Алгоритми швидкого пошуку схожих документів

НАДРИГАЙЛО Т.Ж., ДЕМЕНІКОВ А.В.

Дніпродзержинський державній технічний університет

У даної роботі розглянуті основні задачі, що виникають при комп'ютерній обробці текстів, основні алгоритми їх рішення та яким чином їх можна пристосувати для вирішення даної задачі, а також алгоритм шинглів, його модифікація для вирішення задачі швидкого індексованого пошуку по текстовій базі, оптимізація алгоритму, описані особливості програмної реалізації.

В данной работе рассмотрены основные задачи, которые возникают при компьютерной обработке текстов, основные алгоритмы их решения и каким образом их можно приспособить для решения данной задачи, а также алгоритм шинглов, его модификация для решения задачи быстрого индексированного поиска по текстовой базе, оптимизация алгоритма, описаны особенности программной реализации.

In hired basic tasks which arise up at computer text manipulation are considered, basic algorithms of their decision and how they can be adjusted for the decision of this task, and also algorithm of shingles, his modification for the decision of task of rapid index search on a text base, optimization of algorithm, the features of programmatic realization are described.

На протязі останніх років бурхливого розвитку зазнали технології машинної обробки текстів, як то перевірка орфографії, програми-перекладачі, спам-фільтри на поштових серверах, з'явились програми для автоматичного анотування тексту, антиплагіат-системи, тощо.

Для вирішення багатьох з цих задач, важливим етапом є оцінка схожості текстів між собою, виділення в текстах подібних фрагментів. Однією з найбільш значимих є швидкий пошук та оцінка схожості документа на всі документи з деякого масиву. Оскільки це досить трудомістка задача, інколи приходиться жертвувати точністю заради швидкості виконання алгоритму.

Перейдемо до розгляду існуючих алгоритмів.

Алгоритм Рабіна-Карпа. Алгоритм Рабіна-Карпа – це алгоритм пошуку строки, який шукає підстроки в тексті використовуючи хешування. Він був розроблений у 1987 році Майклом Рабіном та Ричардом Карпом.

Алгоритм рідко використовується для пошуку одиночного шаблону, але має значну теоретичну важливість і дуже ефективний у пошуку множинних збігів шаблонів. Для тексту довжини n і шаблону довжини m, його середній час виконання і кращий час виконання це $O(n)$, але в гіршому випадку він має продуктивність $O(nm)$, що є однією з причин того, чому він не занадто широко використовується. Однак, алгоритм має унікальну особливість знаходити будь-яку з k строк менше, ніж за час $O(n)$ в середньому, незалежно від розміру k.

Оскільки кількість рядків, які ми шукаємо, дуже велика, звичайні алгоритми пошуку одиночних строк стають неефективними.

Пошук підстрок зрушенням і конкуруючі алгоритми. Основною проблемою алгоритму є знаходження сталого рядка довжини m, що називається зразком, в тексті довжини n, наприклад, знаходження рядка "*sun*" в реченні "*Hello sunshine in this vale of tears*". Один з найпростіших алгоритмів для цієї задачі просто шукає підстроками у всіх можливих місцях

Цей алгоритм добре працює в багатьох практичних випадках, але зовсім не ефективний наприклад на пошуку строки з 10000 "*a*", за якими слідує "*b*" в рядку з 10 мільйонів букв "*a*". У цьому випадку він показує своє найгірше час виконання $\Theta(mn)$.

Алгоритм Кнута - Морріса - Пратта зменшує цей час до $\Theta(n)$ тільки один раз використовуючи предвичисленія для кожного символу тексту. Алгоритм Бойера - Мура пропускає не один символ, а стільки скільки максимально можливе для того, щоб пошук вдався, ефективно зменшуючи кількість ітерацій через зовнішній цикл, тому кількість символів, з якими проводиться порівняння, можна порівняти з n/m в кращому випадку, на відміну від цих алгоритмів, Алгоритм Рабіна-Карпа фокусується на прискоренні строк.

