

Tartu Ülikool  
Matemaatika-informaatikateaduskond  
Arvutiteaduse instituut

# **Knuth-Morris-Pratti ja Boyer-Moore'i alamsõnede otsimise algoritmide võrdlus**

Autor: Martti Savolainen  
Juhendaja: Ain Isotamm  
Kursus: Infotehnoloogia II (FK)

Tartu 2006

## **Sisukord**

Sissejuhatus .....	3
Knuth-Morris-Pratti algoritm .....	4
Boyer-Moore'i algoritm .....	6
Knuth-Morris-Pratti ja Boyer-Moore'i algoritmide võrdlus .....	9
Baastöö analüüs .....	10
Kokkuvõte .....	11
Lisad .....	12
Kasutatud kirjandus .....	14

## **Sissejuhatus**

Tekstiga seotud rakendustarkvara kasutab pea alati mingisugust alamsõnede leidmise algoritmi. Lihtsaim viis oleks läbida tekst läbi sümbol-haaval ja proovida otsisõnet igale positsioonile, kuid pikemate otsisõnede korral poleks see eriti efektiivne.

See töö kirjeldab kahe alamsõnede leidmise algoritmi (Knuth-Morris-Pratti ja Boyer-Moore'i) tööpõhimõtteid ja võrdleb nende efektiivsust erinevatel juhtudel. Lisaks sellele esitatakse ka näited praktilisest kasutusest mõlema algoritmi kohta.

## Knuth-Morris-Pratti algoritm

Algoritm leiutati 1977. aastal **Donald Ervin Knuthi** ja **Vaughan Ronald Pratti** ning iseseisvalt **J. H. Morrise** poolt, kuid nad avaldasid algoritmi koos.

Knuth-Morris-Pratti (edaspidi KMP) algoritm kasutab otsimise kiirendamiseks abitabelit, mida nimetatakse osalise kattuvuse tabeliks. Selle tabeli koostamiseks analüüsitakse otsisõne ning iga sümboli jaoks leitakse mitme sümboli võrra võib paremale liikuda, kui antud kohal otsisõne sümbol ja teksti osa vastav sümbol, kust parajasti seda sõne otsitakse, ei ole samad. Toimuva mõistmise hõlbustamiseks on alljärgnev pseudokood.

```
algoritm KMP_tabel:
  sisend:
    sümbolite massiiv W (otsisõne)
    täisarvude massiiv T (abitabel)
  väljund:
    ei tagastata midagi, kuid täidetakse tabel T

  (Seame mõned esmased väärtused)
  i := 2 (hetke positsioon, mida me uurime)
  j := 0 (järgmine sümbol, mida me uurime)
  T[0] := -1
  T[1] := 0

  while i < W pikkus do:
    (esimene juht: alamsõne jätkub)
    if W[i - 1] = W[j]:
      T[i] := j + 1
      i := i + 1
      j := j + 1

    (teine juht: alamsõne ei jätku)
    (võtame uue sümboli mida uurida)
    if j > 0:
      j := T[j]

    (kolmas juht: tegemist on sümboli esmakordse esinemisega)
    else:
      T[i] := 0
      i := i + 1
```

Ülaltoodud tabeli koostamise algoritm on keerukusega  $O(w)$ , kus  $w$  on otsisõne  $W$  pikkus. Järgnev pseudokood kirjeldab kuidas toimub otsisõne tekstist leidmine, see osa KMP algoritmist on keerukusega  $O(s)$ , kus  $s$  on teksti  $S$  pikkus.

```

algoritm KMP_otsing:
  sisend:
    sümbolite massiiv W (otsisõne)
    sümbolite massiiv S (tekst millest antud sõne W
    otsitakse)
  väljund:
    ilmtingimata pole tarvidust midagi tagastada, kuid võib
    tagastada otsisõne esinemise asukoha või otsisõnede
    esinemiste arvu tekstis vmt

  (Seame mõned esmased väärtused ja täidame abitabeli T)
  m := 0 (hetke positsioon tekstis S)
  i := 0 (hetke positsioon otsisõnes W)
  KMP_tabel ( W, T )

  while m + i < S pikkus do:
    if W[i] = S[m + 1]:
      i := i + 1
      if i = W pikkus:
        (otsisõne W asukoht tekstis S leitud)
        Siinkohal võib leitud vaste positsiooni
        tagastada:
          return m
        Või suurendada loendurit ja/või märkida üles
        leitud positsioon ning otsingut jätkata.
        Viimasel juhul tuleb liikuda järgmisele
        positsioonile tekstis S ja seada otsisõne
        uurimise positsioon sõne algusesse.
          i := 0
          m := m + 1
    else:
      m := m + i + T[i]
      if i > 0:
        i := T[i]

```

Seega Knuth-Morris-Pratti alamsõnede leidmise algoritm on konstantse keerukusega  $O(w + s)$ , kus  $w$  on otsisõne pikkus ja  $s$  on teksti pikkus, kust antud sõne otsitakse.

