COT6405 -Analysis Of Algorithms

Algorithm Analysis and Implementation - Pattern

Written by:

Christopher Foley
Z15092976

Academic Year: 2016-2017

Table of Contents

1 Summary....3
2 Methodology....3
3 Analysis....3
1 CharChar and StrStr (Naive processing)....6
2 Knuth-Morris-Pratt (KMP)....6
3 Rabin-Karp....7
4 Conlusions....8
5 Appendix – Code Samples....8
1 StringMatch.cpp....8
2 StrStr.c....10
3 StrStrRK.cpp....11
4 StrStrKMP....13
5 Addenum....15

1 Summary

Substring matching is a critical part of modern design. Documents and text readers need to be searched for specific values. Network devices deliver their results in a text based format. Information is entered and validated against known patterns. It was the purpose of this project to investigate algorithms for substring matching in string buffers of varying size. Contents of the string buffer were restricted to random printable ASCII characters. Search strings of a fixed length were created and average search time was calculated. To force worst case scenario, the search string will contain a non printing character which will not match the buffer. Tests will be performed numerous times on each generated buffer and the worst case search time and buffer size will be written to standard output.

This e goal of the project will be to compare substring matching times of two and possibly three algorithms:

- naive matching
- Rabin-Karp
- Knuth-Morris-Pratt

Data was collected concerning string matching times for randomly determined non matching strings. The buffer size will be the independent variable and the search times will be compared.

It should be noted that the algorithms were compared to each other in the tables to avoid a problem that was being observed in the LibreOffice calc system, in which the Calc software appeared to “hang” when tables were moved. Combining the data allowed easier analysis and gave direction to other tests (Character to Character was optimized and retested).

2 Methodology

Contents of the target string buffer was initialized to random printable ASCII characters. Pattern strings were created and average search time calculated. To force worst case scenario, the pattern string was marked with a character which was specifically excluded from the source buffer. Tests will be performed 16 times on each generated buffer and the worst case search time and buffer size will be written to standard output.

3 Analysis

The following chart indicates the search times for certain buffer sizes. As can be seen each performed similarly. Three different algorithms were tested in addition to the C++ library procedure for matching. All times were computed using system library routines which maintained a granularity in the microsecond (us.) range. The algorithm abbreviations are noted as below:

Abbreviation	Algorithm
CharChar	Character by character matching
KMP	Knuth-Morris-Pratt
RK: Rx=96, q=127	Rabin-Karp algorithm, Radix=96, multiplier (q) = 127
Strstr	C++ <cstring> library procedure

