УДК 519.683:519.688

А. А. Дегтярьов, С. В. Чернишенко

Дніпропетровський національний університет ім. Олеся Гончара

РЕАЛІЗАЦІЯ АЛГОРИТМУ ПЕРЕВІРКИ ПРАВОПИСУ ЗА ДОПОМОГОЮ ТЕРНАРНОГО ДЕРЕВА ІЗ ОПТИМІЗАЦІЄЮ ЗА ВІДСТАННЮ ЛЕВЕНШТЕЙНА

Запропоновано підхід до розв'язку задачі визначення правильності написання вхідного слова та формування списку вірних за написанням альтернатив, близьких до вхідного слова. Розглядається задача оптимізації пошуку близьких альтернатив із обґрунтуванням обраного оптимізаційного критерію. Постановка та алгоритмічний розв'язок цих задач надається без прив'язки до конкретної предметної області (насамперед, без прив'язки до лінгвистики), що надає можливість поширювати запропонований підхід на доволі широке коло задач, що зводяться до поставлених.

Вступ. Часто у програмних додатках, що пов'язані із інтерактивним введенням лінгвістично змістовного тексту, виникає необхідність мати зручний інструментарій задля перевірки правопису. Такими додатками можуть бути, наприклад, текстові процесори (MS Word, OpenOffice Writer), де необхідно сповістити користувача про можливо невірний ввід слова та, за можливістю, запропонувати декілька близьких, але вірних за написанням альтернатив.

Такий засіб може також знадобитись у пошуковій системі, коли необхідно проаналізувати слова пошукового запиту. У разі можливого невірного написання деяких з них до кожного такого слова обирається вірне альтернативне написання та після цього користувачу пропонується відповідний запит: добре знайомим є повідомлення «***Did you mean …»*** у популярних пошукових системах Google та Yandex. У всіх цих випадках задача полягає в тому, що необхідно визначити, чи є з мовної точки зору подане слово правильно написаним, і, якщо ні, то запропонувати список інших, близьких за написанням, вірних варіантів.

Крім того, може виникати необхідність обрати найкращу з альтернатив, використовуючи оптимізацію за певними критеріями.

© А. А. Дегтярьов, С. В. Чернишенко, 2008

Аналіз досліджень та постановка проблеми. У даній статті ми пропонуємо зручний підхід до розв'язку задачі перевірки правопису і обрання альтернатив та детально розглядаємо принципи роботи інструменту, що добре його реалізує. Для перевірки правопису ми не спираємось на морфологічні властивості слів (на чому, наприклад, базується бібліотека iSpell та деякі інші поширені мовні бібліотеки). Морфологічні правила тісно пов'язані з конкретною мовою, крім того їх застосування часто невиправдане, з точки зору машинного часу та повноти і точності результатів. Також, необхідно створювати достатньо непросту базу правил для кожної мови та реалізовувати парсер, що буде застосовувати їх до вхідного слова; в загальному випадку, це є нетривіальною та працемісткою задачею.

Натомість, оберемо наступний концептуальний підхід: будемо вважати, що маємо сукупність наперед заданих слів певної мови (словник), які вважаємо вірними. Ці слова зберігатимемо в контейнері, який реалізовано за допомогою швидкісної структури даних, що дозволяє зручно маніпулювати символами слова (ключа) при його пошуку. Проводячи прості по символьні маніпуляції з вхідним словом одночасно з пошуком за цією структурою даних, будемо обирати альтернативи виключно із словника, визначаючи при цьому найкращі з них за певним оптимізаційним критерієм. Маючи на увазі, що словники більшості світових мов є у широкому доступі в мережі Інтернет і задача найчастіше виникає у інтерактивних додатках, де швидкість є приоритетною вимогою, то запропонований підхід вважаємо обґрунтованим та має практичний зміст. Крім того, він забезпечує достатню повноту результатів при доброму наповненні словника і його реалізація не потребує значних вкладень часу та праці, достатньо лише мати якісний словник. Реалізацію цього підходу легко можна розширити на будь-яку кількість мов, так як у ньому немає прив'язки до особливостей конкретної мови.

