УДК: 005:37

Білощицький Андрій Олександрович
Доктор технічних наук, завідувач кафедри інформаційних технологій
Діхтяренко Олександр Васильович
Аспірант кафедри основ інформатики

Київський національний університет будівництва і архітектури, Київ

ОПТИМІЗАЦІЯ СИСТЕМИ ПОШУКУ ЗБІГІВ ЗА ДОПОМОГОЮ ВИКОРИСТАННЯ АЛГОРИТМІВ ЛОКАЛЬНО-ЧУТЛИВОГО ХЕШУВАННЯ НАБОРІВ ТЕКСТОВИХ ДАНИХ

Анотація. *Запропоновано використання алгоритмів локально чутливого хешування як способу збільшення повноти вибірки у процесі перевірки текстових документів системою пошуку збігів. Розглянуто кілька відомих алгоритмів та зроблено теоретичну оцінку доцільності їх застосування. Описано принципи роботи кожного з методів та спосіб використання в рамках системи, що розробляється.*

Ключові слова: хеш-функція, локально-чутливе хешування, коефіцієнт жаккара, ssdeep, minhash, simhash

Аннотация. *Предложено использование алгоритмов локально-чувствительного хеширования для увеличения полноты выборки при проверке текстовых документов системой поиска совпадений. Рассмотрено несколько известных алгоритмов и сделана теоретическая оценка целесообразности их использования. Сделано описание принципов роботы каждого метода и способ использования в разрабатываемой системе.*

Ключевые слова: хеш-функция, локально-чуствительное хеширование, коэффициент Жаккара, ssdeep, minhash, simhash

Annotation. *The general goal of this article is a review of algorithms of locally sensitive hashing (LSH) and optimization of duplicate matching system by these algorithms. The LSH algorithms differ from cryptographic hash functions, it allows us to get similar output values for similar input data. Briefly describes the features of each algorithm and evaluated the possibility of integration into the system. Locally sensitive hashing can be applied when we're comparing the shingles from one document to another. The duplicate matching system works in two stages: indexing and search. The indexing is performed for each document only once, so it makes sense to move as much as possible operations at the stage of indexing. The index of document is a set of the hash values from the shingles and information about original position of the elements in a original text document. Therefore, the implementation of LSH is possible without special manufacturing costs. In the second stage (matching) hashes of the elements are compared to full compliance. If we use LSH, the comparing of hashes should be processed using Hamming metric or addition modulo 2 (XOR), so, theoretically, the performance should not be adversely affected. The disadvantage of using LSH is the possibility of false-positive findings, when using of cryptographic hash functions can give results with very high accuracy.*

Keywords: hash function, locality-sensitive hashing, Jaccard index, ssdeep, minhash, simhash

Вступ

Стаття написана в рамках проекту пошуку збігів у текстах. Розглянуто етап безпосередньої звірки документів між собою, методи оптимізації швидкості цього процесу та збільшення якості пошуку. Визначено загальновідомі методи перевірки елементів на схожість та запропоновано свій теоретичний варіант. Підведено підсумок щодо раціональності використання методів. Рішення, яке дозволило б знаходити неповні дублікати елементів, й досі є об'єктом, який потребує детальнішого дослідження. Хоча потреба в реалізації цього рішення виникла достатньо давно – єдиного підходу до проблеми не визначено.

Мета статті

Мета статті – проаналізувати кілька варіантів визначення схожості наборів даних (у даному випадку це текстові дані), алгоритми яких знаходяться у відкритому доступі, опубліковані статті з описами.

Виклад основного матеріалу

Існує набагато більше варіантів визначення неповних дублікатів, але вони є комерційною таємницею. Закриті рішення використовуються в різних пошукових системах, причому це не тільки інтернет-пошуковики, які індексують гіпертекстові сторінки, а й антивірусні системи, яким доводиться розпізнавати віруси навіть якщо їх код був обфускований різними способами. Швидкість і повнота результату роботи системи прямо пропорційно впливає на оцінку програмного забезпечення. Іншими словами, ніхто не буде ставити антивірус, якому для роботи необхідно 100% потужності комп'ютера, або який використовує лише 1%, але нічого не знаходить. Тому, цілком можливо, що готове продуктивне рішення існує, але знаходиться в таємниці.