## Boyer-Moore'i algoritm

Idee Boyer-Moore'i (edaspidi BM) algoritmi taga on sarnane KMP algoritmiga. Selleks, et teada saada kas otsisõne  $W$  sobib mingisse kohta tekstis  $S$  sobitame  $W$ -d alates  $W$  viimasest positsioonist, st. paremalt vasakule, mitte vasakult paremale nagu seda tegi KMP algoritm. Kui  $S$ -is vastaval positsioonil olev märk ei ole sama, mis  $W$  viimane märk, siis saab liigutada otsimispositsiooni tekstis edasi. Kui seda märki ei eksisteeri sõnes  $W$ , siis saame liikuda tekstis edasi terve  $W$  pikkuse, kui aga märk esineb mingil positsioonil sõnes  $W$ , siis liigume edasi täpselt nii palju, et vastavad märgid oleksid kohakuti. Järgnev skeem illustreerib, kuidas sõne leidmine tekstist toimub.

Otsisõne: **EXAMPLE**  
Tekst: **HERE IS A SIMPLE EXAMPLE**

Me ei leidnud vastet, „E” ja „S” ei ole võrdsed. Kuna „S” ei esine sõnas „EXAMPLE”, siis võime edasi liikuda terve otsisõne pikkuse võrra.

Otsisõne: **EXAMPLE**  
Tekst: **HERE IS A SIMPLE EXAMPLE**

„P” ei ole võrdne „E”-ga, kuid „P” eksisteerib sõnas „EXAMPLE”. Seega liigutame otsisõne edasi nii, et „P”-d oleksid kohakuti.

Otsisõne: **EXAMPLE**  
Tekst: **HERE IS A SIMPLE EXAMPLE**

Hakates otsisõnes märke võrdlema paremalt vasakule, näeme, et kuni „A”-ni on kõik võrdlused tõesed. Kuna „E” eksisteerib sõnas „EXAMPLE” rohkem kui üks kord, kuid märgid „M”, „P” ja „L” ainult ühe korra, siis otsimispositsiooniks võtta „E” asukoha.

Otsisõne: **MPEEXAMPLE**  
Tekst: **HERE IS A SIMPLE EXAMPLE**

Võrdleme märke „E” ja „P”. Kuna „P” eksisteerib sõnas „EXAMPLE”, siis liigutame otsimispositsiooni edasi nii, et „P”-d oleksid kohakuti.

Otsisõne: **EXAMPLE**  
Tekst: **HERE IS A SIMPLE EXAMPLE**

Leidsime otsisõne esinemise tekstist.

Et märkide eksisteerimist sõnes  $W$  iga iteratsioon kontrollima ei peaks, loome enne otsima asumist kaks abitabelit.

Esimese tabeli arvutamine on kerge. Seame loenduri  $c$  väärtuseks 0 ja alustame otsisõne viimasest märgist ning liigume ettepoole. Iga kord kui vasakule liigume, suurendame loenduri väärtust 1 võrra. Kui vastaval positsioonil olev märk otsisõnest pole veel salvestatud, siis salvestame ta koos loenduri hetke-väärtusega (selleks võime kasutada *hash* tabelit või massiivi mille võtmeteks on otsisõne märgid).

```

algoritm BM_korduvuste_tabel:
  sisend:
    otsisõne W
  väljund:
    tabel T

  c := 0 (Seame loenduri algväärtuseks 0)

  for i := W pikkus - 1 to 0 do:
    if ei eksisteeri T[ W[i] ]:
      T[ W[i] ] := c
      c := c + 1

  return T

```

Teine tabel kajastab võimalikke alamsõnede kordumisi otsisõne sees, mis võimaldab meil teatud portsjone tekstist vahele jätta. Kõige ilmekamalt illustreerib seda protsessi pseudokood.

```

algoritm BM_vahelejätmise_tabel:
  sisend:
    otsisõne W
  väljund:
    tabel T

  i := W pikkus
  j := W pikkus + 1
  f[i] := j

  while i > 0 do:
    while j ≤ m and W[i - 1] != W[j - 1] do:
      if T[j] = 0:
        T[j] := j - i
        j := f[j]

      i := i - 1
      j := j - 1
      f[i] := j

  j := f[0]
  i := 0
  while i ≤ m do:
    if T[i] = 0:
      T[i] := j
    if i = j:
      j=f[j]

  return T

```

BM algoritmi võiks implementeerida lähtuvalt järgmisele pseudokoodile.