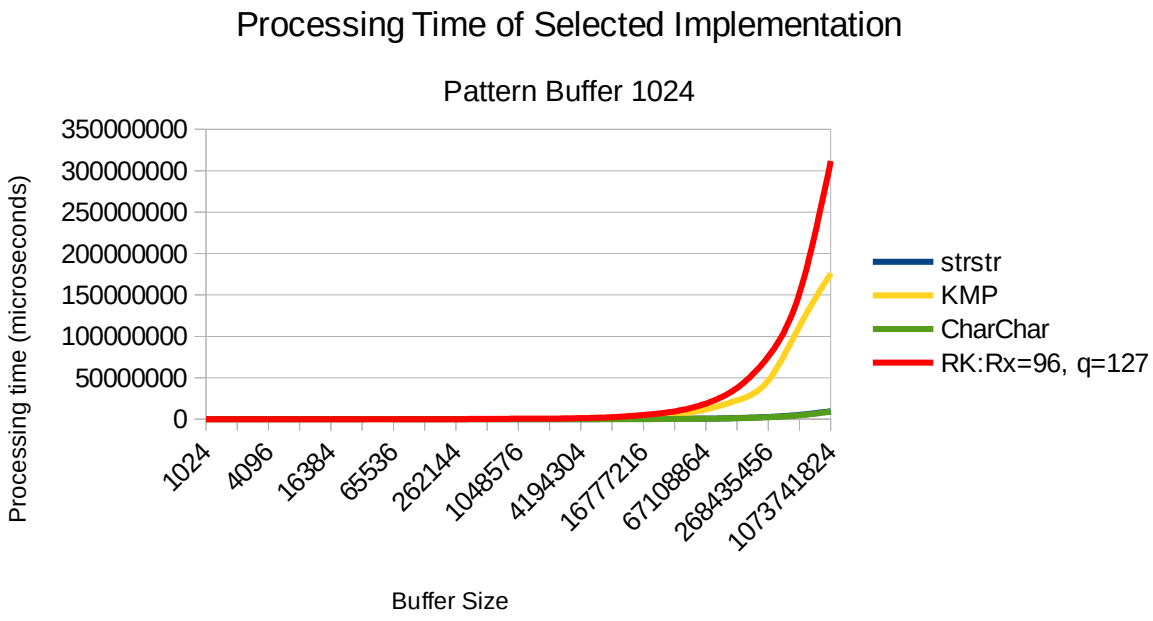


Illustration 1: Processing Times

Each method will be briefly reviewed, with the exception of Rabin-Karp which will have additional analysis.

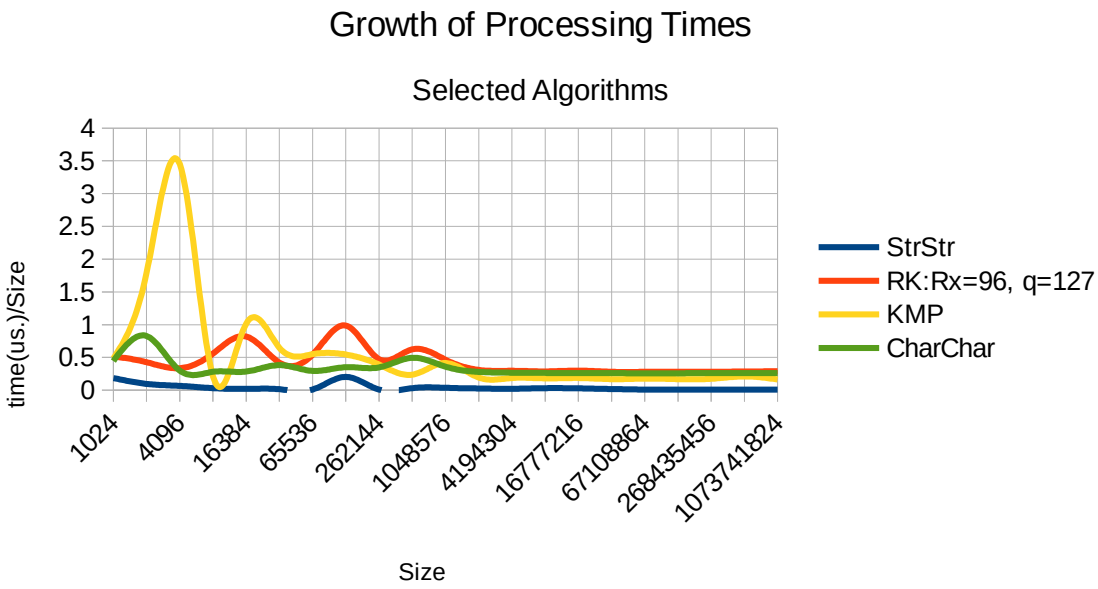
When comparing the algorithms, it was necessary to estimate the expected processing times. A review of "Introduction To Algorithms" (2nd Edition) indicates that worst case processing times may be of the following orders:

Algorithm	Preprocessing	Matching Time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\theta(m)$	$O((n-m+1)m)$
Knuth-Morris-Pratt (KMP)	$\theta(m)$	$\theta(n)$

Since the largest buffer was 1GB with a pattern string of 1KB. Matching times on the order of 10^{12} cycles were expected. Early experiments indicated that a 1KB buffer could be matched with the last character different in 190 us.. Since the match times can be expected to grow linearly a constatnt value could be computed on the upper bound such that: $0 \leqslant f(n) \leqslant cg(n)$ for some value c and function g(n). Since the buffer sizes doubled it is reasonable to assume that processing times would grow similarly. Therefore, times of 1.9×10^{8} us could be expected. Since this was reasonable testing continued.

