Prolog-perusteinen konstruktori-orientoitunut lähestymistapa asiantuntijajärjestelmän nopeaan prototyypitykseen

Markus Salakari

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Timo Niemi
Huhtikuu 2008

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Tekijän Nimi: Markus Salakari
Pro gradu -tutkielma, 54 sivua, joista 8 liitesivua
Huhtikuu 2008

Tämä pro gradu -tutkielma tarkastelee asiantuntijajärjestelmiä, nopeata prototyypitystä ja tiedon rakenteistamista konstruktoriorientoituneesti. Näiden kolmen mielenkiintokohteen kohtaamista tarkastellaan erityisesti Prolog-ohjelmointikielellä toteutetussa esimerkkiohjelmistossa. Tämä konstruktori-orientoituneesti Prologilla ohjelmoitu RoutePlanner-reitinsuunnitteluasiantuntijaohjelmisto, on toteutettu pikaprototyypityksen periaatteita noudattaen ja antaa näin mahdollisuuden tarkastella konstruktori-orientoituneen Prolog-ohjelmoinnin soveltuvuutta juuri tutkielman esimerkkiohjelmiston kaltaisen asiantuntijajärjestelmän pikaprototyypitykseen.

RoutePlanner-reitinsuunnittelijan toteutuksen tuloksena on sellaisen asiantuntijajärjestelmän onnistunut pikaprototyyppi, jossa tiedon rakenteellisuus on toteutettu konstruktoreilla. RoutePlanner selviää sille asetetuista käyttötapaustestauksista ja prototyypitystä voidaan siksi pitää onnistuneena. Esimerkkiohjelmiston onnistunut toteutus antaa olettaa, että konstruktori-orientoitunut Prolog-ohjelmointi soveltuu pikaprototyypitysmenetelmäksi sellaisille asiantuntijajärjestelmille, joiden sisältämän tiedon rakenteellinen hallinta on tärkeää.

Avainsanat ja -sanonnat: Prolog, konstruktori-orientoitunut lähestymistapa, tiedon rakenteistaminen konstruktoreilla, nopea prototyypitys, asiantuntijajärjestelmät.

Sisällysluettelo

1. Johdanto

Tässä pro gradu -tutkielmassa käsitellään *nopeaa prototyypitystä* (rapid prototyping), *konstruktoriorientoitunutta lähestymistapaa* ja *asiantuntijajärjestelmiä*. Erityisesti kiinnitetään huomiota näiden kolmen alueen liittymiseen toisiinsa *Prolog-ohjelmointikielellä* toteutetussa RoutePlanner-prototyypissä. RoutePlanner on prototyyppi sellaiselle asiantuntijajärjestelmälle, joka avustaa käyttäjäänsä matkareitin suunnittelussa ottaen huomioon erilaisia käyttäjän määrittelemiä rajoitteita koskien tietämyskannan tietoja matkustusmuodoista ja -yhteyksistä.

Nopea prototyypitys on menetelmä, jolla varmistetaan ohjelmiston vaatimusmäärittelyiden ja käyttötapausten kattavuus. Samalla varmennetaan tilaajan vaatimukset ja varmistetaan, että ohjelmistoprojekti on mahdollista viedä päätökseen asti. Konstruktoriorientoituneisuus on ajattelutapa rakenteellisen tiedon hallintaan. Loogisten konstruktorien avulla on mahdollista rakenteistaa tietoa myös logiikkaohjelmointiympäristössä, jonka ainoana tiedon esitystapana on looginen termi. Prolog valitaan toteutusmenetelmäksi tutkielman logiikkaohjelmointisuuntautuneisuuden vuoksi. Asiantuntijajärjestelmän tietämyskanta sisältää usein rakenteellista tietoa, jonka esittäminen konstruktoreilla on kiinnostava mahdollisuus. Tutkielman neljä mielenkiintokohdetta liittyvät siis intuitiivisesti toisiinsa. Siksi on kiinnostavaa testata niiden todellista soveltuvuutta käytännössä.

Ensin tutkielmassa tutustutaan mielenkiintokohteisiin omissa luvuissaan (luvut 2, 3 ja 4). Aihealueiden perusteiden pintapuolisen tarkastelun jälkeen pohditaan mielenkiintokohteiden keskinäistä soveltuvuutta edelleen teoriassa (luku 5). Tämän tutkielman perustana on kuitenkin kiinnostus näiden asioiden testaamiseen käytännössä. Tästä tuloksena on em. esimerkkiohjelmiston toteutus. Niinpä tämän RoutePlanner-esimerkkiohjelmiston toimintaa ja ohjelmointiratkaisuja sekä jatkokehitysmahdollisuuksia tarkastellaan yksityiskohtaisesti luvussa 6.

2. Nopea prototyypitys

2.1. Nopean prototyypityksen periaatteet

Nykyisessä tuottavuutta ja tehokkuutta painottavassa ohjelmistotuotannossa pyritään nopeuteen, tehokkuuteen ja kulujen karsimiseen. Samalla on kuitenkin varmistettava myös ohjelmistojen laatu ja toiminnallisuuden kattavuus, jotta asiakas on tyytyväinen saamaansa tuotteeseen. Näiden ongelmien ratkaisuksi on kehitetty nopean prototyypityksen käsite. Nopealla prototyypillä mahdollistetaan nopeasti ja pienillä resursseilla ajettava prototyyppi eli koeversio kehitettävästä ohjelmistosta. Ajettavasta prototyypistä on jo nähtävissä varsinaisen ohjelmiston keskeiset piirteet ja sen rakentamisen ongelmakohdat. Prototyyppi antaa myös ohjelmiston tilaajan ja valmistaja välille konkreettisen pohjan ja yhteisen lähtökohdan keskustella ja sopia ohjelmiston vaatimuksista. Prototyypin avulla voidaan myös kirjata keskeiset ominaisuudet, jotka varsinaisen ohjelmiston on täytettävä.

Ennen varsinaisen ohjelmiston toteuttamista ja siihen tarvittavien resurssien käyttöä on tärkeää selvittää sekä toteuttamisessa mahdollisesti kohdattavat ongelmat että varsinaisen ohjelmiston toiminnalliset vaatimukset. Vaatimukset vaikuttavat erityisesti siihen, mitä tietoja tarvitaan ja miten tämä tietämys tulisi järjestelmässä esittää. Keskeisenä osana tiedon mallintamiseen liittyy myös tietoihin kohdistuvan käsittelytarpeen kartoitus. Toisin sanoen on selvitettävä, miten tietoja tullaan käsittelemään, jotta tiedoille voidaan löytää tarkoituksenmukainen esitystapa. Tähän tarkoitukseen on prototyyppi hyvä väline. Prototyypillä voi suorittaa monenlaisia testauksia ja niiden avulla valmistajan ja tilaajan on mahdollista tarkentaa ohjelmiston vaatimuksia. Prototyypin avulla pyritään varmistamaan yhteisymmärrys ohjelmistoon liittyvästä toiminnallisuudesta.

Perinteisesti prototyyppi on ollut lähinnä vain kehitettävän ohjelmiston varhainen kehitysversio, jota on sitten edelleen kehitetty eteenpäin ja muutettu valmiiksi tuotteeksi vaatimusten ja ominaisuuksien täsmennyttyä ja niihin liittyvien ongelmien ratkettua. Ohjelmiston prototyyppiä on siis pidetty jonkinlaisena mekaanisen prototyypin vastineena ohjelmistomaailmassa, kuten esimerkiksi Lantz prototyypin määrittelee [Lantz 1986].

Sen sijaan *nopealla prototyypityksellä* (rapid prototyping) [Song et al. 2005] tarkoitetaan pienillä resursseilla tehtävää *poisheitettävää* (throwaway) tuotetta, jota ei sellaisenaan käytetä valmiin ohjelmiston tekemiseen. Nopean prototyypityksen tuloksena on kokeiluversio, jolla tilaaja ja valmistaja voivat varmistaa, että molempien osapuolien käsitykset kehitettävän järjestelmän sisältämästä informaatiosta ja ilmaisuvoimasta ovat yhtenevät. Nopean prototyypityksen ideana on tuottaa nopeasti ja halvalla prototyyppi tähän tarkoitukseen. Tärkeää on, että prototyyppi on ajettava, jotta sen toiminnallisuuksia pystytään testaamaan [Chiang 2004]. Nopean prototyypityksen tuloksena syntyvän prototyypin keskeiset ominaisuudet ovat siis *kehittämisnopeus*, *edullisuus* ja *ajettavuus*.

2.2. Nopean prototyypityksen hyödyllisyys

Perinteinen prototyyppi on osoittanut vuosien mittaan hyödyllisyytensä monilla tuotantoaloilla. On ollut hyödyllistä kehittää suunnitelmaa järjestelmällisesti kehitysversioittain, kunnes on saatu valmis ja vaatimukset täyttävä testattu tuote. Alasta riippumatta tämä on antanut mahdollisuuden löytää mahdolliset ongelmakohdat kehitystyön varhaisessa vaiheessa. Ohjelmistopuolella on kuitenkin keskeiseksi tavoitteeksi tullut säästää sekä kustannuksissa että ohjelmistoprojektien kestossa. Ohjelmistokehitysprosessien tarkka vaatimusmäärittely on tullut entistä vaikeammaksi ohjelmistojen kehityttyä entistä laajemmiksi kokonaisuuksiksi. Uudet ohjelmistot joutuvat myös käsittelemään yhä suurempia tietomääriä. Näin ollen vaatimusmäärittely ja kattava käyttötapaustestaus ovat erittäin tärkeitä ohjelmistoprojektin vaiheita ja niiden toteuttamisen apuvälineeksi voidaan kehittää nopea prototyyppi. Sen tarkoitus on mahdollistaa vaatimusmäärittelyiden riittävyyden testaus nopeasti ja pienillä kustannuksilla.

Nopea prototyyppi eroaa siis perinteisestä prototyypistä jo hieman erilaisen päämäärän vuoksi. Lisäksi sen yhteydessä päätetään määrittelyvaiheessa myös, mitä nopean prototyypityksen kannalta epäoleellisia piirteitä jätetään prototyypissä testaamatta. Nopean prototyypin piirteitä karsittaessa on huomattava, että karsinnan kohteena ei ole itse lopullinen ohjelmisto – vaan itse nopea prototyyppi. Karsittavia piirteitä prototyyppivaiheessa ovat esimerkiksi tehokkuus ja käyttöliittymä. Koska nopean prototyypityksen tuloksena syntynyt ohjelmisto on asiakkaan käytettävissä, voi hän suorit-

taa toiminnallisuuteen liittyviä testauksia. Prototyyppiä kehitetään niin kauan, kunnes se tyydyttää asiakasta.

Prototyypin sisältämä toiminnallisuus avustaa lopullisen ohjelmiston moduulisuunnittelua. Prototyyppi sisältää ohjelmiston tarvittavan toiminnallisuuden ja täsmentyneen käsityksen käsiteltävästä informaatiosta, joihin perustuen on mahdollista laatia lopullisen ohjelmiston moduulikaavio. Kaaviosta selviää lopputuotetta kehitettäessä tarvittavat toiminnalliset kokonaisuudet ja niiden väliset yhteydet. Prototyyppi siis tukee lopullisen ohjelmistotuotteen suunnittelua ja toteutusta.

Prototyypin riittävistä ominaisuuksista voidaan varmistua soveltamalla sitä erilaisiin lopullisen ohjelmiston aitoihin käyttötilanteisiin. Tämä testaus saattaa johtaa itse prototyypin iterointiin. Käyttötilannetestaus on tehtävä huolellisesti ja sen on varmistettava, että testattavien käyttötapausten joukko on kattava. Toisin sanoen kaikki ne erilaiset tilanteet, joissa lopullista ohjelmistotuotetta tullaan käyttämään, pitää testata prototyypin yhteydessä.

Nopea prototyypitys on siis myös eräänlainen yhteinen oppimistapahtuma, jossa osapuolille muotoutuu yhteinen käsitys rakennettavasta järjestelmästä. Kun prototyyppi on kummankin osapuolen hyväksymä, voidaan varsinaisen ohjelmistotuotteen kuvaus perustaa siihen. Edellä mainitut käyttötilannetestaukset on myös syytä dokumentoida ja hyväksyttää tilaajalla, jotta myös testien riittävyys on molempien osapuolien hyväksymä. Ohjelmistoprojekti käyttää prototyyppiä jatkokehityksen pohjana ja siksi yhteinen varmennus on tärkeää myös mahdollisten tulevien riitatilanteiden käsittelyn kannalta. Mikäli tilaaja tahtoo jälkikäteen lisätä ominaisuuksia järjestelmään, on olemassa hyväksytty prototyyppi, josta voidaan tarkistaa, sisältyvätkö ko. piirteet järjestelmään.

Pienillä resursseilla tuotettu nopea prototyyppi antaa mahdollisuuden hylätä prototyyppi täysin ryhdyttäessä varsinaisen ohjelmiston valmistamiseen. Nopean prototyypin pääpaino on ohjelmiston suunnittelussa, joten se on tärkeä osa ohjelmistoprojektia, vaikkei mitään sen komponentteja sellaisinaan käytettäisikään lopullisessa ohjelmistossa. Niin sanotusti poisheitettykin prototyyppi on palvellut ohjelmistoprojektia käyttötapatestauksessa ja vaatimusmäärittelyiden muokkaamisessa. Prototyyppi on voinut

tuoda esiin myös toteutuksellisia ongelmia, jotka voivat ohjata varsinaisen ohjelmiston toteutustapojen valintaa. Täysin poisheitetyksi päätyneen prototyypin anti on onnistuneen ja täsmällisen vaatimusmäärittelyn lisäksi siinä, että vääriä valintoja, huonoja ratkaisuja ja riittämättömiä ominaisuuksia on havaittu projektin alkuvaiheessa. Ilman prototyyppiä voisivat ongelmat paljastua mahdollisesti vasta varsinaisen ohjelmiston kehityksen ollessa pitkällä.

2.3. Nopean prototyypityksen menetelmiä

Myös menetelmiltään nopea prototyypitys eroaa perinteisestä prototyypityksestä. Perinteisen prototyypin lähestymistapa ja ohjelmointimenetelmät ovat samat kuin varsinaisen ohjelmiston yhteydessä. Tämä tarkoittaa siis sitä, että jo prototyypin määrittelyvaiheessa on päätetty käytettävät ohjelmointimenetelmät, rajapinnat ja käyttöliittymäratkaisut. Nämä saattavat aiheuttaa suuria ongelmia, jos projektiin joudutaan tekemään vaatimusmuutoksia. Nopeassa prototyypityksessä sen sijaan rakennetaan prototyyppi valmiiksi kiinnittämättä huomiota esimerkiksi tehokkuuteen tai käyttöliittymään. Nopean prototyypin pääpaino on toiminnallisuudessa ja sen testaamisessa. Tilaajan hyväksynnän jälkeen ohjelmistokehittäjä voi vapaasti tehdä tarkat suunnitelmat ja valita parhaat lähestymistavat varsinaisen ohjelmiston toteuttamiseen. Esimerkiksi projektin toteutusaikataulu voidaan laatia prototyypin toiminnallisuuden perusteella. Varsinaisen ohjelmiston toteutustavat voivat siis olla täysin eri kuin prototyypissä käytetyt – aina toteutuskieltä ja tietokantaratkaisuja myöten.

Ohjelmistojen prototyypityksen suorittamiseen on kehitetty monenlaisia järjestelmiä ja menetelmiä. Tällaisia työkaluja ovat esimerkiksi erilaiset *CASE-ohjelmistot* (Computer Aided Software Engineering). Nopean prototyypityksen kannalta on valitettavaa, että suurin osa erilaisista valmiista työkaluista keskittynyt käyttöliittymän suunnitteluun. Ohjelmistoa simuloivien valmiiden ohjelmistojen lisäksi usein nopeaan (ja myös perinteiseen) prototyypitykseen on käytetty nk. *neljännen sukupolven ohjelmointikieliä* (4th generation languages). Pikaprototyypitystä voi tehdä millä tahansa ohjelmointikielellä (esim. VisualBasic [VB] ja Prolog [Chiang 2004] ovat suosittuja tällaisia). Varsinaisia pikaprototyyppiohjelmistoja on kuitenkin olemassa. Esimerkiksi QuintPro [Feudenthal et al. 2004], tietokantaohjelmiston toteuttamiseen suunniteltu RadVolution [RadVolution] sekä CASE-sovellustyyppinen MicroCreator [MicroCreator] ovat

tällaisia ohjelmistoja. Nopean prototyypityksen suosion vuoksi on oletettavissa, että tällaisia erityisesti nopean prototyypityksen suorittamiseen tarkoitettuja ohjelmistoja luodaan lisää.

