- Course Name :YAPISAL PROG. GİRİŞ
- Course Group :GRUP 1
- Instructor Name :AHMET ELBİR
- Algorithm Name :KMP Pattern Matching
- Delivery Date of the Project:18.06.2021
- Student Id :20011023
- Student Name and Surname :MEHMET ALPEREN ÖLÇER

- Video Link :https://youtu.be/zeoPfvNahEE

Knuth Morris Pratt Pattern Matching Algorithm

Algoritma verilen bir metin parçasını istenilen başka bir metinde arar, düzenlenmesine bağlı olarak bulunursa yerini veya göre kaç kere bulunduğunu bize söyleyebilir. Substring araması gereken yerlerde kullanılabilir. Bunu yaparken kendini rakiplerinden öne çıkaran bazı yönleri vardır.
Algoritma aranan kelimenin, aranan metinde bulunmaması durumunda, kelimenin içerisindeki harflerden yola çıkarak birden fazla ihtimali elemektedir. Klasik bir metin arama işleminde aranan kelime metindeki bütün ihtimallerde denenir. (örneğin doğrusal arama (linear search) bu şekilde çalışır). KMP algoritmasında ise aranan metindeki bütün ihtimaller denenmez.
Doğrusal aramada M uzunluğundaki metinde (AAAAAAAB) N uzunluğunda bir kelime (AAB) arandığında kelimenin eşleşmeyen bir harfine denk gelindiğinde metinde geri gidilir ve kelimenin başına dönülür. Fakat KMP algoritmasında kelimenin kendi içindeki benzerlikler aramaya başlanmadan belirlenir ve kelimenin eşleşmeyen bir harfine denk gelindiğinde metinde geri gidilmez, kelimede en sonki benzerliğe geri gidilir.
Örneğin:

```
AAAAAAAB
A          +
 A         +
  B         -
  A         +
    B       -
    A       +
     B      +
       .....
          B +      Bulundu
```

Bu örnekten de anlaşılacağı gibi KMP algoritmasında kelime içindeki tekrarlardan yararlanılarak arama tekrarından kaçınılır.
Doğrusal aramada en kötü senaryoda yaklaşık M*N (metin uzunluğu * kelime uzunluğu) kadarlık bir karmaşıklık vardır. KMP algoritmasında ise yaklaşık M+N 'lik karmaşıklık vardır.
Gerçek dünyada KMP algoritması, sembolleri küçük kardinaliteye sahip bir alfabeden alınan uzun dizilerde desen eşleştirmenin yapıldığı uygulamalarda kullanılır. İlgili bir örnek, sadece 4 sembolden (A,C,G,T) oluşan DNA alfabesidir. KMP'nin bir "DNA örüntü eşleştirme probleminde" nasıl çalışabileceğini hayal edin: aynı harfin birçok tekrarı birçok sıçramaya izin verdiği ve dolayısıyla daha az hesaplama zamanı israfına izin verdiği için gerçekten uygundur. KMP' nin avantajı karmaşıklığının az olmasıdır ama bir dezavantajı da vardır. Dezavantajı ise ek olarak kelime uzunluğu kadar bir sayı dizisi oluşturur.
Rakipleri :

In the following compilation, *m* is the length of the pattern, *n* the length of the searchable text, *k* = |Σ| is the size of the alphabet, and *f* is a con