Since the search buffer was of fixed size the growth was linear and easier to track. The times appeared to grow as shown below:

Illustration 2: Growth of Processing Times



Not surprising there is a great deal of variation until the buffers reach 4MB and then the ratios appear to stabilize. The volatility in the time can be due to the operating system performing additional tasks with the swap memory, however as the requirements grew the task was swapped out less frequently.

Analysis of the data indicated that the growth was less than expected. The calculation of the growth constant limit could be found by examining the data in the following table:

Size	CharChar	StrStr	Rabin-Karp	KMP
1024	0.4462890625	0.185546875	0.5166015625	0.484375
2048	0.8266601563	0.095703125	0.4267578125	1.7866210938
4096	0.2990722656	0.0651855469	0.3347167969	3.4509277344
8192	0.2797851563	0.0305175781	0.5583496094	0.2110595703
16384	0.2839355469	0.0194702148	0.8189086914	1.0159301758
32768	0.382019043	0.0140380859	0.4200134277	0.6481018066
65536	0.2944335938	0.0101470947	0.5428771973	0.5486450195
131072	0.3501739502	0.2026138306	0.9860458374	0.5441589355
262144	0.3472480774	0.0087547302	0.4738121033	0.39037323
524288	0.4929790497	0.030790329	0.6230373383	0.2364616394
1048576	0.3588695526	0.0363769531	0.4652252197	0.4141807556
2097152	0.2786402702	0.0259428024	0.3084411621	0.1928792
4194304	0.2662363052	0.0213131905	0.2955212593	0.1853227615
8388608	0.2630431652	0.0307723284	0.2863683701	0.1800221205
16777216	0.2611192465	0.0278816819	0.2976660132	0.1859007478
33554432	0.2601379454	0.017336309	0.2805121541	0.1685526073
67108864	0.2563109994	0.0094361752	0.2799686491	0.1758794188
134217728	0.2575859651	0.0098113492	0.2812812701	0.1685108989
268435456	0.260254547	0.0098353513	0.2818773687	0.173833441
536870912	0.2604966965	0.0095938854	0.283728499	0.2094682548
1073741824	0.2600430697	0.0089212656	0.2904064069	0.1639175406
Average	0.3326349364	0.0414280334	0.4310531785	0.5492915215

Table 1: Processing times vs. Search Buffer Size

It can be seen that the functions average a growth ranging from a low of 0.04 (strstr) to a high of 0.55 (KMP). This is below the expected growth rate of 1.

1 CharChar and StrStr (Naive processing)

The Character to Character and StrStr processing algorithms appear to match character to character. References in C++ literature refer the reader to the C implementation and a subset of the module used in the library is added in the appendix. The method is very basic and searches for a matching character then tries to match the substring until end of string is encountered for either source or target and if the target is non zero a match is declared.

2 Knuth-Morris-Pratt (KMP)

The KMP algorithm uses the preprocessing phase to determine shifts of the pattern string. During mismatches it then shifts to the next occurrence of the first characters of the processing sequence,

bypassing characters that will never match the beginning of the target sequence. The KMP algorithm works best when there is no match and there is a sequence of characters to repeat.

3 Rabin-Karp

The Rabin-Karp algorithm is the most interesting. It acts upon the principle that if two (sub)strings are identical, their hash values are identical. It relies on a rolling hash of the source and target buffer to determine when to look for a match. Key to the algorithm are a chosen radix and multiplier. Numerous online sources appear to suggest that a radix of 33 and a multiplier of 101 generally work well.