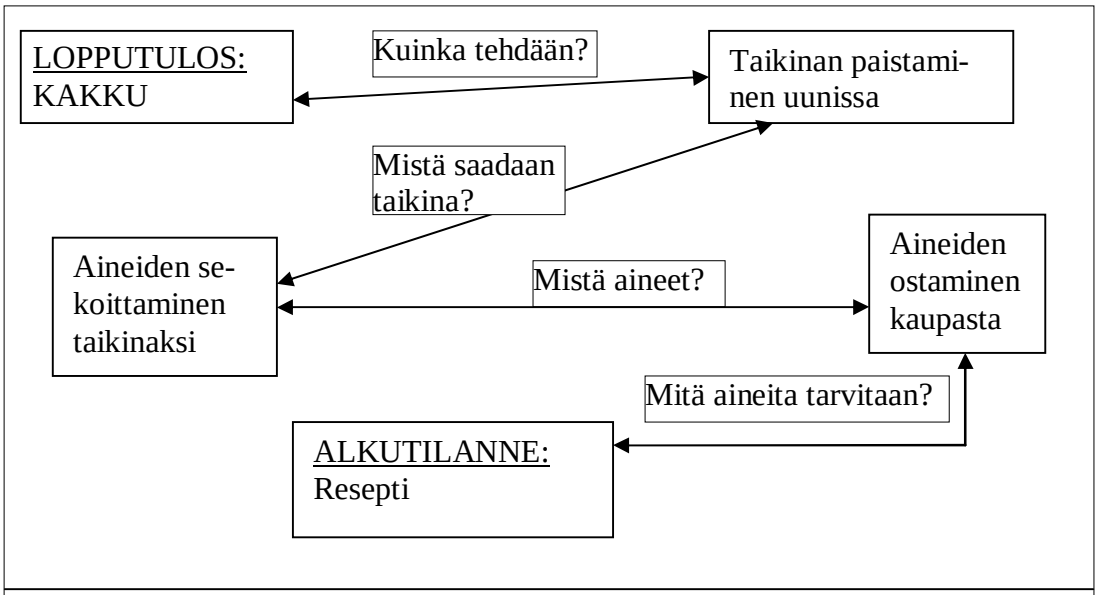
Prototyypitystä voidaan suorittaa jopa paperiprototyypillä (esim. [Bailey et al. 2008]), jossa piirretyt näytönkuvat vastaisivat valmiin ohjelman toimintoja. Kuitenkin on huomattava, että paperiprototyyppi ei täytä pikaprototyypin ominaisuusvaatimuksia, koska se ei anna käyttäjälle selkeää kuvaa järjestelmän tarjoamasta vuorovaikutuksesta, eikä paperiprototyypillä ole mahdollista selkeästi havainnollistaa prosessoinnin monimutkaisia lopputuloksia. Paperiprototyyppi ei ole myöskään ajettavissa ja juuri ajettavuus on keskeistä nopealle prototyypitykselle.

Chiang on lähestynyt nopeaa prototyypitystä artikkelissaan, jossa hän käsittelee TUG-spesifikaatioiden (Tree Unification Grammar) nopeaa automaattista prototyypitystä. TUG- spesifikaatiot muodostetaan Chiangin kehittämillä Prolog- metodeilla automaattisesti tiettyjen sääntöjen mukaan. Chiangin mukaan tällaisella automaattisella prototyypityksellä saavutetaan taloudellisesti parempi tulos kuin käsin tehdyn prototyypin kanssa. [Chiang 2004]

Kuten em. Chiangin tutkimuksessa, myös tässä tutkielmassa toteutusvälineenä on Prolog, mutta menetelmänä käytetään konstruktoripohjaista Prolog-mallinnusta [Niemi & Järvelin -88, -91 & -97]. Prolog [Sterling 1994] on siis itsessäänkin yksi suosituista prototyypitykseen käytetyistä ohjelmointirajapinnoista. Prologin lähestymistapa tukee deklaratiivista tulkintaa. Deklaratiivisuus tarkoittaa sitä, että keskitytään mieluummin spesifioimaan mitä haluttu lopputulos sisältää, kuin miten lopputulos tuotetaan. Jälkimmäinen piirre on puolestaan tyypillistä proseduraalisille ohjelmointikielille. Prolog avustaa nopeata prototyypitystä, kun tulkki itse hoitaa lopputuloksen prosessoinnin.

Täydellistä deklaratiivisuutta ei ole pystytty toteuttamaan Prologissakaan, koska sen käyttäjän on otettava huomioon Prologin prosessointimekanismeja. Prologin soveltamisessa tarvitaan usein top-down ajattelua. Esimerkiksi kakun leipominen tällä top-down periaatteella tapahtuisi Kuvan 1 osoittamalla tavalla. Kuva 1 havainnollistaa myös sitä, miten lähellä arkipäiväistä ajatteluamme deklaratiivinen ajattelutapa on. Useimmiten ongelmatilanne on sellainen, että tiedetään haluttu päämäärä ja pohditaan,

mitä tarvitaan sen saavuttamiseksi. Varsin harvoin siis lähdetään liikkeelle aineksista ja mietitään miten niitä sekoittamalla saadaan aikaiseksi kakku.



Kuva 1. Kaavio leipomisesta.

Ohjelmistokehityksen avuksi on kehitetty erityisiä ohjelmistoja ns. sovelluskehittimiä, jotka auttavat rakennettavan ohjelmiston ominaisuuksien testaamisessa. Tällaiset sovelluskehittimet ovat erityisen hyvä keino testata esimerkiksi rakennettavan järjestelmän käyttöliittymän toimivuutta. On kuitenkin huomattava, että jokaisen kehittimen käyttö vaatii oman asiantuntemuksensa. Siksi onkin mielekästä arvioida kehittimen opiskelun hyötyä suhteessa varsinaisen ohjelmiston toteutukseen käyttökelpoisen ohjelmointiympäristön käyttöön. Vaikka pikaprototyypin kohdalla yhtenä lähtökohtana pidetäänkin poisheitettävyyttä, lisääntyy prototyypin käyttökelpoisuus, mikäli joitakin sen komponentteja voi käyttää myös osana varsinaista ohjelmistoa. Jo prototyyppivaiheessa saattaa olla mielekästä ottaa huomioon, että monet ohjelmointiympäristöt tarjoavat nykyään mahdollisuuksia upottaa koodiin toisella ohjelmointikielellä koodattuja osia. Tällaisia ohjelmistoja kutsutaan hybridiohjelmistoiksi. Prolog-koodiin on esimerkiksi voitu integroida SQL-koodia mahdollistamaan relaatiotietokantojen käsittely niin, että Prologia sovelletaan johdetun tiedon määrittelyyn, kun taas SQL:lla poimitaan relaatiotietokannasta tieto, johon johtaminen kohdistuu (esimerkiksi [NED]).

3. Asiantuntijajärjestelmät

3.1. Asiantuntija ja asiantuntijuus

Jotta pystytään tarkastelemaan asiantuntijajärjestelmiä ja niiden ominaisuuksia, on aluksi selvitettävä, mitä asiantuntijuudella tarkoitetaan. *Asiantuntija* (expert) ja *asiantuntijuus* (expertise) ovat sanoja, joiden intuitiivinen merkitys on varsin selvä. Esimerkiksi Encyclopedia Britannican verkkoversio kertoo asiantuntijan olevan henkilö, jolla on erityistaito tai jonkin aihealueen erityinen tietämys [Encyclopedia]. Collinsin englantilaisen tietosanakirjan mukaan asiantuntijuus tarkoittaa erityistaitoja tai -tietoja, jotka on saavutettu harjoittelun, opiskelun tai käytännön toimien kautta. Asiantuntija on puolestaan Collinsin määritelmän mukaan henkilö, jolla on asiantuntijuuden määrittelemiä ominaisuuksia [Collins].

Asiantuntijuuden määritelmään sisältyy siis tiedon lisäksi myös tiedon soveltamisen taito. Internetpohjaisiin asiantuntijajärjestelmätoteutuksiin erikoistuneella Expertise2Go-sivustolla [Expertise2Go] määritellään asiantuntija henkilöksi, jolla on erikoistunutta taitoa, tietoa ja kokemusta, jota kaikilla ei ole, sekä kykyä soveltaa näitä erilaisten ongelmien ratkaisemiseen [Harmon & King 1985]. Jotta asiantuntijan neuvoja pidetään hyödyllisinä, on niiden useimmiten oltava niin hyviä, että asiantuntija säilyttää maineensa. Asiantuntijan ei siis oleteta antavan täydellisiä neuvoja, vaan neuvojen on vain oltava tarpeeksi hyviä, jotta asiantuntija säilyttää asiantuntijan maineensa [Hayes-Roth et al. 1983]. Näin asiantuntijuudella tarkoitetaan siis jonkin tietyn osaamisalueen tai asiakokonaisuuden sekä erityisen laajaa että samalla myös erittäin syvällistä hallitsemista. Asiantuntija puolestaan on henkilö, jonka tiedot ja taidot kattavat jonkin alan asiantuntijuuden vaatimukset. Yleensä asiantuntijalta pyydetään neuvoja tai mielipiteitä silloin kun muutoin ei hallita kyseistä alaa tarpeeksi kattavasti turvallisten ja kelvollisten ratkaisujen tekemiseksi.

3.2. Tekoäly

Tekoälyn (artificial intelligence eli AI esim. [Winston 1984] tai [Arnold & Bowie 1986]) tarkoituksena on tuottaa vaikutelma päättelystä, ajattelusta tai analysoinnista

ohjelmiston käyttäjälle. Tässä tutkielmassa tekoälyä pidetään jonkinlaisena keinotekoisena ajatteluna, vaikka ajattelu myös omissa aivoissamme saattaa perustua hyvin samankaltaisiin päättelyketjuihin kuin tekoälysovelluksissa. Itse asiassa Winston toteaa tekoälyn kehittämisen auttavan ihmisiä ymmärtämään älykkyyttä ylipäätänsä [Winston 1984].

Nykyään tekoälyä sovelletaan paljon erilaisissa tietokonepeleissä. Peliteollisuus on viime vuosina voimakkaasti kasvanut ala ja onkin nykyään taloudelliselta merkitykseltään haastanut vakavasti elokuvateollisuuden. Jokaisen pelin sisältäessä jonkinlaista tekoälyä on ilmeistä, että tekoäly on myös varsin suosittu ja tutkittu tietojenkäsittelyopin osa-alue. Tekoälyn tutkimuksessa ja kehityksessä esiintyy kuitenkin myös paljon tietotekniikkaoptimismia. Tietotekniikan kehityksen alkuaikoina kuviteltiin tietotekniikan ratkaisevan kaikki ongelmat – nyt tämä sama toive on suunnattu tekoälyyn. Kyseessä on lumivyöryn kaltainen kierre, jossa positiiviset tutkimustulokset ja kehitys synnyttävät haaveita ja toiveita sekä antavat sovellutusmahdollisuuksia viihdeteollisuuden käyttöön. Viihdeteollisuuden huimat visiot johtavat lisähaaveiluun ja jopa vääristyneisiin todellisuuskuviin ja ylilyöviin arvioihin järjestelmien mahdollisuuksista. Toisaalta nämä fiktiiviset visiot saattavat puolestaan antaa tutkijoille ja kehittelijöille lisäideoita omaan työhönsä ja kenties näin syntyy taas uusia tutkimustuloksia, jotka puolestaan aloittavat uuden soveltamiskierteen. Siksi on varsin vaikea ennustaa tekoälyohjelmistojen kehityskulkua. Tekoäly kuitenkin mahdollistaa monenlaista vuorovaikutteista toimintaa sovellutuksen ja käyttäjän välillä olematta kuitenkaan syvällisesti ajateltuna älykästä.

Asiantuntijajärjestelmät ovat ehkä tekoälyn haaroista kaikista käytännönläheisimpiä tuoden siis tekoälysovellutukset arkeen auttamaan käyttäjiä arkipäiväisessä ongelmanratkaisussa. Tekoälymenetelmillä älykkyyttä upotetaan asiantuntijajärjestelmään ja näin tuotetaan käyttäjälle informaatiota hänen päätöksenteko-ongelmansa ratkaisun tueksi..

3.3. Asiantuntijajärjestelmien piirteitä

Asiantuntijajärjestelmä tarjoaa käyttäjälleen asiantuntijan neuvoja tukeutuen järjestelmään sisällytettyyn asiantuntijuuteen. Asiantuntijajärjestelmän asiantuntijuus pe-

rustuu siis järjestelmään koodatun tiedon automaattiseen analysointiin. Asiantuntijajärjestelmät ovat tietoyhteiskunnassa jatkuvasti kasvavassa roolissa. Asiantuntijajärjestelmiin liittyy paljon tietotekniikkaoptimismia, mutta siitä huolimatta ja juuri sen vuoksi asiantuntijajärjestelmien kirjo on jatkuvasti kasvanut. Yhä useammalla eri alalla otetaan käyttöön erilaisia asiantuntijajärjestelmiä, jotka mahdollistavat alan ammattilaisille vertaisasiantuntijuutta ja antavat mahdollisuuden käyttää tietotekniikan lähes rajattomia prosessointi ja muistiominaisuuksia päätöksiensä teon tukena.

Asiantuntijajärjestelmissä nähdään tällä hetkellä potentiaalia erilaisten kulujen vähentämiseen ja tästä johtuen niiden kehittämiseen on panostettu. On esimerkiksi ajateltu, että jonkinlainen terveydenhuollon asiantuntijajärjestelmä voisi avustaa kunnallisen terveydenhuollon piirissä olevia siten, että sen avulla voitaisiin välttää joitakin tarpeettomia terveyskeskuskäyntejä. Tämä puolestaan pienentäisi omalta osaltaan jonoja terveyskeskuksissa ja vähentäisi terveydenhuoltoon kuluvia varoja. Terveydenhuollon asiantuntijajärjestelmien (esim. Internet-perusteinen diagnosointijärjestelmä CMDS (*Chinese Medical Diagnostic System*) [Huang & Chen 2007]) lisäksi asiantuntijajärjestelmiä on käytössä myös esimerkiksi kaivostoiminnassa (esim. mineraalien tunnistusohjelmisto GOLD [Lister et al. 1988]), maataloudessa (esim. kasvien tuholaisten tunnistusjärjestelmä DIARES-IPM [Mahaman et al. 2003] tai lampaiden jalostusarvon ennustamisohjelmisto [Ossa et al. 2007]), maankäytössä (esim. jätteenkäsittelyn, kauppakeskusten, jätevesijärjestelmien sijoittelu [Witlox 2005]) ja monessa muussa eri sovellutuksessa. Asiantuntijajärjestelmiä kehitetään jatkuvasti uusille aloille – kuten esimerkiksi juridiikkaan (esim. SHYSTER-MYCIN [O'Callaghan et al. 2003] tai HYPO [Rissland & Ashley 1987]).

Asiantuntijajärjestelmien joukko on melko epämääräinen, koska jonkinlaisesta asiantuntijajärjestelmästä on aina kyse, jos ohjelmisto antaa neuvoja tai suosituksia johonkin tietyn aihealueen päätöksenteko-ongelmaan liittyen. Asiantuntijajärjestelmät toimivat ikään kuin interaktiivisena tietokirjasarjana. Käyttäjän ei tarvitse etsiä tietoa itse, vaan järjestelmä pyrkii tarjoamaan mahdollisimman lähelle osuvaa oikeantyyppistä tietoa (suosituksia) perustuen käyttäjän antamaan palautteeseen. Esimerkiksi käyttäjä voisi tuntemattoman eläimen nähtyään käydä eläinlajintunnistusohjelmiston kanssa kuvassa 2 esitetyn keskustelun pyrkiessään tunnistamaan eläimen (ongelmanratkaisutilanne).

- Oliko eläimellä turkki?
- Ei.
- Oliko eläin havaittaessa suomalaisessa järvessä?
- Kyllä.
- Onko eläimellä eviä?
- Kyllä.
- Kyseessä on siis kala.
- Onko kalalla suomuja?
- Kyllä.
- Onko kala hopeanvärinen?
- Kyllä.
- Ovatko kalan silmät punaiset?
- Kyllä.
- Selvä. Eläin on siis kala ja nimeltään särki.

Kuva 2. Keskustelu, jonka käyttäjä (alleviivatut vastaukset) saattaisi käydä eläinlajintunnistusohjelmiston kanssa.

