Ohjelmistoarkkitehtuurin monitavoiteoptimointi

Tommi Perkola

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Erkki Mäkinen
Toukokuu 2014

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Tommi Perkola: Ohjelmistoarkkitehtuurin monitavoiteoptimointi
Pro gradu -tutkielma, 62 sivua
Toukokuu 2014

Tiivistelmä

Tutkin monitavoiteoptimointia ohjelmistoarkkitehtuurien suunnittelussa. Tutkimuskohteenani on arkkitehtuurin suunnitteluun laadittu järjestelmä nimeltään Frankenstein. Frankenstein optimoi arkkitehtuuria erilaisin heuristiikoin, tässä tutkimuksessa geneettisillä algoritmeilla.

Esittelen myös laajemmin monitavoiteoptimointia ja sen teoriaa. Aluksi yleisesti heuristisia menetelmiä, yksi- ja monitilaisia menetelmiä ja luvussa 3 erilaisia monitavoiteoptimoinnin menetelmiä.

Tutkimuksen tuloksena sain, että Frankenstein täyttää annetut suppeat kriteerit monitavoiteoptimoinnissaan, mutta varsinaisen monitavoiteoptimoinnin teorian kannalta olisi myöhemmin kiinnostava selvittää, miten järjestelmä selviää jostain uudesta arkkitehtuurista vai onko se vain sovittanut toimintansa kahteen käytettyyn esimerkkiarkkitehtuuriin.

Sisällys

1 Johdanto . . . 1
2 Heuristiset menetelmät . . . 4
  2.1 Ilmaisen lounaan teoreema . . . 5
  2.2 Yksitilaiset etsintämenetelmät . . . 6
  2.3 Populaatiopohjaiset menetelmät . . . 12
3 Monitavoiteoptimointi . . . 20
  3.1 Monitavoiteoptimoinnin ongelmia . . . 21
  3.2 Ideaalivektori ja Pareto-optimi . . . 22
  3.3 Aggregoivat menetelmät . . . 26
  3.4 Pareto-pohjaiset menetelmät . . . 28
  3.5 Muut menetelmät . . . 32
  3.6 Pareto-rintamien vertailua . . . 34
4 Ohjelmistoarkkitehtuurit . . . 36
  4.1 Komponentit ja rajapinnat . . . 38
  4.2 Suunnittelumallit . . . 38
  4.3 Arkkitehtuurityylit . . . 40
  4.4 Arkkitehtuurien arviointi . . . 42
5 Frankenstein-ohjelma . . . 45
  5.1 Mutaatio ja risteytys . . . 46
  5.2 Fitnessin mittaaminen . . . 52
  5.3 Muunnelmia ja laajennuksia . . . 53
  5.4 Skenaariot . . . 55
  5.5 Monitavoiteoptimointi Frankenstein-järjestelmässä . . . 56
6 Johtopäätökset . . . 59
Viiteluettelo . . . 61

1 Johdanto

Ohjelmistoarkkitehtuuri kuvaa tietokoneohjelmiston korkeimman tason rakennetta. Arkkitehtuurin suhde varsinaiseen ohjelmistoon on sama kuin rakennuksen arkkitehdin tuottamien piirustusten suhde valmiiseen taloon. Arkkitehti ei sinänsä tuota mitään valmista, mutta asioista tietävä voi nähdä jo piirustuksista, onko jokin asia sitä mitä haluttiin, toimiiko se käytännössä tai miltä tuleva talo näyttäisi.

Tietokoneohjelmistojen monimutkaisuus tekee niistä hyvin vaikeasti rakennettavia ja ylläpidettäviä. Arkkitehtuuri antaa ohjelmistolle rakenteen, kartan jolla ohjelmistoa voi hahmottaa ja suuntaviivat, joiden mukaan sitä voi kehittää edelleen. Samalla tavalla kuin rakennustekniikassa, arkkitehtuurista voi jo ennen ohjelmiston toteuttamista päätellä sen ominaisuuksia, vahvuuksia ja heikkouksia, ja asioita voi kehittää ja korjata jo ennen kuin varsinaiseen ohjelmointityöhön ryhdytään. [Koskimies & Mikkonen, 2005]

Ohjelmistojen arkkitehtuurien suunnittelu on perinteisesti vaatinut erityisasiantuntemusta ja pitkää kokemusta. On kuitenkin myös metriikoita ja heuristiikkoja, joilla arkkitehtuureja voidaan tutkia ja arvottaa. Tällaisia ovat rakennusten arkkitehtuurista innoituksensa saaneet *suunnittelumallit*, jotka ovat käytännössä toimiviksi osoittautuneita korkean tason patenttiratkaisuja. Suunnittelumallien ohella kirjallisuudessa on esitetty myös *antimalleja* eli vastaavasti huonoksi osoittautuneita ratkaisuja, jotka hyvin todennäköisesti tuottavat vaikeasti hallittavia virheitä ja jotka kannattaa tunnistaa, että niitä voidaan välttää. Lisäksi on *metriikoita* eli tapoja, joilla kaaviona ilmaistusta arkkitehtuurista voidaan mitata ja löytää ominaisuuksia. Lisäksi *laadullisessa arvioinnissa* esitetään mahdollisia käyttötapauksia eli skenaarioita ja tutkitaan, miten arkkitehtuuri vastaisi näiden esittämiin tarpeisiin.

Herää ajatus, olisiko arkkitehtuuri järjestelmä, jota voitaisiin optimoida tietokoneella tekoälyn keinoin kuten muitakin monimutkaisia järjestelmiä. Tietokoneohjelmat menestyvät usein ihmistä paremmin ongelmissa, jotka ovat liian monimutkaisia ihmiselle, mutta jotka voidaan abstrahoida muotoon, jossa tietokone voi käyttää muistiaan ja suoritusnopeuttaan hyväksi.

Frankenstein on Java-kielellä toteutettu järjestelmä, joka optimoi annettua perusarkkitehtuuria. Se saa syötteekseen arkkitehtuurin, joka toteuttaa tarvitut vaatimukset ja muokkaa ja kehittää sitä tiettyjen annettujen kriteerien mukaan.

Arkkitehtuureissa kuten monissa muissakin käytännön ongelmissa on useita yhtäaikaisia vaatimuksia, ja nämä vaatimukset ovat usein ristiriidassa keskenään. Yleensä ei ole yhtä universaalia optimiratkaisua, vaan saadaan mahdollisesti suurikin joukko vastauksia, joissa yksi asia joudutaan uhraamaan toisen edestä ja päinvastoin.

Tällaista useiden yhtäaikaisten ja keskenään ristiriitaisten tavoitteiden optimointia kutsutaan monitavoiteoptimoinniksi. Monitavoiteoptimointi ei yleensä tuota yhtä ratkaisua, vaan optimi on joukko vastauksia, joille pätee, etteivät ne ole yksiselitteisesti huonompia kuin jokin toinen olemassaoleva vastaus. Tätä optimia kutsutaan *Pareto-optimiksi.* [Coello, 2009]

Monitavoiteoptimoinnin ongelmat ovat usein laskennallisesti raskaita. Niissä voi vain harvoin käyttää niin sanottua raa'an voiman menetelmää, jossa käydään läpi kaikki mahdolliset vastaukset ja valitaan niistä parhaat. Normaalisti on aikaa ja resursseja tutkia vain olemattoman pientä osaa *ongelma-avaruudeksi* kutsutusta mahdollisten vastausten joukosta.

Menetelmiä, joissa haetaan hyvää ratkaisua käymällä läpi vain osa tapauksista, kutsutaan *heuristisiksi menetelmiksi.* Heuristisissa menetelmissä käytetään hyväksi satunnaisuutta ja havaintoja ongelma-avaruudesta. Yksinkertaisimpia heuristisia menetelmiä ovat satunnainen haku ja paikallinen haku, jossa kuljetaan jonkin ratkaisun lähimaastossa kohti parhaalta näyttävää suuntaa ja saavutetaan ainakin *paikallinen optimi,* joka saattaa tietenkin olla paljon huonompi kuin paras mahdollinen tulos eli niin sanottu *globaali optimi.*

Heuristisia menetelmiä on useita ja jokaisesta niistä erilaisia versioita ja muunnelmia. Niin sanottu *ilmaisen lounaan teoreema* todistaa, ettei ole olemassa universaalia toimivaa heuristiikkaa. Käytännössä heuristiset menetelmät toimivat usein varsin tehokkaastikin. Eri ongelma-alueet poikkeavat toisistaan ja siksi vaikka optimointimenetelmiä tutkitaan ja kehitetään, niiden suorituskyky saattaa vaihdella ratkaisevasti erilaisissa ongelmissa.

Kehittyneistä heuristisista menetelmistä esitän *simuloidun jäähdytyksen* ja *geneettiset algoritmit.* Mainitsen lyhyesti myös muita optimointimenetelmiä. Näistä on paljon erilaisia versioita jo tavallisessa yhden tavoitearvon optimoinnissa eli skalaarioptimoinnissa, ja monitavoiteoptimointi antaa näille vielä lisähaasteita. Monitavoiteoptimointiin kehitetyt versiot simuloidusta jäähdytyksestä ja geneettisistä algoritmeista ovat astetta monimutkaisempia ja monimuotoisempia kuin skalaarioptimointiin kehitetyt versiot.

Esitän luvussa 2 yleisesti teoriaa ja periaatteita heuristisista menetelmistä. Seuraavassa luvussa käyn läpi monitavoiteoptimointia, sen ongelmia ja erilaisia menetelmiä, jolla monitavoiteoptimointia on yritetty ratkaista.

Luvussa 4 kuvaan ohjelmistoarkkitehtuureita, niiden periaatteita ja tapoja mitata arkkitehtuurien kelpoisuutta. Luvussa 5 kuvaan Frankenstein-järjestelmän, sen toimintaperiaatteen ja toteutustavan. Viimeisessä luvussa eli luvussa 6 tutkin Frankenstein-järjestelmää monitavoiteoptimoinnin näkökulmasta.

Monitavoiteoptimointi on vaikeudestaan huolimatta nouseva ala, koska sen sovelluksia on paljon ja automaation ja tekoälysovellusten lisääntymisen takia sen tarve kasvaa voimakkaasti. Useimmat tekniikat ovat hyvin uusia ja ne muuttuvat jatkuvasti. Tämän vuoksi hyvin harvalle käsitteistä ja tekniikoista on vakiintunut suomenkielinen käännös. Olen pyrkinyt kääntämään nimiä mahdollisimman paljon, mutta käytännössä tietoa kannattaa hakea aina englanninkielisin termein.

2 Heuristiset menetelmät

Optimoinnissa ongelmat hahmotetaan funktioiksi, jotka tuottavat jollain syötearvolla x palautuvan arvon y. Tutuin esimerkki tällaisesta tilanteesta on yhden muuttujan funktion kuvaaja, jossa x-akselille kuvatut syötearvot saavat tulosarvot y-akselilla. Optimoitaessa haetaan tällaisesta kuviosta suurinta tai pienintä arvoa.

Funktio voi saada myös useampia arvoja, jolloin siihen syötetään vektori $\mathbf{x}$. Kaksi arvoa voidaan kuvata x ja y-akseleille ja nyt tulosarvo kuvautuisi kolmannelle eli z-akselille. Tällainen esitys kuvaa jotain *ongelma-avaruutta* ja kuvaajaa kutsutaan joskus *maisemaksi* (problem landscape) koska yhden syötettävän muuttujan tapauksessa kuvaaja voidaan ajatella taivaanrannaksi ja kahden muuttujan tapauksessa pinnaksi eli *maastoksi*, jossa x- ja y-akselit osoittavat kartan koordinaatteja ja y pinnan korkeutta tietyssä x:n ja y:n määrittämässä pisteessä.

Yleisemmin puhutaan *ongelma-avaruudesta.* Jos syötettäviä muuttujia on enemmän kuin kaksi, kuvaajasta tulisi vähintään neliulotteinen ja siksi vaikea visualisoida.

Saatu tulos on näissä tapauksissa paljas yksiulotteinen luku eli *skalaari,* mutta tulos voi olla myös korkeampiulotteinen vektori. Tällöin puhutaan monitavoiteoptimoinnista, jota kutsutaan myös multiobjektiiviseksi optimoinniksi tai vektorioptimoinniksi.

Kun haetaan parasta arvoa jostain tällaisesta ongelma-avaruudesta, paras arvo voi olla skalaarioptimoinnin tapauksessa maksimi tai minimi. Monitavoiteoptimoinnissa asia on hieman monimutkaisempi, ja sitä käsitetään myöhemmin lisää.

Yleensä optimista puhutaan miniminä koska maksimointitehtävän voi muuttaa minimointitehtäväksi ottamalla optimoitavan funktion tulokseksi tehtävän tuloksen vastaluvun tai käänteisluvun.

Kaikkien vaihtoehtojen tutkimista ja niistä parhaan valitsemista kutsutaan *raa'an voiman menetelmäksi.* Usein vaihtoehtoja on liikaa, jotta niitä voitaisiin käydä kaikkia läpi. Useissa optimointitilanteissa kohdataan niin sanottu *eksponentiaalinen räjähdys,* jossa tapausten lukumäärä kasvaa jollakin kertoimella joka kerran, kun ongelma-alue kasvaa uudella komponentilla.

Heuristisissa menetelmissä etsitään hyvää ratkaisua, joka saavutetaan käymällä läpi vain pieni osa kaikista mahdollisuuksista. Ratkaisu ei tietenkään ole välttämättä paras mahdollinen.

2.1 Ilmaisen lounaan teoreema

Niin sanotun ilmaisen lounaan teoreeman *(No free lunch)* mukaan jokainen heuristiikka, joka ei hukkaa resursseja lukemalla samaa dataa useampaan kertaan, on tarkalleen yhtä tehokas. Ei siis ole olemassa parempia eikä huonompia heuristiikkoja vaan jokainen toimii yhtä hyvin kuin satunnainen valinta. Käytännön sovelluksissa toiset heuristiikat toimivat kuitenkin merkittävästi paremmin toiset, sillä kohdealueet, joihin niitä sovelletaan, eivät ole täysin satunnaisia.

Ilmainen lounas -teoreeman todistus on äärellisessä tapauksessa melko yksinkertainen. Oletetaan, että on olemassa M erilaista strategiaa, joilla voidaan hakea parasta ratkaisua ja ratkaisu voi olla mikä tahansa N:stä erilaisesta arvosta. Kaikki mahdolliset ongelma-avaruudet voi kuvata $M \times M^N$ -matriisilla, jossa rivit kuvaavat käytettyä strategiaa ja kukin sarake yhtä mahdollista maailmaa, jossa kukin strategia saa jonkin tulosarvon. Tarkemmin sarakkeet edustavat kaikkia mahdollisia M-ulotteisia vektoreita, joiden jäsenten arvot kuuluvat johonkin sallitusta N:stä arvosta.

Tällainen matriisi voidaan järjestää niinsanotuksi *desimaalimatriisiksi*, joka kuvaa kaikki M-pituiset N-kantaiset luvut. Jos sallitut arvot ovat 0 ja 1 ja luvun pituus 4, ensimmäinen sarake olisi $[0, 0, 0, 0]^T$, toinen $[0, 0, 0, 1]^T$, kolmas $[0, 0, 1, 0]^T$, toiseksi viimeinen $[1, 1, 1, 0]^T$ ja viimeinen $[1, 1, 1, 1]^T$. On helppo huomata, että desimaalimatriisin jokaisella rivillä on tarkalleen sama määrä ykkösiä ja nollia. Yleisesti voidaan todeta, että vaikka kantaluku olisi kakkosta suurempi, jokaisen rivin lukujen summa on sama. Näin jokainen strategia tuottaa keskimäärin tarkalleen saman tuloksen, kun otetaan huomioon kaikki mahdolliset ongelma-avaruudet. [Ho & Pepyne, 2001]

Teoreemasta kannattaa huomata, että useimmissa tapauksissa N^M on hyvin suuri luku ja valtavassa enemmistössä sarakkeiden kuvaamia maailmoja ei ole minkäänlaista säännönmukaisuutta, jolla niitä voitaisiin hahmottaa. Ilmaisen lounaan teoreema osoittaa, että periaatteessa optimointi on mahdotonta, mutta käytännön maailmassa se näyttäisi olevan melko usein mahdollista.

2.2 Yksitilaiset etsintämenetelmät

Yksinkertaisimmissa heuristisissa menetelmissä eli yksitilaisissa etsintämenetelmissä (single-state search methods) otetaan yksi ratkaisuehdokas, testataan miten se toimii ja joko hyväksytään se tai vaihdetaan se toiseen tai muokataan sitä jollain tavalla.

2.2.1 Satunnaishaku

Yksinkertaisin heuristinen hakumenetelmä on niin sanottu satunnaishaku. Kun raa'an voiman menetelmässä pitäisi hakea ja testata jokainen mahdollinen ratkaisu, satunnaishaussa poimitaan ratkaisuja satunnaisesti ja valitaan niistä paras.

Ajatuksena on, että kun otos on riittävän suuri ja kun erilaiset ratkaisut poimitaan riittävän satunnaisesti ja tasaisesti pitkin ongelma-avaruutta, tulee aina vain epätodennäköisemmäksi, että valittuiksi ratkaisuiksi osuisi vain hyvin huonoja ratkaisuja.

Satunnaishaun pseudokoodi on esitetty algoritmissa 1.

```
aika ← aikaraja ;
Paras ← satunnainen ;
while aika > 0 do
    R ← satunnainen ;
    if fitness(R) > fitness(Paras) then
        Paras ← R ;
    end
    aika ← aika − 1 ;
end
return Paras ;
```

Algoritmi 1: Satunnaishaku

Algoritmissa 1. valitaan aikaraja *aika* ja toistetaan silmukkaa kunnes aika on kulunut loppuun. Joka kierroksella tuotetaan uusi satunnainen ehdokas R, kokeillaan onko se parempi kuin sen hetkinen paras ratkaisu S, ja jos on, korvataan S uudella ratkaisulla. [Luke, 2009, 20]

Satunnaishaku on eri asia kuin *satunnaiskulku* (random walk.) Satunnaishaussa voidaan ottaa mikä tahansa ongelma-avaruuden alkio, kun taas satunnaiskulku

hakee seuraavan jonkin tietyn askeleen päästä. Satunnaiskulussa siis seuraava askel jää jonnekin nykyisen pisteen lähialueelle. [Luke, 2009, 19]

2.2.2 Paikallinen haku

Satunnaishaku toimii tietenkin huonosti, jos hyvät ratkaisut ovat harvinaisia huonoihin verrattuna. Lisäksi se on siinä mielessä tyhmä menetelmä, ettei se huomaa, vaikka sen löytämän ratkaisun lähellä olisi vielä paljon parempia ratkaisuja.

Tätä ongelmaa yrittää korjata **paikallinen haku** (local search) eli lokaali haku. Se käyttää tietoa siitä, millaiset ovat jotain ratkaisua lähellä olevat ratkaisut, mutta sen ei tarvitse tutkia muita ratkaisuja. Koko ongelma-avaruuden tutkimisen sijasta se voi tutkia tehokkaasti pientä avaruuden osaa.

