Sami Mönkölä

Sääntöpohjaiset tiedonlouhintamenetelmät ohjelmistojen ymmärtämisen tukena

Tietotekniikan pro gradu -tutkielma

15. toukokuuta 2013

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Sami Mönkölä

Yhteystiedot: sakamonk@gmail.com

Ohjaajat: Ilkka Pölönen ja Ville Tirronen

Työn nimi: Sääntöpohjaiset tiedonlouhintamenetelmät ohjelmistojen ymmärtämisen tukena

Title in English: Rule-based data mining methods for software knowledge support

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 60+1

Tiivistelmä: Teknologian nopean kehityksen myötä digitaalisessa muodossa oleva tietomäärä kasvaa kaikkialla. Tietovarastojen koon kasvaessa tarpeellista tietoa tallennettuun tietomäärään nähden on hyvin vähän ja tärkeän informaation löytäminen on haasteellista. Tähän ongelmaan ratkaisuna on tiedonlouhintatekniikat. Tiedonlouhintaa käytettäessä tavoitteena on löytää datajoukosta uusia tuloksia ja näkökohtia tiettyyn kyseessä olevaan ongelmaan. Tutkielmassa keskitytään ohjelmistoaineistojen louhintaan, jonka avulla voidaan saada hyödyllistä informaatiota ohjelmistoprojektin vaiheista ja siinä tapahtuvista virheistä ja niiden ehkäisemisestä.

Avainsanat: pro gradu -tutkielma, tiedonlouhinta, ohjelmistoaineistojen louhinta, assosiaatiosäännöt

Abstract: Rapidly expanding and evolving technology makes digitally stored information volume growing everywhere. When database size grows there is only a little bit of necessary information in relation to the all information saved and finding important information is challenging. Solution to this problem is data mining techniques. The purpose of using data mining is to find new results and viewpoints to a given problem from data sets. In this thesis we concentrate on software mining which helps to find useful information from software project phases and errors happening in those and how to prevent errors.

Keywords: Master's Thesis, data mining, software mining, association rules

Termiluettelo

AiS	Tiedonlouhinta-algoritmi useasti esiintyvien tietuejoukkojen löytämiseen.
Algoritmi	Sovellus tai sen osa matemaattisen ongelman ratkaisemiseen.
API	`Application programming interface`, sovelluskomponentin ohjelmointirajapinta.
Apriori	Tiedonlouhinta-algoritmi assosiaatiosääntöjen löytämiseen.
Assosiaatiosääntö	Todennäköisyysväite tiettyjen tapahtumien esiintymisestä samanaikaisesti samassa tietojoukossa.
DHP	Apriori-algoritmista johdettu tiedonlouhinta-algoritmi.
Eclat	Tiedonlouhinta-algoritmi assosiaatiosääntöjen löytämiseen.
FP-Growth	Tiedonlouhinta-algoritmi assosiaatiosääntöjen löytämiseen.
FP-puu	FP-Growth -algoritmin käyttämä tiedonkäsittelymalli.
Frekvenssi	Tietyn asian esiintymistiheys.
Funktio	Sovelluksen osakokonaisuus, joka on määritetty ohjelmointivaiheessa.
Git	Versionhallintajärjestelmä tiedostojen varastointiin.
GNU GPL	Gnu-hankkeen luoma lisenssi vapaiden ohjemistojen julkaisemiseen, (ks. Free Software Foundation 2007).
IHP	Tiedonlouhinta-algoritmi assosiaatiosääntöjen löytämiseen.
Iteraatio	Tietty työvaihe , jota toistetaan useita kertoja kunnes haluttu tulos saavutetaan.
jEdit	GNU GPL -lisensioitu koodieditori ohjelmoijien käyttöön.
Kandidaattialkiot	Tiedonlouhinta-algoritmien löytämiä tietoalkioita, joiden perusteella assosiaatiosäännöt johdetaan.
Kvantitatiivinen	Määrää koskeva luokittelutapa.
Leveyshaku	Algoritmien tapa käsitellä puumaisessa muodossa olevaa tietojoukkoa.
M-DHP	DHP-algoritmista johdettu tiedonlouhinta-algoritmi.
Minimituki	Tiedonlouhinta-algoritmin käyttämä parametri, joka määrittää

	käytetyn prosentuaalisen arvon joukoille, jotka sisältävät kaikki tietyssä assosiaatiosäännössä käytetyt alkiot.
Parametri	Sovellukselle välitettävä yksittäinen tieto.
Regressio	Tiedonlouhintatekniikka, jolla yhtälö saadaan sisällytettyä tietojoukkoon.
Relaatiotietokanta	Yleisesti käytetty tietokantatekniikka, joka pohjautuu relaatiomalliin.
SETM	Tiedonlouhinta-algoritmi assosiaatiosääntöjen löytämiseen.
SPADE	Tiedonlouhinta-algoritmi assosiaatiosääntöjen löytämiseen.
SQL	Relaatiotietokantojen yhteydessä käytetty kyselykieli.
SVN	Versionhallintajärjestelmä tiedostojen varastointiin.
Tietämyksenhallinta	Tieteen ala, jossa tietoa käsitellään hallittavissa ja johdettavissa olevan käsitteenä.
UML	Standardoitu graafinen mallinnuskieli, joka sisältää erilaisia kaavioita, joilla voidaan kuvata rakennetta, käyttäytymistä ja vuorovaikutusta.
Versionhallinta	Tekniikka, jolla ylläpidetään tiedostoihin tehtyjä muutostietoja.
XML	Sääntöjoukko rakenteisen tiedon esittämiseen käytettyjen tekstiformaattien suunnitteluun.

Kuviot

Taulukot

Sisältö

1 Johdanto

Teknologian nopean kehityksen myötä digitaaliseen muotoon varastoidun tiedon määrä kasvaa todella nopeaa vauhtia. Vain murto-osaa tästä tiedosta tullaan koskaan käyttämään, koska tietovarastot ovat yksinkertaisesti liian suuria käsiteltäviksi (Pyle 1999). Suurin syy tähän on se, että usein tietoa kerätessä keskitytään vain tehokkaaseen tiedon varastointiin eikä ajatella miten tietoa tullaan lopulta käyttämään ja analysoimaan. (Kantardzic 2002). Tällaisten tietovarastojen analysointia helpottamaan on kehitetty menetelmä, tietämyksenhallinta, jonka historia pohjautuu tilastotieteen ja koneoppimisen menetelmiin (Mitchell 1999). Tietämyksenhallintaa voidaan käyttää eri tieteen alojen tutkimiseen ja sitä onkin käytetty supermarkettien markkinointianalysoinnista aina rikostutkintaan asti. Tietämyksenhallintaprosessin tavoitteena on löytää tietovarastosta uusia tuloksia ja näkökohtia kyseessä olevaan ongelmaan ja ilmoittaa näistä kerätty yhteenveto ihmiselle helposti ymmärrettävässä muodossa.

Tietämyksenhallinta on iteratiivinen prosessi ja tiedonlouhinta on yksi sen osa-alueista. Tiedonlouhintamenetelmät voidaan jaotella tutkivaan data-analyysiin, kuvaavaan mallintamiseen, hahmojen ja sääntöjen etsimiseen sekä hakuun sisällön perusteella (Hand, Smyth ja Mannila 2001). Tässä tutkielmassa keskitytään hahmojen ja sääntöjen etsimiseen. Tällöin käytetään erilaisia tiedonlouhinta-algoritmeja assosiaatiosääntöjen löytämiseksi. Assosiaatiosäännöt ennustavat, mitä mahdollisesti tulee tapahtumaan tiettyjen ennalta tiedettyjen tietojen perusteella (Witten ja Frank 2000). Tällainen yksinkertainen sääntö voisi olla esimerkiksi `Jos asuu opiskelija-asunnossa, on opiskelija`. Sääntöön kuuluu siis ensin mainittu ehto ja jäljempänä mainittu seuraus. Löydetty sääntö on yksisuuntainen, eikä sitä voi luotettavasti käyttää toiseen suuntaan, esimerkiksi niin, että `Jos on opiskelija, asuu opiskelija-asunnossa`. Todennäköisyyssääntö muuttaa perussääntöä siten, että jos ehto on tosi, tällöin seuraus on tosi tietyllä todennäköisyydellä (Kantardzic 2002). Assosiaatiosääntöjen etsimisen suurin haaste on löytää suurista tietomassoista säännöt, jotka täyttävät annetut tarkkuuskriteerit (Juhola 2006). Assosiaatiosääntöjen löytäminen tapahtuu muodostamalla tietovaraston, `datajoukon`, alkioista usein esiintyviä joukkoja ja karsimalla ne yhdistelmät, jotka eivät täytä annettuja kriteeriarvoja (Han ja Kamber 2000).

Tutkielman empiirisessä osuudessa sovelletaan tarkoitukseen sopivaa sääntöpohjaista tiedonlouhinta-algoritmia jEdit-projektin versionhallintavarastoon. jEdit on ohjelmointikäyttöön tarkoitettu tekstinmuokkausohjelmisto. jEdit on avoimen lähdekoodin ohjelmisto (engl. `open source software`) ja sen kehittäminen alkoi vuonna 1998. Nykyään ohjelmiston ja siihen kuuluvien komponenttien (engl. `plugins`) kehityksestä vastaa maailmanlaajuinen kehitysyhteisö. Vuoden 2007 aikana projektin versionhallinnan kautta on päivitetty noin 8000 tiedostoa. Manuaalinen tiedon analysointi tämän kokoisesta tietovarastosta ei olisi enää kannattavaa eikä tehokasta. Sääntöpohjaisella tiedonlouhintamenetelmillä jEdit-projektin versionhallintavarastosta voitaisiin saada selville esimerkiksi se, mitkä komponentit vaikuttavat toisiinsa eniten kehitysnäkökulmasta. Tällä tiedolla voidaan muun muassa pienentää vaadittavien korjausten määrää, mikäli tietyn komponentin päivittäminen luo virheitä muiden komponenttien toimivuudessa.

Tutkielma on jaettu seuraavasti: luvussa 2 määritellään tiedonlouhinta, käydään läpi sen tarvetta, menetelmiä ja lähestymistapoja. Luvussa 3 käsitellään eri tiedonlouhinta-algoritmeja, assosiaatiosääntöjä ja sarjallisia hahmoja. Luvussa 4 kerrotaan tarkemmin tutkielman empiirisessä osiossa käytetystä jEdit-ohjelmistosta. Luvussa 5 tarkastellaan empiirisen osan soveltamista jo olemassa olevaan ohjelmistoon. Luvussa 6 kootaan johtopäätökset yhteen.

2 Tiedonlouhinta

Tiedonlouhinta on osa suurempaa kokonaisuutta, *tietämyksenhallintaa* (engl. *knowledge discovery from data*). Tietämyksenhallinnan perusteet ovat tekoälytutkimuksessa. Tietämyksenhallinta on iteratiivinen prosessi ja käsittää useita eri vaiheita, kirjallisuudesta riippuen vaiheiden lukumäärä vaihtelee hieman. Tarkastellaan tässä seitsemänvaiheista tietämyksenhallintaprosessia (Han ja Kamber 2000).

Tietämyksenhallintaprosessin ensimmäinen vaihe on *tiedon puhdistaminen*. Tiedon puhdistamista (engl. *data cleaning)* tarvitaan, koska käsiteltävä data on usein puutteellista, virheellistä tai epäyhtenäistä. Puutteellinen data voi johtua siitä, että tietoa ei ole syötetty tai tietoa syötettäessä on tullut esimerkiksi ohjelmistosta johtuva virhe. Virheellinen data voi johtua yksinkertaisesta kirjoitusvirheestä tai virheistä tiedonsiirrossa. Epäyhtenäisellä datalla tarkoitetaan samaa objektia, jonka kirjoitusasu vaihtelee tai esimerkiksi eri datalähteistä peräisin olevia erilaisia arvosteluasteikkoja (Han 2005). Tiedon puhdistamisella on suuri merkitys lopputulosta ajatellen. Jos tiedon puhdistaminen jätetään kokonaan tekemättä, kärsii siitä koko tietämyksenhallintaprosessin tehokkuus ja laatu. Toista vaihetta kutsutaan *tiedon yhdistämiseksi*. Tiedon yhdistäminen (engl. *data integration*) tarkoittaa useammassa eri lähteessä, esimerkiksi tiedostoissa tai tietokannoissa, olevan datan yhdistämistä yhdeksi kokonaisuudeksi. Kolmas vaihe on *tiedon valitsemista*. Tiedon valitsemisessa (engl. *data selection*) valitaan yhdistetystä tiedosta kyseistä analyysia varten tarvittavat tiedot. Neljäs vaihe on *tiedon muuntaminen*. Tiedon muuntaminen (engl. *data transformation*) käsittää käsiteltävän tiedon muuttamisen samaan yksikkömuotoon. Esimerkkinä kaikkien valuuttasummien muuttaminen oikeaan arvoon käyttäen kaikille samaa valuuttayksikköä. Viidentenä vaiheena on tiedonlouhinta (engl. `data mining`). Tiedonlouhinnalla tarkoitetaan uusien mallien ja sääntöjen löytämistä suuresta datajoukosta (Wur ja Leu 1999). Tiedonlouhintatekniikoita ovat assosiaatiosääntöjen johtaminen, luokittelu, regressio ja ryhmittelyanalyysi (Han ja Kamber 2000). Kuudenteen vaiheeseen kuuluu *mallien arvioiminen*. Mallien arvioinnissa (engl. *pattern evaluation*) seulotaan mielenkiintoiset assosiaatiosäännöt erilleen muista assosiaatiosäännöistä. Seulontaan käytetään erilaisia mittareita, joista tarkemmin luvussa 2.2. Seitsemäs vaihe on *tiedon esittämistä*. Tiedon esittämisessä (engl. *data reporting*) tulokset

esitetään ihmiselle helposti ymmärrettävässä muodossa. Tulosten esitystavat voivat poiketa riippuen siitä mihin käyttötarkoitukseen ja kenelle tulokset ovat luotu. Esitystapoina voidaan käyttää erilaisia listoja, selventäviä graafisia esityksiä tai yhteenvetotauluja (Elmasri ja Navathe 1999).

Tässä tutkielmassa keskitytään sääntöpohjaisiin tiedonlouhintamenetelmiin ja niiden käyttöön ohjelmistoissa. Tiedonlouhinnan perusajatuksena on etsiä suuresta, olemassa olevasta tietomäärästä, *datajoukosta*, erilaisia tietojen välisiä suhteita ja kerätä näiden pohjalta yhteenveto, joka on uutta ja hyödyllistä sekä ihmiselle selkeässä muodossa olevaa tietoa. Yleensä datajoukko on valmiina olevaa aineistoa, joka on kerätty muussa kuin tiedonlouhintatarkoituksessa. Täten tiedonlouhinnan tavoitteet eivät ota kantaa tiedon keräämiseen. Useasti laajat datajoukot säilytetään prosessin aikana levyillä tai nauhoilla, koska ne eivät kokonaisuudessaan mahdu keskusmuistiin (Hand, Smyth ja Mannila 2001).

Tiedonlouhinnan tuloksia kutsutaan globaaleiksi *malleiksi* ja lokaaleiksi *hahmoiksi*. Hahmolla tarkoitetaan rakennetta tai sääntöä, joka kuvaa kohdealuetta. Malli taas on aineistossa usein toistuva hahmo. Nämä ovat tiedonlouhinnan avulla saatuja ominaisuuksia ja yhteenvetoja. Malleja ja hahmoja ovat esimerkiksi *ryppäät*, *lineaariset yhtälöt*, *graafit* ja *puurakenteet*.

2.1 Menetelmiä

Tiedonlouhintatoimintojen päätyyppejä ovat *tutkiva data-analyysi* (engl. *exploratory data analysis*), *kuvaava mallintaminen* (engl. *descriptive modeling*), *ennustava mallintaminen* (engl. *predictive modeling*), *hahmojen ja sääntöjen etsiminen* (engl. *finding patterns and rules*) ja *haku sisällön perusteella* (engl. *retrieval by content*). Tutkivassa data-analyysissä ei ole tarkkaa suunnitelmaa siitä, mitä etsitään. Tämä tehtävä onkin usein luonteeltaan vuorovaikutteista ja visuaalista. Suppeiden datajoukkojen esitystä varten on monipuolisia graafisia esitystapoja, mutta datajoukkojen suurentuessa myös visualisointi käy hankalammaksi. Kuvaavassa mallintamisessa kuvataan koko data. Esimerkkeinä todennäköisyysjakaumamallit ja muuttujien välisten suhteiden kuvausmallit. Ennustavassa mallintamisessa käytetään luokitusta ja regressiolaskentaa. Tässä pyritään muodostamaan malli, jonka avulla pystytään

ennustamaan yhden muuttujan arvo, kun muiden muuttujien arvo tiedetään. Ennustaminen sopii esimerkiksi osakemarkkinoiden tutkimiseen. Ennustettava muuttuja on luokituksessa luokkatyyppinen ja regressiolaskennassa kvantitatiivinen. Hahmojen ja sääntöjen etsimisessä pyritään havaitsemaan tiettyjä hahmoja ja sääntöjä, kuten nimikin jo ilmaisee. Vaikeinta tässä yhteydessä on poikkeavien havaintojen keksiminen, eli päättää missä kulkee epätavallisen ja tavallisen käyttäytymisen raja. (Hand, Smyth ja Mannila 2001) Hahmojen ja sääntöjen etsimistä on käytetty muun muassa etsittäessä matkapuhelinten laittomia käyttäjiä. Hakua sisällön perusteella käytetään, kun käyttäjällä on kiinnostava hahmo. Tämän hahmon avulla etsitään datajoukosta muita samanlaisia hahmoja. Tehtävä on erityisen suosittu teksti- ja kuvatietojoukkoja käsiteltäessä. Tekstitietojoukkoja käsiteltäessä hahmo koostuu avainsanoista ja käyttäjä etsii samanlaisia hahmoja suuresta dokumenttijoukosta.