Esimerkistä on huomattavissa, miten järjestelmä kyselemällä karsii *tietämyskannassaan* olevista eläimistä pois kriteereihin sopimattomia vaihtoehtoja. Karsinta tietämyskannassa onkin asiantuntijajärjestelmän keskeinen tehtävä. Tietämyskanta sisältää kaiken sen informaation, johon perustuen järjestelmä tekee päätelmiään ja antaa ratkaisuehdotuksiaan. Tietämyskanta koostuu siis käytännössä sekä erilaisista järjestelmään talletetuista tiedoista että niiden käsittelyyn liittyvistä säännöistä. Tietämyskannan (tai tietämyksen yleensä) keskeisen aseman vuoksi asiantuntijajärjestelmiä kutsutaan usein *tietämysperusteisiksi järjestelmiksi* (knowledge-based systems esim. [Sagheb-Tehrani 1993]). Tällä tarkoitetaan siis ohjelmiston kykyä käyttää sisältämäänsä tietämystä asiantuntijuuden vaatimusten mukaiseen toimintaan

Asiantuntijajärjestelmän toimintatapa vaihtelee riippuen siitä, mihin sitä käytetään. Eräs lähestymistapa on, että järjestelmä toimii edellä esitetyllä tavalla jatkuvasti käyttäjän kanssa kommunikoiden. Toisaalta lähtökohta voi olla myös sellainen, että käyttäjä valitsee annetusta joukosta omaan tarpeeseensa sopivia ominaisuuksia, minkä jälkeen järjestelmä prosessoi relevantit tiedot ja antaa ratkaisuehdotuksen. Usein tarkoituksenmukainen lähestymistapa riippuu siitä, tarvitaanko ohjelmistoa määrätyn ennalta tiedetyn yksittäisongelman ratkaisemiseen vai johonkin ongelmatilanteeseen liittyvien tietojen kartoittamiseen. Esimerkiksi em. diagnostiset ohjelmistot edustavat ensin mainittua lähestymistapaa, kun taas esimerkiksi Tampereen kaupungin liikennelaitok-

sen reitti- ja aikatauluopas Repa Reittiopas [Repa] edustaa jälkimmäistä lähestymistapaa.

3.4. Asiantuntijajärjestelmien luokittelu

Erilaisia asiantuntijajärjestelmiä on vaikea luokitella tarkasti ja kattavasti niissä esiintyvien asiantuntijakomponenttien moninaisuuden tähden. Esimerkiksi asiantuntijuus voi olla joko piilotettuna ominaisuutena (tekoälynä) pelissä tai asiantuntijuus on koko ohjelmiston itsetarkoitus. Tämän tutkielman näkökulmasta kiinnostavampi ja keskeisempi alue ovat itsetarkoitukselliset asiantuntijajärjestelmät. Siksi seuraavassa pyritään jaottelemaan tällaisia asiantuntijajärjestelmiä toteutuksen ja toiminnallisuuden perusteella. Kuitenkin on merkille pantavaa, että ohjelmistoa voi olla vaikea sijoittaa yksikäsitteisesti johonkin tiettyyn kategoriaan, koska useat ohjelmistot sijainnevat usean eri kategorian välimaastossa sisältäen ominaisuuksia monesta kategoriasta.

Asiantuntijajärjestelmiä luokiteltaessa voidaan tarkastella toteutuksellista rakennetta. Silloin huomio kiinnittyy erityisesti tiedon mallinnustapaan ts. tapaan tallentaa tieto järjestelmään. Friedland jakaa asiantuntijajärjestelmät kolmeen eri luokkaan tietämyksen tallennustavan mukaan ja nämä tavat hänen esityksessään ovat *sääntöperusteisuus* (rule-based), *kehysperusteisuus* (frame-based) ja *logiikka* (logic) [Friedland 1985]. Friedlandin luokittelun lisäksi on olemassa muitakin luokitteluja, sillä nykyään asiantuntijajärjestelmien toteutukseen on useammanlaisia mahdollisuuksia kuin Friedlandilla. Esimerkiksi Liao [Liao 2004] jakaa tarkastelussaan asiantuntijajärjestelmät yhteentoista eri kategoriaan, joista uusina luokkina esimerkiksi *neuroverkot* (neural networks), *oliomalli* (object-oriented methodology), *älykkäät agentit* (intelligent agent), *sumean logiikan asiantuntijajärjestelmät* (fuzzy logic expert systems) ja *tapauspohjainen päättely* (case-based reasoning). Tässä tutkielmassa luokittelutavaksi valitaan kuitenkin Friedlandin kolmeluokkainen jaottelu. Friedlandin luokittelun metodologioiden lisäksi luvun lopussa pohditaan vielä asiantuntijajärjestelmien eroja käyttäjälle näkyvässä toiminnallisuudessa.

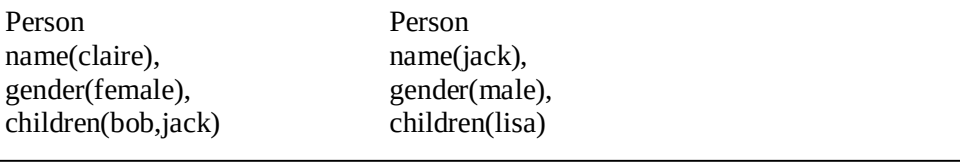
Sääntöperusteinen (esim. [Hayes-Roth 1985]) toteutus perustuu logiikka-pohjaisten sääntöjen laatimiseen. Yksinkertainen jos/sitten tulkinta liittyy tällaiseen sääntöperusteisuuteen. Kaikki tieto siis sisällytetään sääntöihin. Ratkaisun löytäminen suoritetaan

näitä sääntöjä prosessoimalla. Sääntöperusteinen päättelyketju voidaan toteuttaa joko *eteenpäin* tai *taaksepäin ketjuttaen* (forward-chaining ja backward-chaining). Erona näissä menetelmissä on se, että eteenpäin ketjutuksessa lähdetään liikkeelle tunnetuista tosiasioista, kun taas taaksepäin ketjutuksessa jonkin oletuksen oikeaksi todistaminen aloitetaan säännöistä. AI-Depotin sääntöperusteisten järjestelmien oppaan [AI-Depot] liitteessä James Freeman-Hargis antaa valaisevan esimerkin taudin diagnosoinnista eteenpäin- ja taaksepäinketjutuksilla (ks. Liite 1). Eteenpäin ketjutuksessa lähdetään liikkeelle siitä, että on vuotava nenä, kun taas taaksepäin ketjutuksessa päättely aloitetaan olettamuksesta, että potilaalla on flunssa.

Kehysperusteisesti (esim. [Fikes & Kehler 1985]) toteutettuna tieto on esitetty kehyksissä, jotka muistuttavat kortistotietuetta tai olio-ohjelmoinnin oliota. Kehys siis sisältää aina yhden olion tiedot. Päättelyitä suoritetaan kehysten tietoja vertailemalla ja sovittamalla niitä saatavilla oleviin sääntöihin. Tutkielmassa kehitetty RoutePlanner-ohjelmisto käyttää eräänlaista kehysrakennetta, koska toisaalta järjestelmän sisältämä tietämys on sisällytetty toisaalta reittikuvauksiin ja toisaalta niiden käsittelemiseksi toteutettuihin sääntöihin. Reittikuvaus sisältää aina yhteen reittiin liittyvät tiedot, mikä antaa mahdollisuuden mieltää kuvaus rakenteellisesti kehykseksi. Kehysperusteisessa mallissa monimutkainenkin tieto on haettavissa yhdistelemällä kehyksestä löytyviä tietoja.

Kehykset voidaan kuvata esimerkiksi Prolog-faktoilla, kuten kuvassa 3 on esitetty. Kuvan 3 a-kohdan Person-kehysten käsittelyyn liittyvät säännöt eivät ole näkyvissä [Kuipers 1994], mutta vertailun vuoksi Prologilla toteutetun isoäitiesityksen (kuva 3.b) säännöt annetaan eksplisiittisesti. Isoäititoteutuksen kehykset on toteutettu käyttäen järjestettyä jonoa, jonka symbolina on t ja parametreina ovat kehyksen tiedot. Järjestetyn jonon käsitteeseen palataan myöhemmin käsiteltäessä konstruktoreita (ks. luku 4.2).

(a)



(b)

```
t(type(person),name(jutta),gender(female),children([markus, eeva, anna])).
t(type(person),name(markus),gender(male),children([mikael, elisa])).
t(type(person),name(eeva),gender(female),children([aapeli])).
t(type(person),name(anna),gender(female),children([])).

is_grandmother(Name):-
        t(type(person),name(Name),gender(female),children(Children)),
        has_children(Name),have_children(Children).
has_children(Name):-
        t(type(person),name(Name),gender(_),children(Children)),
        length(Children,L),L>0.
have_children([Child|_]):- has_children(Child),!.
have_children([Child|Rest]):- have_children(Rest).
```

Kuva 3. Isoäitiyden toteutus. Kehysperusteisesti (a) mukaillen Kuipersin algernon- opasta [Kuipers 1994]. Prologilla ohjelmoitu isoäititoteutus (b), jossa henkilötiedot on esitetty järjestettynä jonona.

Kolmantena luokkana Friedlandin asiantuntijajärjestelmien luokituksessa on siis logiikka [Friedland 1985]. Loogisten termien ja päättelyketjujen avulla voidaan tietämys siis mallintaa järjestelmään [Genesereth & Ginsberg 1985]. Todetaan kuitenkin, että tämän pro gradu -tutkielman yhtenä mielenkiintokohteena on Prolog-ohjelmointi. Prolog puolestaan on ohjelmointikielenä yksi käytetyimmistä tekoälyohjelmointikielistä. Ohitetaan kuitenkin logiikkaohjelmoinnin käsittely tässä, koska sitä sivutaan tutkielmassa useaan otteeseen käsiteltäessä Prolog-toteutuksia.

Edellä esitellyn luokittelun lisäksi asiantuntijajärjestelmissä on eroja myös käyttäjälle näkyvässä toiminnallisuudessa. Em. toiminnallisuus koostuu mahdollisesta lähtötietojen antamisesta, ongelman määrittelystä ja ratkaisuehdotuksen tarjoamisesta. Seuraavassa vertaillaan siis näiden toimintojen toteutusmenetelmiä.

Asiantuntijajärjestelmässä lähtötiedot voidaan saada kolmella tavalla:

1) Kaikki tarvittavat tiedot on talletettu järjestelmän tietämyskantaan.
2) Osa tiedoista saadaan tietämyskannasta ja osa käyttäjältä ajon aikana.
3) Kaikki tiedot saadaan käyttäjältä.

Kohdan 3 tapaus on varsin harvinainen ja periaatteellisesti jopa mahdoton, sillä mikäli asiantuntijajärjestelmän sääntökokoelmat katsotaan osaksi tietämyskantaa, niin kuin yleensä tehdään, ei pelkillä käyttäjän tiedoilla ole mahdollista päättelyä suorittaa. Yleisin vaihtoehto on kohdan 2 toimintatapa, jossa järjestelmän tietämyskanta sisältää perussäännöstön ja perustietokannan. Tässä vaihtoehdossa käyttäjä antaa osan ongelmanratkaisussa tarvittavista tiedoista, jotka liittyvät tapauskohtaiseen ongelmatilanteeseen.

Asiantuntijajärjestelmä on usein ohjelmoitu käsittelemään vain tiettyä yksittäistä ongelmaa. Esimerkiksi auton käynnistymisongelmaa pohtiva Auto Diagnosis Advisor [Expertise2Go] on ohjelmoitu ratkaisemaan ainoastaan auton käynnistymisongelmaa. Repa Reittiopas puolestaan yrittää vain löytää parhaan yhteyden paikasta toiseen käyttäen hyväksi ainoastaan kaupungin liikennelaitoksen bussiyhteyksiä ja jalankulkua [Repa]. Suuremman ongelmaratkaisualueen hallitsevassa asiantuntijajärjestelmässä on otettava huomioon, että ongelmanratkaisualueen sisältä löytyy erilaisia yksittäisongelmia. Esimerkiksi terveydentilan diagnosointiin tarkoitetun asiantuntijaohjelmiston pitäisi hallita erilaisten sairauksien tunnistamiseen liittyviä tekijöitä. Mikäli asiantuntijaohjelmisto hallitsee laajaa ongelmanratkaisualuetta, on käsiteltävän ongelman tarkka määrittely tärkeää, jotta järjestelmä löytää juuri oikean ratkaisuehdotuksen.

Asiantuntijajärjestelmissä ongelman tarkka ja tapauskohtainen määrittely piilotetaan usein alkutietojen kyselyyn. Esimerkiksi tutkielmassa konstruoidun esimerkkiohjelmiston RoutePlannerin ainoa ongelma on selvittää matkustusreittejä. Kuitenkin ongelman yksityiskohtainen määrittely tapahtuu alkutietojen kyselyn yhteydessä, kun käyttäjältä pyydetään erilaisia matkustusreittiin kuuluvia kriteereitä. Diagnostiikan lisäksi on olemassa lääketieteellisiä asiantuntijajärjestelmiä, jotka antavat suosituksia esimerkiksi lääkehoidosta tai leikkausajoista. Diagnosointiin keskittyvässä asiantuntijajärjestelmässä perusongelmana on kuitenkin selvittää laajemmin asiakkaan terveysongelma. Alkutietojen saamisen jälkeen ja tietämyskannan päätelmien perusteella ratkaistava ongelma tarkentuu esimerkiksi muotoon ”Mistä asiakkaan jalkasärky johtuu?”.

Päättely asiantuntijajärjestelmässä tapahtuu joko avustetusti tai itsenäisesti. Asiantuntijajärjestelmä saattaa lähtötietojen antamisen jälkeen tehdä ratkaisuehdotuksen täysin

itsenäisesti, kuten esimerkiksi RoutePlanner-ohjelmiston tapauksessa tapahtuu. Järjestelmä voi vaihtoehtoisesti myös olla vuorovaikutuksessa käyttäjän kanssa ja lisäkyselyiden avulla tarkentaa päättelyketjuaan rajoittamalla mahdollisia päättelymahdollisuuksia. Oletettavasti esimerkiksi juuri edellä mainittu lääketieteellinen diagnosointiohjelmisto toteuttaa ongelmanratkaisua kyselemällä käyttäjältä oireita ja ohjaten päättelyketjua ja jatkokyselyjä käyttäjän antaman palautteen pohjalta. Myös kuvassa 2 esitetty eläinlajin määrittely vaatii käyttäjältä tietojen antamista, kunnes lopullinen ratkaisuehdotus on löytynyt.

Asiantuntijajärjestelmä pyrkii tarjoamaan käyttäjälleen aina jonkin ratkaisuehdotuksen sille esitettyyn ongelmaan. Asiantuntijajärjestelmä päätöksenteon tukijärjestelmänä eli päättelytukena avustaa käyttäjäänsä päätöksenteossa. Kuitenkin myös ratkaisuvaihtoehtojen tarjoaminen vaihtelee järjestelmittäin. Järjestelmä saattaa antaa käyttäjälleen joko yhden ratkaisun tai tarjota erilaisia ratkaisuvaihtoehtoja. On myös tapauksia, jolloin järjestelmä ei löydä ratkaisua ja tällaisesta tilanteesta selviäminen on hoidettu erilailla eri järjestelmissä. Periaatteellisia vaihtoehtoja ovat, joko kerätä lisäinformaatiota uusilla kyselyillä tai antaa lähimmäksi oikeaan osunut vaihtoehto tai jättää ainoaksi vaihtoehdoksi uuden ajon suorittaminen. Esimerkiksi tutkielman RoutePlanner tarjoaa käyttäjälleen vain löydetyt reittivaihtoehdot tai tiedon siitä, ettei kyseisillä kriteereillä ole löydettävissä yhtään soveltuvaa matkustusreitistöä.

Pätevistäkin ratkaisuehdotuksista huolimatta asiantuntijaohjelmisto toimii varsin harvoin varsinaisena asiantuntijana tai lopullisen päätöksen tekijänä. Esimerkiksi aiemmin mainitut lääketieteelliset ohjelmistot toimivat hyvin asiakkaan tai hoitavan lääkärin päätöksenteon tukena, mutta varsinaisen lopullisen hoitopäätöksen tekee kaikki olosuhteet ja oman kokemuksensa huomioonottava hoitohenkilö.

4. Konstruktori-orientoitunut lähestymistapa tietojen esittämiseen

4.1. Konstruktori-orientoituneisuuden lähtökohta

Konstruktori-orientoitunut lähestymistapa on keino esittää tietojen välinen rakenteellisuus järjestelmällisesti. Varsinkin olio-ohjelmointiympäristöissä tiedot mallinnetaan joukolla järjestelmän mahdollistamia rakentimia l. konstruktoreita. Prologin kaltaisessa logiikkaohjelmointiympäristössä on lopulta vain yksi tapa esittää tietoja. Jokainen tieto on esitettävä jonakin loogisena terminä. Logiikkaohjelmoinnissa loogisen termin eri vaihtoehdot ovat:

a) vakio,
b) muuttuja tai
c) yhdistetty termi.

Koska logiikkaohjelmoinnissa kaikki informaation esittäminen ja käsittely tapahtuu loogisten termien avulla, on ollut tarpeellista kehittää systemaattinen tapa rakenteellisen tiedon kuvaamiseksi tässä ympäristössä. Konstruktori-orientoituneessa lähestymistavassa konstruktoreja käytetään tiedon rakenteistamiseen. Lisäksi konstruktori-orientoitunut lähestymistapa sisältää joukon operaatioita, joita voidaan hyödyntää konstruktoreilla mallinnetun tiedon käsittelyssä.