В рамках роботи над системою пошуку збігів у текстах дисертацій, було вирішено поділити весь робочий механізм на два незалежних. Отже, розв'язання задачі буде здійснюватися за допомогою двох процесів:

1) Додавання документів у базу та їх індексація;

2) Пошук збігів документа з наявними в базі.

Перший процес має забезпечувати попередню обробку документів та збереження їх в базі даних у вигляді, придатному для швидкої перевірки.

Другий – діставати уже підготовлені дані з бази та робити пошук збігів. У даному випадку для користувача важливішою є швидкість виконання другого процесу, тому доцільно перенести в перший максимальну кількість операцій. У поточному варіанті система пошуку збігів, для перевірки двох документів на схожість, отримує масив елементів першого та другого документу. Кожен елемент містить хеш від частки тексту, якій він відповідає, та інформацію про положення цієї частини в тексті.

Кожен хеш з набору одного документу шукається в наборі іншого. В разі знаходження відповідності, результат повертається у вигляді масиву двох елементів, де перший – порядковий номер елементна з першого набору і відповідно другий – з другого. Масив {55, 109} означає, що 55-й елемент першого документу збігається з 109-м другого документу. Кінцевий результат перевірки – масив таких масивів: { {55, 109}, {55, 211}, {58, 300}, {112, 3}… }.

Завдання даної статті – розглянути можливість отримання повнішого масиву збігів без значних втрат продуктивності та точності. Якщо документ А містить x елементів, а документ В – y, то для пошуку збігів нам необхідно виконати x x y операцій. Якщо вся база документів сумарно містить z елементів, то для перевірки документа з всією базою необхідно K = x x z, операцій. Оскільки розмір набору кожного документу лежить у вузьких рамках, припускаємо, що всі набори містять однакову кількість елементів, та візьмемо змінну n як кількість документів у базі. Тоді кількість операцій пошуку в усій базі K = x x nx, K = nx^2. Оскільки x – це більш-менш стала величина, то складність даного алгоритму становитиме O(n) і залежатиме лише від кількості документів у базі. Збільшення витрати часу на перевірку одного документа лінійно збільшить час на перевірку з усією базою. Тому, швидкість поточного методу перевірки умовно візьмемо за одиницю і всі наступні методи будемо оцінювати відносно цієї одиниці.

Великий мінус поточного методу – велика чутливість до мінімальних змін. Для стиснення даних, що потребують перевірки та для прискорення перевірки використовуються криптографічні хеш-функції. Основна вимога до таких функцій – рівномірність розподілу їх значень при випадковому виборі аргументів [3]. Рівномірно не значить впорядковано, тому при мінімальних змінах аргументу функції – результат буде абсолютно інший, навіть якщо у великому тексті з 60000 символів замінити лише одну букву. Ідея оптимізації – заміна криптографічної хеш-функції іншою функцією, яка б генерувала для схожих наборів даних схожі «хеші».

Ідеально, якби існувала хеш-функція, яка давала б однакові результати для схожих наборів даних, але на жаль – це неможливо. Для прикладу розглянемо два набори даних: «настала осінь дерев опало листя» і «настала осінь дерев опадало листя». Як критерій «схожості» використаємо метрику Левенштейна [4]. За методикою розрахунку необхідно визначити мінімальну кількість операцій (таких, як вставка, видалення та заміна символу) для перетворення однієї стрічки в іншу. Для наступних двох послідовностей це буде 2 одиниці:

настала осінь дерев опало листя

*настала осінь дерев опа**да**ло листя* ← вставка 2-х символів.