Використання хешування для пошуку підстрок зрушення. Замість того, щоб використовувати більш розумний пропуск, алгоритм Рабіна-Карпа намагається прискорити перевірку еквівалентності зразка з підстроками в тексті використовуючи хеш-функцію.

Хеш-функція – це функція, яка перетворює кожен рядок в числове значення, що називається хеш-значення, наприклад, ми можемо мати *hash*("hello") = 5. Алгоритм використовує той факт, що якщо два рядки однакові, то і їх хеш-значення також однакові. Таким чином, все, що нам потрібно, це порахувати хеш-значення тієї підстроки, яку ми шукаємо, а потім знайти підстроки з таким-самим хеш-значенням.

Однак, існують дві проблеми, пов'язані з цим. Перша – так як існує дуже багато різних строк, для того, щоб мати невеликі хеш-значення, ми повинні мати деякі рядки, хеш-значення яких співпадають. Це означає, що незважаючи на те, що хеш значення збігаються, строки можуть не збігатися; нам необхідно перевіряти, що це дійсно так, що займає досить багато часу для довгих підстрок. На вдачу, добра хеш-функція забезпечує нам те, що при досить хороших початкових значеннях це не відбуватиметься дуже часто, і в результаті середній час пошуку невеликий.

Якщо ми наївно перераховуватимемо хеш-значення для підстрок s[i+1..i+m], то це буде вимагати час $\Omega(m)$, і, так як це робиться в кожному циклі, алгоритм буде вимагати час $\Omega(mn)$, тобто такий самий, як і у найбільш простих алгоритмів. Прийом для вирішення цього завдання полягає в тому, що змінна hs вже містить хеш-значення для s [i.. I + m-1]. Якщо ми зможемо використовувати його для розрахунку наступного хеш-значення за сталий час, тоді наша задача буде вирішена. Ми будемо робити це використовуючи так званий кіль-

цевої хеш. Кільцевої хеш – це хеш-функція, що використовується спеціально для цієї операції. Найпростішим прикладом кільцевого хеша є додавання значень кожного наступного символу в підстроками. Потім, ми можемо використовувати цю формулу для підрахунку кожного наступного хеш-значення за фіксований час: *hash*(s[i+1..i+m]) = *hash*[i..1+m-1] – s[i] + s[i+m]. Ця проста функція працює, але в результаті вираз в 8 рядку буде виконуватися частіше, ніж в разі використання інших, більш розумних, кільцевих хеш-функцій.

Зауважимо, що якщо ми дуже невдачливі, або маємо дуже погану хеш-функцію, таку як константну функцію, то рядок 8 дуже ймовірно буде виконуватися *n* раз, на кожній ітерації циклу. Так як вона вимагає часу Ω(*m*), то алгоритм повністю буде вимагати час Ω(*mn*).

Використовувана хеш-функція. Ключем до продуктивності алгоритму Рабіна-Карпа є ефективне обчислення хеш-значення послідовних підстроками тексту. Одна популярна і ефективна кільцева хеш-функція інтерпретує кожну підстроку як число в деякій системі відліку, основа якої є великим простим числом. Наприклад, якщо підстроками "hi" і основа системи числення 101, хеш-значення буде 104 × 1011 + 105 × 1010 = 10609 (ASCII код 'h' - 104 і 'i' - 105). Технічно, цей алгоритм тільки подібний до цього числа в не десятковій системі числення, так як для прикладу ми взяли "основу" менше, ніж одну з його "цифр".