2.2 Sääntöpohjaiset menetelmät

Sääntöpohjaisessa tiedonlouhinnassa käytetään *assosiaatiosääntöjä* (engl. *association rules*). Sääntöjen käyttäminen tapahtuu tiedonhallintaprosessin viidennessä vaiheessa, tiedonlouhinnassa. Näitä sääntöjä käytetään ennen löytämättömien yhtäläisyyksien löytämiseen ja esittämiseen datajoukosta (Amir et al. 2005). Perinteisin ongelmatyyppi, joka assosiaatiosäännöillä voidaan ratkaista, on ollut ostoskoriongelma. Tässä ongelmassa yritetään selvittää, mitä tavaroita ostetaan usein yhdessä ja sen perusteella edistää tuotteiden myyntiä muun muassa muuttamalla niiden sijaintia kaupassa. Assosiaatiosäännöille voidaan asettaa pienin todennäköisyys- ja frekvenssiarvo. Jos sääntöjen tulokset eivät täytä näitä asetuksia, niitä ei pidetä enää mielenkiintoisina ja ne hylätään. (Mooney ja Bunescu 2005)

Assosiaatiosäännöt löydetään etsimällä datajoukosta yhdessä esiintyviä tekijöitä. Assosiaatioanalyysin tulos annetaan *todennäköisyys-frekvenssi*-parina (engl. *support-confidence*). Esimerkkinä voidaan miettiä kurssi-ilmoittautumista. Jos opiskelija on ilmoittautunut tietoliikenteen ja UNIX-ohjelmoinnin kurssille, niin hän mahdollisesti ilmoittautuu myös C-ohjelmoinnin kurssille. Tämä sääntö voitaisiin esittää tuloksineen seuraavasti:

$$kurssille(oppilas, tietoliikenne) \cup kurssille(oppilas, UNIX - ohjelmointi)$$

$$\Rightarrow kurssille(oppilas, C-ohjelmointi)\ [56\%, 12\%]$$

, jolloin tapahtumalle on 56 %:n todennäköisyys ja 12 %:n frekvenssi. Frekvenssi osoittaa sen, kuinka monen oppilaan kohdalla sääntö on toteutunut koko otosjoukkoa tarkasteltaessa.

Assosiaatiosääntöjen johtaminen tapahtuu kahdessa vaiheessa (Han ja Kamber 2000). Ensimmäiseen vaiheeseen kuuluu usein esiintyvien alkiojoukkojen etsiminen jollakin algoritmilla. Toisessa vaiheessa näistä löydetyistä alkiojoukoista johdetaan assosiaatiosääntöjä. Saatujen assosiaatiosääntöjen määrä voi olla hyvinkin suuri eikä niitä manuaalisesti pystytä karsimaan, tämän vuoksi käytössä on minimiarvot todennäköisyydelle ja frekvenssille. Näitä käyttämällä saadaan vain kulloinkin tarvittavat assosiaatiosäännöt käyttöön, koska arvojen alapuolelle jääneet säännöt karsiutuvat automaattisesti pois. Assosiaatiosääntöjä etsivissä algoritmeissa on käytössä minimituki-parametri (engl. *minimum support threshold*). Parametrin avulla löydetään usein esiintyvät alkiojoukot. Minimituki-parametri voi olla algoritmeissa yhtenäinen tai epäyhtenäinen. Parametrin arvo määritellään todennäköisyyden ja frekvenssin perusteella. Menetelmän hyötypuolena on myös sovellusriippumattomuus.

Liu & ym. määrittelevät *harvinaisten alkioiden ongelman* (engl. *rare item problem*). Kyseisen ongelman mukaan sääntöihin ei saada mukaan harvoin esiintyviä alkioita, mikäli käytettävän algoritmin minimituki-parametrin arvo asetetaan liian korkeaksi. Harvoin esiintyvät alkiot saadaan mukaan minimituki-parametrin arvoa laskemalla. Tällöin seurauksena on sääntöjen räjähdysmäinen kasvu, sillä usein esiintyvät alkiojoukot yhdistyvät assosiaatiosäännöiksi kaikilla mahdollisilla tavoilla (Liu, Hsu ja Ma 1999). Teksti- ja tapahtumatietokantojen käsittelyä verrattaessa parametrissa käytetään erilaisia arvoja. Tekstitietokannoissa ollaan kiinnostuneempia niistä sääntöihin mukaan saaduista alkioista, jotka ovat lähellä minimituki-parametrin arvoa. Parametrin arvosta kauemmaksi jääneet alkiot ovat tällöin esimerkiksi artikkeleita tai prepositioita.

Minimituki-parametri kuuluu *mittareihin*, joita käytetään löydettyjen mallien analysointiin. Mittarit voidaan jakaa *objektiivisiin* ja *subjektiivisiin mittareihin* (Liu et al. 2000). Objektiiviset mittarit perustuvat käytettävien algoritmien rakenteeseen, kun taas subjektiivisiin mittareihin liittyy käyttäjän subjektiivisuus assosiaatiosääntöjen tulkinnassa. Objektiivisissa mittareissa on käyttäjän kontrolloitavissa oleva kynnysarvo. Objektiivisia mittareita on kolme

kappaletta: *yksinkertaisuus* (engl. *simplicity*), *hyödyllisyys* (engl. *utility*) ja *varmuus* (engl. *certainty*). Yksinkertaisuus-mittarissa voidaan käyttää käyttäjän asettamaa kynnysarvoa, jolloin tavoitteena on saada assosiaatiosäännöistä yksinkertaisempia. Tämä ominaisuus voidaan lisätä algoritmiin funktiona, jossa algoritmin suoritusaikana lasketaan ylittääkö sääntö käyttäjän antaman kynnysarvon, jolloin kyseinen sääntö on liian monimutkainen ja se voidaan pudottaa pois jatkokäsittelystä. Pienempi yksinkertaisuus-arvo tarkoittaa siis lyhyempää ja sitä kautta helpommin ymmärrettävää sääntöä. Sääntöjä yksinkertaistaessa niiden lukumäärä kuitenkin kasvaa. Hyödyllisyys-mittarissa käytetään minimituki-parametriä. Varmuus-mittarissa voidaan käyttää tiettyä *uskottavuus* (engl. *confidence*)-kynnysarvoa, joka assosiaatiosääntöjen on toteutettava. Tällä tarkoitetaan sitä, kuinka monessa tapauksessa assosiaatiosäännön implikaation vasemmasta puolesta seuraa implikaation oikea puoli. Kynnysarvo on käyttäjän määriteltävissä. Tällä saadaan karsittua ne säännöt, jotka eivät ole tilastollisesti merkittäviä. (Han ja Kamber 2000) Uskottavuus saadaan esille kaavalla

$$conf(X,Y) = \frac{supp(X \cup Y)}{supp(X)} \tag{2.1}$$

, jossa siis verrataan alkiojoukkojen X ja Y esiintymistä pelkän X:n esiintymiseen datajoukossa (Rantzau 1997). $supp(X)$ tarkoittaa alkion X tukiarvoa tietyssä tietojoukossa. Jos säännön uskottavuusarvo on 100, sitä sanotaan *tarkaksi* (engl. *exact*) (Han ja Kamber 2000). Jos säännöllä on alhainen tuki, korkeasta uskottavuudesta ei ole hyötyä (Ullman, Garcia-Molina ja Widom 2001). Tällöin säännön esiintymisaste on niin pieni, että sääntöä ei kannata käyttää. Korkean uskottavuuden ja tuen omaavia assosiaatiosääntöjä kutsutaan *vahvoiksi säännöiksi* (engl. *strong rules*) (Chen, Han ja Yu 1996). Subjektiiviset mittarit muuttavat tuloksia enemmän käyttäjän tietojen, taitojen ja kokemusten perusteella, kuin objektiiviset mittarit. Subjektiivisia mittareita on kahdenlaisia: *yllätyksellisyys* (engl. *unexpectedness*) ja *toiminnan sallittavuus* (engl. *actionability*). Yllätyksellisyys tarkoittaa sitä, että löydetty sääntö on mielenkiintoinen, mikäli se on ristiriidassa käyttäjän aikaisemman tiedon kanssa. Toiminnan sallittavuus taas tarkoittaa, että löydetyt säännöt aiheuttavat käyttäjässä toimenpiteitä. Käyttäjä voi esimerkiksi käyttää löydettyjä sääntöjä hyödyksi työssään ja perustaa ratkaisujaan uusille, löydetyille säännöille. On kehitetty järjestelmiä, jotka huomioivat käyttäjän aikaisemmat kokemukset ja tiedot. Tällaisista järjestelmistä on hyötyä, koska kaikki käyttä-

jät ovat erilaisia ja reagoivat sääntöihin eri tavoilla. Näillä järjestelmillä pystytään etsimään mielenkiintoisia sääntöjä juuri tietty käyttäjä huomioiden. Yksi tällainen järjestelmä on *IAS-järjestelmä* (engl. *Interestingness Analysis System*). IAS-järjestelmä hyödyntää käyttäjän antamaa informaatiota jokaisella iteraatiokerralla. Käyttäjän antamien tietojen mukaan säännöt jaetaan neljään luokkaan. Ensimmäiseen luokkaan sisältyvät sellaiset säännöt, jotka toteuttavat käyttäjän odotukset. Muissa luokissa olevat säännöt ovat yllätyksellisiä. Käyttäjällä on mahdollisuus poistaa ja merkata mielenkiintoisiksi löydettyjä sääntöjä. IAS on ottanut vaikutteita Apriori-algoritmista, josta enemmän luvussa 3.1. (Liu et al. 2000)

Assosiaatiosäännöt voidaan jakaa kolmeen eri luokkaan, joita ovat *tiedon tyyppi*, *tiedon ulottuvuus* ja *tiedon hierarkia* (Han ja Kamber 2000).Yksinkertaisin muoto tiedon tyypistä on totuusarvon omaava assosiaatiosääntö. Tällöin siis datajoukon käsiteltävillä alkioilla ja niistä syntyvillä säännöillä voi olla vain yksi totuusarvo, joko *tosi* tai *epätosi*. Monimutkaisempia tyyppejä syntyy, kun datajoukossa on mukana kvantitaviisia arvoja ja pitää tutkia datajoukkoa esimerkiksi tietyn ikäjakauman perusteella (Fukuda et al. 1996). Kvantitatiivisilla arvoilla kuvataan alkioiden numeerista tietoa. Tällöin yksinkertaisin, mutta myös hitain tapa olisi käsitellä datajoukon alkiot yksi ikäryhmä kerrallaan ja lopuksi kerätä näistä syntyneet säännöt yhteen. Tehokkaampi tapa on kuitenkin osioida kvantitatiiviset arvot sopiviin osajoukkoihin (Srikant ja Agrawal 1996). Tästä käymme esimerkkinä läpi sääntöjen käsittelemisen henkilötietoja sisältävän datajoukon avulla. Käsiteltävä datajoukko kuvataan taulukossa 1.

Tietue ID	**Ikä**	**Naimisissa**	**Autojen lkm**
100	23	ei	1
200	25	kyllä	1
300	29	ei	0
400	34	kyllä	2
500	38	kyllä	2

Taulukko 1. Datajoukko

Annetaan minimituki-parametrin arvoksi 40. Sopiva arvo kyseiselle parametrille riippuu aina käytetystä lähtöaineistosta ja käyttötilanteesta. Yksi menetelmä sopivan arvon löytämi-

seksi on valita ensin suuri arvo ja pienentää sitä vähitellen, kunnes aineistosta on löydetty riittävä määrä sääntöjä. Nyt jaetaan esiintyneet iät sopiviin ryhmiin tasaisesti jaoteltuna, jolloin saadaan ikäryhmätaulukko 2. Ikäjaon jälkeen saatu datajoukko on taulukon 3 mukainen. Seuraava vaihe on numeroida (kokonaislukuarvoin) alkiot kaikissa ryhmissä, jotka sisältävät kvantitatiivisia arvoja. Esimerkissä tällaisia ryhmiä ovat *ikä* ja *naimisissa*. Taulukossa 4 datajoukko numeroituna. Tuloksena taulukossa 5 esitetty osa useasti esiintyvistä tietuejoukoista (engl. *frequent itemsets*) ja taulukossa 6 osa löydetyistä säännöistä.

Ikäjakso
20..24
25..29
30..34
35..39

Taulukko 2. Ikäryhmätaulukko

Tietue ID	**Ikä**	**Naimisissa**	**Autojen lkm**
100	20..24	ei	1
200	25..29	kyllä	1
300	25..29	ei	0
400	30..34	kyllä	2
500	35..39	kyllä	2

Taulukko 3. Datajoukko ikäryhmittelyn jälkeen

Yksinkertaisin assosiaatiosääntö tiedon ulottuvuuden kannalta on *yksiulotteinen assosiaatiosääntö*. Tällöin tarkastellaan vain yhtä sääntöä, esimerkiksi jos henkilö ostaa kahvia, voidaan tarkastella ostaako hän myös suodatinpusseja. Tällöin tutkittava ulottuvuus on ostaminen.

Tietue ID	Ikä	Naimisissa	Autojen lkm
100	1	2	1
200	2	1	1
300	2	2	0
400	3	1	2
500	4	1	2

Taulukko 4. Datajoukko numeroituna

Tietuejoukko	Todennäköisyys
{ (Ikä: 20..29) }	3
{ (Ikä: 30..39) }	2
{ (Naimisissa: kyllä) }	3
{ (Naimisissa: ei) }	2
{ (Autojen lkm: 0..1) }	3
{ (Autojen lkm: 2) }	2
{ (Ikä: 30..39), (Naimissa: kyllä) }	2

Taulukko 5. Usein esiintyviä tietuejoukkoja

Moniulotteisella assosiaatiosäännöllä tarkoitetaan sääntöä, jossa tutkitaan useampaa ulottuvuutta samanaikaisesti. Tällöin yhdistellään tarkasteltavaksi useampia alkioita, jolloin säännöt tulevat käyttäjälle informatiivisemmiksi. (Pinto 2001) Esimerkki kolmeulotteisesta assosiaatiosäännöstä, jossa tarkasteltavina ulottuvuuksina ovat ikä, sukupuoli ja ostotapahtuma:

$$ikä(X, 20..25) \cup sukupuoli(X, mies) \cup ostaa(X, kahvia) \Rightarrow ostaa(X, suodatinpusseja)$$

Säännön mukaan 20-25-vuotiaat miehet, jotka ostavat kahvia, ostavat samalla myös suodatinpusseja. Relaatiotietokannat ovat hyvä lähde moniulotteisten assosiaatiosääntöjen löytämiseen. Tapahtumaperäiseen tietokantaan verrattuna relaatiotietokannasta saadaan esille

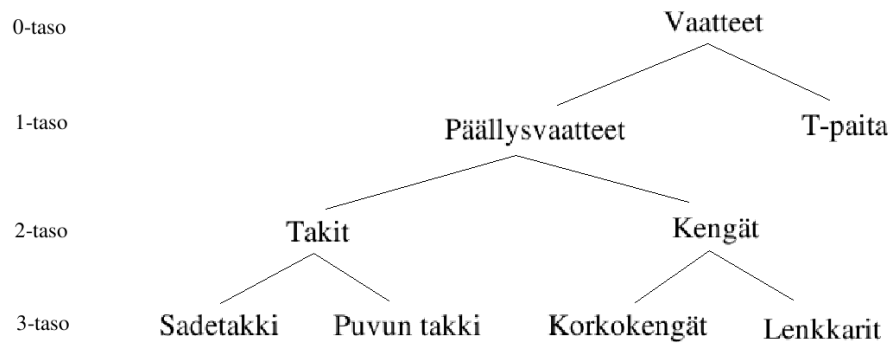
Sääntö	**Todennäköisyys**	**Frekvenssi**
(Ikä: 30..39) $\cup$ (Naimisissa: kyllä) $\Rightarrow$ (Autojen lkm: 2)	40 %	100 %
(Ikä: 20..29) $\Rightarrow$ (Autojen lkm: 0..1)	60 %	66.6 %

Taulukko 6. Löydettyjä assosiaatiosääntöjä

tapahtumaan liittyviä attribuutteja, kuten edellisessä esimerkissä käytetyt ikä ja sukupuoli. Edellisen esimerkin sääntöä voidaan kutsua myös *profiili-assosiaatiosäännöksi* (engl. *profile association rule*). Tällaisessa assosiaatiosäännössä implikaation vasemmalla puolella on asiakasprofiilin tiedot ja implikaation oikealla puolella oletettu käyttäytyminen (Dunham et al. 2000).

