jamk.fi

CAD-kaavioiden tiedon jäsennyksen ja keräyksen helpottaminen suunnittelu-työssä

Markus Leppänen

Opinnäytetyö
Toukokuu 2020
Tekniikan ala
Insinööri (AMK), sähkö- ja automaatiotekniikka

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Leppänen, Markus	Opinnäytetyö, AMK	Toukokuu 2020
	Sivumäärä 31	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
CAD-kaavioiden tiedon jäsennyksen ja keräyksen helpottaminen suunnittelutyössä

Tutkinto-ohjelma
Insinööri (AMK), sähkö- ja automaatiotekniikka

Työn ohjaaja(t)
Hytönen, Vesa

Toimeksiantaja(t)
Rejlers Finland Oy

Tiivistelmä

Tiedon keräystä vanhoista dokumenteista tarvitaan, kun yritetään siirtää tietoa uudempaan suunnittelujärjestelmään. Suunnittelutyössä voi myös tiedon hallinnassa käydä tilanteita, joissa samaa suunnittelutietoa ylläpidetään monessa dokumentissa samanaikaisesti. Mikäli näissä tilanteissa tietoa unohdetaan päivittää jokaisessa dokumentissa tai jokin dokumenttityyppi tarvitaan myöhemmin projektin aikana, tietoa joudutaan keräämään muista dokumenteista. Tietoa voidaan joutua keräämään vanhoista CAD-kaavioista, mikä on työlästä ja aikaa vievää, jos sitä ei pystytä automatisoimaan.

Tavoitteena oli vähentää kaaviotiedon keräykseen kulutettavaa suunnittelijan työaikaa helpottamalla tiedon keräystä ja jäsennystä ohjelmoinnin avulla. Tarkoituksena oli toteuttaa tämä työkalu samaan aikaan toteutettavien projektien tarpeisiin.

Ohjelma kirjoitettiin AutoCADin Visual Lisp-ohjelmointikielellä. Kehittämisen aikana työkalua kokeiltiin projektien käytössä, mistä kerättiin työkalulle lisää kehityskohteita liittyen käytettävyyteen ja saavutettuun ajalliseen hyötyyn.

Lopputuloksena saatiin ohjelma, jolla voidaan luoda tyyppikaavion perusteella hakukomento, jolla voi hakea tämän tyyppisistä kaavioista tietoa automaattisesti. Ohjelma myös oppii haettuja kuvioita ajan myötä, jolloin hakukomennon luonti helpottuu, mikä auttaa yksittäisten kaavioiden tiedon keräyksessä. Näillä ominaisuuksilla saavutettiin säästöä työajassa. Ohjelmalla on monia jatkokehitysmahdollisuuksia, mutta Visual Lisp-ohjelmointikielen rajoitteiden takia olisi kannattavaa siirtää ohjelma toiselle kielelle, jos ohjelman kehitystä jatketaan.

Avainsanat (asiasanat)
tietokoneavusteinen suunnittelu, tiedon keräys, AutoCAD, ohjelmointi

Muut tiedot (Salassa pidettävät liitteet)

jamk.fi

Description

Author(s)	Type of publication	Date
Leppänen, Markus	Bachelor's thesis	May 2020
		Language of publication: Finnish
	Number of pages: 31	Permission for web publication: x

Title of publication

Simplifying data collection and structuring from CAD drawings

Degree programme

Bachelor's degree programme in Electrical and Automation Engineering

Supervisor(s)

Hytönen, Vesa

Assigned by

Rejlers Finland Oy

Abstract

When trying to move data into a newer engineering system, it is necessary to be able to collect data from old documents. There are also situations in engineering work, where the same data is being maintained in multiple documents simultaneously. If in these situations it is forgotten to update some information in every document, or if some new document type is required later in the project, data needs to be gathered from the other documents. Data may need to be collected from old CAD drawings, which is laborious and time-consuming if it cannot be automatized.

The objective was to reduce the workload that results from data collection from old drawings, by utilizing programming to simplify data collection and structuring. The tool was to be produced for the needs of ongoing projects.

The program was implemented with AutoCAD's Visual Lisp programming language. During development, the program was tested in the ongoing projects to get development pointers on the usability and the achieved time savings of the program.

The result was a program which can be used to create a query based on a type drawing. This query can then be applied in similar drawings to fetch data automatically. The program also learns the queried shapes as the queries are created, which makes the query creation easier and helps with data collection from individual drawings. With these properties, the reduction in workload and consumed time was achieved. The program has many options for further development, but because of the limitations in the Visual Lisp programming language, it is recommended to rewrite the program in another language in the case the development is continued.

Keywords/tags (subjects)

computer-assisted design, data collection, AutoCAD, programming

Miscellaneous (Confidential information)

Sisältö

Kuviot

1 Johdanto, tavoitteet

Sähkö- ja instrumentointisuunnittelussa käsitellään monesti kaavioita, joihin liittyvää tietoa ei ole säilytetty muualla. Kun tätä tietoa halutaan käyttää myöhemmin, tietoon käsiksi pääseminen vie paljon aikaa. Tämän lisäksi, vanhalla tekniikalla tuotetuissa CAD-kaavioissa on usein käytetty tiedon esittämiseen irrallisia tekstiobjekteja, minkä takia tiedon automaattinen keräys on hankalaa. Suunnittelija joutuu usein tarkastelemaan kaavioita yksitellen, mikä on työlästä, epämieluisaa ja vie paljon suunnittelijan työtunteja.

Tämän opinnäytetyön tavoitteena oli luoda työkalu, jolla voi tuoda kaavioista jäsenneltyä tietoa taulukkomuotoon myöhempää käyttöä varten. Toimeksiantajan tavoitteena on vähentää tiedonkeräykseen kuluvia työtunteja. Tämä näkyy säästönä projekteille varatuissa työtunneissa, ja suunnittelijan työaika säästyy hankalampiin tehtäviin. Suunnittelutyö tehostuu, kun kaaviotiedon käsittely ja siirtäminen helpottuu. Suunnittelutyön mielekkyys ja lopputuloksen laatu paranee, kun tämän kaltaista monotonista työtä siirretään ihmisiltä ohjelmille. Ohjelmalle oli tiedossa välitöntä käyttöä käynnissä olevissa projekteissa.

Omana tavoitteenani oli oppia, miten kaavioiden rakenteetonta tietoa voisi ryhmitellä ja jäsentää. CAD-ohjelmat antavat rajallisia mahdollisuuksia tiedon ryhmittelyyn. Symboleilla eli blokeilla on mahdollista ryhmittää yksittäisen laitteen tiedot, mikä auttaa näiden tietojen käsittelyssä CAD-ohjelman ulkopuolella, mutta esimerkiksi laitteiden väliset yhteydet ovat usein luettavissa ainoastaan kaaviosta. Rakenteiden luomisessa irrallisten tekstien välille olisi monia käyttökohteita. Tietoa voisi muokata loogisina kokonaisuuksina myös kaavion ulkopuolella. Tiedon saatavilla oleminen avaa mahdollisuuksia siirtää kaavioiden tietoa uudempiin suunnittelujärjestelmiin.

Työ toteutettiin AutoCADin Visual Lisp-ohjelmointikielellä sen rajoitteista huolimatta, koska se oli minulle ennestään tuttu ja työhön ei ollut varattu aikaa uuden AutoCADin ohjelmointirajapinnan opiskeluun.

Toimeksiantaja Rejlers Finland Oy

Työ tehtiin Rejlers Finland Oy:n Jyväskylän toimistolle. Rejlers Finland Oy on osa Rejlers AB-yhtiötä, joka on monialainen insinööripalveluja tarjoava yritys. Konsernilla on toimintaa Suomessa, Ruotsissa, Norjassa ja Abu Dhabissa. Työntekijöitä konsernilla on yli 2400, joista lähes tuhat tekevät töitä Suomessa. (Sorri n.d.)

Jyväskylän toimistolla on henkilöitä noin 80.

1.1 Kehittämistyö

Opinnäytetyö toteutettiin kehittämistyönä. Kehittämistyö koostuu usein jonkin työelämän ongelman ratkaisuun kehitettävästä tuotteesta, työvälineestä tai toimintasuunnitelmasta, ja tätä kehitysprosessia kuvailevasta raportista. (Liukko & Perttula 2019)

Kehittämistyö perustuu tiedon suunnitelmalliseen soveltamiseen. Kehittämistyö muodostuu tarpeen, suunnittelun, kehityksen ja havainnoinnin kierteestä. (Salonen, Eloranta, Hautala & Kinos 2017, 33) Kehittämistyössä pyritään konkreettiseen, käytettävään lopputuotokseen. (mts. 37)

2 CAD-pohjainen suunnittelu ja sen haasteet

Perinteisesti sähkö- ja instrumentointisuunnittelua tehdään CAD-ohjelmien (CAD = computer aided design), kuten AutoCADin, avulla. CAD-ohjelmilla piirretään kaavioita, joiden avulla suunniteltava kohde on tarkoitus toteuttaa. Nämä erinäiset kaaviot, kuten vaikka piirikaaviot, ovat CAD-pohjaisen suunnittelun lopputuote. Kaavioiden tuottamisessa hyödynnetään muuta suunnittelun aikana tuotettua suunnittelutietoa, mutta usein kaavioissa pyritään lopulta esittämään tästä tiedosta mahdollisimman suuri osa. Näin tehdään, jotta kaikki tieto olisi esitetty yhdessä paikassa ja

tietoa olisi helpompi ylläpitää kohteen elinkaaren aikana. Tiedon ylläpitäminen monessa dokumentissa samanaikaisesti altistaa virheille, koska kaikkia dokumentteja ei aina muisteta päivittää. Kaaviot ovat helposti luettavissa ja helppoja päivittää, mistä ajan kuluessa seuraa useimmiten se, että kaavioiden sisältämä tieto on ainut ajan tasalla oleva tieto.

Suurten kaaviomäärien tuottaminen yksitellen CAD-ohjelmalla on työlästä ja useimmiten samanlaisia kaavioita tuotetaan monta kappaletta, minkä takia monessa CAD-ohjelmassa on mahdollisuus luoda uusia kaavioita jonkin toisen kaavion perusteella. Näitä kaavioita kutsutaan usein tyyppikaavioiksi tai pohjakuviksi, ja uusien kaavioiden luontia kutsutaan generoinniksi. Tyyppikaaviot koostuvat useimmiten muun sisällön lisäksi yksilöllisistä teksteistä, joiden sisältö voidaan korvata generoidessa etsi-ja-korvaa-toiminnolla. Tällä tavalla suuretkin projektit voidaan toteuttaa melko vähillä panostuksilla itse CAD-piirtämiseen.

