Stringide otsimine

Stringide töötlemisega on seotud erinvad algoritmid, põhiosa moodustavad mitmesugused alamstringide otsimise algortimid. Lihtsaim näide on tavaline tekstitöötlusprogramm, kus kasutajale pakutakse võimalust teksti seest sõnu otsida. Kiirus on siin oluline.

Stringide otsimise algoritmid jagunevad nelja suuremasse klassi:

- **Lihtne stringi otsimine**, mida saab kasutada näiteks tekstitöötlusprogrammides sõnade ja sõnaosade otsimiseks, vajalik võib olla ka mitme dokumendi läbivaatamine; kui tekst on pikk, muutub otsimise kiirus oluliseks.
- **Stringi otsimine malli** *(pattern)* **järgi**, kasutades näiteks metamärke *, ? jms; ka sellist võimalust pakuvad tihti tekstiredaktorid.
- **Sarnase stringi otsimine** tähendab seda, et leida ei ole vaja täpselt samasugust märkide jada, vaid lubatud on väikesed erinevused.
- **Keelte analüsaatorid** *(parser)*, millega lahatakse näiteks kompilaatoris ja interpretaatoris programmeerimiskeelt või mida saab kasutada ka tavalise keele analüüsimiseks.

Alamstringi otsimine ülesanne *(string-matching problem)*

Alamstringi otsimise ülesanne seineb järgnevas: on antud kaks stringi: tekst T pikkusega N ja nn näidis P pikkusega M, kus M<=N (mõlemaid käsitletakse kui märkidest koosnevaid massiive, kusjuures kõik märgid mõlemas massiivis peavad kuuluma ühte ja samasse kindlasse alfabeeti).

> **Alfabeet** on lõplik märkide hulk, (näiteks `S={0,1}` või `S={a,b,...,z}`).
>
> Massiive `T[1..N]`, mida alfabeedi märkidest moodustada saab, nimetatakse **stringideks** või **sõnadeks** antud alfabeedis.
>
> Näidis P on teksti T **alamstring** nihkega S *(occurs with shift S)*, kui kõik märgid näidises ja tekstis alates `S+1` positsioonist on täpselt samad. Võib ka öelda, et näidis P on teksti T alamstring alates `S+1` positsioonist.
>
> **Stringiotsimise ülesande vastuseks** on kõik nihked S, millega näidis P leitav on.

Sõltub programmeerimiskeelest, millised on selles võimalused stringidega manipuleerimiseks. Algoritmide esitamisel lähtutakse sellest, et kahe stringi võrdlemiseks, tuleb neid sümbol haaval võrrelda.

Mõisted

Stringi **pikkus** on sümbolite arv stringis, tähistame teda `|x|` (stringi x pikkus).
Alfabeedi peal moodustatav lõplike stringide hulk sisaldab ka **tühja stringi** *(empty string)* pikkusega 0.
Kahe stringi **liitmiseks** ehk **konkatenatsiooniks** nimetatakse seda, kui esimese stringi lõppu lisatakse teine string (stringide `x` ja `y` liitmine on `xy`). Uue stringi pikkus on liidetud stringide pikkuste summa (`|x|+|y|=|xy|`).

String x on teise stringi y **prefiksiks** *(prefix)*, kui stringi x sümbolid kattuvad stringi y algusega.
String x on teise stringi y **sufiksiks** *(suffix*), kui stringi x sümbolid kattuvad stringi y lõpuga. Iga stringi prefiksiks ja suffiksiks on ka tühi string.
Näiteks: `abc` on stringi `abccda` prefiks ja `da` on sama stringi sufiks.

Lihtsaim alamstringi otsimise algoritm (Naive string matching)

Lihtsaim algoritm alamstringi otsimiseks, mida peetakse ka nö jõumeetodil lahenduseks on naiivne algoritm. Algoritm seisneb järgnevas: alustatakse teksti T esimesest ja näidise P esimesest märgist ning võrreldakse neid sümbolhaaval omavahel, kuni esimese mittevõrdumise tekkimiseni või näidise lõpuni. Kui jõuti näidise lõpuni, on alamstring leitud. Kui sümbolid omavahel ei võrdu, nihutatakse tekstis järg ühe sümboli võrra edasi ja korratakse sama protsessi. Kirjeldatud tegevust (järje nihutamist ja võrdlemist) korratakse seni, kuni võrdlemine lõppeb edukalt (jõutakse näidise lõppu) või kuni tekstis jõutakse positsioonini, mis on `|tekst|-|näidis|+2`, sest siis ei mahuks näidis enam teksti lõppu ära.

