($vector, $hashSize = 64) {
    $simHash = '';

    for ($i = 0; $i < $hashSize; $i++) {
        $simHash .= $vector[$i] >= 0 ? '1' : '0';
    }

    return $simHash;
}
```

SimHash 1: 0101010111000111101101100100101111111111111111111111111111111111

Рисунок 5. Результат преобразования вектора весов шинглов тестовой строки
в итоговый хэш

Для нахождения расстояния Хэмминга достаточно сравнить 2 строки посимвольно и количество несоответствий сохранить в соответствующую переменную. Это и есть расстояние Хэмминга:

```
function hammingDistance($a, $b) {
    $distance = 0;
    for ($i = 0; $i < strlen($a); $i++) {
        if ($a[$i] !== $b[$i]) {
            $distance++;
        }
    }
    return $distance;
}
```



```

    let toastBody = toastElement.querySelector('.toast-
body');

    toastBody.innerHTML = `
        Решение <a
href="code_solutions.php?solution_id=${solutionId}">#${solutionI
d}</a> было отправлено ранее (<b>${checkTime}</b>).<br>
    `;

    let toast = new bootstrap.Toast(toastElement);
    toast.show();
}

```

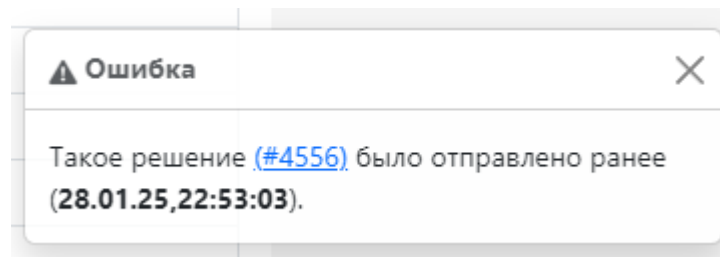


Рисунок 7. Сообщение о том, что студент уже загружал такое решение ранее

4 Выводы

В данной статье был рассмотрен SimHash и его применение для нахождения совпадающих текстовых данных, а также описана работа алгоритма нахождения расстояния Хэмминга на основе хэшей, построенных с помощью SimHash.

Поэтапно описана работа SimHash, её реализация на языке программирования PHP, а также интеграция в существующую систему обучения программированию на базе Приамурского государственного университета имени Шолом-Алейхема.

Библиографический список

1. Яхонтова И.М., Нилова Н.М. Исследование и разработка информационной системы анализа и загрузки данных при помощи нечеткого сравнения строк // Modern Economics: Problems and Solutions. 2023. Т. 11. С. 8-19.
2. Куляшова Н.М. Программное обеспечение и алгоритмы распознавания адресных структур // Информационные технологии. 2021. Т. 27. № 5. С. 275-280.
3. Шаповалова Э.В., Свищева И.В. Выявление плагиата с использованием алгоритма Рабина-Карпа // Моя профессиональная карьера. 2022. Т. 2. № 43. С. 204-208.
4. Гао М. Модель кластеризации заголовков новостей с использованием расстояния Хэмминга // Актуальные проблемы авиации и космонавтики : Сборник материалов VII Международной научно-практической конференции, посвященной Дню космонавтики: в 3 томах, Красноярск, 12–16 апреля 2021 года / Под общей редакцией Ю. Ю. Логинова. Том 2.

Красноярск: Федеральное государственное бюджетное образовательное учреждение высшего образования "Сибирский государственный университет науки и технологий имени академика М.Ф. Решетнева", 2021. С. 71-75.

5. Миронов А.В. Формирование и анализ подхода Matching Table для поиска четких и нечетких дубликатов // Взаимодействие науки и практики в условиях инновационного развития общества : Сборник научных трудов / МНИЦ «Наукосфера». Смоленск: Общество с ограниченной ответственностью "Смоленский социологический центр", 2022. С. 33-36.