Different Radii and multipliers were used and their results plotted in Illustration 3: Rabin Karp processing times, below. What is interesting is that the processing times are affected by choice of radix and multplier. The CLRS text[1] suggests that the optimal radix is the number of characters in the alphabet being processed and the multiplier be a prime number (q) such that 10q fits into one unit of memory. The use of a prime means that the multiplier does not have factors which in the RabinKarp can easily appear since it is a "rolling" hash. The character set was the ASCII characters from ' ' to '}', excluding '@' which used 95 characters. A radix of 96 was used to represent the value when the value of ' ' was subtracted from '}'. The primes 127 and 8191 appeared to have lowest processing times.

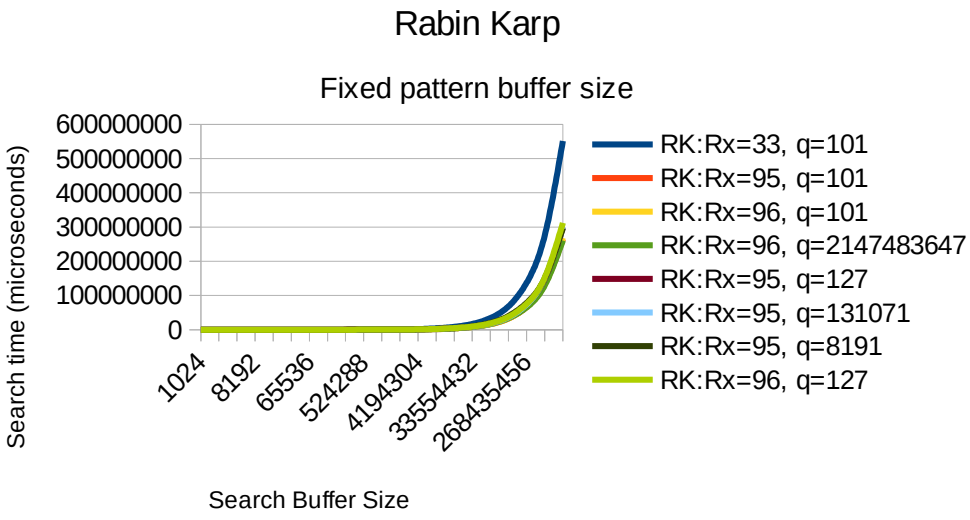
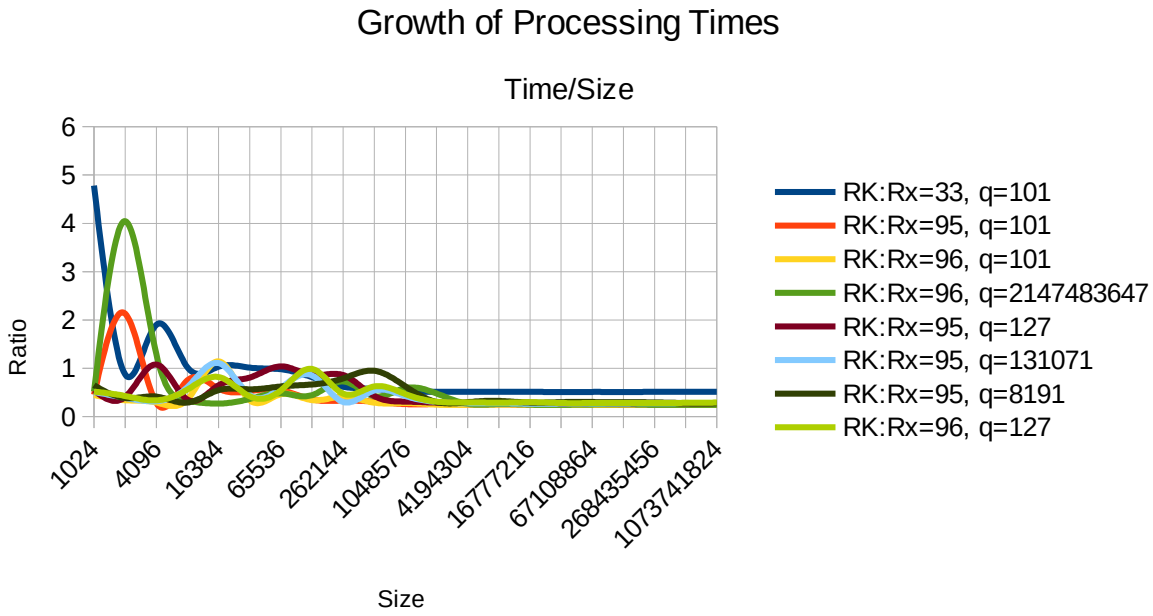