Näiteks tekstist**:** `acaadaabacd` otsime näidist `aab`

```
acaadaabacd
aab
 ^
```

```
acaadaabacd
 aab
 ^
acaadaabacd
  aab
   ^
acaadaabacd
   aab
    ^
acaadaabacd
    aab
    ^
acaadaabacd
     aab
```

Algoritm, mis tekstist T otsib näidist P.

```
Naive-String-Matching(T,P)
     n<-length(T);
     m<-length(P);
     for s<-0 to n-m do
         if P[1...m]=T[s+1...s+m] then
            print('Alamstring on nihkega',s)
         endif
     endfor
```

Algoritmi tööaeg on `O((n-m+1)m) (kui n=|T|, m=|P|)`. Keskmiselt tööaeg siiski väga kehv pole.

Algoritm on naiivne, sest olles teinud ära palju tööd kahe stringi võrdlemisel (eriti kui erinevus tekkis näiteks 10 sümboli juures), visatakse omandatud teadmised nö minema ja alustatakse uuel sammul lolli järjekindlusega otsast peale. Samal ajal on tehtud juba hulk tööd ja saadud teada, mida huvitavat peidab string `'T'`, st teatakse, millises positsioonis tekkis esimene konflikt. Seda infot võiks kasutada ja proovida mööda teksti liikuda edasi mitte ühe sammu (sümboli) võrra vaid pikemate hüpetega.

Knuth-Morris-Pratti algoritm

Knuth-Morris-Pratti (KMP) algoritm kasutab järgmise nihke `s'` leidmiseks infot, mis saadakse eelmise nihke `s` juures näidise ja teksti võrdlemisel ja peaks seega kiiremini töötama võrreldes naiivse algoritmiga.
Näidisega P seotakse prefiks-funktsioon. Funktsioon annab infot selle kohta, kus ja kuidas on näidises P leitavad tema enda prefiksid. Funktsiooni abil saab vältida osade mittelubatud nihete kontrollimist.

Kui näidisest on läbi vaadatud L sümbolit, mis tekstiga kattusid ja L+1 sümbol enam ei kattu, siis L sümboli järgi on võimalik otsustada, millised nihked kindlasti mõttetud on, sest ka tekstis on samad L sümbolit. Kui näiteks näidise 1. ja 2. sümbol on erinevad ja võrdlemisel tekkis viga kaugemal, siis nihke suurendamine 1 võrra on mõtetu.

Näide:

```
T: acbababaabcbcbab
P:    ababaca
```

Nihe `S=3`, kattuvad `q=5` sümbolit ja `q+1` sümboli kohal on konflikt. Suurendades näidise nihet 1 sümboli võrra on selge, et konflikt tekib 1. võrdlemisel (a satub koahkuti b-ga). Kahe koha võrra nihutades aga kattuvad 3 sümbolit `aba` ja edasi on vaja võrdlema hakata.

Formaalsemalt:
Olgu

`P[1..q]=T[s+1..s+q].`

Milline oleks väikseim nihe `s'`, mille puhul kehtiks:

`P[1..k]=T[s'+1..s'+k]` ja `s'+k=s+q`

Ehk näidise alguse k sümbolit kattuksid tekstist läbivaadatud osa lõpust k sümboliga.

Kõige parem on, kui nihe `s'=s+q` – vahele jääks `q-1` lubamatut nihet. Väiksema nihke korral võib võrdlemata jätta esimesed k sümbolit, sest on teada, et nad langevad kokku tekstiga. Lubatavate nihete leidmiseks pole vaja tutvuda tekstiga, piisab näidise läbi vaatamisest. Näidise kohta koostatakse tabel, mille alusel saab rehkendada iga positsiooni kohta, et kui vastavas positsioonis tekkis viga, siis milline on parim nihe `s'`.