Tämän tyyppinen etsi-ja-korvaa-generointi vaatii usein generointitaulukoita, joissa määritellään, mitä arvoja kaavioiden tekstit saavat. Generointitaulu on usein kopioitua tietoa muista suunnittelunaikaisista dokumenteista, mikä luo riskin, koska samaa tietoa esitetään monessa paikassa. Kun generointitaulu on valmis, ensimmäiset kaavioiden versiot generoidaan. Suunnitteluprojektin aikana kaavioihin ja muihin suunnittelunaikaisiin dokumentteihin tulee monia muutoksia. Generointitaulukot ja kaaviot sisältävät tässä vaiheessa samaa tietoa, mutta koska generointitaulua on hankalampi ylläpitää, päädytään usein päivittämään vain siihen hetkeen tarvittavia dokumentteja ja generointitaulun tieto jätetään ennalleen. Jos joidenkin kaavioiden tyyppi muuttuu ja haluttaisiin generoida kaaviot uudelleen uusien tyyppien mukaan, voidaan joutua tekemään generointitaulukko uudelleen, koska sen tiedot ovat vanhentuneet. Pahimmassa tapauksessa suunnittelutieto on jo sisällytetty kaavioihin ja se on ainutta ajan tasalla olevaa tietoa, jolloin joudutaan käyttämään huomattavasti aikaa kaavioiden yksitellen käsittelyyn.

Kaaviotiedon keräystä voidaan siis tarvita CAD-pohjaisessa suunnittelussa jo suunnittelun aikana tai myöhemmin, vanhoja kaavioita käsitellessä. Tarkalla tiedon ylläpi-

dolla tätä keräystä ei tarvitsisi tehdä, ja uudemmat, tietokantapohjaista suunnittelutapaa hyödyntävät suunnittelujärjestelmät pyrkivät tiedon helppoon ylläpitoon. Tietokannassa tietoa pyritään säilyttämään vain yhdessä paikassa. Samaa tietoa käytetään useassa paikassa luomalla linkkejä, jolloin tiedon muuttuessa se päivittyy koko tietokantaan. Näin tiedon ylläpidon riskit pienenevät huomattavasti.

Etsi-ja-korvaa-generoinnin sivuoireena on kaavioiden luonti mahdollisimman yksinkertaisista elementeistä. Irrallisten tekstien lisäys tyyppikaavioon on helpompaa kuin blokkien tekeminen. Irralliset tekstiobjektit menettävät kaiken niihin liittyneen rakenteen generoinnin yhteydessä, koska kuvauksettomat tekstit eivät kerro mitään niiden sisällöstä, eivätkä siitä, mihin toisiin teksteihin ne liittyvät. Blokkien luonti auttaa tiedon ryhmittelyssä, mutta vanhoissa tai vanhalla tekniikalla tuotetuissa kaavioissa näitä objekteja ei ole hyödynnetty. Vaikka blokkeja olisi käytetty, monet laitteiden väliset yhteydet, kuten esimerkiksi laitteiden kytkentä, ovat nähtävissä ainoastaan kaavioista vanhoissa CAD-kuvissa. Tämän takia CAD-suunnittelu voi sisältää huomattavasti ylimääräistä työtä kaavioiden käsittelyssä ja tarkastelussa. Kytkentä- ja muiden luetteloiden ylläpito perinteisessä kaaviopohjaisessa suunnittelussa on hankalaa.

3 CAD-ohjelmointi

CAD-ohjelmat on tehty monipuoliseen kaavioiden käsittelyyn, ja monet niistä tarjoavat mahdollisuuden toiminnallisuuden laajentamiseen ohjelmoinnilla. Jos jotain toimintoa ei ole suunnitteluohjelmassa saatavilla, on mahdollisuus rakentaa se itse, mikä hyödyttää suunnittelutyössä. CAD-ohjelmointia voi tehdä erilaisten ohjelmointirajapintojen kautta. Jotkin näistä ovat jonkin ohjelmointikielen kirjastoja, jotka yhdistyvät CAD-ohjelmaan, kun taas jotkin rajapinnoista ovat kokonaisia ohjelmointikieliä. Tämä opinnäytetyö on toteutettu AutoCADin Visual Lisp-ohjelmointirajapintaa hyödyntämällä. Visual Lispin toiminta ohjelmointikielenä on moneen käyttöön rajallinen, mutta kaavioiden käsittely sen avulla on nopeasti opeteltavissa.

3.1 Lisp ohjelmointikielenä

Lisp on John McCarthyn 1950-luvun lopussa kehittämä ohjelmointikieli. Sen nimi muodostuu sanoista list processor, eli listakäsittelijä. Sen vahvuuksia on yksinkertainen syntaksi ja lambdalaskentaan perustuva ajatusmaailma, joka sallii algoritmien esittämisen funktioina ja siten matemaattisen analysoinnin. (McCarthy, 1960)

Lispiä käytettiin aktiivisesti 1990-luvulle asti, jonka jälkeen sen suosio on hiipunut. Moderneja ja nykyään ylläpidettyjä Lisp-murteita ovat Racket, Clojure ja Common Lisp -ohjelmointikielet.

Lispin syntaksi eli lausemuoto koostuu symbolisista lausekkeista (s-lauseke), jotka esittävät listatietorakennetta tai yksittäisiä arvoja kuten lukuja, tekstiä tai symboleita. Kielen toiminnot koostuvat funktioista. Toisin kuin monessa muussa kielessä, kieli ei sisällä erityisiä operaattoreita kuten +, -, ==, jotka toimisivat eri tavoin kuin muut toiminnot, vaan näiden operaattorien tilalla on vastaavat toiminnot suorittavat funktiot. (Learn Clojure – Syntax. N.d.)

Lisp-syntaksi on yksinkertainen. Suoritettavan funktion symboli laitetaan sulkuihin ensimmäisenä ja funktion käyttämät arvot tulevat funktiosymbolin jälkeen. Laskenta suoritetaan sisimmät sulkeet ensimmäisenä, kuten matematiikassa. Tämä syntaksi pätee kaikkeen, esimerkiksi pluslasku kirjoitetaan lispissä näin: (+ 1 2). + on funktio siinä missä muutkin. Esimerkin lasku palauttaa tulokseksi 3. Tämä funktiokutsu on samalla myös listarakenne, jossa on jäseninä +, 1 ja 2. Tämä listojen ja koodin samankaltaisuus mahdollistaa kielen käsittelyn paljon helpommin kuin muissa kielissä, koska ohjelmoijan ei tarvitse ajatella kielen parsimista, vaan syntaksi on suoraan luettavissa listana. (Learn Clojure – Syntax. N.d.)

Syntaksin helppoudesta seuraten suurimpaan osaan Lisp-murteista on lisätty ohjelmointimahdollisuus makroille, jolla voi lisätä kieleen omaa syntaksia. Tämä on metaohjelmointia, ohjelmointia ohjelmoinnin muuttamiseksi, helpottamiseksi tai vaikka ohjelmien generoimiseksi. Vaikka käytetyssä Lisp-murteessa ei olisi makroja,

on kielen rakenne niin yksiselitteinen, että tällaiseen murteeseen jonkinlaisten makrojen lisääminen tulee ohjelmoijalle lähes luonnostaan kysymykseen.

3.2 Funktionaalinen ohjelmointi

Lambdalaskenta on funktioihin perustuva malli laskennasta, jonka kehitti Alfonzo Church 1930-luvulla. Lambdakalkyyli rakentuu funktioille ja symboleille. Lambda tarkoittaa tässä laskentatavassa funktiota, joka ottaa yhden arvon ja palauttaa sen perusteella lasketun arvon. Symboli tarkoittaa muuttujaa, jonka arvo voi olla mikä vain funktiolle annettu arvo kuten luku tai toinen funktio. Esimerkki lambda-funktiosta voisi olla f(x) => x*2, joka ottaa numeerisen muuttujan x ja palauttaa sen kerrottuna kahdella. Lambdalaskennasta tulee hyödyllistä, kun funktioille antaa kuvaavia nimiä. Funktion sisäisestä rakenteesta ei tarvitse välittää, jos tietää, mitä funktion tulisi tehdä ja voi luottaa, että funktio on tehty oikein. Näin voidaan abstraktoida tasoa, jolla laskentaa käsitellään. Funktion valmistuttua ei enää tarvitse ajatella funktion sisäisiä yksityiskohtia, jolloin ajattelu voi vapautua suunnittelemaan monimutkaisempia algoritmeja. Nämä algoritmit koostuvat vastaavanlaisista valmiista, pienistä funktioista. (Alama & Korbmacher 2019; Hutton 2017)

Lambdalaskennan tyyppinen laskennan funktiointi on mahdollista jokaisessa modernissa ohjelmointikielessä sen salliman yksinkertaistuksen ja paketoinnin takia. Funktioiden sijaan käytetään usein nimitystä aliohjelma, koska monissa kielissä funktioiden ei tarvitse palauttaa mitään, vaan laskenta suoritetaan tuottamaan vaikutuksia muualla ohjelmassa. Tämä on tehokasta tietokoneen resurssien käyttöä, mutta suuremmissa ohjelmissa tämä voi tuottaa vaikeuksia, kun ohjelmaa muokataan. Lispin ja muiden lambdalaskentaan perustuvien, funktionaalisiksi kutsuttujen, kielien tapana on välttää sivuvaikutuksia, jos se vain on mahdollista. Tämä helpottaa funktioiden analysointia, koska tiedetään, että funktio ottaa tiedon ja palauttaa muokatun tiedon, eikä tee muutoksia muualle. Sivuvaikutuksettomat funktiot ovat myös helpompia uusiokäyttää, koska ne eivät ole sidoksissa alkuperäiseen käyttökohteeseen. (Hutton 2017; Functional programming n.d.)

Funktionaalisten kielten heikkous on siinä, että tiedon muokkaaminen vie usein enemmän resursseja, koska sen sijaan että muokattaisiin olemassa olevaa tietoa muistissa, luodaan siitä kopio, jottei muiden tietuetta käyttävien funktioiden arvo muutu kesken suorituksen. Joissain kielissä, kuten Clojuressa, on luotu tietorakenteita, jotka vähentävät tätä ongelmaa sallimalla koko tietorakenteen kopioinnin sijaan tietorakenteen pienempien osien kopioinnin. Kokoelmasta monen kopion pitäminen muistissa vaatii kuitenkin paljon tilaa. Puhtaasti funktionaalisissa kielissä, kuten Haskellissa, säästetään resursseja suorittamalla laskentaa vasta, kun sille on tarve. Funktionaaliset kielet vaativat myös usein eri ajatusmaailman kuin useimmat modernit ohjelmointikielet. Olioiden ja niiden käytöksen ja ominaisuuksien sijaan ajattelu siirtyy siihen, mitä toimenpiteitä datalle tulisi suorittaa. (Functional programming n.d.)