**algoritm** BM\_otsing:

**sisend:**

sümbolite massiiv W (otsisõne)  
sümbolite massiiv S (tekst millest antud sõne W  
otsitakse)

**väljund:**

ilmtingimata pole tarvidust midagi tagastada, kuid võib  
tagastada otsisõne esinemise asukoha või otsisõnede  
esinemiste arvu tekstis vmt

m := 0

T1 := **BM\_korduvuste\_tabel** ( W )

T2 := **BM\_vahelejätmistete\_tabel** ( W )

**while** m < ( S pikkus - W pikkus ) **do:**

  j := W pikkus - 1

**while** W[j] = S[j + m] **do:**

**if** j = 0:

      (otsisõne W asukoht tekstis S leitud)

      Siinkohal võib leitud vaste positsiooni

      tagastada:

**return** m

      Või suurendada loendurit ja/või märkida üles

      leitud positsioon ning otsingut jätkata.

      Viimasel juhul tuleb liikuda järgmisele

      positsioonile tekstis S ja seada otsisõne

      uurimise positsioon sõne lõppu.

      j := W pikkus

      m := m + 1

  j := j - 1

  m := m + **max**( T2[j], j - T1[ S[j + m] ] )

Kui  $w$  on otsisõne  $W$  pikkus ja  $s$  on teksti  $S$  pikkus, siis BM algoritmi abitabelite arvutamine on konstantse keerukusega  $O(w)$  ja otsimisalgoritm on keerukusega üldjuhul keerukusega  $O(s/w)$  ja halvimal juhul keerukusega  $O(s)$ .

## Knuth-Morris-Pratti ja Boyer-Moore'i algoritmide võrdlus

### Knuth-Morris-Pratti algoritm

- + Konstantne keerukus
- + Lihtsam implementeerida kui Boyer-Moore'i algoritm
- Pikkade otsisõnede puhul on antud algoritm aeglane võrreldes Boyer-Moore'i algoritmiga

### Boyer-Moore'i algoritm

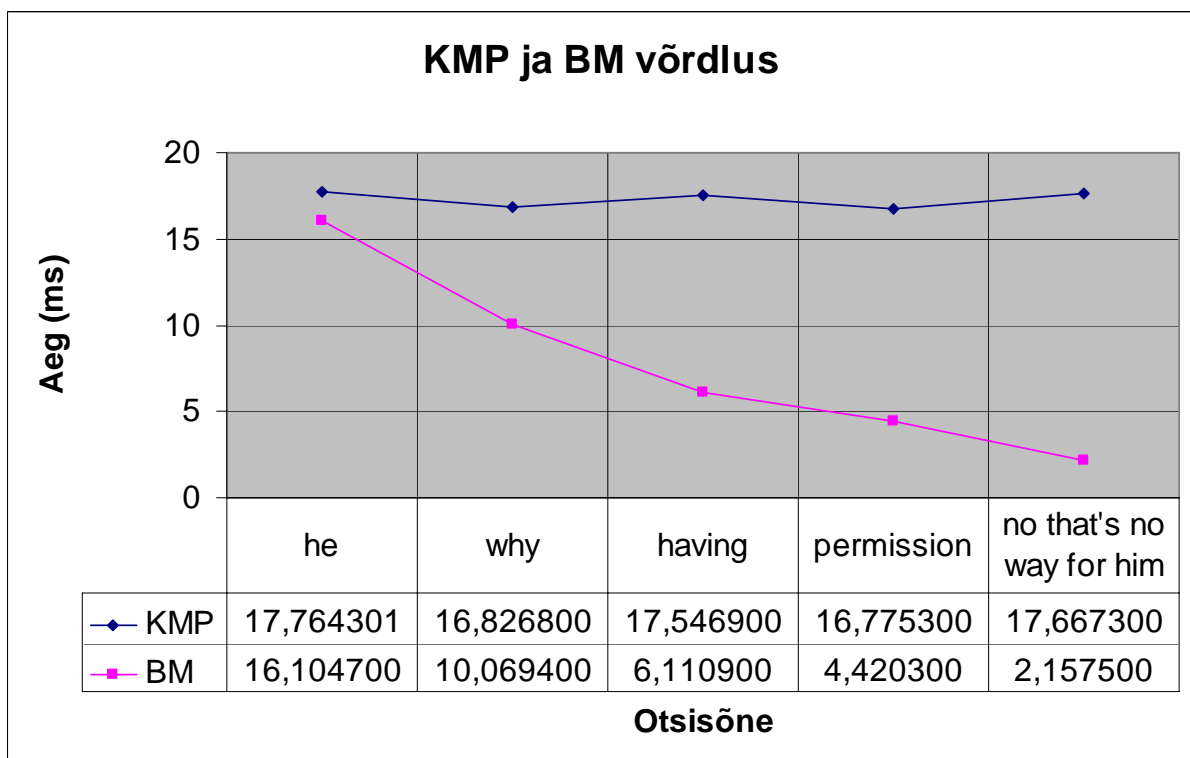
- + Pikemate otsisõnede puhul kiirem kui Knuth-Morris-Pratti algoritm.
- Suurte alfabeetide juures on initsialiseerimine ressursimahukas
- Keerulisem implementeerida kui Knuth-Morris-Pratti algoritm