Істотна користь досягається таким поданням, яке дає можливість для розрахунку хеш-значення наступної підстроки зі значення попередньої шляхом виконання тільки постійного набору операцій, незалежно від довжин підстрок. Наприклад, якщо ми маємо текст "abracadabra" і шукаємо зразок довжини 3, ми можемо розрахувати хеш підстроками "bra" з хеша підстроками "abr" (попередня підстроками), віднімаючи число додане для першої літери 'a' з "abr", тобто 97 × 1012 (97 - ASCII для 'a' і 101 – основа, яку ми використовуємо), множення на основу і нарешті додаючи останнє число для "bra", тобто 97 × 1010 = 97. Якщо підстроки у запиті досить довгі, то цей алгоритм досягає великої економії у порівнянні з багатьма іншими схемами хешування. Теоретично, існують інші алгоритми, які можуть забезпечити подібний обсяг обчислень, наприклад, помножуючи ASCII-значення усіх символів, в результаті зрушення підстроки треба буде тільки поділити на перший символ і помножити на останній. Обмеженням, однак, є розмір типу даних цілого числа і необхідність використовувати модульну арифметику для зменшення результатів хешування; тим не менше, ці прості хеш-функції, які швидко не виробляють великі числа, такі як просто додавання ASCII - кодів, найбільш ймовірно генерують велику кількість колізій і отже - сповільнюють алгоритм. З цього випливає, що описана хеш-функція є бажаною для алгоритму Рабіна-Карпа.

Алгоритм Рабина-Карпа для пошуку множини зразків. Алгоритм Рабіна-Карпа в пошуку одиночного зразка гірше алгоритму Кнута - Моріса - Пратта, алгоритму Бойера - Мура та інших швидких алгоритмів пошуку рядків через його повільність в найгіршому випадку. Однак, алгоритм Рабіна-Карпа – це найкращий алгоритм у справі пошуку множинних зразків.

Таким чином, якщо ми хочемо знайти будь-яке з великого набору, скажімо довжини *k*, зразків фіксованої довжини в тексті, ми можемо створити простий варіант алгоритму Рабіна-Карпа, який використовує хеш-таблицю або будь-яку іншу структуру даних множини(set data structure) для перевірки того, чи належить хеш даного рядка набору хеш значень зразків, які ми шукаємо.

Тут ми припускаємо, що всі підстроки мають фіксовану довжину *m*, але це припущення може бути прибрано. Ми просто порівнюємо поточний хеш-значення с хеш-значеннями всіх підстрок одночасно, використовуючи швидкий перегляд в нашій структурі даних множини, і потім перевіряючи будь-який збіг, який ми знаходимо, з усіма підстроками з цим хеш-значенням.

Інші алгоритми можуть шукати одиночний зразок за час O(*n*), і отже, вони можуть бути використані для пошуку *k* зразків за час O(*nk*). На противагу їм, варіант алгоритму Рабіна-Карпа вище може знайти всі *k* зразків за очікуваний час O(*n*+*k*), тому що хеш-таблиця, яка використовується для перевірки випадку коли хеш підстроки дорівнює хешу будь-якого із зразків використовує O(l) часу.

Алгоритм Кнута-Моріса-Пратта. Поставимо наступну задачу: є зразок ***x*** та рядок ***y***, потрібно визначити індекс, починаючи з якого зразок ***x*** міститься в рядку ***y***. Якщо ***x*** не міститься в ***y*** – повернути індекс, який не може бути інтерпретований як позиція у рядку (наприклад, від'ємне число).

Цей алгоритм з'явився в результаті ретельного аналізу brute-force-алгоритму. Дослідники хотіли знайти способи більш повно використовувати інформацію, отриману під час сканування рядки (алгоритм грубої сили її просто відкидає).

Розмір зсуву зразка можна збільшити, одночасно запам'ятавши частини тексту, що збігаються зі зразком. Це дозволить уникнути зайвих порівнянь і, тим самим, різко збільшити швидкість пошуку.

Розглянемо порівняння рядків на позиції *i*, де зразок *x*[0, m–1] зіставляється з частиною тексту *y*[*i*, *i* + *m*–1]. Припустимо, що перша розбіжність сталася між *y*[*i*+*j*] та *x*[*j*], де 1<*j*<*m*. Тоді *y*[*i*,*i*+*j*–1] = *x*[0,*j*–1] = *u* й *a* = *y*[*i*+*j*] ≠ *x*[*j*] = *b*.

При зсуві можна очікувати, що префікс (початкові символи) зразка *u* дорівнюватиме якому-небудь суфіксу підрядку тексту *u*. Довжина найбільш довгого префікса, що є одночасно суфіксом називається префікс-функцією від рядка. Це приводить нас до наступного алгоритму: нехай *mp_next*[*j*] – префікс-функція від рядка *x*[0,*j*–1]. Тоді після зсуву ми можемо відновити порівняння з місця *y*[*i*+*j*] та *x*[*j*– *mp_next*[*j*]] без втрати можливого місцезнаходження зразка. Таблиця значень *mp_next*[*j*] може бути обчислена за O(*m*) порівнянь перед початком пошуку.