Постановка задачі. У рамках описаного підходу надамо математичну постановку задачі в загальному вигляді: нехай маємо мову $L(A)$, де $A = \{a_1, a_2, \ldots, a_n\}$ – лінійно впорядкований алфавіт і мова $L(A)$ не містить порожнього слова. Нехай також $D = \{\alpha_1, \alpha_2, \ldots, \alpha_k\} \subseteq L(A)$ – деяка наперед задана підмножина слів мови $L(A)$, яку будемо називати словником. Задамо множину операцій Ω, одне застосування кожної з яких переводить будь-яке

слово $\alpha_i \in L(A)$ в деяке слово $\alpha_j \in L(A)$, причому $\alpha_i \neq \alpha_j$. Уведемо тепер цілочисельну функцію відстані $d(\alpha, \beta; \Omega)$ на множині $L(A)$ (далі $d(\alpha, \beta)$), визначаючи її як мінімальну кількість операцій із Ω, шляхом послідовного застосування яких можна із слова $\alpha \in L(A)$ отримати слово $\beta \in L(A)$.

Задаючи верхню межу d_0 для відстані, можна сформулювати задачу пошуку словникових альтернатив таким чином: необхідно знайти таку множину слів $R \subseteq D$, відстань між кожним словом якої і заданим словом $\alpha_0 \notin D$ не перевищує d_0, тобто

$$R = \{\alpha \mid (\alpha \in D) \wedge (d(\alpha, \alpha_0) \leq d_0)\}. \qquad (1)$$

У свою чергу, задача пошуку найкращих альтернатив зводиться до мінімізації відстані між словом α_0 і словами із словника та формулюється таким чином: необхідно знайти таку множину $B \subseteq D$, що

$$B = \{\alpha \mid (\alpha \in D) \wedge (d(\alpha, \alpha_0) \leq d_0) \wedge (d(\alpha, \alpha_0) = \min_{\beta \in D} d(\beta, \alpha_0))\} \qquad (2)$$

Очевидно, що значення функції відстані і, відповідно, результати оптимізації залежать від заданого набору операцій Ω. Якщо визначити Ω як сукупність трьох операцій:

1) *заміна* одного символа у слові іншим символом алфавіту,
2) *видалення* одного символа у слові,
3) *вставка* одного символу алфавіту у слово,

то визначена таким чином відстань носить назву *відстані Левенштейна* ([5]).

Ми будемо розглядати задачу (2), тому що її розв'язок можна легко узагальнити і отримати розв'язок задачі (1). При цьому визначимо функцію відстані як відстань Левенштейна, так як відповідний набір операцій добре відображає сутність помилок у написанні слів та достатньо точно покриває більшість із них.

Алгоритмічна реалізація розв'язку. Задля програмної реалізації розв'язку поставленої задачі необхідно обрати структуру даних для контейнера словника, що, якнайбільше, є комплементарною описаним вище міркуванням. При цьому структура даних словника

повинна задовольняти вимозі високої швидкості пошуку та забезпечувати можливість виконання операцій 1)–3) одночасно з пошуком.

Після проведених досліджень було зроблено висновок, що у даному випадку доцільним є обрання в якості контейнера слів тернарне дерево пошуку (*Ternary Search Tree*), або просто ***тернарне дерево***. Як легко переконатись, ця структура даних зручна для маніпуляцій із символами ключа при пошуку, що не прослідкується в хешах, списках, масивах і деяких деревоподібних структурах; окрім того, вона є оптимізованою за пам'яттю та за часом пошуку, тобто алгоритм як точного, так і наближеного пошуку слова за тернарним деревом має найкращі характеристики серед інших класичних деревоподібних структур даних (червоно-чорні дерева, AVL-дерева та інші бінарні дерева, які є оптимізованими переважно на вставку або видалення елементів – див. [2; 3]).