3.3 AutoCADin Visual Lisp-ohjelmointirajapinta

Visual Lisp on AutoCADin täysversiossa mukana tuleva ohjelmointirajapinta, minkä avulla voi automatisoida kaavioiden käsittelyä. Ennen nimeä Visual Lisp ohjelmointikielen nimi oli AutoLISP. AutoLISP lisättiin AutoCADiin vuonna 1986. Sen tarkoituksena oli toimia suunnittelijan apuna AutoCADin mukauttamiseen suunnittelijan omaan käyttöön sopivaksi. Basis Softwaren kehittämä AutoLISPin lisäosa Vital-Lisp lisäsi kieleen VBA-objektien käsittelymahdollisuuden ja tapahtumakäsittelyn AutoCAD-objekteille. Autodesk osti Vital-Lispin Basis Softwarelta ja integroi sen AutoCADiin vuonna 2000. Laajentuneiden ominaisuuksien perusteella AutoLISP nimettiin uudelleen Visual Lispiksi, ja tämä nimi on edelleen käytössä. Samassa päivityksessä AutoCADiin lisättiin kehitysympäristö VLIDE Visual Lispiä varten. (AutoLISP n.d.; Ambrosius & Byrnes 2006; Poleschuk 2001)

Kaavioiden käsittelijän työnkuva ei usein sisällä ohjelmointia, minkä takia Visual Lisp on kehitetty kokemattoman ohjelmoijan tarpeisiin. Autodesk on tehnyt useita kompromisseja kielen monipuolisuuden ja ymmärrettävyyden välillä.

Visual Lisp on dynaamisesti tyypitetty kieli, jossa muuttujilla on dynaaminen näkyvyysalue (engl. dynamic scope). Visual Lispissä on monia funktionaaliseen ohjelmoinnin piirteitä, mutta sisältää perinteisiä silmukkarakenteita kuten while ja foreach. Nimettyjen funktioiden määrittely on mahdollista defun-komennolla, ja nimettömien eli anonyymien funktioiden luonti tehdään lambda-muodolla. Funktioita voidaan käyttää muuttujissa ja antaa funktioille parametreina kuten muita arvoja. Muuttujien dynaaminen näkyvyys estää sulkeumien (engl. closure) käytön ohjelmoinnin keinona. Funktio ei säilytä luomishetken ympäristöä, joten funktion luomiseen käytetyt muuttujat “unohtavat” arvonsa, jos funktio palautetaan luomisympäristön ulkopuolelle. Funktio poimii käytettyjen muuttujien arvon aina kutsuhetken ympäristöstä. (AutoLISP n.d.)

Visual Lispin arvot ovat muuttumattomia, mikä tarkoittaa, että kun arvo on luotu, sitä ei voi enää muuttaa. Tällä on turvattu eri kohdissa ohjelmaa tehtyjen viittausten pysyvyys. Jos esimerkiksi muuttuja x sisältää listan (1 2 3), joka tallennetaan muuttujaan y, näitä listoja voi muokata välittämättä, muuttuuko myös toisen muuttujan arvo. Muuttujan arvoksi voi asettaa uuden, muokatun arvon, mutta tämä ei vaikuta muualla käytettyihin viittauksiin vanhasta arvosta. Arvojen muuttumattomuus yhdistettynä muuttujien dynaamiseen näkyvyyteen vaikeuttaa sivuvaikutusten luomista funktion ulkopuolelle. Sivuvaikutuksia voi edelleen luoda käyttämällä aliohjelmassa globaaleja muuttujia tai kutsuketjussa ylemmän funktion muuttujia. Hyvän ohjelmointitavan mukaista on kuitenkin minimoida näiden käyttö.

Dynaaminen tyypitys on lispissä vain pinnallista. Muuttujalla eli symbolilla itsellään ei ole tyyppiä, vaan se on muuttujan sisältämällä arvolla. Muuttujan tyyppiä ei tarvitse ilmoittaa erikseen, vaan kieli tarkastaa tyypit ajon aikana. Tyyppejä ei yritetä automaattisesti muuttaa sopiviin muotoihin, vaan ohjelmoijan tulee pitää huoli, että funktiot saavat oikeantyyppisiä arvoja. Perustietotyyppejä ovat kokonaisluku, reaaliluku, merkkijono, symboli, lista, valintajoukko, funktio, entiteettinimi ja tiedostoosoitin. Yleisinä tietueina käytetään olioiden sijasta assosiaatiolistoja. Visual Lispin osana on tullut ActiveX-safearray sekä VBA-rajapinnan variant ja VLA-objekti, joita voi käyttää vain Windowsissa. (AutoLISP: Reference 2018; Poleschuk 2001)

Myöhemmin tässä opinnäytetyössä sanalla lisp viitataan erityisesti AutoCADin Visual Lisp-rajapintaan.

Pääsy AutoCAD-kaavion tietokantaan

AutoCADin vanha kaavioiden tallennusmuoto on DXF (Drawing Interchange Format), jonka Autodesk kehitti 1980-luvulla kuvatiedostojen jakamiseen eri CAD-järjestelmien välillä. (AutoCAD DXF n.d.)

Lispissä on lähes täysi pääsy AutoCAD-kaavion objektien sisäiseen esitysmuotoon. Kun kuva avataan, AutoCAD lukee tiedoston ja luo kustakin objektista ohjelmallisen esityksen kuvan tietokantaan. Tietokannassa eri objekteilla on omat, määritellyt tietueet, joita voi muokata lispissä. Näiden tietueiden perusteena on DXF-rakenne. Ohjelmassa voi hakea objekteja niiden esimerkiksi niiden tyypin ja sijainnin perusteella. Lispissä on myös pääsy VBA-objekteihin, mutta näiden käyttö on rajattu Windows-ympäristöön. (Ambrosius & Byrnes 2006; Poleschuk 2001)

Kuvatietokannan sisäinen esitysmuoto graafisille objekteille on entiteetti. Kuvatietokannasta voi hakea päällimmäisiä entiteettejä lispin ssget-funktiolla. Näistä AutoCAD luo valintajoukon (engl. selection set). Päällimmäisillä entiteeteillä tarkoitetaan entiteettejä, joilla ei ole alientiteettejä. Esimerkiksi blokit kuuluvat tähän ryhmään. Blokeilla on oma määrittelytietonsa kuvakannan taulut-osiossa, block-taulussa. Blokkien kuvassa esitetyt instanssit on tallennettu entiteettiosioon INSERT-tyyppisinä entiteetteinä. Blokkien attribuutit ovat INSERTien alientiteettejä, joten niitä ei voi suoraan hakea valintajoukkoon, vaan ne tulee etsiä blokkien alta entnext-funktiolla. AutoCAD säästää resursseja piirtämällä blokin sisäiset, muuttumattomat osat blokin määritelmän perusteella, minkä takia INSERTin alla ei näy blokin sisällä olevia tekstejä tai viivoja. Mikäli näihin halutaan päästä käsiksi, etsitään ne block-taulusta. Muita opinnäytetyössä tarvittuja suoralla ssget-haulla löydettäviä entiteettejä kuvatietokannassa ovat irralliset TEXT-, MTEXT- ja ATTDEF-entiteetit. (Ambrosius & Byrnes 2006; AutoLISP: Reference 2018; Poleschuk 2001)

Kullakin objektilla on yksilöllinen tunniste, handle, joka säilyy objektilla kuvan tallennusten välissä. Tämä yksilöllinen tunniste muuttuu, jos objekti poistetaan tai tilalle asetetaan uusi objekti. Handlejen käyttö mahdollistaa kuvan tietojen muokkauksen AutoCADin ulkopuolella, koska muokattu tieto voidaan tuoda tarkasti samaan paikkaan kuin mistä tieto poimittiin. (AutoLISP: Reference 2018)

4 Algoritmien tehokkuusanalyysi

Tietokoneen voi ohjelmoida tekemään saman asian monella eri tavalla. Komennoista koostuvissa algoritmeissa on kuitenkin tehokkuuseroja. Algoritmin tehokkuutta voidaan mitata vaadituilla alkeisoperaatioilla suhteessa syötteen pituuteen, kun algoritmin lopputulos on sama. (Grimson 2016)

Alkeisoperaatiolla tarkoitetaan algoritmille olennaisen pienimmän suoritusajan vaatimaa operaatiota. Yksinkertaisimmillaan tämä tarkoittaa prosessorin ymmärtämän konekielen operaatiota. Konekielen operaatiot on rakennettu prosessorin laitteistoon, ja ne suoritetaan useimmiten yhdessä tai muutamassa prosessorin kellosyklissä. Nämä alkeisoperaatiot ovat prosessorin nopeimmin suoritettavia käskyjä, ja niistä rakentuu kaikki tietokoneella suoritettavien ohjelmien toiminta. Siten voidaan olettaa, että ohjelman suoritusaika kasvaa lineaarisesti suoritettavien alkeisoperaatioiden määrään nähden. (Grimson 2016)

Käytännössä algoritmien tehokkuusanalyysissä ei käsitellä alkeisoperaatioita aivan konekielen tasolla, sillä moderneissa ohjelmointikielissä pääsy konekielen tasolle on monen käännöksen takana ja tämän jälkeen ohjelman lukeminen ja ymmärtäminen voi olla ohjelmoijalle lähes mahdotonta. Sen sijaan alkeisoperaatioita käsitellään abstraktilla tasolla, jossa ohjelmointikielellä ei ole väliä. (Grimson 2016)

Syötteellä tarkoitetaan ohjelmalle annettuja alkioita, joilla ohjelma suorittaa algoritmin määrittelemät toiminnot. Alkio voi olla esimerkiksi luku tai teksti, ja syöte voi koostua useasta alkiosta. Syötteen pituudella tarkoitetaan syötteessä olevien alkioiden määrää.

Esimerkkinä alkeisoperaatioiden laskemisesta voidaan kuvitella tilanne, jossa lisätään kaksi lukua toisiinsa 1 + 2. Tässä alkeisoperaatioita on tasan yksi, kahden luvun summa. Mikäli syöte olisi pidempi, esimerkiksi viisi lukua 1 + 2 + 3 + 4 + 5, summa-alkeisoperaatioita suoritettaisiin neljä kertaa. Jos lukuja lisätään syötteeseen, huomataan, että vaadittavien alkeisoperaatioiden määrä syötteen summaamiselle noudattaa funktiota f(n) = n - 1, jossa n on syötteen pituus eli summattavien lukujen määrä. Näin esimerkiksi sadalla luvulla suoritettaisiin f(100) = 100 - 1 = 99 alkeisoperaatiota. Tämän “summausalgoritmin” tarkka suoritusaika on siis n - 1 kertaa alkeisoperaation vaatima aika.