Paikallinen haku voi olla **gradienttihaku,** jossa siirrytään parhaaseen naapuruston ratkaisuun, jos se on parempi kuin nykyinen ratkaisu. Hakua jatketaan edelleen uudesta pisteestä, kunnes löydetään paikallinen optimi. Yksinkertaisimmillaan optimi voi tarkoittaa sitä, ettei pisteen läheltä löydy enää parempia ratkaisuja. [Luke, 2009, 13]

Aina ei ole mahdollista käydä edes kaikkia lähiympäristön vaihtoehtoja läpi. **Vuorikiipeilijä** on algoritmi, joka vertaa nykyistä ratkaisua johonkin lähiympäristön ratkaisuun ja vaihtaa paikkaansa, jos uusi ratkaisu on nykyistä parempi. Gradienttihaun ja puhtaan vuorikiipeilijän välillä on **jyrkimmän suunnan vuorikiipeilijä,** joka hakee otoksen lähiseudun pisteistä ja valitsee niistä parhaan. [Luke, 2009, 17-18]

Yleinen paikallishaun algoritmi on esitetty algoritmissa 2.

$S \leftarrow$ satunnainen ;
while *S ei ole paikallinen maksimi* **do**
  $R \leftarrow Muokkaa(S)$;
  if *fitness(R) > fitness(S)* **then**
    $S \leftarrow R$;
  end
end
return S ;

Algoritmi 2: Paikallishaku

Erilaiset paikallishaut poikkeavat toisistaan sen mukaan, miten *Muokkaa*-aliohjelma toimii niissä. Gradienttihaussa *Muokkaa* palauttaa pisteen S naapuruston parhaan ratkaisun, ja jos se ei ole parempi kuin S, algoritmi lopettaa koska sen ympäristössä ei ole enää parempia ratkaisuja. Vuorikiipelijäalgoritmit voivat jatkaa edelleen pisteessä S, kunnes esimerkiksi annettu aikaraja on täyttynyt. Ne eivät tunne kaikkia ympäristönsä ratkaisuja ja saattavat löytää nykyistä paremman ratkaisun ympäristöstään uudelleen kokeilemalla.

Paikallisen haun algoritmit toimivat hyvin silloin, kun ongelma-avaruuden kuvaama pinta on jokseenkin tasainen eli siinä on verraten vähän huippuja ja lähellä toisiaan sijaitsevien pisteiden tulosarvot ovat myös melko lähellä toisiaan. Mitä vaihtelevampi ja ennakoimattomampi avaruus on, sitä paremmin satunnaishaku toimii paikallisiin hakuihin verrattuna. Täysin satunnaisessa avaruudessa satunnaishaku antaa paremman tuloksen. [Luke, 2009, 21-22]

Paikallisen haun menetelmien pahin puute on se, että vaikka ne liikkuvat kohti optimia, ne voivat jäädä kiinni paikalliseen optimiin, joka on huonompi kuin jokin toinen optimi.

Topologiassa aluetta sanotaan *konveksiksi* jos jokaiselle kahdelle alueeseen kuuluvalle pisteelle pätee, että jokainen näiden kahden pisteen väliselle suoralle kuuluva piste kuuluu myös alueeseen. Arkisen havainnollisesti voidaan sanoa, että alue tai kappale on konveksi jos siinä ei ole kuoppia eikä reikiä. Optimoinnissa konveksius on tärkeä ominaisuus siksi, että jos tulosavaruus on konveksi, siinä ei ole paikallisia optimeja. Useimmat optimointimenetelmät toimivat paremmin konveksissa kuin ei-konveksissa avaruudessa.

Jos avaruus on konveksi eli sillä ei ole paikallisia optimeja, paikallinen haku tuottaa globaalisti parhaan tuloksen, jos voidaan estää sen jääminen kiinni *satulapisteisiin* eli kohtiin, joissa maasto ei nouse eikä laske, vaikka se ei ole maksimi eikä minimi. [Luke, 2009, 133-134]

Satunnaishaun ja paikallisen haun yhdistelmä on **paikallinen haku satunnaisin alkupistein.** Siinä aloitetaan haku jostain satunnaisesta kohdasta, noustaan paikalliseen optimiin, pannaan tulos muistiin ja aloitetaan uudestaan jostain toisesta satunnaisesta paikasta. Tätä jatketaan kunnes ollaan saatu riittävän hyvä tulos. Hakua voi myös viedä enemmän kohti satunnaishakua niin, että annetaan jokin maksimiaika paikalliselle haulle jonka jälkeen nousu katkaistaan ja paikallinen haku alkaa uudestaan jostain uudesta paikasta. [Luke, 2009, 21]

2.2.3 Simuloitu jäähdytys

Simuloitu jäähdytys *(simulated annealing)* on saanut ideansa teräksen mellotuksesta. Jäähdytys on sikäli harhaanjohtava nimitys, että mellotuksessa terästä kuumennetaan, jotta se jäähtyisi riittävän hitaasti. Hitaassa jäähtymisessä metallin atomeilla on aikaa hakeutua optimaaliseen hilarakenteeseen. Atomien optimaaliset asemat ovat sellaisia, missä ne ovat paikallaan lujasti. Kuumassa tilassa voimakas lämpöliike irrottaa löysästi paikalleen kiinnittyneet atomit. Tämä estää atomien kiinnittymisen epäoptimaalisiin asemiin. Lämpöä lasketaan hallitun hitaasti, että jokainen atomi ehtisi kiinnittyä optimaaliseen paikkaan.

Jossain mielessä voidaan sanoa, että simuloitu jäähdytys on yhdistelmä satunnaishakua ja paikallista hakua. Voidaan sanoa, että se alkaa satunnaishakuna ja muuttuu algoritmin edetessä vuorikiipeilijähauksi. [Luke, 2009, 25]

Simuloidun jäähdytyksen pseudokoodi on esitetty algoritmissa 3.

$t \leftarrow$ alkulämpötila, yleensä korkea ;
$S \leftarrow$ ensimmäinen kokeiltava ratkaisu ;
$Paras \leftarrow S$;
while $t > 0$ **do**
  $R \leftarrow Muokkaa(S)$;
  $rand \leftarrow$ satunnainen luku väliltä [0,1] ;
  if $fitness(R) > fitness(S) \vee rand < e^{\frac{fitness(R)-fitness(S)}{t}}$ **then**
    $S \leftarrow R$;
  end
  $t \leftarrow t - 1$;
  if $fitness(S) > fitness(Paras)$ **then**
    $Paras \leftarrow S$;
  end
end
return *Paras ;*

Algoritmi 3: Simuloitu jäähdytys

Algoritmissa 3 asetetaan aluksi lämpötila ja ensimmäinen ratkaisuehdokas S. Aliohjelma $Muokkaa$ generoi sen hetkisestä ratkaisuehdokkaasta uuden ratkaisun R, joka valitaan uudeksi ratkaisuehdokkaaksi jos se on parempi kuin S tai jos satunnainen nollan ja ykkösen välisen suljetun välin luku on pienempi kuin kaavan $e^{\frac{fitness(R)-fitness(S)}{t}}$ luku. Kannattaa huomata, että kun lämpötila t pienenee, eksponentti kasvaa ja disjunktion oikean puolen merkitys vähenee ja S:n ja R:n keskinäisen paremmuuden merkitys valinnassa kasvaa. [Luke, 2009, 25]

Toisin kuin esimerkiksi satunnaishaussa tai paikallisessa haussa, paras saavutettu ratkaisu on sijoitettu omaan $Paras$-muuttujaansa. Näin algoritmin kulloinenkin kokeiltava arvo voi kulkea epäoptimaalisilla alueilla hakemassa uusia vaihtoehtoja. Tämä mahdollistaa ratkaisun tilapäisen huonontumisen.

2.2.4 Tabuhaku

Tabuhaussa pidetään yllä *tabulistaa* äskettäin kokeilluista ratkaisuista. Listalla olevat ratkaisut ovat nimensä mukaan koskemattomia, eli niihin ei saa palata vaan uusi ratkaisu pitää hakea muualta. Tabulista toteutetaan *jonona* eli tietorakenteena, josta poistetaan aina vuorollaan sen vanhin jäsen (first in first out).

Algoritmi 4 tallentaa l viimeksi kokeiltua ratkaisua tabulistaan L. Se hakee uutta ratkaisua muokkaamalla n kertaa käsillä olevaa ratkaisua S n kertaa ja vertaa kulloistakin saatua ratkaisua ratkaisuun S ja korvaa sen, jos uusi ratkaisu on parempi eikä kuulu tabulistaan.

Tabuhaun pseudokoodi on algoritmissa 4.

```
l ← suurin sallittu tabulistan pituus ;
n ← uuden ratkaisun muokkausten lukumäärä ;
S ← ensimmäinen kokeiltava ratkaisu ;
Paras ← S ;
L ← uusi tyhjä tabulista ;
Lisää S listaan L ;
repeat
    if L.pituus > l then
        poista L:n vanhin jäsen ;
    end
    R ← Muokkaa(S) ;
    for i = 0 ; i < n ; i++ do
        W ← Muokkaa(S) ;
        if W ∉ L ∧ (fitness(W) > fitness(R) ∨ R ∈ L) then
            R ← W ;
        end
    end
    if R ∉ L ∧ fitness(R) > fitness(S) then
        S ← R ;
        Lisää R listaan L ;
    end
until Paras on ideaaliratkaisu tai aika loppuu;
if fitness(R) > fitness(Paras) then
    Paras ← R ;
end
return Paras ;
```

Algoritmi 4: Tabuhaku

2.2.5 Iteroitu paikallishaku

Iteroitu paikallishaku on kehittyneempi versio aiemmin mainitusta paikallishausta satunnaisin alkupistein. Sitä voisi kuvata eräänlaiseksi *huipulta huipulle* -algoritmiksi. Iteroitu paikallishaku etsii paikallisia optimeja ja löydettyään sellaisen tutkii läheisiä paikallisia optimeja kunnes löytää nykyistä paremman. Algoritmi perustuu ajatukseen, että globaalia optimia on tehokkaampaa hakea nykyisen paikallisen optimin lähistöltä kuin satunnaisesti koko ongelma-avaruudesta. [Luke, 2009, 28-29]

2.3 Populaatiopohjaiset menetelmät

Populaatiopohjaisissa *(population based methods)* menetelmissä ratkaisuehdokkaita voi olla useampi kuin yksi. Näin voidaan tutkia kerralla suurempaa määrää ehdokkaita, löytää rinnakkain erilaisia mahdollisia ratkaisuja ja vertailla ratkaisujen toimivuutta toisiinsa. Raja yksitilaisten menetelmien ja populaatiopohjaisten välillä ei ole aivan yksiselitteinen. Esimerkiksi geneettisiin algoritmeihin kuuluvaa evoluutiostrategiaa käytettiin aluksi yksitilaisena menetelmänä, jossa vain yksi yksilö tuotti yhden mutaation itsestään.

Evolutionaarisissa menetelmissä jäljitellään biologista evoluutiota kilpailuttamalla ratkaisuja keskenään ja valitsemalla seuraavaan sukupolveen parhaat ratkaisut. Näistä tuotetaan lapsia mutatoimalla ja risteyttämällä. Evolutiivisissa algoritmeissa erotetaan toisistaan *genotyyppi* ja *fenotyyppi.* Fenotyyppi tarkoittaa ratkaisua sellaisena kuin sitä todellisuudessa käytetään. Ratkaisu koodataan genotyypiksi, jota voidaan mutatoida ja risteyttää algoritmisesti. Genotyypin pitää sisältää olennainen tieto ratkaisusta ja tuotetut uudet genotyypit pitää voida kääntää takaisin todellisiksi ratkaisuiksi eli fenotyypeiksi. Genotyyppiä kutsutaan usein kromosomiksi tai DNA:ksi, ja ne esitetään tyypillisesti merkkijonoina, esimerkiksi määrämittaisena ykkösten ja nollien jonona.

Erilaiset evolutiiviset algoritmit voidaan ilmaista yleisesti algoritmissa 5 esitetyllä pseudokoodilla.

$P \leftarrow$ alkupopulaatio ;
$Paras \leftarrow Null$;
repeat
  määritä P:n jäsenten kelpoisuudet ;
  for *jokaiselle* $P_i \in P$ **do**
    if $Paras = Null \vee fitness(P_i) > fitness(Paras)$ **then**
      $Paras \leftarrow P_i$;
    end
  end
  $P \leftarrow P$:n parhaiten menestyneistä yksilöistä generoitu uusi sukupolvi ;
until *Paras on ideaalinen ratkaisu tai aika loppuu*;
return $Paras$;

Algoritmi 5: Abstrakti evolutionaarinen algoritmi

Luodaan siis alkupopulaatio ja siitä uusi populaatio parhaiten menestyneistä yksilöistä. Erilaiset evolutiiviset algoritmit poikkeavat toisistaan siinä, miten suuri osa populaatiosta valitaan jatkamaan sukua ja miten seuraavan sukupolven generointi on toteutettu. [Luke, 2009, 32]

Algoritmissa 5 on valittu yksilö $Paras$ joka palautetaan lopuksi. On myös algoritmeja, joissa parasta tulosta ei panna muistiin vaan palautetaan yksi tai useampia yksilöitä viimeisestä populaatiosta. Monitavoiteoptimoinnissa tyypillisesti palautetaan useampi yksilö.

Algoritmeissa puhutaan usein *eksploitatiivisuudesta* ja *eksploratiivisuudesta.* Eksploitatiivisuus tarkoittaa sitä, että algoritmi käyttää hyväkseen jo saatua tietoa esimerkiksi tallentamalla parhaat löydetyt ratkaisut ja vertaamalla uusia ratkaisuja niihin. Eksploratiivisuus kuvaa sitä, miten laajalti algoritmi käy läpi ongelmaavaruutta.

2.3.1 Evoluutiostrategia

Vanhin geneettinen tai evoluutioon pohjautuva menetelmä on Saksassa 1960-luvulla kehitetty evoluutiostrategia *Evolutionsstrategie*. Toisin kuin myöhemmissä geneettisissä algoritmeissa, evoluutiostrategiassa ei yleensä käytetä risteytystä vaan pelkkää mutaatiota. [Luke, 2009, 33]

Yksinkertainen evoluutiostrategia on (μ, λ), jossa μ tarkoittaa niiden yksilöiden lukumäärää, jotka päästetään luomaan seuraavaa populaatiota ja λ populaation kokoa. Algoritmi 6 kuvaa evoluutiostrategian pseudokoodina.

Algoritmissa 6 siis luodaan aluksi λ:n satunnaisesti generoidun yksilön populaatio P. Siitä valitaan μ parasta yksilöä jatkamaan sukua seuraavaan sukupolveen. Jokainen näistä tuottaa λ/μ mutanttijälkeläistä, jotka liitetään uuteen populaatioon. [Luke, 2009, 33]

Evoluutiostrategiassa on kolme arvoa, jota voi muuttaa: populaation koko λ, seuraavan sukupolven tuottavan joukon koko μ, joka määrää, miten *selektiivinen* algoritmi on, sekä mutaation voimakkuus eli miten paljon mutatoituneet jälkeläiset poikkeavat vanhemmastaan.

Toinen evoluutiostrategia on $(\mu + \lambda)$. Se poikkeaa algoritmista 6 niin, että kun algoritmissa 6 seuraavaan sukupolveen pääsivät vain uudet lapset, tässä sinne otetaan mukaan myös vanhemmat. Kunkin sukupolven koko on siis tässä $\mu +$

```
μ ← vanhempien lukumäärä ;
λ ← vanhempien tuottamien lasten lukumäärä ;
P ← uusi tyhjä populaatio ;
for ind = 0; ind < λ ; ind++ do
    P ← P∪ { uusi satunnainen yksilö } ;
end
Paras ← Null ;
repeat
    for jokaiselle P_i ∈ P do
        if Paras = Null ∨ fitness(P_i) > fitness(Paras) then
            Paras ← P_i ;
        end
    end
    Q ← μ populaation P fitness-arvoltaan korkeinta yksilöä ;
    tyhjennetään P ;
    for jokaiselle Q_j ∈ Q do
        for ind = 0; ind < λ/μ; ind++ do
            P ← P ∪ {Mutatoi(Q_j)} ;
        end
    end
until Paras on optimiratkaisu tai aika loppuu;
return Paras ;
```

Algoritmi 6: Evoluutiostrategia

λ ja siinä vanhemmat kilpailevat lapsiaan vastaan. Algoritmissa $(\mu + \lambda)$ hyvin pärjänneet vanhemmat lisäävät kunkin sukupolven valintapainetta. Sen ongelma on, että hyvälaatuiset vanhemmat voivat aiheuttaa ennenaikaista konvergenssia eli algoritmi jää kiinni varhain löydettyihin ratkaisuihin hakematta uusia ja ehkä vielä parempia. [Luke, 2009, 34-35]

2.3.2 Geneettiset algoritmit

Geneettiset algoritmit kehitettiin 1970-luvulla. Yksinkertainen geneettinen algoritmi poikkeaa (λ, μ)-evoluutiostrategiasta siinä, että se tuottaa jälkeläiset risteyttämällä eikä vain mutatoimalla. Geneettisen algoritmin perusmuoto on esitetty algoritmissa 7. [Luke, 2009, 37]

```
koko ← populaation koko ;
P ← tyhjä populaatio ;
for ind = 0; ind < koko; ind++ do
    P ← P∪ uusi satunnainen yksilö ;
end
Paras ← Null ;
repeat
    for jokaiselle P_i ∈ P do
        if Paras = Null ∨ fitness(P_i) > fitness(Paras) then
            Paras ← P_i ;
        end
    end
    Q ← tyhjä populaatio ;
    for ind = 0; ind < koko/2; ind++ do
        P_a ← Valitsevanhemmaksi(P) ;
        P_b ← Valitsevanhemmaksi(P) ;
        C_a, C_b ← Risteyta(P_a, P_b) ;
        Q ← Q ∪ {Mutatoi(C_a), Mutatoi(C_b)} ;
    end
    P ← Q ;
until Paras on optimiratkaisu tai aika loppuu;
return Paras ;
```

Algoritmi 7: Geneettinen algoritmi (GA)

Geneettinen algoritmi toimii samalla tavalla kuin evoluutiostrategia seuraanvan sukupolven generointiin saakka. Siellä valitaan vanhemmat algoritmilla *Valitsevanhemmaksi* ja tuotetaan lapset uuteen sukupolveen risteyttämällä vanhemmat *Risteyta*-algoritmilla. Näillä molemmilla on useita erilaisia toteutus- ja toimintatapoja.

Yksinkertaisin tapa risteyttää kaksi yksilöä on katkaista niiden DNA jostain kohtaa. Lapsi C_a saadaan liittämällä vanhemman P_a alkupää vanhemman P_b loppupäähän ja lapsi C_b liittämällä vanhemmän P_a loppupää vanhemman P_b alkupäähän. Yksinkertainen risteytys johtaa usein ei-toivottuihin tuloksiin, ja on myös kehittyneempiä tapoja risteyttää kromosomit. Ne voidaan esimerkiksi katkaista kahdesta kohtaa. Eräs tapa on valita satunnaisesti kunkin merkin tai bitin kohdalla kummalta vanhemmalta se otetaan. [Luke, 2009, 37-39]

Mutaatio tarkoittaa kromosomin muuttamista satunnaisesti. Esimerkiksi jos kromosomi koostuu ykkösistä ja nollista, muutetaan tietyllä todennäköisyydellä ykkonen nollaksi tai nolla ykköseksi. [Luke, 2009, 37-38]

Vanhempien valintaan kolme suosituinta algoritmia ovat seuraavat: [Luke, 2009, 43-45]

- **Rulettivalinta** (fitness proportionate selection), jossa todennäköisyys tulla valituksi vanhemmaksi on verrannollinen yksilön kelpoisuuteen. Nimi rulettivalinta tulee ajatuksesta, että parhaille ratkaisuille on varattu ruletinpyörässä kelpoisuuden suhteessa suurimmat sektorit ja huonoimmille pienimmät.