Algorithm	Preprocessing time	Matching time[1]	Space
Naïve string-search algorithm	none	Θ(mn)	none
Optimized Naïve string-search algorithm (libc++ and libstdc++ string::find)[3][4]	none	Θ(mn/f)	none
Rabin–Karp algorithm	Θ(m)	average Θ(n + m), worst Θ((n−m)m)	O(1)
Knuth–Morris–Pratt algorithm	Θ(m)	Θ(n)	Θ(m)
Boyer–Moore string-search algorithm	Θ(m + k)	best Ω(n/m), worst O(mn)	Θ(k)
Bitap algorithm (*shift-or*, *shift-and*, *Baeza–Yates–Gonnet*; fuzzy; agrep)	Θ(m + k)	O(mn)	
Two-way string-matching algorithm (glibc memmem/strstr)[5]	Θ(m)	O(n+m)	O(1)
BNDM (Backward Non-Deterministic DAWG Matching) (fuzzy + regex; nrgrep)[6]	O(m)	O(n)	
BOM (Backward Oracle Matching)[7]	O(m)	O(mn)	
FM-index	O(n)	O(m)	O(n)

```
TEXT length : 10^2     PATTERN length : 10^1      RUNTIME (seconds) : 0.000032    COMPLEXITY :     112       HISTOGRAM(digits) : ###
TEXT length : 10^3     PATTERN length : 10^1      RUNTIME (seconds) : 0.000022    COMPLEXITY :    1022       HISTOGRAM(digits) : ####
TEXT length : 10^3     PATTERN length : 10^2      RUNTIME (seconds) : 0.000021    COMPLEXITY :    1221       HISTOGRAM(digits) : ####
TEXT length : 10^4     PATTERN length : 10^1      RUNTIME (seconds) : 0.000144    COMPLEXITY :   10612       HISTOGRAM(digits) : #####
TEXT length : 10^4     PATTERN length : 10^2      RUNTIME (seconds) : 0.000149    COMPLEXITY :   10998       HISTOGRAM(digits) : #####
TEXT length : 10^4     PATTERN length : 10^3      RUNTIME (seconds) : 0.000167    COMPLEXITY :   11374       HISTOGRAM(digits) : #####
TEXT length : 10^5     PATTERN length : 10^1      RUNTIME (seconds) : 0.001435    COMPLEXITY :  106728       HISTOGRAM(digits) : ######
TEXT length : 10^5     PATTERN length : 10^2      RUNTIME (seconds) : 0.001304    COMPLEXITY :  100165       HISTOGRAM(digits) : ######
TEXT length : 10^5     PATTERN length : 10^3      RUNTIME (seconds) : 0.001416    COMPLEXITY :  101513       HISTOGRAM(digits) : ######
TEXT length : 10^5     PATTERN length : 10^4      RUNTIME (seconds) : 0.001647    COMPLEXITY :  122765       HISTOGRAM(digits) : ######
TEXT length : 10^6     PATTERN length : 10^1      RUNTIME (seconds) : 0.013190    COMPLEXITY : 1051362       HISTOGRAM(digits) : #######
TEXT length : 10^6     PATTERN length : 10^2      RUNTIME (seconds) : 0.013434    COMPLEXITY : 1066299       HISTOGRAM(digits) : #######
TEXT length : 10^6     PATTERN length : 10^3      RUNTIME (seconds) : 0.011942    COMPLEXITY : 1008834       HISTOGRAM(digits) : #######
TEXT length : 10^6     PATTERN length : 10^4      RUNTIME (seconds) : 0.013012    COMPLEXITY : 1078113       HISTOGRAM(digits) : #######
TEXT length : 10^6     PATTERN length : 10^5      RUNTIME (seconds) : 0.014623    COMPLEXITY : 1315313       HISTOGRAM(digits) : #######
```

KOD

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 15

void driver(int text_len, double time[SIZE], double complexity_array[SIZE], int index);
double KMPSearch(char* pattern, char* text, int M, int N);
double values_array(char* pattern, int M, int* values);
char* random_string_create(int len);
int get_base(int base, int ex);
void output(double time[SIZE], double complexity_array[SIZE]);
void sharp_digits(double number);

int main()
{
   srand(time(NULL));
   double time[SIZE], complexity_array[SIZE];  // algoritmanin zaman ve karmasiklik analizlerini
saklamak icin
   driver(2, time, complexity_array, 0);
   output(time, complexity_array); // algoritmanin zaman ve karmasiklik analizlerini yaziyor
   return 0;
}

// algoritmayi yoneten fonksiyon
void driver(int text_len, double time[SIZE], double complexity_array[SIZE], int index)
{
   /*
   random olusturulan icinde arama yapilacak stringler 10^2 10^3 10^4 10^5 10^6 uzunluklarinda
   random olusturulan aranacak stringler 10^1 10^2 10^3 10^4 10^5 uzunluklarinda
   substring aramayi textlerden kisa butun patternler icin yapacagiz
   */
   int pattern_len;
   if (text_len == 7)      // rekursifligin durma kosulu
      return;
   char* text;
   char* pattern;
   text = random_string_create(get_base(10, text_len));        // text olusturma
   //printf("%s\n\n", text);
```

```
   for ( pattern_len = 1; pattern_len < text_len; pattern_len++)   // text'den kisa butun patternleri okumak icin dongu
   {
      pattern = random_string_create(get_base(10, pattern_len));     // pattern olusturma
      clock_t tic = clock();      // zaman sayaci baslatma
      complexity_array[index] = KMPSearch(pattern, text, get_base(10, pattern_len), get_base(10, text_len));     // stringleri ve uzunluklarini algoritmaya gonderme ve complexity donutunu alma
      clock_t toc = clock();      // zaman sayaci bitirme
      time[index++] = (double)(toc - tic) / CLOCKS_PER_SEC;   // zamani kaydetme
   }
   driver(text_len+1, time, complexity_array, index);   // bir sonraki test icin fonksiyonu tekrarlama
}

