Suffix Type String Matching Algorithms Based on Multi-windows and Integer Comparison

Hongbo Fan[1,2], Shupeng Shi[1,2], Jing Zhang[1,2(✉)], and Li Dong[1,2]

[1] Department of Computer Science,
Kunming University of Science and Technology, Kunming 650500, China
[2] Computer Technology Application Key Laboratory of Yunnan,
Kunming 650500, Yunnan, China
{hongbofan, shupengshi, jingzhang, lidong,
270677673}@qq.com

Abstract. In this paper, 3 classic suffix type algorithms: QS, Tuned BM and BMH*q* were improved by reducing the average cost of basic operations. Firstly, the multi-windows method was used to let the calculations of the jump distance run in parallel and pipelining. Secondly, the comparison unit was increased to integer to reduce the total number and the average cost of comparisons. Especially for BMH*q*, the jump distance was increased by good prefix rule and the operations to get the jump distance were simplified by unaligned integer read. Thus, 3 algorithms named QSMI, TBMMI and BMH*q*MI were presented. These algorithms are faster than other known algorithms in many cases.

Keywords: String matching · Single pattern · Multi-windows · Integer comparison algorithm

1 Introduction

String matching is the performance bottleneck and key issue in many important fields. The design of exact single pattern matching algorithm owns very important significance. Especially in our focus real-time information processing and security field, high performance matching is strongly demanded.

In the string $S = s_0 s_1 \ldots s_{m-1}$, for $0 < k \leq m$, we denote the prefix, suffix and factor of S of length k as $pref(S,k)$/$suff(S,k)$/$fac(S,k)$. SCW is used to denote the text in the slide window. All algorithms in this paper are belonging to exact single pattern matching algorithm, which means that for given alphabet Σ ($|\Sigma|=\sigma$, Σ^* is closure of Σ), and for given text $T = t_0 t_1 \ldots t_{n-1}$ of length n/patten $P = p_0 p_1 \ldots p_{m-1}$ of length m, P, $T \in \Sigma^*$, seeking the window that $P[i] = SCW[i]$ for $\forall i \in [0 \ldots m-1]$ in all possible sliding window. Algorithms are described in C/C++.

This paper improved three classical suffix matching algorithms: Quick Search [1], Tuned BM [2] and BMH*q* [3]. We added Multi-window [4] and presented an integer comparison method in them. Thus the three series of algorithm named QSMI, TBMMI and BMH*q*MI were presented, and they are very fast for short patterns.

S. Qing et al. (Eds.): ICICS 2015, LNCS 9543, pp. 414–420, 2016.
DOI: 10.1007/978-3-319-29814-6_35

2 Accelerating Method: Multi-window and Integer Comparison

Multi-window [4] (shown in Fig. 1) let the text be equally divided as $k/2$ areas in k window mechanism (k is even). Each area has two windows and respectively matches from both ends toward the middle region until they are overlapped and each window matching procedure by tunes. It is a general accelerate method for string matching.

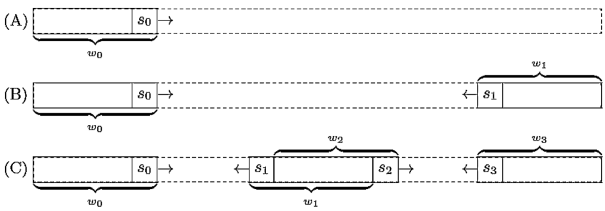

Fig. 1. Mechanism of two windows and multiple windows

There are many compares in suffix marching. Let the delay of branch prediction failure be signed *punishment*. The average character compare branch cost is about $1 - \sigma^{-1} + \sigma^{-1} * punishment$, e.g., 10.5 ticks on DNA sequence on Prescott. If unaligned read $pref(SCW, w)$ into an integer, compare with the integer of $pref(P, w)$. Only when they are equal, other compares are needed. One integer comparison is equivalent to w times of character comparison and the average cost of branch is reduced to $1 - \sigma^{-w} + \sigma^{-w} * punishment$. To compare uint16_t/uint32_t on Prescott and DNA sequence, the average branch cost will obviously reduce to 3.27/1.15 ticks.

3 Improved Algorithms Based on QS, Tuned BM and BMHq

By introducing above method into QS [1], a new algorithm called QSMI_*wkXc* was presented, which k is the number of windows and X is the integer type for comparison: S:short/uint16_t, I:int/uint32_t, L:long long/uint64_t. The code of QSMI_w4Ic is listed as Algorithm 1.