Suurus k on suurim arv (k<q), mille puhul P_k on P_q sufiksiks. Otstarbekas ongi tabelis hoida arvu k, mis näitab, mitu sümbolit peale nihet s' kokku langevad. Nihke suurus s' leitakse valemist: s'=s+(q-k)

Prefiks-funktsioon määratakse selliselt, et see on näidise P kõige pikema prefiksi pikkus, mis on ühtlasi P_q sufiksiks, seal juures peab k<q. Muidu oleks string Pq ise kattuvaks suffiksiks ja asjast poleks üldse kasu. On ilmne, et mida väiksem on k (kattuva osa pikkus), seda suurema sammuga edasi astuda saab ehk seda rohkematest mittelubatud nihetest saab korraga üle astuda.

Prefiks-funktsiooni leidmise algoritm

```
Prefix-Function(P)
m<-length[P]
Pi[1]<-0
k<-0
for q<-2 to m do
   while k>0 and P[k+1]<>P[q] do
      k<-pi[k]
   endwhile
   if P[k+1]=P[q] then
      k<-k+1
   endif
   Pi[q]<-k
endfor
```

Kui prefiksite tabel Pi on leitud, saab asuda näidist tekstiga võrdlema. Järgnev algoritm kasutab eelnevat prefiksite tabeli tegemise funktsiooni.

```
KMP(T,P)
n<-length[T]
m<-length[P]
Pi<-Prefix-Function(P)
q<-0
for i<-1 to n do
   while q>0 and P[k+1]<>P[q] do
      q<-Pi[q]
   endwhile
   if P[q+1]=T[i] then
      q<-q+1
   endif
   if q=m then
      print”Näidis kattub nihkega” i-m
      q<-Pi[q]
   endif
endfor
```

Algoritmi analüüs: kui prefiksite tabel on arvutatud, töötab algoritm ajaga O(N). Tabeli arvutamine hinnatakse samuti lineaarseks O(M). Seega kokkuvõttes on tegemist lineaarse ajaga O(M+N).

Boyer-Moore'i algoritm

Kui näidis on pikk ja alfabeet on suur (palju märke), loetakse efektiivseimaks algoritmiks R. Boyer'i ja J. Moore'i poolt leitutatut.
Kõige lihtsamal juhul on BM algoritmi puhul tegemist jõumeetodil töötava algoritmiga (naiivse algoritmiga), mis alustab võrdlemist näidise viimasest sümbolist. Samal ajal liigutakse mööda teksti ikka vasakult paremale. Kui leitakse esimnene mittevastavus, siis nihutatakse näidist ühe koha võrra mööda teksti paremale ja alustatakse uuesti võrdlemist näidise paremast otsast (sõna lõpust). Ilmselt pole selle algoritmi jõumeetodi variant oluliselt parem naiivsest algoritmist.

Tegelikult on algoritmil oluline parandus ja nihutamine ei pea toimuma ühe sümboli kaupa, juhul kui otsitavat näidist enne tekstiga võrdlema hakkamist veidi uuride. Kui teksti ja näidise märk omavahel ei võrdu, tuleb leida võimalikult pikk nihe, mille võrra näidist edasi nihutada. Parim olukord tekib siis, kui tekstis olnud sobimatut sümbolit näidises polegi. See annab võimaluse näidist nii palju edasi nihutada, et ta sobimatust sümbolist edasi algaks.

```
a b b a d a b a c b a
b a b a c
  -->     b a b a c
```

Alati nii hästi ei lähe. Kui sümbol on näidises olemas, kasutatakse kahte strateegiat ja nende hulgast valitakse alati see, mis pikema nihke annab.

Halva märgi strateegia *(Bad character heuristics)*
Kui teksti sümbol, mis ei võrdunud, on näidises olemas, nihutatakse näidist edasi nii, et sümbol näidises ja tekstis kohakuti jääksid (kui seda sümbolit on näidises mitu, pannakse kohakuti teksti vastav koht ja kõige parempoolsem selline sümbol näidises, st tehakse väikseim võimalik nihe. Seejärel alustatakse uuesti võrdlemist näidise lõpust.

