Оптимізація Пошукового Алгоритму Боєра-Мура-Хорспула для Бінарних Даних

Ігор Завадський
Факультет кібернетики і комп'ютерних наук
КНУ ім. Шевченка
Київ, Україна
ihorza@gmail.com

Василь Остапчук
Факультет кібернетики і комп'ютерних наук
КНУ ім. Шевченка
Київ, Україна
vasylysk@gmail.com

The Boyer-Moore-Horspool Pattern Matching Algorithm Optimization for Binary Data

Igor Zavadskyi
Dept. Of Cybernetics and Computer Science
T. Shevchenko Kyiv National University
Kyiv, Ukraine
ihorza@gmail.com

Vasyl Ostapchuk
Dept. Of Cybernetics and Computer Science
T. Shevchenko Kyiv National University
Kyiv, Ukraine
vasylysk@gmail.com

***Анотація*—У статті описано метод пошуку підрядка у двійковому рядку. Метод являє собою адаптацію та оптимізацію алгоритму пошуку підрядка Боєра-Мура-Хорспула для бінарних даних. Середня величина зсуву вікна пошуку у запропонованому методі суттєво перевищує аналогічний показник для інших відомих адаптацій алгоритмів сімейства Боєра-Мура для бінарних даних.**

***Abstract*—The method of exact pattern matching on binary alphabet is presented. It represents the adaptation and optimization of well-known Boyer and Moore pattern matching algorithm for binary alphabet. The average search window shift length is essentially greater than in other known adaptation of Boyer and Moore algorithm.**

Ключові слова—алгоритм Боєра-Мура; пошук патерна; бінарні дані; зсув вікна пошуку

Keywords—Boyer-Moore; Pattern matching; binary data; search window shift

I. Вступ

Однією з найактуальніших задач кібернетики протягом останніх десятиліть є розробка ефективних алгоритмів пошуку підрядка (патерна) в тексті. Більша частина відомих на сьогодні алгоритмів в цій галузі в тій чи іншій мірі використовує ідеї алгоритму Боєра-Мура [1]. Йдеться про три основні ідеї:

1. Сканування тексту зліва направо, порівняння символів вікна пошуку справа наліво.
2. Зсув вікна пошуку за евристикою стоп-символу.
3. Зсув вікна пошуку за евристикою його суфікса.

Алгоритм Боєра-Мура-Хорспула [2] являє собою спрощений варіант алгоритму Боєра-Мура. Він не включає в себе евристику суфікса. Цей алгоритм і дотепер залишається одним з найефективніших для текстів на великих алфавітах (64 символи і більше).

Однак чи не найголовніший недолік сімейства алгоритмів Боєра-Мура полягає в тому, що на малому алфавіті величина зсуву за евристикою стоп-символу є в загальному випадку невеликою. Розглянемо більш конкретний приклад, а саме алфавіт, що складається з 0 та 1, тобто бінарні дані. Використання евристики стоп-символу на таких даних, крім крайніх випадків (патерн складається з довгих послідовностей вигляду 0…0 та 1…1), даватиме мінімальну величину зсуву, тобто 1 символ.

Цю проблему зазвичай вирішують, розглядаючи при порівнянні замість конкретного біта групу бітів, тобто машинне слово (надалі ми вважатимемо машинним словом один байт, що складається з 8 бітів, хоча загалом його величину можна варіювати як завгодно). Так, у [3] розглянуто варіант алгоритму пошуку патерна за алгоритмом Боєра-Мура для бінарних даних, який не використовує ніякі бітові маніпуляції, а працює лише з байтами. Тим самим значно зменшується кількість необхідних порівнянь. Однак запропонований у [3] метод недостатньо оптимізований і в цій статті ми опишемо, як виправити його вади.

Слід зауважити, що є кілька алгоритмів пошуку підрядка, які на бінарному алфавіті ефективніші за алгоритм Боєра-Мура-Хорспула, зокрема алгоритм Shift-And [4] для коротких патернів та Hashq [5] для патернів середньої довжини. Однак у них, як і в оригінальних версіях алгоритмів [1] і [2], не враховується можливість одночасної обробки байтів тексту. Більше того, виконання побітових операцій є невід'ємною складовою алгоритмів [4]–[5], і тому існування їхніх «байтових» версій видається проблематичним.

Ще один підхід може полягати в опрацюванні байтів як елементів 256-символьного алфавіту, однак його очевидним недоліком є неможливість пошуку патернів, що включають частини байтів.

II. Ідея методу

Нехай *t[i]* і *p[i]* - *i*-ий біт тексту та патерна відповідно. Позначатимемо *i*-ий байт тексту через text[i], а *j*-ий байт патерну після відступу на *sh* біт вправо – через *pat[sh, j]*. Оскільки довжини патерну й тексту можуть бути не кратними байту (рис. 1.), для патерна використовується маска *mask[sh,j]*, як і в [3].

