УДК 519.72 https://doi.org/10.17721/1812-5409.2019/3.8

І.О. Завадський, к. ф.-м. н., доцент

Пошук рядка в тексті з урахуванням обмежень на обсяг кеш-пам’яті

Київський національний університет імені Тараса Шевченка, 03680, м. Київ, пр-т. Глушкова 4д, e-mail: zava@ukr.net

Igor O. Zavadskyi, Associate Professor, Ph. D.

Pattern matching by the terms of cache memory limitations

Taras Shevchenko National University of Kyiv, 03680, Kyiv, Glushkova av., 4d

Удосконалено кілька відомих підходів до розв’язання задачі пошуку рядка в тексті, таких як двобайтне читання та цикл пропуску. У застосуванні до задачі пошуку рядка в тексті на 256-символьному алфавіті ці вдосконалення дали змогу побудувати родину пошукових алгоритмів, які перевершують за швидкодією всі відомі аналоги для всіх досліджуваних довжин рядка.
Ключові слова: пошук рядка в тексті, алгоритм Боєра-Мура-Хорспула, вікно пошуку.

A few known techniques of exact pattern matching, such as 2-byte read, skip loop, and sliding search windows, are improved and applied to pattern matching algorithms, performing over 256-ary alphabets. Instead of 2-byte read, we offer “1.5-byte read”, i.e. reading more than 8 but less than 16 bits of two sequential bytes of a text at each iteration of a search loop. This allows us to fit the search table into L1 cache memory, which significantly improves the algorithm performance. Also, we introduce the so-called double skip loop instead of single one, resolve problems caused by endianness of a machine, and adopt the sliding windows technique to our algorithms. The experimental results averaged over 500 runs of algorithms on 40 different computers show that our algorithms outperform all other tested methods for all tested pattern lengths.
Key words: pattern matching, Boyer-Moore-Horspool, fast search, text search, sliding windows.

Статтю представив д.ф.-м.н., проф. Анісімов А.В.

1. Вступ

У даній роботі досліджується одна з найважливіших задач кібернетики: пошук усіх точних входжень заданого рядка у великий текст. Надалі використовуються такі позначення:

T – вхідний текст;
P – шуканий рядок (патерн);
n – довжина вхідного тексту;
m – довжина патерна;
Σ – алфавіт вхідного тексту і патерна;
$|\Sigma|$ – розмір алфавіту Σ

Як правило, ефективність алгоритмів пошуку оцінюють на площині ($|\Sigma|,m$). Ми розглядаємо на цій площині смугу $|\Sigma|=256$. Це важливий частинний випадок, оскільки у 256-символьному тексті кожен символ займає 1 байт і всі біти цього байту значущі, а отже, бінарні дані (потік бітів) можна розглядати як текст на 256-символьному алфавіті. Тому всі алгоритми пошуку бітового рядка в бінарних даних базуються на певних алгоритмах пошуку рядка в текстах на 256-символьному алфавіті.

Згідно з [1] та нашими власними експериментами, найвищу швидкодію у згаданому випадку демонструють такі алгоритми, як Fast Search (FS, [2]), що базується на порівнянні символів тексту, та родина алгоритмів BNDM, що використовують ідею бітового паралелізму. «Першопредком» усіх вищеназваних алгоритмів є алгоритм Боєра-Мура-Хорспула (BMH, [3]).

Як показано у [4], швидкодію BNDM-алгоритмів можна суттєво покращити завдяки *двобайтному читанню*, тобто формуванню індексу таблиці зсувів на основі не одного, а двох послідовних байтів тексту. Однак цей підхід призводить до суттєвого збільшення обсягу пошукових таблиць. Як свідчать експерименти, час виконання алгоритму стрімко зростає, щойно пошукові таблиці, які він використовує, разом із іншими допоміжними даними, перестають вміщуватися в кеш-пам’ять L1, типовий обсяг якої становить від 16 до 64 Кб. Для 256-символьного алфавіту обсяг пошукової таблиці із двобайтними індексами становить

© І.О. Завадський, 2019

якраз 64 Кб, що в більшості випадків призводитиме до згаданого сповільнення.

Інша відома методика прискорення різноманітних алгоритмів пошуку рядка в тексті полягає у використанні так званого *циклу пропуску* [5], що дає змогу уникати перевірки можливого збігу патерна з вікном пошуку на кожній ітерації пошукового циклу.

У цій роботі ми вдосконалюємо техніки як двобайтного читання, так і циклу пропуску, і разом з ще однією відомою методикою — *вікон пошуку, що ковзають,* застосовуємо їх до алгоритму BMH як до базового рішення. У результаті ми конструюємо родину алгоритмів пошуку рядка у 256-символьному тексті, які, за даними експериментів, перевершують за швидкодією всі інші відомі рішення.