Tiedon esittämisen kannalta on ensisijaisesti pohdittava periaatteita, jotka liittyvät tiedon rakenteistamiseen. Rakenteellisessa tiedon esityksessä tieto koostetaan pienemmistä osasista, joiden keskinäiset suhteet suuremmassa tietoyksikössä organisoidaan kehitettyjä periaatteita noudattaen. Rakenteellisuuden kannalta on tärkeätä tietää, ovatko tiedon välittömät komponentit rakenteellisesti keskenään homogeenisia vai heterogeenisiä, eli koostuuko rakenne erilaisista vai samanlaisista komponenteista. Toinen keskeinen periaate rakenteellisuuden mallintamisessa on rakenteistettavan tiedon välittömien komponenttien keskinäisen järjestyksen merkitys. Toisin sanoen rakennetta muodostettaessa on kiinnitettävä huomiota rakenteen sisältämien komponenttien järjestyksen merkityksellisyyteen ja komponenttien keskinäiseen rakenteelliseen samankaltaisuuteen..

4.2. Rakenteistaminen konstruktoreilla

Niemi ja Järvelin [Niemi & Järvelin -88, -91 & -97] ovat Prolog-ympäristössä esitelleet rakenteellisen tiedon esittämiseen joukon konstruktoreita. Heidän esittelemiään konstruktoreita on kaikkiaan neljä. Näihin konstruktoreihin liittyvät erilaiset soveltamisperiaatteet. *Järjestettyä jonoa* (tuple) käytetään tiedon esittämiseen silloin, kun mallinnettava tieto koostuu rakenteellisesti homogeenisista osista, joiden keskinäinen järjestys on olennainen. *Joukko* (set) puolestaan koostuu rakenteellisesti homogeenisista tietoyksiköistä, joiden keskinäisellä järjestyksellä ei ole merkitystä. Mikäli rakenteen sisältämät komponentit ovat keskenään rakenteellisesti heterogeenisia, silloin rakenne esitetään aina *puuna* (tree). Rakenteellisesti samankaltaisista tietoyksiköistä koostuva tietojen kokoelma yhdistetään toiseen rakenteellisesti homogeeniseen tietojen kokoelmaan *kuvaus* (map) -konstruktorilla. Kuvassa 4 annetaan BNF- määrittely yo. konstruktoreille [Niemi & Järvelin -88 ja -91]. Seuraavissa kappaleissa tarkastellaan jokaista konstruktoria erikseen.

```
      <n-tuple> -> t(<component sequence>)
(a)   <component sequence> -> <component> {,<component>}*
      <component> -> <any legal Prolog term>

      <set> -> [<component sequence>]
(b)   <component sequence> -> <component> {,<component>}*
      <component> -> <any legal Prolog term>

      <tree> -> <tree indicator>(<component sequence>)
(c)   <tree indicator> -> <any legal Prolog atom>
      <component sequence> -> <component> {,<component>}*
      <component> -> <any legal Prolog term>

      <map object> -> [<map pair> {,<map pair>}*]
(d)   <map pair> -> <map indicator>(<domain element>,<range element>)
      <map indicator> -> <any legal Prolog atom>
      <domain element> -> <any legal Prolog term>
      <range element> -> <any legal Prolog term>
```

Kuva 4. BNF- määrittelyt konstruktoreille: (a) järjestetty jono, (b) joukko, (c) puu ja (d) kuvaus. Niemen ja Järvelinin [Niemi & Järvelin -88 ja -91] julkaisujen pohjalta koottu esitys.

Järjestetyllä jonolla tarkoitetaan siis rakenteellista termiä, joka sisältää järjestetyn joukon rakenteellisesti homogeenisia termejä. Järjestetyllä jonolla on seuraava eksplisiittinen esitystapa

$$t(e_1, e_2, e_3,\ldots e_n).$$

Tässä esityksessä sulkeiden sisään on lueteltu jonon komponentit. Termi $t(e_1, e_2, e_3,\ldots e_n)$ on laillinen järjestetty jono vain, mikäli termit $e_1, e_2, e_3,\ldots e_n$ ovat rakenteellisesti keskenään homogeenisiä. Esimerkiksi $t(t(e, e_2, e_3), t(t(e, e_4, e_5), t(e_1)))$ on laillinen järjestetty jono, koska sen molemmat komponentit ovat rakenteellisesti homogeenisia – tässä tapauksessa järjestettyjä jonoja. Sen sijaan termi $t(t(e_1, e_2), e_3, [e, e_2, e_3])$ ei ole järjestetty jono, koska sen komponentit ovat heterogeenisia sisältäen järjestetyn jonon, atomin ja joukon.

Joukolla tarkoitetaan rakennetta, joka muodostetaan keskenään rakenteellisesti homogeenisista komponenteista. Periaatteellisesti ne eroavat toisistaan vain suhteessa järjestyksen olennaisuuteen. Toisin sanoen, jos järjestys on epäolennaista, niin muodostetaan homogeenisista komponenteista joukko. Joukko esitetään matemaattisesti yleensä kirjaamalla komponentit aaltosulkujen sisään. Kuitenkin kuvassa Y joukko esitetään Prolog-listana

$$[e_1, e_2, e_3,\ldots e_n].$$

Kun rakenteistettava tieto koostuu rakenteesta, jonka heterogeenisten komponenttien keskinäinen järjestys on olennainen, niin silloin tieto esitetään puuna. Puun eksplisiittinen esitystapa on seuraava termi

$$nimi(e_1,e_2,\ldots e_n).$$

Siinä *nimi* voi olla mikä tahansa atomi ja alkiot $e_1, \ldots, e_n$ puolestaan puun komponentteja eli haaroja.

Kuvaus- konstruktori yhdistää *lähtöjoukon* (domain) alkioita arv*ojoukon* (range) alkioihin. Kuvauksen yhteydessä sekä lähtö- että arvojoukon on sisällettävä keskenään rakenteellisesti homogeenisiä alkioita, mutta lähtöjoukon alkioiden ja arvojoukon alkioiden välinen homogeenisuus ei ole olennaista. Kuvaus esitetään muodossa

$$[map(e_1,e_2), map(e_3,e_4), \ldots],$$

jossa siis komponentit e_1 ja e_3 ovat lähtöjoukon alkioita ja komponentit e_2 ja e_4 ovat puolestaan arvojoukon alkioita. Kuvaus yhdistää tietyn lähtöjoukon alkion tiettyyn

arvojoukon alkioon map-nimisenä binääripuuna. Esimerkiksi kuvauselementillä map(t(tampere,helsinki),t(train, 130, 1.8, 24.90)) voidaan yhdistää kaksi järjestettyä jonoa toisiinsa siten, että kaupunkien välistä reittiä merkitsevään jonoon t(tampere,helsinki) kiinnitetään toinen jono, joka sisältää informaatiota kyseisestä reitistä.

On syytä painottaa, että tämän tutkielman jatkossa konstruktoreilla tarkoitetaan vain em. logiikkaohjelmointipohjaisia konstruktoreita. Esimerkiksi oliotietokannat tarjoavat toteuttajille joukon em. konstruktoreiden kanssa analogisia konstruktoreita ja niistä poikkeavia konstruktoreita. Usein oliotietokannat sisältävät seuraavat konstruktorit: jonot (tuple), joukot (set), taulukot (array) ja listat (list). [Paton et al. 1996]

5. Konstruktori-orientoituneisuuden, Prologin, asiantuntijajärjestelmien ja nopean prototyypityksen keskinäinen yhteensopivuus

5.1. Prolog toteutusvälineenä

Aiemmissa luvuissa on käsitelty pikaprototyypityksen hyödyllisyyttä ja myös asiantuntijajärjestelmien ominaisuuksia on luonnehdittu. Tässä luvussa pohditaan Prologin soveltuvuutta asiantuntijajärjestelmien, konstruktorien ja pikaprototyypityksen tekoon. On osoitettu, että keskeiset konstruktorit voidaan toteuttaa Prolog-pohjaisesti [Niemi & Järvelin 1991]. Nämä konstruktorit ovat sovellutusriippumattomia ja ne tukevat sellaisten sovellusten käsittelyä, joissa on käsiteltävä monimutkaisia rakenteita tietojen välillä.

Prolog sellaisenaan soveltuu hyvin nopean prototyypityksen välineeksi, kuten mm. Chiang ja Vasconselos ryhmineen ovat artikkeleissaan [Chiang 2004 ja Vasconcelos et al. 2004] osoittaneet. Prolog-ilmaukset perustuvat logiikan käyttöön suhteiden määrittelyssä, mikä lähtökohtaisesti tarjoaa hyvän uudelleenkäytettävyyden asteen (esim. listojen yhdistämiseen määriteltyä predikaattia voidaan käyttää myös listan jakamiseen kahteen mielivaltaiseen osaan). Tämä yhdessä Prologin korkealla abstraktiotasolla olevien ilmausten kanssa tukee ilmaisuvoimaisen ja tiiviin koodin muodostamista, mikä puolestaan nopeuttaa ohjelmointia ja siten tukee nopeaa prototyypin kehittämistä. Deklaratiivisuuteen pyrkivänä ohjelmointikielenä Prolog itsessään sisältää tarvittavan prosessoinnin automaattisesti ja tässä vaiheessa voidaan haluttaessa sivuuttaa proseduurien tehokkaat toteutukset, jotka voidaan ottaa huomioon vasta varsinaisen ohjelmiston toteutuksen yhteydessä. Tämä tukee nopeaa prototyypitystä.

Lisäksi Prolog-määrittelyt ovat ajettavissa eli syntynyt prototyyppi on testattavissa. Tutkielmassa syntynyt ohjelmisto on eräs esimerkki tästä. Esimerkkiohjelmiston yhteydessä tullaan testaamaan ohjelmiston keskeiset vaatimusmäärittelyt. Mikäli prototyyppi on systemaattisesti tehty konstruktori-orientoitunut Prolog-pikaprototyyppi, niin sen taloudellisuus kasvanee jatkossa entisestään. Tämä johtuu siitä, että sen uusiokäyttö toisen ohjelmiston pikaprototyyppinä saattaa olla suhteellisen helppoa, kuten esimerkkiohjelmiston yhteydessä tullaan toteamaan.

Tekoälysovellusten toteuttamiseen Prolog on paljon käytetty ohjelmointikieli. Itse asiassa se lienee nykyisin käytetyin tekoälysovellusten ohjelmointikieli (esimerkiksi [Bratko 2001]). Asiantuntijajärjestelmän tarvitseman tietämyskannan rakenteellinen organisointi voidaan samoin saavuttaa Prolog-ympäristössä käyttämällä em. konstruktori-orientoitunutta tapaa tiedon esittämisessä. Tämä osoitetaan esimerkkiohjelmiston yhteydessä. Prologia on käytetty muunkin tyyppisten asiantuntijajärjestelmien, kuten agenttijärjestelmien (esim. [NED-2]) tekemiseen.

5.2. Prolog-perusteisten konstruktorien soveltuvuus asiantuntijajärjestelmien nopeaan prototyypitykseen

Konstruktorien avulla jäsennelty tieto on rakenteistettu systemaattisesti ja tietyn konstruktorin mukaisesti organisoidun tiedon käsittelyyn on kehitetty joukko valmiita operaatioita. Konstruktorit ovat käsitteellisiä rakenteita, jotka antavat useita eri mahdollisuuksia toteuttaa ne fyysisesti varsinaista järjestelmää toteutettaessa, mikä siis antaa vapauksia myös koko järjestelmän toteutustavan valintaan. Kun tieto on rakenteellisesti organisoitu, voidaan sitä käsittelevä toteutus vaihtaa perustuen konstruktoreilla kuvattuun tietojen rakenteellisuuteen.

Systemaattisesti rakenteistettu tieto helpottaa myös itse tiedon muokkaamista, mikäli prototyyppiä testattaessa huomataan puutteita tiedoissa tai epäolennaisia tietoja. Toisin sanoen on siis mahdollista muokata vain itse rakenteistettua tietoa, eikä tarvitse juurikaan muuttaa sitä käsittelevää sovellusta. Konstruktoreille luodut operaatiot sisältävät itsessään niiden yleisen Prolog-pohjaisen toteutuksen ja tarjoavat täten käyttäjälle Prologin perinteisiä ilmauksia korkeammalla abstraktiotasolla olevia ilmauksia (ns. meta-sääntöjä (ks. liite 2) [Niemi & Järvelin 1991]). Tämä puolestaan nopeuttaa prototyypin kehittämistä puhtaaseen Prologiin verrattuna. Kun siis myös osa rakenteesta on valmiiksi suunniteltua ja tiedon käsittelyyn on olemassa valmiita menetelmiä, on ajettavan prototyypin rakentamiseksi käytettävä työmäärä pienempi ja näin ollen myös poisheitettävyys toteutuu helpommin. Mikäli resursseja käytettäisiin paljon prototyypin kehittämiseen, olisi vaikeaa hylätä prototyyppi varsinaisen ohjelmiston rakentamista aloitettaessa.

Asiantuntijajärjestelmien tietämyskanta on monissa sovellutuksissa rakenteellisesti monimutkainen, jolloin konstruktoreihin liittyvä systemaattisuus tukee niihin liittyvän rakenteellisuuden mallintamista. Edellä esitetyn konstruktorikokoelman peruslähtökohtana on ollut se, että kombinoimalla niitä keskenään, voidaan useimmissa sovellutuksissa tarvittava rakenteellisuus esittää.

Tutkielman keskeisenä tavoitteena on tutkia Prolog-pohjaisen konstruktori-orientoituneen lähestymistavan soveltuvuutta asiantuntijajärjestelmän pikaprototyypitykseen. Aiemmin on keskusteltu kyseisistä piirteistä erikseen ja niiden keskinäisistä yhteyksistä. Seuraavaksi muodostetaan esimerkkisovellutus, jossa kaikki tutkielman em. mielenkiintoalueet esiintyvät.

6. Esimerkkisovellutus ja sen toteuttaminen

6.1. Esimerkkiasiantuntijajärjestelmän luonnehdinta

Esimerkkinä käytetään nopean prototyypin rakentamista matkasuunnitteluasiantuntijaohjelmistoksi. Se avustaa käyttäjäänsä matkan suunnittelussa asiantuntijajärjestelmien periaatteiden mukaisesti tarjoten ratkaisuja ja käyttäjän antamiin tietoihin perustuen. Matkasuunnitelman tekeminen on sinänsä tunnettu ongelma. Niin kauan kuin on matkusteltu paikasta toiseen, on myös pohdittu sitä, miten matkan voi tehdä pienimmällä mahdollisella resurssihävikillä. Jo ennen tietoteknologian syntymistä mietittiin lähinnä kulkukelpoista reittiä ottamalla huomioon sää ja maasto sekä pyrittiin pohtimaan voimien ja ajan kulumista sekä lauman tai joukon heikoimpien jaksamista. Nykyäänkin sama ongelma on tuttu sotilasjohtajille pohdittaessa sotastrategioita ja liikuteltaessa joukkoja.

Tämän tutkielman mielenkiinnon kohteena on kuitenkin rauhanomainen kulkeminen eri kaupunkien välillä. Ongelmaa on kutsuttu ns. kauppamatkustajaongelmaksi jo vuosia. Kauppamatkustajaongelma on eräs graafiteorioiden perusongelmista. Sitä ja sen eri muunnelmia tutkitaan edelleen aktiivisesti. Ongelman perusajatuksena on löytää kauppamatkustajalle lyhyin reitti annettujen kaupunkien joukossa siten, että hän kulkee jokaisen kaupungin kautta. Kauppamatkustajaongelman, sen sovelluksien ja ratkaisujen esittelyyn on omistettu ainakin yksi kokonainen internetsivusto. [TSP]

Tässä tutkielmassa tarkastellaan matkustamista kaupungista toiseen käyttämällä julkisia kulkuneuvoja. Nykyään on olemassa monenlaisia tällaista matkansuunnittelua avustavia ohjelmistoja. Internetistäkin niitä löytyy useita. Ne käsittelevät kaupunkien tietyn tyyppisten julkisten kulkuneuvojen aikatauluja (esim. [Repa]) tai kaupunkien välisiä kulkuyhteyksiä (esim. [VR]). Ongelma ei siis ole vanhentunut eikä tämän prototyypin olekaan tarkoitus olla ensimmäinen ko. sovellusalan ohjelma. Tutkielman käsittelemä prototyyppi sisältää kuitenkin ominaisuuksia, joita tiettävästi ei löydy tunnetuista ohjelmistoista. Esimerkkinä kehitettävän RoutePlanner- reitinsuunnittelijaohjelman tärkeimpiä ominaisuuksia on mahdollisuus ottaa reittisuunnittelussa huomioon useita eri tekijöitä samalla kertaa. Esimerkiksi lentopelko, merisairaus, tietyn

kaupungin välttäminen ja erilaiset aikamääreet voivat kaikki olla samalla kertaa rajoittamassa niitä vaihtoehtoja, jotka käyttäjälle soveltuvat.