Кожний вузол тернарного дерева характеризується асоційованим з ним одним символом алфавіту *(split char)*, який за відповідних умов долучається до формування поточного ключа при пошуку патерна, має три дочірніх вузла, які умовно назвемо «менший», «еквівалентний» та «більший», і має поле даних, яке індексується ключем даного вузла (іншими словами, відповідним словом, що зберігається в дереві). З метою економії ми не будемо детально описувати організацію тернарного дерева (для отримання детальної інформації див. [1; 4; 7]).

Натомість, розглянемо принципи реалізації алгоритму пошуку альтернатив для вхідного патерну у тернарному дереві із мінімізацією за відстанню Левенштейна. З побудови тернарного дерева видно, що так як поточний ключ (гіпотетичний результат) у процесі пошуку поступово формується з символів, асоційованих з деякими вузлами, що зустрічаються при спуску за деревом, і при цьому пошуковий патерн проходиться зліва направо посимвольно, то операції 1)-3), що визначають відстань Левенштейна, відповідно реалізуються при спуску наступним чином:

1) неврахуванням розбіжності між поточним символом патерна та асоційованого з поточним вузлом символу (заміна символу)
2) неврахуванням деякого символу у патерні при пошуці (зайвий символ у патерні, його видалення)
3) неврахуванням наступного «еквівалентного» вузла при черговому спуску при формуванні поточного ключа (пропущений символ у патерні, його додання за рахунок пропущеного вузла).

Згідно із сказаним, задаючи початковий ліміт відстані (запас штрафу) d_0 і подаючи його на вхід пошуковому алгоритму при першому виклику, та потім викликаючи його рекурсивно для всіх вузлів кожної четвірки «батьківський-молодший-еквівалентний-старший» почергово, одночасно при кожному новому виклику збільшуючи використаний штраф за відстанню і/або збільшуючи зміщення поточного символу у патерні, можемо поступово за принципом черги сформувати список знайдених альтернатив, для яких був використаний найменший штраф (тобто, список таких словникових альтернатив, відстань Левенштейна яких до пошукового патерну є мінімальною). Таким чином, позначаючи зміщення поточного символу у патерні через i і поточну максимально дозволену кількість операцій (запас штрафу) через d, умовно відобразимо множення гілок рекурсії при перегляді кожного наступного вузла на шляху на рис. 1. Надписом над кожною стрілкою позначено відповідні зміни параметрів i та d при наступному виклику рекурсивної процедури.

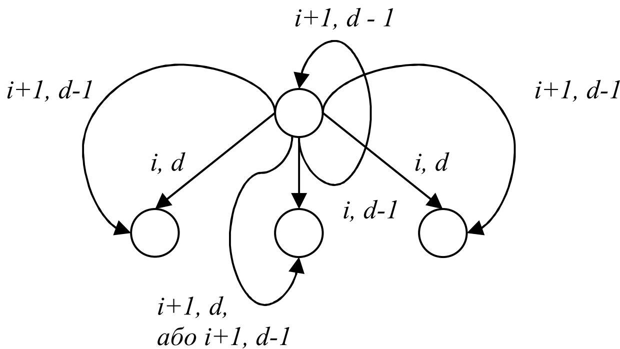

Рис. 1. Множення гілок рекурсії на кожній ітерації пошукового алгоритму

Кожна гілка рекурсії або продовжує спуск по дереву без змін параметрів, або реалізує деяку операцію із набора, змінюючи параметри i і/або d при кожному наступному виклику. Множення гілок рекурсії по вузлах завершується в одному із чотирьох випадків:

1) відповідний дочірній вузол відсутній,

2) використаний весь ліміт штрафу за відстанню,

3) розглянуті всі символи пошукового патерна,

4) в списку альтернатив є слово, штраф якого менше, ніж той, що вже використаний на даній ітерації рекурсії і подальший пошук можна припинити, бо всі знайдені слова з цієї гілки будуть більш віддалені від патерна. Зазначимо, що умова 4) дозволяє значно скоротити кількість ітерацій по дереву.