Illustration 3: Rabin Karp processing times

1 "introduction To Algorithms", Thomas H. Cormen, ...et al., MIT Press, 2004

Rabin-Karp experiences similar linear growth after a period of fluctuation.



It appears as if the slowest growth rate is radix=96 and prime of 101 although a prime of 127 is only marginally worse.

4 Conlusions

All methods appear to have reasonable processing times. An optimized character to character comparison is significantly faster than other algorithms. Referring to Table 1: Processing times vs. Search Buffer Size on page 6, we can see that the constant of growth or ‘c’ ranged from 0..04 to 0.44 therefore our estimate of 1.0 was high.

5 Appendix – Code Samples

The following code was created and used for the data analysis. The strstr.c was taken from the gnu repository on GitHub and adapted for the test by deleting certain C preprocessor directives.

1 StringMatch.cpp

```
#include <iostream>
#include <iomanip>
#include <ctime>
#include <cstring>
#include <unistd.h>
```

```
#include <sys/time.h>
#include "StringMatch.h"


using namespace std;


/*
 *        name: find_match - search for substring pattern in source
 *                           uncomment to switch methods
 */
char *find_match(const char *source, const char *pattern)
{
        char *dummy;
        char *s = (char *)source;
        char *p = (char *)pattern;
        dummy = strstr(s, p);                                   // standard
library (uses Rabin Karp)
        //dummy  = strstrByChar(s, p);                          // character by
character match
        //dummy  = strstrRK(s, p);                              // pattern matching
Rabin Karp
        //dummy = strstrKMP(s, p);                              // pattern matching
Knuth-Morris-Pratt
        return dummy;
}

int main()
{
        char *source;
        char *pattern;
        char *dummy;
        struct timeval starttime, endtime;
        long  sometime = 0;
        for (long i = 1024; i <= 1024*1024*1024; i *=2)
        {
           source  = (char *)malloc(i);
           pattern = (char *)malloc(1024);
           char *sfill = source;
           char *pfill = pattern;

                gettimeofday(&starttime, NULL);
           while (sfill < source+i-1)                           // for worst case: make source
& patterns
           {                                                                    //
identical except for last character
                *sfill = (char)(rand()%('}' - ' ') + ' ');
                if (*sfill == '@') *sfill = ' ';
                if (pfill < pattern+1024) *pfill++ = *sfill;
                sfill++;
           }
                *(pattern+1024-2) = '@';
                *(pattern+1024-1) = '\0';                       // null terminate buffers
                *(source+i-1) = '\0';

                gettimeofday(&endtime, NULL);
```

```
                long filltime = (endtime.tv_sec - starttime.tv_sec)*1e6 +
                                        (endtime.tv_usec - starttime.tv_usec);
            sometime = 0;
            for (int j = 1; j <= 16; j++)
            {
                        gettimeofday(&starttime, NULL);

                        dummy = find_match(source,pattern);
                        if (dummy != NULL) cout << "match" << endl;
                        gettimeofday(&endtime, NULL);
                        sometime += (long)((endtime.tv_sec - starttime.tv_sec)*1e6 +
                                (endtime.tv_usec -starttime.tv_usec));
                }
                free(pattern);
                free(source);
                cout << "size: " << i
                    << " Elapsed time: " << sometime
                    << " us. fill: " << setprecision(12) << filltime << " us." << endl;
        }
        cout << "Hello world" << endl;
        return 0;

}
```