Ülalnimetatud algoritmide võrdlemiseks tehti mõõtmised, kus otsiti James Joyce'i romaanist „Ulysses” järgmisi sõnu ja fraase: „he”, „why”, „having”, „permission” ja „no that's no way for him has he no manners nor no refinement nor no”. Viimast fraasi leidub raamatus vaid 1 kord. Iga sõna ja fraasiga tehti 10 000 mõõtmist. Mõõtmised teostati arvutil protsessori taktsagedusega 2100MHz (AMD Athlon 64).

Saadud mõõtmiste tulemused (ajad on antud millisekundites)

	he	why	having	permission	no that's no way ...
KMP	17,764301	16,826800	17,546900	16,775300	17,667300
BM	16,104700	10,069400	6,110900	4,420300	2,157500

Sama tabel graafikuna



## ***Baastöö analüüs***

Valitud teema baastöö autoriks on Tauno Palts[6]. Ülesehituselt oli töö loogiline ja hästi liigendatud. Siiski andis töö teemast ainult kesise ülevaate ning suurem osa informatsioonist tuli ise juurde otsida. Samuti ei olnud väga selgelt kirjeldatud algoritmide tööpõhimõtteid ja esitatus pseudokood läks vastuollu juuresoleva tekstiga. Silma hakkas ka korduvalt esinev kirjaviga V. R. Pratti nimes.

Algoritmide tööaja mõõtmise algandmeteks kasutasime sama raamatut mida baastöö autorgi. Testide tulemused olid esitatud korrektselt ja võib näha sarnasusi käesolevas töös saadud tulemustega.

## **Kokkuvõte**

Mõlemad algoritmid on sobilikud kasutamiseks tekstiga ja/või selle töötlemisega seotud rakendustarkvaras. Pikemate sõnede korral osutub efektiivsemaks Boyer-Moore'i algoritm, seega oleks otstarbekas rakendustes neid algoritme kombineerida.

## Lisad

### Knuth-Morris-Pratti algoritmi implementatsioon keeles C

```
/**
 * KMP.h
 *
 * See on Knuth-Morris-Pratti alamsõnede leidmise algoritmi implementatsioon.
 * Autor: Martti Savolainen (FKIT II)
 * 2006-11-21
 */
int KMP_search_count(char *text, char *search, int verbose) {

    /// Abitabeli koostamisega seotud muutujad
    int i=2; /// Hetke positsioon, mida me uurime
    int j=0; /// JÄrgmine sümbol, mida me uurime

    int L = strlen(search); /// Otsisõne pikkus
    int S = strlen(text); /// Teksti pikkus
    int count=0; /// Leitud vastete arv

    int T[L]; /// Abitabel ehk osalise kattuvuse tabel

    /// KMP algoritmiga seotud muutujad
    int m=0; /// Hetkel leitud otsisõne alguskoht tekstis
    int p=0; /// Hetke positsioon otsisõnes

    /// Kõigepealt koostame osalise kattuvuse tabeli
    /// Seame määrad algväärtused
    T[0] = -1;
    T[1] = 0;

    /// Täidame tabeli
    while (i < L) {
        if (search[i-1] == search[j]) {
            T[i++] = ++j;
        }
        else if (j > 0)
            j = T[j];
        else
            T[i++] = 0;
    }

    /// Nüüd asume tekstist vasteid otsima
    while ( (m + p) < S ) {
        if (search[p] == text[m + p]) {
            p++;
            if (p == L) {
                /// Otsisõne leitud (võiksime ka tagastada viida leitud asukohale)
                if (verbose) {
                    printf("Leitud positsioonil %d. (\\"..", m);
                    int start = MAX((int)(m-10), (int)0);
                    int end = MIN((int)(m+p+10), (int)S);
                    for (i=start; i<end; i++)
                        if (text[i] != '\r')
                            printf("%c", (text[i] == '\n' ? ' ' : text[i]));
                    printf("...\\"..");
                }
                /// Suurendame tulemuste hulka ning liigume tekstis edasi
                count++;
                p=0;
                m++;
            }
        }
        else {
            m = m + p - T[p];
            if (p > 0)
                p = T[p];
        }
    }

    return count;
}
```