Yksi- ja moniulotteiset assosiaatiosäännöt voidaan jakaa edelleen kolmeen eri luokkaan. Kaikki yksiulotteiset säännöt kuuluvat *ulottuvuuden sisäisiin assosiaatiosääntöihin* (engl. *intradimension association rule*). Tällöin säännön ulottuvuus siis toistuu assosiaatiosäännön implikaation molemmilla puolilla. *Ulottuvuuksien väliset assosiaatiosäännöt* (engl. *interdimension association rule*) kuuluvat moniulotteisiin assosiaatiosääntöihin. Tällöin jokaista ulottuvuutta käytetään säännössä vain kerran. Kolmannen luokan sääntöjä kutsutaan *ulottuvuuksiltaan hybrideiksi assosiaatiosäännöiksi* (engl. *hybrid-dimension association rule*). Tällöin assosiaatiosäännössä käytetään vähintään yhtä ulottuvuutta useammin kuin kerran. (Han ja Kamber 2000)

Assosiaatiosäännöissä voidaan huomioida alkioiden määrittelyjen tasot, mikäli alkioiden luokitteluna käytetään *monitasoista käsitehierarkiaa* (engl. *concept hierarchy*). Käsitehierarkia voidaan esitellä puumaisena rakenteena. Tällöin hierakian tasot numeroidaan ylhäältä alaspäin ja ylin taso saa arvon 0. Tämä nolla-taso on yleisluontoinen taso ja siihen kuuluvat kaikki alkiot. Hierarkian tasoja alaspäin mentäessä alkiot jakautuvat puumaisesti eri lohkoihin ja niiden ryhmämäärittely on tarkempi kuin edellisen tason. (Han ja Kamber 2000) Alimmaisin taso on siis kaikista tarkin. Vaatteet voidaan kuvata hierarkisesti ominaisuuksien ja tuotemerkin mukaan (Yen ja Chen 2001), kuten kuviossa 1. Yksiulotteiset assosiaatiosäännöt sijoittuvat hierarkiassa aina yhdelle tasolle, jolloin tasoja ei huomioida.



Kuvio 1. Esimerkki vaatteiden hierakisesta kuvauksesta.

Tietokannan ominaisuuksista riippuen datajoukon muodostava data voi esiintyä tietokannassa suhteellisen usein tai voi myös esiintyä suhteellisen harvoin (Yun et al. 2003). Datajoukon alkioiden esiintymistodennäköisyys vaihtelee ja tämän vuoksi myös alkioiden todennäköisyys yltää assosiaatiosääntöihin vaihtelee (Wang, He ja Han 2003). Esimerkiksi ostostapahtuman datajoukossa maito ja leipä esiintyvät huomattavasti useammin kuin esimerkiksi juusto ja piimä. Tällainen harvoin esiintyvä data voi kuitenkin olla lopputulosta ajatellen merkittävä. Tiedon harvinaisuus antaa oikeampaa informaatiota käyttäjälle (Cohen et al. 2001).

Assosiaatiosäännöt sopivat erittäin hyvin ostostapahtuman tarkasteluun, missä haetaan kahden tuotteen välisiä yhtäläisyyksiä, eli tietoa ostetaanko jokin tuote usein toisen tuotteen kanssa. Säännöt ovat kuitenkin saaneet myös kritiikkiä osakseen, sillä ne eivät ole kaiken kattavia sääntöjä. (Silverstein, Brin ja Motwani 1998) Esimerkiksi assosiaatio

$$Hehkulamput \Rightarrow Kissanhiekka$$

ei pysty kertomaan sitä yhtäläisyyttä, että jos henkilö ostaa hehkulamppuja, hän yleensä ei osta samalla kertaa kissanhiekkaa.

2.3 Ohjelmistoaineistojen louhinta

Yhteiskunta on nykyään hyvin riippuvainen erilaisista ohjelmistoista. Riippuvuus kattaa kaikki sektorit mukaan lukien hallituksen, teollisuuden ja yksityisen sektorin. Tästä syystä ohjel-

mistojen kehittämiseen ja ylläpitoon tulisi kiinnittää huomiota. Ohjelmistojärjestelmät kuitenkin usein kärsivät ikääntymisestä ja sen mukanaan tuomista ongelmista, kun niitä yritetään mukauttaa muuttuviin vaatimuksiin. Tämä johtuu suureksi osaksi siitä, että perinteisessä ohjelmistokehityksessä ohjelmistojen ylläpitoa ja mukauttamista ei vielä pidetä suuressa arvossa. (Mens et al. 2005) Ohjelmistokehityksen onnistuminen ei riipukaan pelkästään kustannuksista ja aikataulusta vaan näiden lisäksi myös laadusta (Song et al. 2006). Ainoa tapa välttää ohjelmistojen ikääntymisen haittavaikutukset on asettaa muutokset ohjelmistokehitysprosessin keskipisteeseen. Mikäli ohjelmiston muutokseen ja kehittymiseen ei panosteta täsmällisesti ja välittömästi, tuloksena on liian monimutkainen ja epäluotettava ohjelmisto. Tällainen ohjelmistojen negatiivinen vaikutus on vain kasvamassa teknologia- ja yritysinnovaatioiden, lainsäädäntöjen ja jatkuvan kansainvälistymisen myötä. Jotta tilanteeseen saataisiin parannusta aikaan, on ohjelmistokehityksessä unohdettava rajoittunut näkökulma ja edettävä sen ulkopuolelle painottaen parempaa ja entistä enemmän tukea ohjelmistojen mukauttamiseen ja evoluutioon. (Mens et al. 2005)

Monet muutostyöt ohjelmistojärjestelmissä vaativat ohjelmistokehittäjiltä usean eri kohdan muuttamista järjestelmän lähdekoodissa. Kehittäjä voi löytääkseen olennaiset kohdat lähdekoodista tiettyä tehtävää varten käyttää työkalua joka staattisesti tai dynaamisesti analysoi lähdekoodin eri osien väliset riippuvuudet. Tällaiset työkalut voivat auttaa kehittäjää löytämään mielenkiintoiset osat lähdekoodista, mutta ne eivät pysty löytämään aina kaikkia muutokselle alttiita lähdekoodin osia. (Ying et al. 2004)

Nykyään lähes kaikki ohjelmistoevoluutiota tukevat työkalut ovat kohdistettu lähdekoodin käsittelyyn. Suunnittelu- ja mallinnusvaiheet, esimerkiksi UML-työkalut, tarjoavat useimmiten paljon vähemmän tukea ohjelmistojen evoluution hallintaan. Tämä voidaan yleistää siten, että ohjelmistojen evoluutioon kohdistetut tekniikat tulisi nostaa korkeammalle abstraktion tasolle. Tällöin siihen voitaisiin sijoittaa tällä hetkellä olevan ohjelmistojen evoluution lisäksi analyysi- ja suunnitelumallit, ohjelmistoarkkitehtuurit, vaatimusmäärittelyt ja niin edelleen. (Mens et al. 2005)

Ohjelmistojen mukauttamiseen ja evoluutioon liittyvät tukitoimet tulee osoittaa useammalla tutkimus- ja kehitystoimen tasolla. Siinä vaaditaan ensinnäkin formalismin ja teorioiden perustavaa tutkimusta, jonka avulla voidaan analysoida, ymmärtää, hallita ja säädellä ohjel-

mistomuutoksia. Myös mallien, kielten, työkalujen, metodien, tekniikoiden ja heurestiikan kehittäminen kuuluu näihin tukitoimiin, jotta voidaan tarjota täsmällistä tukea ohjelmistomuutoksiin. Sekä lisää validointia ja tapaustutkimuksia suurten, pitkään käytössä olleiden ja monimutkaisten teollisten ohjelmistojärjestelmien osalta. (Mens et al. 2005)

Ohjelmistoihin jäljelle jäävien vikojen tai heikkouksien määrän ennustamista voidaan pitää yhtenä ohjelmiston laatutekijänä. Ohjelmistokehittäjä voi käyttää sitä mittarina, jonka avulla hän voi kontrolloida ohjelmiston kehitysprosessia ja esimerkiksi arvioida ohjelmiston laatua ja päättää voidaanko kyseinen ohjelman osa päästää seuraavaan kehitysvaiheeseen vai pitääkö se ottaa vielä lähempään tarkasteluun. Voidaan myös sanoa, että kehityksen aikana löytyneet heikkoudet ovat merkki heikkouksista koko kehitysprosessissa. Tällöin voidaan löytää erilaisia assosiaatioita ohjelmiston heikkouksien välillä. Jos heikkouksista kirjataan tietoa, kuten sen tyyppi, lähde, missä vaiheessa kehitysprosessia se ilmeni, missä ohjelmiston osassa se ilmeni ja niin edelleen, voidaan löytää kehitysprosessissa ilmenevä ongelma ja toteuttaa sen korjaava toimenpide. Ja täten kehitysprosessin toimintaa saadaan kehitettyä. Heikkouksien korjaamisen osalta kahden asian tietäminen on oleellista, ensinnäkin mitkä heikkoudet voivat esiintyä samalla kertaa tietyn annetun heikkouden tai niiden joukon kanssa, ja toiseksi kuinka paljon resursseja kyseisen heikkouden tai niiden joukon korjaaminen vie. Nämä voidaan selvittää tiedonlouhinnan ja assosiaatiosääntöjen avulla. (Song et al. 2006)

Suuret ohjelmistoprojektit tarkoittavat useimmiten monien kymmenien ellei jopa satojen ohjelmistokehittäjien työskentelyä ohjelmiston kehityksen parissa. Tämä taas tarkoittaa suurta määrää lähdekoodia ja mahdollisesti hyvin monia pieniä muutoksia siihen. Tämä voi tehdä ohjelmistosta hyvin monimutkaisen, vaikka versionhallinta olisikin käytössä. Tavalliset ohjelmistovirheet, kuten virheet käyttöjärjestelmän ajureissa, turvallisuusvirheet tai virheet luotettavuudeltaan kriittisissä järjestelmissä, voidaan löytää analysointityökaluja käyttäen ja ne korjataan yleensä nopeasti niiden havaitsemisen jälkeen. Monet muut virheet kuitenkin ovat ominaisia yksittäiselle sovellukselle tai sovellusalustalle. Tällaisissa tapauksissa sovellukselle yksilöityjen ohjelmointisääntöjen rikkominen on syynä suurelle määrälle virheitä. Näitä kutsutaan virhemalleiksi (engl. *error patterns*) ja niillä on tapana lisääntyä lähdekoodissa uudestaan ja uudestaan sen seurauksena, että projektissa on mukana monta ohjelmistokehittäjää. Virhemallit ovat yleinen syy ohjelmiston heikkoukseen. Jokainen yksittäinen

malli voi aiheuttaa vain pieniä virheitä tietyssä kohtaa projektia, mutta otettaessa huomioon koko projektin elinkaari, näiden virhemallien vaikutus on melko vakava eikä niitä voida sivuuttaa, jos kehitetyn ohjelmiston halutaan olevan laadukas. Ohjelmistoaineistoja louhimalla virhemallit voidaan löytää ja korjata ohjelmistosta esimerkiksi käyttäen ohjelmistoprojektin versionhallintatietoja. (Livshits ja Zimmermann 2005)

Versionhallintaa louhimalla voidaan ohjata ohjelmistokehittäjiä toisiinsa liittyvien muutosten teossa. Voidaan löytää esimerkiksi sellaista tietoa, että jotkut kehittäjät ovat muuttaessaan tiettyjä funktioita muuttaneet samalla myös muita funktioita. Tämä auttaa kehittäjää tietämään mitä muuta hänen tulee mahdollisesti muuttaa tehdessään yksittäisen muutoksen jonnekin lähdekoodiin. Kun tiedetään joukko tapahtuneita muutoksia, assosiaatiosääntöjen avulla voidaan ehdottaa ja ennustaa todennäköisesti tapahtuvat muutokset, löytää kahden muutoksen yhteenkytkentä, jota ei havaita ohjelmiston analysoinnissa, ja ehkäistä vaillinaisista kytköksistä johtuvat virheet. (Zimmermann et al. 2005)

Livshits ja Zimmermann esittävät artikkelissaan (2005) tiettyjä havaintoja versionhallintatietojen louhintaan liittyen. Useista ohjelmistokomponenteista, jotka käyttävät samaa APIa, löytyy yleensä yhteisiä virheitä liittyen kyseiseen APIin. Lähdekoodiin useasti yhdessä liitetyt metodikutsut usein muodostavat tietyn mallin. Pienet muutokset versionhallinnassa usein liittyvät virheiden korjaukseen.

3 Algoritmit

Tiedonlouhinta-algoritmien osia ovat *malli- ja hahmorakenne*, *pisteytysfunktio*, *optimointi- ja hakumenetelmä* sekä *tiedonhallintastrategia*. Malli- ja hahmorakenteessa määrätään datajoukosta etsittävät rakenteet ja muodot. Pisteytysfunktioita käytetään todentamaan, sopiiko käytetty malli paremmin annettuun datajoukkoon kuin jokin muu malli. Esimerkiksi neliösummapisteytysfunktio on muotoa

$$\sum_{i=1}^{n} (y(i) - \hat{y}(i))^2$$

jossa ennustetaan n kohdearvoa $y(i)$, $1 \leq i \leq n$. Ennusteet esitetään muuttujalla $\hat{y}(i)$. Optimoinnin ja hakumenetelmien tarkoituksena on määrätä rakenne ja parametriarvot, joista saadaan pisteytysfunktion raja-arvo, minimi tai maksimi. Tiedonhallinan strategioilla tarkoitetaan datajoukon tallentamista, indeksointia ja saantia.

Assosiaatiosääntöjen louhimisen historiaan liittyy kaksi algoritmia, *AIS* ja *SETM*. AIS-algoritmi on kehitetty vuonna 1993 ja nimetty keksijöidensä mukaan (Agrawal, Imielinski ja Swami). Algoritmi suunniteltiin asiakkiden ostoskoritietojen analysointia varten (Srikant 1996). AIS-algoritmissa käytettiin minimituki-parametria ja algoritmin avulla löydettiin kaikki assosiaatiosäännöt, mutta säännöt olivat suppeita. SETM-algoritmi kehitettiin vuonna 1995 ja se on muunnelma AIS-algoritmista. SETM kehitettiin relaatiotietokantojen käsittelyä varten. Algoritmissa käytetään relaatio-operaatioita ja siten tiedonrikastus voidaan tehdä myös SQL-kyselykieltä käyttämällä. Myös SETM-algoritmissa käytetään minimitukiparametria. (Adamo 2001)

AIS- ja SETM-algoritmit eivät ole enää nykyisin kovinkaan käytännöllisiä, koska Apriori-algoritmi on tehokkaampi assosiaatiosääntöjen louhimisessa (Adamo 2001). Tämän vuoksi AIS ja SETM -algoritmit sivuutetaan maininnalla ja keskitytään tärkeämpiin ja käytännöllisempiin algoritmeihin. Näitä algoritmeja esitellään seuraavissa luvuissa.

3.1 Apriori

Useiden hahmonlouhinta-algoritmien perustana on käytetty vuonna 1994 esitettyä *Apriori*-algoritmia. Algoritmin suorituksessa hyödynnetään aikaisempaa tietoa, josta algoritmi on saanut nimensä. (Han ja Kamber 2000) Apriori-algoritmi käyttää leveyshakumenetelmää (engl. *breadth first search*) (Hegland 2003).

Apriori-algoritmissa käytetään yhtenäistä minimituki-parametria. Minimituki-parametri on tällöin sama kaiken kokoisille alkiojoukoille.

Apriori-algoritmille annetaan lähtötietoina tapahtumaperäinen tietokanta D, sekä käyttäjän ilmoittama numeerinen minimituki-parametri min_supp. Algoritmi käy tietokantaa läpi iteratiivisella leveyshaulla. Ensin siis etsitään todennäköisyys alkiojoukoille, joihin kuuluu vain yksi alkio. Toisella kierroksella etsitään todennäköisyys alkiojoukoille, joihin kuuluu kaksi alkiota ja niin edelleen. Tällöin saadaan aikaisempaa tietoa hyödyntäen selville kandidaattialkiojoukot C_k. Kandidaattialkiojoukot ovat mahdollisia usein esiintyviä alkiojoukkoa L_k. Kandidaattialkiojoukkojen frekvenssiä ja annettua minimituki-parametriä vertaamalla löydetään tietokannassa usein esiintyvät alkiojoukot, joista lopuksi johdetaan assosiaatiosäännöt. (Han ja Kamber 2000)

Apriori-algoritmiin kuuluu kaksi vaihetta. Ensin luodaan kandidaattialkiojoukot C_k. Tämän jälkeen käydään tietokanta läpi ja lasketaan kandidaattialkiojoukkojen esiintymislukumäärät. Usein esiintyvät alkiojoukot L_k saadaan karsimalla kandidaattialkiojoukoista ne, joiden esiintymislukumäärä on alle annetun minimituki-parametrin. Näiden vaiheiden suoritusta jatketaan niin kauan, että usein esiintyviä alkiojoukkoja ei enää löydy. (Han ja Kamber 2000) Algoritmin perusosa on seuraavanlainen:

Input: D, a database of transactions;
min_supp, the minimum support count threshold.

Output: L, frequent itemsets in D.