2. Опрацювання нецілого числа байтів

В алгоритмі BMH довжина максимального зсуву вікна пошуку за евристикою «поганого символу» дорівнює m, а його ймовірність для випадкових тексту та патерну становить $(255/256)^m$. Це достатньо багато, якщо патерн короткий, однак якщо, наприклад, m=512, то ця величина становить лише приблизно 0,135. Ситуацію можна виправити якщо, наприклад, виконувати зсув за евристикою двох суміжних символів. Тоді для m=512 ця ймовірність стає більшою за 0,99. Ефективна реалізація цього підходу обговорюється в [4]: вираз $Z[T[i]]$ перетворюється на Z[word(T[i], T[i+1])], де функція word перетворює 2 послідовних байти пам'яті на двобайтне слово в регістрі процесора. У мові програмування C цю функцію можна реалізувати за допомогою механізмів перетворення типів даних, наприклад як *(unsigned short*)(T+i). Фактично, часова складність обчислення значень $Z[T[i]]$ та Z[word(T[i], T[i+1])] є однаковою, проте витрати пам'яті зростають із 256 байтів, потрібних для 1-байтного зчитування, до 64 Кб, потрібних для зберігання таблиці зсувів із 2-байтними індексами.

Однак для розумних довжин патерна, в межах 1000 символів, витрати пам'яті можуть бути суттєво знижені, й при цьому вплив на довжину зсуву буде незначним. Для цього в індексі таблиці зсувів достатньо використовувати не всі біти двобайтного слова. Наприклад, можна застосувати маску: Z[word(T[i]; T[i+1])&mask]. Якщо, скажімо, m=512 і маска *mask* містить 14 одиничних бітів, то ймовірність максимального зсуву для випадкових тексту та патерну становитиме $((2^{14}-1)/\ 2^{14})^{511}\approx 0{,}97$, що всього лише на 0,022 менше, ніж для двобайтного зчитування. Водночас таблиця зсувів міститиме 2^{14} елементів, а не 2^{16}, тобто буде меншою вчетверо. Звичайно, в пошуковому циклі необхідно буде виконувати додаткову операцію *&mask*, однак витрати на неї цілком компенсуються тим, що пошукова таблиця вміщуватиметься в кеш-пам'ять L1.

3. Подвійний цикл пропуску

Іншим недоліком алгоритму BMH є потреба перевіряти збіг патерна з вікном пошуку кожної ітерації пошукового циклу. Проте цієї операції можна уникнути, застосувавши так званий «цикл пропуску», що був вперше введений у [5], а згодом реалізований у низці алгоритмів. Особливо ефективним цей прийом стає, коли зсув виконується за евристикою більш ніж одного символу і ймовірність максимального зсуву є високою. Тоді точне значення довжини зсуву можна в таблиці зсувів не зберігати. Достатньо знати лише, чи є зсув певної фіксованої довжини, наприклад m чи m–1 безпечним, тобто таким, що не може призвести до пропуску патерна. Якщо це так, то ми виконуємо цей зсув фіксованої довжини, інакше перевіряємо збіг і обчислюємо значення подальшого зсуву за деяким іншим алгоритмом.

Враховуючи, що розглянутий у попередньому підрозділі зсув «за 1,5 байтами» з високою ймовірністю є максимальним, ми розвинемо ідею циклу пропуску, зробивши його подвійним. Базове припущення полягає в тому, що значення Z[word(T[i–1], T[i])] може сигналізувати про максимальний зсув з тією ж ймовірністю, що і значення Z[word(T[i], T[i+1])]. Це означає, що навіть якщо умова продовження циклу пропуску не виконується, ми можемо відступити 1 крок назад і, дуже ймовірно, продовжити цикл пропуску з цієї позиції. Такий подвійний цикл пропуску буде особливо ефективним для довгих патернів, коли ймовірність завершення циклу пропуску вище, а вплив кроку на 1 символ назад — нижче.

4. Довші зсуви

Припустимо, що T[i] та T[i+1] — два останніх символи вікна пошуку. Якщо значення Z[word(T[i]; T[i+1])&mask] свідчить про можливість максимального зсуву, це означає що пара символів T[i] та T[i+1] не може належати патерну. Проте символ T[i+1] може збігатися з

P[0] і тому вікно пошуку можна безпечно перемістити на *m*–1 символ вправо, але не на *m*. Якщо патерн короткий, то це зменшення довжини максимального безпечного зсуву може суттєво знижувати продуктивність алгоритму.