4.1 Asymptoottinen suoritusaika ja ordo-notaatio

Kun kasvatamme summattavien lukujen määrää lisää, summafunktion suoritusmäärän vakio-osa -1 menettää vaikutustaan suoritusajan pituuteen. Syötteen pituuden n lähentyessä ääretöntä tuo vakio-osa menettää täysin merkityksensä, koska n viemä suoritusaika on niin suuri verrattuna yksittäiseen operaatioon. Tätä kutsutaan asymptoottiseksi suoritusajaksi. Asymptoottinen suoritusaika on algoritmille ominainen suoritusaika syötteen pituuden lähentyessä ääretöntä. Syötteen kasvaessa asymptoottinen aika on huomattavasti yksinkertaisempi tapa verrata algoritmien tehokkuutta keskenään, sillä vähemmän merkitykselliset osat putoavat verrattavista funktioista pois, ja jäljelle jää nopeimmin kasvava, skaalautumiseen eniten vaikuttava osa. Yllä olevan summausalgoritmin asymptoottinen suoritusaika on siis f(n) ~ n. (Grimson 2016)

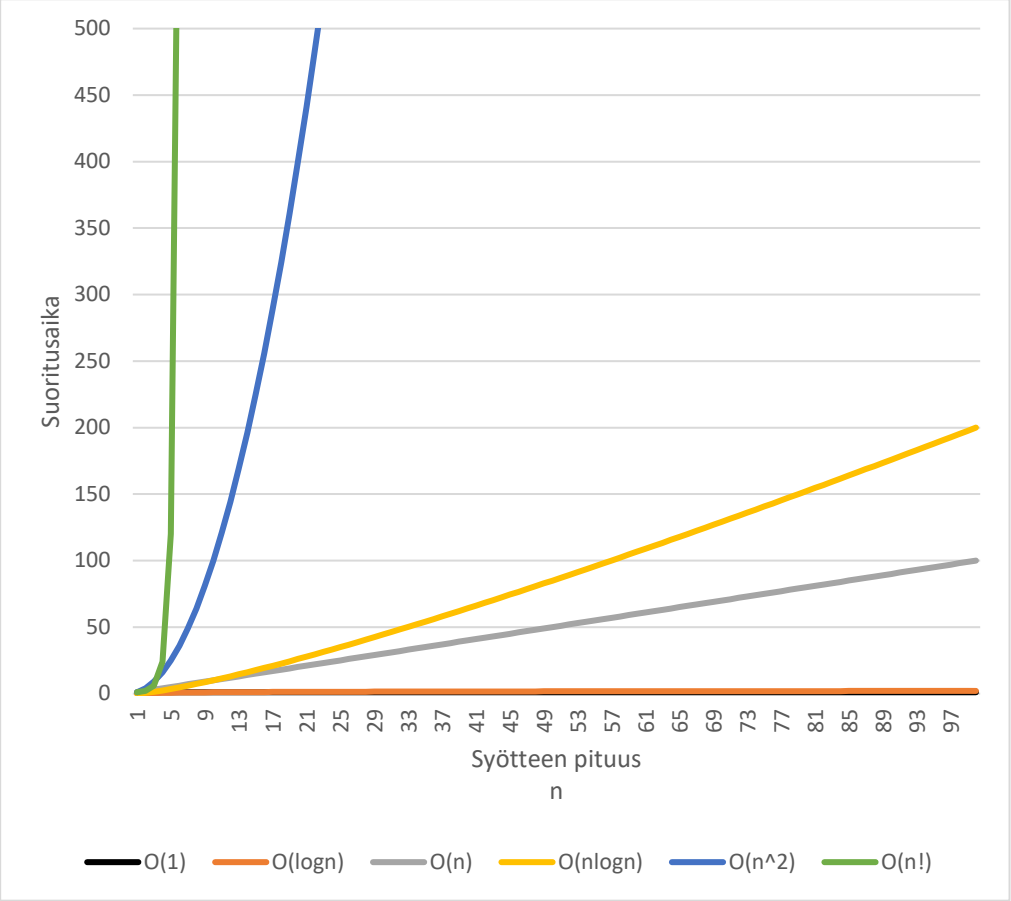
Algoritmit eivät välttämättä suoriudu jokaisesta tapauksesta tismalleen samalla nopeudella. Esimerkiksi lajittelualgoritmi, joka järjestää numerot pienimmästä suurimpaan, voi suoriutua valmiiksi järjestyksessä olevasta syötteestä käymällä luvut vain kerran läpi tarkastaen järjestyksen, kun taas sekaisin olevan syötteen kanssa järjestämiseen kuluisi huomattavasti kauemmin. Ohjelmoijalle merkityksellisin aika on huonoimman tapauksen kuluttama aika. Tätä kuvataan ordo-notaatiolla (engl. Big-O notation). Ordo-notaatio on tapa ilmaista algoritmin huonoimman tapauksen

asymptoottinen suoritusaika. Yllä oleva summausesimerkki ilmaistaisiin ordo-notaationa O(n), mikä tarkoittaa, että käytetty summausalgoritmi vie huonoimmassa tapauksessa noin n alkeisoperaatiota. (Grimson 2016)

Ordo-notaatiolla saa nopeasti selville, mikä on siedettävästi skaalautuva algoritmi. $O(n^2)$ on esimerkki huonosti skaalautuvasta algoritmista: kun syötteen pituus tuplaantuu, suoritusaika nelinkertaistuu. Syötteen kannalta tämä tarkoittaa sitä, että jokaista käsiteltävää alkiota n kohden täytyy käydä koko syöte läpi eli n * n. (Grimson 2016)

Hakualgoritmit toimivat hyvänä esimerkkinä algoritmien tehokkuuden vertailuun, sillä niillä on ollut käyttöä tässä opinnäytetyössä. Jos haetaan pienimmästä suurimpaan järjestetystä syötteestä lukua 15 lähimpänä olevaa lukua, voidaan naivisti käydä kaikki syötteen luvut läpi verraten kaikkia 15:een. Tämä “lineaarihaku” on O(n)-kertaluokkaa oleva algoritmi, koska kukin alkio käydään läpi kerran. Parempi algoritmi tähän on puolitushaku (engl. binary search): koska tiedetään, että syöte on järjestyksessä, voidaan hypätä syötteen puoliväliin, verrata onko syötteen alku- vai loppupään arvo lähempänä 15:ä, ja sitten toistaa sama valitulle puoliskolle, kunnes joukkoa ei enää voi puolittaa. Tämän algoritmin asymptoottinen suoritusaika on $O(\log_2 n)$. Jälkimmäinen on huomattavasti nopeampi algoritmi, kuten näkee kuviosta alla. Tuhannella alkiolla lineaarihaku käyttää tuhat alkeisoperaatiota, kun puolitushaku käyttää noin kymmenen. Vastaavasti jokin $O(n^2)$ -kertaluokan algoritmi käyttäisi miljoona operaatiota samaan. (Binary search algorithm n.d.; Grimson 2016)

Yleisesti hyvänä kertaluokkana suurien massojen käsittelyyn pidetään O(nlogn)-kertaluokkaa, sillä se on lähes lineaarisesti skaalautuva kertaluokka ja kuluva aika on siten helposti ennustettavissa. Tätä raskaammat algoritmit skaalautuvat yleensä niin huonosti, ettei niitä kannata käyttää kuin ongelmiin, joissa syötteen pituus on valmiiksi tiedossa tai jos suoritukselle voidaan varata paljon aikaa.



Kuvio 1. Suoritusaika, kun ordo-notaation mukainen algoritmi suoritetaan

5 Toteutus

Kehittämistyö alkaa siitä, että huomataan tarve parannukselle nykyisessä tilanteessa. Tilanteen analysoinnin jälkeen pyritään keksimään ratkaisu, jolla saavutettaisiin positiivinen muutos. Tätä kutsutaan ideointivaiheeksi. Ideoinnin jälkeen suunnitellaan ratkaisua tarkemmin. Tästä siirrytään toteutusvaiheeseen, jossa pyritään toteuttamaan suunniteltu ratkaisu. Uutta ratkaisua kokeillaan ja tutkitaan, ja mahdollisesti korjataan suunnitelmaa sopimaan paremmin havaittuihin tarpeisiin. Suunnitelman muutoksen jälkeen palataan toteutukseen. Tätä toteutuksen ja reflektoinnin sykliä jatketaan, kunnes ollaan tyytyväisiä ratkaisuun. Kehittämistyö päätetään ja ratkaisu otetaan käyttöön. (Salonen ym. 2017, 52-54)

Kehittämistyön toteutusvaiheen sykli näkyy ohjelmistokehityksessä vahvasti, koska ohjelmoija testaa ohjelmaa kirjoittaessa jatkuvasti. Jatkuvalla testauksella voidaan

huomata virheet lähellä niiden kirjoitushetkeä. Ohjelmatestauksella voidaan varmistaa vaatimusten täyttyminen. Tämän lisäksi tehdään käyttökokeiluja, joissa arvioidaan suurempia käyttöön vaikuttavia tekijöitä. Näiden tekijöiden takia voi tarvita muokata suunnitelmaa huomattavastikin. (Pyhäjärvi & Pöyhönen 2006)

5.1 Toiminnon suunnittelu

Projekteissa oli huomattu tarve työkalulle, jolla saataisiin kerättyä jäsennettyä tietoa irrallisista tekstiobjekteista koostuvista kaavioista. Ensimmäisessä projektissa tavoitteena oli saada kytkentäluettelo tuhannesta piirikaaviosta tyyppikaavioiden perusteella. Olin aiemmin käsitellyt tällaista tietoa Visual Lispillä, mutta tällaisen tiedon jäsennys oli haaste. Tyyppikaavioita oli kehityksen alussa noin 25 kappaletta ja näiden perusteella tuli tehdä hakuja, jotka toisivat tiedot kaaviosta taulukkoon.

Kaaviotyypin tunnistus

Kaavioista oli saatavilla alkuperäiseen generointiin käytetyt tyyppikaaviot ja lista mitä kuvapohjaa kuhunkin kuvaan oli käytetty. Ensi alkuun tuli selvittää, paljonko kaavioihin oli tehty muutoksia niiden alkuperäisen generoinnin jälkeen. Tämä oli mielestäni tarpeen, koska irrallisten tekstitietojen haku tulisi olemaan sidonnainen joko handletietoon tai tekstin sijaintiin. Lyhyellä tarkastelulla huomattiin, että kaavioihin oli tehty huomattavasti lisäyksiä. Samalla pohjakuvalla luoduista kuvista saattoi olla kolmeakin eri versiota, minkä takia alkuperäisen pohjakuvan mukaan tehdyllä haulla ei saataisi kerättyä kaikkia tietoja. Lisätehtäväksi tuli siis selvittää, mitä muokatuista kuvista voitaisiin käyttää uusina pohjakuvina, jotta haku tuottaisi mahdollisimman paljon oikeaa tietoa.