Префікс-функція від рядка (позначається π(s,*i*);) – довжина найбільшого префікса рядка s[1..*i*], що не збігається з цим рядком і одночасно є його суфіксом. Часто префікс-функцію записують у вигляді вектора завдовжки |s|–1. Наприклад, для рядка "abcdabscabcdabia":

π(abcdabscabcdabia) = (000120012345601).

Алгоритм Бойера-Мура. Алгоритм Бойера - Мура пошуку рядка вважається найбільш швидким серед алгоритмів загального призначення, призначених для пошуку підстроками в рядку. Був розроблений Робертом Бойером (англ. Robert S. Boyer) і Джеєм Муром (англ. J Strother Moore) в 1977 році. Перевага цього ал-

горитму в тому, що ціною деякої кількості попередніх обчислень над шаблоном (але не над рядком, в якому ведеться пошук) шаблон порівнюється з вихідним текстом не в усіх позиціях – частина перевірок пропускаються як ті, що дають результату. Загальна оцінка обчислювальної складності алгоритму –

$$O(|haystack| + |needle| + |\Sigma|)$$

на неперіодичних шаблонах і

$$O(|haystack| \cdot |needle| + |\Sigma|)$$

на періодичних, де haystack – рядок, в якому виконується пошук, needle – шаблон пошуку, Σ – алфавіт, на якому проводиться порівняння.

Алгоритм заснований на трьох ідеях.

1. Сканування зліва направо, порівняння справа наліво. Поєднується початок тексту (рядки) і шаблону, перевірка починається з останнього символу шаблону. Якщо символи збігаються, порівнюється передостанній символ шаблону і т. д. Якщо всі символи шаблону співпали з символами рядки - значить, підрядок знайдено, і пошук закінчено. Якщо ж будь-який символ шаблону не збігається з відповідним символом рядка, вираз зсувається на кілька символів праворуч, і перевірка знову починається з останнього символу.

Відстань зсуву, обчислюються за двома еврістикам.

2. Еврістіка стоп-символу. Припустимо, що ми виконуємо пошук слова «колокол». Перша ж буква не співпала - «к» (назвемо цю букву стоп-символом). Тоді можна зрушити шаблон вправо до останньої букви «к».

Рядок:	*	*	*	*	*	*	к	*	*
Шаблон:	к	о	л	о	к	о	л		
Наступний крок:			к	о	л	о	к	о	л

Якщо стоп-символу в шаблоні взагалі немає, шаблон зміщається за цей стоп-символ.

Рядок:	*	*	*	*	*	а	л	*	*	*	*	*	*
Шаблон	к	о	л	о	к	о	л						
Наступний крок:						к	о	л	о	к	о	л	

Якщо стоп-символ «к» опинився за іншою літерою «К», еврістіка стоп-символу не працює:

Рядок:	*	*	*	*	к	к	о	л	*	*	*	*	*
Шаблон		к	о	л	о	к	о	л					
Наступний крок	к	о	л	о	к	о	л		?	?	?		

У таких ситуаціях виручає третя ідея АБМ - еврістіка співпавшого суфіксу.

3. Еврістіка співпавшого суфікса. Якщо при порівнянні рядку і шаблону співпало один або більше символів, шаблон зсувається в залежності від того, який суфікс співпав.

Рядок:	*	*	*	т	о	к	о	л	*	*	*	*	*
Шаблон:		к	о	л	о	к	о	л					
Наступний крок:						к	о	л	о	к	о	л	

У даному випадку збігся суфікс «кол», і шаблон зсувається вправо до найближчого «кол». Якщо підрядку «кол» в шаблоні більше немає, але він починається на «кол», зсувається до «кол», і т. д.