- **SUS** (stochastic universal sampling), joka on muuten sama kuin rulettivalinta, mutta sillä tavalla painotettu, että kaikki parhaat ratkaisut tulevat välttämättä valituiksi ainakin kerran.

- **Turnausvalinta,** jossa valitaan satunnainen yksilö S populaatiosta ja valitaan sille annetun turnausluvun t verran satunnaisia vastustajia. Yksilö S asetetaan vuorotellen kunkin vastustajansa kanssa kilpailemaan vastakkain ja jos vastustajan fitness-arvo on korkeampi, S korvataan vastustajallaan. Lopuksi palautetaan jäljelle jäänyt S.

Geneettisiä algoritmeja on usein tehostettu tavoilla, joilla voidaan säästää parhaita yksilöitä lisäämässä valintapainetta. Elitismi eli parhaiden yksilöiden tallentaminen omaksi populaatiokseen varsinaisen populaation ulkopuolelle mainittiin jo aiemmin. Elitismiä käytetään paljon erityisesti monitavoiteoptimointialgoritmeissa. Toinen tehostus on ns. *Steady-State Genetic Algorithm,* jossa koko populaatiota ei tuoteta kerralla vaan lapsia tuotetaan yksitellen vanhaan populaatioon. Populaation muut jäsenet lisäävät uuden yksilön valintapainetta. [Luke, 2009, 46-48]

Geneettisiä algoritmeja on myös tehostettu yhdistämällä niitä vuorikiipeilijäalgoritmeihin. Näitä on kutsuttu myös *lamarckilaisiksi algoritmeiksi* Jean-Baptiste Lamarckin biologiassa hylätyn teorian mukaan, että yksilöt välittäisivät hankittuja ominaisuuksiaan jälkeläisilleen. Lamarckilaisissa algoritmeissa populaation jäsenet voivat kehittää itseään paikallishaulla ja pyrkivät periyttämään näin hankitut ominaisuutensa myös jälkipolville. [Luke, 2009, 50-51]

Verraten monimutkainen muunnelma on *Scatter search*, jossa valinta määräytyy paitsi kelpoisuudesta myös siitä, miten paljon yksilö poikkeaa populaation muista jäsenistä. Näin pyritään yhdistämään eksploitatiiviseen geneettisten algoritmien muunnelmaan myös eksploratiivisuutta ja takaamaan populaation monimuotoisuus. [Luke, 2009, 52-53]

Evolutionaariset algoritmit lähestyvät satunnaishakua, kun populaatiota kasvatetaan, ja vuorikiipeilijää, kun populaatiota supistetaan. Samoin voimakas valintapaine lähentää algoritmia vuorikiipeilijään ja heikko valintapaine satunnaiskulkuun. Suuri lapsiluku lähentää algoritmia jyrkimmän suunnan vuorikiipeilyyn ja pieni lapsiluku tavalliseen vuorikiipeilyyn [Luke, 2009, 52].

Sitä, miksi geneettiset algoritmit tarkalleen ottaen toimivat, ei vielä tunneta. Suosittu teoria on, että suotuisat *skeemat* rikastuvat populaatiossa evoluution myötä. Skeemalla tarkoitetaan rakennetta eli kromosomin pätkää, johon kuuluu kiinteitä ja vaihtuvia merkkejä. Kun vaihtuvat merkit merkitään tähdellä, esimerkiksi skeeman 1**1 mukaisia rakenteita ovat kromosomit $\{1001,1011,1101,1111\}$. [Reeves, 2002, 65-66]

2.3.3 Parvioptimointi

Parvioptimointi PSO (particle swarm optimization) on populaatiopohjainen optimointimenetelmä, joka muistuttaa jonkin verran evolutionaarista laskentaa, mut-

ta jonka esikuva on eläinten käytös parvissa ja laumoissa. Parvioptimoinissa ei synny uusia ratkaisuja, eikä siinä ole luonnonvalintaa. Populaatio pysyy koko ajan samana, mutta sen jäsenet lähestyvät *ohjautuvalla mutaatiolla* (directed mutation) parhaiten menestyviä yksilöitä. [Luke, 2009, 55-56]

Parven jäseniä kutsutaan usein partikkeleiksi tai hiukkasiksi. Niillä on ominaisuudet *paikka* ja *nopeus:* [Luke, 2009, 56]

- partikkelin paikka $\mathbf{x} = \langle x_1, x_2, \ldots \rangle$ vastaa evolutiivisten algoritmien genotyyppiä,
- partikkelin nopeus $\mathbf{v} = \langle v_1, v_2, \ldots \rangle$ kuvaa partikkelin liikettä paikasta toiseen; jos partikkeli $\mathbf{x}$ on hetkellä t paikassa $\mathbf{x}^{(t)}$ ja hetkellä $t+1$ paikassa $\mathbf{x}^{(t+1)}$, sen nopeus hetkellä $t+1$ on $\mathbf{v}^{(t+1)} = \mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}$.

Parvioptimoinnin alussa jokainen partikkeli on satunnaisessa paikassa ja sille on annettu nopeudeksi satunnainen nopeusvektori. Lisäksi pidetään kirjaa seuraavista asioista: [Luke, 2009, 56]

- Kunkin partikkelin $\mathbf{x}$ paras tähänastinen sijainti $\mathbf{x}^*$.
- Partikkelin $\mathbf{x}$ *informanttien* paras tähänastinen sijainti $\mathbf{x}^+$. Aluksi informantit olivat partikkelia lähimpänä olevia partikkeleita. Ne saattavat olla myös parven satunnaisesti valittu osajoukko. Partikkeli on aina myös oma informanttinsa.
- Paras tähänastinen sijainti koko parvessa $\mathbf{x}^!$.

Optimoinnin joka askeleella tehdään seuraavat kolme operaatiota: [Luke, 2009, 56]

1. Mitataan kunkin partikkelin sijainnin fitness ja päivitetään äsken mainittuja parhaiden tähänastisten sijaintien muuttujia mikäli tarpeellista.

2. Mutatoidaan kunkin partikkelin nopeusvektori. Mutaatiossa lisätään joillain sopivilla painoarvoilla nopeusvektoriin vektori, joka osoittaa kohti parasta tähänastista sijaintia $\mathbf{x}^*$, vektori, joka osoittaa parasta informantin sijaintia $\mathbf{x}^+$, ja vektori, joka osoittaa globaalisti parhaaseen tähänastiseen sijaintiin $\mathbf{x}^!$. Näiden lisäksi nopeusvektoriin lisätään hälyä, oma satunnainen arvonsa kullekin ulottuvuudelle.

3. Mutatoidaan kukin partikkeli liikuttamalla sitä nopeusvektorinsa mukaan.

3 Monitavoiteoptimointi

Verraten harvoin käy niin, että voidaan keskittyä optimoimaan yhtä arvoa ja jättää muut huomiotta. Usein pyritään useaan, jopa keskenään ristiriitaiseen tavoitteeseen yhtä aikaa. Esimerkiksi autosta halutaan sekä kestävä, helppo ja edullinen valmistaa, suorituskykyinen, turvallinen, tilava ja polttoainetta säästävä.

Tällaisessa tilanteessa ei ole yhtä oikeaa vastausta. Voidaan tietenkin pyrkiä valmistamaan täydellinen auto, mutta tällöin sen hinta voi karata niin suureksi, ettei kukaan osta sitä. Kallista ja hienoa autoa ei siis voi välttämättä sanoa paremmaksi kuin halpaa mutta toimivaa autoa.

Jos taas on kaksi autoa, joista ensimmäinen on parempi jollain mittarilla kuin toinen, mutta toinen ei ole millään mittarilla ensimmäistä parempi, ei ole mitään syytä valita jälkimmäistä. Tällöin sanotaan, että ensimmäinen auto dominoi toista autoa annetuilla kriteereillä.

Usean muuttujan yhtäaikaista optimointia kutsutaan multiobjektiiviseksi optimoinniksi tai vektorioptimoinniksi, koska siinä optimoitava arvo ei ole paljas skaalariluku vaan useamman luvun muodostama vektori. Monitavoiteoptimoinnissa ratkaisut eivät siis selvästikään ole täydellisessä järjestyksessä eli että jokainen ratkaisun tuottama arvo olisi parempi tai huonompi kuin toinen. Sen sijaan dominanssisuhde määrää niiden välille osittaisjärjestyksen.

Ratkaisua, jota mikään muu ratkaisu ei dominoi, kutsutaan Pareto-optimiksi ja Pareto-optimaalisten ratkaisujen joukkoa Pareto-joukoksi tai Pareto-rintamaksi. Optimoinnissa pyritään löytämään yhden arvon sijasta Pareto-joukko, tai jos se on ääretön tai muuten liian suuri, edustava otos siitä. Edustavuus tarkoittaa sitä, että löydettyjen ratkaisujen pitää edustaa koko Pareto-joukkoa eikä vain osaa siitä. Sanotaan, että vastauksilta vaaditaan *diversiteettiä.* [Lucas, 2006]

Monitavoiteoptimointi tuottaa usein yhden vastauksen sijaan suuren joukon mahdollisia vastauksia. Voikin sanoa, että monitavoiteoptimoinnissa on yhden sijasta kaksi ongelmaa: ensinnäkin varsinainen optimointi ja toiseksi vastauksen tai vastausten valinta tuloksena saadusta joukosta. Jälkimmäinen voidaan tehdä joko ennen optimointia, sen jälkeen tai optimoinnin aikana. Etukäteen valinta tarkoittaa useimmiten eri osafunktioiden painoarvojen määrittämistä. Jälkikäteinen tulosten valintaa Pareto-optimaalisesta joukosta.

Optimoinnin aikana evoluutiota voidaan ohjata pitämällä tavoitteet samoina ja muuttamalla niiden tärkeysjärjestystä (static objectives and soft preferences), tai muuttamalla tavoitteita ja pitämällä tärkeysjärjestys samana (soft objectives and static preferences). Myös sekä tavoitteita että tärkeysjärjestystä voi muuttaa dynaamisesti, jolloin puhutaan *koevoluutiosta* (coevolution). [Lucas, 2006]

3.1 Monitavoiteoptimoinnin ongelmia

Monitavoiteoptimoinnilla on neljä päätavoitetta. [Lucas, 2006]

1. Kattaa koko ongelma-avaruus niin, että menetelmä löytää kaikki hyvät ratkaisut, ei jää kiinni paikallisiin optimeihin eikä jätä olennaisia osia ongelma-avaruudesta tutkimatta.

2. Seurata Pareto-rintamaa niin läheltä kuin mahdollista eli löytää optimaaliset ratkaisut.

3. Tarjota monimuotoisia mahdollisimman erilaisia optimiratkaisuja, jotka kattavat koko Pareto-rintaman eivätkä klusteroidu vain yhteen tai muutamaan paikkaan.

4. Löytää vastaukset kohtuullisessa ajassa.

Näiden lisäksi myös kelpoisuusfunktion valinnassa ja sekä haun toteutuksessa on omat ongelmansa. Kelpoisuusfunktion mahdollisia ongelmia ovat seuraavat:

- **Kontekstisidonnaisuus** eli kelpoisuuteen vaikuttavat esimerkiksi populaation koko, ympäristö, historia tai vuorovaikutus toteutuneiden ratkaisujen välillä.

- **Informaatiopuute** eli ei tiedetä tarpeeksi eri tekijöiden käytöksestä tai painoarvoista.

- **Intransitiivisuus**, jonka vuoksi vertailu johtaa kehämäisiin tilanteisiin, kuten $A > B > C > A$.

- **Muuttuvat vaatimukset:** olosuhteet ja vaatimukset saattavat muuttua ajan kuluessa tai ne voivat vaihdella syklisesti.

- **Epälineaariset riippuvuussuhteet:** eri tavoitteet eivät ole toisistaan riippumattomia vaan vaikuttavat toisiinsa. Jos vaikka taloudessa kolme tarvittavaa resurssia on raha, aika ja ideat, voidaan ajatella, että ideat vaativat aikaa, ajan järjestäminen rahaa ja ideoita ja niin edelleen. Keskinäisesti vaikuttavat funktiot voivat muodostaa *outoja attraktoreita,* jolloin ne käyttäytyvät kaoottisesti ja luovat ongelma-avaruuteen erilaisia vaikeasti hahmotettavia järjestyksen ja epäjärjestyksen alueita.

- **Skaalaus ja normalisaatio** vaikuttavat tuloksiin tai ovat tapauskohtaisia.

Yksi ongelma käytännöllisessä monitavoiteoptimoinnissa on ratkaisujen robustisuus eli se, vaatiiko vastaus tarkalleen tietyt arvot ja olosuhteet ja mitä tapahtuu, jos nämä muuttuvat. Pudottaako pienikin muutos ratkaisun epäoptimaaliseksi? [Lucas, 2006]

3.2 Ideaalivektori ja Pareto-optimi

Monitavoiteoptimoinnissa optimoidaan funktiota, joka saa syötteekseen vektorin ja tuottaa ratkaisunaan vektorin. Tässä esityksessä syötettävä vektori on n-ulotteinen ja tulosvektori k-ulotteinen.

Ongelman mahdolliset ratkaisut ajatellaan siis n-paikkaisiksi vektoreiksi, jossa n reaalilukua edustaa arvoja, joista ratkaisu koostuu:

$$\mathbf{x} = [x_1, x_2, ..., x_n]^T.$$

Kaavassa T edustaa transpoosia. Vektorit esitetään tavallisesti sarakematriiseina, mutta tässä lähinnä esteettisistä syistä rivimuodossa.

Ratkaisuilla voi olla rajoitteita. Rajoitteet voivat vaatia, että ratkaisu $\mathbf{x}$ tuottaa m rajoitefunktiolla vähintään arvon nolla:

$$g_i(\mathbf{x}) \geq 0 \quad \text{kun} \quad i = 1, 2, ..., m. \tag{3.1}$$

Lisäksi otetaan p kappaletta rajoitefunktioita, joilla arvon pitää olla vektorilla $\mathbf{x}$ tasan nolla:

$$h_i(\mathbf{x}) = 0 \quad \text{kun} \quad i = 1, 2, ..., m. \tag{3.2}$$

Jokainen rajoitefunktio voidaan tietenkin saattaa helposti nollamuotoon lisäämällä tai vähentämällä siitä sopiva vakio.

Luvut m ja p voivat olla myös nollia, mikä tarkoittaa yksinkertaisesti sitä, ettei tässä tapauksessä tällaisia rajoitteita ole.

Optimoidaan funktiota f:

$$f(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_k(\mathbf{x})]^T. \tag{3.3}$$

Funktio f koostuu siis k:sta funktiosta, jotka saavat syötteekseen vektorin $\mathbf{x}$ ja jotka kukin tuottavat positiivisen reaaliluvun. Positiivisen siksi, että tällöin tulosavaruuden $\mathbb{R}^k_+$ nollapisteestä tulee mahdollisten tulosten absoluuttinen alaraja.

Optimaalisen tuloksen tuottavasta vektorista käytetään merkintää $\mathbf{x}^*$. Näitä optimaalisia ratkaisuja voi olla useampia kuin yksi.

Rajoitteet määrittävät niinsanotun käyttökelpoisen alueen F (feasible region), josta ratkaisuun kelpaavat vektorit $\mathbf{x}$ haetaan. Funktio f kuvaa alueen F joukolle X, joka sisältää kaikki kelvolliselta alueelta saavutettavat tulokset. Joukko $F \subseteq \mathbb{R}^n$ on siis funktion f lähtöjoukko ja $X \subseteq \mathbb{R}^k_+$ sen maalijoukko. [Coello, 2009, 111]

Perinteisessä optimoinnissa kaikki optimaaliset ratkaisut tuottavat saman tuloksen. Näin ei ole asian laita monitavoiteoptimoinnissa. Tavoitteet ovat usein ristiriidassa keskenään. On harvinaista, että ratkaisu $f(\mathbf{x})$ optimoisi jokaisen f:n osafunktion. Tällaisessa tapauksessa olisi olemassa ainakin yksi sellainen piste $\mathbf{x}^*$, että jokaisella osafunktiolla f_i olisi minimi yhdessä ja samassa $f_i(\mathbf{x}^*)$. Jos näitä pisteitä $\mathbf{x}^*$ on useampia, ne kaikki kuvautuvat samalle arvovektorille.

Olkoon vektori

$$\mathbf{x}^{0_{(i)}} = [x_1^{0_{(i)}}, x_2^{0_{(i)}}, ..., x_n^{0_{(i)}}]^T \tag{3.4}$$

ratkaisu, joka optimoi funktion i:nnen osatavoitteen eli funktion $f_i(\mathbf{x})$. Merkitään i:nnen funktion saamaa optimiarvoa $f_i(\mathbf{x}^{0_{(i)}})$ lyhyemmin f_i^0. Kannattaa huomata, että f_i^0 on skalaari joten se on myös yksikäsitteinen. Nyt funktion $f(\mathbf{x})$ **ideaalivektori** f^0 on vektori, joka koostuu jokaisen tavoitefunktion erikseen saamasta optimiarvosta:

$$f^0 = [f_1^0, f_2^0, ..., f_k^0]^T. \tag{3.5}$$

Ideaalivektori siis määrittää ratkaisun, jota parempi yksikään mahdollinen eli käyttökelpoiselta joukolta F kuvattu tulosjoukon X ratkaisu ei voi olla. Ideaalivektori on yksikäsitteinen ja sitä kutsutaan myös utopistiseksi ratkaisuksi koska ei ole mitenkään välttämätöntä, että se vastaisi mitään todellista ratkaisua. Ideaalivektori $f^0 \in \mathbb{R}_+^k$ ei siis välttämättä kuulu mahdollisten tulosten joukkoon X. [Coello, 2009, 111]

Perinteisessä skalaarioptimoinnissa jokaisesta kahdesta keskenään erilaisesta ratkaisusta toinen on parempi kuin toinen. Tämä seuraa siitä, että skalaarilukujen joukko on hyvinjärjestetty, eli jokainen skalaariluku on suurempi, pienempi tai yhtä suuri kuin toinen. Vektoreilla tällaista ominaisuutta ei ole. Vektori $[1, 11]^T$ on ensimmäiseltä arvoltaan pienempi kuin vektori $[2, 3]^T$ mutta toiselta arvoltaan suurempi. Näiden välille ei siis voi asettaa suuruusjärjestystä.

