Jaottomien polynomien löytäminen äärellisten kuntien yli - Toteutusdokumentti

Aineopintojen harjoitustyö: Tietorakenteet ja algoritmit

Sebastian Björkqvist

7. syyskuuta 2014

Ohjelman rakenne

Käytettävät algoritmit on jaoteltu eri paketteihin käyttötarkoituksen mukaan. Lopullinen, jaottoman polynomin löytämiseen käytettävä algoritmi löytyy paketin polynomial luokasta IrreduciblePolynomialFinder. Tämä luokka taas käyttää saman paketin luokkaa PolynomialUtil, joka puolestaan käyttää math-paketin luokkaa MathUtil sekä polynomirajapintaa IPolynomial.

Koko projektia kuvaava luokkakaavio löytyy generoitujen javadocien sivulta Overview.

Käytetyt tietorakenteet

Polynomit

Algoritmin perustana ovat polynomit, jotka on toteutettu sekä kahteen suuntaan linkitettynä listana (LinkedListPolynomial) että taulukon avulla (ArrayPolynomial). Molemmat toteutukset toteuttavat rajapinnan IPolynomial. Polynomit hoitavat itse omat peruslaskutoimituksensa (yhteen-, vähennys, kerto- ja jakolasku), ja mikään laskutoimitus ei muokkaa polynomeja jotka osallistuvat kyseiseen laskutoimitukseen, vaan laskutoimituksen tuloksena on aina uusi polynomi jolla ei ole mitään riippuvuutta lähtöpolynomeihin. Kaikki laskutoimitukset on toteutettu naiivisti: yhteen-, vähennys- ja kertolaskussa loopataan polynomit läpi, ja jakolasku on toteutettu polynomien jakokulman avulla. Laskutoimitusten sekä polynomin evaluoinnin aikavaativuudet on näkyvät taulukossa 1.

Toimenpide	Aikavaativuus
Yhteenlasku	$\mathcal{O}(n)$
Vähennyslasku	$\mathcal{O}(n)$
Kertolasku	$\mathcal{O}(n^2)$
Jakolasku	$\mathcal{O}(n^2)$
Evaluointi	$\mathcal{O}(n\,log\,n)$

Taulukko 1: Polynomien laskutoimitusten sekä evaluoinnin aikavaativuudet. Tässä n viittaa polynomien asteeseen.

Laskutoimituksia ei voi suorittaa polynomeilla joilla on eri toteutus. Tämänkin voisi toki implementoida, mutta polynomin tyypin vaihto hidastaisi laskutoimituksia niin paljon että se ei ole järkevää. Mielummin kannattaa alussa valita mitä polynomitoteutusta käytetään ja pysyä siinä.

Muut tietorakenteet

Polynomien lisäksi ainoa itse toteutettu tietorakenne on kahteen suuntaan linkitetty lista SortedIntegerList joka koostuu IntegerNode-olioista. Nimensä mukaisesti SortedIntegerList voi sisältää ainoastaan kokonaislukuja. Lisäksi se pitää itsensä aina järjestettynä.

Jaottoman polynomin löytäminen

Algoritmi jaottoman polynomin löytämiseksi muodostuu kahdesta osasta: Ensin valitaan jonkin heuristiikan avulla polynomi jonka jälkeen polynomille ajetaan Rabinin jaottomuusalgoritmi. Tätä menetelmää toistetaan kunnes jaoton polynomi löytyy.

Rabinin jaottomuusalgoritmi

Rabinin jaottomuusalgoritmi saa syötteenään jonkin yhden muuttujan polynomin f. Olkoon seuraavassa polynomin f aste d ja sen karakteristika c. Algoritmi etenee seuraavasti ([1]):

1. Lasketaan kaikki asteen d uniikit alkutekijät. Merkitään näitä $p_1, \ldots, p_k$.
2. Lukujen p_i avulla lasketaan luvut $d_i = d/p_i$.
3. Lasketaan jokaiselle d_i polynomi $g_i = x^{c^{d_i}} - x \bmod f$.

4. Lasketaan sitten polynomien f ja g_i suurin yhteinen tekijä. Jos suurin yhteinen tekijä ei ole vakio, niin polynomi f **on jaollinen** ja algoritmin suoritus loppuu.

5. Lopuksi lasketaan polynomi $g = x^{c^d} - x \bmod f$. Jos g on nollapolynomi niin f **on jaoton**. Muuten f **on jaollinen**.