Method:

L_1 = find_frequent_1-itemsets(D);
for ($k = 2$; $L_{k-1} \neq \emptyset$; k++) {
    C_k = **apriori-gen(** L_{k-1} **);**

for each transaction $t \in D$ { // scan D for counts
C_t = subset(C_k, t); // get the subsets of t that are candidates
for each candidate $c \in C_t$
c.count++;
}
$L_k = \{ c \in C_k \mid c.count \geq min\_supp \}$
}
return $L = \cup_k L_k$;

Algoritmin funktiolla `find_frequent_1-itemsets` etsitään kaikki tietokannassa usein esiintyvät alkiot. Kandidaattialkiot muodostetaan Apriori-algoritmin funktiolla `apriori_gen`:

procedure apriori_gen(L_{k-1}: frequent(k-1)-itemsets)
for each itemset $l_1 \in L_{k-1}$
for each itemset $l_2 \in L_{k-1}$
if (l_1[1] = l_2[1]) $\wedge$ (l_1[2] = l_2[2]) $\wedge$ ... $\wedge$ (l_1[k-2] = l_2[k-1])
$\wedge$ (l_1[k-1] < l_2[k-1]) **then** {
$c = l_1 \bowtie l_2$; // join step: generate candidates
if has_infrequent_subset(c, L_{k-1}) **then**
delete c; // prune step: remove unfruitful candidate
else add c **to** C_k;
}
return C_k;

Apriori-ominaisuutta, eli algoritmin funktiota `has_infrequent_subset`, käytetään algoritmin tehostamiseen. Tätä kutsutaan *alaspäin suunnatuksi sulkeumaksi* (engl. *downward closure*). Apriori-ominaisuudella hylätään ne alkiojoukot, jotka eivät ole usein esiintyviä, eli joiden alijoukko ei ole usein esiintyvä. (Han ja Kamber 2000) Funktio on seuraavanlainen:

procedure has_infrequent_subset(c: candidate k-itemset;
L_{k-1}: frequent (k-1)-itemsets); // use prior knowledge
for each(k-1)-subset s **of** c

if $s \notin L_{k-1}$ **then**
return TRUE;
return FALSE;

Tarkastellaan Apriori-algoritmin toimintaa esimerkin avulla. Asetetaan algoritmin minimitukiparametri `min_supp` = 2. Tällöin usein esiintyvien alkiojoukkojen on siis esiinnyttävä datajoukossa vähintään kahdesti. Olkoon datajoukko taulukon 7 mukainen.

Alkiot
{A, F}
{A, B, C}
{A, C, E}
{A, B, F}
{A, B, C}
{A, D, G}
{A, D, I}

Taulukko 7. Datajoukko.

Ensimmäiseksi algoritmissä etsitään datajoukosta usein esiintyvät 1-alkiojoukot, eli alkiojoukot, jotka sisältävät yhden alkion. Etsiminen tapahtuu funktiolla `find_frequent_-1-itemsets`. Tuloksena taulukon 8 mukainen alkiojoukko L_1.

Alkiot
{A}
{B}
{C}
{D}
{F}

Taulukko 8. Alkiojoukko L_1.

Seuraavaksi etsitään alkiojoukosta L_1 kandidaatti-2-alkiojoukot liitosoperaatiolla $L_1 \bowtie L_1$, josta tuloksena taulukon 9 mukainen alkiojoukko C_2.

Alkiot	**Tuki**
{A, B}	3
{A, C}	3
{A, D}	2
{A, F}	2
{B, C}	2
{B, D}	0
{B, F}	1
{C, D}	0
{C, F}	0
{D, F}	0

Taulukko 9. Alkiojoukko C_2.

Minimituki-parametriin verrattuna turhat alkiot voidaan unohtaa, jolloin tuloksena on taulukon 10 mukainen alkiojoukko L_2.

Alkiot
{A, B}
{A, C}
{A, D}
{A, F}
{B, C}

Taulukko 10. Alkiojoukko L_2.

Tämän jälkeen suoritetaan seuraava iteraatio, eli alkiojoukosta L_2 luodaan kandidaatti-3-al-

kiojoukot liitosoperaatiolla. Esimerkissä ainoastaan yhdellä alkiolla on tarpeeksi suuri frekvenssi minimitukeen verrattuna ja muut alkiot voidaan unohtaa, tällöin lopullisena tuloksena taulukon 11 mukainen alkiojoukko L_3. Alkiojoukko L_3 on tapahtumaperäisen tietokannan usein esiintyvä alkiojoukko ja assosiaatiosääntöjen johtamisen ensimmäinen vaihe on saatu päätökseen.

Alkiot
{A, B, C}

Taulukko 11. Alkiojoukko L_3.

Assosiaatiosääntöjen johtamisen toisessa vaiheessa usein esiintyvistä alkiojoukoista muodostetaan assosiaatiosääntöjä. Jotta assosiaatiosääntö on vahva, tulee sen ylittää uskottavuus- ja tukimittarit, joita käsiteltiin luvussa 2.2. Apriori-algoritmia käytettäessä lopulliset usein esiintyvät alkiojoukot ylittävät automaattisesti minimituki-parametrin (Han ja Kamber 2000). Jäljelle siis jää selvittää, ylittävätkö usein esiintyvät alkiojoukot myös uskottavuus-parametrin arvon ja onko siten löydetty varsinaisia assosiaatiosääntöjä. Jatketaan edellistä esimerkkiä, eli tutkitaan ylittääkö alkiojoukko `{A, B, C}` uskottavuus-parametrin arvon. Jaetaan ensin usein esiintyvä alkiojoukko epätyhjiksi alijoukoiksi. Näitä epätyhjiä alijoukkoja esimerkin mukaisesti ovat `{A}`, `{B}`, `{C}`, `{A, B}`, `{A, C}` ja `{B, C}`. Saadut joukot yhdistetään säännöiksi ja niiden uskottavuus saadaan laskettua kaavalla 2.1. Uskottavuusarvot esimerkin mukaisesta usein esiintyvästä alkiojoukosta johdetuille assosiaatiosäännöille on laskettuna taulukossa 12. Esimerkiksi säännön $C \Rightarrow A \wedge B$ uskottavuusarvo saadaan kaavalla

$$uskottavuus = \frac{\#\{A,B,C\}}{\#\{C\}}$$

eli tarkastelemalla kuinka monta kertaa alkiojoukko *{A, B, C}* esiintyy taulukon 7 datajoukossa ja asettaa tämä jakolaskun jaettavaksi luvuksi, jakava luku saadaan tarkastelemalla kuinka monta kertaa alkiojoukko `{C}` esiintyy kyseisessä datajoukossa. Tästä kyseiselle säännölle saadaan uskottavuudeksi 67. Jos uskottavuus on 100, sanotaan kyseistä sääntöä tarkaksi. Jos asetamme uskottavuus-parametrin arvoksi 55, vain yksi sääntö alittaa tämän.

Loput säännöt ovat siis varsinaisia assosiaatiosääntöjä. Uskottavuus-parametrin sopivan arvon valinta riippuu minimituki-parametrin tavoin käytetystä lähtöaineistosta ja kuinka paljon assosiaatiosääntöjä halutaan tuloksena löytää.

Alkiot	**Uskottavuus**
$B \wedge C \Rightarrow A$	2/2 = 100 %
$A \wedge B \Rightarrow C$	2/3 = 67 %
$A \wedge C \Rightarrow B$	2/3 = 67 %
$A \Rightarrow B \wedge C$	2/7 = 29 %
$B \Rightarrow A \wedge C$	2/3 = 67 %
$C \Rightarrow A \wedge B$	2/3 = 67 %

Taulukko 12. Uskottavuusarvot alkiojoukosta *{A, B, C}* johdetuille assosiaatiosäännöille.

Datajoukot voivat olla hyvinkin suuria ja Apriori-algoritmia käyttämällä löydetään valtavasti kandidaattialkiojoukkoja niiden kasvaessa joka iteraatiokerralla. Alkiojoukon suuruus vaikuttaa kandidaattien määrään kaavan

$$\sum_{i=1}^{L_k} \left(\frac{L_k}{i}\right) = 2^{L_k} - 1 \tag{3.1}$$

mukaisesti (Kantardzic 2002), jossa L_k tarkoittaa alkioiden määrää usein esiintyvässä alkiojoukossa. Esimerkiksi 100-alkiojoukon muodostamiseen tarvittaisiin kaavan mukaan noin 10^{30} kandidaattialkiojoukkoa. Apriori-algoritmistä löytyy myös tehokkaampia versioita. Suorituskykyä parantavat vaihtoehdot voidaan jakaa kolmeen luokkaan (Yang, Pan ja Chung 2001). Ensimmäisessä luokassa käytetään hajautustekniikkaa vähentämään kandidaattialkiojoukkojen määrää. Toisessa luokassa käytetään osiointia, jolloin datajoukon läpikäyntien määrä vähenee. Kolmannessa luokassa muodostetaan L_k-alkiojoukkoja *kokoavan* (engl. *bottom-up*) ja *osittavan* (engl. *top-down*) *haun* yhdistelmällä. Näiden lisäksi voidaan käyttää otantaa ja *dynaamisen alkiojoukkojen laskentaa* (engl. *dynamic itemset counting*). (Han ja Kamber 2000)

Apriorista on kehitetty tekstiaineiston louhintaan paremmin soveltuva *Pehmeä Apriori* (engl.

Soft-Apriori). Tapahtumatietokannoissa tieto on yleensä strukturoidussa tai formaalissa muodossa ja käytössä on useasti esitystavan perusteella yksilöiviä tunnisteita, esimerkiksi tuotenumerot. Tekstitietokannoissa tietosisältö taas on luonnollista kieltä. Sanan kieliopillinen ulkomuoto voi olla erilainen eri yhteyksissä. Myöskään erisnimiä ei tunnisteta samaksi tiedoksi. Pehmeässä Apriorissa voidaan analysoida myös alkioiden samanlaisuutta. Tätä voidaan tutkia esimerkiksi vertaamalla sanoja jo ennalta määrättyyn sanastoon (engl. *vector space model*) tai määritellä kuinka helposti merkkijono voidaan muuttaa toiseksi (engl. *edit distance*). (Nahm ja Mooney 2002)

3.2 DHP

DHP (*Direct Hashing and Pruning*) eli *Suora hajautus- ja karsinta- algoritmi* on kehitetty vuonna 1997. Algoritmilla on kolme tehokkuusominaisuutta. Näitä ovat suurien alkiojoukkojen muodostaminen, tapahtumapohjaisen tietokannan koon pienentäminen ja tietokannan läpikäynnin määrän vähentäminen (Park, Chen ja Yu 1997). DHP on johdettu Apriori-algoritmista ja ensimmäisen laskentakierroksen osalta nämä kaksi algoritmia toimivatkin samalla tavoin (Adamo 2001). Seuraavilla iteraatiokerroilla DHP kuitenkin käyttää hajautusta ja karsintaa osajoukkojen muodostamiseen ja esiintymistodennäköisyyksien laskentaan. Jos ehdokasosajoukkojen yhdistelmistä ei löydy haluttua esiintymistodennäköisyyttä, ne karsitaan saman tien. Tekstiaineistoa käsiteltäessä hajautusalgoritmit toimivat Apriori-algoritmia tehokkaammin. (Holt ja Chung 2001)

Hajautuksessa muodostetaan tiettyä hajautusfunktiota käyttäen hajautustaulu jokaisella laskentakierroksella. Jokaiselle ehdokasalkiojoukolle on oma hajautusarvonsa. Jos ehdokasalkiojoukko esiintyy tietyn iteraation datajoukossa, kyseisen alkiojoukon hajautusarvoa kasvatetaan. Ehdokasalkiojoukko karsitaan seuraavilta iteraatioilta, mikäli sen hajautusarvo on pienempi kuin määritelty minimituki-parametrin arvo. (Holt ja Chung 2001)

Tarkastellaan DHP-algoritmin toimintaa luvun 3.1 esimerkin avulla. Asetetaan algoritmin minimituki-parametri `min_supp` = 2. Tällöin usein esiintyvien alkiojoukkojen on siis esiinnyttävä datajoukossa vähintään kahdesti. Olkoon datajoukko Apriori-algoritmin esimerkin taulukon 7 mukainen. Tällöin Apriori-esimerkin tavoin saadaan taulukon 8 mukainen alkio-

joukko L_1. Nyt luodaan datajoukon perusteella kaikki 2-alkiojoukot, tulos on taulukon 13 mukainen.

Alkiojoukot
{ {A, F} }
{ {A, B}, {A, C}, {B, C} }
{ {A, C}, {A, E}, {C, E} }
{ {A, B}, {A, F}, {B, F} }
{ {A, B}, {A, C}, {B, C} }
{ {A, D}, {A, G}, {D, G} }
{ {A, D}, {A, I}, {D, I} }

Taulukko 13. Kaikki 2-alkiojoukot.

Nyt on vuorossa hajautustaulun luominen. Olkoon hajautusfunktio esimerkiksi

$$h(\{x,y\}) = (pos_{C1}(x) * 10 + pos_{C1}(y)) \; mod \; 7 \tag{3.2}$$

jossa pos_{C1} tarkoittaa alkion järjestysnumeroa ja *mod* moduloa. Nyt annetaan alkioille aakkosjärjestyksen mukainen juokseva numerointi, täten siis esimerkiksi $pos_{C1}(A) = 1$ ja $pos_{C1}(I) = 8$, koska datajoukossa ei ole mukana alkiota *G*. Hajautusfunktiolla 2-alkiojoukolle laskettu arvo ilmoittaa sen, mihin hajautustaulun osoitteeseen kyseinen alkiojoukko asetetaan. Esimerkiksi $\{A,F\}$ asetetaan hajautustaulun osoitteeseen 2, koska *h(A, F) = (1*10 + 6) mod 7 = 2*. Hajautustaulun osoitteen mukaista painoa kasvatetaan aina yhdellä, kun kyseiseen osoitteeseen asetetaan alkiojoukko. Lopullisena tuloksena taulukon 14 mukainen hajautustaulu H_2.

Nyt hajautustaulusta voidaan unohtaa ne alkiojoukot, joiden osoitteen mukainen painoarvo on alle minimituki-parametrin arvon. Näin on saatu taulukon 15 mukainen alkiojoukko L_2.

Esimerkissä siis hajautusalgoritmin toiminta yhden iteraation osalta. Seuraavat iteraatiot toi-

	{A, D} {A, D} {C, E}	{A, E}	{B, C} {A, F} {B, C} {A, F}	{A, G}	{A, I}	{D, G} {A, B} {B, F} {A, B} {A, B}	{D, I} {A, C} {A, C} {A, C}
paino	3	1	4	1	1	5	4
osoite	0	1	2	3	4	5	6

Taulukko 14. Hajautustaulu H_2.

Alkiot
{C, E}
{A, D}
{A, F}
{B, C}
{A, B}
{B, F}
{D, G}
{A, C}
{D, I}

Taulukko 15. Alkiojoukko L_2.

mivat samalla tavalla alkiojoukkojen kokoa suurentamalla.

DHP-algoritmin toiminta riippuu siitä, kuinka iso hajautustaulu valitaan käyttöön. DHP on kuitenkin Apriori-algoritmia tehokkaampi (Adamo 2001). Ensimmäisen iteraation aikana DHP-algoritmin suoritusaika on jonkin verran Apriori-algoritmin suoritusaikaa hitaampi, mutta seuraavilla iteraatioilla DHP-algoritmin suoritusaika on nopeampi (Park, Chen ja Yu 1997). Molemmat algoritmit lukevat datajoukkoa jokaisen iteraation aikana, mutta DHP-algoritmissa jokaisella iteraatiolla tutkittavan datajoukon määrä vähenee. Tämän periaate on siinä, että jokaisen alkion, joka kuuluu usein esiintyvään alkiojoukkoon, tulee itsekin olla usein esiintyvä (Holt ja Chung 2001). Jos näin ei ole, voidaan alkio karsia pois saman tien.

DHP-algoritmi ei sinällään Apriori-algoritmin tavoin sovellu tekstiaineiston louhintaan suuren muistinkäytön takia. DHP-algoritmi tarvitsee reilusti muistia laskiessaan monien potentiaalisten usein esiintyvien alkiojoukkojen esiintymiskertoja. DHP-algoritmistä onkin kehitetty *M-DHP* eli *Toisteinen hajautus- ja karsinta-algoritmi*, jossa muistin tarvetta on saatu vähennettyä osoioimalla kandidaatti-1-alkiojoukot, jonka jälkeen jokainen osio käsitellään erikseen. Osioita käsiteltäessä käytetään hajautusta ja karsintaa, jolloin käsiteltävä joukko saadaan pienenemään nopeasti jokaisella iteraatiolla. M-DHP soveltuu tekstiaineiston louhintaan huomattavasti paremmin kuin DHP. (Holt ja Chung 2001)

3.3 IHP

IHP (*Inverted Hashing and Pruning*) eli *Käänteinen hajautus- ja karsinta-algoritmi* on kehitetty vuonna 2002 DHP-algoritmista ja eroaakin siitä hajautuksen toteutustavan osalta. Hajautuksessa kandidaattiosajoukot korvataan tunnisteella (engl. *transaction id*) ja tunnisteet hajautetaan hajautustauluun. Epäoleellisten alkiojoukkojen karsiminen voidaan aloittaa jo yksialkioisista joukoista ja näin saadaan vaikutettua tulevien iteraatioiden kandidaattialkiojoukkojen lukumäärään. Tämä nopeuttaa algoritmin kokonaissuoritusaikaa tekstiaineistoa käsiteltäessä. (Holt ja Chung 2002)

3.4 Toistuvan hahmon kasvatus

Aikaisemmin esitetyt algoritmit perustuvat kandidaattijoukkojen muodostamiseen ja iteraatioiden läpikäymiseen. *Toistuvan hahmon kasvatus* (engl. *Frequent Pattern Growth*) -algoritmissa ei käytetä kandidaattialkiojoukkojen käsittelyä, vaan algoritmi perustuu *hajoita ja hallitse* (engl. *divide-and-conquer*) -menetelmään. Ensin datajoukko tiivistetään siten, että usein esiintyvistä alkioista muodostetaan *FP-puu* (engl. *Frequent-Pattern Tree*) ja tämän jälkeen se jaetaan ehdollisiin alkiojoukkoihin. (Han ja Kamber 2000) Algoritmi on seuraavanlainen:

Input: D, a database of transactions;
min_supp, the minimum support count threshold.

Output: The complete set of frequent patterns.