Kaavioiden määrän takia tyyppikuvatarkastukseen oli hyvä keksiä automatisointia. Uuden työkalun kehittäminen tarkastukseen olisi vienyt projektilta liikaa aikaa, minkä takia tästä luovuttiin ja sen sijaan hyödynnettiin jo ennalta käytössä olevaa ohjelmaa, joka tuo kaavioista kullekin entiteetille koordinaatit, handlen, tyyppitiedon ja tekstisisällön. Ratkaistavaksi jäi, miten tietoa vertailtaisiin kaavion ja tyyppikaavioiden välillä.

Generoinnissa objektien handlet säilyvät, mutta handle ei ole luotettava, jos kaavioita on muokattu huomattavasti generoinnin jälkeen. Tämä johtuu siitä, että usein kaavioita käsin muokatessa tekstejä poistetaan, siirretään ja lisätään vapaasti. Handlen validiteetti katoaa poistettaessa ja uusia tekstejä lisättäessä, koska handlet voivat siirtyä eri tekstille kuin alkuperäisessä kuvassa on tarkoitettu. Samoin voi tapahtua tekstiä siirrettäessä, koska teksti voidaan siirtää paikkaan, jossa sitä käytetään eri tiedon esittämiseen. Jos paljon muokatussa kaaviossa tietoa haetaan puhtaasti handlen perusteella, voisivat esimerkiksi johdintunnukset vaihtua keskenään, mikä muuttaisi kytkennän merkityksen. Handlet ovat myös herkkiä käsin tuotetuille kaavioille. Handle ei säily kaaviosta toiseen kopioidessa, joten jos kaavio on tehty käsin käyttämättä tyyppikuvaa generointiin, handleen liittyvä tieto voi olla tekstin sijasta mikä vain muu kuvasta löytyvä objekti.

Sijaintitiedon perusteella tekstiä haettaessa voidaan korjata osa tämän tyyppisistä käsin muokkauksen oireista, mutta ongelmaksi nousee käsin siirretyn tekstin sijainnin tarkkuus. Mikäli kaavioissa ei ole käytetty snapia, sijaintia ei voi käyttää ilman jonkinlaista suodatusta etäisyyden perusteella. Jos jotain kuvan osaa on siirretty kokonaisuudessaan, sijaintitieto menettää validiteettinsa samalla tavalla kuin handle-tieto, koska sijaintihaku etsii tietoa väärästä kohdasta. Tämä voi jälleen johtaa ristiin poimittuun tai puuttuvaan tietoon.

Handle-tieto yhdistettynä sijainnin tarkastukseen on huomattavasti parempi mittari tekstin validoimisessa. Jos handle on sama kuin tyyppikuvassa ja teksti on lähellä hakupistettä, teksti hyväksytään, ja jos handle ei täsmää, tekstiä ei hyväksytä. Tämäkään ei ollut tarkka validointikeino, mutta se oli tarpeeksi hyvä kaavioiden vertailuun.

Projektin kaikista kuvista ajettiin vertailu pohjakuviin handlejen ja sijaintien perusteella. Vertailussa tallennettiin osuneiden handlejen ja sijaintien prosentit kaikista kaavion teksteistä. Tästä datasta ilmeni, että 75 prosenttia kaikista kaavioista oli käsitelty niin, ettei handlet ja sijainnit täsmänneet alkuperäisiin pohjakuviin. Muutaman pistokokeen perusteella vertailuskriptiä muokattiin niin, että jos handlejen osuvuus mihinkään tyyppikaavioon oli pienempi kuin 90%, tai jos sijaintien osuvuus oli pie-

nempi kuin 97%, lisättiin vertailtava kaavio tyyppikaavioiden joukkoon. Tällä näytettiin saavuttavan varmuus siitä, että kaavioihin voitaisiin käyttää samaa hakukomentoa.

Uuden vertailun tuottamasta luettelosta huomattiin, että usean kuvan pohjana toimineita kaavioita olikin 25 sijaan 70, jotka kattoivat 800 tuhannesta kaaviosta. Loppujen 200 kaavion tiedot oli kerättävä yksitellen.

Taulukon rakentaminen

Kaavioista olisi tuotava tiedot Excel-taulukkoon. Excelin tiedostomuotoon ei ole helppo tallentaa tietoa suoraan, joten vaihtoehtona oli tuottaa tieto suoraan toisessa formaatissa. Taulukko luotaisiin csv-tiedostomuotoon. Csv- eli “character separated value”-tiedosto on yksinkertainen tekstitiedosto, jossa taulukon solujen arvot on erotettu jollakin merkillä. Excelin suomenkielisessä versiossa tämä on oletuksena puolipiste “;”. Csv-tiedosto on hyvin helppo luoda, ja se avautuu suoraan Excelissä, mikä oli käytettävyyden kannalta hyödyksi. Csv:ssä otsikkorivinä toimii tiedoston ensimmäinen rivi.

5.2 Hakukomentojen luonti

Ensimmäisen projektin tarpeisiin ohjelman tuli kerätä kaapeleiden tiedot kustakin kuvasta. Kaapeleita oli kuvassa yhdestä viiteen kappaletta. Tässä projektissa taulukkoon haluttiin kunkin johtimen osalta kaapelitunnus, kaapelityyppi ja johtimen nimi sekä kaapelin molempien päiden laite ja liitin, johon pää on kytketty. Jotta tiedot saataisiin taulukossa oikeaan sarakkeeseen helposti, kunkin tiedon tunniste oli hyvä olla taulukon sarakkeen otsikkoa vastaava merkkijono. Tavoitetietue näkyy kuviossa alla.

```
(
  ("Kaapelitunnus" . tieto)
  ("Kaapelityyppi" . tieto)
  ("Johtimen nimi" . tieto)
  ("Laite (mistä)" . tieto)
  ("Liitin (mistä)" . tieto)
  ("Laite (minne)" . tieto)
  ("Liitin (minne)" . tieto)
)
```

Kuvio 2. Tavoitetietue, jossa kukin tunnus merkkijonona

Kirjoitin ensin funktion, jolla tietoa voi hakea handlen ja sijainnin perusteella kaaviosta. Nimesin tämän funktion fetchSingleDatumiksi, koska sen tehtävä oli hakea yksi tieto annettujen lähtötietojen perusteella. Lähtötiedoiksi annettiin tietue, jonka kentät olivat “name”, “handle”, “type”, “x” ja “y” (ks. kuvio 3). Tämän lisäksi blokkien attribuuttitietoja haettaessa tietueeseen sisällytettiin myös attribuutin koordinaatit, “attx” ja “atty”, jotta voitaisiin verrata paremmin tekstin osuvuutta. Name- eli nimikenttä tarkoittaa tiedon otsikkoa, kuten “kaapelityyppi”. Tämän tiedon perusteella voidaan haettu teksti asettaa oikeaan sarakkeeseen taulukossa. Funktio palautti yksittäisen nimi-arvo-parin, sisältäen tekstin sisällön (ks. kuvio 4).

```
(
  ("NAME" . "Kaapelityyppi")
  ("HANDLE" . "5DA7")
  ("TYPE" . "TEXT")
  ("X" . 30.5)
  ("Y" . 100.0)
)
```

Kuvio 3. Hakufunktiolle annettu tietue

```
_> (fetchSingleDatum '(("NAME" . "Kaapelityyppi") ("HANDLE" . "5DA7") ("TYPE" . "TEXT") ("X" . 30.5) ("Y" . 100.0)))
("Kaapelityyppi" . "EX 3x1,5")
```

Kuvio 4. Funktiokutsu ja palautusarvo

Tätä komentoa toistamalla voidaan tuottaa tavoitetietueen kukin kenttä. FetchSingleDatum ei sellaisenaan ole erityisen hyödyllinen, koska hakukomentoa tehdessä pitäisi tietää haettavasta tekstistä hyvin tarkkoja yksityiskohtia. Jotta yksittäisten tietojen poimimiseen ei tarvitsisi käyttää liikaa aikaa, tämän haun generointiin kirjoitettiin komento, joka kysyy käyttäjältä, mistä tekstistä halutaan poimia tieto. Funktio poimii valitusta tekstistä lähtötiedot, ja tallentaa kokonaisen haun lopulta tiedostoon. Tiedostoon tallennetun hakukomennon voi sitten suorittaa tyyppikuvaa vastaavassa kaaviossa. Kyselyfunktion nimeksi annoin GetSingleDatum, koska sen tehtävä oli pyytää yksi lähtötieto käyttäjältä. Käyttäjää koskevat funktiot ovat AutoLISPissä yleensä get-alkuisia, joten ajattelin tämän nimen olevan sopiva. Yhden johtimen haku toteutettiin tässä vaiheessa kuvion 5 mukaisella komennolla, joka tuotti tuloksena kuvion 6 mukaisen hakukomennon.

```
(list
  (getSingleDatum "Kaapelitunnus")
  (getSingleDatum "Kaapelityyppi")
  (getSingleDatum "Johtimen nimi")
  (getSingleDatum "Laite (mistä)")
  (getSingleDatum "Liitin (mistä)")
  (getSingleDatum "Laite (minne)")
  (getSingleDatum "Liitin (minne)")
)
```

Kuvio 5. Johtimen haku

```
(
  (fetchSingleDatum '(("NAME" . "Kaapelitunnus") ... ))
  (fetchSingleDatum '(("NAME" . "Kaapelityyppi") ... ))
  (fetchSingleDatum '(("NAME" . "Johtimen nimi") ... ))
  (fetchSingleDatum '(("NAME" . "Laite (mistä)") ... ))
  (fetchSingleDatum '(("NAME" . "Liitin (mistä)") ... ))
  (fetchSingleDatum '(("NAME" . "Laite (minne)") ... ))
  (fetchSingleDatum '(("NAME" . "Liitin (minne)") ... ))
)
```

Kuvio 6. Tuotettu hakukomento

Komentoa tultaisiin käyttämään paljon, joten pyrin tässä vaiheessa vähentämään ylimääräistä työtä. Kaapeleita on kuvassa monta ja kussakin kaapelissa on monta joh-

dinta. Kaapelin kullekin johtimelle halutaan täyttää sama kaapelitunnus ja tyyppi, joten olisi hyvä, jos näitä ei kysyttäisi useaan kertaan käyttäjältä. Jotta tämän voisi tehdä, tuli kaapelin tiedot ja johtimen tiedot irrottaa toisistaan, jotta ne voisi kysyä erikseen ja liittää myöhemmin yhteen. Ennen tätä muutosta johtimien määrällä ei ollut väliä, koska komento oli täysin irrallinen haettavasta tiedosta. Komentoa piti vain toistaa niin kauan, kunnes kaikki tieto on kerätty. Kun yksittäisen kaapelin ja usean johtimen tiedot pitää liittää myöhemmin yhteen, täytyy suunnitella, miten usean johtimen tiedot kerätään.