Seuraavaksi tarkastellaan järjestelmää, jossa reittisuunnittelulla on keskeinen rooli. Fonseca ryhmineen on kehittänyt asiantuntijajärjestelmän, joka auttaa reittivalintojen tekemisessä [Fonseca et al. 2003]. Ryhmän kehittämän asiantuntijajärjestelmän tavoitteena on avustaa ajallisesti lyhimmän ja nopeimman reitin löytämisessä. Heidän ongelmansa on kuitenkin eri kuin tässä tutkielmassa, sillä he ovat keskittyneet ennen kaikkea kaupunkien liikenneruuhkiin ja niiden minimointiin. Kyseinen järjestelmä on lisäksi lähinnä yksityisautoilijoille suunnattu apuväline. Fonsecan artikkelin mukaan Amerikkalaiset viettivät vuonna 1992 keskimäärin 40 tuntia ruuhkassa istuen. Vuonna 1997 määrä oli noussut jo 57 – 82 tuntiin [Fonseca et al. 2003]. Autojen lukumäärä tuskin on vähentynyt tuon tutkimuksen jälkeen. Toisin sanoen reittisuunnittelu näyttää eri syistä olevan edelleenkin merkittävä ongelma.

6.2. Tiedon rakenteellinen mallintamien RoutePlanner-esimerkkiohjelmistossa

Lähtökohdan ollessa prototyypin nopea kehittäminen konstruktori-orientoituneesti, on järjestelmässä käsiteltävä tieto esitettävä konstruktorien avulla. Esimerkkinä käytettävän asiantuntijajärjestelmän tiedot ovat tietokannassa, joka on rakenteellisesti mallinnettu konstruktoreilla. Tietokannan sisältämän tiedon käsittelyyn ja päättelyn suorittamiseen on puolestaan olemassa joukko erilaisia operaatioita. Nämä operaatiot muodostavat tietokannan kanssa asiantuntijajärjestelmän tietämyskannan.

RoutePlannerissa reittitiedot esitetään kuvaus- konstruktoreilla, joissa jokaiseen reittiin kiinnitetään tietoja reitin ominaisuuksista. Kaupunkien välinen reitti esitetään muotoa t(A, B) olevana järjestettynä jonona, joka ilmaisee perusyhteyden lähtöpaikan A ja kohdepaikan B välillä. Tällä tavalla muodostettuun lähtöjoukon alkioon liittyvä kohdejoukon alkio esitetään myös järjestettynä jonona t(A, B, C, D), jossa A ilmaisee matkustustavan, B etäisyyden, C arvioidun ajan ja D hinnan. Sama kaupunkien väli voi siis esiintyä useampaan kertaan lähtöjoukon alkiona, koska sama väli saattaa olla mahdollista kulkea eri kulkuvälinein tai muuten erilaisin ominaisuuksin. Esimerkiksi Tampereelta Helsinkiin on mahdollista kulkea linja-autolla, lentokoneella tai junalla. Kuhunkin kulkutapaan liittyen on puolestaan valittavissa sekä ajankäytöltään että hin-

naltaan vaihtoehtoisia yhteyksiä. RoutePlannerin tietämyskanta sisältää siis kuvan 5 esimerkin mukaisesti organisoidun reitti-informaation (kuva 5.a). Esimerkkinä tietämyskannan sisältämistä säännöistä on kuvassa esitettynä lähtöpaikka- kriteerin käsittelyyn liittyvä Prolog predikaatti (kuva 5.b). Se tarkistaa, onko kokonaisreitin ensimmäisen perusreitin lähtöpaikka ”CityA” eli kaupunki, jonka käyttäjän on antanut lähtöpaikka- kriteeriä (start_point_criterium) kysyttäessä.

(a) map(t(tampere,helsinki), t(train, 130, 1.8, 24.90)).

(b) validate_criterium([t(CityA,_;_,_,_)|_], start_point_criterium(CityA))

Kuva 5. Esimerkki RoutePlannerin tietämyskannan osista: (a) Kuvaus reitistä (b) Kriteerin ”Lähtöpaikka” käsittelyyn liittyvä sääntö.

Itse käsittelyn edetessä jokainen reittivaihtoehto kuitenkin transformoidaan (predikaatti tupleconcatenation ks. liite 2) omaksi järjestetyksi jonokseen niin, että jokaista eri reittivaihtoehtoa kohti on evaluoinnin ajan käytössä yksi jono. Toisin sanoen esimerkiksi kuvaus *map(t(TownA,TownB),t(Vehicle,Distance,Time,Price)* muutetaan evaluoinnin ajaksi muotoon *t(TownA,TownB,Vehicle,Distance,Time,Price)*.

Järjestelmän sisältämä reitti-informaatio esitetään em. tavalla. Tietämyskannassa saatavilla olevat kaupungit ja kulkutavat niiden välillä selvitetään esimerkiksi reittikuvauksia analysoimalla. Esimerkiksi reittikuvauksen

map(t(tampere,helsinki), t(train, 130, 1.8, 24.90))

olemassa olemiseen liitetään seuraavat tulkinnat:

1) Järjestelmä sisältää kaupungin ”tampere”.
2) Järjestelmä sisältää kaupungin ”helsinki”.
3) Järjestelmässä on kulkuväline ”train”.
4) Kaupunkien ”tampere” ja ”helsinki” välillä on kulkuyhteys.
5) Käsillä oleva kulkuyhteys koskee vain ”train”-kulkutapaa.
6) Kulkuyhteyden pituus on 130km tällä kulkutavalla.
7) Kulkuyhteyden hinta on 24,90 euroa tällä kulkutavalla.
8) Kulkuyhteyden kesto on 1,8 tuntia (n.1h 50min) tällä kulkutavalla..

Yllä oleva esimerkkimallinnus havainnollistaa yhteen reittikuvaukseen sisältyvien tulkintojen määrää, joka konstruktoreiden taakse kätkeytyy.

Tutkielman esimerkkiprototyypin tietämyskanta sisältää itsessäänkin vajavaisen joukon em. tavalla mallinnettuja reittejä. Varsinaiseen ohjelmistoon reitti-informaatiota olisi lisättävä paljon kattavan reitistön luomiseksi. Prototyypin tapauksessa reittitietokanta on kuitenkin riittävän laaja kattavien testien suorittamiseksi. Kuvassa 6 on esitettynä osa tästä RoutePlanner- prototyypin sisältämästä reitti-informaatiosta.

```
map(t(helsinki,turku), t(train, 120, 2, 24.90)).
map(t(turku,helsinki), t(train, 120, 2, 24.90)).
map(t(helsinki,tampere), t(train, 130, 1.8, 24.90)).
map(t(tampere,helsinki), t(train, 130, 1.8, 24.90)).
map(t(helsinki,tampere), t(train, 130, 2.1, 20.90)).
map(t(tampere,helsinki), t(train, 130, 2.1, 20.90)).
map(t(helsinki,tampere), t(bus, 140, 3.0, 17.90)).
map(t(tampere,helsinki), t(bus, 140, 3.0, 17.90)).
map(t(helsinki,tampere), t(bus, 150, 3.75, 18.90)).
map(t(tampere,helsinki), t(bus, 150, 3.75, 18.90)).
map(t(helsinki,tampere), t(plane, 125, 0.75, 44.90)).
map(t(tampere,helsinki), t(plane, 125, 0.75, 44.90)).
map(t(tampere,jyvaskyla), t(train, 140, 1.5, 19.40)).
map(t(jyvaskyla,tampere), t(train, 140, 1.5, 19.40)).
map(t(turku,jyvaskyla), t(train, 260, 3.5, 39.10)).
map(t(turku,joensuu), t(train, 460, 9.8, 57.00)).
… Tästä karsittu reittejä tilan säästämiseksi …
map(t(helsinki,tallinna), t(ship, 200, 2, 15)).
map(t(tallinna,helsinki), t(ship, 200, 2, 15)).
map(t(helsinki,tallinna), t(plane, 200, 1, 37)).
map(t(tallinna,helsinki), t(plane, 200, 1, 37)).
map(t(turku, tukholma), t(ship, 400, 5, 25)).
map(t(tukholma, turku), t(ship, 400, 5, 25)).
map(t(helsinki, tukholma), t(ship, 450, 6, 20)).
map(t(tukholma, helsinki), t(ship, 450, 6, 20)).
map(t(helsinki, tukholma), t(ship, 450, 6, 20)).
map(t(tukholma, helsinki), t(ship, 450, 6, 20)).
map(t(helsinki, tukholma), t(plane, 450, 2.2, 55)).
map(t(tukholma, helsinki), t(plane, 450, 2.2, 55)).
map(t(tampere, lontoo), t(plane, 900, 4, 75)).
map(t(lontoo, tampere), t(plane, 900, 4, 75)).
```

Kuva 6. Osa RoutePlannerin sisältämästä reitti-informaatiosta l. reittitietokannasta.

6.3. Järjestelmän toiminnan kuvaus

RoutePlanner -ohjelmiston tavoite on mahdollistaa lukuisten erilaisten rajoitusten ilmaiseminen matkasuunnitelmaa tehtäessä. Nykyään ihmisillä on monenlaisia vaatimuksia matkustamisen suhteen ja RoutePlanner pyrkii ottamaan ne huomioon. RoutePlanner kykenee suunnittelemaan matkan perustuen käyttäjän toiveisiin erilaisista matkustusmuodoista. Nykyiset ohjelmat eivät osaa yhdistää eri matkustusmuodoilla kuljettavia reittiosuuksia toisiinsa. Muista järjestelmistä poiketen RoutePlanner ottaa huomioon käyttäjän määrittelemiä erityistoiveita, kun perinteiset järjestelmät on suunniteltu ottamaan huomioon lähtö- ja päätepisteet yhteen matkustustapaan perustuen. RoutePlannerissa käyttäjä voi määritellä

1) lähtöpaikan,
2) päätepisteen,
3) halutut välietapit,
4) kierrettävät välietapit,
5) välietappien määrän (vähintään ja korkeintaan),
6) matkustusajan (vähintään ja korkeintaan),
7) matkan pituuden (vähintään ja korkeintaan),
8) hinnan (vähintään ja korkeintaan),
9) ei halutut kulkuneuvot (johtuen lentopelosta, merisairaudesta, jne.) ja
10) halutut kulkuneuvot.

RoutePlannerissa on myös mahdollista jättää osa rajoitteista pois. Toisin sanoen edellä mainituille rajoitteille käyttäjä voi antaa arvon ”any” indikoimaan, että kyseinen rajoite on hänelle yhdentekevä. Minimissään käyttäjän ei siis tarvitse määritellä mitään ominaisuutta matkan suunnittelua aloitettaessa. Tässä tapauksessa RoutePlanneria voi käyttää laatimaan listan kaikista järjestelmän sisältämistä matkoista, jotka voidaan tehdä järjestelmän sisältämien kaupunkien välillä erilaisia kulkuvälineitä käyttämällä. Esimerkiksi Tampere-Helsinki väli löytyisi sekä bussilla, junalla että lentokoneella suoritettavana matkana ja näistäkin vielä useampana erinopeuksisena ja -hintaisena versiona. Lisäksi mukana olisivat myös reitit Turku-Tampere-Hämeenlinna-Helsinki, Tampere-Hämeenlinna-Helsinki ja Tampere-Hämeenlinna-Riihimäki-Helsinki jne. vastauksen sisältäessä kaikki mahdolliset reittiyhdistelmät.

Tutkielman RoutePlanner-prototyyppi on toteutettu käyttämällä niin kutsuttua ”generoi ja testaa” -tekoälyohjelmointimenetelmää. Sen mukaisesti muodostetaan mahdolliset vaihtoehtoiset ratkaisut, joista sitten testaamalla rajoitetaan ratkaisuvaihtoehdoksi sopivat. Vaikka ”generoi ja testaa” -menetelmä ei välttämättä olekaan tehokkain tapa tämänkaltaisen ohjelmiston kehittämiseen, se tarjoaa nopeasti toteutettavan keinon prosessointiin ja pikaprototyypitykselle on muutenkin tunnusomaista, että siinä tehokkuusvaatimukset sivuutetaan.

Prototyypin toteutuksen ytimenä toimii Prolog-predikaatti nimeltään tuple_set. Tuple_set-predikaatti muodostaa annetusta joukosta sellaisen joukon, joka sisältää jokaisen alkuperäisen joukon *potenssijoukkoon* (powerset = P) kuuluvan alkion (itsessään joukko) jokaisen mahdollisen *järjestyskombinaation*. RoutePlannerissa lähtöjoukkona oleva joukko on kaikkien järjestelmän sisältämien reittikuvausten joukko. Em. tavalla generoidusta massiivisesta joukosta tuple_set-predikaatti poimii sen osajoukon, jonka alkiot täyttävät annetut kriteerit saaden näin joukon ratkaisuvaihtoehtoja (ratkaisujoukko voi olla myös tyhjä joukko).

On huomattavissa miten suureksi tietomäärä kasvaa kun tätä potenssijoukon alkioiden kaikkien järjestyskombinaatioiden joukkoa aletaan muodostaa. Esimerkiksi alkuperäisen joukon ollessa kolmealkioinen (n=3), sisältää potenssijoukko jo 8 alkiota (2^n) ja edelleen tästä joukosta tehty kaikkien alkioiden kaikki permutaatiot sisältävä joukko sisältää jo 16 alkiota. Tämän generoidun joukon alkioiden täsmällinen määrä on

$$\sum |s|! \text{ , } \textit{missä } |s| \textit{ on joukon s koko, kun s on eräs P(S):n osajoukko, ja S on puolestaan lähtöjoukko.}$$

Kuvassa 7 on esitettynä kahden ja kolmen alkion joukkojen (Kuva 7.a) potenssijoukot (Kuva 7.b) ja lopulliset permutaatiojoukot (Kuva 7.c) sekä taulukko alkiomääristä erilaisien alkujoukkojen alkiomäärien kohdalla (Kuva 7.d).

(A)
<1, 2> (n=2)

<1, 2, 3> (n=3)

(B)
<1, 2>, <1>, <2>, <>

<1, 2, 3>,
<1, 2>, <1, 3>,
<2, 3>, <1>,
<2>, <3>, < >

(C)
<1, 2>, <2, 1>, <1>, <2>, < >

<1, 2, 3>, <1, 3, 2>, <2, 1, 3>,
<2, 3, 1>, <3, 1, 2>, <3, 2, 1>,
<1, 2>, <1, 3>, <2, 1>, <2, 3>,
<3, 1>, <3, 2>,
<1>, <2>, <3>, < >

(D)

n:	2	3	4	5	6	7	8	-	n
p:	4	8	16	32	64	128	256	-	2^n
f:	5	16	65	326	1957	13700	109601	-	(*)

n_= lähtöjoukon alkioiden määrä, p_= potenssijoukon alkioiden määrä,
f _= potenssijoukon kaikkien alkioiden kaikkien permutaatioiden joukon alkioiden määrä, (*) ks. kaava edellä

Kuva 7. Lähtöjoukon (A), sen potenssijoukon (B) ja sen alkioiden permutaatioista muodostetun joukon (C) keskinäiset suhteet, sekä joukkojen kokoja (D).

Itse tuple_set-predikaatin Prolog-toteutus on esitetty kuvassa 8. Predikaatti tuple_set käyttää toiminnassaan apunaan järjestelmäpredikaattia findall, joka muodostaa joukon T_Set niin, että ottaa mukaan kaikki mahdolliset Good_Set-muuttujan arvot. Jokainen kelvollinen Good_Set puolestaan on get_one_possible_set-predikaatin em. tavalla potenssijoukon alkioita permutoimalla luoma joukko, joka on läpäissyt check_criteria-predikaatin suorittamat kriteeritarkistukset. Toisin sanoen get_one_possible_set-predikaatti muodostaa em. massiivisen joukon alkioita, jotka get_one_good_set-predikaatti tarkastuttaa kriteerien osalta chech_criteria-predikaatilla. Tarkastuksen läpäisseet joukot, eli tutkielman tapauksessa reitit, findall-predikaatti kerää T_Set-joukoksi. Tästä joukosta poistetaan vielä mahdolliset kaksoisesiintymät, jonka jälkeen ratkaisuehdotusten joukko on valmis.

```
tuple_set(Set, Criteria, Final_T_Set):-
        findall(Good_Set,get_one_good_set(Set,Criteria,Good_Set),T_Set),
        remove_duplicates(T_Set,Final_T_Set).

get_one_possible_set(Set,Possible_Set):-
        power_set(Set,P_Set),
        permutation(P_Set,Possible_Set).

get_one_good_set(Set,Criteria,Good_Set):-
        get_one_possible_set(Set,Good_Set),
        Good_Set\=[],
        check_criteria(Good_Set,Criteria).
```

Kuva 8. Tuple_set -predikaatti RoutePlanner -prototyypissä on ohjelmoitu niin, että se käyttää apunaan kahta muuta predikaattia reittivaihtoehtojoukon luomiseen ja ratkaisuvaihtoehtojen poimimiseen.