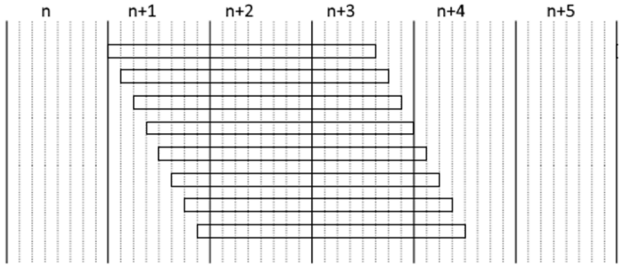

Рис. 1. Зсуви патерна відносно меж байта тексту

Маска показує, які біти конкретного байта патерна при порівнянні є значущими. Наприклад, *mask[0,3] = 11111000, mask[0,2] = 11111111, mask[2,1] = 00111111*. Операція перевірки збігу *i*-го байту тексту з *j*-м байтом патерна з відступом *sh* виглядає так:

$$(text[i] \text{ XOR } pat[sh,j] \text{ AND } mask[sh,j]) == 0.$$

Згідно з алгоритмом Боєра-Мура, величина зсуву вікна пошуку за евристикою стоп-символу залежить від позиції найбільш правого входження останнього символу вікна пошуку в патерн, а в середньому – від частоти входження певного символу алфавіту в патерн. Для бінарних алфавітів символом слід вважати байт. Однак при порівняні останнього байта в патерні ми обмежені його маскою і порівнюємо насправді не 8 бітів, а, залежно від розміщення патерна, від 1 до 8. Середня величина зсуву за такою евристикою є набагато меншою, ніж у разі порівняння 8 бітів. Це і є головним недоліком методу [3].

Основна ідея даної роботи полягає в тому, щоб базувати евристику стоп-символу не на останньому байті, який може бути неповним, а з передостанньому, який гарантовано є повним. Оскільки неповний останній байт до перевірки не залучається, використання евристики суфікса є не можливим. Тому наш алгоритм є модифікацією для бінарних текстів алгоритму Боєра-Мура-Хорспула, у якому не використовується евристика суфіксу, на відміну від [3], де до бінарних даних адаптовано саме класичний алгоритм Боєра-Мура.

Пояснимо переваги нашого методу детальніше. Ймовірність того, що в конкретній позиції випадкового тексту зустрінеться конкретний 8-бітовий байт випадковим чином згенерованого патерну становить *1/256*. Якщо припустити, що всі зсуви останнього байта патерну відносно байта тексту відбуваються з однаковою ймовірністю, то ймовірність того що останній не повний байт зустрінеться в тексті дорівнюватиме *1/8(1/2+1/4+1/8+…+1/256) = 255/256*8 = 31,875/256.* Тобто при порівнянні неповного байта ймовірність того, що він трапиться в певній позиції патерна збільшується, а отже, зменшується середня величина зсуву.

Однак у випадку, коли розмір патерна менший або рівний двом байтам, середня величина зсуву для нашого алгоритму є меншою, ніж для алгоритму [3], оскільки передостанній байт може бути неповним і сенс використання його як бази для евристики зсуву втрачається.

III. Алгоритм

На стадії передобчислень будуються таблиці *delta1[sl,b]* (*sl* $\leq$*bytesize i* $0 \leq b < 2^{sl}$) та *Correction[sh,j]*. Перша з них – це таблиця довжин зсуву вікна пошуку за евристикою стоп-символу. *delta1[sl, b] = paternLength−l*, де *l* це індекс останнього біта найправішого входження в патерн *sl*-бітового значення *b*. Для прикладу, нехай *sl* = 4 і *P = 0010101011101101*, найправіше входження *b=5=0101* підкреслене, тоді *delta1[*4,5*] = 16-9=7.* Але слід розглянути ще один варіант, коли *b* не трапляється в патерні, але суфікс *b* є префіксом патерна. Для прикладу при *sl=5* і *b=17=10001*, *b* не входить в патерн, але його суфікс є префіксом патерна. Тоді патерн не може бути зсунутий на свою максимальну довжину, тобто *delta1[*5,17*]=16–3=13*, а не *16*. Алгоритм ініціалізації таблиці *delta1* наведено

нижче (значення по замовчуванню для *delta1[sl,b] = paternLength*).

Для збігу з префіксом патерна:

```
for i=1 to sl-1
  for всі значення b такі, що p[1]…p[i] є суфікс b
    delta1[sl,b] = paternLength-i
```

В усіх інших випадках:

```
for i =1 to paternLength-sl
  b=p[i]…p[i+sl-1]
  delta1[sl, b] = paternLength–i-sl+1.
```

Також позначимо через *last[sh]* номер останнього байту патерна після зсуву *sh*. Формально *last[sh] = (sh+ patternLength -1)/bytesize +1*. Для прикладу на рис 1. *last[sh] = 3* для *sh=0,1,2,3* і *last[sh] = 4* для *sh = 4,5,6,7*.

