Aho-Corasick -algoritmi

Tietorakenteiden harjoitustyö, Syksy 2010

Ohjaaja: Matti Luukkainen

Tekijä: Janne Vento

Tietojenkäsittelytieteen laitos

janne.vento@cs.helsinki.fi

TiraLabra: Aho-Corasick –algoritmi

Käyttökohde:

Merkkiesiintymien etsiminen annetusta tekstistä yhdellä läpikäynnillä.

Toiminta-ajatus:

Algoritmin toiminta perustuu tila-automaattiin, jonka siirtymät saadaan hakusanoista muodostetusta Triepuu-rakenteesta. Tilojen joukkoon liittyy tulostefunktio, jonka perusteella tulostetaan tilaan liittyvät hakusanat.

Vaihe 1: Hakusanojen lisääminen

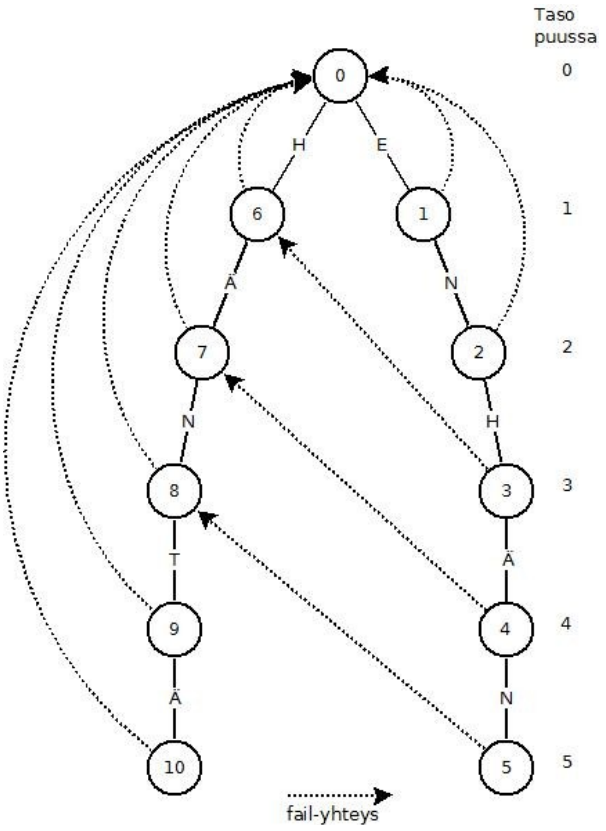
Alussa Trie-puu sisältää yhden solmun, juuren. Syötteenä saatu hakusana lisätään alkaen juuresta merkki kerrallaan siten, että kullekin merkille on oma kaarensa ja kaaren päässä uusi solmu. Oheisen kuvan (Kuva 1) puuhun on lisätty sanat seuraavassa järjestyksessä: 'ENHÄN', 'HÄN' ja 'HÄNTÄ'. Viimeisen sanan kohdalla tulee ilmi, kuinka kaaria seurataan, kunnes kirjaimelle ei löydy enää sopivaa kaarta, jolloin luodaan uusi kaari.

Jokaisen sanan lisäämisen jälkeen lisätään tilaan liittyvät tulosteet. Esimerkkipuussa tulostefunktioon olisi lisätty tulosteet tiloille 5, 8 ja 10 (f(5)=”ENHÄN”, f(8)=”HÄN” ja f(10)=”HÄNTÄ”).

Vaihe 2: Siirtymien muodostaminen

Siirtymien muodostuminen on hyvin suoraviivaista. Solmun kaariin liittyy aina jokin kirjain, joka suoraan määrää siirtymän solmusta solmuun. Poikkeuksena tilanteet jolloin jollekin kirjaimelle ei löydy kaarta. Tasolla 0 nämä siirtymät osoittavat takaisin juureen, mutta korkeammilla tasoilla saadaan tuloksena epäonnistunut haku(fail).