Tämä esimerkki selventää myös aiemmin mainittua monikäyttöisyyttä. Tuple_set-predikaatti saa siis parametreinaan lähtöjoukon ja ongelmaan liittyvät kriteerit. Näiden analysoinnin jälkeen se palauttaa lopuksi mahdolliset vastausehdotukset. Lähtöjoukko kerätään suoraan tietokannasta. Mikäli tietokanta nyt muutettaisiinkin sisältämään eläinlajeja ja niiden ominaisuuksia ja käyttäjältä kysyttäisiin nähdyn eläimen tuntomerkkejä (eikä matkustuskriteerejä), voitaisiin näillä predikaateilla suorittaa pienin muutoksin eläinlajin tunnistamiseen liittyviä päättelyitä. Tuple_set-predikaatti keräisi edelleen siinä toteutuksessa samalla periaatteella eläinjoukosta mahdolliset ratkaisuehdotukset tarkistettuaan kriteerit. Eläinlajeihin liittyvät kriteerikohtaiset tarkistukset olisi lisättävä ja potenssijoukon muodostaminen voitaisiin jättää pois. Tietenkin ohjelman antama palaute olisi myös muokattava, mutta verrattuna kokonaisen prototyypin luomiseen, nämä ovat kuitenkin pieniä muutoksia.

RoutePlanner järjestelmän keskeinen toiminta on siis riippuvainen reittikuvauksista, tuple_set-predikaatista ja kriteereiden tarkistuksen suorittavista predikaateista. Lisäksi mukana on tulostamiseen ja käyttäjälle esitettyjen kyselyiden hallintaan liittyviä komponentteja. Mutta kolme edellä mainittua osaa ovat toiminnan kannalta tärkeimmät. Tuple_set-predikaatti on siis kaiken ydin, joka käyttää kahta muuta osiota hyväkseen. Kriteerien tarkistuksen tuple_set jättää check_criteria-predikaatin (ks. määrittely kuvassa 9) huoleksi. Tämä predikaatti toimii niin, että se saa parametreinaan reitin, sekä kriteerien joukon. Kriteerit käydään läpi Prologin menetelmin testaamalla käsiteltävää

reittiä jokaisen kriteerijoukon alkiota (l. yksittäisen kriteerin) mukaan. Mikäli testaus epäonnistuu eli reitti ei täytä kriteerin vaatimuksia, niin check_criteria predikaatti arvottuu epätodeksi ja tuple_set-predikaatissa jätetään kyseinen reitti pois ratkaisuehdotusten joukosta. Kuvassa 9 esitetään em. kriteerin toteutuksessa käytettävät predikaatit. Check_criteria tarkistuttaa jokaisen kriteerijoukon kriteerin ko. kriteerille tarkoitetulla vaihtoehdolla. Kuvassa on esitetty kriteerin connected (yhdistetty) käsittely, joka siis tarkistaa onko reitillä olevat peräkkäiset kaupungit yhteydessä toisiinsa.

```
check_criteria(_,[]):-!.
check_criteria(Route,[Criterium|Criteria]):-
        validate_criterium(Route, Criterium),!,
        check_criteria(Route,Criteria).

validate_criterium([t(_,_,_,_,_,_)],connected_criterium):-!.
validate_criterium([t(_,CityB,_,_,_,_),t(CityB,_,_,_,_,_)],connected_criterium):-!.
validate_criterium([Tuple1,Tuple2|RestRoute],connected_criterium):-
        Tuple1=t(_,City,_,_,_,_),
        Tuple2=t(City,_,_,_,_,_),
        validate_criterium([Tuple2|RestRoute],connected_criterium).
```

Kuva 9. RoutePlannerin kriteerien käsittelyä. Ensin check_criteria -predikaatti, joka käy läpi kaikki kriteerit sisältävän joukon. Alla validate_criterium esimerkki connected -kriteeristä. Tämä osio siis tarkistaa onko reittiehdotuksen kaksi peräkkäistä kaupunkia liitettävissä toisiinsa millään reitillä.

Pikaprototyypityksen ideana on pieneen tietomäärään perustuen testata rakennettavan järjestelmän sisältämän toiminnallisuuden riittävyys. Tästä syystä käytettyyn ”generoi ja testaa” -menetelmään liittyvä ilmeinen tehottomuus ei tässä vaiheessa ole olennainen tekijä. Varsinaista järjestelmää rakennettaessa tehokkuusvaatimukset pitää ottaa aivan eri painolla huomioon. Vaikka prototyyppi toimii testausvälineenä varsinaisen järjestelmän toiminnallisuudelle, sen yhteydessä on mahdollista löytää niitä toteutuksen kohtia, joiden tehokkaaseen toteuttamiseen pitää kiinnittää erityistä huomiota. Yllä olevassa toteutuksessa se osa, joka perustui "generoi ja testaa" -menetelmälle, on juuri tällainen kohta.

6.4. Käyttötapausten testaaminen RoutePlannerilla

RoutePlanner -prototyyppiä testattiin viiden eri käyttötapauksen mukaisesti. Käyttötapaukset valittiin siten, että ne sisältävät kaikki olennaiset järjestelmän tulevat käyttötavat. RoutePlannerin heikon suoritusnopeuden vuoksi jouduttiin kuitenkin jokaiseen aiemmin määriteltyyn käyttötapaukseen lisäämään muutamia rajoitteita. Rajoitteet sinänsä eivät olleet mahdottomia tai outoja, mutta muuttivat hieman käyttötapausten lähtökohtia. Muunnetut käyttötapaukset toimivat kuitenkin nyt testitapauksina ja ajavat asiansa ohjelmiston toiminnallisuuden demonstroinnin suhteen.

Käyttötapaustestauksen ja RoutePlannerin toiminnan havainnollistamiseksi on käyttäjän ja ohjelmiston välinen kanssakäyminen esitetty seuraavissa kuvissa yhden käyttötapauksen osalta. Varsinaiset testitapauskuvaukset esitetään esimerkkiajoina, joiden lopputulokset on esitetty liitteessä 3. Seuraavissa kyselyitä sisältävissä kuvissa on selkeyden vuoksi käyttäjän antamat syötteet värjätty ohjelman ajon jälkeen. Ensimmäisessä kuvassa (Kuva 10) esitellään RoutePlannerin antama alkuinformaatio ja ohjeet, joita ohjelmisto antaa käyttäjälle ennen varsinaisia kyselyitä.

```
Welcome to RoutePlanner!

Database includes 40 routes.
Now then, on with the business. Do you want to see cities available in database?
Answer y. or n.
|: y.

keuruu haapamaki     seinajoki     joensuu     jyvaskyla
kuopio tallinnaturku helsinki      tampere
lontoo  tukholma

Do you want to see vehicles available in database? Answer y. or n.
|: y.

train   bus     ship    plane

Now I will start asking criteria for evaluation, order might seem strange but it is
carefully selected, do not panic, just answer all questions and remember to in-
clude . as the last character of every answer.
```

Kuva 10. RoutePlannerin tervetulotoivotus, jonka jälkeen ohjelma esittelee tietämyskannan tuntemat kaupungit ja kulkuneuvot käyttäjän niin halutessa, sekä antaa vielä ohjeita tuleviin kyselyihin vastaamisesta.

Seuraavaksi käyttäjältä kysytään ongelmatapaukseen liittyvät kriteerit (Kuva 11), joiden mukaan järjestelmä antaa ratkaisuehdotuksen. Kriteerikyselyiden jälkeen ohjelmisto tulostaa annetut tiedot käyttäjän nähtäväksi (Kuva 12) ja pyrkii näin ennen päättelyn aloittamista varmistamaan, että tiedot on annettu oikein.

Give maximum price in Euros and . (or any.) any.

After that criteria database still has 40 possible routes.
Write vehicles NOT accepted as a list [train,plane,bus,ship]. (or any.) [plane,ship,bus].

After that criteria database still has 25 possible routes.
Give maximum combined lenght of trip in kilometres and . (or any.) 250.

After that criteria database still has 21 possible routes.
Give NOT wanted cities as a list ([cityA,cityB,cityC]) and . (or any.) [jyvaskyla,kuopio,joensuu,haapamaki].

After that criteria database still has 8 possible routes.
Give maximum time in hours and . (or any.) 4.

After that criteria database still has 8 possible routes.
Okay, now the first selections have been done and from 40 routes these criterias left 8 possible ones for the rest of the evaluation.
Now I will ask for all the rest of the criterias.

Now give me "g." to "go on" or "a." to "abort evaluation here" ! g.
Okay, on we go...
Give start location and . (or any.) tampere.
Give end location and . (or any.) turku.
Give minimum time in hours and . (or any.) any.
Give minimum combined lenght of trip in kilometres and . (or any.) any.
Give minimum amount of waypoints and . (or any.) any.
Give maximum amount of waypoints and . (or any.) any.
Give cities wanted as a list ([cityA,cityB,cityC]) and . (or any.) any.
Give minimum price in Euros and . (or any.) any.
Write wanted vehicles as a list [train,plane,bus,ship]. (or any.) any.

Kuva 11. RoutePlanner kysyy käyttäjältä ongelmatapauksen rajoitteet.

```
User-Given Criteria are:
 * Starting point of trip is tampere.
 * This trip will end to turku.
 * Trip has to include following cities:  [ --- ] .
 * Trip can not include following cities: [jyvaskyla, kuopio, joensuu, haapamaki].
 * Vehicle(s) [plane, ship] can not be used.
 * Vehicle(s)  [ --- ]  must be used at least once.
 * This trip may include maximum of  [ --- ]  cities.
 * This trip has to include minimum of  [ --- ]  cities.
 * This trip has to cost less than  [ --- ]  Euros.
 * This trip has to cost more than  [ --- ]  Euros.
 * This trip may take maximum of 4 hours to complete.
 * This trip has to take more than  [ --- ]  hours.
 * This trip has to be shorter than 250 kms.
 * This trip has to be more than  [ --- ]  kms long.
The symbol [ --- ] means that you did not lock any restrictions to that matter.

And now after those first selections I made earlier, I will now combine still available routes and
make up all possible route combinations and then I will check which of those satisfy your criteria.

Now give me "g." to "go on" or "a." to "abort evaluation here" ! g.
Okay, on we go...
```

Kuva 12. RoutePlanner tulostaa annetut tiedot. Käyttäjä voi vielä tässäkin vaiheessa halutessaan keskeyttää ohjelman suorituksen.

Kuvassa 13 näytetään, miten RoutePlanner esittää reittiehdotukset käyttäjälle. Käyttäjän antamien rajoitteiden jälkeen vastaukseksi saadaan kaksi mahdollista reittiä, jotka ohjelma tulostaa siten, että käyttäjän olisi ne helppo tulkita. Tulostuksessaan RoutePlanner antaa myös reitin kokonaistiedot ja yksittäisten välietappien ominaisuudet. Käyttäjä voi tulostuksesta eksplisiittisesti tarkistaa, ettei hänen antamiaan kriteerejä ole rikottu.

```
I found more than one solution (2) and these solutions are:

This solution is a trip from tampere to turku and it consists 1 routes, which are:
   ~   tampere to turku by train. Route length of 120kms in 1.7 hours. Price : 22.7 Euros.
The total length of this trip is 120kms. It takes total of 1.7 hours to complete.
The total price of this trip is 22.7 Euros.

This solution is a trip from tampere to turku and it consists 2 routes, which are:
   ~   tampere to helsinki by train. Route length of 130kms in 1.8 hours. Price : 24.9 Euros.
   ~   helsinki to turku by train. Route length of 120kms in 2 hours. Price : 24.9 Euros.
The total length of this trip is 250kms. It takes total of 3.8 hours to complete.
The total price of this trip is 49.8 Euros.
```

Kuva 13. RoutePlannerin tulostusmuoto löydetyiksi reittiratkaisuiksi.

Liitteen 3 testitapausten perusteella pyritään varmistumaan siitä, että järjestelmä tarjoaa käyttäjälle riittävät mahdollisuudet erilaisten kriteerien antamiseen ja että järjestelmä sisältää riittävän toiminnallisuuden. Asiakkaalle nopeaa prototyyppiä esiteltäessä ja hänen kanssaan prototyyppiä testattaessa on painotettava, ettei suorituskyky tai visuaalinen ulkoasu vastaa lopullista ohjelmiston suorituskykyä tai tulostustapaa. Tässä vaiheessa on tärkeintä saada asiakkaan hyväksyntä järjestelmän sisältämälle ilmaisuvoimalle.

6.5. Arviointi ja jatkokehitys

Prototyypin testaus voi periaatteessa johtaa kahteen lopputulokseen: joko prototyypin muuttamiseen vastaamaan asiakkaan toiveita tai varsinaisen ohjelmiston kehittämistyön alkamiseen prototyypin sisältämän ilmaisuvoiman pohjalle. Varsinaisen ohjelmiston kehittäminen alkaa vain jos asiakas hyväksyy prototyypin testauksessa olleiden käyttötapausten riittävyyden ja prototyypin antamat vastaukset niiden yhteydessä. Prototyyppi ja käyttötapaukset on syytä dokumentoida tarkasti varsinaisen ohjelmiston kehittämisen yhteydessä mahdollisesti tulevia ristiriitatilanteita varten.

Tutkielmassa kehitetty RoutePlanner toimii esimerkkinä asiantuntijaohjelmiston nopeasta prototyypistä soveltamalla konstruktori -orientoitunutta Prolog-ohjelmointia. Tutkielman keskeinen tavoite on arvioida RoutePlannerin toteutuksen pohjalta, miten hyvin konstruktori -orientoitunut Prolog-pohjainen lähestymistapa soveltuu asiantuntijajärjestelmien nopeaan prototyypitykseen. Lisäksi tarkastellaan RoutePlanner -prototyypin jatkokehitysmahdollisuuksia.

Tutkielmassa kehitetyn prototyypin toteutuksessa esiintyi pieniä ongelmia. Pääsääntöisesti konstruktori -orientoituneisuus tuki reittitietokannan rakenteellisuuden hallintaa. Samoin konstruktoreille määritellyt operaatiot mahdollistivat korkealla abstraktiotasolla olevat ilmaisut, jotka omalta osaltaan nopeuttivat prototyypin laatimista. Suurin ongelma liittyi ”generoi ja testaa” -menetelmän soveltamiseen toteutuksessa. Suuren tietomäärän läpikäyminen tällä metodilla on varsin raskasta ja hidasta johtuen generoitujen vaihtoehtojen suuresta määrästä. Tutkielmassa osoitettiin, että ”generoi ja testaa” -menetelmällä tuotettujen reittivaihtoehtojen määrä kasvaa nopeasti perustuen tietämyskannassa oleviin perusreitteihin. Prototyypissä kyseisestä piirteestä ei ole

haittaa, mutta varsinaista ohjelmistoa kehitettäessä ko. piirre pitää korvata huomattavasti tehokkaammalla käsittelyllä.

RoutePlanner on valitettavasti siinä suhteessa vajavainen, että se ei tällä hetkellä pysty käyttämään tiettyä reittiyhteyttä kahdesti reittipolussa. Tietokannan reittiyhteydet voivat olla vain kerran mukana järjestelmän tuottamassa kokonaisreitissä. Tämä piirre estää esimerkiksi poikkeamisen samassa kaupungissa sekä meno- että tulomatkalla, mikäli kaupunkiin johtaa vain yksi reitti. Tekstikäyttöliittymäkin jättää paljon kehittämistä. Olisi esimerkiksi huomattavasti mukavampaa valita kohteita ja kriteereitä perustuen graafisiin käyttöliittymiin kuten lomakkeisiin ja karttoihin. Jos tuloksena tuotettavat reittivaihtoehdot olisi mahdollista visualisoida kartalle, olisi myös tulosten tulkinta havainnollisempaa.

Puutteistaan huolimatta RoutePlanneria voidaan pitää aitona asiantuntijajärjestelmänä, joka on muodostettu nopean prototyypityksen periaatteita noudattaen. Prototyyppi on ajettavissa ja mahdollistaa täten käyttötapausten testaamisen. RoutePlanner selvisi kaikista niistä testitapauksista, joista prototyypin oletettiin selviytyvän. Itse perusongelma RoutePlannerin taustalla oli monessa mielessä haastavampi kuin monien nykyään reittisuunnitteluun kehitettyjen ohjelmistojen. Täten RoutePlanner tarjoaa myös erään lähtökohdan näiden ohjelmistojen kehittämiseksi. Koska konstruktori-orientoituneella Prolog-ohjelmoinnilla pystyttiin laatimaan toimiva nopea prototyyppi vaativalle asiantuntijajärjestelmälle, on luultavaa, että se on sopiva lähestymistapa nopeaan prototyypitykseen samantyyppisten asiantuntijajärjestelmien yhteydessä.