Через *sl* (рис. 2.) позначимо кількість біті в останьому байті патерна після зсуву *sh*: *sl = (sh+patternLength-1) mod bytesize + 1*. *sl* буде обчислюватися в ході виконання програми, тому для цієї величини таблиця не створюється.

Рис. 2. Схематичне зображення *sl* та *sh*

В оригінальному алгоритмі Боєра-Мура [1] таблиця *delta1* застосовувалася незалежно від того, який байт вікна пошуку не дорівнює відповідному байту в патерні, тому що *delta1* показувало насправді величину, на яку поточний покажчик на текст *i* міг би бути збільшений, а не кількість символів патерну, на які можна зсунути вікно пошуку. У бінарному варіанті *delta1* дорівнює максимально можливому зсуву для вікна пошуку, якщо рахувати від кінця вікна пошуку. Тому потрібно ввести ще одну таблицю констант *Correction[sh,j],* яку будемо використовувати, якщо не співпадіння відбулося не на останньому байті. *Correction[sh,j]* показує на скільки далеко від кінця патерна відбулося не співпадіння і на скільки вліво (тобто назад) ми повинні зсунути патерн, якщо не співпадіння відбулося на *j*-ому байті:

Correction[sh,j] = sl +(last[sh]−1−j) bytesize.*

Наведемо приклад, коли потрібно використовувати таблицю *Correction*. Припустимо, відбувається порівняння *i*-ого байта вікна пошуку та відповідного патерна, як показано на рис. 3. Позначимо цей байт тексту через *x*. У випадку, коли байт *x* не збігається з відповідним байтом патерна, а ми повинні зсунути вікно пошуку.

Рис. 3. Приклад використання таблиці *Correction*

Для цього ми використовуємо таблицю *delta1*, яка показує величину зсуву вікна пошуку у випадку, коли останній байт вікна пошуку не збігається з останнім байтом патерна. Ця величина дорівнює відстанню від кінця патерна до найправішого входження байта *x* в патерн, або довжині патерна, якщо такого входження немає. Припустимо, що байт *z* патерна є найправішим входженням байта *x* в патерн, тоді *delta1[sl,x] = dist(z), dist(z)* – це відстань від кінця патерна до *z*. Але якщо зсунути патерн на таку відстань, байт *x* не збігатиметься з байтом *z*, тому потрібно скоректувати зсув таким чином, щоб байт *x* опинився над байтом *z*. Зрозуміло, що величина зсуву тоді дорівнює *dist(z) – dist(x)*. Але *x* – це *i*-ий байт поточного вікна пошуку, тоді *dist(x) = Correction[sh, i]*, звідси величина зсуву можна обчислити так:

$$dist(z) - dist(x) = delta1[sl,x] - Correction[sh, i] = delta1[sl,text[i]] - Correction[sh, i]$$

Зрозуміло, що за такою формулою можливий варіант коли величина зсуву буде від'ємною, наприклад, коли байт *y* дорівнює *x*. У цьому випадку відстань до байта *y* є меншою за відстань до байта *x* і для збігу *x* з *y* потрібно патерн назад. Однак даний варіант розміщення патерна був уже відкинутий або розглянутий на попередньому етапі алгоритму. Тому, щоб уникнути цього небажаного зсуву, назад в алгоритм була додано перевірка (*d1* – величина зсуву після корекції):

```
if(d1<=0)
      d1=1
```

Згідно з основною ідеєю даної роботи, на стадії пошуку байти вікна пошуку звіряються з патерном від передостаннього до першого і лише в разі збігу їх усіх перевіряється збіг останнього неповного байта вікна пошуку із відповідним байтом патерна. Таким чином, ми не оперуємо значенням останньому байту, крім випадку, коли ми перевірили уже всі байти від передостаннього до першого і повернулися до останнього.