Обидві еврістікі вимагають попередніх обчислень – в залежності від шаблону пошуку заповнюються дві таблиці. Таблиця стоп-символів за розміром відповідає алфавіту (наприклад, якщо алфавіт складається з 256 символів, то її довжина 256); таблиця суфіксів - шуканому шаблону. Саме через це алгоритм-Бойера Мура не враховує співпавший суфікс і неспівпавший символ одночасно - це потребувало б надто багато попередніх обчислень.

Таблиця стоп-символів. У таблиці стоп-символів вказується остання позиція у needle (виключаючи останню букву) кожного із символів алфавіту. Для всіх символів, що не увійшли до needle, пишемо 0 (для нумерації з 0 - відповідно, -1). Наприклад, якщо needle = "abcdadcd", таблиця стоп-символів буде виглядати так.

Символ:	a	b	c	d	(інші)
Остання позиція:	5	2	7	6	0

Якщо розбіжність сталася на позиції *i*, а стоп-символ *c*, то зсув буде *i* – *StopTable*[*c*].

Таблиця суфіксів. Для кожного можливого суфікса S шаблону needle вказуємо найменшу величину, на яку потрібно зрушити вправо шаблон, щоб він знову збігся з S. Якщо такий зсув неможливий, ставиться |needle| (в обох системах нумерації). Якщо шаблон починається і закінчується однією і тією ж комбінацією букв, |needle| взагалі не з'явиться в таблиці.

Існує швидкий алгоритм обчислення таблиці суфіксів. Цей алгоритм використовує префікс-функцію рядка.

Тут suffshift[0] відповідає всьому співпавшому рядку; suffshift[m] – пустому суфіксу. Оскільки префікс-функція обчислюється за O(|needle|) операцій, обчислювальна складність цього кроку також дорівнює O(|needle|).

Доказ корректності цього алгоритму приведений у [2].

Алгоритми пошуку найбільшого спільного підрядку. Найбільша загальна підстрока (longest common substring) – підстрока двох чи більше рядків, що має максимальну довжину.

Формально, найбільшою загальною підстрокою строк $s_1, s_2, \ldots, s_n$ називається рядок ω^*, яка задовольняє умові $\left\|\omega^*\right\| = \max\left(\left\{\|\omega\| \,\middle|\, \omega \in s_i, i = 1, \ldots n\right\}\right)$.

Операція $\omega \in s_i$ позначає що рядок ω є (можливо невласних) підстрокою рядка s_i.

Рішення задачі пошуку найбільшої загальної під-

строки для двох рядків s_1 і s_2, довжини яких m і n відповідно, полягає в заповненні таблиці $A_{i,j}$ розміром $(m+1) \times (n+1)$ за наступним правилом, приймаючи, що символи в рядку нумеруються від одиниці.

$$\begin{cases} A_{0,j} = 0, \quad j = 0 \ldots n, \\ A_{i,0} = 0, \quad i = 0 \ldots m, \\ A_{i,j} = 0, \quad s_1[i] \neq s_2[j], \quad i \neq 0, \quad j \neq 0 \\ A_{i,j} = 1, \quad s_1[i] = s_2[j], \quad i \neq 0, \quad j \neq 0 \end{cases}$$

Максимальне число $A_{u,v}$ в таблиці це і є довжина найбільшою загальної підстроки, сама підстрока: $s_1[u - A_{u,v} + 1] .. s_1[u]$ і $s_2[v - A_{u,v} + 1] .. s_2[v]$.

Всі перелічені алгоритми добре працюють у випадку коли потребується знайти точне включення зразка в вихідному тексті але якщо потребується знайти нечітке входження зразка (наприклад які-небудь виправлення, помилки, незначні перефразування тощо), то ці алгоритми стають безсилі. Розглянемо декілька прийомів пристосування перелічених алгоритмів для нечіткого пошуку включень:

1. Видалення орфографії і стоп-слов.

За допомогою видалення орфографії обходяться такі модифікації тексту як розбиття речень, орфографічні помилки. Стоп слова – це слова, що майже не несуть смислового навантаження і служать тільки для зв'язки слів або оборотів, наприклад 'це', 'як', 'так', 'і', 'в', 'над', 'до' тощо. При аналізі тексту вони частіше всього відкидаються.