Vektoreiden välille voi kuitenkin määritellä osittaisjärjestyksen, jossa joillakin vektoreilla on keskenään järjestys mutta toisilla ei ole. Vektoreiden osittaisjärjestyksessä vektori on eräässä mielessä pienempi kuin toinen, jos mikään sen alkioista ei ole suurempi kuin toisen vastaava alkio ja vähintään yksi sen alkioista on pienempi kuin toisen alkio. Siis $[2, 2, 1]^T$ edeltää tässä osittaisjärjestyksessä vektoria $[2, 2, 2]^T$ mutta vektorin $[1, 2, 3]^T$ kanssa kummallakaan ei ole järjestyssuhdetta.

Monitavoiteoptimoinnin ratkaisujen välillä on edellä kuvattua osittaisjärjestystä noudattava suhde, jota kutsutaan *dominanssiksi.* Ratkaisu $\mathbf{x}$ dominoi ratkaisua $\mathbf{y}$ jos minimoitaessa jokaiselle $\mathbf{x}$:n arvolle pätee

$$f_i(\mathbf{x}) \leq f_i(\mathbf{y}). \tag{3.6}$$

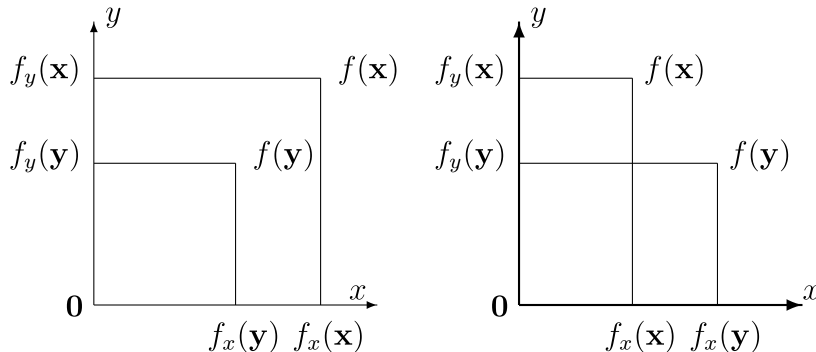
Tämä määrittelee niinsanotun *heikon dominanssin.* Dominanssi on vahva, jos se toteuttaa heikon dominanssin ehdot, mutta lisäksi vektori $\mathbf{x}$ saa ainakin yhdellä osafunktiolla vektoria $\mathbf{y}$ pienemmän arvon. Siis

$$f_i(\mathbf{x}) \leq f_i(\mathbf{y}) \text{ ja } \exists j \in 1, 2, ...k, \text{ jolle } f_j(\mathbf{x}) < f_j(\mathbf{y}). \quad (3.7)$$

Jokainen vahva dominanssi on myös heikko dominanssi, mutta jokainen heikko dominanssi ei välttämättä ole vahva dominanssi. Vektorien $\mathbf{x}$ ja $\mathbf{y}$ heikkoa dominanssia merkitään $\mathbf{x} \preceq \mathbf{y}$ ja vahvaa dominanssia $\mathbf{x} \prec \mathbf{y}$.

Kun ratkaisut ajatellaan kaksiulotteisina vektoreina, dominanssi tarkoittaa sitä, että kun piirretään neliö, jota rajoittavat x- ja y-akselit sekä pisteestä $f(\mathbf{y})$ akseleille piirretyt kohtisuorat, piste $f(\mathbf{x})$ dominoi pistettä $f(\mathbf{y})$ jos se on tämän nelikulmion sisällä.

Kuvan (1) vasemmalla puolella piste $f(\mathbf{y})$ dominoi pistettä $f(\mathbf{x})$ koska sekä $f_x(\mathbf{y}) < f_x(\mathbf{x})$ että $f_y(\mathbf{y}) < f_y(\mathbf{x})$. Oikealla puolella kumpikaan pisteistä ei dominoi toista. Osafunktio $f_y(\mathbf{y}) < f_y(\mathbf{x})$ mutta $f_x(\mathbf{y}) > f_x(\mathbf{x})$.



Tämän voi yleistää miten moneen ulottuvuuteen tahansa, jolloin nelikulmion korvaa ensin kolmessa ulottuvuudessa suorakulmainen särmiö ja useampien ulottuvuuksien tapauksessa *hyperkuutio* tai oikeammin *hypersuorakulmio* (hyperrectangle). Piste, jana, suorakulmio ja suorakulmainen särmiö ovat hypersuorakulmion 0, 1, 2 ja 3-ulotteisia tapauksia.

Pareto-optimiksi kutsutaan ratkaisua, joka ei ole vähintään yhdellä osafunktiollaan huonompi ja kaikilla muilla korkeintaan yhtä hyvä kuin mikään toinen ratkaisu. Toisin sanoen Pareto-optimista ei voi liikkua pois ilman, että jokin osa-alue saisi huonomman ratkaisun kuin se saa optimissa.

Vektori $\mathbf{x}^*$ on siis Pareto-optimaalinen, jos jokaiselle joukon F vektorille $\mathbf{x} \neq \mathbf{x}^*$ pätee joko

$$\bigwedge_{i \in I} (f_i(\mathbf{x}) = f_i(\mathbf{x}^*))$$

tai ainakin yhdellä indeksillä $i \in 1, 2, ..., k$

$$f_i(\mathbf{x}) > f_i(\mathbf{x}^*).$$

Pareto-optimaalista ratkaisua kutsutaan myös *dominoitumattomaksi* ratkaisuksi. Kaikkien Pareto-optimaalin tuottavien ratkaisujen $\mathbf{x}^*$ joukkoa kutsutaan **Pareto-joukoksi** ja ratkaisujen joukkoa f^* **Pareto-rintamaksi,** koska Pareto-rintaman pisteet muodostavat tulosjoukon X reunan.

Pareto-rintaman pisteet, joita mikään toinen piste ei dominoi edes heikosti, muodostavat sen *vahvasti dominoitumattoman reunan* ja muut pisteet *heikosti dominoitumattoman reunan.* [Coello, 2009, 112]

3.3 Aggregoivat menetelmät

Aggregoivissa menetelmissä useamman tavoitteen optimointi palautetaan tavalla tai toisella yhden tavoitteen optimoinniksi. Ne ovat tavallisesti laskennallisesti kevyitä ja usein algoritmia kokeillaan useilla eri parametreilla optimia haettaessa.

3.3.1 Tavoitteiden summaaminen

Ehkä yksinkertaisin tapa ratkaista monen tavoitteen optimointiongelma on laskea tavoitteiden arvot yhteen ja yrittää minimoida niiden summa. Yleinen kaava summaavalle k:n tavoitteen algoritmille on

$$\min \sum_{i=1}^{k} w_i f(\mathbf{x}) c_i, \tag{3.8}$$

kun w_i määrittää kyseisen tavoitteen painoarvon ja c_i vakioi kunkin funktion niin, etteivät osafunktiot, joiden tulokset kuuluvat suuremmalle vaihteluvälille, dominoi toisia osafunktioita.

Algoritmista on myös versioita, joissa yhteenlaskun sijaan tavoitefunktiot kerrotaan keskenään. Kertominen kannattaa erityisesti silloin, jos on tärkeää, ettei

mikään funktio saa arvoa nolla. Esimerkiksi jos hyvinvointia määrittävät raha, ruoka, hengitysilma ja valta, ratkaisut joissa hengitysilmaa ei saa ollenkaan, ovat tuskin tavoiteltavia, vaikka muut muuttujat saisivat niissä kuinka korkeita arvoja.

Summaavat algoritmit ovat laskennallisesti tehokkaita. Niiden heikkous on siinä, että annetuilla painoarvoilla on suuri merkitys. Tämä ei ole ongelma, jos optimoija tietää jo tarpeeksi ongelmastaan ja osaa antaa oikeat arvot, mutta jos painoarvoja ja vaihteluvälejä ei tunneta, algoritmi voi alkaa optimoida painoarvojen eikä tavoitefunktioiden mukaan. Tätä voidaan yrittää estää vaihtelemalla painoarvoja. [Coello, 2009, 115]

3.3.2 Tavoiteohjelmointi

Tavoiteohjelmoinnissa asetetaan tavoitteet, jotka halutaan saavuttaa ja optimoidaan sen mukaan, että pisteen etäisyys tavoitteesta minimoidaan

$$\min \sum_{i=1}^{k} |f_i(\mathbf{x}) - T_i|, \tag{3.9}$$

kun T_i on osafunktiolle annettu tavoite ja $\mathbf{x} \in F$. Menetelmän pääheikkous on siinä, että hyvän tavoitteen valinta vaatii melko paljon tietoa ongelmasta ja ongelmakentästä. Usein tavoitteeksi valitaan ideaalivektori.

Jos tavoite kuuluu mahdolliseen joukkoon X, sen saavuttaminen voi olla laskennallisesti hyvin tehokasta. Tosin tällöin piste on todennäköisesti dominoitu. Jos tavoite ei kuulu joukkoon X, algoritmi voi jäädä kiinni melko huonoihin ratkaisuihin löytämättä parhaita ratkaisuja. [Coello, 2009, 117]

3.3.3 Epsilon-rajoite

Epsilon-menetelmässä pyritään optimoimaan kaikkein tärkeintä osafunktiota niin, että muut funktiot saavuttaisivat kukin jonkin siedettäväksi katsotun tavoitearvon ε_i. Kun siis f_r on tärkeimmäksi katsottu osafunktio, haetaan vektoria $\mathbf{x}^*$, jolle pätee

$$f_r(\mathbf{x}^*) = \min_{\mathbf{x} \in F} f_r(\mathbf{x}) \tag{3.10}$$

sekä

$$f_i(\mathbf{x}) \leq \varepsilon_i \quad \text{kun} \quad i = 1, 2, ..., k \quad \text{ja} \quad i \neq k. \tag{3.11}$$

Dominoivia ratkaisuja eli Pareto-rintamaa haetaan vaihtelemalla rajoitteiden ε_i arvoja. Voidaan myös kokeilla eri vaihtoehtoja tärkeimmäksi tavoitteeksi ε_1. [Coello, 2009, 118-119]

3.4 Pareto-pohjaiset menetelmät

Pareto-pohjaisissa menetelmissä on ideana kerätä ratkaisuista dominoitumattomat ja kerätä ne algoritmin lopuksi palautettavien joukkoon. Uuden sukupolven ratkaisuja verrataan paitsi toisiinsa, myös tähän tallennettujen dominoitumattomien ratkaisujen joukkoon. Dominoitumattomat uudet ratkaisut lisätään joukkoon, ja jos uusi ratkaisu dominoi arkistossa olevia ratkaisuja, ne poistetaan arkistosta.

Tärkein ongelma Pareto-pohjaisissa algoritmeissa on se, ettei ole tehokasta tapaa tutkia ja valita joukosta dominoitumattomat ratkaisut. Laskenta käy raskaaksi etenkin populaation koon kasvaessa.

3.4.1 Geneettinen monitavoitealgoritmi MOGA

MOGA *(Multiple Objective Genetic Algorithm)* arvottaa ratkaisut sen mukaan, miten moni ratkaisu dominoi sitä. Tämä arvo antaa ratkaisulle *asteen* (rank) jonka mukaan se luokitellaan. Ratkaisun $\mathbf{x}_i$ arvo sukupolvessa t lasketaan seuraavalla kaavalla, kun $p_i^{(t)}$ on sitä sukupolvessa t dominoivien ratkaisujen lukumäärä:

$$rank(\mathbf{x}_i, t) = 1 + p_i^{(t)}. \tag{3.12}$$

Dominoitumattomat eli Pareto-rintaman ratkaisut saavat siis asteekseen arvon 1. Evolutiivisilla algoritmeilla on usein taipumusta konvergoitua liian nopeasti ensin löytyneiden ratkaisujen ympärille. MOGA yrittää estää tätä kahdella tavalla. Ensinnäkin tasaamalla

Ratkaisujen luokittelun ohella MOGA valitsee seuraavaan sukupolveen päästettävät ratkaisut myös sen mukaan, miten paljon ratkaisuja tässä kohtaa rintamaa

on jo löydetty. Lyhyesti ilmaistuna seuraavaan sukupolveen pääsee korkeammalla astearvolla jos ratkaisu poikkeaa merkittävästi jo löydetyistä eikä sen kaltaisia ratkaisuja vielä ole kovinkaan montaa.

MOGA on nopea ja helppo ohjelmoida. [Coello, 2009, 132-133]

3.4.2 NSGA ja NSGA-II

NSGA *(Nondominated Sorting Genetic Algorithm)* lajittelee ratkaisut dominanssin perusteella. Ensin kerätään dominoitumattomat ratkaisut ensimmäiseen ryhmään eli dominanssirintamaan, annetaan niille fitness-arvo rintamansa mukaan ja poistetaan ne populaatiosta. Tämän jälkeen kerätään nyt dominoitumattomat ratkaisut toiseen ryhmään ja niin edelleen. Seuraavaan sukupolveen valitaan vanhemmat satunnaisesti, mutta niin, että parhaiden ryhmien ratkaisut pääsevät todennäköisemmin lisääntymään. Algoritmin tehokkuus perustuu siihen, että valinta tapahtuu lajittelussa saadun fitness-arvon mukaan eikä jokaista monitavoitteisen funktion osafunktiota tarvitse laskea erikseen.

NSGA-algoritmi pseudokoodina on esitetty algoritmissa 8. [Coello, 2009, 134]

```
sukupolvi ← 0 ;
sukupolvimax ← sukupolvien lukumäärä ;
while sukupolvi < sukupolvimax do
    rintama ← 1 ;
    while koko populaatio ei ole vielä luokiteltu do
        etsi dominoitumattomat yksilöt ;
        anna niille fitness-arvo ;
        poista ne omaan luokkaansa ;
        rintama ← rintama + 1 ;
    end
    valitse lisääntyvät yksilöt fitness-arvonsa mukaan ;
    risteytä ;
    mutatoi ;
    sukupolvi ← sukupolvi + 1 ;
end
```

Algoritmi 8: NSGA

Pareto-perustaisten algoritmien heikkous on usein siinä, että ratkaisujoukon keskinäisten dominanssisuhteiden laskeminen on raskasta. Lajittelu luokkiin käyttää hyväkseen dominanssirelaation transitiivisuutta eli jos $\mathbf{x} \preceq \mathbf{y}$ ja $\mathbf{y} \preceq \mathbf{z}$, seuraa että myös $\mathbf{x} \preceq \mathbf{z}$. NSGA:n lajittelussa keskinäisten vertailujen lukumäärää pudottaa se, että aina kun löydetään dominanssisuhde $\mathbf{x} \preceq \mathbf{y}$ kahden vektorin välillä, dominoitu vektori $\mathbf{y}$ voidaan poistaa vertailusta. Sen dominanssilla muiden vektoreiden suhteen ei ole merkitystä, koska $\mathbf{x}$ dominoi myös kaikkia sen dominoimia vektoreita.

NSGA-II on NSGA-algoritmin kehittyneempi versio, jossa on tehokkaampi menetelmä ratkaisujen lajitteluun dominanssisuhteen mukaan sekä menetelmiä, joilla pyritään säilyttämään ratkaisujen monimuotoisuutta. [Coello, 2009, 133-136]

3.4.3 PAES

Arkistoivan evoluutiostrategian PAES *(Pareto Archived Evolutionary Strategy)* tarkoituksena on käyttää pelkästään paikallista hakua ja kohdella kaikkia dominoitumattomia ratkaisuja yhtäläisinä. Yksinkertaisimmillaan PAES on $(1,1)$-evoluutiostrategia, mutta siitä on myös muita versioita. PAES kerää dominoitumattomat ratkaisut arkistoon.

PAES-strategian pseudokoodiesitys on algoritmissa 9. [Knowles & Corne, 1999, 101]

$S \leftarrow$ generoi satunnainen ratkaisu ;
$A \leftarrow$ tyhjä arkisto ;
$A \leftarrow A \cup \{S\}$;
repeat
    $R \leftarrow Mutatoi(S)$;
    if $fitness(R) \preceq fitness(S)$ **then**
        vertaa ratkaisua R arkistoon ;
        päivitä arkisto ;
        $S \leftarrow$ satunnaisesti S tai R ;
    end
until *lopetusehto toteutuu*;

Algoritmi 9: PAES

PAES siis luo ensin satunnaisen ratkaisun ja lisää sen arkistoon. Ratkaisu jätetään nykyiseksi ratkaisuksi. Nykyisestä ratkaisusta S mutatoidaan uusi ratkaisu R ja

jos R dominoi nykyistä ratkaisua, sitä verrataan arkiston ratkaisuihin. Arkisto päivitetään niin, että sieltä poistetaan R:n mahdollisesti dominoimat ratkaisut ja jos mikään arkiston ratkaisu ei dominoi sitä, R lisätään arkistoon. Lopuksi valitaan uudeksi nykyiseksi ratkaisuksi joko S tai R.

PAESin muunnelmassa $(1 + \lambda)$ nykyisestä ratkaisusta luodaan λ mutaatiota ja muunnelmassa $(\mu + \lambda)$ on yhden nykyisen ratkaisun sijaan μ:n nykyisen ratkaisun joukko. [Knowles & Corne, 1999]

3.4.4 SPEA ja SPEA-2

SPEA *(Strength Pareto Evolutionary Algorithm)* on arkistoiva algoritmi, joka valitsee seuraavan sukupolven tuottavat yksilöt niiden vahvuusluvun perusteella. Aluksi siirretään jokainen dominoitumaton yksilö arkistoon.

Jokaiselle arkiston ratkaisulle $\mathbf{x}$ annetaan vahvuusluku $S(\mathbf{x}) \in [0, 1)$ joka lasketaan jakamalla ratkaisun $\mathbf{x}$ dominoimien tai yhtäläisten (eli sellaisten, jotka eivät dominoi ratkaisua eivätkä tule sen dominoimiksi) ratkaisujen lukumäärä kaikkien ratkaisujen lukumäärällä. Arkistoon kuuluvan ratkaisun $\mathbf{x}$ fitness $F(\mathbf{x})$ on sama kuin sen vahvuusluku $S(\mathbf{x})$.

Arkistoon kuulumattomien ratkaisujen $\mathbf{y}$ fitness on sitä dominoivien arkistoon kuuluvien ratkaisujen vahvuuslukujen $S(\mathbf{x})$ summa lisättynä ykkösellä.

Ratkaisut valitaan jatkamaan sukua binäärisillä turnajaisilla eli asettaen kaksi ratkaisua vastakkain ja valitsemalla niistä fitness-arvoltaan pienempi. Koska arkistossa olevien ratkaisujen fitness on korkeintaan yksi ja populaatiossa olevien vähintään yksi, jokaisen arkistossa olevien ratkaisujen todennäköisyys päästä luomaan seuraavaa sukupolvea on suurempi kuin yhdenkään populaation yksilön.

Populaation jäsenen todennäköisyys päästä jatkamaan sukua taas on pienempi mitä vähemmän sitä dominoidaan. Näin siis populaation ratkaisut, jotka ovat lähellä Pareto-rintamaa tai vähän kansoitetuilla alueilla pääsevät etusijalle.