Припустимо, що допустима відмінність 5 одиниць, отже 2 дані набори схожі. Лишилося знайти функцію, яка давала б однаковий результат для цих наборів. Тут і криється основна проблема, а саме в кількості можливих «схожих елементів». Припустимо, що існує деяка функція f, яка дає однаковий результат, якщо дані схожі. Але проблема в тому, що у нас немає «базового» вигляду набору, тому якщо «холодильник» і «холодильники» дають однакові результати, тоді такий самий повинен бути

результат і для «х**а**лодильника». Задавши міру відмінності в 1 зміну, можна створити ланку: холодильник -> халодильник -> халадильник -> халалильник -> ралалильник… і тд. Кожне наступне слово в ланці відрізняється від попереднього на 1 символ, а значить, наша функція повинна давати однаковий результат для кожного з цих слів. Тобто, за результатами роботи функції, «ралалильник» буде еквівалентно слову «холодильник». Неважко здогадатися, що ланку можна продовжити і змінюючи по одному символу привести до того, що початкове слово «холодильник» буде дорівнювати будь-якому іншому, тобто умови існування нашої функції *f* – це генерація однаково хешу для будь-яких вхідних даних. Тому «ідеальна» функція на практиці також і безкорисна.

Теоретично, дану функцію можна створити, додавши на діапазон їх значень певні «базові точки». Припустимо, що існує функція, яка дає подібні результати при «подібних» аргументах і результатом якої може бути число від 0 до 1000. Тоді, за «опорні» елементи можна взяти кожну 10-ту точку числа 0,10,20,30… (рис. 1).

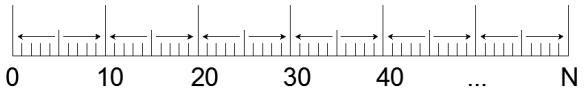

Рис. 1. Шкала значень теоретичної хеш функції з точками округлення

Стрілками вказані напрями округлення до найближчих «опорних точок». Якщо округлювати до 10 за математичними правилами. Тоді, якщо слову «опало» відповідатиме результат 578, а слову «опадало» результат 582 – вони обидва округляться до 580. Таким чином будуть певні «групи» схожих елементів. Проблема в тому, що «межі» ми поставили вручну. Отже, якщо слово «схема» дасть результат 124, а слово «схем» – 125, вони, за цією логікою виявляться несхожими.

Таким чином, частково схожість буде визначатися випадково, залежно від результату функції та вибраних нами точок. Це не дозволить знайти всі «схожі» варіанти, які можуть бути в наборах під час перевірки, але на певний відсоток повнота вибірки збільшиться – а це краще, ніж нічого. Даний варіант лише теоретична пропозиція, і потребує значного опрацювання до можливості його використання, тому розглянемо інші варіанти.

Коефіцієнт Жаккара

Коефіцієнт для визначення схожості двох наборів даних, часто використовується у алгоритмах порівняннях, тому розглянемо спочатку його. Коефіцієнт названо на честь французького вченого Пола Жаккара, який вперше запропонував цей спосіб визначення міри подібності [1]. Графічно його можна показати як спільну ділянку двох площин (рис. 2).

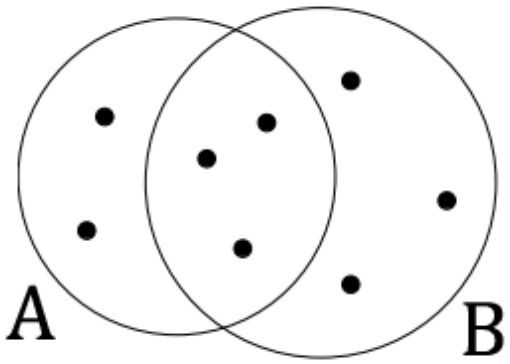

Рис. 2. Графічне представлення визначення коефіцієнта Жаккара

Алгоритм визначення досить простий, для прикладу візьмемо ті ж самі 2 фрази «настала осінь опа(да)ло листя». Розбивши обидві фрази на окремі слова, отримаємо дві множини А і В з єдиною різницею: набір А містить слово «опало», а набір В – «опадало». Коефіцієнт визначається за формулою:

$$J(A,B)=\frac{|A\cap B|}{|A\cup B|}.$$

На перетині ми отримаємо 4 елементи (настала, осінь дерев, листя), а в об'єднанні – 6 (настала, осінь, дерев, опало, опадало, листя). Таким чином коефіцієнт Жаккара для цих двох наборів буде дорівнювати 0,66. Ідентичні набори будуть мати коефіцієнт рівний одиниці, абсолютно різні – 0. Даний коефіцієнт можна застосувати для оптимізації процесу знаходження однакових хешів, але хеші при цьому доведеться замінити множинами слів. Тобто, операція хешування пропускається і документ містить не набір хешів, а набір множин. При перевірці двох документів кожна множина одного документу перевіряється на подібність кожній множині другого. Недоліком такого підходу є те, що програмі доведеться робити в n^2+2 операцій більше, де n – кількість слів в одному наборі. Також необхідно зберігати повні набори даних замість хешів, що приведе до збільшення використання місця на диску.

Locality-sensitive hashing

Локально-чутливе хешування – загальна назва методів, що дозволяють отримати схожі сигнатури для схожих наборів даних. Як зрозуміло з назви, основна ціль методу – локальна чутливість, тобто при зміні якоїсь частини об'єкта, що хешується, сам хеш не повинен абсолютно змінюватися. В основному алгоритми, що входять в дану групу, використовують поділ вхідних даних на частини та хешування цих частин з наступним випадковим (але згідно певного правила чи ознаки) відбиранням елементів з отриманої колекції [8]. Відібрані елементи одного масиву даних можуть порівнюватися з елементами іншого за допомогою

коефіцієнта Жаккара або конкатенуватися у текстову строку і порівнюватися, використовуючи метрику Левенштейна [1] або Хеммінга [9].

Context Triggered Piecewise Hashing (ssdeep)

Метод хешування, що дозволяє отримувати схожі результати для схожих даних. Метод розроблявся для визначення схожих файлів і в принципі може працювати з будь-якими даними. Даний алгоритм використовує rolling hash (ковзаючий хеш) [6; 7] для визначення контексту хешування. Цей хеш пряцює як ковзаюче вікно, що з кожним кроком додає кілька байтів в кінець хешу та відкидає на початку. Подібний спосіб використовується і при індексації тексту в системі пошуку збігів, але замість байтів використовуються слова, тому зсув «вікна» відбувається не на сталу величину, а на величину поточного слова. Далі, базуючись на поточному результаті ковзаючого хешу, вираховується міні-сигнатура отриманих у «вікні» даних. При цьому можуть використовуватися різні функції, залежно від того чи відповідає отриманий результат певним умовам (наприклад, чи ділиться чисельний еквівалент хешу на 64 без останку). Загальних хеш визначається конкатенацією місцевих хешів, тому часткова зміна даних змінить лише частину результату, а не абсолютно весь хеш. Базуючись на даному способі, було розроблено алгоритм отримання сум файлів та реалізовано у вигляді програмного забезпечення під назвою ssdeep (http://ssdeep.sourceforge.net/). Проведені дослідження показують, що даний метод у 9 разів повільніший при застосуванні його до даних розміром до 1 Мб [5]. Швидкість підрахування різниці між двома хешами не досліджувалася. Спосіб підходить для використання у системі пошуку збігів, оскільки основне навантаження йде на створення хешів, порівняння відбувається за допомогою метрики Хеммінга і не потребує значних ресурсів.

Minhash

Спосіб використовує кілька вже розглянутих методів. Даний алгоритм дуже схожий на метод шинглів [2], але з деякими оптимізаціями. Для отримання сигнатур документу в даному випадку виконується така послідовність дій:

1) розбиття документу на набір шинглів;
2) розраховування хешу від кожного шинглу;
3) визначення і збереження мінімального значення з усіх отриманих результатів;
4) повторення кроків 2-3 ще N разів, використовуючи інші N хеш-функції.