Päädyin rakentamaan kaapelitietueeseen listan sen sisältämistä johtimista. Lopullinen tietue ei salli tämänkaltaista sisäkkäistä tietoa, koska sen on tarkoitus olla rivikohtaista tietoa taulukossa. Tietorakenteen muutokseen tarvittiin apufunktioita tasoittamaan lista lopulliseen formaattiin sopivaksi. Tein nämä funktiot sallimaan syvempiäkin tietorakenteita, jotta ohjelmaa voidaan käyttää jatkossa moniulotteisemman tiedon keräykseen. Kirjoitetun hakukomennon takia tämä tiedon käsittely tuli tapahtua itse tiedon keräyksen jälkeen, mikä osoittautui myöhemmin virheeksi ohjelman monimutkaisuuden takia.

5.3 Tarve muutokselle, ongelma käyttöliittymässä

Työkalua kokeiltiin instrumentointiryhmän kaapelitietojen keruussa kahden tyyppikuvan perusteella. Kokeilussa huomattiin, että hakupohjan teko oli hyvin hidasta ja työlästä. Huomasin vaivan myös itse, kun aloin tehdä hakupohjia. Tällä vältettiin hakemasta useasta kuvasta tietoa manuaalisesti, joten työkalusta oli hyötyä jo tässä vaiheessa, mutta työn mielekkyys ei parantunut, sillä tekemäni käyttöliittymä hidasti tietojen valitsemista. Käyttäjältä kysyttiin myös moneen kertaan saman muotoista dataa, kuten esimerkiksi saman näköisiä kaapeleita. Jos kuvassa oli viisi samanlaista kaapelia tai tyyppikuvien kesken oli samanlaisia kaapeleita, tämä samankaltaisuus jäi kokonaan hyödyntämättä. Käyttöliittymä auttoi toimivan mallin luomisessa muistiin, mutta käyttäjälle se esiintyi ylimääräisenä työnä. Jos työkalua aiottiin käyttää jatkossa moniin kuvapohjiin, toiminnon oli oltava intuitiivisempi käyttää.

Suurin osa haettavista kaapeleista toistivat jo kerättyjen kaapeleiden ulkomuotoa. Kaapeliin liittyvät tiedot olivat kuvassa suurin piirtein yhtä etäällä toisistaan. Toistuvien kokonaisuuksien syötöltä haluttiin välttyä, joten piti tutkia, miten ohjelma voisi oppia valitsemaan kaapeliin sopivat tekstit. Visual Lisp on vain AutoCADilla ja muilla CAD-ohjelmilla toimiva ohjelmointikieli. Näiden ohjelmien käyttäjäkunnasta suurin osa ei käytä lispiä ollenkaan ja ohjelmointiin tarkemmin perehtyneitä on sitäkin vähemmän, minkä takia valmiita ohjelmakirjastoja koneoppimiseen ei ole saatavissa. Visual lisp on myös hidas kieli suurien datamäärien käsittelyyn, koska lisp ei anna mutatoida mitään arvoa paikallaan, vaan jokaisella operaatiolla luodaan kopio edellisestä tiedosta. Monet tietorakenteet puuttuvat kielestä kokonaan. Näiden ja projektien ajallisten rajoitteiden vuoksi päädyttiin alkeelliseen instanssiperusteiseen oppimiseen.

Instanssiperusteista oppimista kutsutaan välillä laiskaksi oppimiseksi tai myös muistiperusteiseksi oppimiseksi, koska datalle ei yritetä luoda tarkkaa luokitteluskeemaa tietoa syöttäessä, vaan kukin näyte, instanssi, tallennetaan muistiin sellaisenaan. Uutta näytettä lisätessä käytetään k-lähimmän naapurin hakua, jonka avulla lasketaan lähin vastaavuus uudelle tiedolle. K-lähimmän naapurin haulla luodaan jonkinasteinen abstraktio kerätylle tiedolle, jolloin algoritmi soveltuu myös uusien tapauksien luokitteluun. (Aha, Kibler & Albert 1991)

Käyttämäni algoritmi oli alkeellinen, koska sen sijaan, että olisin käyttänyt k-lähimmän naapurin hakua tiedon abstraktointiin, käytin normaalia hakua. Tarpeeksi lähelle osuvan instanssin löytyessä uusi instanssi luokiteltiin kuuluvaksi tämän ryhmään ja mikäli tarpeeksi lähelle osuvaa instanssia ei löydy muistista, uusi näyte lisätään muistiin uutena tapauksena. Projektin kytkentätietojen tapauksessa tämä tarkoitti sitä, että kun kaapelitietue kerätään, sen sisältämien tekstien sijainnit toisistaan tallennetaan muistiin, jotta niiden perusteella voidaan myöhemmin hakea vastaavaa kuviota. Oppimisalgoritmina tämä on todella huono, koska en yrittänyt luoda tiedolle abstraktiota, mutta algoritmi oli helppo toteuttaa ja sillä voitiin hyvin vähällä datalla vähentää suunnittelijan täyttämien duplikaattien määrää. Algoritmin voisi luokitella paremminkin hakualgoritmiksi kuin oppimiseksi.

Toisessa projektissa kaivattiin AutoCAD-taulukon irrotusta kaaviosta. Nämäkin taulukot oli kasattu rakenteettomilla tekstiobjekteilla, joten ohjelmalle oli tarvetta, jotta taulukot saataisiin ymmärrettävässä muodossa ulos. Tämän tyyppisen haun luomiseen kirjoitettiin funktio, joka toistaa haun komentoja siirtäen koordinaatteja käyttäjältä kysytyn etäisyyden päähän. Tässä hakukomento toimi mainiosti, koska kaaviopohjia oli vain yksi, minkä takia ohjelman ei tarvinnut välittää erilaisista kuvioista. Tällä haulla ajettiin taulukot monesta kymmenestä kaaviosta.

5.4 Algoritmin optimointi

Kun ohjelma oli suurimmalta osin valmis, alkoi testaus irrallisiin kaavioihin. Muutaman kaavion jälkeen huomattiin, että ohjelma hidastui nopeasti. Ensin ajattelin tämän johtuvan instanssiperusteisen oppimisen huonoista puolista. Tietoa ei ollut mitenkään abstraktoitu, minkä takia hakuun käytetty aika kasvaa lineaarisesti näytteiden määrän kanssa. Tämän ei kuitenkaan pitäisi johtaa yli minuutin mittaisiin hakuihin parillakymmenellä kuviolla. Myöhemmin haun kesto kasvoi monen minuutin mittaiseksi. Ohjelmassa oli siis muutakin vikaa, minkä takia ohjelmaa oli syytä optimoida.

Pistettä lähimmän tekstin hausta löytyi yksi aikasyöppö. Kun haettiin lähintä tekstiä, funktio järjesti kaikki kuvan tekstit pisteeseen mitatun etäisyyden perusteella, jonka jälkeen tästä järjestetystä listasta poimittiin ensimmäinen, ja loppulista jätettiin käyttämättä. Listan järjestäminen on turhaa, koska järjestettyä listaa ei käytetä myöhemmin. Sen sijaan lista voidaan käydä kertaalleen läpi valiten yksi teksti pienimmän etäisyyden perusteella. Listan järjestäminen on O(nlogn)-tasoinen operaatio kun taas pienimmän etäisyyden valinta on O(n)-operaatio. Tämän lisäksi lähintä tekstiä hakiessa noudettiin kaikki kuvan tekstientiteetit uudestaan tietokannasta ja luotiin uusi assosiaatiolista entget-komennolla. Koska tietokantaan ei tehtäisi tietojen hakemisen aikana muutoksia, toistuvan hakemisen sijaan assosiaatiolistat voitaisiin hakea vain kerran muistiin ja käyttää muistissa olevaa versiota myöhemmissä hauissa. Assosiaatiolistojen luonti on aikavaativuudeltaan O(n), mikä muutoksen jälkeen tehtiin vain kerran kuvaa kohden.

Kuvio valittiin etsimällä kuvion kutakin tekstiä kuvasta, ja jos kaikki tekstit löytyivät, ehdotettiin tätä kuviota käyttäjälle. Kunkin kuvion kaikki tekstit käytiin läpi vaikka jokin teksteistä ei olisikaan löytynyt. Muutin tämän vaihtamaan kuviota seuraavaan heti, kun kuviossa tapahtui ensimmäinen virhe. Tällä saatiin karsittua huomattavasti turhia hakuja. Samalla kuitenkin tajusin, että koska kuviohaku perustui lähimmän tekstin hakuun, nämä yhdessä loivat O(m * n) algoritmin, jossa m on testattavien kuvioiden määrä ja n on kuvassa olevien tekstien määrä. O(m * n) on pahimmillaan $O(n^2)$ silloin, kun m ja n ovat samansuuruiset, minkä takia komento alkoi viedä nopeasti enemmän aikaa kuvioiden määrän kasvaessa. Tämänlainen kompleksisuus ei ole hyväksyttävää, jos käyttäjän pitää odotella ohjelman suorituksen välissä, joten jompaakumpaa hakua piti nopeuttaa huomattavasti.

Pistettä lähimmän tekstin hakuun on olemassa monia eri vaihtoehtoja, minkä takia päädyin kehittämään tätä hakua. Päädyin rakentamaan tekstien taustalle graafin eli verkon, minkä perusteella pystyisi hakemaan lähimpää tekstiä toisten tekstien kautta closest-first-haulla. Verkko rakennettaisiin triangulaation eli kolmioinnin avulla.