Цю ситуацію можна виправити, якщо всім елементам таблиці зсувів Z вигляду Z[word(c, P[0])&mask], $c \in \Sigma$, присвоїти значення, що сигналізує про немаксимальний зсув. Тобто якщо останній байт вікна пошуку міститиме символ *P*[0], то зсув вважатиметься немаксимальним, а величина максимального зсуву тоді може дорівнювати *m*. Здавалось би, це підвищить ймовірність немаксимального зсуву у випадковому тексті не більш ніж на 1/256, що небагато. Однак потрібно брати до уваги порядок зчитування байтів машинного слова із пам'яті в регістр процесора (endianness). Як уже зазначалося, функція word(T[i], T[i+1]) може бути реалізована у вигляді операції перетворення типів *(unsigned short*)(T+i). Однак результат цієї операції залежить від того, як байти машинного слова завантажуються з пам'яті у процесор: зліва направо (наприклад, у комп'ютерах Apple) чи справа наліво (наприклад, в IBM-сумісних комп'ютерах, x86). Зчитування справа наліво є небажаним, якщо індекс таблиці зсувів компонується з повного останнього байту вікна пошуку та неповного передостаннього, оскільки результуюче значення буде зсунуто до лівої межі двобайтного слова (рис. 1а), і обсяг пошукової таблиці стане таким самим, як і в разі використання таблиці зсувів із двобайтним індексом. Проте ситуація, зображена на рис. 1б — неповний останній байт вікна пошуку і повний передостанній — також є небажаною, оскільки, щоб зсув став немаксимальним, достатньо, щоб лише частина останнього байту вікна пошуку збігалася з частиною байту P[0], а отже, ймовірність немаксимального зсуву зростатиме не на 1/256, а на $1/2^{k-8}$, де *k* — кількість одиничних бітів у масці.

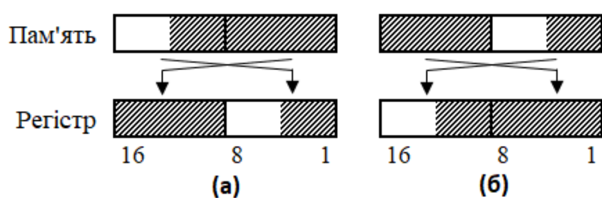

Рис. 1. Завантаження двобайтного слова з пам'яті до регістра процесора на комп'ютері зі зчитуванням байтів справа наліво: (а) повний молодший байт, неповний старший; (б) неповний молодший байт, повний старший.

Тим не менше, ситуація, зображена на рис. 1б, стає прийнятною, якщо пошук відбувається справа наліво: тоді збільшення максимального зсуву з *m*–1 до *m* байтів призводить до зниження його ймовірності не більше ніж на 1/256, а індекс таблиці зсувів формується з молодших бітів двобайтного слова. Ми називатимемо алгоритм пошуку справа наліво *реверсним* і позначатимемо його як *RZk*, де *k* — кількість одиничних бітів у масці.

Псевдокод алгоритму *RZk* наведено на рис. 2. На стадії передобчислень, крім основної пошукової таблиці *Z*, ініціалізується також допоміжна таблиця RQS, що реалізує зсув за алгоритмом [6], але в реверсному варіанті. Подвійний цикл пропуску реалізовано в рядках 11–16. Також припускається, що текст зліва доповнено «стоп-патерном».

```
1  mask ← 2^k–1;                          // Передобчислення
2  foreach i ∈ [0; 2^k) do Z[i] ← 1;
3  for i ← 0 to m–2 do
4    Z[word(P[i]; P[i + 1])] ← 0;
5  for i ← 0 to 2^(k–8) do
6    Z[(i << b) | P[m – 1]] ← 0;
7  foreach c ∈ Σ do RQS[c] ← m+1;
8  for i ← m–1 downto 0 do RQS[P[i]] ← i+1;
9  pos ← n;                               // Пошук
10 repeat
11   repeat
12     pos ← pos–m;
13     while Z[word(T[pos]; T[pos+1]) & mask]≠0 do
14       pos ← pos–m;
15     pos ← pos+1;
16   until Z[word(T[pos]; T[pos + 1]) & mask]≠0;
17   pos ← pos–1;
18   перевірити збіг у позиції pos;
19   pos ← pos – RQS[T[pos – 1]] + m;
20 until pos ≥ m;
```

Рис. 2. Реверсний пошуковий алгоритм *RZk*

5. Порівняння швидкодії алгоритмів

Наведений на рис. 2 алгоритм було реалізовано мовою C++ як в «реверсній», так і в «прямій» версії, а також застосовано до нього відому методику *вікон, що ковзають* (sliding windows). Кількість таких вікон у табл. 2 вказано після літери w. Реверсні алгоритми з максимальним зсувом вікна пошуку на *m* байтів позначено літерами RZ, а «прямі» алгоритми з максимальним зсувом вікна пошуку на *m*–1 байтів — літерою Z. Для порівняння були взяті найкращі з відомих алгоритмів у базовій версії,

із вікнами, що ковзають, а також, де це доречно, із застосуванням двобайтного читання. Час виконання найшвидших алгоритмів для кожної довжини патерна зафарбовано сірим.