Наведемо формальний опис алгоритму за допомогою псевдокоду.

```
sh = 0
i = last[0]
lastbit = patternLength
// Довжина тексту в байтах
while i <= (textLength-1)/byteSize+1
  // Починаємо перевірку з передостаннього байта
  j = last[sh]
  oldj = j, oldi = i
  j = j – 1, i = i - 1
```

```
sl=1+(patternLength+sh-1)mod byteSize
// indic вказує, чи збігається поточний байт тексту і патерна
// перевіряємо байти від передостанього до першого
indic = (text[i] XOR pat[sh, j])
AND mask[sh, j]
while j > 0 AND indic == 0
   i = i – 1
   j = j - 1
   if (j != 0)
      indic = (text[i] XOR
      pat[sh, j]) AND mask[sh, j]
// якщо indic = 0, то збігаються всі байти, від передостанього до першого
if indic == 0 // тоді перевіряємо останній байт
   indic = (text[oldi] XOR
   pat[sh,oldj]) AND mask[sh, oldj]
   if(indic==0)
      report match (byteSize*i+sh)
      lastbit++
   else
                  // інакше обчислюємо зсув за евристикою стоп-символу
                  i = oldi
                  j = oldj
                  b = (text[i]/1<<(byteSize - sl)) mod (1<<K)
      d1 = delta1[min(sl, K), b]
      if(d1<=0)   d1=1
      lastbit = lastbit + d1
else
   // якщо немає збігу, обчислюємо зсув за евристикою стоп-символу
   d1 = delta1[K, text[i] mod
      (1<<K)]– Correction[sh, j]
   if(d1<=0)
      d1=1
   lastbit = lastbit + d1
i = ((lastbit - 1) / byteSize) + 1
sh = (lastbit - patternLength)      mod byteSize
```

Обсяг пам'яті, необхідної для зберігання таблиці *delta1*, можна обчислити за формулою (1).

$$\sum_{sl=1}^{byteSize} 2^{sl} = 2^{byteSize+1} - 2 \qquad (1)$$

Тобто він збільшується експоненційно, однак залишається допустимим для значень *byteSize* 8 або 16. Але вже при *byteSize* = 32 зберігати такий обсяг даних може виявитися складно. Тому було введено додаткову змінну *K*, яка за замовчуванням дорівнює *byteSize*, але може бути зменшена. Вона показує, скільки бітів із кожного байту беруть участь в обчисленні максимально можливого зсуву, що дає змогу під час обчислення *delta1* не обчислювати значення *sl*, які більші за *K*, і суттєво зекономити пам'ять ціною зменшення величини зсуву.

IV. Обчислювальний експеримент

Експеримент було проведено на випадковим чином згенерованих текстах довжиною 40000 бітів та патернах довжиною від 10 до 1000 бітів. На рис. 4 зображено середнє значення величини зсуву залежно від довжини патерна. Продемонстровано дані для 3 алгоритмів:

1. ABM – алгоритм Боєра-Мура, адаптований для бінарних даних, згрупованих у байти [3].
2. AHBM – алгоритм Боєра-Мура-Хорспула, адаптований для бінарних даних, згрупованих у байти.
3. AHBM-ZV – запропонований нами алгоритм.

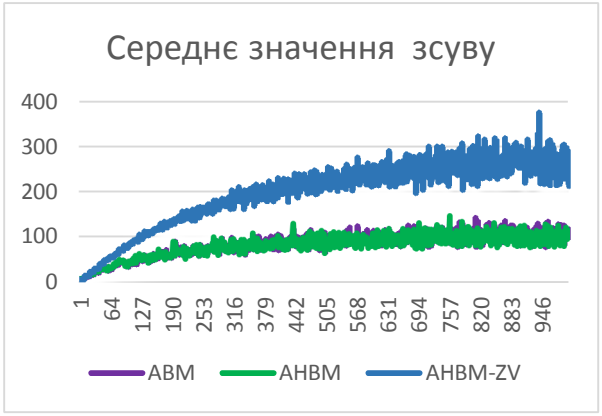

Рис. 4. Середня величина зсуву вікна пошуку

V. Висновки

Як свідчить комп'ютерний експеримент, середня величина зсуву вікна пошуку в нашому алгоритмі для достатньо довгих патернів перевищує цю величину в алгоритмі [3] більш ніж удвічі. Отже, запропонований алгоритм є ефективною адаптацією алгоритму Боєра-Мура-Хорспула для пошуку підрядка у двійковому тексті, якщо довжина підрядка перевищує 32 біти.

Зазначимо, що запропонований алгоритм не оптимальний і є кілька можливих шляхів його вдосконалення. Найбільш перспективним із них видається використання лише тих зсувів патерна, які вирівнюють його початок на межу байтів. Це дасть змогу уникнути операцій, що визначають положення початку вікна пошуку всередині байта.

Література References

[1] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Comm. ACM 20 (1977) 762–772.

[2] R.N. Horspool, Practical fast searching in strings, Software — Practice and Experience 10 (1980) 501–506.

[3] S.T. Klein, M. Ben-Nissan. (2007) Accelerating Boyer– Moore Searches on Binary Texts. Proc. 12th Int. Conf. Implementation and Application of Automata (CIAA 2007), Lecture Notes in Computer Science 4783, 130–143.

[4] S. Wu, U. Manber Fast text searching allowing errors. Commun. ACM, 35(10):83–91, 1992.

[5] T. Lecroq Fast exact string matching algorithms. Inf. Process. Lett., 102(6):229–235, 2007.