Näide: `b<>c`. Näidises on kaks `b`-d – 1. ja 3. positsioonis, nihutada tuleb nii, et 3. positsiooni b teksti b-ga kohakuti satuks.

```
a b b a b a b a c b a
b a b a c
--> b a b a c
```

Realiseerimiseks koostatakse tabel, kus on kirjas kõik teksti moodustava alfabeedi märgid koos nende parempoolsema kohaga näidises. Eespool oleva näidise (`b a b a c`) jaoks on see tabel järgmine (oletame, et alfabeeti kuuluvad märgid `a, b, c, d, e`).

a	b	c	d	e
4	3	5	0	0

Mittevõrdumise koha ja tabelis oleva positsiooni järgi on võimalik leida nihke suurus: `s<-s+j-k+1`, kus
`s` – nihe
`j` – positsioon näidises, kus tekkis konflikt
`k` – tabelist leitud arv

Tegelikult kvalifitseerub selle strateegia alla ka olukord, kus teksti märki näidises pole, st tabelis on 0 (null).

Hea lõpu strateegia *(good suffix heuristics)*
Kirjeldatud strateegia ei sobi alati. Järgneva näite puhul tuleks näidist hoopis tagasi nihutada.

Näidis: `b<>a`, parempoolsem a on näidises konfliktsest kohast paremal, seega tekiks negatiivne nihe.

```
a b a a b a b a c b a
c a b a b
--> c a b a b
```

Lihtsaim variant sel puhul on teha nihe pikkusega 1 (selliselt BM-algoritmi ka mõnikord kirjeldatakse). Kavalam on arvestada sellega, mis näidises ja tekstis läbiuuritud on. Täpsemalt selgitatakse välja, kas kogu läbivaadatud osa näidises eespool kordub ja kui kordub, siis tehakse selline nihe, et see juba vaadeldud tekstiga kohakuti jääks (ühesugused osad tekstis ja näidises jäävad üksteise alla – vt eelmisest näidisest tema sufiksit `'ab'` – see kordub näidises ka eespool).

Järgnevas näites ei ole näidises olemas tervet läbivaadatud sufiksit `bab`, vaid on selle lõpp `ab`. Nihutada tuleb nii, et `ab` näidises ja tekstis kohakuti jääks. Oluline on, et see sufiks peab esinema näidise alguses.

```
a a b a b a b a c b a
a b b a b
 -->  a b b a b
```

Strateegia realiseerimiseks tehakse “heade lõppude tabel”, kus kirjas iga positsiooni `i` jaoks nihke suurus, kui positsioonis `i-1` märgid ei võrdu. Tabeli tegemiseks vaadatakse kahte juhust:
a) kokkulangev string esineb näidises eespool
b) kokkulangeva stringi lõpp esineb näidise alguses

Näide
Järgnev nihutamiste tabel annab nihke positsioonis `i`, kui konflikt tekkis positsioonis `i-1`. String on tabelis kirjas. Rida `f` on prefiksite tabel, kust saame teada antud positsioonis algava stringi pikima sufiksi alguse, mis võrdub prefiksiga. Rida s on nihete tabel, kus positsioonid 4 ja 6 täidetakse stringi `babab` järgi:

- kattuva sufiksi `bab` puhul on nihe 2 (`s[4]=4-2` – sufiksi algus – prefiksi algus)
- kattuva sufiksi `b` puhul on nihe 4 (`s[6]=6-2`)

Rea s saamiseks tuleb esimeses faasis vaadata ainult neid stringide prefikseid/sufikseid, mida vasakule poole laiendada ei saa. Kumbagi sufiksit (`bab` ja `b`) ei saa laiendada vasakule, et `p[1]<>p[3] bab` jaoks ning `p[1]<>p[5] b` jaoks. Lahtrid `s[0], s[1], s[2], s[3], s[5]` jäävad peale 1. töötlust täidetuks nullidega. 2. töötlus toimub järgmise loogika järgi: `f[0]=5` – see on näidise kõige pikema sufiksi algus, mis võrdub kogu näidise prefiksiga (`ab`). Kui tervet head lõppu kusagil mujal näidises ei leidu, võib näidist nii pikalt edasi nihutada kui nimetatud prefiks/sufiks lubavad. Et aluseks võetakse pikim kattuv prefiks/sufiks, tuleb sel hetkel, kui tabeli täitmisega sufiksi piiresse jõutakse, lülituda ümber järgmise lühema sufiksi peale. Näites täidetakse kõik 0-ga võrduvad lahtrid `f[0]`-ga.