2. Застосування стеммерів

Стемінг – це процес знаходження основи слова для заданого вихідного слова. Основа слова не обов'язково збігається з морфологічним коренем слова.

За допомогою стеммерів обходяться деяке перефразування тексту, помилки при написанні слів. Але у різних мовах різні принципи словотворення, тому при стемуванні виникають такі додаткові проблеми як визначення мови тексту або його фрагментів для декількамовних текстів, урахування морфології багатьох мов для правильного застосування стеммерів.

Для того, щоб було можливо встановлювати взаємопов'язаність текстів за допомогою цих алгоритмів нам необхідно знати, які саме фрази, або речення (зразки) тексту необхідно шукати. Тут є кілька варіантів : можна шукати входження найбільш змістовних частин тексту – для цього необхідно користуватися принципати автоматичного анотування тексту, або можна користуватися запитом користувача, тобто конкретним пошуковим запитом.

Усі розглянуті *пеп* алгоритми добре працюють при порівнянні «один проти одного», але вони погано масштабуються на великі масиви документів і, отже, якщо стоїть задача швидкого пошуку по базі (навіть досить грубого), то їх використання стає недоцільним.

Постановка задачі. Дана база текстових документів. Необхідно створити бібліотеку, що реалізовувала б порівняння довільного документу з усіма текстами бази на схожість. Алгоритм повинен буди оптимізован для порівняння «один проти багатьох», мати множину деяких, попередньо обчислених, характеристик, яку можна було б зберігати в якості індексу.

Бібліотека розраховується як попередній фільтр, який би вибирав найбільш схожі документи з бази для більш детального аналізу.

Алгоритм повинен мати велику бистродійність та не залежати від мови вихідного тексту.

Реалізація алгоритму. Для реалізації поставленої задачі був обраний алгоритм шинглів, так як він дозволяє створити індекс з попередньо обчислених хешів шинглів для кожного документа, а час його виконання лінійній відносно обсягу бази.

1. Формування індексу. Кожен документ перед занесенням до індексу:

На першому етапі документ приводиться до канонічного виду, за виключенням стемування слів, так як алгоритм повинен працювати без урахування морфології, замість цього слова приводяться до верхнього регістру.

На другому етапі виконується розбиття тексту на шингли, обчислення їх хеш-значень. Так як алгоритм передбачається досить грубим то обирається лише перший шингл з кожного речення. Всі обчислені хеш-значення групуються у зв'язки <назва документу, хеш-код шинглу> і заносяться до бази даних. Вибір лише першого шинглу з кожного речення забезпечує невеликий обсяг бази, велику швидкість складання індексу та велику швидкість пошуку схожих документів по базі при достатній якості пошуку.

2. Пошук схожих документів по базі.

Для пошуку схожих документів, документ-еталон також проходить етапи приведення до канонічної форми й перетворення на множину хеш-кодів шинглів, після чого вираховуються значення - коефіцієнти нечіткої схожості для тексту-еталону й кожного тексту з бази. Користувач отримує список документів, відсортований за спаданням коефіцієнта нечіткої схожості.

Також у роботі реалізовано клас для нечіткого порівняння двох текстів. Він заснований на тому самому алгоритмі, що й бібліотека, але в якості документів виступають абзаци двох текстів. Оскільки в цій задачі точність важлива, то до бази включаються усі шингли.

Слабким місцем алгоритму шинглів є перестановки слів. Так пошукова система Яndex також використовує алгоритм шинглів, довжина шинглу 4 слова, шингли йдуть встик. Способом обходження її фільтру однакових документів є перестановка кожного четвертого слова в тексті. Враховуючи цей факт, я вирішив сортувати слова в кожному шинглі перед обчисленням його хеш-значення з метою обходження модифікацій такого типу.

Висновки

В даній роботі розглянуто декілька найбільш поширених алгоритмів пошуку входження конкретної підстроки до строки, та способи їх використання у даній задачі. Найбільш дієвим з них було визначено алгоритм шинглів.