Kun kaarista ei löydy sopivaa kaarta työn alla olevalle kirjaimelle, tarvitaan ns. failuresiirtymä (tai *korjaussiirtymä*). Oletuksena failuresiirtymä kohdistuu juurisolmuun. Esimerkkipuun tilat 3, 4 ja 5 ovat poikkeuksia. Esimerkiksi tilassa 5 oltaessa on luettu jo kirjaimet HÄN, mutta kaarta kirjaimelle T ei löydy. Failuresiirtymä siirtääkin aktiivisen tilan aina puussa matalammalle tasolle, jonka jälkeen yritetään etsiä kaarta tästä tilasta. Tätä toistetaan kunnes siirtymä onnistuu. Onnistuminen varmistetaan sillä, että haut onnistuvat aina oltaessa puun tasolla 0.



Kuva 1

Failuresiirtymien muodostaminen tapahtuu seuraavasti:

Merkitään:

- ***g(s,c)*** tarkoittamaan siirtymä tilasta ***s*** kirjaimella ***c***
- ***f(s)*** tarkoittamaan korjaussiirtymää tilasta ***s***
- ***tuloste(s)*** tarkoittamaan tilaan ***s*** liittyviä tulosteita
-

Ensin asetetaan juurisolmulle ***f(0)=0***.

Tämän jälkeen juuren lapset ***f(1)=f(6)=0***. Lisätään solmut 1 ja 6 jonoon.

Seuraavaksi otetaan solmu r jonosta ja käydään sen kaaret läpi yksitellen seuraavasti:

- Olkoon ***s*** käsittelyssä olevan kaaren päässä oleva solmu.
- Lisätään ***s*** jonoon.
- Kaareen liittyvä kirjain olkoon ***a***.
- Asetetaan ***state = f(r)***. Niin kauan kuin ***g(state, a) = fail***, asetetaan ***state = f(state)***.
- Asetetaan ***f(s) = g(state,a)***
- Yhdistetään tulosteet ***output(s) = output(s) + output(f(s))***

Selitys:

Solmuun ***s*** on aina tultu jollain kirjaimella ***a***.

Solmun ***s*** failuresiirtymää varten tutkitaan pääseekö sen vanhemman failuresiirtymän osoittamasta solmusta kirjaimella ***a*** johonkin solmuun(***state***). Jos pääsee, niin saatu kohdesolmu ***state*** asetetaan käsittelyssä olevan solmun ***s*** failuresiirtymäksi. Muutoin tutkitaan solmun ***state*** failuresiirtymää ja tätä toistetaan kunnes löytyy solmu josta pääsee syvemmälle puuhun kirjaimella ***a***.

Haku pysähtyy koska se kohdistuu puussa aina juurta kohti ja juuressa haku onnistuu aina.

Vaihe 3: Materiaalin läpikäynti

Etsinnän aluksi tilaksi asetetaan 0. Jokainen kirjain merkitsee vähintään yhtä tilasiirtymää(huomionarvoista on, että juurisolmusta ja sen lapsisolmuista voidaan tilasiirtymien jälkeen päätyä samaan tilaan kuin alkutilanteessakin). Koska tilasiirtymät voivat suuntautua joko yhden askeleen syvemmälle puussa tai failuresiirtymien kautta matalammalle tasolle jolloin siirtymiä voi olla maksimissaan lähtötilan korkeustason verran. Poikkeuksena luonnollisesti juurisolmu, jonka taso on 0, mutta josta voidaan silti tehdä siirtymä.

Jos siirtymäfunktio ilmoittaa epäonnistuneesta siirrosta, siirrytään failuresiirtymän osoittamaan tilaan ja yritetään siirtyä tästä tilasta samalla kirjaimella. Tätä toistetaan, kunnes saadaan siirtymäfunktiolta onnistunut siirtymä.

Tilasiirtymän jälkeen tutkitaan liittyykö tilaan tulostetta, eli ollaanko löydetty jokin hakusana.

Aikavaativuuksia

Tilasiirtymät ovat vakioaikaisia. Kaaria solmulla on n kappaletta, jotka ovat suuruusjärjestyksessä. Näiden valitsemiseen käytetään binäärihakua, jonka aikavaativuus on O(log n). Tilanteessa jossa kirjaimelle ei ole kaarta, joudutaan tilasiirtymiä tekemään maksimissaan h (= solmun korkeus puussa). Epäonnistuneessa siirtymässä joudutaan ensin aina läpikäymään kaarien joukko (log n) ja sen jälkeen tekemään korjaussiirtymiä aina juureen saakka.