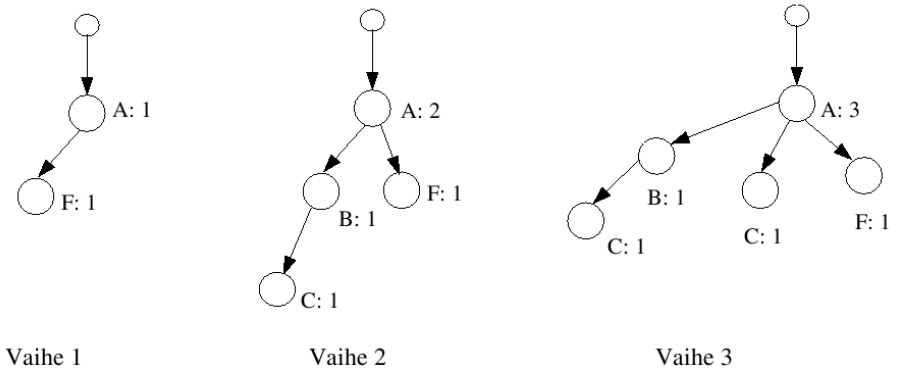
Method:

procedure FP_growth(*Tree*, α)
  if *Tree* contains a single path P **then**
    for each combination (denoted as β) of the nodes in the path P
      generate pattern $\beta \cup \alpha$ with *support_count = minimum support count of nodes in* β;
  else for each a_i in the header of *Tree* {
    generate pattern $\beta = a_i \cup \alpha$ with *support_count* = a_i.*support_count*;
    construct β's conditional pattern base and then β's conditional FP_tree $Tree_\beta$;
    if $Tree_\beta \neq \emptyset$ **then**
      call **FP_growth**($Tree_\beta$, β);
  }

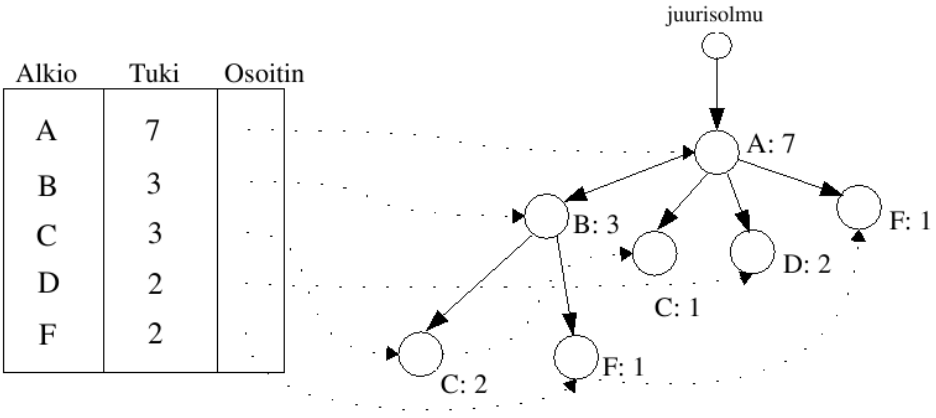
Tarkastellaan algoritmia esimerkin avulla. Olkoon datajoukkona Apriori-esimerkin taulukon 7 datajoukko ja minimituki-parametrin arvona *min_supp = 2*. Etsitään usein esiintyvät 1-alkiojoukot ja järjestetään ne laskevaan suuruusjärjestykseen datajoukon esiintymiskertojen mukaisesti. Esimerkissä löydetyt alkiojoukot ovat jo valmiiksi laskevassa suuruusjärjestyksessä, sillä alkio *A* esiintyy datajoukossa seitsemän kertaa, alkio *B* kolme kertaa ja niin edelleen. Tulos on siis taulukon 8 mukainen. Seuraavaksi tästä tuloksesta luodaan FP-puu. Puu koostuu solmuista, joihin talletetaan alkioiden tiedot. Solmut järjestetään hierarkisesti

vanhempi-lapsi-suhteiden mukaan. Tällöin jokaisella solmulla on lapsisolmu ja vanhempisolmu poikkeuksena erityiset solmut juurisolmu ja lehtisolmut. Juurisolmuksi kutsutaan hierarkisesti kaikkein vanhinta solmua, jolla ei ole vanhempisolmua ja joita voi olla vain yksi jokaisessa puussa. Lehtisolmut ovat kaikkein nuorimpia solmuja, joilla ei siis ole lapsisolmuja. Lehtisolmuja voi esiintyä useita samassa puussa. Puun tekeminen aloitetaan luomalla juurisolmu, jolla ei vielä ole lapsisolmuja. Tämän jälkeen puuhun lisätään solmuja vaiheittain datajoukon mukaisesti, kuten kuviossa 2 on esitetty. Solmut tulee lisätä puuhun laskevan suuruusjärjestyksen mukaisesti. Jos siis datajoukossa olisi alkiojoukko $\{A,F,C\}$, tuli ne lisätä puuhun poluksi järjestyksessä $\{A,C,F\}$, koska A esiintyy datajoukossa useammin kuin C, joka taas esiintyy datajoukossa useammin kuin F. Ensimmäinen datajoukon alkiojoukko on $\{A,F\}$, joten ensimmäisessä vaiheessa asetetaan juurisolmulle lapsisolmu, jonka arvo on A ja esiintymiskertojen lukumäärä 1 ja tälle solmulle asetetaan lapsisolmu F, jonka esiintymiskertojen lukumäärä on 1. Toinen vaihe on lisätä alkiojoukko $\{A,B,C\}$. Juurisolmun lapsisolmuksi siis pitää tulla A, mutta tällainen on jo juurisolmun lapsisolmuna. Lisätään siis vain tämän solmun esiintymiskertojen lukumäärää yhdellä, lisätään tämän solmun lapsisolmuksi B ja edelleen polkua pitkin lapsisolmu C. Kolmannessa vaiheessa lisätään alkiojoukko $\{A,C\}$. Lisätään juurisolmun lapsisolmun A esiintymiskertojen lukumäärää yhdellä ja lisätään tämän lapsisolmuksi C. Näin jatketaan, kunnes kaikki usein esiintyvät alkiojoukot datajoukosta on lisätty puuhun. Lopputulos on kuvion 3 mukainen. Kuviossa näkyvissä myös luotu alkiotaulukko, josta ilmenee alkio, sen esiintymiskertojen lukumäärä datajoukossa sekä osoitin alkion esiintymään FP-puussa. Näin ei tarvitse enää louhia datajoukkoa, vaan voidaan käsitellä pelkästään luotua FP-puuta. Tästä päästäänkin FP-puun louhimiseen.

FP-puun louhiminen aloitetaan viimeisestä usein esiintyvästä 1-alkiojoukosta, kun alkiojoukot ovat järjestetty laskevaan järjestykseen. Tässä tapauksessa siis alkiosta F. Tämä alkio löytyy FP-puussa kahdesta eri polusta, joita ovat $\{A,B,F:1\}$ ja $\{A,F:1\}$. Jos F siis kuvitellaan jälkeläiseksi, sen kaksi esipolkua ovat tällöin $\{A,B:1\}$ ja $\{A:1\}$. Nämä esipolut muodostavat alkion F *ehdolliset hahmot* (engl. *conditional pattern base*). Tästä taas saadaan alkion F ehdollinen FP-puu. Tämä ehdollinen FP-puu sisältää vain yhden polun, joka on $\{A:2\}$. B:n tuki ehdollisissa poluissa on alle *min_supp*-arvon, jolloin se voidaan unohtaa ja jäljelle jää vain A. Tästä taas saadaan alkion F usein esiintyvät hahmot, joita ovat $\{A,F:2\}$. Näin jatketaan myös muiden usein esiintyvien 1-alkiojoukkojen alkioille. Tuloksena taulu-



Kuvio 2. FP-puun luominen vaiheittain.



Kuvio 3. Esimerkin mukainen valmis FP-puu.

kon 16 mukaiset polut ja hahmot.

Alkio	Ehdolliset hahmot	Ehdolliset FP-puut	Usein esiintyvät hahmot
F	{A, B: 1}, {A: 1}	{A: 2}	{A, F: 2}
D	{A: 2}	{A: 2}	{A, D: 2}
C	{A, B: 2 }, {A: 1}	{A: 3, B: 2}	{A, C: 3}, {A, B, C: 2}
B	{A: 3}	{A: 3}	{A, B: 3}

Taulukko 16. Esimerkin mukaisen FP-puun louhinnan tulos

Toistuvan hahmon kasvatus helpottaa ongelmaa pitkien usein esiintyvien hahmojen löytämisestä etsimällä lyhyempiä usein esiintyviä hahmoja rekursiivisesti ja sen jälkeen yhdistämällä jälkeläiset. Algoritmi on suuruusluokaltaan Apriori-algoritmia nopeampi ja soveltuu niin lyhyiden kuin pitkienkin usein esiintyvien hahmojen etsimiseen. (Han ja Kamber 2000) Suuria datajoukkoja käsiteltäessä muistikapasiteetti voi tulla rajoitteeksi, mutta tällaisissa tilanteissa kannattaa datajoukko osioida pienempiin osiin ja käyttää algoritmia näihin osiin rekursiivisesti.

3.5 Pystysuuntaisia datajoukkoja käsittelevät algoritmit

Kaikissa edellisissä algoritmeissa käytetty datajoukko on *vaakasuuntaisessa* (engl. *horizontal data format*) muodossa koko algoritmin suorituksen ajan. Vaakasuuntaisella datajoukolla tarkoitetaan esimerkiksi tapahtumapohjaista tietokantaa, jossa ensimmäisellä sarakkeella on mainittu tapahtumatunnus (ID-arvo) ja seuraavalla sarakkeella on tieto alkiojoukoista. Sellaisiakin algoritmeja on olemassa, jotka toimivat toisinpäin, eli *pystysuuntaisesti* (engl. *vertical data format*). Tämä on esimerkiksi *Eclat* ja *SPADE* -algoritmien keskeinen osa (Zaki 2000). SPADE-algoritmin on todettu toimivan tehokkaasti, se tarvitsee kaiken kaikkiaan vain kolme datajoukon käsittelykertaa (Zaki 2001). Sillä on kuitenkin kaksi heikkoutta, joista ensimmäinen on pystysuuntaisia datajoukkoja käsitteleville algoritmeille hyvin yleinen. Monissa tietojärjestelmissä tieto tallennetaan vaakasuuntaisesti, jolloin pystysuuntaisen algoritmin käyttöä varten datajoukko on ensin muunnettava pystysuuntaiseen muotoon. Tämä tietenkin lisää koko prosessin suoritusaikaa. Toinen SPADE-algoritmin heikkous on se, että se luo suurel-

la datajoukon koolla pitkiä tapahtumatunnusjoukkoja ja vaakasuuntaista datajoukkoa, mikä lisää muistikapasiteetin tarvetta. (Ching ja Ng 2003)

Tässä tutkielmassa perehdytään yleisemmin pystysuuntaisten datajoukkojen käsittelyyn usein esiintyvien hahmojen etsimisessä, kuin lähemmin jonkin tietyn tällaisen ominaisuuden toteuttavan algoritmin tarkasteluun, koska on harvinaisempaa, että tietojärjestelmän datajoukko olisi tallennettu pystysuuntaisesti. Tarkastellaan pystysuuntaisen datajoukon käsittelyä esimerkin avulla. Olkoon minimituki-parametri $min\_supp = 2$ ja datajoukkona taulukon 17 mukainen tapahtumatietokanta. Tässä datajoukko on vielä vaakasuuntaisena ja käymällä se kerran läpi, saadaan taulukon 18 mukainen pystysuuntainen datajoukko.

Tapahtuma ID	**Alkiot**
T100	{A, F}
T200	{A, B, C}
T300	{A, C, E}
T400	{A, B, F}
T500	{A, B, C}
T600	{A, D, G}
T700	{A, D, I}

Taulukko 17. Tapahtumapohjainen vaakasuuntainen datajoukko.

Tästä voidaan luoda 2-alkiojoukot yhdistämällä kaikki usein esiintyvät 1-alkiojoukot keskenään pareiksi. Tässä alkiot E, G, ja I eivät esiinny datajoukossa riittävän useasti, joten ne voidaan karsia pois. Tällöin saadut kandidaatti-2-alkiojoukot ovat taulukon 19 mukaiset.

Tästä saadaan usein esiintyvät 2-alkiojoukot karsimalla pois ne alkiojoukot, jotka eivät esiinny riittävän usein, eli tässä tapauksessa alkiojoukko $\{B,F\}$. Seuraavaksi luodaan usein esiintyvistä 2-alkiojoukoista yhdistelmät 3-alkiojoukkoja varten ja lasketaan näiden esiintymät. Erilaisiksi yhdistelmiksi muodostuvat alkiojoukot $\{A,B,C\}$, $\{A,B,D\}$ ja $\{A,B,F\}$. Alkiojoukko {A, B, D} ei kuitenkaan esiinny 2-alkiojoukon mukaisessa datajoukossa kertaakaan, joten se voidaan karsia pois. Alkiojoukko $\{A,B,F\}$ esiintyy datajoukossa vain kerran (`T400`), joten sekin voidaan poistaa usein esiintyvien 3-alkiojoukkojen listalta. Lopullinen

Alkio	Tapahtuma ID -joukko
{A}	{T100, T200, T300, T400, T500, T600, T700}
{B}	{T200, T400, T500}
{C}	{T200, T300, T500}
{D}	{T600, T700}
{E}	{T300}
{F}	{T100, T400}
{G}	{T600}
{I}	{T700}

Taulukko 18. Tapahtumapohjainen pystysuuntainen datajoukko

Alkiojoukot	Tapahtuma ID -joukko
{A, B}	{T200, T400, T500}
{A, C}	{T200, T300, T500}
{A, D}	{T600, T700}
{A, F}	{T100, T400}
{B, C}	{T200, T500}
{B, F}	{T400}

Taulukko 19. 2-alkiojoukot.

usein esiintyvien 3-alkiojoukkojen ryhmä on taulukon 20 mukainen.

Alkiojoukot	**Tapahtuma ID -joukko**
{A, B, C}	{T200, T500}

Taulukko 20. Usein esiintyvät 3-alkiojoukot.

Pystysuuntaisessa datajoukon käsittelyssä on se hyöty Apriori-algoritmiin verrattuna, että tietokantaa ei välttämättä tarvitse käydä joka iteraatiolla läpi, jotta löydettäisiin alkiojoukon esiintymiskerrat. Alkiojoukkojen esiintymiskerrat voidaan nimittäin laskea suoraan *Tapahtuma ID* -joukkojen lukumäärästä. Nämä joukot voivat kuitenkin kasvaa melko suuriksi, jolloin muistikapasiteetin tarve kasvaa ja suoritusaika pitenee. Tapahtumatunnus-joukkojen pitkiä listauksia voidaan helpottaa pitämällä kirjaa vain niistä joukoista, jotka muuttuvat iteraatioiden välillä. (Han ja Kamber 2000) Tarkastellaan edellisen esimerkin mukaisesti taulukkoa 18. Alkiolla $\{B\}$ on tapahtumatunnusjoukko $\{T200, T400, T500\}$ ja seuraavalla iteraatiolla (taulukko 19) alkiojoukolla $\{B,C\}$ on tapahtumatunnusjoukko $\{T200, T500\}$. Tällöin tapahtumatunnusjoukon muutos $diffset(\{B,C\},\{B\}) = \{T400\}$. Tällöin tarvitsee ilmoittaa vain yksi tapahtumatunnus kahden sijasta. Tässä esimerkissä saatu säästö ei vielä muistikapasiteetissa eikä suoritusajassa merkittävää muutosta tuo, mutta varsinkin suurilla tapahtumatunnusjoukoilla, joissa sama tapahtumatunnus esiintyy useasti, tämä menetelmä vähentää muistikapasiteetin tarvetta sekä algoritmin suoritusaikaa.

4 jEdit

jEdit on avoimen lähdekoodin tekstieditointisovellus ohjelmointikäyttöön. jEditin kehitys on aloitettu vuonna 1998 ja se on käytettävissä GPL 2.0 (Free Software Foundation 1991) lisenssiehtojen mukaisesti. jEdit on toteutettu Java-ohjelmointikieltä käyttäen ja se toimii useimmissa käyttöjärjestelmissä. jEdit sisältää tekstin väritysohjeet yli 130 tiedostomuodolle, joiden lisäksi niitä voi lisätä XML-tiedostoja käyttäen. Siitä löytyy tuki useille eri merkistökoodauksille. Sovellus on muokattavissa ja laajennettavissa tehokäyttöä varten erilaisilla skriptikielillä toteutettavien makrojen avulla. jEdit tarjoaa lisäosia eri sovellusalueille yli 150 kappaletta, joiden avulla siitä saa esimerkiksi tehokkaan HTML-editorin tai yhdistetyn sovelluskehitysympäristön (engl. *integrated development environment*) kääntäjineen sekä koodin täydennys- ja testaustyökaluineen. (jEdit Community 2007)

jEdit-projektin versionhallintatiedot ovat lähdekoodin tapaan vapaasti saatavilla lisenssiehtojen mukaisesti. Versionhallintatiedoista voidaan päätellä, että projekti on varsin aktiivinen ja tällä hetkellä projektin versionhallintaan päivitetään noin 500 tiedostoa kuukaudessa. Kirjoitushetkellä (28.4.2013) projektin SVN-versionhallinnan tärkeimmät tilastotiedot ovat koottuna taulukkoon 21.

Kehittäjien lukumäärä	130
Revisioiden lukumäärä	22 955
Tiedostojen lukumäärä	15 856
Java-tiedostojen lukumäärä	6 391
Lähdekoodin rivimäärä	3 216 648

Taulukko 21. Tilastotietoa jEdit-projektin SVN-versionhallinnasta.