Arkisto päivitetään. Uudet dominoitumattomat ratkaisut siirretään arkistoon, duplikaatit poistetaan arkistosta samoin kuin ratkaisut, joita uudet ratkaisut dominoivat. [Zitzler et al., 2001]

3.4.5 AMOSA

Geneettiset algoritmit ovat olleet suosittuja monitavoiteoptimoinnissa sen takia, että ne voivat kehittää yhtä aikaa useita erilaisia ratkaisuja. Koska simuloitu jäähdytys on yksitilainen (single-state) algoritmi, sen on katsottu sopivan paremmin skalaarioptimointiin. Simuloitua jäähdytystä on yleensä käytetty monitavoiteoptimoinnissa aggregoivissa menetelmissä eli tapauksissa, missä monitavoiteongelma on tavalla tai toisella palautetty skalaarioptimoinniksi.

Arkistoiva simuloitua jäähdytystä käyttävä monitavoiteoptimointialgoritmi AMOSA (Archived Multiple Objective Simulated Annealing) kerää dominoitumattomat ratkaisut erilliseen arkistoon. Arkistoa päivitetään joka kierroksella niin, että uusien ratkaisujen dominoimat ratkaisut poistetaan arkistosta ja sinne lisätään löytyneet uudet dominoitumattomat ratkaisut. [Bandyopadhyay et al., 2008].

3.5 Muut menetelmät

3.5.1 VEGA

VEGA *(Vector Evaluated Genetic Algorithm)* on seuraavan sukupolven valintaa lukuunottamatta samanlainen kuin perinteinen geneettinen algoritmi. Populaatio vain jaetaan k:hon yhtä suureen alipopulaatioon, jossa kussakin ratkaisut pisteytetään yhden tavoitteen mukaan. Seuraava sukupolvi saadaan sekoittamalla kunkin populaation parhaiten menestyneet ratkaisut ja jakamalla ne taas seuraavalla kierroksella satunnaisesti alipopulaatioihin. Tarkoituksena on luoda sukupolvien mittaan ratkaisuja, jotka menestyvät riippumatta siitä, mihin alipopulaatioon ne joutuvat. VEGAn heikkous on se, että käytännässä se johtaa usein *lajiutumiseen*, jossa osa populaatiosta erikoistuu menestymään joissakin alipopulaatioissa. [Coello, 2009, 120-122]

3.5.2 Sanakirjajärjestys

Leksikografinen eli sanakirjajärjestys tarkoittaa sitä, että tavoitefunktiot numeroidaan tärkeytensä mukaan. Ensimmäinen eli f_1 on tärkein, toinen eli f_2 toisiksi tärkein ja viimeinen eli f_k vähiten tärkeä.

Osafunktiot käydään järjestyksessä läpi. Ensimmäisellä kierroksella minimoidaan

$$\min f_1(x)$$

niin, että annetut rajoitteet toteutuvat ja ratkaisu x kuuluu kelvolliseen joukkoon F. Ensimmäisen osafunktion minimoiva tulos ${f_1}^* = {x_1}^*$ kiinnitetään uudeksi rajoitteeksi. Toisella kierroksella optimoidaan toinen osafunktio f_2 sillä rajoituksella, että tuloksen pitää tuottaa ensimmäisellä funktiolla sen globaali optimiarvo ${x_1}^*$. Saatu toisen osafunktion optimi ${x_2}^*$ kiinnitetään uudeksi lisärajoitteeksi. Kannattaa huomata, ettei tämä ole enää välttämättä funktion globaali optimi, vaan optimi rajoituksella, että ratkaisun pitää tuottaa optimiarvo tärkeimmälle osafunktiolle.

Yleisesti ratkaisun i. askel voidaan ilmaista muodossa

$$\min f_i(x) \quad \text{rajoitteella, että} \quad f_l(x) = {f_l}^* \quad \text{kun} \quad l = 1, 2, ..., i-1. \tag{3.13}$$

Viimeisen funktion f_k optimiratkaisu ${x_k}^*$ valitaan lopulliseksi ratkaisuksi. [Coello, 2009, 122]

3.5.3 Peliteoria

Ajatellaan kahden tavoitefunktion optimointia pelinä, jossa pelaaja A haluaa optimoida funktion f_1 ja pelaaja B funktion f_2.

Kun funktiot ratkaisevalla vektorilla $\mathbf{x}$ on n argumenttia, ensimmäinen pelaaja valitsee argumentin x_1 arvon, toinen pelaaja argumentin x_2 arvon, kolmannen argumentin x_3 valitsee taas ensimmäinen pelaaja kunnes viimeinenkin argumentti x_n on valittu. Jokainen pelaaja yrittää valita arvonsa niin, että se optimoisi mahdollisimman hyvin pelaajan oman tavoitefunktion.

Pelejä ja strategioita voi olla monenlaisia ja pelaajiakin enemmän kuin kaksi. Ilmeisin taktiikka on käyttää *maximin*-menetelmää eli tehdä valintansa niin, että huonoin mahdollinen tulos on mahdollisimman hyvä riippumatta siitä, miten myöhemmät pelaajat valitsevat. [Coello, 2009, 123-125]

3.5.4 Sukupuoli

Geneettisessä algoritmissa voi optimoida useampia tavoitefunktioita antamalla kullekin tavoitteelle sukupuoli. Kunkin sukupuolen edustajat arvotetaan tavoitteensa mukaan, huonoimmat poistetaan populaatiosta ja loput pariutuvat kukin jonkun vastakkaisen sukupuolen edustajan kanssa ja luovat seuraavan sukupolven, missä lapsille määrätään sukupuolet satunnaisesti.

Sukupuolia voi olla useampia kuin kaksi ja parinvalinnalle tarkempiakin taktiikoita.

3.6 Pareto-rintamien vertailua

Erilaisia monitavoiteoptimointimenetelmiä on paljon ja ne menestyvät hyvin vaihtelevasti eri ongelmissa. Kun on tarkoitus löytää paras tai parempi menetelmä, niiden tuotoksia pitäisi jotenkin verrata. Skalaarioptimointi tuottaa paljaan luvun, ja koska skalaarilukujen joukko on hyvinjärjestetty, kahta tulosta voi aina verrata. Monitavoiteoptimointi taas tuottaa keskenään dominoitumattomien vektoreiden joukon, joka yrittää kuvata todellista Pareto-rintamaa niin läheltä ja tarkkaan kuin mahdollista.

Useimmissa tapauksissa vertailu on vaikeaa. Kutsutaan Pareto-rintamaa nimellä Z^*. Onko sellainen algoritmi parempi, joka tuottaa yhden vektorin $\mathbf{x} \in Z^*$ vai sellainen, joka tuottaa joukon B, jonka vektoreita $\mathbf{x}$ ei dominoi? Ensimmäinen tuottaa varman tuloksen ja jälkimmäisen tuloksessa tuntuisi olevan enemmän informaatiota, mutta pelkästään näillä tiedoilla ei voida edes päätellä, miten tarkasti B kuvaa Pareto-rintamaa Z^*.

S-metriikka (S Metric) valitsee rintaman jokaisen pisteen dominoiman referenssipisteen $\mathbf{z}^{ref}$ rintaman ulkopuolelta. Piirretään suorakulmainen kappale, jonka jokainen sivu on koordinaattiakselien suuntainen ja jota rajoittavat $\mathbf{z}^{ref}$ ja rintaman pisteet. Lasketaan tämän kappaleen hypertilavuus. Rintama, jonka määrittelemän kappaleen tilavuus on suurin, on paras. Menetelmä on helposti ymmärrettävä ja monikäyttöinen, mutta riippuu onnistuneesta pisteen $\mathbf{z}^{ref}$ valinnasta. Lisäksi kappaleen hypertilavuuden laskeminen on laskennallisesti raskasta varsinkin jos ulottuvuuksia on paljon. [Knowles & Corne, 2001, 2-3]

Suhteellinen virhe ER (Error Ratio) määritetään kaavalla $(\sum_{i=1}^{n} e_i)/n$, missä e_i saa arvon 0 jos vektori i kuuluu Pareto-rintamaan Z^*, ja muuten arvon 1.

Tämän menetelmän heikkous on siinä, että se vaatii tietoa todellisesta Pareto-rintamasta. Suhteellinen virhe on helppo ymmärtää ja kevyt laskea.

Keskimääräinen etäisyys GD (Generational Distance) muistuttaa edellistä, mutta siinä lasketaan yhteen kunkin rintaman pisteen etäisyys lähimmästä Pareto-rintaman Z^* pisteestä ja jaetaan tulos pisteiden lukumäärällä.

Maksimivirhe MPEE (Maximum Pareto Front Error) laskee suurimman etäisyyden tulokseksi saadun joukon Z ja todellisen Pareto-rintaman Z^* välillä. Tarkemmin siinä haetaan jokaisen joukon Z pisteen etäisyydet kutakin lähimpään joukon Z^* pisteeseen ja valitaan näistä etäisyyksistä suurin. Maksimivirhe on nopea laskea ja se kertoo, onko saatu rintama joltain kohtaa kaukana Pareto-optimista. Se kuitenkin vaatii tiedon todellisesta optimista eikä kerro sitä, miten lähellä rintama yleisesti tätä yhtä pistettä lukuunottamatta on optimia.

4 Ohjelmistoarkkitehtuurit

Esittelen tässä luvussa ohjelmistoarkkitehtuurit tasolla, joka tarvitaan seuraavan luvun ymmärtämiseen.

Ohjelmistoarkkitehtuureilla on useampia määritelmiä. Yhteistä näille määritelmille on, että arkkitehtuuri paitsi kuvaa ohjelmiston korkeimman tason rakenteen, mutta myös sen osien keskinäiset suhteet ja vuorovaikutukset sekä suhteet ympäristöön. [Koskimies & Mikkonen, 2005, 18] Lisäksi arkkitehtuuri sisältää periaatteet, jotka ohjaavat järjestelmän suunnittelua ja evoluutiota. Erään määritelmän mukaan arkkitehtuuri on jotain, joka estää toteuttajia ja ylläpitäjiä käyttämästä tarpeetonta luovuutta. Toisin sanoen arkkitehtuuri antaa suuntaviivat sille, millainen ohjelmiston pitää olla ja miten sitä voidaan kehittää. Kun arkkitehtuurin tuntevat tietävät nämä asiat, heidän on helpompi ymmärtää paitsi olemassaolevaa ohjelmistoa, myös osia jotka siihen toteutetaan vasta myöhemmin.

Koskimies [Koskimies & Mikkonen, 2005, 18-19] kuvaa arkkitehtuuria ohjelmiston perustuslaiksi. Sitä pitää noudattaa järjestelmää rakennettaessa ja kehitettäessä, ja sitä voidaan muuttaa vain erityisen painavilla perusteilla. Arkkitehtuuri määrittelee rajat, joita pitää noudattaa, kun järjestelmää rakennetaan ja ylläpidetään

Olennaista on myös, että arkkitehtuuri on dokumentoitu. Dokumentoinnista puhutaan usein *arkkitehtuurikuvauksena.* Itse arkkitehtuuri onkin oikeastaan annettu kuvaus samoin kuin talojen arkkitehtuurissa arkkitehti tuottaa piirustukset, muttei yleensä itse rakenna taloa. Arkkitehtuurikuvauksesta asioita voi tarkistaa ja asioita voi kommunikoida toisille. Samoin dokumenteissa asiat säilyvät unohtumatta. Koskimiehen mukaan kuvaamattomasta arkkitehtuurista ei voi edes puhua arkkitehtuurina, koska vaikka järjestelmällä varmasti onkin jokin rakenne, eri ihmisillä on hyvin erilaiset mielikuvat ja katsantokannat, eikä heidän ole helppo tulla yhteisymmärrykseen siitä, mikä todellinen arkkitehtuuri loppujen lopuksi on.

Arkkitehtuuri ottaa Koskimiehen mukaan kantaa seuraaviin ohjelmiston toteutusta, käyttöä ja ylläpitoa koskeviin perusasioihin. Lista ei ole täydellinen, mutta kuvaa hyvin arkkitehtuurin alan ja merkityksen moninaisuutta:

- Ohjelmiston jakaminen osiin ja osien välistä vuorovaikutusta ja kommunikaatiota.

- Prosesseja ja niiden välistä kommunikointia.
- Tiedon saantitapoja.
- Pysyvyyden toteuttamista.
- Toiminnallisuuden sijoittelua järjestelmän eri osiin.
- Hajautetun järjestelmän fyysistä rakennetta.
- Järjestelmän kykyä käsitellä suuria tieto- ja käyttäjämääriä.
- Tehokkuutta.
- Varautumista tulevaisuuden tarpeisiin.
- Uudelleenkäytettävyyttä. [Koskimies & Mikkonen, 2005, 19]

Arkkitehtuuri on myös siinä mielessä olennainen ohjelmiston osa, että jo arkkitehtuuritasolla voidaan tehdä virheitä, jotka vaikuttavat merkittävästi ohjelmaan ja sen toteuttamiseen. Huono arkkitehtuuri voi esimerkiksi estää koko ohjelmiston toteutuksen, jos se vaikkapa tekee käytettävissä olevasta teknologiasta oletuksia, joita ei ole olemassa, käytettävissä tai toteutettavissa. Huonosti suunniteltu arkkitehtuuri voi olla niin vaikea toteuttaa, ettei ohjelmistoa saada valmiiksi tarkoitetussa ajassa. [Koskimies & Mikkonen, 2005, 17]

Huono arkkitehtuurivalinta voi myös vaikuttaa ohjelmiston suorituskykyyn. Se ei välttämättä suoriudu kuin pienistä tieto- tai käyttäjämääristä tai se saattaa olla niin hidas, että sen käytettävyys kärsii. Arkkitehtuuri, joka ei tue osien erillistä testaamista ja muokkaamista, tekee järjestelmän kehittämisestä ja ylläpidosta raskasta ja kallista. Samoin järjestelmä, jonka osien ja toimintojen rajoja ei olla kunnolla määritelty, saattaa olla vaikeaa tai mahdotonta siirtää uuteen ympäristöön. [Koskimies & Mikkonen, 2005, 17]

4.1 Komponentit ja rajapinnat

Arkkitehtuurin perusosat ovat komponentit ja rajapinnat. Komponentit ovat arkkitehtuurin pienimpiä perusyksiköitä. Niiden käyttö perustuu ajatukseen, että ohjelmisto voitaisiin koota keskenään vaihdettavista osista. Yksittäisen komponentin toteutus ei vaikuttaisi mitenkään järjestelmän muihin osiin. Koskimies määrittelee ohjelmistokomponentin *itsenäiseksi ohjelmistoyksiköksi, joka tarjoaa palvelujaan hyvin määritellyn rajapinnan kautta.* [Koskimies & Mikkonen, 2005, 53]

Komponentti ei kuitenkaan ole aivan täysin itsenäinen, vaan vaatii käytännössä jonkinlaisen *infrastruktuurin,* jonka puitteissa se voi toimia. Tällainen voi olla esimerkiksi jokin tietty sovellus, komponenttiteknologia tai ohjelmistokehys. Komponentin koolla ei ole rajoituksia. Se voi olla hyvin pieni ja yksinkertainen tai kokonainen sovellus. Olennaisempaa on, että se toimii komponentin roolissa. Komponentin sisäinen rakenne ei ole riippuvainen ympäristöstä eikä näy käyttäjille. Ennen kaikkea komponentilla on selkeät ja ilmaistavissa olevat velvollisuudet. Yhtenä käytännön sääntönä on pidetty, että komponentti on yhden henkilön vastuulla. [Koskimies & Mikkonen, 2005, 54-56]

On pidetty tärkeänä erottaa, mitä halutaan tehdä ja miten se saadaan aikaan. Se, mitä halutaan tehdä, erotetaan abstraktioksi. Käyttäjän ei tarvitse tietää muuta kuin tämän abstraktion, jota kutsutaan *rajapinnaksi.* Tämän rajapinnan toteuttaa yksi tai useampi komponentti. [Koskimies & Mikkonen, 2005, 57-58]

Sanotaan, että komponentti *tarjoaa* rajapinnan silloin, kun se toimii ja sitä voidaan käyttää jonkin tietyn rajapinnan antamien sääntöjen mukaisesti. Komponentti *vaatii* rajapinnan, kun se edellyttää, että sen kanssa vuorovaikutuksessa olevat ohjelmiston osat toimivat kuten jokin tietty rajapinta määrittelee. [Koskimies & Mikkonen, 2005, 59]

4.2 Suunnittelumallit

Suunnittelumallilla (*Design Pattern*) tarkoitetaan korkean abstraktiotason menetelmää, jolla voidaan ratkaista jokin ohjelmistotuotannossa usein esiintyvä ongelma. Se ei ole valmis osa, joka voitaisiin liittää ohjelmaan, vaan malli, jota voidaan soveltaa ohjelmaa toteutettaessa. Suunnittelumalli kuvaa, miten tietty usein esiintyvä ongelma voidaan ratkaista. Suunnittelumallit eivät edellytä mitään tiettyä

teknologiaa, suunnittelumenetelmää tai ohjelmointikieltä. [Koskimies & Mikkonen, 2005, 101]

Suunnittelumalli ja sen käyttöönotto ei muuta ohjelmiston toiminnallisuutta vaan parantaa sen jotain laatuominaisuutta. Suunnittelumallien filosofiaan kuuluu myös se, ettei niiden kuulu olla uusia ja hienostuneita ratkaisuja vaan yleiskäyttöisiä, koeteltuja ja ymmärrettäviä keinoja, jotka voidaan esittää tietyssä määrätyssä muodossa. [Koskimies & Mikkonen, 2005, 103-104]

Suunnittelumallien inspiraatio tuli rakennusten arkkitehtuurista. Vuonna 1977 julkaistussa kirjassaan *A Pattern Language: Towns, Buildings, Construction* arkkitehti Christopher Alexander esitti, että toimiva ja käytännöllinen arkkitehtuuri perustuu pitkän ajan kuluessa kertyneeseen kokemustietoon siitä, miten asiat kannattaa tehdä, mitkä ratkaisut toimivat ja mitkä eivät toimi. Näitä valmiita hyväksi koettuja ratkaisuja oli joka tasolle yhdyskunnista talojen ja asuntojen kautta huoneiden suunnitteluun. Alexander esitti kirjoissaan mallin, jonka avulla voitaisiin suunnitella laadukkaita rakennuksia soveltamalla systemaattisesti tästä käytännön tiedosta koottua tietovarastoa. Ajatus valmiiden suunnittelumallien soveltamisesta levisi rakennustekniikasta tietojenkäsittelyyn 1980- ja 1990-lukujen kuluessa. [Koskimies & Mikkonen, 2005, 102]

Suunnittelumalli ilmaistaan dokumentteina, joilla on yhtenäinen ja systemaattine rakenne. Näissä dokumenteissa suunnittelumalli jaetaan kolmeen osaan: ongelma (problem), ongelmayhteys (context) ja ratkaisu (solution). [Koskimies & Mikkonen, 2005, 105]

- Ongelman pitää olla yleinen suunnitteluongelma. Se ei saa vaatia mitään tiettyä tekniikkaa tai ohjelmointikieltä ja sen pitää esiintyä usein ja erilaisissa järjestelmissä. Joskus on jopa annettu lukumäärä, kuinka monta kertaa suunnittelumallia on pitänyt soveltaa onnistuneesti eri yhteyksissä, jotta se hyväksytään vakiintuneeksi suunnittelumalliksi.

- Ongelmayhteys kertoo, missä tilanteissa suunnittelumallia sovelletaan sekä sen, mitkä ovat ratkaisulle asetettavat vaatimukset.