Algoritmi siis käyttää apunaan kolmea muuta algoritmia:

Kohdassa 1 lasketaan kaikki asteen d uniikit alkutekijät. Tämä tehdään Eratostheneen seulan avulla jolloin aikavaativuus on $\mathcal{O}(d)$.

Kohdassa 4 lasketaan polynomien f ja g_i yhteinen tekijä. Tämä tehdään Eukleideen algoritmin avulla. Laskettaessa polynomia g_i otetaan lopuksi aina mod f, joten sen aste on korkeintaan d. Koska algoritmin joka askeleella lasketaan jakolasku, on yhden askeleen aikavaativuus $\mathcal{O}(d^2)$, ja suurimman yhteisen tekijän algoritmin aikavaativuus on täten $\mathcal{O}(kd^2)$, missä k on tarvittavien askelten määrä. Tämä määrä on varmasti vähemmän kuin polynomin aste d, eli syt-algoritmin aikavaativuus on $\mathcal{O}(d^3)$.

Kohdassa 3 ja 5 lasketaan polynomit $g_i = x^{c^{d_i}} - x \bmod f$ ja $g = x^{c^d} - x \bmod f$. Tämä laskenta on toteutettu käyttäen binääristä neliöintiä (*repeated squaring*). Idea on seuraava: Aloitetaan polynomista x^c, ja kerrotaan kyseistä polynomia itsensä kanssa c kertaa (ja ottamalla joka kerta modulo f), jolloin saadaan polynomi $x^{c \cdot c} = x^{c^2} \bmod f$. Jatketaan sitten kertomalla polynomia x^{c^2} itsensä kanssa c kertaa jolloin saadaan polynomi $x^{c \cdot c^2} = x^{c^3} \bmod f$ jne. kunnes saavutaan polynomiin $x^{c^{d_i}} \bmod f$. Lopuksi vielä poistetaan x polynomista ja otetaan se modulo f, jolloin päädytään haluttuun polynomiin g_i.

Algoritmi suorittaa siis korkeintaan $c \cdot d$ laskutoimitusta. Koska jokaisessa algoritmin vaiheessa lasketaan polynomeja modulo f, on laskettavan polynomin aste korkeintaan $2d$. Täten, kun ensiksi kerrotaan kaksi korkeintaan astetta $2d$ olevaa polynomia ja sitten suorittamalla jakolasku kahden korkeintaan astetta $2d$ olevan polynomin välillä (jotta saadaan polynomi modulo f), on jokaisen askeleen aikavaativuus $\mathcal{O}(2 \cdot 4d^2) = \mathcal{O}(d^2)$. Koska askelia on korkeintaan $c \cdot d$ kappaletta, on siis kohdan 3 ja 5 alialgoritmin kokonaisaikavaativuus korkeintaan $\mathcal{O}(cd^3)$. Kohdat 3 ja 5 siis dominoivat muita kohtia aikavaativuuden suhteen, ja täten algoritmin yhteisaikavaativuus voidaan laskea näiden kohtien aikavaativuuden avulla.

Algoritmin kohta 3 suoritetaan $\omega(d)$ kertaa, missä $\omega(d)$ kuvaa luvun d uniikkeja alkutekijöitä. Hardyn-Ramanujanin lauseen mukaan uniikkeja alkutekijöitä on noin $log(log\, d)$ kappaletta ([2]). Täten Rabinin algoritmin toteutuksen kokonaisaikavaativuus on $\mathcal{O}(cd^3\, log(log\, d))$. Rabinin artikelissa ([1] mainitaan että algoritmi voidaan toteuttaa niin, että sen aikavaativuus

on vain $\mathcal{O}(log\, c\, d^2\, log^2\, d\, log(log\, d))$. Tähän vaaditaan paremmin toteutettuja polynomien laskutoimituksia (esimerkiksi Fast Fourier Transform-algoritmia taulukkopolynomeille). Lisäksi kohtien 3 ja 5 algoritmia voisi tehostaa hieman ja pudottaa sen aikavaativuus tasolle $\mathcal{O}(log\, c\, d^3)$ suorittamalla binäärinen neliöinti paremmin. Pienillä karakteristikoilla (joihin tässä työssä keskityttiin) tällä ei kuitenkaan ole juurikaan merkitystä.

Kandidaattipolynomien generointi

Yleinen periaate jaottomien polynomien löytämiseen on seuraava: Generoidaan polynomeja satunnaisesti ja ajetaan niille Rabinin jaottomuustesti. On osoitettu että jos polynomi valitaan täysin satunnaisesti niin tarvitaan keskimäärin n yritystä kunnes jaoton polynomi löytyy ([3]). On kuitenkin helppo nähdä että tietyt polynomityypit eivät koskaan ole jaottomia, joten näitä polynomeja on turha ajaa aikaavievän Rabinin jaottomuusalgoritmin läpi. Seuraavat lemmat antavat muutaman tavan karsia pois polynomeja.