5 Soveltaminen

Tämän tutkielman empiirisen osuuden aineistona käytetään jEdit-sovelluksen versionhallintatietoja. Versionhallinta voidaan jakaa eri revisioihin (engl. *revision*). Kun tiedostosta tallennetaan uusi versio versionhallintaan, kutsutaan tätä revisioksi ja se saa versionhallinassa revisionumeron.

5.1 Aikaisemmat tutkimukset

Vastaavia, joskaan ei täysin identtisiä tutkimuksia jEdit:n lähdekoodiaineiston louhimisesta on tehty aiemmin.

Zimmermann et al. (2005) ovat kehittäneet työkalun nimeltä `ROSE`, jonka tehtävä on ohjata sovellusohjelmoijaa tarvittavien lisämuutosten tekemisessä samalla, kun hän tekee muutoksia jEdit:n lähdekoodiin. Työkalu käyttää Apriori-algoritmia assosiaatiosääntöjen löytämiseen. Testiaineistoksi on otettu jEdit:n vuonna 2003 käytössä olleen CVS-versionhallintajärjestelmän 577 viimeisintä revisiota 31.7.2003 lähtien. Työkalun toimivuutta antamalla pisteytys sen saanti- (engl. *recall*), tarkkuus- (engl. *precision*) ja todennäköisyysominaisuudesta (engl. *likelihood*) eri tilanteissa. Toimivuuspisteet eri tilanteissa ovat esitettyinä taulukoissa 22 ja 23.

Navigointi			**Virheiden ennaltaehkäisy**		**Väärät ehdotukset**	
saanti	**tarkkuus**	**todennäköisyys**	**saanti**	**tarkkuus**	**saanti**	**tarkkuus**
0.07	0.16	0.52	0.004	0.59	1.0	0.986

Taulukko 22. ROSE-työkalun toimivuuspisteet matalan tason testitilanteissa.

Työkalun navigoiminen lähdekoodirakenteessa ohjelmoijan tehdessä muutoksia sai saantipisteiksi 0.07, tarkkuuspisteiksi 0.16 ja todennäköisyyspisteiksi 0.52. Nämä pisteet tarkoit-

Navigointi			Virheiden ennaltaehkäisy		Väärät ehdotukset	
saanti	**tarkkuus**	**todennäköisyys**	**saanti**	**tarkkuus**	**saanti**	**tarkkuus**
0.25	0.22	0.68	0.01	0.44	1.0	0.984

Taulukko 23. ROSE-työkalun toimivuuspisteet korkean tason testitilanteissa.

tavat sitä, että työkalu ehdotti oikein 7 % kaikista ehdottamistaan lisämuutoksista tiettyjen muutosten yhteydessä, 16 % kaikista työkalun esittämistä huomioista oli aiheellisia ja 52 %:n todennäköisyydellä lisämuutoksen vaatima kohde sijoittui työkalun ehdottaman kolmen todennäköisimmän kohteen joukkoon. Luvut ovat melko pienet ja artikkelin mukaan tämä johtuu siitä, että jEdit-projektissa tulee jatkuvasti lisää uusia ominaisuuksia, joita ei voida ennustaa olemassa olevan versionhallintatiedon perusteella. Työkalun virheiden ennaltaehkäisy toimii siten, että se tarkastaa tiedostot ennen kuin ohjelmoija tallentaa ne versionhallintaan ja jos työkalu huomaa tekemättömiä, mutta kuitenkin tarpeellisia lisämuutoksia, se kertoo tästä ohjelmoijalle. Tämä ominaisuus sai 0.004 saantipistettä ja 0.59 tarkkuuspistettä. Virheiden ennaltaehkäisyn yhteydessä testattiin myös, kuinka monta lisämuutosehdotusta työkalu antaisi siinä tapauksessa, jos yhtään lisämuutosta ei tarvitse tehdä. Tarkkuuspisteitä tälle testi sai 0.986 ja koska odotettu tulos on tyhjä joukko, saantipisteiksi tulee aina 1.0. Edellä kerrotut pisteet pätevät tilanteessa, jossa työkalun etsimät muutoskohteet ovat muuttujia tai funktioita tiedostojen sisällä. Toisessa testitilanteessa työkalu painottui etsimään muutoskohteita korkeammalta eli tiedostotasolta. Tämän testitilanteen tulos oli seuraava. Navigointiosio sai 0.25 saanti-, 0.22 tarkkuus- ja 0.68 todennäköisyyspistettä. Virheiden ennaltaehkäisy sai 0.01 saanti- ja 0.44 tarkkuuspistettä. Väärien ehdotusten osalta tuloksena oli 1.0 saanti- ja 0.984 tarkkuuspistettä. Yleisesti otettavana tuloksena siis oli se, että jEdit:n kohdalla työkalusta oli eniten hyötyä toisessa testitilanteessa eli kun keskityttiin etsimään muutoksia tiedostotasolla.

Livshits ja Zimmermann (2005) käsittelevät jEdit:n versionhallintatietoja kehittämällään `DynaMine`-työkalulla. Työkalu toimii funktiotasolla ja louhii versionhallinnasta todennäköisiä funktioiden käyttö- ja virhemalleja. Esimerkiksi tiettyä funktiokutsua voidaan käyttää tiedoston avaamiseen sen sisällön lukemista tai uuden sisällön tallentamista varten. Kun tiedoston käsittely on lopetettu tulee se sulkea käyttämällä tiettyä toista funktiokutsua. Nä-

mä funktiokutsut muodostavat siis tässä tapauksessa parin ja virhemalli on sellainen, jossa on käytetty vain toista näistä funktiokutsuista. Tiedonlouhinnassa työkalu käyttää Apriori-algoritmia, jota on kehitetty skaalautumaan paremmin heidän käyttöönsä. Työkalu käyttää dynaamista tekniikkaa etsien virhemalleja sovelluksen suoritusaikana. Tämän tekniikan avulla saadaan parempi skaalautuvuus tutkittaviin suuren määrän lähdekoodia omaaviin sovelluksiin, sillä suurin osa tutkittavista malleista on hajautunut monien funktioiden alle, jolloin mallien analysointikin täytyy suorittaa funktiotasolla ja staattisin analysointimentelmin tähän kuluisi runsaasti aikaa. Toinen etu dynaamisen analysoinnin käytöstä on löydettyjen mallien validointi. Versionhallintatiedoista louhittujen mallien voidaan tarkastaa kuuluvan usein käytettyihin malleihin tarkastamalla kuinka monta kertaa kukin malli esiintyy sovelluksen suoritusaikana.

Jos löytyy malleja, jotka esiintyvät suoritusaikana useita kertoja, mutta virheellisesti käytettynä vain yksittäisiä kertoja, kyseessä on todennäköisesti sellainen virhemalli, joka ei ole kovin kiinnostava. Myös työkalun ilmoittamista virhemalleista, jotka eivät todellisuudessa olekaan virhemalleja, päästään eroon dynaamista tekniikkaa käyttäen, koska testattavan sovelluksen suoritusaikana esiintyvät virhemallit todella ovat olemassa ja tapahtuvat. Testiaineistona jEdit:n osalta oli vuonna 2000 silloisen CVS-versionhallintajärjestelmän 144 495 viimeisintä revisiota. Työkalun avulla tutkittiin funktiopareja sekä klassista että korjaavaa tiedonlouhintamenetelmiä käyttäen. Klassinen menetelmä löysi jEdit:n versionhallintatiedoista 13 paria ja korjaava menetelmä 8 paria. Korjaavassa menetelmässä on käytetty kehittynyttä versiota Apriori-algoritmista, jolloin muun muassa kohinaa on saatu vähennettyä jättämättä huomioitta hyvin usein Java-ohjemointikelessä käytetyt funktiokutsut. Tämä optimointi on myös nopeuttanut tiedonlouhintaprosessia. Tarkemmin klassisen ja korjaavan menetelmän eroavaisuuksista ja tuloksista löytyy tietoa artikkelin taulukosta 5 (*Figure 5*).

Vastaavissa aiemmissa tiedonlouhintatekniikoin toteutetuissa tutkimuksissa on jEdit-sovelluksen versionhallintatiedoista siis löytynyt virhemalleja ja näiden perusteella on myös pystytty osoittamaan parannusehdotuksia. Täten sovelluskehittäjät ovat saaneet tietoa millaisissa tilanteissa tietyt virhemallit syntyvät ja miten sovellusta edelleen kehitettäessä näiden virhemallien käyttöä voisi välttää. Varsinkin suuria määriä lähdekoodia sisältävien sovellusten kohdalla tämä on tärkeä tieto kehitysprojektin sujumisen ja määrätyissä resursseissa pysy-

misen kannalta.

5.2 Malli ohjelmistojen muutoksille

Isokokoisissa ohjelmistoprojekteissa lähdekoodia tuotetaan yleensä iteraatioissa, ensin peruskomponentit ja niiden toiminnallisuudet ja tämän jälkeen lisätään komponentteja ja lisätoiminnallisuuksia tarpeen mukaan. Tämä näkyy versionhallinnassa siten, että projektin alkuvaiheessa luodaan eniten uusia tiedostoja ja loppua kohden näiden olemassa olevien tiedostojen sisältöä muutetaan. Muutokset voivat johtua muun muassa joko tarvittavan toiminnallisuuden lisäämisestä, lähdekoodin tarkemman kommentointitarpeen vuoksi, virheiden korjauksesta, lähdekoodin siistimisestä, tai tarpeesta saada eri komponentit toimimaan yhdessä. Muutoksia tulee, kun monet sovelluskehittäjät vaikuttavat projektiin ja osa kehittäjistä voi myös vaihtua projektin aikana. Varsinkin tällaisessa tilanteessa monta eri kehittäjää tulee käsittelemään samaa tiedostoa versionhallinnassa ja inhimilliset virheet ovat mahdollisia. Nykyisin projektiaikataulutkin ovat melko hektisiä, joten virheiden syntyminen on todennäköistä ja täten pieniä muutoksia tulee runsaasti virheenkorjausten vuoksi.

Erilaisia muutosskenaarioita ovat siis toiminnallisuuden lisääminen, toiminnallisuuden poistaminen, virheenkorjaus, lähdekoodin siistiminen, ja kokonainen uudelleenkirjoitus. Toiminnallisuuden lisäämisellä tarkoitetaan useiden rivien lisäämistä lähdekooditiedostoon ja tätä tehdään projektin kuluessa hyvin usein. Toiminnallisuuden poistamisella tarkoitetaan lisätyn toiminnallisuuden eli useiden rivien poistamista lähdekooditiedostosta, tätä tapahtuu projektin kuluessa harvoin. Virheenkorjauksella tarkoitetaan yksittäisten rivien muuttamista lähdekooditiedostossa ja tätä tapahtuu projektin kuluessa usein varsinkin projektin loppuvaiheessa. Lähdekoodin siistimisellä voidaan optimoida ja yksinkertaistaa koodia, joka näkyy tiedostotasolla yleensä rivimäärän vähenemisenä ja muutamien rivien muutoksena niillä kohdin, missä koodia on siistitty. Tätä suoritetaan projektin aikana jonkin verran. Kokonainen uudelleenkirjoitus tarkoittaa sitä, kun lähdekoodin osaa, esimerkiksi yksittäistä komponenttia, ei kannata enää muokata vaan se kehitetään alusta alkaen uudelleen. Tätä tapahtuu projektin edetessä hyvin harvoin.

Fokusoitaessa lähdekoodin muutosten ominaisuuksiin Purushothaman ja Perry (2005) ovat

selvittäneet, että todennäköisyys uusien virheiden synnylle muutettaessa ainoastaan yhtä riviä lähdekooditiedostossa on alle 4 % ja lähes 50 % kaikista sovellusprojektin aikana tehdyistä muutoksista versionhallintatietoihin on pieniä muutoksia eli suurimmillaan noin 10 rivin mittainen muutoksia. Lähes 10 % kaikista ylläpidollisista muutoksista ovat ainoastaan yhden rivin mittaisia muutoksia. Lähes 40 % virheenkorjauksista johtuvista muutoksista aiheuttivat lisää virheitä, mutta suurimmillaan 10 rivin lähdekoodin poistaminen ei aiheuttanut virheitä. Muutoksista aiheutuviin ilmiöihin vaikuttavat johtuuko muutos lähdekoodirivien lisäämisestä, poistamisesta vai muokkaamisesta sekä kuinka monen rivin mittainen muutos on. (Purushothaman ja Perry 2005)

Esitettyä mallia voidaan demonstroida simuloidulla aineistolla, jonka rakenne tunnetaan tarkasti. Simuloitu malli on luotu tekemällä testiaineisto käyttäen keksittyjä tiedostoja ja todennäköisyyksiä sille kuinka usein yhtä tiedostoa muutettaessa muita tiedostoja tulee muuttaa samalla kertaa. Tiedostot ovat `Config.java`, `Editor.java`, `Constants.java`, `Export.java` ja `Import.java`. Config.java-tiedostoa muutettaessa ei milloinkaan muuteta samanaikaisesti mitään muuta tiedostoa. Jokaista tiedostoa muutettaessa todennäköisyydet muiden tiedostojen samanaikaisille muutoksille ovat esitetty taulukossa 24. Testiaineisto sisältää 10 000 joukkoa. Tämän aineiston louhimiseen käytetään Borgeltin (2013) kehittämää Apriori-algoritmia.

	Config.java	Editor.java	Export.java	Import.java	Constants.java
Config.java	1	0	0	0	0
Editor.java	0	1	0.2	0.5	0.7
Export.java	0	0	1	0	0
Import.java	0	0	0	1	0
Constants.java	0	0	0	0	1

Taulukko 24. Todennäköisyydet muiden tiedostojen muutoksille samaan aikaan tietyn tiedoston kanssa.

Tämän aineiston perusteella voidaan sanoa, että siitä löytyy korkeintaan kolme assosiaatiosääntöä, jotka kaikki liittyvät Editor.java-tiedoston muokkaamiseen, sillä se on ainoa tiedosto

jonka kanssa muokataan yhtä aikaa muita tiedostoja. Eniten esiintyvä eli suurimman tukiarvon omaavan säännön tulisi olla $Editor.java => Constants.java$, toiseksi eniten esiintyvän $Editor.java => Import.java$ ja vähiten esiintyvän säännön $Editor.java => Export.java$. Lopulliseen tuloksessa saatuun sääntöjen määrään vaikuttaa se kuinka korkeaa minimituki- ja uskottavuus-parametrin arvoja halutaan käyttää.

Louhitut useimmin esiintyvät joukot simuloidusta mallista ovat esitetty taulukossa 25. Tässä minimituki-parametrin arvoksi on valittu 5, jotta saadaan esimerkin omaisesti runsaasti tuloksia. Tuloksissa siis esimerkiksi Editor.java-tiedoston muutoksen yhteydessä muutetaan samalla sekä Import.java- että Constants.java-tiedostoa, tulee tämän joukon tueksi 6.99 eli tässä tapauksessa kyseinen yhdistelmä esiintyy simuloidun mallin aineistossa 6.99 %:n verran kaikista aineistossa esiintyvistä joukoista, kun taas esimerkiksi tiedosto Constants.java esiintyy 33.21 %:n verran kaikissa aineiston joukoissa.

Varsinainen assosiaatiosääntöjen louhimisen tulos on esitetty kuviossa 4. Louhiminen on toteutettu jälleen Apriori-algoritmilla, käyttäen minimituki-parametrina arvoa 20 ja uskottavuus-parametrina arvoa 30. Tuloksena on siis kaksi sääntöä:

- Tiedostoa Editor.java muokattaessa muokataan samalla myös tiedostoa Import.java
- Tiedostoa Editor.java muokattaessa muokataan samalla myös tiedostoa Constants.java

Kuviossa olevien sääntöjen perässä esitetyt sulkeissa olevat luvut esittävät kyseisen säännön tuki ja uskottavuus -arvoja. Tuloksissa ei mainita tiedostoja Export.java ja Config.java ollenkaan siksi, että tiedostojen Editor.java ja Export.java yhtäaikaisella muutoksella on liian pieni uskottavuus-parametrin arvo (16.556) käytettyyn minimiarvoon verrattuna ja täten se karsiutuu tuloksista. Ja Config.java-tiedosto ei muodosta mitään sääntöä muiden tiedostojen kanssa sillä sitä muokattaessa mitään muuta tiedostoa ei muokata samanaikaisesti.

Tulokst siis pätevät ennakko-odotusten kanssa, sillä on saatu kaksi assosiaatiosääntöä yhden säännön jäädessä tulosten ulkopuolelle. Ja ulkopuolelle jääneellä säännöllä *Editor.java => Export.java* on tietenkin pienin tukiarvo, seuraavaksi suurin arvo on säännöllä *Editor.java => Import.java* ja kaikkein suurin säännöllä *Editor.java => Constants.java*, kuten pitikin olla. Syntyneitä assosisaatiosääntöjä voidaan hyödyntää ohjelmointivirheiden ehkäisemisessä. Ohjelmoijan tulisi tässä simuloidun mallin tapauksessa muokatessaan tiedostoa Edi-

tor.java tarkastaa aina myös tiedostot Import.java ja Constants.java tarvittavien muutosten varalta.

Joukko	**Joukon tukiarvo**
Import.java Constants.java	6.99
Editor.java Import.java Constants.java	6.99
Editor.java Import.java	9.71
Editor.java Constants.java	13.61
Editor.java	19.64
Config.java	20.14
Export.java	24.1
Import.java	30.22
Constants.java	33.21

Taulukko 25. Useimmin esiintyvät joukot simuloidusta mallista minimituki-parametrin ollessa 5.

```
+-- Associations
    +-- EDITOR.JAVA
        +-- Import.java (30.22, 32.131)
        +-- Constants.java (33.21, 40.9816)
```

Kuvio 4. Assosiaatiosäännöt simuloidusta mallista.