2 StrStr.c

```
#include "strstr.h"
using namespace std;

char * strstrByChar (const char *source, const char *pattern)
{
        char *sptr = (char *)source;
        char *substring1, *substring2;

        while (*sptr)
        {
                  substring2 = (char *)pattern;
                  while (*sptr && (*sptr != *substring2)) sptr++;
                if (*sptr)                        // if not at end
                {
                  substring1 = sptr;
                  while ((*substring1 == *substring2) && *substring1 &&
*substring2)
                        substring1++, substring2++;

                  if (!*substring2)
                        return(sptr);

                  sptr++;
                }
        }

        return 0;
```

```
}
```

3 StrStrRK.cpp

```
/*
 * strstrRK.cpp
 *
 *  Created on: Apr 19, 2017
 *      Author: chris
 */
#include <iostream>
#include <iomanip>
#include <cstring>
#include <unistd.h>
#include <math.h>
using namespace std;
/*
 *  name: strstrRK - search for substring pattern in source using
 *                   rabin karp algorithm, hash pattern.
 *  input: char * source, char *pattern
 *  return: pointer to match in source
 */
 char *strstrRK(const char *source, const char *pattern)
 {
          int m, n;
          char *s;
          char *p;
//        const int radix = 33;
          const int radix = '}' - ' '+1;
//        const int prime = 101;
```

```
const int prime = 127;
int hash = 1;
int sourceHash = 0;
int patternHash = 0;

s = (char *)source;
p = (char *)pattern;
n = strlen(s);
m = strlen(p);

for (int i = 0; i < m-1; i++) hash = (hash * radix) % prime;

for (int i = 0; i < m; i++)                // preprocessing
{
        sourceHash = (radix*sourceHash + s[i]) % prime;
        patternHash = (radix*patternHash + p[i]) % prime;
}

for (int i = 0; i < n-m; i++)   // matching
{
        if (sourceHash == patternHash)
        {
                bool match = true;
                for (int j=0; (j < m) && match; j++)
                {
                        match = match && (s[i+j] == p[j]);
                }
                if (match) return s+i;
        }
        sourceHash = (radix*(sourceHash-s[i]*hash)+s[i+m+1]) % prime;
}
```

```
            //cout << m << ":" << n << endl;
            return NULL;

    }
```

4 StrStrKMP

```
/*
 * strstrKMP.cpp
 *
 *  Created on: Apr 19, 2017
 *      Author: chris
 */

#include "strstrKMP.h"
using namespace std;
void preKMP(const char *pattern, int f[])
{
    int m = strlen(pattern);
    f[0] = -1;
    for (int i = 1; i < m; i++)
    {
        int k = f[i - 1];
        while (k >= 0)
        {
            if (pattern[k] == pattern[i - 1])
                break;
            else
                k = f[k];
        }
```

```
        f[i] = k + 1;
    }
}



//check whether target string contains pattern

char *strstrKMP(const char *source, const char *pattern)

{
          char *s = (char*)source;
    int m = strlen(pattern);

    int n = strlen(source);

    int f[m];

    preKMP(pattern, f);

    int i = 0;

    int k = 0;

    while (i < n)
    {
        if (k == -1)
        {
            i++;
            k = 0;
        }
```

```
            else if (source[i] == pattern[k])
            {
               i++;
               k++;
               if (k == m)
                  return s+k;
            }
            else
               k = f[k];
         }
         return 0;
      }
```

5 Addenum

Data processing was performed with LibreOffice Calc.

Data files and source files will be emailed upon request.

A GitHub repository will be created containing the project data over the summer in preparation for my Internship.