## Boyer-Moore'i algoritmi implementatsioon keeles C

```
/**
 * BM.h
 *
 * See on Boyer-Moore'i alamsõnede leidmise algoritmi implementatsioon.
 * Autor: Martti Savolainen (FKIT II)
 * 2006-11-22
 */

#define OCC_MAX 256 // Maksimaalne esinemiste tabeli pikkus
static int BM_needlematch(const unsigned char* needle, size_t nlen,
                          size_t portion, size_t offset)
{
    ssize_t virtual_begin = nlen - offset - portion;
    ssize_t ignore = 0;
    if(virtual_begin < 0) { ignore = -virtual_begin; virtual_begin = 0; }

    if(virtual_begin > 0 && needle[virtual_begin-1] == needle[nlen-portion-1])
        return 0;

    return memcmp(needle + nlen - portion + ignore,
                  needle + virtual_begin, portion - ignore) == 0;
}

int BM_search_count(char *text, char *search, int verbose) {

    int count = 0; // Leitud vastete arv
    size_t L = strlen(search); // Otsisõne pikkus
    size_t S = strlen(text); // Teksti pikkus
    size_t skip[L]; // Vahele jäetavate osade tabel
    ssize_t occ[OCC_MAX]; // Korduvate tähtede esinemise tabel
    size_t i; // Abimuutujad
    unsigned long pos;
    if(L > S || L <= 0 || !text || !search) return 0;

    // Täidame esinemiste tabeli
    for(i=0; i<OCC_MAX; ++i) occ[i] = -1;
    for(i=0; i<L-1; ++i) occ[(unsigned char)search[i]] = i;

    // Täidame tabeli mis lubab meil osa tekstist vahele jätta
    for(i=0; i<L; ++i) {
        size_t value = 0;
        while(value < L && !BM_needlematch(search, L, i, value))
            ++value;
        skip[L-i-1] = value;
    }
    // Otsime
    for(pos=0; pos <= S-L; ) {
        size_t npos=L-1;
        while(search[npos] == text[npos+pos]) {
            if(npos == 0) {
                // Otsisõne leitud (võiksime ka tagastada viida leitud asukohale)
                if(verbose) {
                    printf("Leitud positsioonil %lu. (\\"...", pos);
                    unsigned char *p;
                    i = 20;
                    if(pos < i/2) p = text;
                    else p = text + pos - i/2;

                    while (*p != '\\0' && i > 0) {
                        printf("%c", (*(++p) == '\\n' && *p != '\\r' ? ' ' : *p));
                        i--;
                    }
                    printf("...\\"\\n");
                }
                count++; npos=L; pos++;
            }
            --npos;
        }
        pos += (unsigned)MAX((int)(skip[npos]), (int)(npos - occ[(unsigned char)text[npos+pos]]));
    }
    return count;
}
```

## ***Kasutatud kirjandus***

- [1] The Knuth-Morris-Pratt algorithm (viimati vaadatud 27.11.2006)  
[http://en.wikipedia.org/wiki/Knuth-Morris-Pratt\\_algorithm](http://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm)
- [2] KPM String Searching Example (viimati vaadatud 27.11.2006)  
<http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/kpm-example.html>
- [3] The Boyer-Moore Fast String Searching Algorithm (viimati vaadatud 27.11.2006)  
<http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/>  
<http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/fstrpos-example.html>
- [4] Boyer-Moore algorithm (viimati vaadatud 27.11.2006)  
<http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm>
- [5] Boyer-Moore string search algorithm (viimati vaadatud 27.11.2006)  
<http://en.wikipedia.org/wiki/Boyer-Moore>
- [6] Tauno Palts, „Knuth-Morris-Pratti ja Boyer-Moore'i alamsõnede otsimise algoritmide võrdlus”, Tartu 2005 (käsikiri)