Algorithm 1. Algorithm of QSMI_w4Ic .

```
QSMI_w4Ic (P, T, m, n) {if m<4 then return ("Too short.");
for c∈∑ do qsBC[c]←qsBCr[c]←m+1;
for i∈[0……m-1] do qsBC[P[i]]←qsBCr[P[m-1-i]]←m-i;
s0←0;s1←n/2-1;s2←n/2;s3←n-m;
uint32_t IC; IC←*(uint32_t*)P;
while s0<s1 and s2<s3 do{
 if IC = *(uint32_t*)(T+s0) then
     if memcmp(P+4, T+4+s0, m-4)=0 then Report(s0);
if IC = *(uint32_t*)(T+s1) then
     if memcmp(P+4, T+4+s1, m-4)=0 then Report(s1);
if IC = *(uint32_t*)(T+s2) then
     if memcmp(P+4, T+4+s2, m-4)=0 then Report(s2);
if IC = *(uint32_t*)(T+s3) then
     if memcmp(P+4, T+4+s3, m-4)=0 then Report(s3);
s0←s0+qsBC[T[s0+m]];s1←s1-qsBCr[T[s1-1]];
s2←s2+qsBC[T[s2+m]];s3←s3-qsBCr[T[s3-1]];}
matching T[s0…s1+m-1] and T[s2…s3+m-1] by QSMI_w2Ic;}
```

By introducing continuous jump method of Tuned BM into QSMI, TBMMI was proposed. Firstly, determine whether the window match occurs by integer comparison in the each window. And then, bad character jumping of Quick Search continuous jump once and bad character jumping of Horspool jump several times. We use once QS jump and twice Horspool jump twice in the TBMMI. TBMMI_w4Ic that is obtained only by the bad character jump table of QS are shown as Algorithm 2.

We improved BMH*q* [3], by using good-prefix rule to increase the jump distance, unaligned read to reduce read operation and add the method in Sect. 3, an algorithm named BMH*q*MI was proposed. BMH2MI_w4Ic is shown in Algorithm 3.

Algorithm2. Algorithm of TBMMI_w4Ic

```
TBMMI_w4Ic (P, T, m, n) {
 if m<4 return ("Too short."); IC←*(uint32_t*)P;
for c∈∑ do qsBC[c]←qsBCr[c]←m+1;
for  i∈[0……m-1]do qsBC[P[i]]←qsBCr[P[m-1-i]]←m-i;
s0←0;s1←n/2-1; s2←n/2;s3←n-m;uint32_t IC;
while s0<s1 and s2<s3 do{
if IC = *(uint32_t*)(T+s0) then
   if memcmp(P+4, T+4+s0, m-4)=0 then Report(s0);
if IC = *(uint32_t*)(T+s1) then
   if memcmp(P+4, T+4+s1, m-4)=0 then Report(s1);
if IC = *(uint32_t*)(T+s2) then
   if memcmp(P+4, T+4+s2, m-4)=0 then Report(s2);
if IC = *(uint32_t*)(T+s3) then
   if memcmp(P+4, T+4+s3, m-4)=0 then Report(s3);
s0←s0+qsBC[T[s0+m]];s1←s1-qsBCr[T[s1-1]];
s2←s2+qsBC[T[s2+m]];s3←s3-qsBCr[T[s3-1]];
s0←s0+qsBC[T[s0+m-1]]-1;s1←s1-qsBCr[T[s1]]+1;
s2←s2+qsBC[T[s2+m-1]]-1;s3←s3-qsBCr[T[s3]]+1;
s0←s0+qsBC[T[s0+m-1]]-1;s1←s1-qsBCr[T[s1]]+1;
s2←s2+qsBC[T[s2+m-1]]-1;s3←s3-qsBCr[T[s3]]+1;}
matching T[s0…s1+m-1] and T[s2…s3+m-1] by TBMMI_w2Ic;}
```

When $suff(SCW, q-1) \notin fac(P)$, BMH$q$ make the window slide from the win0 to the win1 show as Fig. 2. If $suff(SCW, q-1) \neq pref(P, q-1)$ and win1 can not matching. So the window should keep sliding until find the first k satisfy $suff(SCW, k) = pref(P, k)$ (the window get extra jump to the win2).