Пошук виконувався у файлі обсягом 10 Мб, що являв собою архів англомовних текстів, створений одним із сучасних архіваторів. Результати усереднено за 500 пробігами кожного алгоритму на 40 різних комп'ютерах і нормовано таким чином, щоб значущість усіх комп'ютерів була однаковою.

Алгоритм / *m*	2	8	32	128	512
Z8-w2	40.63	19.64	14.58	12.52	12.35
RZ13	67.50	21.21	11.80	8.82	4.51
RZ14	67.92	21.16	11.85	8.92	4.40
RZ13-w2	49.34	15.90	10.36	8.33	3.50
RZ14-w2	49.59	15.97	10.47	8.41	3.41
RZ13-w3	49.45	15.67	9.59	7.93	3.86
RZ14-w3	49.78	15.91	9.74	8.02	3.70
RZ13-w5	50.89	16.04	9.50	8.09	4.32
RZ14-w5	51.38	16.31	9.70	8.19	4.06
Z13-w3	94.31	15.95	9.44	7.90	3.83
Z14-w3	99.79	16.21	9.60	8.03	3.71
RZ16-w2	49.87	17.94	10.86	8.29	4.43
FSBNDM	69.76	20.29	11.61	12.04	12.26
FSBNDM-2byte	90.33	30.55	16.47	17.02	17.13
GSBNDMq2	-	20.84	11.45	11.70	11.82
GSBNDMq2-2byte	-	29.15	15.67	16.06	16.19
SBNDMq2	130.04	21.44	11.50	11.69	11.82
SBNDMq2-2byte	199.07	37.28	16.89	17.32	17.43
SBNDM/BMH	152.10	43.68	19.86	20.38	20.51
LBNDM	118.87	38.63	18.93	13.83	7.33
FSw4	77.47	21.91	9.96	9.23	8.80
FSw6	62.46	18.23	9.65	8.97	7.56
FSw8	57.15	19.11	9.62	8.88	7.07

Табл. 2. Результати експериментів з вимірювання швидкодії алгоритмів (мс)

6. Висновки

Як свідчать результати експериментів, для всіх досліджуваних довжин патерна різні алгоритми із запропонованої родини мають вищу швидкодію, ніж усі відомі аналоги.

З огляду на широту експериментальної бази, ці результати можна вважати достатньо надійними.

Список використаних джерел

1. Faro S. The Exact Online String Matching Problem: a Review of the Most Recent Results. / S. Faro, T. Lecroq // ACM Computing Surveys (CSUR) Surveys Homepage archive. – 2013. – v. 45 (2), February 2013, Article No. 13.
2. Cantone D. Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm. / Cantone D., Faro S. // J. Automata, Languages and Combinatorics. – 2005. – v. 10(5/6), P. 589–608.
3. Horspool R. N. Practical fast searching in strings. / R. N. Horspool // Softw. Pract. Exp. – 1980. – v. 10 (6). – P. 501–506.
4. Peltola H. String matching with lookahead. / H. Peltola, J. Tarhio // Discrete Applied Mathematics. – 2014. – v. 163. – P. 352–360.
5. Hume A. Fast string searching. / A. Hume, D. Sunday // Softw. Pract. Exp. – 1991. – v. 21(11). – P. 1221–1248.
6. Sunday D. M. A very fast substring search algorithm. / D. M. Sunday // Commun. ACM. – 1990. – v. 33 (8). – P. 132–142.

References

1. FARO S., LECROQ T. (2013) The Exact Online String Matching Problem: a Review of the Most Recent Results, *ACM Computing Surveys (CSUR) Surveys Homepage archive*, 45(2), Art. 13.
2. CANTONE D., FARO S. (2005) Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm. *J. Automata, Languages and Combinatorics*, 10(5/6), pp. 589–608.
3. HORSPOOL R. N. (1980) Practical fast search in strings, *Softw. Pract. Exp.* 10(6), pp. 501–506.
4. PELTOLA H., TARHIO J. (2014) String matching with lookahead, *Discrete Applied Mathematics*, 163, pp. 352–360.
5. HUME A., SUNDAY D. (1991) Fast string searching, *Softw. Pract. Exp.* 21(11), pp. 1221–1248.
6. SUNDAY D. M. (1990) A very fast substring search algorithm, *Commun. ACM* 33(8), pp. 132–142.

Надійшла до редколегії 01.10.2019