Lemma 1. *Olkoon f polynomi jonka kertoimet ovat kunnassa $\mathbb{Z}_p$, eli kunnassa jonka karakteristika on p, ja olkoon sen aste $d \geq 2$. Tällöin jos f:llä on juuri, se ei ole jaoton kunnan Z_p yli.*

Todistus. Jos f:llä on juuri a, niin se voidaan kirjoittaa muodossa $f(X) = (X - a) \cdot g(X)$, missä g on polynomi jonka aste on $d - 1$. □

Lemma 2. *Olkoon f polynomi jonka kertoimet ovat kunnassa $\mathbb{Z}_p$ ja olkoon sen aste $d \geq 2$. Mikäli f:llä ei ole vakiokerrointa, se ei ole jaoton.*

Todistus. Oletetaan että f:llä ei ole vakiokerrointa, eli se on muotoa $a_d X^d + \cdots + a_1 X$. Tällöin luku 0 on f:n juuri, joten edellisen lemman nojalla f ei ole jaoton. □

On siis täysin turhaa generoida polynomeja ilman vakiokertoimia, koska ne eivät kuitenkaan ole jaottomia. Koska puolella polynomeista ei ole vakiokerrointa, puolittuu tarvittavien yritysten määrä kunhan pidetään huoli siitä että generoitu polynomi sisältää vakiokertoimen. Ohjelman sisältämä heuristiikka **naive** generoi polynomeja täysin satunnaisesti, mutta varmistaa kuitenkin että se sisältää vakiokertoimen.

Ylläolevat lemmat antavat myös toisen tavan karsia polynomeja pois ennen kuin ne annetaan Rabinin jaottomuusalgoritmille: Evaluoidaan polynomi jokaisessa kunnan Z_p alkiossa ja katsotaan löytyykö polynomille juuri. Jos juuri löytyy, niin tiedetään että polynomi ei ole jaoton ja voidaan siirtyä seuraavaan polynomiin. Polynomin evaluointi voidaan helposti tehdä ajassa

$\mathcal{O}(n \log n)$, joten tämä ylimääräinen tarkistus ei vaikuta algoritmin asymptoottiseen aikavaativuuteen. Tämä juurien tarkistamiseen keskittyvä heuristiikka kulkee ohjelmassa nimellä **checkroots**. Tällä tavalla pystytään karsimaan pois osa polynomeista, mutta karsittujen polynomien tarkka määrä ei ole tiedossa yleisessä tapauksessa.

Jos kerroinkunta on $\mathbb{Z}_2$ pystymme tarkistamaan onko polynomilla juuria ilman että sitä pitää evaluoida:

Lemma 3. *Olkoon f polynomi jonka kertoimet ovat kunnassa $\mathbb{Z}_2$ ja joka sisältää vakiokertoimen. Tällöin f:llä on juuri jos ja vain sillä on parillinen määrä nollasta eroavia kertoimia, eli toisin sanoen parillinen määrä kertoimia 1.*

Todistus. Olkoon n f:n nollasta eroavien kertoimien lukumäärä.

Jos n on parillinen, niin pätee $f(1) = 0$. Tämä siksi, että $1^k = 1$ kaikilla k, joten $f(1) = n = 0$, sillä $n \equiv 0 \bmod 2$.

Jos taas n on pariton, niin $f(1) = 1$ ja myös $f(0) = 1$. Edellinen pätee, koska $f(1) = n = 1$, ja jälkimmäinen koska f:llä on vakiokerroin (joka siis on 1). □

On selvää, että puolet vakiokertoimen sisältävistä polynomeista jonka kertoimet ovat kunnassa $\mathbb{Z}_2$ sisältää parillisen määrän nollasta eroavia kertoimia. Täten ylläolevan lemman avulla voimme pudottaa tarvittavien yritysten määrän neljäsosaan alkuperäisestä mikäli kerroinkunta on $\mathbb{Z}_2$. Ohjelmassa tätä tapaa generoida polynomeja kutsutaan nimellä **smartchar2**.

Ei ole täysin samantekevää minkälaisen jaottoman polynomin algoritmi palauttaa. Koska polynomia on periaattessa tarkoitus käyttää äärellisen kunnan generointiin, on polynomi parempi mitä vähemmän nollasta eroavia kertoimia sillä on. Tämä siksi että äärellisessä kunnassa kaikki laskutoimitukset tehdään modulo jaoton polynomi, ja polynomin laskutoimitusten aikavaativuudet riippuvat polynomin monomien määrästä (kun käytetään listaesitystä).