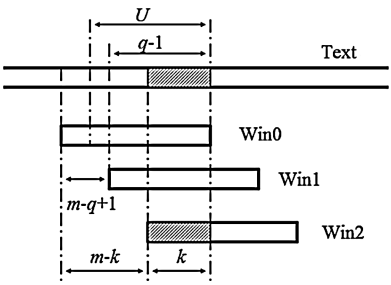

Fig. 2. Increase jump distance by good-prefix method.

To store the jump distance for q-grams needs q-Dimension table, which a table lookup need q times read. Unaligned read can simulate original q-Dimensional table lookup by once read and table lookup. Since on little-endian processor, *(uint16_t*) $(T + i + m - 2) = T[i+m-2] + b = T[i+m-1] * 256$. If the 2-Dimensional jump distance table is *shift*, build a 1-Dimensional table *shift*1D and for $\forall a, b \in \Sigma$, $shift1D[a+b*256] = shift[a][b]$. So, $shift1D[*(\text{uint\_16}*)(T+i+m-2)] = shift[a][b]$. If the read string is $T[i + m - 2 \ldots i + m]$ for $q = 3$, * (uint32_t*)$(T + i+m\text{-}2)$ &0x00ffffffu = $T[i + m - 2]$ + (int)$T[i + m - 1] * 256$ + (int)$T[i + m] * 65536$ can be used.

Algorithm3. Algorithm of BMH2MI_w4Ic

```
BMH2MI_w4Ic (P, T, m, n) {if m<4 return ("too short.");
uint32_t a, b, IC←*(uint32_t*)P, m1←m-1;
for a∈∑, b∈∑  do qs2BC[a+b*256]← qs2BCr[a+b*256]←m+1;
for a∈∑ do{qs2BC[P[m-1]+a*256]←qs2BCr[P[0]+a*256]←m;}
for i∈[0,……m-2] do { qs2BC[P[i]+(int)P[i+1]*256]←m-i-1;
qs2BCr[P[m-1-i]+(int)P[m-2-i]*256]←m-i-1;}
s0←n/2-1;s2←n/2;s3←n-m;
while s0<s1 and s2<s3 do{
 if IC = *(uint32_t*)(T+s0) then
     if memcmp(P+4, T+4+s0, m-4)=0 then Report(s0);
if IC = *(uint32_t*)(T+s1) then
     if memcmp(P+4, T+4+s1, m-4)=0 then Report(s1);
if IC = *(uint32_t*)(T+s2) then
     if memcmp(P+4, T+4+s2, m-4)=0 then Report(s2);
if IC = *(uint32_t*)(T+s3) then
    if memcmp(P+4, T+4+s3, m-4)=0 then Report(s3);
s0←s0+qs2BC[*(uint16_t*)(T+s0+m1)];
s1←s1-qs2BCr[*(uint16_t*)(T+s1-1)];
s2←s2+qs2BC[*(uint16_t*)(T+s2+m1)];
s3←s3-qs2BCr[*(uint16_t*)(T+s3-1)];}
matching T[s0…s1+m-1] and T[s2…s3+m-1] by BMH2MI_w2Ic;}
```

4 Experiment and Results

We did the following experiment based on SMART 13.02 [6], it gave the implements of most known algorithms (in EI or SCI paper) as of Feb. 2013. The platform of this experiment is Intel Core2 E3400 @ 3.0 GHz/Ubuntu 12.10 64 bit desktop/g++4.6/-O3 optimization. The tested texts include three samples of text [8] listed as follow: DNA sequence (E.coil), pure English text (Bible.txt) and the sample of English nature language (world192.txt). This experiment compared all algorithms in SMART 13.02 and added some newer algorithms not be included in SMART, such as SBNDM*q*b [9], GSB2b [9], FSO [10], HGQSkip [11], *k*SW*x*C [12], SufOM [13], Greedy-QF [14], etc. If an algorithm with different parameters are called different algorithms, there were more than 1000 algorithms are compared, which covered most of known algorithms. The experiment data (dozens of thousands of records) can not be listed all. In this paper only list the highest performance of three algorithms under each match condition. The data of experiment show as Table 1 and the unit is MB/s.