Tilojen tulosteet on taulukossa ja yhteen tilaan voi liittyä useita tulosteita, joten sen tulostaminen on aikavaativuudeltaan lineaarinen suhteessa tilan tulosteiden määrään.

Tietorakenteet ja algoritmit

Ohjelmassa käytetyt oleellisimmat tietorakenteet ovat jono, muuttuvapituuksinen taulukko ja TriePuu.

Jono(JonoLista) on yhteen suuntaan linkitetty lista, johon alkiot lisätään loppuun ja poistettaessa otetaan jonosta ensimmäinen. Toteutuksessa jonon solmuihin tallennetaan viittaus TriePuun solmuun(TrieSolmu). Jonoa tarvitaan kun käydään puuta läpi leveyssuuntaisesti.

Jonon lisäys- ja poisto-operaatiot ovat vakioaikaisia.

Taulukkototeutus(DynaaminenTaulukko) on geneeristä tyyppiä ja osaa käsitellä mitä tahansa olioita. Lisättäessä täyteen taulukkoon, sen koko kaksinkertaistetaan.

Taulukon lisäysoperaatio on yleisesti ottaen vakioaikainen, paitsi jos taulukon varaama tila loppuu ja sen kokoa joudutaan kasvattamaan. Tällöin joudutaan luomaan uusi isompi taulukko, kopioimaan kaikki alkiot uuteen taulukkoon ja suorittamaan vielä alkuperäinen lisäys.

Trie-rakenteen muodostaminen hoituu TriePuu luokalla. Puun solmuihin liittyy aina KaariJoukko-olio, jonka sisältämiin kaariin liittyy tieto kirjaimesta ja solmusta kaaren päässä. Kaarijoukko pitää kaaret aina järjestyksessä, käyttäen hyväksi char-tietotyypin vertailumahdollisuutta. Etsittäessä tästä joukosta käytetään binäärihakua, jonka aikavaativuus on O(log n).

KaariJoukko-luokkaa voi myös ajatella eräänlaisena ArrayList:n johdannaisena, se on kuitenkin toteutettu erillään johtuen sen tiukasta yhteydestä TriePuun solmuihin.

Hakusovelluksen käyttö:

Ohjelman vaatimat luokat löytyvät lähdekoodina ja käännettynä jar-pakkauksesta nimeltä ahoCorasickSearch.jar. Paketti sisältää myös testaukseen käytetyt luokat lähdekoodeineen.

Käytön kannalta helpointa on kopioida paketin sisältämästä kansiosta *ahoCorasick kaikki* class-päätteiset tiedostot samaan hakemistoon, kuin missä niitä käyttää. Paketin käyttöön suoraan jar-pakkauksesta voi lukea ohjeita esimerkiksi internetistä.

Ohjelman käyttöönotto projektiin tapahtuu tuomalla paketti rivillä:

import ahoCorasick.*.

AhoCorasickSearch-olio luodaan parametrittomalla konstruktorilla:

AhoCorasickSearch hakukone = new AhoCorasickSearch()

tai parametrina voidaan antaa määre isojen ja pienien merkkien erottelusta(epätosi samaistaa isot ja pienet):

AhoCorasickSearch hakukone = new AhoCorasickSearch(boolean)

Hakusanojen lisäys tapahtuu metodilla

lisaaHakuSana(String)

Jonka jälkeen voidaan alkaa syöttämään tekstiä metodilla

String etsiMerkkijonosta(String tutkittavaTeksti),

jonka paluuarvona tulee String-olio, joka sisältää löytyneet hakusanat ja merkin järjestysnumeron ensimmäisesti syötetystä lähtien. Palautteen muoto on ”{n, string}”.

Hakukoneen lukemien merkkien määrä saadaan metodilla

int annaMerkkienMaara()

ja hakuun liittyviä tietoja voidaan tulostaa metodilla

tulostaTiedot()

Jar-paketissa tulee myös ACHaku ja ACHaku_v2-ohjelmat, joiden avulla voidaan suorittaa hakuja tiedostoista.