7. Loppupäätelmä

Tutkielman tutkimusongelmana on ollut arvioida konstruktori-orientoituneen Prolog-pohjaisen lähestymistavan soveltuvuutta asiantuntijajärjestelmien nopeaan prototyypitykseen. Tätä lähestymistapaa sovellettiin reitinsuunnitteluun kykenevän asiantuntijajärjestelmän prototyypin konstruointiin. Prototyyppiä testattiin useisiin käyttötapauksiin perustuen. Niiden avulla voitiin varmistua prototyypin riittävästä toiminnallisuudesta. Tutkielmassa vedetään se johtopäätös, että tutkitulla lähestymistavalla voidaan nopeasti konstruoida ajettavissa ja testattavissa olevia prototyyppejä sellaisille asiantuntijajärjestelmille, joissa rakenteellisuuden mallintaminen ja käsittely on keskeistä.

8. Viiteluettelo

[AI-Depot] Rule-Based systems tutorial, James Freeman-Hargis, tarkistettu maaliskuussa 2008, <http://ai-depot.com/Tutorial/RuleBased.html>.

[Arnold & Bowie 1986] William R. Arnold, John S. Bowie, Artificial intelligence : a personal, commonsense journey. Prentice-Hall, 1986.

[Bailey et al. 2008] Brian Bailey, Jacob Biehl, Damon Cook, Heather Metcalf, Adapting paper prototyping for designing user interfaces for multiple display environments. In *Personal & Ubiquitous Computing Vol.* **12** *Issue 3 March 2008,* 269-277, 2008.

[Bratko 2001] Bratko, Ivan, Prolog programming for artificial intelligence. Addison-Wesley , 2001.

[Chiang 2004] Chia-Chu Chiang, Automated rapid prototyping of TUG specifications using Prolog. In: *Information and Software Technology* **46***(2004)*, 857-873, 2004.

[Collins] Collins Cobuild English language dictionary. Collins Birmingham University International Language Database, HarperCollins Publishers, London, UK 1994.

[Encyclopedia] Encyclopedia Britannican verkkoversio, tarkistettu maaliskuussa 2008, <http://www.britannica.com/>.

[Expertise2Go] Asiantuntijajärjestelmiä Internetiin toteutettuna, tarkistettu maaliskuussa 2008, <http://www.expertise2go.com/>.

[Fikes&Kehler 1985] Richard Fikes and Tom Kehler, The role of frame-based representation in reasoning. In: *Communications of the ACM. Vol.* **28**, *no. 9, 904-920.* ACM, USA, 1985.

[Fonseca et al. 2003] Daniel J. Fonseca, Siwawong Daosuparoch, Gary P. Moynihan and Der-San Chen, A computer-based system for road selection. In: *Expert Systems; vol. **20** Issue 3, 133-140,* July 2003.

[Freudenthal et al. 2004] Adinda Freudenthal, Marc P. A. J. de Hoogh, David V. Keyson, Intelligent product builder: a rapid prototyping environment for software in context aware hardware products. In: *ACM International Conference Proceeding Series; Vol.* **65**, *Proceedings of the conference on Dutch directions in HCI, Amsterdam, Holland*. ACM New York, USA 2004.

[Friedland 1985] Peter Friedland, Special section on architectures for knowledge-based systems. In: *Communications of the ACM. Vol. 28, no. 9, 902-903*. ACM, USA, 1985.

[Genesereth&Ginsberg 1985] Michael R. Genesereth and Matthew L. Ginsberg, Logic programming. In: *Communications of the ACM. Vol. **28**, no. 9, 933-941*. USA, 1985.

[Harmon & King 1985] P. Harmon and D. King, Expert Systems: Artificial Intelligence in Business, Wiley, 1985.

[Hayes-Roth et al. 1983] F. Hayes-Roth, D. Waterman, D. Lenat (edit.), Building Expert Systems, Addison-Wesley, 1983.

[Hayes-Roth 1985] Frederick Hayes-Roth, Rule-based systems. In: *Communications of the ACM. Vol.* **28**, *no. 9, 921 - 932*. ACM, USA, 1985.

[Huang & Chen 2007] M. J. Huang, M. Y. Chen, Integrated design of the intelligent web-based Chinese Medical Diagnostic System (CMDS) – Systematic development for digestive health. In: *Expert Systems with Applications vol.* **32**, *658 - 673,* 2007.

[Kuipers 1994] Benjamin Kuipers, Algernon for Expert Systems, Computer Science Department, University of Texas at Austin, USA, January 1994.

[Lantz 1986] Kenneth E. Lantz, The Prototyping Methodology, Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1986.

[Liao 2004] Shu-Hsien Liao, Expert system methodologies and applications—A decade review from 1995 to 2004. In: *Expert Systems with Applications, Volume* **28**, *Issue 1, 93-103*, January 2005.

[Lister et al. 1988] Raymond Lister, Kamal Ali, Renato Buda, Chris Horsfall, Wray Buntine, GOLD – an expert system for mineral identification from reflectance spectra. In: J. S. Gero and R. Stanton (edit.), *Artificial Intelligence Developments and Applications*. Elsevier Science Publishers B.V. , Netherlands, 1988.

[Mahaman et al. 2003] B. D. Mahaman, H. C. Passam, A. B. Sideris, C. P. Yialouris, DIARES-IPM: a diagnostic advisory rule-based expert system for integrated pest management in Solanaceous crop systems. In: *Agricultural Systems, Vol.* **76**, *Issue 3, p. 1119-1135*, June 2003.

[MicroCreator] Prototyypitysohjelmisto MicroCreator, internetsivusto, tarkistettu huhtikuussa 2008, <http://www.systemsmodeling.com/casetool/public_html/>.

[NED] F. Maier, D. Nute, W. Potter, J. Wang, M. Twery, M. Rauscher, P. Knopp, S. Thomasma, M. Dass, and H. Uchiyama, Efficient Integration of PROLOG and Relational Databases in the NED Intelligent Information System. In: *Proceedings of the 2003 International Conference on Information and Knowledge Engineering (IKE'03), 364-369*, June 2003, Las Vegas, Nevada, USA.

[NED-2] Donald Nute, Walter D. Potter, Frederick Maier, Jin Wang, Mark Twery, H. Michael Rauscher, Peter Knopp, Scott Thomasma, Mayukh Dass, Hajime Uchiyama and Astrid Glende, NED-2: An Agent-Based Support System for Forest Ecosystem Management. In: *Environmental Modelling and Software, Vol.* **14**, No. 5, 2003

[Niemi & Järvelin 1988] Timo Niemi and Kalervo Järvelin, A Meta-Information Approach to Relational Database Representation and Manipulation Based on Prolog.

University of Tampere, Dept. of Computer Science, Series of Publications **A-1988-11**, December 1988.

[Niemi & Järvelin 1991] Timo Niemi and Kalervo Järvelin, Prolog-Based Meta-Rules For Relational Database Representation and Manipulation. In: *IEEE Transactions on Software Engineering, vol.* **17**, *no. 8,* August 1991.

[Niemi & Järvelin 1997] Timo Niemi and Kalervo Järvelin, Constructor-Oriented Query Processing Strategy for NF^2 Relational Queries: Part I. Implementation on Top of a Relational Database. University of Tampere, Dept. of Computer Science, Series of Publications **A-1997-9**, August 1997.

[O'Callaghan et al. 2003] Thomas A. O'Callaghan, James Popple and Eric McCreath, SHYSTER-MYCIN: a hybrid legal expert system. In: *Proceedings of the 9th international conference on Artificial intelligence and law, 103 - 104, Scotland, United Kingdom*, 2003, ACM, USA.

[Ossa et al. 2007] Luis de la Ossa, M. Julia Flores, José A. Gámez, Juan L. Mateo and José M. Puerta, Initial breeding value prediction on Manchego sheep by using rule-based systems. In: *Expert Systems with Applications, Vol.* **33**, *Iss. 1, 96-109*, 2007.

[Paton et al. 1996] Norman Paton, Richard Cooper, Howard Williams and Philip Trinder. Database Programming Languages, Prentice Hall, 1996.

[RadVolution] RadVolution prototyypitysohjelmisto, internetsivusto, tarkistettu huhtikuussa 2008, <http://www.developguidance.com/radvolution_prototype.htm>.

[Repa] Tampereen kaupungin liikennelaitoksen verkossa toimiva Repa-Reittiopas, tarkistettu huhtikuussa 2008, <http://atlas.tripplanner.fi/tkl/fi/>.

[Rissland & Ashley 1987] E. L. Rissland and K. D. Ashley, A case-based system for trade secrets law. In: *Proceedings of the 1st international conference on Artificial intelligence and law, Boston, USA, 1987, 60 - 67*, ACM, New York, USA, 1987.

[Sagheb-Tehrani 1993] Mehdi Sagheb-Tehrani, Expert Systems Development: Some Problems, Motives and Issues in an Exploratory Study. Lund University Press, Sweden, 1993.

[Song et al. 2005] Xiping Song, Arnold Rudorfer, Beatrice Hwong, Gilberto Matos and Christopher Nelson, S-RaP: A Concurrent, Evolutionary Software Prototyping Process. In: M. Li, B. Boehm, and L.J. Osterweil (Eds.), SPW 2005, *Lecture Notes in Computer Science 3840, 164 – 176*, Springer-Verlag Berlin Heidelberg, 2005.

[Sterling 1994] Leon Sterling and Ehud Shapiro, The art of Prolog: advanced programming techniques. Cambridge (Mass.), MIT Press, cop. 1994, MIT Press series in logic programming.

[TSP] Traveling Salesman Problem, internetsivusto, tarkistettu huhtikuussa 2008 <http://www.tsp.gatech.edu/>.

[Vasconcelos et al. 2004] W. Vasconcelos, D. Robertson, C. Sierra, M. Esteva, J. Sabater and M. Wooldridge, Rapid prototyping of large multi-agent systems through logic programming. In: *Annals of Mathematics and Artificial Intelligence* **41**, *135-169*, 2004.

[VB] Visual Basic Resource Center, internetsivusto, tarkistettu huhtikuussa 2008, <http://msdn.microsoft.com/vbrun/>.

[Winston 1984] Patrick Henry Winston, Artificial Intelligence, Addison-Wesley, USA-Canada, 1984.

[Witlox 2005] Frank Witlox, Expert systems in land-use planning: An overview. In: *Expert Systems with Applications, Vol.* **29**, *Issue 2, 437-445*, Elsevier, August 2005.

[VR] Valtion rautateiden verkkopalvelun aikataulu- ja junayhteyssovellus, tarkistettu huhtikuussa 2008, <http://www.vr.fi/heo/index.html>.

Liitteet

Liite 1.

Esimerkki sääntöperusteisesta toteutuksesta [AI-Depot]

1.1.1. Appendix A -- Forward-Chaining Example: Medical Diagnosis

Assertions (Working Memory):

```
A1: runny nose
A2: temperature=101.7
A3: headache
A4: cough
```

Rules (Rule-Base):

```
R1:    if   (nasal congestion)
            (viremia)
       then diagnose (influenza)
            exit
R2:    if   (runny nose)
       then assert (nasal congestion)
R3:    if   (body- aches)
       then assert (achiness)
R4:    if   (temp >100)
       then assert (fever)
R5:    if   (headache)
       then assert (achiness)
R6:    if   (fever)
            (achiness)
            (cough)
       then assert (viremia)
```

Execution:

1. R2 fires, adding (nasal congestion) to working memory.
2. R4 fires, adding (fever) to working memory.
3. R5 fires, adding (achiness) to working memory.
4. R6 fires, adding (viremia) to working memory.
5. R1 fires, diagnosing the disease as (influenza) and exits, returning the diagnosis

1.1.2. Appendix B -- Backward-Chaining Example: Medical Diagnosis

Use same rules/assertions from Appendix A

Hypothesis/Goal: Diagnosis (influenza)

Execution:

1. R1 fires since the goal, diagnosis(influenza), matches the conclusion of that rule. New goals are created: (nasal congestion) and (viremia) and backchaining is recursively called with these new goals.
2. R2 fires, matching goal nasal congestion. New goal is created: (runny nose). Backchaining is recursively called. Since (runny nose) is in working memory, it returns true.
3. R6 fires, matching goal viremia. Back-chaining recursion with new goals: (fever), (achiness) and (cough)
4. R4 fires, adding goal (temperature > 100). Since (temperature = 101.7) is in working memory, it returns true.
5. R3 fires, adding goal (body-aches). On recursion, there is no information in working memory nor rules that match this goal. Therefore it returns false and the next matching rule is chosen. That rule is R5 which fires, adding goal (headache). Since (headache) is in working memory, it returns true.
6. Goal (cough) is in working memory, so that returns true.
7. Now, all recursive procedures have returned true, the system exits, returning true: this hypothesis was correct: subject has influenza.

Liite 2.

Metasääntöjä konstruktoreilla mallinnetun tiedon käsittelyyn

Tässä liitteessä on koottuna joitakin Niemen ja Järvelinin määrittelemiä meta-sääntöjä konstrutoreilla mallinetun tiedon käsittelyyn [Niemi & Järvelin -88, -91 ja -97]. Erityisesti mukaan on valittu tutkielman RoutePlanner-prototyypissä käytetyt operaatiot.

```
%Primitive operations for Tuple objects
%Def. of tupleconcatenation
tupleconcatenation(Tuple1, Tuple2, Tuple3):-
        Tuple1=.. [t|Args1],
        Tuple2=..[t|Args2],
        append(Args1, Args2, Args),
        Tuple3=..[t|Args].

%Def. of tupleprojection
tupleprojection(Tuple, [Ind], t(Value)):-
        arg(Ind, Tuple, Value).
tupleprojection(Tuple, [I|I_set], Res):-
        arg(I, Tuple, Comp),
        tupleprojection(Tuple, I_set, Res1),
        tupleconcatenation(t(Comp), Res1, Res).

%Primitive operations for map objects
%Def. of dom_range
dom_range([], [], []).
dom_range([Pair| T1],[Y| T2],[Z| T3]):-
        arg(1, Pair, Y),
        arg(2, Pair, Z),
        dom_range(T1, T2, T3).

%Def. of is_function
is_function([Pair| T]):-
        arg(1, Pair, X),
        arg(2, Pair, Y),
        sameimage(X, Y, T),
        is_function(T).
is_function([]).

sameimage(X, Y, [Pair| T]):-
        arg(1, Pair, X),
        arg(2, Pair, Y),
        sameimage( X, Y, T).
sameimage(X, Y, [Pair| T]):-
        arg(1, Pair, X1),
        X1\= =X,
        sameimage(X, Y, T).
sameimage(X, Y, T).
sameimage(X, Y, []).
```

```
%Primirive operations for set objects
%Def. of set_subset
set_subset([], Ys).
set_subset([X|Xs], Ys):-
        member(X, Ys),
        set_subset(Xs, Ys).

%Def. of set_identical
set_identical(Set1, Set2):-
        set_subset(Set1, Set2),
        set_subset(Set2, Set1).

%Def. of set_difference
set_difference([], Ys, []):-!.
set_difference([X| Xs], Ys, Zs):-
        member(X, Ys),!,
        set_difference(Xs, Ys, Zs).
set_difference([X| Xs], Ys, [X| Zs]):-
        set_difference(Xs, Ys, Zs).

%Def. of set_union
set_union(X, Y, Z):-
        set_difference(X, Y, X1),
        append(X1, Y, Z).

%Def. of set_intersection
set_intersection([], Set2, []).
set_intersection([X| Xs], Set2, [X| Zs]):-
        member(X, Set2), !,
        set_intersection(Xs, Set2, Zs).
set_intersection([X| Xs], Set2, Zs):-
        set_intersection(Xs, Set2, Zs).

% duplicates can be removed from a set by the primitive operation
% remove_duplicates
remove_duplicates([Member| DuplicateSet], NoDuplicateSet):-
        member(Member, DuplicateSet), !,
        remove_duplicates(DuplicateSet, NoDuplicateSet).
remove_duplicates([Member| DuplicateSet], [Member| NoDuplicateSet]):-
        !, remove_duplicates(DuplicateSet, NoDuplicateSet).
remove_duplicates([],[]).
```

Liite 3.

RoutePlannerin testitapaukset

Testitapaus 1

Kuvaus:

Käyttäjä haluaa matkustaa Tampereelta Turkuun käyttämättä lentokonetta tai bussia ja kulkematta yli 250 km. Matka ei myöskään saa kestää yli 4 tuntia. Käyttäjä ei halua kulkea Jyväskylän, Kuopion, Joensuun eikä Haapamäen kautta. Hän tietää myös, ettei voi kulkea laivalla.

RoutePlanner Source Information:

User-Given Criteria are:
** Starting point of trip is tampere.*
** This trip will end to turku.*
** Trip has to include following cities: [---] .*
** Trip can not include following cities: [jyvaskyla, kuopio, joensuu, haapamaki].*
** Vehicle(s) [plane, ship, bus] can not be used.*
** Vehicle(s) [---] must be used at least once.*
** This trip may include maximum of [---] cities.*
** This trip has to include minimum of [---] cities.*
** This trip has to cost less than [---] Euros.*
** This trip has to cost more than [---] Euros.*
** This trip may take maximum of 4 hours to complete.*
** This trip has to take more than [---] hours.*
** This trip has to be shorter than 250 kms.*
** This trip has to be more than [---] kms long.*
The symbol [---] means that you did not lock any restrictions to that matter.