Можна використати N=30 різних хеш-функцій, а можна і 300. Чим більше функцій, тим більшу точність буде давати метод. Одна з особливостей хеш-функцій – рівномірний розподіл значень, вірогідність отримання великого чи малого значення однакова [3]. Відбираючи з кожного набору мінімальне значення, ми гарантовано отримуємо випадкову вибірку [8]. Хеш-функції можна замінити складенням по модулю (XOR) з випадково підібраними числами. Головне – прослідкувати, щоб числа були однакові для всіх елементів, які порівнюються (тобто числа вибирають випадково, але 1 раз на всю колекцію). Отримані значення порівнюють між собою за допомогою метрики Хеммінга.

Simhash

Алгоритм, названий simhash, був розроблений одним із співробітників Google [10] для порівняння великих наборів даних. Метод дозволяє створювати сигнатури для наборів даних. Порівняння сигнатур замість перевірки всього набору дає великий виграш у швидкості. Крім цього, на даному алгоритмі базується і запатентована в Google технологія корекції правопису слів http:// google.com/patents/US8661341. У нашому випадку кожен шингл – це текстовий рядок з фіксованою кількістю слів, тому його можна розбити на окремі слова і також вважати набором.

Для створення сигнатури необхідно виконати такі дії:

1) вибираємо розмір сигнатури (наприклад 32 біти);
2) створюємо масив V такого ж розміру, як кількість бітів у сигнатурі і заповняємо його нулями;
3) вибираємо хеш-функцію, що дає результати розміру аналогічного вибраному (наприклад crc32);
4) для кожного набору, для кожного елементу в наборі розраховуємо значення хеш-функції;
5) для кожного отриманого хешу в наборі, для кожного біта i:
   a. Якщо біт = 1, додаємо 1 до V[i]
   b. Якщо біт = 0, віднімаємо 1 від V[i]
6) Для кожного елемента i масиву V:
   a. Якщо V[i] > 0, V[i] = 1
   b. Якщо V[i] <= 0, V[i] = 0

Отримані сигнатури матимуть вигляд двійкових кодів (пункти 1 і 2):

1. 01000110010111010101001100010101
2. 01000111110110010100001100011101

XOR----------

3. 00000001100001000001000000001000

Виконавши операцію побітового додавання по модулю 2 (XOR), отримаємо двійкове значення (пункт 3), яке і є умовним значенням відмінності наборів. У відсотковому значенні міра подібності буде:

$$\left(1-\frac{5}{32}\right)\cdot 100\% = 84\%.$$

Таким чином ми можемо визначати відсоток подібності наборів. Деякі праці [11] відмічають великий відсоток фальшивих позитивних результатів для великої кількості наборів з різною структурою збереження даних (наприклад,

порівняння pdf з doc, розбиваючи не на слова, а на ланцюжки даних). У нашому випадку всі елементи мають однакову структуру та кількість елементів, тому алгоритм має дати якнайкращі результати, але перевірка необхідна в будь-якому випадку.

Висновок

Таким чином, систему пошуку теоретично можна оптимізувати і значно збільшити кількість знайдених збігів, використавши для створення індексу документа функції локально-чутливого хешування. При цьому існує ризик збільшення кількості помилок і знаходження «фальшивих» збігів, тому кожен з методів потребує емпіричних досліджень. Оскільки всі способи використовують для порівняння метрику Хеммінга або додавання по модулю 2, то операція порівняння не займе багато ресурсів. Для обраного способу можна визначити один раз допустимий відсоток подібності в умовних одиницях для конкретного методу, і не вираховувати кожного разу відсоткове значення. Самі алгоритми локально-чутливого хешування використовують більше ресурсів, ніж алгоритми криптографічних функцій, але це не є критичним недоліком, оскільки індексація документів відбувається всього один раз.