Triangulaatio tarkoittaa, että pisteistä muodostuva pinta jaetaan kolmioihin liittämällä pisteet toisiinsa. Kun kolmioiden kyljet eli pisteiden väleille piirretyt viivat eivät kulje toistensa päältä, kutsutaan graafia maksimitasograafiksi. Delaunayn triangulaatio-algoritmi pyrkii luomaan maksimitasograafin kolmioinnin, jossa kunkin kolmion ympäröivän, kolmion pisteiden läpi kulkevan, ympyrän sisällä ei ole muita pisteitä. Tästä kolmioinnista muodostuva verkko on ominaisuuksiltaan erityisen hyvä lähimmän tekstin hakuun. Koska algoritmi pyrkii minimoimaan kunkin kolmion ympäröivän ympyrän kokoa, verkon rakenne on tasainen ja kukin piste yhdistyy sen lähimpiin ympäröiviin pisteisiin. Kaikki pisteet yhdistyvät niitä ympäröiviin pisteisiin, joten verkkoa pitkin voi kulkea haettua pistettä kohti ilman esteitä. (Delaunay triangulation n.d.)

Kun triangulaatio on rakennettu, voidaan tekstientiteetit liittää yhteen yhdeksi verkoksi kolmioiden kylkiä pitkin. Verkon rakentamiseen kuluu muutama sekunti aikaa, mutta koska tämä verkko tarvitsee rakentaa vain kerran avattua kaaviota kohden, se ei haittaa työntekoa liiaksi.

Tästä verkosta tekstiä haettaessa voidaan lähteä jostain tekstistä liikkeelle, verrata sen naapureiden etäisyyksiä haettuun pisteeseen ja valita näistä paras. Sama toistetaan valitulle tekstille, kunnes valitun tekstin naapureista ei enää löydy pistettä lähempänä olevia tekstejä. Tällöin voidaan todeta valitun tekstin olevan lähimpänä pistettä. Tätä kutsutaan closest-first-hauksi, koska naapureista valitaan aina kohdetta lähinnä oleva piste jatkoon. Viimeksi haettu piste tallennetaan seuraavaksi lähtöpisteeksi, koska usein peräkkäiset haut ovat lähellä toisiaan. Tämän algoritmin tehokkuus on suurimmassa osassa tapauksista O(logn), jolloin kuviohaun kokonaiskompleksisuudeksi saadaan käyttökelpoinen O(nlogn). Pahimmillaan verkosta haku vaatii O(n)-operaation, mutta nämä tapaukset ovat harvassa, koska Delaunayn triangulaatio pyrkii minimoimaan tällaiset tilanteet. (Cazals 2016)

Delaunayn triangulaatiossa tehtävän closest-first-haun aikavaativuus O(logn) perustuu siihen, että haun rakenne voidaan muodostaa puuksi, jonka kerros on edetyn haaran naapurit. Tämän puun syvyys on naapureiden määrän pohjainen logaritmi n:stä, eli logn. Kussakin kerroksessa täytyy tarkastaa kunkin naapurin etäisyys haettuun pisteeseen, mutta koska naapureiden määrä on normaalisti lähes vakio, tämän merkitys asymptoottiseen suoritusaikaan on pieni ja se voidaan jättää huomiotta. (Cazals 2016; Grimson 2017)

Näiden jälkeen eniten aikaa vievä osa ohjelmassa oli roskien keräys (engl. garbage collection). Lispissä kaikki arvot ovat muuttumattomia. Muuttujan arvoa muuttaessa muokataan muistissa ainoastaan kopiota edellisestä arvosta, minkä takia vanha arvo voi jäädä käyttämättömäksi. Roskien keräyksellä tarkoitetaan näiden käyttämättömien arvojen puhdistusta muistista, jotta ne eivät turhaan veisi muistista tilaa. Tämä voi viedä huomattavasti suoritusaikaa, minkä takia turhien kopioiden luomista on hyvä välttää. Moni kirjoittamani aliohjelma loi listoja tallentamalla niitä muuttujaan silmukan sisällä. Koska pienen muutoksen tapahtuessa tämä luo usein kopion edellisestä listasta, silmukka kehitti paljon roskaa. Listojen kopiointia ei tapahdu, jos listan osia käytetään ohjelmakutsussa, joten silmukoiden sijaan oli hyvä suosia rekursiota ohjelmia rakentaessa. Rekursion rajoitteena on kutsupino, joka tukee kutsuja noin 15000 syvyyteen asti. Tämä ei haittaa suurimmassa osassa ohjelmista, joten muutin aliohjelmia suorittamaan tehtävänsä rekursiivisesti.

Ohjelman toiminta saatiin näillä optimoinneilla puristettua monen minuutin hausta muutaman sekunnin hakuun.

6 Johtopäätökset ja pohdinta

Lopputuloksena saatiin ohjelma, jolla voidaan tuoda irrallisten tekstien sisältöä taulukkoon vähemmällä työllä kuin kaavioiden yksittäin tarkastelulla. Kytkentätietojen tuonti taulukkoon vaati haun luomisen 70 tyyppikaaviolle, joilla saatiin haettua 800 kaavion tiedot. Tämän lisäksi ohjelmalla pystyi keräämään kytkentätiedot 200 irrallisesta kaaviosta nopeammin, kuin jos tietoa olisi täyttänyt taulukkoon käsin. Toisen projektin käytössä vaadittiin vain yksi tyyppikuvahaku, jolla kerättiin tiedot muutamasta kymmenestä kaaviosta. Työkalua voi soveltaa monenlaiseen dataan, mutta opinnäytetyön aikana ei ehditty tehdä käyttöliittymää monenlaisen datan keruuseen. Tämän sijaan kustakin eri keräyskohteesta täytyy kirjoittaa oma lisp-komento, mikä ei ole käyttäjäystävällistä. Mikäli ohjelmalle tulee paljon käyttöä, siihen olisi hyvä kehittää komento, jolla voi lisätä ja muokata haettavia tietuetyyppejä. Toinen vaihtoehto olisi laittaa ohjelma lukemaan haettava tietuetyyppi tallennustiedoston otsikoista.

Jatkossa ohjelmaan voisi lisätä pelkkien kuvioiden perusteella hakemisen. Tyyppikaavioiden käyttö vaatii useiden samankaltaisten pohjien tekemisen, vaikka näissä käytettäisiinkin samoja hakukuvioita. Kuvion voisi tunnistaa jonkin kentän sisällän perusteella. Esimerkiksi kaapelitunnus olisi helposti tunnistettavissa regex-haulla, minkä jälkeen tätä kaapelitunnusta vasten voisi verrata kuvioita. Mikäli jostain kaapelista ei saada haettua tietoa jonkin olemassa olevan kuvion perusteella, pyydettäisiin käyttäjää luomaan uusi kuvio. Näin tyyppikaavioita ei välttämättä tarvitse haun tekoon ollenkaan, vaan automaattisen haun osuus kasvaa kuvioiden kertyessä.

Regex-hakuihin liittyen, automaattisten hakujen luomiseen voisi kehittää haetun tiedon validoimista. Jos tiedetään jonkin tiedon noudattavan regex- tai wildcard-kuviota, voitaisiin tarkastaa, onko tieto oikeanlaista, ja siten voitaisiin paremmin arvioida tiedon sisällyttämistä tuloksiin.

Lispin syntaksin hyödyt tulivat hakukomentojen teossa esiin. Lispin ohjelmakoodi voidaan lukea listarakenteena kuin mikä tahansa muu listamuotoinen data. Tämän takia ohjelmakoodia on myös helppo käsitellä kuin listaa. Ohjelmia on helppo generoida, kun niitä voi käsitellä suoraan kieleen rakennetuilla menetelmillä. Huomasin kuitenkin tekeväni tätä komentojen generointia liian aikaisessa vaiheessa, mikä vaikeutti myöhempää tiedon käsittelyä.

Valitsemani komentorakenne on hyödyllinen, koska se on suoraan verrattavissa rakennettaviin tietueisiin. Se näyttää, miten tieto rakentuu annetuista komennoista ja on helposti muokattavissa erilaisiin kohteisiin. Sen sijaan komentojen tuottama lispkoodi oli lyhytkatseinen päätös, sillä se loi monia rajoitteita monipuolisempien funktioiden luontiin. Koska data oli liimattuna komentoihin, datan käsittely vaati jokaiselta aliohjelmalta datan kaivamisen komentorakenteesta, mikä johti funktioiden riippuvuuteen toisistaan. Ohjelmien muokattavuus ja uudelleenkäyttömahdollisuudet kärsivät tämänlaisen logiikan toistamisen takia. Komennot estivät minua käsittelemästä dataa erillisinä ryhminä, minkä takia jouduin kirjoittamaan monta apuohjelmaa tiedon käsittelyyn. Esimerkiksi monen johtimen kaapelitietuetta rakentaessa olisin säästynyt vähemmällä työllä, jos olisin tehnyt johtimista ja kaapeleista erilliset ryhmät, jotka olisin liittänyt myöhemmin yhteen. Tämä olisi onnistunut lispin valmiilla funktiolla append. Jatkokehityksen kannalta olisi hyvä muuttaa käytetyt komennot tuottamaan tietoa eikä uusia komentoja, koska tämä tekee hakukohteiden muokkaamisesta helpompaa ja katkaisee riippuvuudet komentojen välillä. Muokattuja hakutietueita olisi tämän jälkeen helpompi käyttää tehokkaaseen hakupohjien luontiin.

Myöhemmin projekteissa huomattiin, että ohjelma tuo oikeat tiedot taulukkoon, mutta koska esimerkiksi kaapeli saattoi esiintyä monessa kuvassa, samoista johtimista luotiin taulukkoon monikertoja. Jatkossa voisi kehittää tavan yksilöidä tietueita, jotta haku voisi tunnistaa, milloin yritetään syöttää jo haetun johtimen tietoja. Tähän voisi keksiä tavan yhdistää rivien tiedot keskenään jo haun aikana. Samalla voisi käsitellä näiden monikertojen ristiriidat, mikäli niitä nousee.

Visual lisp on helppo kieli ymmärtää ja käyttää, mutta koska se on vain pienen ryhmän käytössä oleva ohjelmointikieli ja toiminnaltaan rajoitettu, sille ei ole tehty valmiita ohjelmakirjastoja, joita voisi ohjelmoinnissa hyödyntää. Kirjastojen puuttuminen vaatii paljon työtä ohjelmoijalta, koska kukin pieni apuohjelma tulee ohjelmoida itse. Huomasin projektin aikana tekeväni huomattavasti sellaisia funktioita, jotka ovat muissa ohjelmointikielissä jo standardikirjastossa. Samoin huomasin tekeväni funktioita, joista löytyy huomattavasti kirjastoja toisilla kielillä kirjoitettuna. Kunkin tällaisen ohjelman kirjoitus vaatii aikaa. Kirjastojen hyöty on siinä, että joku muu on jo tehnyt kehitystyön ja testauksen, jolloin kirjaston käyttäjä säästää aikaa.