// algoritma
double KMPSearch(char* pattern, char* text, int M, int N)
{
   // M -> pattern uzunlugu        N -> text uzunlugu
   double complexity;
   int i = 0; // text'in indexi
   int j = 0; // pattern'in indexi
   int counter = 0;    // kac kere tekrar ettigini bulmak icin
   int* values;
   values = malloc(M * sizeof(int));   // bu dizi patterndeki prefix suffix durumlarina gore deger tutacak
   complexity = values_array(pattern, M, values);  // values dizisini doldurma, complexity'i de donduruyor
   printf("pattern uzunlugu -> %d   text uzunlugu --> %d",M, N);
   printf("\n");
   while (i < N)   // text'de ilerleme
   {
      complexity ++;
      if (pattern[j] == text[i])  // ayniysa i ve j artiyor
      {
         j++;
         i++;
      }

      if (j == M)     // patternin son elemanina kadar ayni gittiyse substring bulmus oluyoruz
      {
         //printf("Found pattern at index %d \n", i - j);
         counter ++;
         j = values[j - 1];      // mesela text'imiz ABABABAB ve pattern'imiz ABAB ise prefix ve suffix ayniligindan dolayi j'yi values'daki indexe esitleyip ayniligin oldugu yere kadar tekrar kontrol etmiyoruz
      }

      // j ve i eslestikten sonra birer artmis haliyle eslememe durumu
      else if (pattern[j] != text[i] && i < N)
      {
         // j yi bir onceki prefix'in baslangicina aktarma
         if (j != 0)
            j = values[j - 1];
```

```
            else
                i = i + 1;
        }
    }
    printf("Found %d times.\n", counter);
    return complexity;
}

// values'u verilen pattern gore dolduruyor
// pattern'in icindeki tekrarlari saklamak icin yapiyoruz.
double values_array(char* pattern, int M, int* values)
{
    // en sonki en uzun prefix ve suffix'in uzunlugu
    int len = 0;
    double complexity=0;
    values[0] = 0;  // 0 olmak zorunda

    // patterndeki tekrarlarin yerlerini saklamak icin gerekli bilgilerin values dizisine doldurulduguu
dongu
    int i = 1;
    while (i < M)
    {
        complexity ++;
        if (pattern[i] == pattern[len]) // substring in kaldigimiz yerindeki yeni harfi stringde kaldigimiz
yerdeki harfle ayni mi
        {
            len++;
            values[i] = len;        // KMP search'te mesela patternin sonuna kadar geldigimizde ve
eslesmediginde, pattern'in basina donmek yerine eslesmeyen harfin values'daki index degerine geri
donucez pattern'de
            i++;
        }
        else
        {
            if (len != 0)
            {
                len = values[len – 1];         // eslesme olmadi ama len'nin icindeki tekrara gore len'i
guncelliyoruz
                // i'yi arttirmiyoruz
            }
            else // if (len == 0)
            {
                values[i] = 0;      // herhangi bir benzerlik durumu kalmadiginda
                i++;                // pattern bitene kadar devam
            }
        }
    }
    return complexity;
}

// ust alma fonksiyonu
int get_base(int base, int ex)
```

```
{
   if (ex == 0)
      return 1;
   else if (ex % 2)
      return base * get_base(base, ex - 1);
   else
   {
      int temp = get_base(base, ex / 2);
      return temp * temp;
   }
}

// Algoritmamizi uzun stringler ile siniyoruz ve bunlari random olusturuyoruz.
// fonksiyona gonderilen parametreye gore farkli uzunlunluklarda dinamik string olusturup geri donduruyor.
char* random_string_create(int len)
{
   char* string;
   int i, temp;
   string = malloc(len*sizeof(char)); // yer acma
   for ( i = 0; i < len; i++)
   {
      /*
      temp = rand()%52;
      if (rand()%2==0)    // buyuk kucuk harf icin
         string[i] = 'a' + temp/2;
      else
         string[i] = 'A' + temp/2;
      */
      string[i] = 'A' + rand()%2;
      // bulmasi icin 2 yaptim
      // yoksa nadiren buluyor
   }
   return string;
}

// tablo yazdirma
void output(double time[SIZE], double complexity_array[SIZE])
{
   int i, j, index = 0;
   for ( i = 2; i < 7; i++)
   {
      for ( j = 1; j < i; j++)
      {
         printf("TEXT length : 10^%d     PATTERN length : 10^%d      RUNTIME (seconds) : %f COMPLEXITY : %7d       HISTOGRAM(digits) : ", i, j, time[index], (int)complexity_array[index]);
         sharp_digits(complexity_array[index]);  // complexity'nin basamak sayisi ile histogram olusturuyoruz.
         printf("\n");
         index ++;
      }
```

```
    }
}

// verilen sayinin basamak sayisi kadar '#' yazdiran fonksiyon
void sharp_digits(double number)
{
    int digits = 0;
    while (number > 1)
    {
        number /= 10;
        digits ++;
    }
    for (; digits > 0; digits--)
        printf("#");
}
```

Raporun Hazırlanmasında Yararlanılan Kaynaklar

https://en.wikipedia.org/wiki/String-searching_algorithm

http://bilgisayarkavramlari.com/2009/04/11/knuth-morris-prat-algoritmasi-kmp-algorithm/