5.3 Aidon datan louhinta

Tutkimusaineistoksi otetaan 100 revisiota 100 revision askelluksin uusimmasta eli revisiosta 22955 alkaen, jolloin saadaan aineistoa kokonaisuudessaan laajemmalta aikaväliltä, kuin valittaessa esimerkiksi 100 uusinta revisiota. Tällöin vanhin aineisto on heinäkuulta 2008 ja

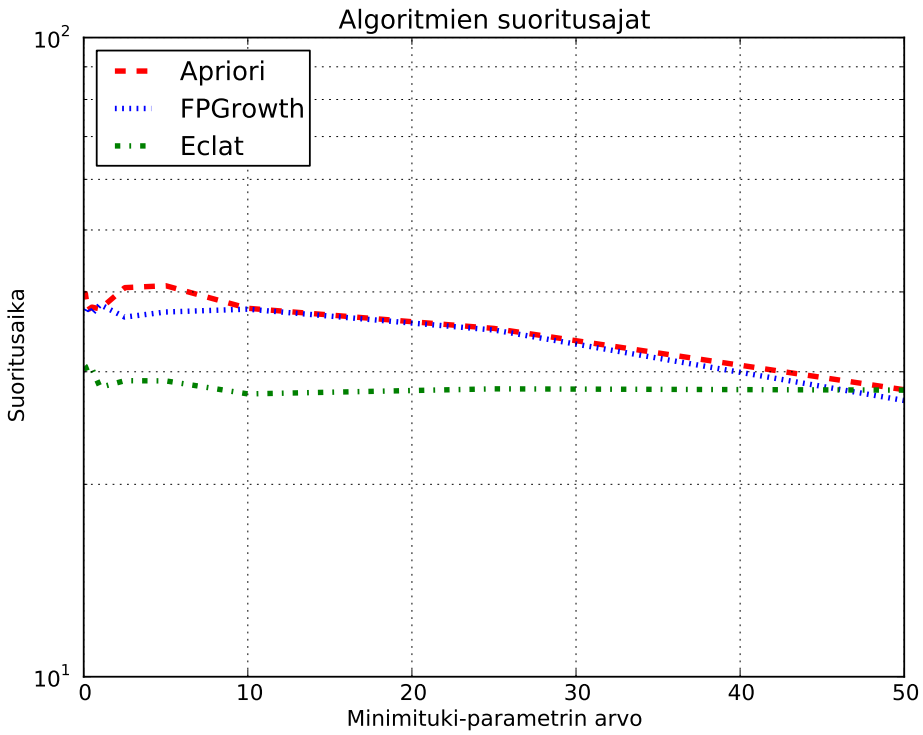
voidaan olettaa, että osa sovelluskehittäjistä on projektissa tällä välin vaihtunut vaikkakin aktiiviset kehittäjät olisivat pysyneet samana. Tästä syystä samojen henkilöiden tekemiä yhtäaikaisia muutoksia versionhallintaan pitäisi löytyä runsaasti. Aineiston saanti tapahtuu ottamalla SVN-yhteys osoitteeseen https://jedit.svn.sourceforge.net/svnroot/jedit jedit. Aineistosta selvitetään tiedonlouhintakeinoin, vaikuttaako jonkin tiedoston muokkaaminen muihin tiedostoihin eli mistä tiedostoista on versionhallintaan tallennettu uusi versio samaan aikaan ja toistuuko tämä samojen tiedostojen kohdalla pidemmällä aikavälillä tarkasteltuna. Aineistosta käsitellään vain java-tiedostot, muut tiedostot jätetään huomioimatta. Tutkimuksessa myös verrataan eri tiedonlouhinta-algoritmien suorituskykyä kyseisellä aineistolla.

Kahden tiedoston yhtäaikainen muokkaaminen ei välttämättä tarkoita molempiin tiedostoihin tehdyn muokkauksen tarkalleen samaan aikaan tehtyä päivitystä versionhallintaan. Esimerkiksi Zimmermann et al. (2005) määrittävät kahden tiedoston muokkauksen liittyvän toisiinsa mikäli molemmat tiedostot ovat saman tekijän päivittämät sekä tiedostojen päivitysaika eroaa toisistaan enintään 200 sekunnin verran. Ying et al. (2004) taas määrittävät termin vieläkin yksityiskohtaisemmin, tiedostojen päivitysten aikaero toisiinsa nähden saa olla enintään 180 sekuntia, molemmat tiedostot tulevat olla saman tekijän päivittämät sekä molemmissa tiedostoissa tulee olla annettu sama kommentti niiden versionhallintaan tehdyn päivityksen yhteydessä. Tässä tutkielmassa käytetään termiä kuten Ying et al. sen määrittelevät.

Tiedonlouhinta-algoritmeina käytetään Borgeltin (2013) kehittämiä Apriori-, FP-Growth- ja Eclat-algoritmeja. Java-kielellä luodaan omat ohjelmointikomponentit, joiden avulla saadaan versionhallintatieto muutettua edellä mainittujen algoritmien ymmärtämään muotoon.

Käytettyjen algoritmien suoritusaikoja verrattiin keskenään lähdeaineistosta useimmin esiintyviä joukkoja etsiessä eri minimituki-parametrin arvoja käyttäen. Käytetyt komennot ovat esitetty liitteessä A. Vertailun tulokset ovat kuviossa 5.

Kuviossa eri algoritmit ovat eritelty eri värein sekä erityylisin viivoin. X-akselilla on käytetty minimituki-parametrin arvo ja logaritmisella y-akselilla on algoritmin suoritusaika sekunteina. Vertailu on suoritettu 32-bittisellä AMD 1.7 GHz prosessorin omaavalla tietokoneella, jossa muistia on ollut käytössä 1 GB ja käyttöjärjestelmänä on toiminut Linux Ubuntu 12.04.



Kuvio 5. Käytettyjen algoritmien suoritusaikavertailu lähdeaineistosta useimmin esiintyviä joukkoja louhiessa.

Valtaosa käytetystä ajasta kului kaikissa algoritmeissa lähdeaineiston lukemiseen ja algoritmin käyttämän sisäisen tapahtumajoukkotietokannan luomiseen. Kaikkein vähiten kaikilla algoritmeilla kului aikaa tulostiedoston kirjoittamiseen. Apriori-algoritmi koostuu kuudesta yksilöidystä vaiheesta: lähdeaineston lukeminen, tapahtumien suodattaminen ja järjestäminen, tapahtumajoukkojen pienentäminen, tapahtumajoukkopuun luominen, osajoukkojen tarkastaminen ja tulostiedoston tekeminen. FP-Growth-algoritmissa on neljä yksilöityä vaihetta: lähdeaineston lukeminen, tapahtumien suodattaminen ja järjestäminen, FP-puun luominen ja tulostiedoston tekeminen. Eclat-algoritmissa yksilöityjä vaiheita on kolme kappaletta: lähdeaineiston lukeminen, tapahtumien suodattaminen ja kirjoittaminen sekä tulostiedoston tekeminen. Apriori on yleensä ottaen tehokkaampi isoilla lähdeaineistoilla kuin FP-Growth ja Eclat on yleensä ottaen tehokkaampi kuin FP-Growth lähdeaineiston koosta riippumatta. Tässä tapauksessa Apriori ja FP-Growth suoriutuvat tiedonlouhinnasta lähestulkoon samassa ajassa. Lähdeaineiston laajuus on kuitenkin pieni verrattuna joihinkin suuriin käytännönläheisiin tutkimuksiin ja siten sillä ei voi selittää eroja näiden algoritmien kesken. Varsinkin, kun suurin osa suoritusajasta kului lähdeaineiston lukemiseen ja molempien algoritmien lähdeaineiston lukuprosessi on toteutettu samalla tavalla. Eclat sen sijaan osoittaa tehokkuutta muihin käytettyihin algoritmeihin verrattuna. Eclat-algoritmin tehokkuus verrattuna FP-Growth-algoritmiin on se, että FP-Growth-algoritmi käsittelee tapahtumajoukkoja sisäisesti monimutkaisena tietorakenteena ja tästä syystä joutuu käyttämään useita viittauksia alkuperäiseen tapahtumajoukkoon, kun Eclat käyttää tietorakennetta, joka sallii yksinkertaisen ja nopean leikkauksen tarvittaessa tiettyyn tapahtumajoukkoon.

Assosiaatiosääntöjä Apriori-algoritmi löysi lähdeaineistosta 79 kappaletta, kun minimitukiparametrina käytettiin arvoa 10 ja uskottavuus-parametrina arvoa 15. Kuviossa 6 näytetään osa näistä löydetyistä säännöistä. Monet löytyneet säännöt vaikuttavat olevan pieniä, 1-5 rivin mittaisia muutoksia molempiin säännössä oleviin tiedostoihin. Välillä on tehty virhekorjauksia, jotka seuraavassa yhteydessä onkin peruttu tai niitä on muutettu lisää, sillä virhe on pysynyt sovelluksessa korjauksesta huolimatta.

Tarkemmin katsottuna esimerkiksi assosiaatiosääntö *org/gjt/spj/jedit/jEdit.java* $=>$ *org/jedit/core/MigrationService.java* liittyy revisioon numero 22142. Molempia tiedostoja on muokattu sekunnilleen samaan aikaan saman käyttäjän toimesta ja niiden versionhallin-

```
+-- Associations
    +-- org/gjt/sp/jedit/BeanShell.java
    |   +-- org/gjt/sp/jedit/io/CharsetEncoding.java(15.22, 16.371)
    |   +-- org/gjt/sp/jedit/Buffer.java (16.21, 16.571)
    |   +-- org/gjt/sp/jedit/browser/VFSBrowser.java (11.16, 15.082)
    |   +-- org/gjt/sp/jedit/browser/VFSFileChooserDialog.java (13.69, 15.216)
    |   +-- org/gjt/sp/jedit/browser/VFSFileNameField.java (13.52, 15.9162)
    |   +-- org/gjt/sp/jedit/gui/CloseDialog.java (11.09, 17.2801)
    |   +-- org/gjt/sp/jedit/gui/StatusBar.java (13.91, 16.917)
    |   +-- org/gjt/sp/jedit/io/VFS.java (17.12, 20.121)
    |   +-- org/gjt/sp/jedit/io/VFSManager.java (16.59, 19.2105)
    |   +-- org/gjt/sp/jedit/jEdit.java (18.90, 24.1286)
    |   +-- org/gjt/sp/jedit/search/HyperSearchRequest.java (10.51, 16.124)
    |   +-- org/gjt/sp/jedit/search/SearchAndReplace.java (10.22, 15.822)
    |   +-- org/gjt/sp/util/TaskManager.java (15.21, 19.002)
    |   +-- org/gjt/sp/util/ThreadUtilities.java (11.75, 16.296)
    +-- org/gjt/sp/jedit/MiscUtilities.java
    |   +-- org/gjt/sp/jedit/bufferio/BufferSaveRequest.java (10.12, 15.019)
    |   +-- org/gjt/sp/jedit/io/VFS.java (13.15, 17.521)
    +-- org/gjt/sp/jedit/Buffer.java
    |   +-- org/gjt/sp/jedit/EditPane.java (16.98, 22.0039)
    +-- org/jedit/options/CombinedOptions.java
    |   +-- org/jedit/options/TabbedOptionDialog.java (17.11, 22.0911)
    +-- org/gjt/sp/jedit/gui/BufferSwitcher.java
    |   +-- org/gjt/sp/jedit/pluginmgr/PluginManager.java (16.08, 17.251)
    +-- org/gjt/sp/jedit/io/VFSManager.java
    |   +-- org/gjt/sp/util/WorkThreadPool.java (11.38, 15.217)
    +-- org/gjt/sp/jedit/bufferio/BufferAutosaveRequest.java
    |   +-- org/gjt/sp/jedit/bufferio/BufferInsertRequest.java (17.95, 21.197)
    |   +-- org/gjt/sp/jedit/bufferio/BufferLoadRequest.java (16.81, 21.0028)
    |   +-- org/gjt/sp/jedit/bufferio/BufferSaveRequest.java (17.03, 20.9621)
    |   +-- org/gjt/sp/jedit/gui/IOProgressMonitor.java (12.67, 15.0772)
    |   +-- org/gjt/sp/jedit/io/VFS.java (10.16, 15.081)
    |   +-- org/gjt/sp/jedit/io/VFSManager.java (10.23, 15.4901)
    |   +-- org/gjt/sp/util/WorkRequest.java(12.95, 17.683)
    |   +-- org/gjt/sp/util/WorkThread.java(14.12, 18.006)
    |   +-- org/gjt/sp/util/WorkThreadFactory.java(11.55, 17.251)
    |   +-- org/gjt/sp/util/WorkThreadPool.java(12.01, 16.9915)
    +-- org/gjt/sp/jedit/jEdit.java
    |   +-- org/jedit/core/MigrationService.java (13.27, 17.704)
    |   +-- org/jedit/migration/CheckFileStatus.java (15.02, 21.129)
    |   +-- org/jedit/migration/OneTimeMigrationService.java (13.25, 17.699)
```

Kuvio 6. Osa lähdeaineistosta löydetyistä assosiaatiosäännöistä minimituki-parametrin ollessa 10 ja uskottavuus-parametrin ollessa 15.

taan tapahtuneen päivityksen yhteydessä on annettu sama päivityskommentti, joka kertoo kyseessä olleen virheenkorjauksen virheelle, joka tapahtuu sovellusta päivitettäessä. jEdit.java-tiedoston muutos on kahden lähdekoodirivin mittainen, ensin liitetään uusi java-paketti tämän luokan käyttöön, ja toinen lisätty koodirivi on kerran kutsuttavan muutospalvelun suorittaminen. MigrationService.java-tiedostoon on vain lisätty samassa yhteydessä yksi kommenttirivi lisää. Assosiaatiosääntö *org/gjt/spj/jedit/jEdit.java => org/jedit/migration/CheckFileStatus.java* liittyy revision 22142 yhteydessä edelliseen sääntöön siten, että CheckFileStatus.java-tiedostoa on muokattu samaan aikaan saman tekijän toimesta samalla päivityskommentilla. Kyseisessä revisiossa tämä tiedosto on tallennettu versiohistoriaan ensimmäistä kertaa, josta syystä siinä on useamman kuin 1-5 rivin muokkaamisesta kyse.

jEdit-sovelluksen versionhallinnasta siis löytyy runsaasti samalla kertaa useampaan tiedostoon tehtäviä pieniä muutoksia joko virheenkorjauksen tai uuden ominaisuuden lisäämisen yhteydessä ja sen sovelluskehitystä voisi tehostaa ennakoimalla yksinkertaiset virheenkorjaustarpeet ohjelmistoaineiston louhintaa käyttämällä.

5.4 Tulosten analyysi

Tuloksina löytyy siis monia tiedostoja, joita sama käyttäjä on päivittänyt versionhallintaan samalla kertaa jonkin muun tiedoston kanssa. Joidenkin tiedostojen kohdalla muokkaustarve on ollut jonkin uuden Java-paketin mukaan ottaminen, jota ei voi tulkita virheelliseksi uuden toiminnallisuuden lisäämisen yhteydessä. Uuden toiminnallisuuden lisäämiseen kuuluu myös se, kun tiedosto tallennetaan versionhallintaan ensimmäistä kertaa, joka lasketaan myös muutokseksi. Kuitenkin myös yksittäisiä virheenkorjaustoimenpiteitä löytyi tuloksista.

Useilla löydetyillä säännöillä on verrattain pieni tuki- ja uskottavuusarvo, josta voidaan päätellä, että säännöt eivät ole kovinkaan vahvoja. Tämä tarkoittaa sitä, että sääntöihin liittyvät tiedostojen muokkaustoimenpiteet ovat olleet enemmän yksittäisiä toimenpiteitä, kuin pidemmällä aikavälillä samoihin tiedostoihin usein kohdistuvia virheenkorjaustoimenpiteitä.

Useasti toistuvia virheitä ja niistä syntyneitä korjaustoimenpiteitä samojen tiedostojen kohdalla pidemmällä aikavälillä tarkasteltuna en aineistosta löytänyt. Syynä tähän voi tosin olla se, että lähtöaineisto on otettu versionhallinnasta 100 revision välein. Otettujen revisioiden

väliin on saattanut jäädä muutostietoa, joka osoittautuisi toistuvaksi virheenkorjaustoimenpiteeksi tiettyjen tiedostojen kohdalla.

jEdit-sovelluksen *Tracker*-tietokannan huomioiminen olisi ollut oleellista. Tracker-tietokanta on muutostietokanta, johon sovelluksen käyttäjät voivat kertoa havaitsemistaan virheistä tai halutuista uusista toiminnoista ja jossa sovelluskehittäjät voivat kertoa tehdyistä muutoksista sovellukseen. Erityisesti tämän muutostietokannan virheosion tietojen seuraaminen ja yhdistäminen versionhallintatietoihin olisi ratkaissut erottamaan selkeästi mitkä muutokset tiedostoissa ovat olleet virheenkorjaustoimenpiteitä ja mitkä esimerkiksi uuden toiminnallisuuden lisäämistä sovellukseen. Tätä ei kuitenkaan tutkielman lähtötilanteessa huomioitu ja lopulta aikaresurssit eivät riittäneet näiden tietojen käsittelyyn.