Table 1. Matching speed of the fastest 3 algorithms and their optimal parameters

	m=4		*m*=8		*m*=16	
World192	**TBMMI_w4Ic**	**3766.3**	**TBMMI_w4Ic**	**5254.2**	**QSMI_w4Lc**	**5736.2**
	QSMI_w4Ic	**3503.1**	**QSMI_w4Lc**	**5034.3**	**TBMMI_w4Lc**	**5602.4**
	SBNDM2_2_sbi32	3007.7	SBNDM2_2_sbi32[16]	4610.4	SBNDM4_sbi32[9]	5556.2
Bible.txt	**TBMMI_w4Ic**	**3531.8**	**TBMMI_w4Ic**	**5005.9**	**TBMMI_w4Lc**	**5604.5**
	QSMI_w4Ic	**3394.2**	**QSMI_w4Lc**	**4747.5**	SBNDM4_4_sbi32	5600.4
	*k*SW*x*C_k6xI[12]	2945.6	SBNDM2_2_sbi32	4162.7	**QSMI_w4Lc**	**5516.6**
E.coli	**BMH2MI_w4Ic**	**2799.6**	**BMH2MI_w4Lc**	**4570.8**	**BMH2MI_w4Lc**	**5310.5**
	*k*SW*x*C_k4xI	2398.6	*k*SW*x*C_k4xL	3520.2	SBNDM4_3_sbi32	5252.3
	UFNDM4b_a64[9]	1187.9	SBNDM4_2_sbi32	3138.4	FSBNDMqb_q6f2i32[9]	4926.3

5 Conclusion

In this paper, three classical suffix match algorithms QS/TBM/BMH*q* are improved by introduce the method of Multi-window and unaligned read integer comparison, and three suffix match algorithms named QSMI/TBMMI/BMH*q*MI were proposed. It is shown in experiment results that these algorithms are faster than other known algorithm under multiple match conditions for matching short patterns.

6 Acknowledgements

This paper is supported by National Natural Science Foundation of Yunnan, China under Grant 2012FB131 and 2012FB137, Key Project of National Natural Science Foundation of Yunnan, China under Grant 2014FA029, and National Natural Science Foundation of China under Grant 61562051.

References

1. Daniel, M.S.: A very fast substring search algorithm. Commun. ACM **33**(8), 132–142 (1990)
2. Andrew, H.: Fast string searching. Softw. Pract. Exp. **21**(11), 1221–1248 (1991)
3. Kalsi, P., Hannu, P., Jorma, T.: Comparison of exact string matching algorithms for biological sequences. BIRD 2008, pp. 417–426. Springer, Berlin (2008)
4. Faro, S., Lecroq, T.: A multiple sliding windows approach to speed up string matching algorithms. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 172–183. Springer, Heidelberg (2012)
5. Horspool, R.N.: Practical fast searching in strings. Softw. Pract. Exp. **10**(6), 501–506 (1980)
6. SMART: string matching research tools. http://www.dmi.unict.it/∼faro/smart/
7. Simone, F., Simone, F., Thierry, L.: The exact online string matching problem: a review of the most recent results. ACM Comput. Surv. **45**(2), 13:1–13:42 (2013)
8. The large canterbury corpus. http://corpus.canterbury.ac.nz/descriptions/
9. Hannu, P., Jorma, T.: Variations of forward-SBNDM. In: PSC2011, pp. 3–14. Czech Technical University, Prague (2011)
10. Fredriksson, K., Grabowski, S.: Practical and optimal string matching. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 376–387. Springer, Heidelberg (2005)
11. Wu, W., Fan, H., Liu, L., Huang, Q.: Fast string matching algorithm based on the skip algorithm. ICM 2012. LNEE, vol. 236, pp. 247–257. Springer, New York (2013)
12. Lv, Z., Fan, H., Liu, L., Huang, Q., et al.: Fast single pattern string matching algorithms based on multi-windows and integer comparison. In: IET International Conference on ICISCE 2012, pp. 1–5 (2012). doi:10.1049/cp.2012.2326)
13. Fan, H., Yao, N.: Tuning the EBOM algorithm with suffix jump. ICITSE 2012. LNEE, vol. 211, pp. 965–973 (2013)
14. Chen, Z., Liu, L., Fan, H., Huang, Q., et al..: A fast exact string matching algorithms based on greedy jump and QF. ICISCE 2012. In: IET International Conference (2012). doi:10.1049/cp.2012.2320
15. Fan, H., Yao, N.: Q-gram variation for EBOM. In: Proceedings of the 2012 International Conference on Information Technology and Software Engineering. LNEE, vol. 211, pp. 453–460 (2013)
16. Branislav, D., Jan, H., Hannu, P., Jorna T.: Tuning BNDM with q-grams. In: ALENEX 2009, pp. 29–37. SIAM, New York (2009)