- Ratkaisu on varsinainen keino, jolla ongelma ratkaistaan. Ratkaisu pitää esittää jollain tietyllä yleisesti käsitettävällä ja sovellettavalla formalismilla. Yleisin tapa on käyttää UML-mallinnuskieltä (unified modelling language).

Käytännössä suunnittelumallit kuvaavat olioita tai vastaavia komponentteja ja niiden välisiä suhteita ja vuorovaikutuksia, erityisesti näiden eri osien keskinäisiä suhteita. [Koskimies & Mikkonen, 2005, 105]

Kahden arkkitehtuurimallin lisäksi Räihä käyttää viittä suunnittelumallia: [Räihä, 2011, 33-34]

1. **Fasadi** (facade, suomeksi joskus 'kulissi') luo yhtenäisen ja yksinkertaistetun käyttöliittymän laajemmalle ohjelmiston osalle. Fasadi tekee osasta yksinkertaisemman käyttää, testata ja ymmärtää. Riippuvuus ohjelmiston osien välillä vähenee, kun liikenne viedään fasadin kautta. Joskus myös korvataan huono API luomalla sen päälle fasadina parempi. [Gamma et al, 1994]

2. **Välittäjä** (mediator) keskittää osien välisen yhteydenpidon yhteen komponenttiin.

3. **Sovitin** (adapter) korvaa järjestelmään sopimattoman käyttöliittymän järjestelmään sopivalla samaan tapaan kuin vaikkapa sähkötöpseleissä.

4. **Strategia** (strategy) antaa yhteisen käyttöliittymän, jonka mukaan valita tapa, jolla eri tavoilla mahdollisesti toteutettavat komponentit toteutetaan.

5. **Template method** tarkoittaa, että luodaan abstrakti yliluokka (template tai malli), jossa on muuttuvia eli abstrakteja ja muuttumattomia metodeja. Muuttumattomat metodit on toteutettu jo yläluokassa, muuttuvat toteutetaan aliluokissa.

4.3 Arkkitehtuurityylit

Kun suunnittelumallit kuvaavat matalan tason ratkaisuja, arkkitehtuurityylit ovat niiden yleistys korkeammalle tasolle koskemaan koko ohjelmistoa. Suunnittelumallien ja arkkitehtuurityylien välinen raja ei ole aivan yksiselitteinen. Nimitys vain vaihtuu jossain kohtaa, kun noustaan ylemmälle tasolle. Yleisesti voidaan kuitenkin todeta, että arkkitehtuurityyli määrittää ohjelmiston kokonaisrakenteen. [Koskimies & Mikkonen, 2005, 125]

Useimmin käytettyjä arkkitehtuurityylejä ovat kerrosarkkitehtuuri ja tietovuoarkkitehtuuri. Kerrosarkkitehtuurissa ohjelmiston arkkitehtuuri järjestetään kerroksiin abstraktiotasonsa mukaan. Korkeammalla olevat palvelut käyttävät alemmalla tasolla olevia palveluita. Ihannetapauksessa kukin palvelu olisi kontaktissa vain sitä seuraavan tason palveluihin. Tasojen ohittamista pidetään virheenä tai ainakin huonona tyylinä. Kerrosarkkitehtuuri helpottaa järjestelmän rakentamista hajottamalla sen eri tasoihin (dekompositioperiaate) ja toisaalta tukee sen ymmärtämistä ryhmittämällä asioita. [Koskimies & Mikkonen, 2005, 126-127]

Tietovuoarkkitehtuurissa (pipes and filters architecture) taas keskeisessä asemassa on tieto, jota kuljetetaan erilaisten tietoa käsittelevien komponenttien ja toisaalta pelkästään tietoa siirtävien reittien tai putkien (pipes) kautta. Räihä ei käytä näitä arkkitehtuureja Frankenstein-järjestelmässä, joten niitä ei käsitellä tämän enempää. [Koskimies & Mikkonen, 2005, 132]

Palveluperustaisissa arkkitehtuureissa toimijat ovat joko palvelun tarjoajan tai palvelun käyttäjän roolissa. Roolit eivät ole tiukkoja vaan sama yksikkö voi toisinaan käyttää palveluita ja toisinaan tarjota niitä.

Asiakas-palvelin-arkkitehtuurissa (client-server architecture) on ajatuksena kapseloida jokin resurssi palvelimeksi niin, ettei sen käyttäjien tarvitse huolehtia resurssin ylläpidosta ja muista siihen liittyvistä teknisistä ongelmista. Käyttäjät ovat palvelimen asiakkaita. Asiakas-palvelin-arkkitehtuuri on todennäköisesti tämän hetken käytetyin arkkitehtuuri. Asiakkaan ja palvelimen yhteydenpito tapahtuu useimmiten *istunnossa.* Palvelin odottaa passiivisesti kunnes asiakas käynnistää istunnon. Kun asiakas on saanut asiansa hoidettua, istunto suljetaan.

Palvelin toimii omassa säikeessään tai prosessissaan. Palvelimen toiminta voidaan näin eriyttää asiakaskomponenttien toiminnasta. Palvelin myös huolehtii istunnon aikaisten transaktioiden eheydestä ja mahdollisesta peruuttamisesta. Asiakas-palvelin-arkkitehtuuri eriyttää selkeästi toimijoiden vastuut ja tarjoaa pohjan hajauttamiselle. [Koskimies & Mikkonen, 2005, 137-138]

Viestinvälitysarkkitehtuuri (message dispatcher) perustuu ajatukseen, ettei komponenttien tarvitse tietää toistensa lukumäärästä tai laadusta. Järjestelmään saattaa tulla mukaan tai sieltä poistua komponentteja. Viestinvälitysarkkitehtuurissa komponentit kommunikoivat keskenään yhteisen viestinvälittäjän eli väylän kautta. Toisin kuin asiakas-palvelin-arkkitehtuureissa, tässä komponenttien rooleja ei olla kiinnitetty. Jokaisella komponentilla on yhteinen rajapinta viestinvä-

littäjään, jonka kautta tieto välittyy. Rajapinta sisältää operaatiot viestien vastaanottamiseen. Viesti kertoo komponentille, mitä sen pitää tehdä. [Koskimies & Mikkonen, 2005, 139]

Viestinvälitysarkkitehtuuri koostuu siis joukosta komponentteja, jotka ovat yhteydessä viestinvälittäjään eli väylään. Ne vastaanottavat viestejä tietämättä mistä ne tulevat tai minne niiden pitäisi omat viestinsä toimittaa. Komponenteilla on operaatioita, joilla ne reagoivat viesteihin, joihin operaatiot on ohjelmoitu reagoimaan.

Viestinvälittäjällä puolestaan on säännöt siitä, miten väylään liitetyt komponentit ja siellä kulkevat viestit rekisteröidään järjestelmälle sekä säännöt siitä, mille komponentille viestit lähetetään. [Koskimies & Mikkonen, 2005, 139-140]

Viestinvälittäjäarkkitehtuurien vahvuus on robustisuus ja muunneltavuus. Komponentteja ja toimintoja on helppo lisätä ja poistaa jopa ajonaikaisesti. Toisaalta sen heikkous on tehottomuus. [Koskimies & Mikkonen, 2005, 140]

4.4 Arkkitehtuurien arviointi

Arkkitehtuurit keskittyvät ratkaisemaan olennaisesti ohjelmiston laatuvaatimuksia. Siksi varsinainen ohjelman toiminnallisuus ei yleensä kuulu arkkitehtuuriin arviointiin. Toiminnalliset seikat, eli mitä ohjelmistolla voi tehdä ja mitä ei, voivat riippua arkkitehtuurista, mutta näiden tutkiminen ja toteaminen on yleensä melko suoraviivaista. [Koskimies & Mikkonen, 2005, 223-224]

Arvioitavia laatuvaatimuksia ovat esimerkiksi:

- Järjestelmän suorituskyky ja tehokkuus.
- Luotettavuus. Järjestelmä pysyy toimintakuntoisena.
- Turvallisuus. Järjestelmä pystyy torjumaan väärinkäytökset ja oikeudettomat käyttäjät aiheuttamatta sillä ongelmia tavallisille käyttäjille.
- Muunneltavuus. Järjestelmään on helppo tehdä muutoksia.
- Siirrettävyys. Järjestelmä on helppo siirtää eri ympäristöihin.
- Muunnteltavuus. Järjestelmään on helppo tehdä muutoksia. [Koskimies & Mikkonen, 2005, 223]

4.4.1 Metriikat

Metriikat ovat menetelmiä, joilla arkkitehtuurien arviointi tuottaa jonkin lukuarvon. Olio-ohjelmoinnissa on tavallista tutkia arkkitehtuuria erilaisilla metriikoilla. Metriikat perustuvat luokkien ominaisuuksiin, niiden välisiin yhteyksiin jne. Metriikoita voi mitata automaattisesti, vaikka ihminen kykenee myös abstraktimpaan arvosteluun.

Tunnetuin on Chidamberin ja Kemererin CK-metriikkajoukko. CK-metriikoihin kuuluu: [Räihä, 2011, 35]

- **WMC** (Weighted methods per class) kertoo metodien lukumäärän luokassa.
- **DIT** (Depth of inheritance tree) kertoo etäisyyden juuriluokassa eli miten syvältä metodeja peritään.
- **NOC** (Number of children) on suorien lapsiluokkien lukumäärä
- **CBO** (coupling between objects) ilmoittaa, kuinka usein käytetään toisessa luokassa määriteltyä metodia tai attribuuttia.
- **RFC** (response for a class) on luokan kutsumien metodien (omien tai toisten) lukumäärä.
- **LCOM** (lack of cohesion in methods) mittaa, käyttävätko metodit olionsa samoja muuttujia.

CBO ja LCOM ilmentävät muunneltavuutta (modifiability), WMC uudelleenkäytettävyyttä (reusability.) Korkea DIT-arvo ennustaa monimutkaisuutta ja vaikeaa ylläpidettävyyttä, mutta hyvää uudelleenkäytettävyyttä. Jos NOC-arvo on korkea, komponentti pitää testata kattavasti, mutta tällä voi myös mitata tehokkuutta ja uudelleenkäytettävyyttä. RFC:llä mitataan ymmärrettävyyttä, ylläpidettävyyttä ja testattavuutta.

Uudempi metriikka on QMOOD, jonka laatumetriikat käsittelevät toiminnallisuutta, tehokkuutta, ymmärrettävyyttä, laajennettavuutta, uudelleenkäytettävyyttä ja joustavuutta. Nämä ominaisuudet muunnetaan suunnittelutavoitteiksi ja edelleen metriikoiksi. Järjestelmässä on 11 suunnittelutavoitetta ja kutakin niistä vastaavaa metriikkaa. Tavoitteita yhdistellään erilaisten ominaisuuksien mittaamiseksi. Metriikat ovat: luokkien koko, hierarkioiden lukumäärä, esiisien lukumäärän keskiarvo, polymorfisten metodien lukumäärä, luokkien käyttöliittymien (interfaces) koko, metodien lukumäärä, tiedonhakumetriikka, suorat yhteydet (coupling) luokkien välillä, luokan metodien koheesio, aggregointi ja toiminnallinen (funktionaalinen) abstrahointi. [Räihä, 2011, 36-37]

4.4.2 Laadullinen arviointi

Ihmisen tekemä laadullinen arviointi keskittyy sellaisiin kysymyksiin kuten miten arkkitehtuuri ratkaisee laatuvaatimuksen x, miksi jokin ratkaisu on parempi kuin toinen. Tyypillisesti esitetään *mikä* tai *miksi*-kysymyksiä, jotka ovat vaikeita koneille. Tyypillinen ja käytetyin laadullinen mittaustapa on ATAM (Architecture Tradeoff Analysis Method.) Siinä kukin laatuvaatimus mitataan joukolla skenaarioita, jotka kuvaavat käyttäjän ja järjestelmän vuorovaikutusta. Skenaarioilla tutkitaan ensisijaisesti sitä, täyttääkö järjestelmä jonkin laatuvaatimuksen. Ne voivat myös paljastaa heikkouksia ja riskejä järjestelmässä. [Räihä, 2011, 37-38]

5 Frankenstein-ohjelma

Frankenstein-ohjelma tuottaa ohjelmistoarkkitehtuurin geneettisillä algoritmeilla. Myös muita metodeja on toteutettu, mm. simuloitu jäähdytys. Räihä [2011] käyttää kahta esimerkkiarkkitehtuuria. Ne ovat automatisoitu älykoti *eHome* sekä robottisotapelisimulaattori *robo*.

Ohjelma saa syötteekseen niin sanotun *nolla-arkkitehtuurin*, jota se alkaa kehittää. Koska Frankenstein ei voi ymmärtää ohjelman tarkoituksia ja päämääriä, ihmisen pitää syöttää sille arkkitehtuuri, jota aletaan rakentaa funktionaalisina vaatimuksina, jotka ilmaistaan sekvenssikaaviona (sequence diagram), joka kertoo, millaisia vuorovaikutuksia ohjelmalla on käyttäjän kanssa tai muuten. Kaavio voidaan muuttaa automaattisesti luokkakaavioksi, ja tämä luokkakaavio syötetään ohjelmaan nolla-arkkitehtuurina. [Räihä, 2011, 52-56]

Nolla-arkkitehtuuri pitää muuttaa muotoon, jossa geneettinen algoritmi voi käsitellä sitä. Ohjelmalle syötetään *supergeeni* nimeltä sg_i.

Supergeeni koostuu kahdenlaisista osista: perusinformaatiosta ja arkkitehtuuritiedosta. Perusinformaatio saadaan ohjelman syötteenä, eikä sitä muuteta ohjelman suorituksen aikana. Arkkitehtuuri on varsinainen muunneltava osa, josta saadaan varsinainen tulos ohjelman lopuksi.

Perusinformaatio käsittää seuraavat osat:

- Operaatioita kuvaava joukko $O_i = \{o_{i1}, o_{i2}, \ldots, o_{in}\}$, missä operaatio o_k seuraa operaatiosta o_j ja on siitä riippuvainen, jos $j < k$.
- Nimi n_i.
- Tyyppi d_i.
- Frekvenssi f_i.
- Parametrien koko p_i.
- Variabiliteetti v_i.
- Annettu nolla-arkkitehtuuri MC_i.

Kannattaa huomata, että nolla-arkkitehtuuri kuuluu näihin syöttötietoihin, joita geneettinen algoritmi ei muuta.

Arkkitehtuuritietoon kuuluvat seuraavat kentät:

- Luokka C_i, johon operaatio o_i kuuluu.
- Rajapinta (interface) I_i, jota se käyttää.
- Viestinvälittäjä (message dispatcher) D_i.
- Operaatiot $OD_i \subseteq O_i$ joita kutsutaan viestinvälittäjän kautta.
- Suunnittelumalli (design pattern) P_i.

Nämä kentät kerätään supergeeniksi, ja jokainen supergeeni ilmaisee yhden operaation. Jos järjestelmässä on n kappaletta erilaisia operaatioita, sitä ilmaisee *kromosomi* $< sg_1, sg_2, \ldots, sg_n >$.

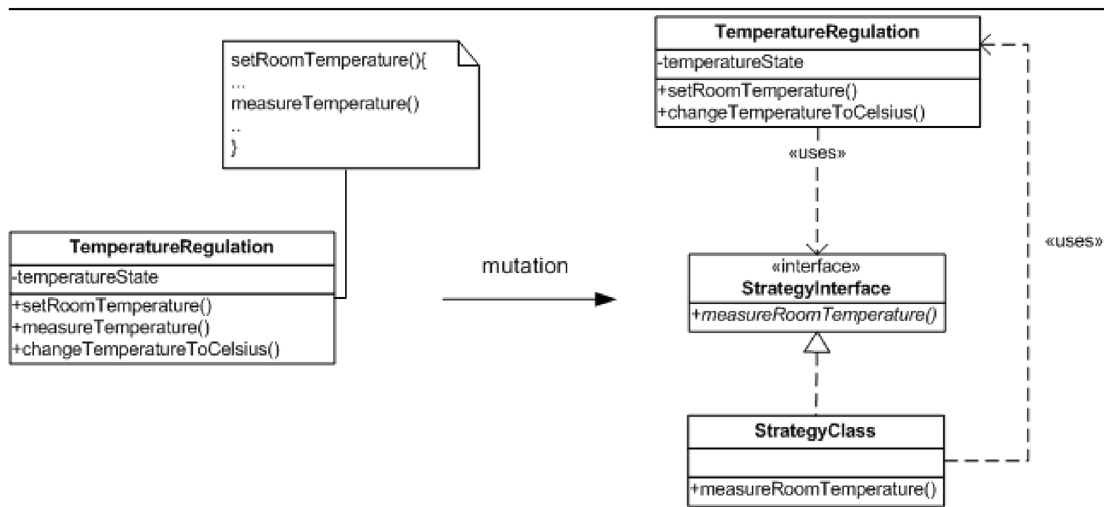
Järjestelmä kootaan siis operaatiokeskeisesti. Kukin supergeeni ilmaisee, miten jokin operaatio voidaan toteuttaa, ei sitä, miten luokat kommunikoivat keskenään.

Koska geneettinen algoritmi tarvitsee lähtötilanteekseen populaation, nolla-arkkitehtuurista pitää generoida erilaisia kopioita, että saadaan joukko, josta evoluutio voidaan aloittaa. Tämä tapahtuu luomalla nolla-arkkitehtuurista sata kopiota ja mutatoimalla kukin niistä kerran. Mutaatiosta kerrotaan seuraavassa kohdassa.

5.1 Mutaatio ja risteytys

Mutaatiot toteutetaan lisäämällä tai poistamalla suunnittelumalleja (fasadi, välittäjä, sovitin, strategia, template method) ja arkkitehtuurityylejä (viestinvälitysarkkitehtuuri, asiakas-palvelin-arkkitehtuuri). Mutaatioihin kuuluu myös *nollamutaatio*, jossa mitään ei muuteta.