В якості оптимізаційного критерію може також виступати відстань Дамерó-Левенштейна (див. [6]) - узагальнення відстані Левенштейна. До набору операцій у такому разі додається операція перестановки символів, яка у попередньому випадку вимагала дві операції – вставки та видалення, тобто відстань Дамерó-Левенштейна у певних випадках може бути більш точним оптимізаційним критерієм. Операція перестановки символів при спуску за деревом реалізується шляхом ігнорування всіх вузлів, що знаходяться між поточним вузлом та найближчим знизу таким, якому відповідає попередній символ патерну, що аналізувався (з індексом *i -1*).

Кожній операції із набору можна надати вагу, що відрізняється від 1, таким чином досягаючи різної зміни штрафу при виконанні маніпуляцій з ключем під час пошуку та отримуючи результати оптимізації, що більш точно задовольняють конкретній задачі. Крім того, можна задати ліміт штрафів окремо по кожній з операцій, таким чином вводячи оптимізацію за вектором „штрафів по операціях” та отримуючи гнучкий контроль за набором операцій, які допускаються над ключем при пошуку.

Очевидно, що $B \subseteq R$, причому обидві множини можуть бути порожніми в силу визначення ліміту d_0. Крім того, якщо після розв’язку задачі (2) отримуємо $|B|>1$, а необхідно отримати лише одну альтернативу у результаті, то на множину D необхідно накласти вимогу строгої лінійної впорядкованості. Щодо програмної реалізації,
то в такому випадку у вузлах дерева необхідно зберігати дані, порівняння яких визначить кращу з альтернатив (наприклад, у пошуковій системі це може бути частотність появи слова у проіндексованому контенті). Також за допомогою певних евристичних алгоритмів можна оцінювати ймовірність допущення тієї чи іншої помилки в кожній з альтернатив відносно заданого слова α_0.

Висновки. Зрозуміло, що із збільшенням d_0 потужність множин B і R не спадає, і, відповідно, розбіжність між заданим словом і результуючими альтернативами стає більш значною. Крім того, слідураховувати, що із збільшенням d_0 час роботи пошукового алгоритму збільшується експоненціально за рахунок збільшення кількості спусків по вузлах і, відповідно, глибини рекурсії, тому в якості d_0 потрібно обирати не дуже велике значення. Експериментально було встановлено, що при визначенні функції відстані як відстані Левенштейна найкраще співвідношення «точність/швидкість» досягається при $d_0 = [length(\alpha_0)/4] + 1 \cdot$

Слід додати, що описаний підхід із поданими принципами реалізації є універсальним і може бути застосований для розв'язку задачі обрання найближчих альтернатив, коли поняття мови і слова носять необов'язково лінгвістичний зміст та в якості словника виступає множина лексем загального вигляду, що задовольняють певній ознаці.

Бібліографічні посилання

1. **Седжвик Р.** Фундаментальные алгоритмы на С++: Анализ. Структуры данных. Сортировка. Поиск / Р. Седжвик. – К.: «ДиаСофт», 2001.
2. **Ахо А. В.** Структуры данных и алгоритмы / А. В. Ахо, Дж. Хопкрофт, Д. Д. Ульман – М.: «Вильямс», 2000.
3. **Мейн М.** Структуры данных и другие объекты в С++ / М. Мейн, У. Саввитч – М.: « Вильямс», 2002.
4. **Sedgewick R.** Plant your data in a ternary search tree / Java algorithms and tips site http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-ternary.html
5. Levenshtein distance / Wikipedia http://en.wikipedia.org/wiki/Levenshtein_distance
6. Damerau-Levenshtein distance / Wikipedia http://en.wikipedia.org/wiki/Damerau-Levenshtein_distance
7. Ternary Search Trees / Dr. Dobbs author's site http://www.ddj.com/windows/184410528

Надійшла до редакції 11.06.08