Samalla voisi olettaa että myös algoritmi itse nopeutuu hieman jos se generoi lyhyempiä polynomeja, koska algoritmihan suorittaa valtavan määrän laskutoimituksia polynomeilla. Täten ohjelma sisältää myös heuristiikkavaihtoehdon missä jokainen generoidun polynomin kerroin (lukuunottamatta korkeinta kerrointa ja vakiokerrointa) on 70 prosentin todennäköisyydellä nolla. Koska jokainen polynomi on kuitenkin mahdollista generoida tälläkin tavalla, löytyy jaoton polynomi varmasti jossain vaiheessa. Tällöin kuitenkaan yllämainittu tulos siitä, että joudutaan kokeilemaan keskimäärin n kappaletta polynomeja ei enää päde, koska polynomeja ei generoida tasaisen jakauman

mukaan. Tehdyissä testeissä (joista kerrotaan lisää testausdokumentissa) vaikutta kuitenkin siltä, että lyhyempien polynomien valinta ei vaikuta yritysten määrään, ja lisäksi algoritmi on tässä tapauksessa marginaalisesti nopeampi. Lyhyitä polynomeja generoivan heuristiikan nimi ohjelmassa on **sparse**.

Heuristiikat checkroots ja smartchar2 käyttävät oletuksena naiivia generointia, mutta ne voidaan yhdistää myös sparse-generoinnin kanssa. Taulukossa 2 on yhteenveto ohjelmaan implementoiduista heuristiikoista:

Nimi	Lisäys aikavaativuuteen	Keskim. yritysmäärä
naive	$\mathcal{O}(1)$	$n/2$
checkroots	$\mathcal{O}(n \log n)$	Alle $n/2$
smartchar2	$\mathcal{O}(1)$	$n/4$
sparse	$\mathcal{O}(1)$	Ei tiedossa

Taulukko 2: Polynomien generointiin käytettävät heuristiikat.

Jatkokehitys

Vaikka ohjelmassa on toteutettu polynomit sekä linkitettyinä listoina (LinkedListPolynomial) että taulukkomuodossa (ArrayPolynomial), käyttää algoritmi tällä hetkellä ainoastaan implementaatiota LinkedListPolynomial. Tämä siksi, että LinkedListPolynomial-implementaatio on pikaisesti tehdyissä testeissä vaikuttanut nopeammalta. Voisi hyvinkin olla mahdollista kehittää heuristiikkoja jotka valitsevat implementaatioista sen, joka sopii kyseiseen tilanteeseen paremmin.

Rabinin jaottomuusalgoritmin aikavaativuutta olisi mahdollista pienentää käyttämällä taulukkomuotoisia polynomeja ja implementoimalla Fast Fourier Transform-algoritmin kertolaskulle. Myös LinkedListPolynomial-luokan kerto- ja jakolasku on tällä hetkellä toteutettu naiivisti. Niitä olisi mahdollisuus nopeuttaa jonkin verran vaikka niiden aikavaativuuden kertaluokka ei tästä muuttuisikaan.

Lyhyitä polynomeja generoiva sparse-heuristiikka on tällä hetkellä hyvin staattinen. Se laittaa jokaisen kertoimen todennäköisyydellä 0,7 nollaksi. Voisi olla järkevää muuttaa heuristiikka siten, että se generoi nollakertoimia suuremmalla todennäköisyydellä kun haetun polynomin aste kasvaa. Täten generoidut jaottomat polynomit pysyisivät aina melko lyhyinä. Haittapuolena olisi se, että generoitujen polynomien jakauma muuttuisi asteen kasvaessa.

Viitteet

[1] Rabin, M.O.: Probabilistic Algorithms in Finite Fields. SIAM Journal on Computing, 1980, Vol. 9, No. 2 : pp. 273-280. Katso myös http://en.wikipedia.org/wiki/Factorization_of_polynomials_over_finite_fields#Rabin.27s_test_of_irreducibility

[2] Hardy, G. H., Ramanujan, S.: The normal number of prime factors of a number n. Quarterly Journal of Mathematics, 1916, Vol. 48: pp. 76-92.

[3] Gao, S., Panario, D.: Tests and constructions of irreducible polynomials over finite fields. Foundations of Computational Mathematics, 1997, pp. 346-361. http://www.math.clemson.edu/~sgao/papers/GP97a.pdf