Haku suoritetaan seuraavasti (käytetään suoraan jar-paketista):

java -cp ahoCorasickSearch.jar ACHaku <tiedostonimi> <hakusana$_1$> … <hakusana$_n$>

Tiedostonimi sisältää hakumateriaalin. Hakusanoja voi olla useita. Käytä lainausmerkkejä hakiessasi useampaa sanaa.

java -cp ahoCorasickSearch.jar ACHaku_v2 <tied.nimi> <hakusanatied.nimi>

Jälkimmäinen parametri on hakusanat sisältävän tiedoston nimi.

Kehitysajatuksia

Haun tehokkuuden kannalta itse tilakoneen muodostaminen näyttelee hyvin pientä roolia, oleellisempaa on se kuinka tehokkaasti sopiva kaari löydetään kaarien joukosta. Ohjelma käyttää etsintään binäärihakua, jolloin haun aikavaativuus on luokkaa O(log n). Tätä voitaisiin tehostaa huomattavasti eri tavoin.

Yksi tapa olisi käyttää hajautustaulua, jolloin kaaren etsintä olisi vakioaikaista(O(1)). Hajautustaulun hankalana puolena olisi hyvän hajautusfunktion löytäminen, koska se riippuu aineistosta. Etukäteen tunnetulla aineistolla, esimerkiksi kirjojen sisällöstä haettaessa, hajautusfunktio olisi hyvinkin helposti muodostettavissa, mutta

Toinen vaihtoehto olisi rajata mahdollisten merkkien määrää ja käyttää taulukkoa kaaren valintaan, siten että kirjaimen ascii-koodi(tai unicode, tms) toimisi indeksinä. Tällöin haku olisi myöskin vakioaikaista. Tämän ratkaisun huonona puolena olisi eksponentiaalisesti kasvava muistintarve.

Eräs parannustapa olisi muodostaa deterministinen äärellinen automaatti, jolloin jokaista syötettä vastaisi täsmälleen yksi tilasiirtymä. Tässä ratkaisussa failuresiirtymien kautta löydetty tila kytkettäisiin suoraan alkuperäisen tilan siirtymäksi. Tällä ratkaisulla voitettaisiin tilasiirtymien määrässä,kun failuresiirtymiä ei tarvitsisi tehdä, mutta muistinkäyttö kasvaisi hieman. Ratkaisu olisi hyödyllinen tilanteessa, jos hakusanat muodostaisivat paljon failuresiirtymiä. Käytännössä toiminta keskittyy juuren lähistölle, joten failuresiirtymien poiston hyödyt ovat hieman kyseenalaiset.

Jos hakukonetta haluttaisiin käyttää muissa sovelluksissa, tulisi muokata tapaa millä syötetään hakumateriaalia koneeseen ja varsinkin kuinka tulokset saadaan ulos. Tehokkaan käytön kannalta tulisi olla mahdollista etsiä useamman tiedoston/lähteen seasta ja tietää jälkikäteen mistä on löydetty mitäkin. Tällainen voitaisiin toteuttaa esimerkiksi siten, että löytyneet löytyneet hakusanat sijainteineen tallennettaisiin taulukkoon, joka pysyisi hakukoneen muistissa ja sen saisi pyydettyä metodilla erikseen. Tällöin tuloksia ei tarvitsisi tulostaa ruudulle haun aikana ja niiden käsittely jälkikäteen olisi helpompaa.

Testaus

Tietorakenteina käytetyt jono, dynaaminen taulukko, Triepuu ja kaaret sisältävä Kaarijoukko-luokka on yksikkötestattu käyttäen JUnitia.

Testaus tutki ainoastaan tietorakenteiden toiminnan oikeellisuutta, varsinaista tehokkuustestiä ei suoritettu. Tehokkuustestiä varten olisi syytä kytkeä tulosteet pois päältä ja sitä optiota ei ohjelmassa ole.

Testiin käytetyt luokat löytyy samasta jar-paketista, mistä itse ohjelmakin.

Lähteet:

Alfred V. Aho, Margaret J. Corasick – Efficient String Matching: An Aid To Bibliographic Search (Communications of the ACM, vol 18, nro 6, s. 333-340)