Result:

I found more than one solution (2) and these solutions are:
This solution is a trip from tampere to turku and it consists 1 routes, which are:
~ tampere to turku by train. Route length of 120kms in 1.7 hours. Price : 22.7 Euros.
The total length of this trip is 120kms. It takes total of 1.7 hours to complete.
The total price of this trip is 22.7 Euros.

This solution is a trip from tampere to turku and it consists 2 routes, which are:
~ tampere to helsinki by train. Route length of 130kms in 1.8 hours. Price : 24.9 Euros.
~ helsinki to turku by train. Route length of 120kms in 2 hours. Price : 24.9 Euros.
The total length of this trip is 250kms. It takes total of 3.8 hours to complete.
The total price of this trip is 49.8 Euros.

Testitapaus 2

Kuvaus:

Käyttäjä haluaa matkustaa Tampereelta jonnekin, mutta ei tiedä minne menisi, kunhan ei joudu Turkuun tai Seinäjoelle. Käytössään hänellä on 100 euroa ja hän haluaa RoutePlannerin kertovan hänelle, mihin hän sillä rahalla pääsee kun on lentopelkoinen, eikä halua käyttää laivaa tai bussia. RoutePlannerille annetaan lähtötiedoiksi siis 100:n euron maksimihinta ja lisäksi käyttäjä haluaa että raha myös menee matkustamiseen ja siksi minimihinnaksi annetaan 90 euroa. Kulkuvälineiksi siis eivät siis kelpaa laiva, lentokone eikä bussi, joten ne annetaan ei haluttuihin kulkuvälineisiin (syöte siis [lentokone, laiva]. prototyyppiä käytettäessä).

RoutePlanner Source Information:

User-Given Criteria are:
** Starting point of trip is tampere.*
** This trip will end to [---] .*

** Trip has to include following cities: [---] .*
** Trip can not include following cities: [seinajoki, turku].*
** Vehicle(s) [plane, ship, bus] can not be used.*
** Vehicle(s) [---] must be used at least once.*
** This trip may include maximum of [---] cities.*
** This trip has to include minimum of [---] cities.*
** This trip has to cost less than 100 Euros.*
** This trip has to cost more than 90 Euros.*
** This trip may take maximum of [---] hours to complete.*
** This trip has to take more than [---] hours.*
** This trip has to be shorter than [---] kms.*
** This trip has to be more than [---] kms long.*
The symbol [---] means that you did not lock any restrictions to that matter.

Result:

I found more than one solution (17) and these solutions are:
This solution is a trip from tampere to jyvaskyla and it consists 5 routes, which are:
- *tampere to haapamaki by train. Route length of 100kms in 1.5 hours. Price : 15 Euros.*
- *haapamaki to keuruu by train. Route length of 20kms in 0.2 hours. Price : 5 Euros.*
- *keuruu to jyvaskyla by train. Route length of 85kms in 1 hours. Price : 13 Euros.*
- *jyvaskyla to joensuu by train. Route length of 200kms in 4.5 hours. Price : 31.4 Euros.*
- *joensuu to jyvaskyla by train. Route length of 200kms in 4.5 hours. Price : 31.4 Euros.*

The total length of this trip is 605kms. It takes total of 11.7 hours to complete.
The total price of this trip is 95.8 Euros.

… Tästä on karsittu 12 ehdotusta …

This solution is a trip from tampere to tampere and it consists 6 routes, which are:
- *tampere to jyvaskyla by train. Route length of 140kms in 1.5 hours. Price : 19.4 Euros.*
- *jyvaskyla to keuruu by train. Route length of 85kms in 1 hours. Price : 13 Euros.*
- *keuruu to jyvaskyla by train. Route length of 85kms in 1 hours. Price : 13 Euros.*
- *jyvaskyla to tampere by train. Route length of 140kms in 1.5 hours. Price : 19.4 Euros.*
- *tampere to haapamaki by train. Route length of 100kms in 1.5 hours. Price : 15 Euros.*
- *haapamaki to tampere by train. Route length of 100kms in 1.5 hours. Price : 15 Euros.*

The total length of this trip is 650kms. It takes total of 8.0 hours to complete.
The total price of this trip is 94.8 Euros.

This solution is a trip from tampere to helsinki and it consists 5 routes, which are:
- *tampere to haapamaki by train. Route length of 100kms in 1.5 hours. Price : 15 Euros.*
- *haapamaki to tampere by train. Route length of 100kms in 1.5 hours. Price : 15 Euros.*
- *tampere to jyvaskyla by train. Route length of 140kms in 1.5 hours. Price : 19.4 Euros.*
- *jyvaskyla to tampere by train. Route length of 140kms in 1.5 hours. Price : 19.4 Euros.*
- *tampere to helsinki by train. Route length of 130kms in 1.8 hours. Price : 24.9 Euros.*

The total length of this trip is 610kms. It takes total of 7.8 hours to complete.
The total price of this trip is 93.7 Euros.

This solution is a trip from tampere to jyvaskyla and it consists 5 routes, which are:
- *tampere to helsinki by train. Route length of 130kms in 1.8 hours. Price : 24.9 Euros.*
- *helsinki to tampere by train. Route length of 130kms in 1.8 hours. Price : 24.9 Euros.*
- *tampere to haapamaki by train. Route length of 100kms in 1.5 hours. Price : 15 Euros.*
- *haapamaki to tampere by train. Route length of 100kms in 1.5 hours. Price : 15 Euros.*
- *tampere to jyvaskyla by train. Route length of 140kms in 1.5 hours. Price : 19.4 Euros.*

The total length of this trip is 600kms. It takes total of 8.1 hours to complete.
The total price of this trip is 99.2 Euros.

This solution is a trip from tampere to jyvaskyla and it consists 5 routes, which are:
- *tampere to helsinki by train. Route length of 130kms in 1.8 hours. Price : 24.9 Euros.*
- *helsinki to tampere by train. Route length of 130kms in 1.8 hours. Price : 24.9 Euros.*
- *tampere to jyvaskyla by train. Route length of 140kms in 1.5 hours. Price : 19.4 Euros.*
- *jyvaskyla to keuruu by train. Route length of 85kms in 1 hours. Price : 13 Euros.*
- *keuruu to jyvaskyla by train. Route length of 85kms in 1 hours. Price : 13 Euros.*

The total length of this trip is 570kms. It takes total of 7.1 hours to complete.
The total price of this trip is 95.2 Euros.

Testitapaus 3

Kuvaus:

Käyttäjällä on kiinnostunut saamaan selville millaisia matkoja eri puolilla maailmaa voisi toteuttaa viidelläkymmenellä eurolla. Siksi hän käynnistää RoutePlannerin ja syöttää ohjelmiston kysyessä hakuehdoiksi ainoastaan maksimihinnan 50e. Kaikkiin muihin kysymyksiin hän vastaa ”any” eli jättää ne rajoitteet avonaisiksi. Käyttäjä ei tahdo Turkuun, Helsinkiin, Kuopioon eikä Seinäjoelle. Hän myös pelkää lentämistä, tulee helposti merisairaaksi ja inhoaa linja-autoja, joten hän jättää myös laivan, bussin lentokoneen pois laskuista.

RoutePlanner Source Information:

User-Given Criteria are:
** Starting point of trip is [---] .*
** This trip will end to [---] .*
** Trip has to include following cities: [---] .*
** Trip can not include following cities: [turku, kuopio, helsinki, seinajoki].*
** Vehicle(s) [ship, plane, bus] can not be used.*
** Vehicle(s) [---] must be used at least once.*
** This trip may include maximum of [---] cities.*
** This trip has to include minimum of [---] cities.*
** This trip has to cost less than 50 Euros.*
** This trip has to cost more than [---] Euros.*
** This trip may take maximum of [---] hours to complete.*
** This trip has to take more than [---] hours.*
** This trip has to be shorter than [---] kms.*
** This trip has to be more than [---] kms long.*
The symbol [---] means that you did not lock any restrictions to that matter.

Result:

I found more than one solution (66) and these solutions are:
This solution is a trip from jyvaskyla to joensuu and it consists 1 routes, which are:
~ jyvaskyla to joensuu by train. Route length of 200kms in 4.5 hours. Price : 31.4 Euros.
The total length of this trip is 200kms. It takes total of 4.5 hours to complete.
The total price of this trip is 31.4 Euros.

This solution is a trip from tampere to keuruu and it consists 2 routes, which are:
~ tampere to haapamaki by train. Route length of 100kms in 1.5 hours. Price : 15 Euros.
~ haapamaki to keuruu by train. Route length of 20kms in 0.2 hours. Price : 5 Euros.
The total length of this trip is 120kms. It takes total of 1.7 hours to complete.
The total price of this trip is 20 Euros.

... Tästä on karsittu 61 ehdotusta …

This solution is a trip from tampere to tampere and it consists 4 routes, which are:
~ tampere to haapamaki by train. Route length of 100kms in 1.5 hours. Price : 15 Euros.
~ haapamaki to keuruu by train. Route length of 20kms in 0.2 hours. Price : 5 Euros.
~ keuruu to haapamaki by train. Route length of 20kms in 0.2 hours. Price : 5 Euros.
~ haapamaki to tampere by train. Route length of 100kms in 1.5 hours. Price : 15 Euros.
The total length of this trip is 240kms. It takes total of 3.4 hours to complete.
The total price of this trip is 40 Euros.

This solution is a trip from joensuu to haapamaki and it consists 3 routes, which are:
~ joensuu to jyvaskyla by train. Route length of 200kms in 4.5 hours. Price : 31.4 Euros.
~ jyvaskyla to keuruu by train. Route length of 85kms in 1 hours. Price : 13 Euros.
~ keuruu to haapamaki by train. Route length of 20kms in 0.2 hours. Price : 5 Euros.
The total length of this trip is 305kms. It takes total of 5.7 hours to complete.
The total price of this trip is 49.4 Euros.

This solution is a trip from tampere to jyvaskyla and it consists 1 routes, which are:
~ tampere to jyvaskyla by train. Route length of 140kms in 1.5 hours. Price : 19.4 Euros.
The total length of this trip is 140kms. It takes total of 1.5 hours to complete.
The total price of this trip is 19.4 Euros.

Testitapaus 4

Kuvaus:

Käyttäjän poika on kateissa ja on tiedossa että tämä oli nähty viimeksi Jyväskylässä 4 tuntia aiemmin ja rahaa pojalla oli ollut 40 euroa. Käyttäjä kysyy RoutePlannerilta mahdollisia reittejä, joita poika voisi 40 eurollaan ostaa Jyväskylästä. RoutePlannerille siis annetaan lähtötiedoiksi maksimihinta 40 euroa ja lähtöpaikka Jyväskylä. Pojan tiedetään vihaavan Turkua, Helsinkiä ja Seinäjokea. Siksi nämä paikat jätetään haun ulkopuolelle. Pojalla ei myöskään ole passia ja oletetaan, ettei hän siis voi käyttää lentokonetta tai laivaa. Hän tulee pahoinvoivaksi bussissa, joten sekin kulkukeino voidaan karsia.

RoutePlanner Source Information:

User-Given Criteria are:
** Starting point of trip is jyvaskyla.*
** This trip will end to [---] .*
** Trip has to include following cities: [---] .*
** Trip can not include following cities: [turku, helsinki, seinajoki].*
** Vehicle(s) [plane, ship, bus] can not be used.*
** Vehicle(s) [---] must be used at least once.*
** This trip may include maximum of [---] cities.*
** This trip has to include minimum of [---] cities.*
** This trip has to cost less than 40 Euros.*
** This trip has to cost more than [---] Euros.*
** This trip may take maximum of 4 hours to complete.*
** This trip has to take more than [---] hours.*
** This trip has to be shorter than [---] kms.*
** This trip has to be more than [---] kms long.*
The symbol [---] means that you did not lock any restrictions to that matter.

Result:

I found more than one solution (11) and these solutions are:
This solution is a trip from jyvaskyla to kuopio and it consists 1 routes, which are:
~ jyvaskyla to kuopio by train. Route length of 100kms in 1.6 hours. Price : 27.2 Euros.
The total length of this trip is 100kms. It takes total of 1.6 hours to complete.
The total price of this trip is 27.2 Euros.

This solution is a trip from jyvaskyla to keuruu and it consists 1 routes, which are:
~ jyvaskyla to keuruu by train. Route length of 85kms in 1 hours. Price : 13 Euros.
The total length of this trip is 85kms. It takes total of 1 hours to complete.
The total price of this trip is 13 Euros.

… Tästä on karsittu 7 ehdotusta …

This solution is a trip from jyvaskyla to keuruu and it consists 3 routes, which are:
~ jyvaskyla to tampere by train. Route length of 140kms in 1.5 hours. Price : 19.4 Euros.
~ tampere to haapamaki by train. Route length of 100kms in 1.5 hours. Price : 15 Euros.
~ haapamaki to keuruu by train. Route length of 20kms in 0.2 hours. Price : 5 Euros.
The total length of this trip is 260kms. It takes total of 3.2 hours to complete.
The total price of this trip is 39.4 Euros.

This solution is a trip from jyvaskyla to jyvaskyla and it consists 2 routes, which are:
~ jyvaskyla to tampere by train. Route length of 140kms in 1.5 hours. Price : 19.4 Euros.
~ tampere to jyvaskyla by train. Route length of 140kms in 1.5 hours. Price : 19.4 Euros.

The total length of this trip is 280kms. It takes total of 3.0 hours to complete.
The total price of this trip is 38.8 Euros.

Testitapaus 5

Kuvaus:

Käyttäjä tahtoo pian pois Turusta ja haluaa selvittää, kuinka pitkälle hän pääsee 4:ssä tunnissa 60:llä eurolla kun on vielä merisairauteen taipuvainen eikä voi siis lähteä laivalla. Käyttäjä antaa RoutePlannerille lähtöpaikkakriteeriksi Turun ja maksimihinnaksi 60 euroa. Ajan suhteen hän päättää että matka saa kestää siis maksimissaan 4 tuntia mutta sen on kestettävä vähintään 2 tuntia, jotenka ohjelmalle maksimikestoksi 4 tuntia ja minimikestoksi 2 tuntia. Ei mahdolliseksi kulkuneuvoksi käyttäjä syöttää vielä laivan. Käyttäjä ei tahdo joutua kulkemaan Kuopion, Seinäjoen, Haapamäen eikä Keuruun kautta.

RoutePlanner Source Information:

User-Given Criterias are:
** Starting point of trip is turku.*
** This trip will end to [---] .*
** Trip has to include following cities: [---] .*
** Trip can not include following cities: [keuruu, haapamaki, seinajoki, kuopio].*
** Vehicle(s) [ship] can not be used.*
** Vehicle(s) [---] must be used at least once.*
** This trip may include maximum of [---] cities.*
** This trip has to include minimum of [---] cities.*
** This trip has to cost less than 60 Euros.*
** This trip has to cost more than [---] Euros.*
** This trip may take maximum of 4 hours to complete.*
** This trip has to take more than 2 hours.*
** This trip has to be shorter than [---] kms.*
** This trip has to be more than [---] kms long.*
The symbol [---] means that you did not lock any restrictions to that matter.

Result:

I found more than one solution(7) and these solutions are:
This solution is a trip from turku to turku and it consists 2 routes, which are:
~ turku to tampere by train. Route length of 120kms in 1.7 hours. Price : 22.7 Euros.
~ tampere to turku by train. Route length of 120kms in 1.7 hours. Price : 22.7 Euros.
The total length of this trip is 240kms. It takes total of 3.4 hours to complete.
The total price of this trip is 45.4 Euros.

… Tästä on karsittu 3 ehdotusta …

This solution is a trip from turku to helsinki and it consists 1 routes, which are:
~ turku to helsinki by train. Route length of 120kms in 2 hours. Price : 24.9 Euros.
The total length of this trip is 120kms. It takes total of 2 hours to complete.
The total price of this trip is 24.9 Euros.

This solution is a trip from turku to tampere and it consists 2 routes, which are:
~ turku to helsinki by train. Route length of 120kms in 2 hours. Price : 24.9 Euros.
~ helsinki to tampere by train. Route length of 130kms in 1.8 hours. Price : 24.9 Euros.
The total length of this trip is 250kms. It takes total of 3.8 hours to complete.
The total price of this trip is 49.8 Euros.

This solution is a trip from turku to turku and it consists 2 routes, which are:
~ turku to helsinki by train. Route length of 120kms in 2 hours. Price : 24.9 Euros.
~ helsinki to turku by train. Route length of 120kms in 2 hours. Price : 24.9 Euros.
The total length of this trip is 240kms. It takes total of 4 hours to complete.
The total price of this trip is 49.8 Euros.