nr	0	1	2	3	4	5	6	7
string	a	b	b	a	b	a	b	
f	5	6	4	5	6	7	7	8
s	5	5	5	5	2	5	4	1

Stringi otsimine Boyer-Moore'i algoritmi kohaselt toimub järgmiselt:
Kõigepealt ehitatakse valmis nii "halva märgi" kui ka "hea lõpu" tabelid. Kui võrdlemisel tekib konflikt võrreldakse nihkeid, mis tehtaks kummagi strateegia puhul ja valitakse nendest suurem.

Järgmises näites annab hea lõpu strateegia suurema nihke kui halva märgi starteegia. Viimase kohaselt tuleks näidist 2 võrra nihutada, et `c`-d kohakuti jääksid. Kuid näidises puudub teine sufiksi `ab` esinemine ja seetõttu võime ta terve pikkuse võrra edasi tõsta.

```
a b c a b a b a c b a
c b a a b
   -->    c b a a b
```

Võrreldes naiivset, KMP ja BM algoritme, siis korduvate sümbolite puhul (näiteks binaarne tekst), on KMP naiivsest algoritmist palju parem ja BM on KMPst natuke parem. Tavalise teksti puhul (näiteks tekst eesti keeles) on KMP naiivsest veidi parem, kuid BM on neist palju kiirem.

Karp-Rabin'i algoritm

*Richard Karp'*i ja *M. O. Rabin'*i algoritmi aluseks on idee, et arvutid on ehitatud nii, et arvutamine ja arvude võrdlemine toimub kiiremini, kui sümbolite võrdlemine. Lisaks stringide hulgas sarnasuste leidmisele töötab nimetatud algoritm ka teistes olukordades, kus on vaja tuvastada sarnasusi. Aga tutvustatud algoritmid võrdlevad stringe just märk haaval. Algatuseks kujutame, et alfabeet koosneb numbritest {0 ... 9}. Sel juhul võib m elemendist koosnevat stringi vaadata kui m kohalist arvu, arvu puhul saame aga võrrelda kogu arvu korraga. Loomulikult on siin agad – kuidas arvu tekstist kiiresti kätte saada, mida teha kui arv liiga suureks läheb? Esimese probleemi jaoks on lahendus: kui arv Ti ... Tj on teada, siis järgmise arvu leidmiseks tuleb vasakult 1 arv lahutada ja paremale üks arv liita. Suure arvu vastu võitlemiseks aitab täisarvulisel jagamisel tekkiv jääk. Võetakse mingi arv q (algarv) ja leitakse jagamisel tekkiv jääk, mis on alati 0 ja q vahel.
Algoritm kasutab sisuliselt paisksalvestuselaadset põhimõtet.

Nimelt arvutatakse otsitavale stringile pikkusega M mingi arvuline väärtus (sümbolitele, mis ei ole arvulised, seatakse vastavusse arvud), nn paiskväärtus ja sama moodi arvutatakse väärtus ka läbivaadatava teksti M-ile sümbolile. Leitud väärtusi võrreldakse omavahel. Kui väärtused on erinevad on tegemist on erinevate stringidega. Kui paiskväärtused on võrdsed, ei saa me kohe väita, et stringid samasugused on. Stringid tuleb sümbolhaaval võrrelda, et veenduda nende võrdumises. Valides q suurema on ka tõenäosus suurem, et leitud sarnasus ongi stringide võrdumine ja et ei ole tehtud asjatut tööd. Olukorda, kus arvutatud paiskväärtus on sama, kuid stringid ei võrdu nimetatakse *(spurious hit)*.

String tuleb teisendada arvuks, kuid seal juures peaks ka arvestama, mitmendana sümbol stringis paikneb. (Et näiteks sõnad `'kala'` ja `'kaal'` sama tulemust ei annaks). Edasi vaadeldakse stringi kui mingisse arvusüsteemi kuuluvat arvu ja määratakse arvusüsteemi alus. Kui näiteks kogu teksti moodustav alfabeet koosneb vaid 24 erinevast sümbolist, siis oleks arvusüsteemi aluseks 24, kui aga kahest sümbolist (nn binaarne tekst), siis oleks see 2.