Mutaatiot kohdistuvat yleensä yhteen operaatioon eli supergeeniin. Poikkeuksena viestinvälittäjä ja fasadi. Mutaatiot toteutetaan kussakin suunnittelumallissa kuvassa 5.1 esitetyllä tavalla:



Kuva 5.1 Strategia-suunnittelumalli

5.1.1 Strategia

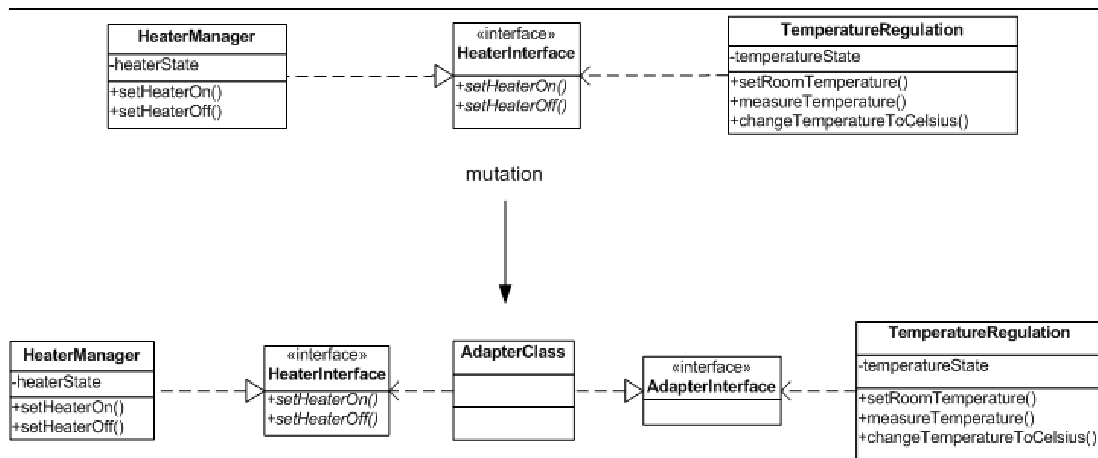
Strategiasuunnittelumalli vaatii, että operaatiota o_i kutsutaan jostain toisesta saman luokan operaatiosta o_k. Kun operaatiolle annetaan strategiasuunnittelumalli, luodaan instanssi SP strategialuokasta. SP sisältää tietoa strategian antamasta rajapinnasta SI, strategian toteuttavat luokat SC ja operaatiot, joita strategia koskee. Muokattava supergeeni sg_i päivitetään niin, että sen arvoksi C_i (eli luokaksi, johon operaatio kuuluu) asetetaan SC, sen rajapinnaksi I_i asetetaan SI ja sen suunnittelumallia osoittavaan osaan P_i asetetaan SP.

Strategia poistetaan yksinkertaisesti muuttamalla supergeenin suunnittelumallia osoittava kenttä P_i nollaksi.

Kuvassa 5.1 mutatoidaan metodi (operaatio) measureTemperature, joka kuuluu luokkaan TemperatureRegulation ja jota metodi setRoomTemperature käyttää. Mutaatiossa peritään Strategia-suunnittelumallista luokka StrategyClass SC ja rajapinta StrategyInterface SI. Metodi measureTemperature siis siirretään omaan luokkaansa, jolla on rajapinta, jonka kautta luokka TemperatureRegulation sitä käyttää.

5.1.2 Sovitin

Sovitin (adapter) edellyttää, että operaatiota o_i kutsutaan jostain toisesta luokasta. Sovitinmalli luo instanssin AP, joka sisältää rajapinnan AI ja toteutus-



Kuva 5.2 Sovitin-suunnittelumalli

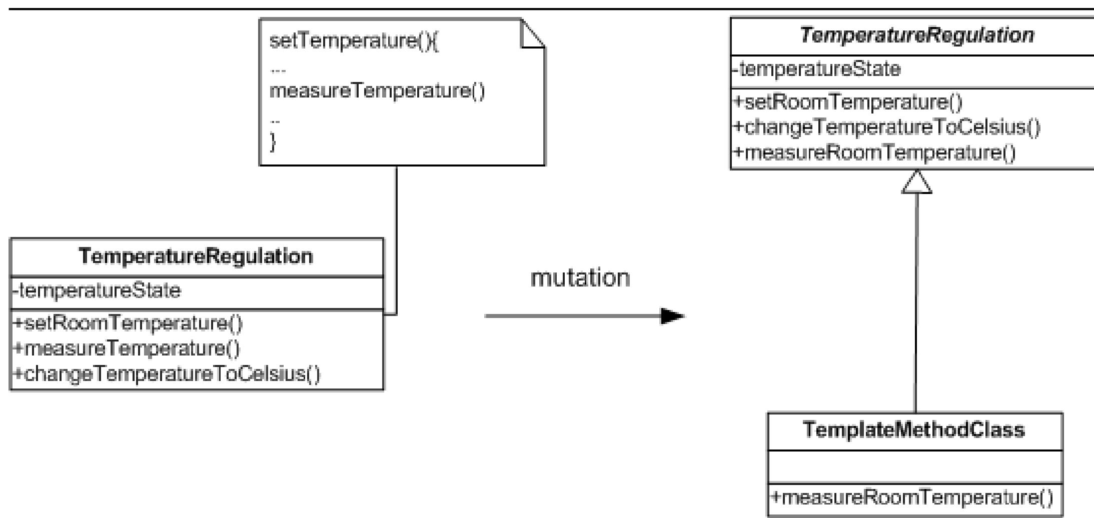
luokan AC sekä operaatiota, jotka ovat mukana sovitinta vaativassa tapauksessa. Suunnittelumallia osoittavaan kenttään P_i lisätään AP ja samoin kuin strategian tapauksessa, poistettaessa kenttä P_i asetetaan nollaksi.

Kuvassa 5.2 sovitin-suunnittelumalli lisää rajapinnan heaterInterface ja luokan TemperatureRegulation väliin tyhjän luokan AdapterClass (AC) sekä rajapinnan AdapterInterface (AI).

5.1.3 Template method

Template methodissa luodaan yliluokka ja se edellyttää, että operaatio käyttää muita saman luokan operaatioita. Nämä muut operaatiot ovat $o_k, \ldots o_t$. Template method luo instanssin TP, joka sisältää sen konkreettisen toteutuksen TC sekä operaatiot, joita asia koskee. TP päivitetään nyt paitsi operaation o_i suunnittelumalliksi P_i, myös muiden templaten alle tulevien operaatioiden suunnittelumallikohtiin eli $P_k, \ldots P_t$.

Kuvassa 5.3 sovelletaan template method -suunnittelumallia metodiin setRoomTemperature (operaatio o_i), joka käyttää toista saman luokan metodia measureTemperature (operaatio o_k). Suunnittelumalli luo uuden luokan TemplateMethodClass, johon sijoitetaan operaation o_i käyttämä metodi nimellä measureRoomTemperature. Luokka perii alkuperäisen luokan TemperatureRegulation ja muuttaa sen abstraktiksi.



Kuva 5.3 Template method -suunnittelumalli

5.1.4 Fasadi

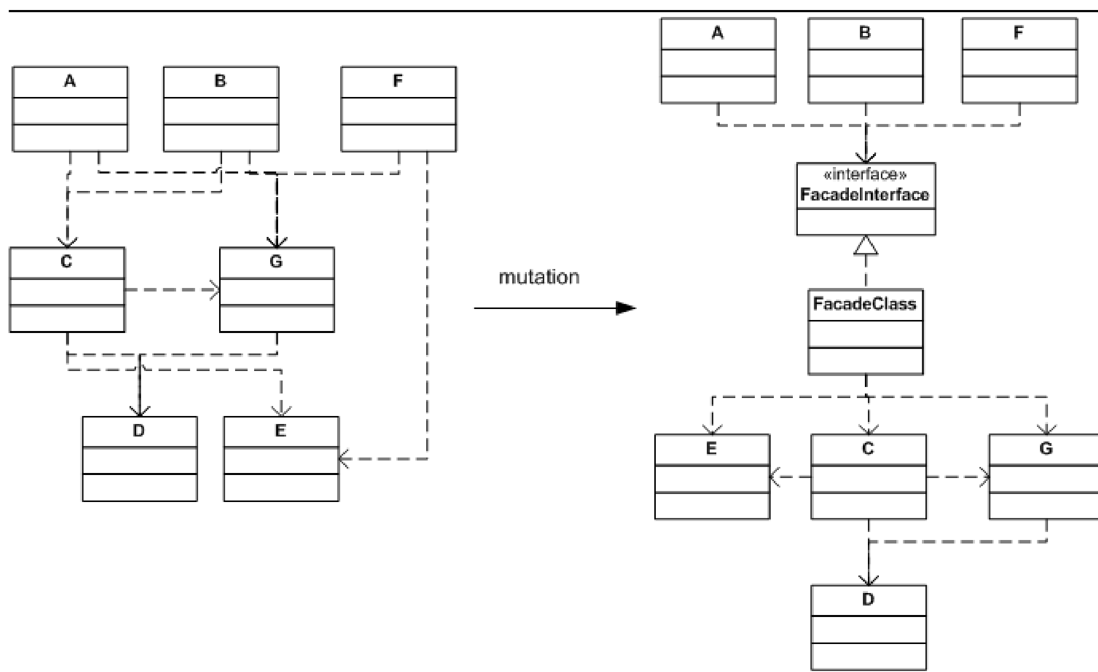
Fasadin tarkoitus on luoda useammalle operaatiolle yhteinen käytettävä rajapinta. Fasadin instanssi FP sisältää rajapinnan FI, joka kopioidaan rajapinnaksi jokaisen operaation $o_k, \ldots, o_t$ supergeenin rajapintakenttään $I_k, \ldots, I_t$. Suunnittelumallikenttiin $P_k, \ldots, P_t$ kopioidaan FP ja poistettaessa kentät $P_k, \ldots, P_t$ asetetaan nolliksi.

Kuvassa 5.4 luokat A, B ja F käyttävät luokkia C, E ja G ja välillisesti luokkaa D. Lisätään luokkajoukkojen $\{A, B, F\}$ sekä luokkien $\{C, D, E, G\}$ väliin fasadi. Se tapahtuu johtamalla vuorovaikutukset luokista A, B ja F uuteen tyhjään luokkaan FacadeClass, jota käytetään rajapinnan FacadeInterface kautta, ja joka johtaa vuorovaikutukset entiseen tapaan muihin luokkiin.

5.1.5 Välittäjä

Välittäjän (mediator) lisäys supergeeneihin tapahtuu samalla tavalla kuin fasadissa.

Kuvassa 5.5 luokkien A, B, C, D, E ja F välillä on monia erilaisia vuorovaikutuksia. Lisätään uusi luokka MediatorClass ja sen rajapinta MediatorInterface. Luokkien jokainen pyyntö kohdistetaan tähän uuteen luokkaan, joka edelleen jatkaa pyynnön eteenpäin haluttuun luokkaan.



Kuva 5.4 Fasadi-suunnittelumalli

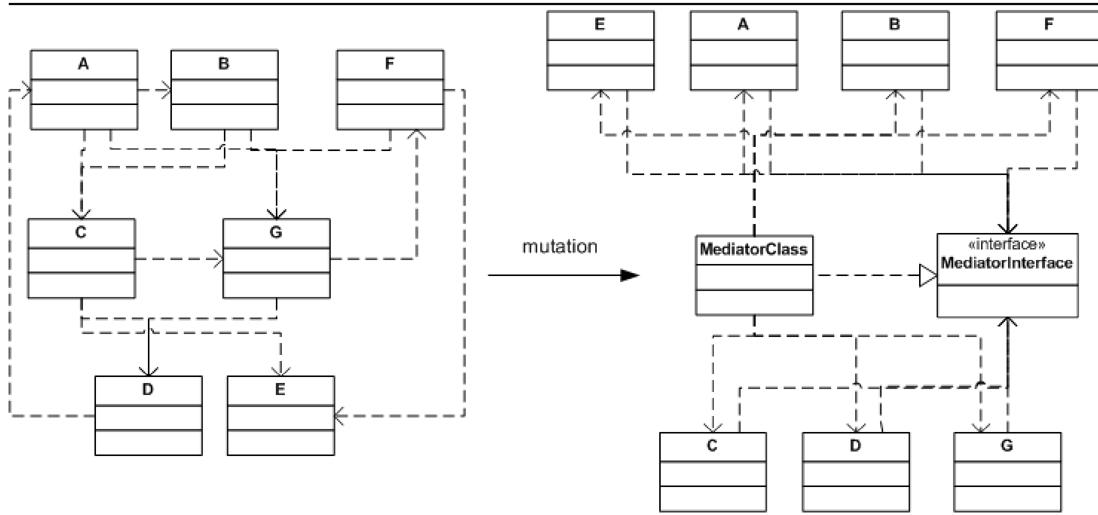
Suunnittelumallien lisäksi on kaksi arkkitehtuurimallia, joiden mutaatio toteutetaan seuraavilla tavoilla.

5.1.6 Arkkitehtuurimallit

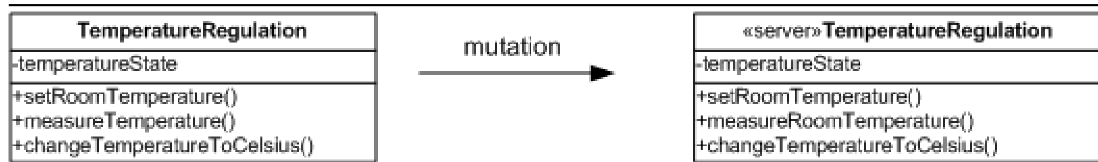
Asiakas-palvelin-arkkitehtuuri (client server architecture) vaikuttaa kaikkiin sen luokan operaatioihin, johon mutatoitava operaatio o_i kuuluu. Lisäksi on ehto, että palvelinyhteys koetaan järkeväksi vain, kun se yhdistää vähintään kolmea operaatioita.

Kuvassa 5.6 luokka TemperatureRegulation muutetaan palvelimeksi käyttämällä palvelin-prototyyppiä. Tämän jälkeen kaikki luokkaa TemperatureRegulation käyttävät operaatiot käyttävät sitä asiakkaina.

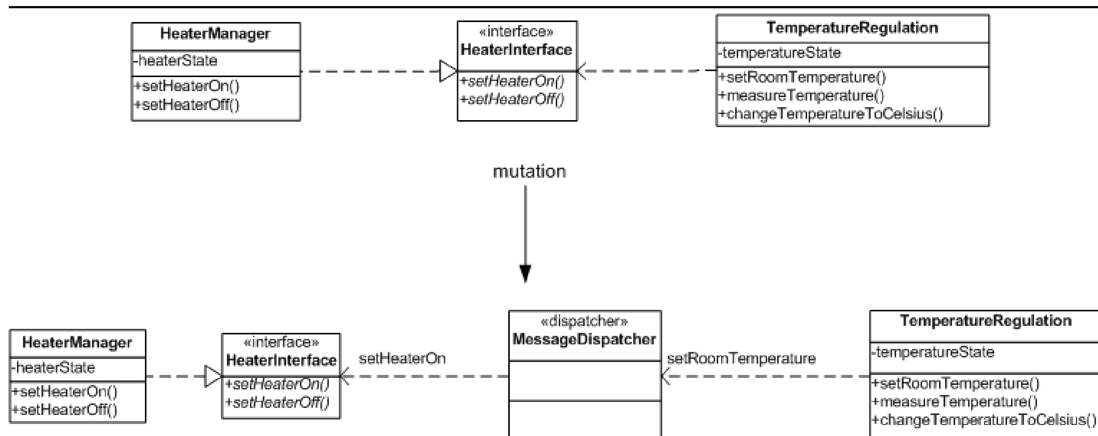
Toisin kuin aiemmissa tapauksissa, viestinvälitysarkkitehtuurin lisääminen tapahtuu kahdessa vaiheessa. Ensimmäisessä lisätään kromosomiin viestinvälittäjä uutena supergeeninä, jonka kaikki muut kentät ovat nollia, mutta jonka kentässä D on arvona ykkönen sen merkiksi, että viestinvälittäjä on käytössä. Seuraavassa vaiheessa eri operaatiot voivat alkaa käyttää viestinvälittäjää kommunikaatioonsa. Näiden operaatioiden pitää sijaita nolla-arkkitehtuurin eri luokissa.



Kuva 5.5 Välittäjä-suunnittelumalli



Kuva 5.6 Asiakas-palvelin-arkkitehtuuri



Kuva 5.7 Viestinvälitysarkkitehtuuri

Kuvassa 5.7 luokat kommunikoivat keskenään. Mutaatiossa luodaan viestinvälittäjä MessageDispatcher, jonka kautta kommunikaatio aletaan hoitaa. Kuvassa on vain kaksi luokkaa, mutta jos siinä olisi useampia, ne kaikki olisivat suoraan yhteydessä vain viestinvälittäjään.

5.1.7 Risteytys

Kahden supergeenin risteytys tapahtuu katkaisemalla ne satunnaisesta kohdasta ja yhdistämällä ne kahdeksi uudeksi lapseksi. Sekä lapset että vanhemmat lisätään populaatioon. Fitness vaikuttaa lisääntymistodennäköisyyteen: todennäköisyys kasvaa lineaarisesti sen mukaan, miten supergeeni sijoittuu fitnessiltään populaatioiden yksilöiden joukossa. Lisäksi kelpoisuudeltaan ylempään puoliskoon kuuluvan lisääntymistodennäköisyys kaksinkertaistetaan ja huonompaan puoliskoon kuuluvan puolitetaan.

Vanhempien valinnassa käytetään rulettivalintaa. Supergeenien lisääntymistodennäköisyydet kerätään rulettipyörään erisuuruisiksi sektoreiksi. Kun rulettipyörää pyöritetään, kuula voi osua joko mutaatioon tai risteytykseen. Jos se osuu mutaatioon, kyseinen supergeeni mutatoidaan ja pyörää pyöritetään uudestaan. Jos kuula osuu toisellakin kerralla mutaatioon, toista mutaatiota ei enää toteuteta. Jos se osuu ensimmäisellä tai toisella kierroksella risteytykseen, supergeeni otetaan mukaan lisääntyvien supergeenien joukkoon eli vanhempien pooliin (parent pool).

Lisääntyvät yksilöt poimitaan vanhempien poolista, risteytetään keskenään, eikä niitä enää palauteta, joten seuraavaan pariin valitaan eri vanhemmat. Jokainen yksilö voi siis lisääntyä vain kerran sukupolven aikana. Lisääntymisen jälkeen tarkistetaan, että syntyneet yksilöt ovat arkkitehtuuriltaan sallittuja (corrective function).

5.2 Fitnessin mittaaminen

Mutaatioiden ja risteytyksen jälkeen populaatio on suurempi kuin sukupolven alussa. Tässä kohtaa aletaan soveltaa luonnonvalintaa, eli heikoimmat yksilöt poistetaan populaatiosta.

Yksilöt arvotetaan kolmen ominaisuuden perusteella: muokattavuus (modifiability), tehokkuus (efficiency) ja kompleksisuus (complexity). Muokattavuus parantaa ohjelmiston ylläpidettävyyttä. Tehokkuus on joissain tapauksissa ristiriidassa muokattavuuden kanssa. Tehokkuus vaatimuksena estää sen, että ohjelma sovittaisi suunnittelumalleja kaikkeen mahdolliseen välittämättä mistään muusta. Kompleksisuus antaa vastapainoa muokattavuudelle myös tapauksissa, joissa ei voida osoittaa selvää haittaa tehokkuudelle.

Lopullinen fitness-arvo mitataan kaavalla

$$f(x) = w_1 * sf_1 - w_2 * sf_2 + w_3 * sf_3 - w_4 * sf_4 - w_5 * sf_5, \tag{5.1}$$

missä sf_1 kuvaa positiivista muokattavuutta, sf_2 negatiivista muokattavuutta, sf_3 positiivista tehokkuutta, sf_4 negatiivista tehokkuutta ja sf_5 kompleksisuutta.

On huomattava, että tässä kaikki suunnittelumallit parantavat muokattavuutta, mutta huonontavat tehokkuutta ja lisäävät kompleksisuutta. Tehokkain on nolla-arkkitehtuuri, koska siinä on minimimäärä luokkia, rajapintoja ja yhteyksiä.