Список літератури

1. *Jaccard, P. Distribution de la flore alpine dans le Bassin des Dranses et dans quelques regions voisines / Jaccard P. // Bull. Soc. Vaudoise sci. Natur. – 1901. – V. 37, Bd. 140. – S. 241–272.*
2. *Білощицький, А. О. Ефективність методів пошуку збігів у текстах / А.О. Білощицький, О.В. Діхтяренко // Управління розвитком складних систем. – 2013. – № 14.– С. 144 – 147.*
3. *Алферов, А. П. Основы криптографии / А. П. Алферов, А. Ю. Зубов, А. С. Кузьмин, А. В. Черемушкин // Гелиос АРВ. – М., 2001. – С. 347-348.*
4. *Левенштейн, В. И. . Двоичные коды с исправлением выпадений, вставок и замещений символов. / В. И. Левенштейн // Доклады Академий наук СССР, 1965. 163.4:845-848.*
5. *Jesse Kornblum Identifying almost identical files using context triggered piecewise hashing / Jesse Kornblum // Digital Investigation 3S (2006) S. 91–S97.*
6. *Karp, Richard M. Efficient randomized pattern-matching algorithms / Karp , Richard M., Rabin, M.O. // IBM Journal of Research and Development , vol.31, no.2, S. 249–260, March 1987 doi: 10.1147/rd.312.0249.*
7. *Томас Х. Кормен Алгоритмы: построение и анализ. – 2-е изд. / Томас Х. Кормен // М. : Вильямс, 2006. – С. 1296. – ISBN 0-07-013151-1.*
8. *Anand Rajaraman Mining of Massive Datasets / Anand Rajaraman, Jeffrey David Ullman // Cambridge University Press, 2011.*
9. *Hamming, Richard W. Error detecting and error correcting codes. / Hamming, Richard W. // Bell System technical journal 29.2 (1950): 147-160.*
10. *Charikar, Moses S. Similarity estimation techniques from rounding algorithms. / Charikar, Moses S. // Proceedings of the thiry-fourth annual ACM symposium on Theory of computing. ACM, 2002.*
11. *Sadowski, Caitlin Simhash: Hash-based similarity detection. / Sadowski, Caitlin, Greg Levin. // Technical report, Google, 2007.*

References

1. *Jaccard, P. (1912). The distribution of the flora in the alpine zone. 1. New phytologist, 11(2), 37-50.*
2. *Biloshchytskyi, A. & Dikhtiarenko, O. (2013). The effectiveness of methods for finding matches in texts. Management of complex systems, 14, pp. 144 – 147.*
3. *Alferov, A. P., Teeth, A. J., Kuzmin, A. C., & Cheremushkin, A. C. (2005). Foundations of cryptography: a textbook. M: Helios ARVs.*
4. *Levenshtein V. I. (1965) Binary codes with correction for deletions, insertions and substitutions of characters. Reports of USSR Academy of Sciences, 163.4: 845-848.*
5. *Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. Digital investigation, 3, 91-97.*
6. *Karp, R. M., & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, 31(2), 249-260.*
7. *Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). Introduction to algorithms. Cambridge : MIT press, 2, 531-549.*
8. *Rajaraman, A., & Ullman, J. D. (2011). Mining of massive datasets. Cambridge University Press.*
9. *Hamming, R. W. (1950). Error detecting and error correcting codes. Bell System technical journal, 29(2), 147-160.*
10. *Charikar, M. S. (2002, May). Similarity estimation techniques from rounding algorithms. In Proceedings of the thiry-fourth annual ACM symposium on Theory of computing (pp. 380-388). ACM.*
11. *Sadowski, C., & Levin, G. (2007). Simhash: Hash-based similarity detection. Technical report, Google.*

Стаття надійшла до редколегії 25.06.2014

Рецензент: д-р техн. наук, проф. С.Д. Бушуєв, Київський національний університет будівництва і архітектури, Київ.