Näitä monia funktioita kirjoittaessa kaipasin staattista tyyppitarkastusta. Kirjoitusvaiheen tyypittömyys ei pienissä projekteissa haittaa, mutta kun projektin sisältö alkaa kasvaa, yhden muutoksen tekeminen voi vaikuttaa moneen paikkaan. Esimerkiksi funktion parametreja muokatessa kukin funktion kutsupaikka voi vaatia muutosta. Nämä kutsupaikat voivat sijaita monessa eri tiedostossa, minkä takia kaikkien löytäminen ja muokkaaminen vie aikaa ja jokin niistä voi jäädä muokkaamatta. Lisp ei kääntövaiheessa tarkasta, onko kaikkia tarvittavia funktioita olemassa tai minkä tyyppisiä parametreja ne ottavat. Tästä syystä virheen huomaa vasta, kun jokin komento ei toimikaan kesken ohjelman suorituksen. Virhe voi pysyä piilossa pitkäänkin, jolloin virheen syyn löytäminen vaikeutuu. Tiukemman tyyppijärjestelmän avulla näitä virheitä ei pääsisi muodostumaan alkuunkaan.

Tekstin haku triangulaation avulla tehdystä verkosta closest-first-periaatteella ei ollut varmin tapa saavuttaa O(logn)-tehokkuus. Pahimmassa tapauksessa jokin teksti on yhteydessä huomattavasti suurempaan määrään tekstejä, jolloin käydään läpi kaikki nämäkin tekstit. Kirjoittamani triangulaatio ei myöskään toiminut, jos pisteitä oli päällekkäin. Koska aika ei projektilla tahtonut riittää, ratkaisin tämän siirtämällä päällekkäiset pisteet verkon ulkopuolelle, jolloin näistä haku vaatii edelleen kunkin päällekkäisen tekstin läpikäynnin. Päällekkäisiä pisteitä ei onneksi yleensä ole monia kuvissa. Jälkikäteen ajatellen olisi ollut parempi tehdä binääripuurakenne, jonne tekstit olisi sijoitettu. Binääristä avaruuden osiointia on käytetty hakurakenteiden luomiseen ennenkin, ja siihen olisi saatavilla myös paljon materiaalia. Tällä olisi taattu O(logn)-haku. Puun käsitteleminen olisi ollut myös huomattavasti helpompaa lispissä.

Triangulaatioverkon rakentaminen vaatii edelleen sekunnista kolmeen sekuntiin aikaa. Verkon rakentaminen on hidasta, koska ohjelmassa joudutaan käymään tekstijoukko monesti läpi, kun naapuripisteitä yhdistetään indeksilinkeillä. Tätä voisi nopeuttaa käyttämällä lispin symboleita hajautustaulun tapaan, jolloin kukin teksti olisi yhden symbolin takana, ja sallisi näin lähes välittömän tiedon muokkauksen ja linkkien seuraamisen. Tämä vaatisi symbolien generoinnin niin, etteivät ne yliaja muita käytettyjä symboleja. Symboleiden käyttö tähän tapaan kuitenkin pakottaa ohjelmoijan huolehtimaan muistin puhdistamisesta itse, koska roskien kerääjä ei tyhjennä symboleita automaattisesti kuin vain kaaviota suljettaessa. Yksittäiseen ohjelmaan tämänlainen ratkaisu voisi toimia, mutta globaalien symbolien käyttö tähän tapaan on haitallista muiden ohjelmien kannalta.

Kuvioiden tallennukseen käytin normaalia listaa. Tästä johtui kuvioiden lineaarinen aikavaade; kukin kuvio tuli käydä läpi yksitellen. Tämä johtui osittain myös siitä, että tallensin kuviot suoraan hakukomentomuodossa, enkä yrittänyt irrottaa dataa näistä hakukomennoista. Komennot sisälsivät lisätietoa siitä, miten tekstejä tulisi käsitellä haun jälkeen, joten tämä tallennusmuoto oli joltain osin suotavaa. Kuitenkin olisin voinut irrottaa puhtaan hakutiedon johonkin toiseen rakenteeseen, jossa sen käsittely olisi ollut helpompaa. Jatkoa varten tähän olisi hyvä kehittää jokin parempi tallennuskeino, joka sallisi työn vähentämisen ja säästäisi tilaa. Yksi vaihtoehto olisi rakentaa puurakenne tietueista. Puu haarautuisi kahdella tavalla: kentän nimen ja kentän sijainnin perusteella. Kentän nimi jakaisi puun haetun tietueen mukaan, esimerkiksi kun haetaan kaapeleiden kuvioita, nimi “kaapelitunnus” aloittaisi haaran kuvioista, jotka alkavat kaapelitunnuksella. Jos jokin kenttä on eri kuvioissa samassa sijainnissa, tämä tietorakenne kompressoisi tiedon yhteen pisteeseen. Jatkoa varten kuviohakua voisi välittömästi optimoida tallentamalla haetut sijainnit muistiin ja tarkastamalla niistä onko jostain sijainnista jo haettu tekstiä. Tällä tavoin kuviohaku hakisi tekstiä kustakin sijainnista vain kerran kaaviota kohden.

Kuvioiden oppimiseen käytettiin vain kuvioiden tallennusta ja näistä lähimmän osuvuuden etsintää. Kun lähin osuvuus löytyi, sen sisältämää tietoa ei yritetty korjata tai abstraktoida. Jotta osumat täsmäisivät jatkossa paremmin ja ohjelma oppisi yleispätevämpiä kuvioita, tallennusvaiheessa voisi lähimmän osuman tietojen sijainneista

laskea juoksevaa keksiarvoa. Kuvioista voisi myös yrittää etsiä yhtäläisyyksiä, joiden perusteella voitaisiin monta eri kuviota yhdistää, millä säästettäisiin hakuhetkellä tehtävää laskentaa. Tiedon abstraktoimiseen suurempi vaikutus saataisiin kuitenkin k-lähintä naapuria-haulla, mutta tämä voisi vaatia enemmän voimaa ohjelmointikieleltä.

Lisp on hidas kieli verrattuna muihin kieliin, joilla AutoCADia voi ohjelmoida. Jos ohjelman olisi toteuttanut C#:lla tai C++:lla, ohjelman optimointia ei olisi todennäköisesti tarvinnut kuin huomattavasti myöhemmin, jos ollenkaan. Valmiiden kirjastojen saatavilla oleminen olisi vähentänyt optimoinnin ongelmia myös huomattavasti, koska minun ei olisi tarvinnut kirjoittaa jokaista toimintoa itse.

Tuotetun ohjelman tarve on rajallinen. Oikea ratkaisu piilee suunnittelukäytännön korjaamisessa tietoa helpommin ylläpitävään muotoon. Tähän on olemassa tietokantapohjaisia suunnitteluohjelmistoja. Näihin siirtymisellä vältetään suurin osa opinnäytetyössä kohdatuista ongelmista. Siirtyminen kuitenkin vaatii, että näiden suunnittelujärjestelmien käyttö olisi helppoa tai siihen olisi saatavilla koulutusta.

Lähteet

Aha, D. W., Kibler, D. & Albert, M. K. 1991. Instance-Based Learning Algorithms. Boston: Kluwer Academic Publishers

Alama, J. & Korbmacher, J. 2019. The Lambda Calculus. Viitattu 28.5.2020. https://plato.stanford.edu/archives/spr2019/entries/lambda-calculus/

Ambrosius, L. & Byrnes, D. 2006. AutoCAD and AutoCAD LT All-in-One Desk Reference for Dummies. Indianapolis: Wiley Publishing, Inc.

AutoCAD DXF N.d. Artikkeli DXF-formaatista Wikipediassa. Viitattu 28.5.2020. https://en.wikipedia.org/wiki/AutoCAD_DXF

AutoLISP N.d. Artikkeli AutoLISPistä Wikipediassa. Viitattu 28.5.2020. https://en.wikipedia.org/wiki/AutoLISP

AutoLISP: Reference 2018. AutoCADin sisäisestä helpistä löytyvä dokumentaatio. Ohjelma Autodesk AutoCAD 2018.

Binary search algorithm N.d. Artikkeli puolitushausta Wikipediassa. Viitattu 28.5.2020. https://en.wikipedia.org/wiki/Binary_search_algorithm

Cazals, F. 2016. Msc. in Data Science: Nearest Neighbors Algorithms in Euclidean and Metric Spaces. Luentovideo CentraleSupélec -YouTube-kanavalla. Viitattu 28.5.2020. https://www.youtube.com/watch?v=rho8QqiHOe4

Delaunay triangulation N.d. Artikkeli Delaunayn triangulaatiosta Wikipediassa. Viitattu 28.5.2020. https://en.wikipedia.org/wiki/Delaunay_triangulation

Functional programming N.d. Artikkeli funktionaalisesta ohjelmoinnista Wikipediassa. Viitattu 28.5.2020. https://en.wikipedia.org/wiki/Functional_programming

Grimson, E. 2016. Understanding Program Efficiency. Luentovideot MIT OpenCourseWare -YouTube-kanavalla. Part 1 ja Part 2 julkaistu 15.2.2017. Viitattu 28.5.2020. https://www.youtube.com/watch?v=o9nW0uBqvEo ; https://www.youtube.com/watch?v=7lQXYl_L28w

Hutton G. 2017. Lambda Calculus. Video Computerphile -YouTube-kanavalla, julkaistu 27.1.2017. Viitattu 28.5.2020. https://www.youtube.com/watch?v=eis11j_iGMs

Learn Clojure - Syntax. N.d. Clojure-opas kielen syntaksiin Clojuren virallisilla sivuilla. Viitattu 28.5.2020. https://clojure.org/guides/learn/syntax

Liukko, S. & Perttula, S. 2019. Opinnäytetyön raportointi. Viitattu 27.5.2020. https://oppimateriaalit.jamk.fi/raportointiohje/4-opinnaytetyon-rakenne/4-2-opinnaytetyon-runko-osa/4-2-1-erilaisia-rakenteita/

McCarthy, J. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Massachusetts: Massachusetts Institute of Technology

Poleschuk, N. 2001. AutoCAD Developer's Guide to Visual LISP. Pennsylvania: A-LIST, LLC

Pyhäjärvi, M. & Pöyhönen, E. 2006. Ohjelmistojen testaus. Kurssimateriaali. Viitattu 27.5.2020. http://users.jyu.fi/~kolli/testaus2006/materiaali/Maaret_27102006.pdf

Salonen, K., Eloranta, S., Hautala, T. & Kinos, S. 2017. Kehittämistoiminta ja kehittämisen menetelmiä ammatillisessa korkeakoulutuksessa. Turku: Turun ammattikorkeakoulu.

Sorri, S. N.d. Yritys. Viitattu 27.5.2020. https://www.rejlers.fi/Yritys