Olgu sõna `sona[1..m]` (pikkusega m).
Paiskfunktsioon tema jaoks oleks:

$h(sona[1..m]) := (sona[1]*AA^{m-1} + sona[2]*AA^{m-2} + ... sona[m]*AA^{0}) \bmod q$
kus `q` on suur arv ja AA on arvusüsteemi alus.

Erinevad arvusüsteemi aluse astmed annavad tähtedele erinevad kaalud (nagu arvusüsteemide puhulgi).

Algoritm paiskväärtuse arvutamiseks:
Iga sümboliga sõnast tee
  teisenda ta 10ndsüsteemi arvuks
  liida ta eelnevale summale juurde, olles enne summat korrutanud AAga

Seda tehakse nii näidisega kui ka teksti esimese M sümboliga.

Algoritm võrdlemiseks:
Arvuta paiskväärtus näidisele.
Arvuta paiskväärtus näidise pikkusele osale tekstis.
Korda kuni pole leitud samasugust stringi ja pole jõutud teksti lõppu
    Korda kuni pole leitud sama paiskväärtust ja pole jõutud teksti lõppu
        paiska tekst ümber (lahuta 1. sümbol ja liida järgmine sümbol)
(Kui leitud vastav paiskväärtus), võrdle näidist ja vaadeldavat tekstiosa sümbolhaaval

Vastav programmifragment Pascalis oleks järgmine:

```
found := FALSE; search := 0;
m := length(sona);
if m=0 then begin
   search := 1;  found := TRUE; end;
{Arvutame välja esimesed paiskväärtused sonale ja tekstile}
Bm := 1;
hsona := 0;  htekst := 0;
n := length(tekst);
if n >= m then
   for j := 1 to m do begin
      Bm := Bm*B;
      hsona := hsona*B + ord(sona[j]);
      htekst := htekst*B + ord(tekst[j]);
   end;
{Ja nüüd hakkame tegelikult otsima}
j := m;
while not found do begin
{Võrreldakse paiskväärtusi}
     if (hsona = htekst) then
        if (sona = copy(tekst,j-m+1,m)) then
        begin
            search := j-m+1;
            found := TRUE;
        end;
{Kui on veel teksti, siis arvutame uue paiskväärtuse}
        if j < n then
        begin
            j := j+1;
            htekst := htekst*B - ord(tekst[j-m])*Bm + ord(tekst[j]);
        end
        else
           found := TRUE;
     end;
 end;
```

Tulemus satub search'i. Kui Search = 0, siis stringi ei leitud. Search võib olla näiteks funktsiooni nimi, sel juhul tagastab funktsioon otsitava stringi positsioone tekstis või 0, kui teda ei leidunud.

Failide pakkimine

Failide pakkimise algoritmide ülesanne on vähendada failide mahtu ja tagada samal ajal info taasesitamise võimalus kogu mahus. Osa algoritme on aga sellised, kus kogu info 100% taastatav ei ole.

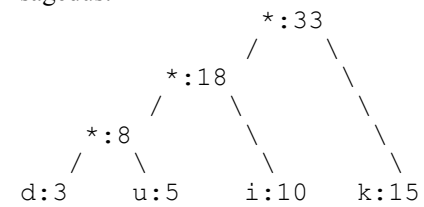
Pakkimisalgoritmid jagatakse nelja suuremasse klassi:

1. Pika rea kood *(Run-Length Encoding)*
2. Muutuva pikkuse kood *(Variable-Length Encoding)*
3. Asenduspakkimine *(Substitutional Compressors)*
4. Pildi ja video pakkimine *(JPEG ja MPEG)*

Erinevad algoritmid on sobivad erinevat tüüpi failide korral.