6 Yhteenveto

Tietoyhteiskunnan myötä tallennetun tiedon määrä on kasvanut runsasta vauhtia, mutta samalla merkitsevän tiedon osuus kaikesta tallennetusta tiedosta on vähentynyt ja haasteen onkin muodostanut se, miten saada tuo merkitsevä tieto esiin kaiken muun tiedon seasta. Tähän haasteeseen vastaa tiedonlouhinta. Se on monivaiheinen prosessi, jonka avulla lähdeaineistosta voidaan löytää uutta mielenkiintoista tietoa, jonka löytäminen muilla keinoin olisi lähes mahdotonta. Esimerkiksi internet-kaupoissa tiedonlouhintaa käyttäen voidaan ehdottaa asiakkaalle jotakin tuotetta, kun tiedetään mitä tuotteita hänellä jo on ostoskorissa. Tällöin puhutaan assosiaatiosäännöistä eli voidaan kartoittaa, mitä tuotteita useimmat asiakkaat ovat ostaneet tietyn tuotteen ostamisen yhteydessä ja sen tiedon myötä uusien tuotteiden suosittelu asiakkaalle on helppoa.

Tutkielmassa tutkittiin jEdit-sovelluksen SVN-versionhallinnassa olevia tiedostoja, mitä niistä muokataan yhtäaikaa muiden tiedostojen muokkauksen yhteydessä ja toistuuko tämä samojen tiedostojen kohdalla pidemmällä tarkasteluvälillä. Aineistosta löydettiin useita tiedostoja, joita muokataan yhtäaikaa ja kuhunkin tiedostoon tehdään vain pieniä muutoksia kerrallaan ja täten ohjelmistoaineiston louhinnalla voitaisiin ehkäistä sovelluskehittäjistä johtuvat inhimilliset virheet ja niistä johtuvat korjaustoimenpiteet kyseisen sovelluksen kohdalla.

Tutkielman heikkoutena on versionhallinnassa kokonaisuudessaan oleviin revisiomääriin verrattuna melko pieni otosjoukko lähdeaineistoksi. Tutkielmassa ei myöskään kiinnitetä huomiota itse tiedostojen sisältöön eikä metadataan muutoin kuin tiedoston viimeisen muokkaajan, muokkausajan ja muokkauksen yhteydessä jätetyn kommentin osalta. jEdit-sovelluksen versionhallinta on varsinkin sen lisäosien osalta siirtynyt Git-versionhallintaan, joten tutkielman aiheessa olisi syventämisen varaa tutkia syvemmin erityisesti sovellukseen liittyviä lisäosia, verrata niitä sovelluksen virhetietokantaan ja mennä tutkimuksessa vielä tiedostotasolta syvemmälle funktiotasolle.

Lähteet

Adamo, Jean-Marc. 2001. *Data mining for association rules and sequential patterns: sequential and parallel algorithms.* Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN: 0-387-95048-6.

Amir, Amihood, Yonatan Aumann, Ronen Feldman ja Moshe Fresko. 2005. "Maximal Association Rules: A Tool for Mining Associations in Text". *J.Intell.Inf.Syst.* 25 (3): 333–345. Viitattu viitattu 17. marraskuuta 2005. `http://portal.acm.org/citation.cfm?id=1107361.1107392`.

Borgelt, Christian. 2013. *Christian Borgelt's Web Pages - Software.* Viitattu viitattu 17. maaliskuuta 2013. `http://www.borgelt.net/software.html`.

Chen, Ming-Syan, Jiawei Han ja Philip S. Yu. 1996. "Data Mining: An Overview from a Database Perspective". *IEEE Trans.on Knowl.and Data Eng.* 8, numero 6 (): 866–883. Viitattu viitattu 15. marraskuuta 2012. `http://dx.doi.org/10.1109/69.553155`.

Ching, Wai Ki, ja Michael Kwok-Po Ng. 2003. *Advances in Data Mining and Modeling.* 181. World Scientific Press. ISBN: 981-238-354-9.

Cohen, Edith, Mayur Datar, Shinji Fujiwara, Aristides Gionis, Piotr Indyk, Rajeev Motwani, Jeffrey D. Ullman ja Cheng Yang. 2001. "Finding Interesting Associations without Support Pruning". *IEEE Trans.on Knowl.and Data Eng.* 13, numero 1 (): 64–78. Viitattu viitattu 15. marraskuuta 2012. `http://dx.doi.org/10.1109/69.908981`.

Dunham, Margaret H., Yongqiao Xiao, Le Gruenwald ja Zahid Hossain. 2000. *A SURVEY OF ASSOCIATION RULES.* Viitattu viitattu 15. marraskuuta 2012. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.1602`.

Elmasri, Ramez A., ja Shankrant B. Navathe. 1999. *Fundamentals of Database Systems.* 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0805317554.

Free Software Foundation. 1991. *GNU General Public License v2.0.* Viitattu viitattu 15. marraskuuta 2012. `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html`.

Free Software Foundation. 2007. *GNU General Public License v3.0.* Viitattu viitattu 15. marraskuuta 2012. http://www.gnu.org/licenses/gpl.html.

Fukuda, Takeshi, Yasukiko Morimoto, Shinichi Morishita ja Takeshi Tokuyama. 1996. "Data mining using two-dimensional optimized association rules: scheme, algorithms, and visualization". Teoksessa *Proceedings of the 1996 ACM SIGMOD international conference on Management of data,* 13–23. SIGMOD '96. Montreal, Quebec, Canada: ACM. ISBN: 0-89791-794-4, viitattu viitattu 15. marraskuuta 2012. http://doi.acm.org/10.1145/233269.233313.

Han, Jiawei. 2005. *CS412: Introduction to Data Mining.* Viitattu viitattu 1. kesäkuuta 2006. http://cs.uiuc.edu/class/fa05/cs412/schedule.htm.

Han, Jiawei, ja Micheline Kamber. 2000. *Data mining: concepts and techniques.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1-55860-489-8.

Hand, David J., Padhraic Smyth ja Heikki Mannila. 2001. *Principles of data mining.* Cambridge, MA, USA: MIT Press. ISBN: 0-262-08290-X, 9780262082907.

Hegland, Markus. 2003. "Advanced lectures on machine learning". Luku Algorithms for association rules teoksessa, toimittanut Shahar Mendelson, Alex Smola ja er J., 226–234. New York, NY, USA: Springer-Verlag New York, Inc. ISBN: 3-540-00529-3, viitattu viitattu 15. marraskuuta 2012. http://dl.acm.org/citation.cfm?id=863714.863722.

Holt, John D., ja Soon M. Chung. 2001. "Multipass algorithms for mining association rules in text databases". *Knowl.Inf.Syst.* 3, numero 2 (): 168–183. Viitattu viitattu 15. marraskuuta 2012. http://dx.doi.org/10.1007/PL00011664.

———. 2002. "Mining association rules using inverted hashing and pruning". *Inf.Process.Lett.* 83, numero 4 (): 211–220. Viitattu viitattu 15. marraskuuta 2012. http://dx.doi.org/10.1016/S0020-0190(01)00330-1.

jEdit Community. 2007. *jEdit - Programmer's Text Editor.* Viitattu viitattu 8. joulukuuta 2007. http://www.jedit.org/.

Juhola, Martti. 2006. *Tiedonlouhinta 2005.* Viitattu viitattu 30. elokuuta 2006. http://www.cs.uta.fi/tl/.

Kantardzic, Mehmed. 2002. *Data Mining: Concepts, Models, Methods and Algorithms.* New York, NY, USA: John Wiley & Sons, Inc. ISBN: 0471228524.

Liu, Bing, Wynne Hsu, Shu Chen ja Yiming Ma. 2000. "Analyzing the Subjective Interestingness of Association Rules". *IEEE Intelligent Systems* 15, numero 5 (): 47–55. Viitattu viitattu 15. marraskuuta 2012. http://dx.doi.org/10.1109/5254.889106.

Liu, Bing, Wynne Hsu ja Yiming Ma. 1999. "Mining association rules with multiple minimum supports". Teoksessa *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining,* 337–341. KDD '99. San Diego, California, United States: ACM. ISBN: 1-58113-143-7, viitattu viitattu 15. marraskuuta 2012. http://doi.acm.org/10.1145/312129.312274.

Livshits, Benjamin, ja Thomas Zimmermann. 2005. "DynaMine: finding common error patterns by mining software revision histories". *SIGSOFT Softw. Eng. Notes* (New York, NY, USA) 30, numero 5 (): 296–305. ISSN: 0163-5948, viitattu viitattu 3. huhtikuuta 2013. doi:10.1145/1095430.1081754. http://doi.acm.org/10.1145/1095430.1081754.

Mens, Tom, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld ja Mehdi Jazayeri. 2005. "Challenges in Software Evolution":13–22. Viitattu viitattu 3. huhtikuuta 2013. doi:10.1109/IWPSE.2005.7. http://doi.ieeecomputersociety.org/10.1109/IWPSE.2005.7.

Mitchell, Tom M. 1999. "Machine learning and data mining". *Commun.ACM* 42, numero 11 (): 30–36. Viitattu viitattu 15. marraskuuta 2012. http://doi.acm.org/10.1145/319382.319388.

Mooney, Raymond J., ja Razvan Bunescu. 2005. "Mining knowledge from text using information extraction". Teoksessa *SIGKDD Explorations,* 3–10. Viitattu viitattu 6. marraskuuta 2005. http://www.acm.org/sigs/sigkdd/explorations/issues/7-1-2005-06/2-Mooney.pdf.

Nahm, Un Yong, ja Raymond J. Mooney. 2002. “Mining soft-matching association rules”. Teoksessa *Proceedings of the eleventh international conference on Information and knowledge management,* 681–683. CIKM ’02. McLean, Virginia, USA: ACM. ISBN: 1-58113-492-4, viitattu viitattu 15. marraskuuta 2012. http://doi.acm.org/10.1145/584792.584918.

Park, Jong Soo, Ming-Syan Chen ja Philip S. Yu. 1997. “Using a Hash-Based Method with Transaction Trimming for Mining Association Rules”. *IEEE Trans.on Knowl.and Data Eng.* 9, numero 5 (): 813–825. Viitattu viitattu 15. marraskuuta 2012. http://dx.doi.org/10.1109/69.634757.

Pinto, Helen. 2001. *Multi-dimensional Sequential Pattern Mining.* Viitattu viitattu 24. elokuuta 2006. http://www-sal.cs.uiuc.edu/~hanj/pubs/theses.html.

Purushothaman, Ranjith, ja Dewayne E. Perry. 2005. “Toward Understanding the Rhetoric of Small Source Code Changes”. *IEEE Trans. Softw. Eng.* (Piscataway, NJ, USA) 31, numero 6 (): 511–526. ISSN: 0098-5589, viitattu viitattu 10. huhtikuuta 2013. doi:10.1109/TSE.2005.74. http://dx.doi.org/10.1109/TSE.2005.74.

Pyle, Dorian. 1999. *Data preparation for data mining.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1-55860-529-0.

Rantzau, Ralf. 1997. *Extended Concepts for Association Rule Discovery.* Viitattu viitattu 21. elokuuta 2006. http://citeseer.ist.psu.edu/rantzau97extended.html.

Silverstein, Craig, Sergey Brin ja Rajeev Motwani. 1998. “Beyond Market Baskets: Generalizing Association Rules to Dependence Rules”. *Data Min.Knowl.Discov.* 2, numero 1 (): 39–68. Viitattu viitattu 15. marraskuuta 2012. http://dx.doi.org/10.1023/A:1009713703947.

Song, Qinbao, Martin J. Shepperd, Michelle Cartwright ja Carolyn Mair. 2006. “Software Defect Association Mining and Defect Correction Effort Prediction”. *IEEE Trans. Software Eng.* 32 (2): 69–82. Viitattu viitattu 3. huhtikuuta 2013. doi:10.1109/TSE.2006.19. http://doi.ieeecomputersociety.org/10.1109/TSE.2006.19.

Srikant, Ramakrishnan. 1996. *Fast algorithms for mining association rules and sequential patterns.* AAI9708697. Viitattu viitattu 31. elokuuta 2006. http://www.rsrikant.com/papers/thesis.pdf.

Srikant, Ramakrishnan, ja Rakesh Agrawal. 1996. "Mining quantitative association rules in large relational tables". Teoksessa *Proceedings of the 1996 ACM SIGMOD international conference on Management of data,* 1–12. SIGMOD '96. Montreal, Quebec, Canada: ACM. ISBN: 0-89791-794-4, viitattu viitattu 15. marraskuuta 2012. http://doi.acm.org/10.1145/233269.233311.

Ullman, Jeffrey D., Hector Garcia-Molina ja Jennifer Widom. 2001. *Database Systems: The Complete Book.* 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0130319953.

Wang, Ke, Yu He ja Jiawei Han. 2003. "Pushing Support Constraints Into Association Rules Mining". *IEEE Transactions on Knowledge and Data Engineering* 15:642–658. Viitattu viitattu 24. elokuuta 2006. http://citeseer.ist.psu.edu/455941.html.

Witten, Ian H., ja Eibe Frank. 2000. *Data mining: practical machine learning tools and techniques with Java implementations.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1-55860-552-5.

Wur, Suh-Ying, ja Yungho Leu. 1999. "An Effective Boolean Algorithm for Mining Association Rules in Large Databases". Teoksessa *Proceedings of the Sixth International Conference on Database Systems for Advanced Applications,* 179–186. DASFAA '99. Washington, DC, USA: IEEE Computer Society. ISBN: 0-7695-0084-6, viitattu viitattu 15. marraskuuta 2012. http://dl.acm.org/citation.cfm?id=646712.703323.

Yang, Don-Lin, Ching-Ting Pan ja Yeh-Ching Chung. 2001. "An Efficient Hash-Based Method for Discovering the Maximal Frequent Set". Teoksessa *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development,* 511–516. COMPSAC '01. Washington, DC, USA: IEEE Computer Society. ISBN: 0-7695-1372-7, viitattu viitattu 15. marraskuuta 2012. http://dl.acm.org/citation.cfm?id=645983.675101.

Yen, Show-Jane, ja Arbee L. P. Chen. 2001. “A Graph-Based Approach for Discovering Various Types of Association Rules”. *IEEE Trans.on Knowl.and Data Eng.* 13, numero 5 (): 839–845. Viitattu viitattu 15. marraskuuta 2012. http://dx.doi.org/10.1109/69.956106.

Ying, Annie T. T., Gail C. Murphy, Raymond Ng ja Mark C. Chu-Carroll. 2004. “Predicting Source Code Changes by Mining Change History”. *IEEE Trans. Softw. Eng.* (Piscataway, NJ, USA) 30, numero 9 (): 574–586. ISSN: 0098-5589, viitattu viitattu 3. huhtikuuta 2013. doi:10.1109/TSE.2004.52. http://dx.doi.org/10.1109/TSE.2004.52.

Yun, Hyunyoon, Danshim Ha, Buhyun Hwang ja Keun Ho Ryu. 2003. “Mining association rules on significant rare data using relative support”. *Journal of Systems and Software* 67 (3): 181–191. Viitattu viitattu 15. marraskuuta 2012. http://www.sciencedirect.com/science/article/pii/S0164121202001280.

Zaki, Mohammed J. 2000. “Scalable Algorithms for Association Mining”. *IEEE Trans.on Knowl.and Data Eng.* 12, numero 3 (): 372–390. Viitattu viitattu 15. marraskuuta 2012. http://dx.doi.org/10.1109/69.846291.

———. 2001. “SPADE: An Efficient Algorithm for Mining Frequent Sequences”. *Mach.Learn.* 42, **numbers** 1-2 (): 31–60. Viitattu viitattu 15. marraskuuta 2012. http://dx.doi.org/10.1023/A:1007652502315.

Zimmermann, Thomas, Peter Weißgerber, Stephan Diehl ja Andreas Zeller. 2005. “Mining Version Histories to Guide Software Changes”. *IEEE Trans. Software Eng.* 31 (6): 429–445. Viitattu viitattu 3. huhtikuuta 2013. doi:10.1109/TSE.2005.72. http://doi.ieeecomputersociety.org/10.1109/TSE.2005.72.

Liitteet

A Käytettyjen ohjelmien suorituskomennot

Tässä liitteessä esitetään yksityiskohtaiset suorituskomennot käytetyistä ohjelmista.

Paikallisen kopion luominen jEdit-sovelluksen SVN-versionhallinnasta:

```
svn co https://jedit.svn.sourceforge.net/svnroot/jedit/jEdit
hakemisto
```

jossa *hakemisto* on paikallisen koneen hakemisto, jonne kopio halutaan luoda.

Useimmin esiintyvien tietojoukkojen etsiminen testaineistosta Apriori-algoritmia käyttäen:

```
./apriori -s20 -ts -q-1 test.data output
```

Tässä siis valitaan minimituki-parametrin arvoksi 20 (*-s20*), etsitään useimmin esiintyviä tietojoukkoja (*-ts*) ja lajitellaan tulokset laskevaan järjestykseen (*-q-1*). Lähdeainesto löytyy tiedostosta test.data ja tulokset ohjataan tiedostoon output. FP-Growth- ja Eclat-algoritmeille suorituskomennon parametrit ovat samat.

Assosiosääntöjen etsiminen testiaineistosta Apriori-algoritmia käyttäen:

```
./apriori -s20 -c25 -tr -q-1 -l-1 test.data output
```

Tässä valitaan edelleen tukiparametille minimiarvo 20 (*-s20*), uskottavuudelle minimiarvo 25 (*-c25*), etsitään assosiaatiosääntöjä (*-tr*) ja lajitellaan tulokset laskevaan järjestykseen (*-q-1 -l-1*). Lähdeaineisto löytyy edelleen tiedostosta test.data ja tulokset ohjataan tiedostoon output.