Pika rea kood

Meetod tugineb faktile, et teatud tüüpi failides on järjest pikad read ühesuguseid sümboleid. Kui failis oleks näiteks kirjas:
`xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyzzzzzzzzzzzzz`
siis võiks seda kodeerida:
`16x 13y 13z`
Kahendfaili puhul:
`11111111110000000011111`
võiks aga kodeering olla:
`10 8 5`
Pole vaja määrata, millist sümbolit kodeeritakse – 1 ja 0 on vaheldumisi.
Kui korduvate sümbolite read on pikad, on meetodist kasu. Tavaliste tekstifailide puhul pole meetod eriti kasulik ja sobib pigem rasterpiltide pakkimiseks. Mitmed pildifaili formaadid seda ideed ka kasutavad.
Muutuva pikkusega kood
Meetod kasutab fakti, et tekstis esinevad mõned sümbolid tihedamini kui teised. Erinevate keelte puhul on tähtede esinemissagedused erinevad. Kui tavaliselt kasutatakse iga tähe kodeerimiseks 8 bitti, siis selle pakkimismeetodi puhul tehakse kodeering ringi – levinuim sümbol asendatakse ühe bitiga, järgmine 2 bitiga jne. Sõna 'kui' tuleks kodeerida `3x8` bitiga, kuid kodeerides tähed ringi esinemissageduse alusel: k – 1, i – 10 ja u – 101 saaksime vastavalt 1 101 10.
Kasu annaks tühikutest loobumine. Kui kodeering teha selline, et ühe sümboli kood ei oleks teise koodi alguseks, võiks faili kirjutada ilma tühikuteta.
Sellist kodeerimisskeemi nimetatakse **Huffmanni kodeerimiseks** tema väljatöötaja auks. Vastavalt esinemissagedusele ehitatakse kahendpuu, kus lubatud koodid on kõik lehtedeks.
Olgu meil tähed oma esinemissagedustega:
k:15, u:5, i:10 ja d:3
Puu ehitamist alustatakse kõige vähem esinevatest sümbolitest – nad ühendatakse ja sellest saadakse summaarne sagedus.



Koodid saadake järgmised:
`d:0001  u:001  i:01  k:1`
Dekodeerimisel tähendab 0 – mine vasakusse alampuusse ja 1 – mine paremasse alampuusse. Lehest leitakse vastav sümbol.
Meetod on kasutatav tekstifailide korral.

Asenduspakkimine

Meetodi ideeks on asendada mingi fraas või lihtsalt sümbolite jada viitega selle jada eelmisele esinemisele samas tekstis. Asendamiseks on kaks erinevat skeemi, mis on välja pakutud Jakob Ziv'i ja Abraham Lempel'i poolt aastatel 1977 ja 1978. Vastavalt räägitakse LZ77 ja LZ78 pakkijatest.
LZ78 ideeks on koostada sõnastik ja faili kirjutada sõnastiku indeksid sõna asemel. Sõnastik konstrueeritakse teksti läbimisega samaaegselt. Uusi stringe lisatakse vaid siis, kui nad sõnastikku laiendavad ja enamasti kipuvad sinna sattuma pikad fraasid. Pakkimise tugevus sõltub ilmselt sõnastiku pikkusest ja kindlasti tuleb sõnastik pakitud faili kaasa panna.
LZ77 idee kohaselt jälgitakse n viimast sümbolit ja kui leitakse sama fraas, mis juba oli, pannakse teksti fraasi asemel tema algus ja pikkus.

Kui tekstis on korduvad alamstringe, on see meetod sobiv.

JPEG ja MPEG

Kirjeldatud meetodid pakivad kadudeta – pakitud fail on 100% õigesti taastatav. Kadudega pakkimismeetodid on JPEG ja MPEG, kus lähtutakse sellest, pilt või kino on vaadatavd ka siis, kui iga punkt ekraanil pole täpselt selline nagu ta

originaalis oli. Silm erinevusi väga täpselt ei erista. JPEG-iga saab väikese kadude hinnaga saavutada suurt kokkupressimist. See on pildi (ja video)-failide juures väga oluline. JPEG-i juures visatakse kõrvale kõrged sagedused ja seejärel võivad kaotsi minna kiired muutsed, samal ajal säilitatakse aeglased muudatused.
MPEG- kopressib kadudega videot. JPEG-laadset pakkimist kasutatakse ühe kaadri jaoks. Edasi salvestatakse muudatused kaadrite vahel.