Jokaisen kierroksen jälkeen yksilöt järjestetään fitnessinsä mukaan. Parhaimman arvon saaneet siirretään suoraan seuraavaan sukupolveen. Muut valitaan rulettivalinnalla, jossa kunkin yksilön sektori määriytyy sen järjestysluvun mukaan populaatiossa. Järjestyslukua käytetään siksi, että fitness-arvojen erot eivät välttämättä kerro niiden todellisesta paremmuudesta.

5.3 Muunnelmia ja laajennuksia

Normaalissa arkkitehtuurisuunnittelussa on epätavallista, että suunniteltaisiin useita keskenään kilpailevia arkkitehtuureja ja kokeiltaisiin satunnaisesti, miten niiden osat sopivat keskenään. Tavallinen lähtökohta on, että suunnitellaan arkkitehtuuri, jota sitten kehitetään, kunnes se täyttää annetut laatuvaatimukset. Frankenstein-ohjelmassa tämä tapahtuu niin, että mutatoidaan arkkitehtuuria. Koska tässä yksilöitä ei risteytetä, puhutaan *aseksuaalisesta* evoluutiosta. Aseksuaalinen evoluutio on sama kuin aikaisemmin esitelty *evoluutiostrategia.*

Risteytystä ei tapahdu, mutta luonnonvalinnassa valitaan parhaat yksilöt. Yksilöitä pitää olla enemmän kuin valittavia yksilöitä, että niitä riittää myös hylättäväksi. Koska ilman risteytystä ei ole lapsia, tuotetaan jokaisen sukupolvikierroksen aluksi kustakin yksilöstä kolme kloonia mutatoitavaksi. Sukupolvi on nyt kolme kertaa valittavien yksilöiden joukon kokoinen. [Räihä, 2011, 77-78]

Kokeiluissa aseksuaalinen evoluutio toimi paljon nopeammin kuin risteytykseen perustuva, mutta se saavutti paikallisen optimin nopeasti. Noin sadan ensimmäisen sukupolven jälkeen kehitystä ei ollut enää havaittavissa. Voidaan olettaa, että vaikka aseksuaalinen evoluutio oli nopeaa, se jäi nopeasti paikalliseen optimiin käymättä läpi koko ongelma-avaruutta. Kun tutkittiin saatuja tuloksia, aseksuaalinen evoluutio suosi tiettyjä ratkaisuja toisten kustannuksella. [Räihä, 2011, 78]

Toinen vaihtoehto aiemmin käytetylle evoluutiolle on niin sanottu *komplementaarinen risteytys* (complementary crossover). Sen perusajatuksena on, että tietyt vanhemmat sopivat geneettisesti paremmin yhteen kuin toiset. Käytännön arkkitehtuurityössä sitä voisi verrata tilanteeseen, jossa kaksi arkkitehtia vertailee tuottamiaan arkkitehtuureita, ja he valitsevat niistä parhaat osat uuteen yhteiseen arkkitehtuuriinsa.

Populaatio järjestetään kahteen ryhmään, missä kummankin ryhmän jäsenille on omat laatuvaatimuksensa. Ryhmät voidaan nähdä sukupuolina. Toinen ryhmä siis käsittää seuraavan sukupolven isät ja toinen sen äidit. Frankenstein-järjestelmässä muunneltavat yksilöt nimitetään äideiksi ja tehokkaat isiksi. Yksilöt pariutetaan kuten aiemminkin, mutta sillä erotuksella, että toisen vanhemman pitää kuulua isien ja toisen äitien ryhmään. [Räihä, 2011, 81-82]

Risteytyksessä geenit katkaistiin ensin satunnaisesta kohdasta. Kehittyneemmässä menetelmässä katkaisukohtaa ei haettu satunnaisesti, vaan etsittiin isän geenin tehokkain kohta ja äidin geenin muunneltavin kohta ja katkaistiin geeni näiden välisestä satunnaisesta kohdasta. [Räihä, 2011, 82-83]

Kokeiluissa komplementaarinen risteytys suosi voimakkaasti viestinvälitysarkkitehtuuria. Komplementaarinen risteytys toimi paremmin kuin tavallinen risteytys, mutta ratkaisuja tutkittaessa huomattiin, että tämä johtui viestinvälitysarkkitehtuurin runsaasta käytöstä. Viestinvälitysarkkitehtuuri parantaa arkkitehtuurin muokattavuutta, ja se nosti saatujen arkkitehtuurien fitness-arvoa. [Räihä, 2011, 83-85]

5.4 Skenaariot

Käytännössä skenaariot ovat suosituin tapa mitata arkkitehtuurin toimivuutta. Frankensteiniin on tuotettu kahdenlaisia skenaarioita, *muutosskenaarioita* (changing scenarios) ja *lisäysskenaarioita* (adding scenarios). Lisäksi skenaariot voi jakaa *staattisiin* ja *dynaamisiin.* Dynaaminen skenaario tapahtuu ohjelman käytön aikana, eikä koodiin tehdä muutosta. Se on ikään kuin skenaario käyttäjän näkökulmasta. Staattisessa skenaariossa voidaan tehdä pieniä muutoksia ohjelmakoodiin. Se edustaa skenaariota ohjelmoijan näkökulmasta. Skenaariot voivat koskea implementaatiota eli toteutusta (implementation) tai semantiikkaa. Toteutusta koskevat skenaariot ovat yleisempiä, ja niissä operaation toteutusta muutetaan. Semanttisissa skenaarioissa operaatio muutetaan joksikin aivan toisenlaiseksi. Semanttiset skenaariot ovat harvinaisempia kuin toteutusta koskevat. Jokaiselle skenaariolle annetaan lisäksi todennäköisyys. [Räihä, 2011, 88]

Skenaariot mitataan niin, että niihin soveltuvat suunnittelumallit pannaan paremmuusjärjestykseen. Mitattaessa ratkaisua sitä verrataan tähän listaan ja sille annetaan pisteitä sen mukaan, miten suositeltua mallia se on käyttänyt. Näin tuotetaan skenaariofitness-funktio

$$sf_s = \sum scenarioProbability * 100/scenarioPreference, \qquad (5.2)$$

missä *scenarioProbability* tarkoittaa skenaarion todennäköisyyttä ja *scenario-* $Preference$ käytetyn suunnittelumallin järjestysnumeroa. [Räihä, 2011, 89]

Molemmille arkkitehtuureille, siis sekä älykoti *ehome*lle että robottisotasimulaattori *robo*lle kehitettiin kolme skenaariota. Älykodin skenaarioita olivat: [Räihä, 2011, 89-90]

- Käyttäjä voi muuttaa musiikkilistan esitystapaa.
- Kehittäjä voi muuttaa tapaa, jolla vesijohto on yhdistetty kahvinkeittimeen.
- Kehittäjä voi luoda uuden tavan näyttää kahvinkeittimen tila.

Näistä ensimmäinen on dynaaminen, eli ohjelman toiminnan aikana tapahtuva, muutoskenaario. Se toteutetaan muuttamalla ohjelman implementaatiota. Toinen

on staattinen eli ohjelman kehittäjän kohtaama muutosskenaario, joka muuttaa komponentin semantiikkaa. Kolmas on staattinen lisäysskenaario, joka koskee ohjelman implementaatiota.

Robottisodassa kolme skenaariota olivat [Räihä, 2011, 90]

- pelaaja voi muuttaa robotin ulkonäköä,
- kehittäjä voi lisätä uuden robottien suojausjärjestelmän ja
- kehittäjä voi muuttaa robottien älyn toteutusta.

Ensimmäinen skenaario on dynaaminen implementaatiota koskeva muutosskenaario, toinen staattinen implementaatiota koskeva lisäysskenaario ja kolmas staattinen muutosskenaario.

Kokeilujen mukaan skenaariot estävät algoritmia päätymästä liian nopeasti paikalliseen optimiin. Niitä käyttämällä pystytään ainakin jossain määrin tuottamaan parempia arkkitehtuureja. [Räihä, 2011, 90-92]

5.5 Monitavoiteoptimointi Frankenstein-järjestelmässä

Frankenstein-järjestelmän viidestä funktiosta muodostetaan kaksi funktiota, joita yritetään optimoida yhtä aikaa. Muokattavuutta ilmaisevista funktioista sf_1 ja sf_2 kootaan yksi muokattavuusfunktio $mf(x) = sf_1(x) - sf_2(x)$ ja tehokkuutta ilmaisevista funktiosta sf_3 ja sf_4 tehokkuusfunktio $ef(x) = sf_3(x) - sf_4(x)$.

Pareto-rintama muodostetaan järjestämällä yksilöt muokattavuutensa mukaan. Muokattavuusarvoltaan korkein yksilö pf_1 otetaan rintamaan. Sen jälkeen käydään yksilöt läpi muokattavuusjärjestyksessä, eli kun $mf(pf_{n-1}) > mf(pf_n)$, jos $ef(pf_n) > ef(pf_{n-1})$, yksilö pf_n otetaan mukaan Pareto-rintamaan. Koska rintamaan valikoidaan vain osa kunkin sukupolven yksilöistä, evoluutiota jatketaan lisäämällä jokaisen sukupolven Pareto-optimaaliset ratkaisut rintamaan, jos ne eivät ole jo siellä. Tätä jatketaan, kunnes rintamassa on sata yksilöä. [Räihä, 2011, 97-98]

Kokeiluissa rintamat painottuivat aluksi tehokkuudeltaan korkeisiin ratkaisuihin. Se oli odotettavissa, sillä nolla-arkkitehtuuri on tavallisesti tehokkain ja suunnittelumallien soveltaminen laskee tehokkuutta muokattavuuden hyväksi. Rintama

on myös voimakkaammin klusteroitunut tehokkuuden maksimoivassa osassa, kun taas korkean muokattavuuden päässä ratkaisut olivat enemmän hajallaan. Se on sikäli ymmärrettävää, että muokattavuutta voi parantaa useammalla eri menetelmällä. Ratkaisuja tutkittaessa huomattiin, että muokattavimmissa ratkaisuissa käytetiin viestinvälitysarkkitehtuuria, jota ei puolestaan esiintynyt tehokkaimmissa ratkaisuissa. [Räihä et al., 2011]

Järjestelmän suoriutumista monitavoiteongelmista tutkittiin ATAM-menetelmällä, jossa luotiin käyttötapauksia ja arvosteltiin arkkitehtuurien menestystä niistä selviämisessä. Tehokkuuskenaarioissa oli tyypillisesti mukana osia, jotka ovat kriittisiä järjestelmän tehokkuuden kannalta. Kun esimerkiksi älykodin arkkitehtuurissa on viisi osajärjestelmää, luotiin käyttötapaukset kullekin näistä osajärjestelmistä ja mitattiin niissä ajankäyttöä suhteessa muihin arkkitehtuureihin. Ajankäyttö mitattiin luomalla mittariksi negatiivinen tehokkuus (efficience penalty), joka laskettiin laskemalla yhteen kutsut luokkien välillä, kutsut viestinvälittäjän kautta kerrottuna kertoimella d, sekä palvelinkutsut kerrottuna kertoimella s ja ottamalla tämän summan vastaluku. [Räihä et al., 2011]

Koska muokattavuuden mittaaminen skenaarioilla ei ole yhtä suoraviivaista, tilattiin ulkopuolisilta asiantuntijoilta 12 skenaariota. Skenaarioista selviytyminen pisteytettiin niin, että

- jos koodia piti muuttaa, annettiin nolla pistettä,
- jos arkkitehtuuri tuki pikemminkin kehitysaikaista kuin ajoaikaista muutosta, annettiin yksi piste tai
- jos olemassaoleva koodi tuki muutosta, annettiin kaksi pistettä. [Räihä et al., 2011]

Nämä saadut pisteet kerrottiin todennäköisyydellä, jolla kyseinen skenaario toteutuu. Näin saatuja tuloksia verrattiin Frankenstein-järjestelmän tuottamaan Pareto-rintamaan niin, että tutkittiin saiko parhaat muokattavuuspisteet saanut ratkaisu parhaan tuloksen myös ATAM-muokattavuusskenaarioissa, ja tehokkuuden kohdalla samalla tavalla. Sekä tehokkuudessa että muokattavuudessa sijoituksella Pareto-rintamassa ja saaduilla ATAM-pisteillä oli selvä yhteys. Tehokkuudessa korrelaatiokerroin oli 0.79 ja muokattavuudessa -0.62. Kun tuloksia verrattiin ohjelmistoarkkitehtuurin kurssin opiskelijoiden tuottamiin arkkitehtuurei-

hin, huomattiin että annettuihin kriteereihin suhteutettuna ohjelmiston tuottaman arkkitehtuurin laatu on suunnilleen sama kuin kolmannen vuoden yliopisto-opiskelijan. Kirjoittajat kuitenkin tähdentävät, että näitä mittareita oli verraten vähän ja ongelma sellainen, että se oli käytännössä melko suoraviivaista ratkaista viestinvälitysarkkitehtuurilla. [Räihä et al., 2011]

6 Johtopäätökset

Esittelin aluksi heuristisia menetelmiä monitavoiteoptimoinnin kannalta. On johdettu yleinen tulos, ettei ole olemassa universaalia menetelmää, jolla heuristinen optimointi toimisi tehokkaammin kuin satunnainen haku. Käytännön elämässä tämä ei kuitenkaan päde, vaan toiset menetelmät ovat tehokkaampia kuin toiset. Menetelmien tehokkuus riippuu kuitenkin paljon sovellusalueesta.

Tietokoneohjelman onnistunut arkkitehtuurivalinta vaikuttaa paitsi sen tehokkuuteen, myös erityisesti sen ylläpidettävyyteen, toteuttamisen helppouteen ja muutettavuuteen. Arkkitehtuurivalintaa on pidetty kokeneiden asiantuntijoiden työnä. Frankenstein-projektissa tutkittiin, miten heuristinen haku menestyy arkkitehtuurin suunnittelussa. Voidaanko ohjelmistojen arkkitehtuurin suunnittelu automatisoida.

Frankenstein-ohjelmaa tutkittiin sovellettuna kahteen arkkitehtuuriin, jotka olivat älykoti *eHome* ja robottisotasimulaattori *robo*. Tutkimuksissa selvisi, että ohjelma saavuttaa näiden arkkitehtuurien luonnissa tason, joka vastaa noin kolmannen vuoden ohjelmistotekniikan opiskelijan tasoa.

Kyseiset arkkitehtuurit eivät kuitenkaan välttämättä edusta kaikkia mahdollisia arkkitehtuureja, ja ne näyttävät ratkeavan parhaiten käyttämällä tiettyjä perusmalleja. Ohjelman menestyksestä arkkitehtuureissa, joille tämä ei päde, ei voida vielä tutkitun perusteella sanoa mitään.

Jatkotutkimuksena Frankensteinia voisi kokeilla soveltaa joihinkin aivan uusiin arkkitehtuuriongelmiin. Näin voitaisiin selvittää, onko se yleinen menetelmä arkkitehtuurien luomiseen vai onko se sattumalta optimoitu menestymään juuri näissä kahdessa arkkitehtuurissa tai yleensä näiden arkkitehtuurien kaltaisissa ongelmissa.

Tekoälyjärjestelmien luonnissa on aina vaarana *ylisovittaa* järjestelmä menestymään annetulla harjoitteluaineistolla niin, että sen toimivuus yleisesti alkaa jopa heiketä, vaikka menestys annetuilla kriteereillä eli käytetyllä aineistolla nousisikin.

Monitavoiteoptimoinnin kannalta Frankenstein edustaa vain pientä osaa mahdollisuuksista, joihin monitavoiteoptimointia voi soveltaa. Erilaisia optimoitavia funktioita on Frankensteinissä vain kaksi tai kolme. Näiden suhteen on helppo tutkia ohjelman toimivuutta, mutta yleisesti monitavoiteoptimoinnin ongelmat

ovat tuotettujen ratkaisujen suuri määrä ja Pareto-rintaman kelvollisuuden vaikea mittaaminen, jotka eivät tulleet tässä esiin.

Optimoitavia funktioita voitaisiin luoda enemmänkin, mutta niiden määrittely ja mittaaminen eivät välttämättä ole helppoja tehtäviä. Lisäksi mitä useampia funktioita tulee optimoitavaksi, sitä vaikeampi on mitata ja arvostella luotuja ratkaisuja. Monitavoiteoptimoinnin teoriassa tähän on kuitenkin työkaluja.

Teoreettisesti tarkasteltuna monitavoiteoptimoinnille on ominaista, että erilaisia menetelmiä on suuri määrä, ja ne menestyvät hyvin eri lailla eri ongelmiin sovellettuina. Erilaisista menetelmistäkin on paljon jatkokehitelmiä, jotka usein kehitetään hyvin monimutkaisiksi. Esimerkiksi esittelemistäni geneettisistä monitavoiteoptimointialgoritmeista on kehitetty uusia huomattavasti vaikeampia versioita. Näiden käyttö ja tutkimus vaatisi kuitenkin ongelmia, joissa saadaan paljon tuloksia ja lukuarvoja mittaamaan menetelmän onnistumista.

Ohjelmistoarkkitehtuurin optimointi Frankenstein-ohjelmalla ei ole tällainen, eivätkä esittelemäni monitavoiteoptimoinnin menetelmät ole ainakaan pidemmälle vietyinä versioina sen tutkimiseen kovinkaan hedelmällisiä. Sen sijaan jatkotutkimuksissa voitaisiin soveltaa luvun 3 menetelmiä perusmuodossaan. Esimerkiksi tutkia järjestelmällisesti, miten ne menestyvät tässä ongelmassa. Näin ollaan jo alustavasti tehtykin, ja Frankensteinia on tutkittu muutenkin kuin vain eri tavoitteet summaavilla menetelmillä. Jatkotutkimuksissa voitaisiin koota listaan kaikki riittävän yksinkertaiset optimointimenetelmät ja tutkia niiden menestystä.

Yleisesti voidaan todeta, ettei tämän tutkielman teoreettista osuutta päästy soveltamaan koko laajuudessa annettuun ongelmaan. Soveltaminen vaatisi käytännössä menetelmien ohjelmoimista järjestelmään ja niiden menestyksen tutkimista. Erityisesti toteutuksen hionta toimimaan hyvin annetuissa ongelmissa ja tulosten tutkiminen ja mittaaminen olisivat vaatineet paljon työtä. Tutkielmaa ei olisi kuitenkaan voinut keventää alkupäästä, koska uuden menetelmän ottaminen tutkimuksen alle vaatisi yleiskatsauksen mahdollisiin menetelmiin ja siihen, mitä niistä tiedetään.

Viiteluettelo

[Bandyopadhyay et al., 2008] Sangamitra Bandyopadhyay, Sripana Saha, Ujjwal Maulik & Kalyanmoy Deb. A Simulated Annealing-Based Multiobjective Optimization Algorithm: AMOSA. *IEEE Transactions on Evolutionary Computation* 12, 3 (June 2008) 269-283.

[Coello, 2009] Carlos A. Coello Coello. An Updated Survey of GA-Based Multiobjective Optimization Techniques. *ACM Computing Surveys*, 32, 2 (June 2000) 109-143.

[Gamma et al, 1994] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides *Design patterns : elements of reusable object-oriented software* Addison-Wesley, 1994.

[Ho & Pepyne, 2001] Yu-Chi Ho & David L. Pepyne. Simple Explanation of the No Free Lunch Theorem of Optimization *In Proceedings of the 40th Conference on Decision and Control.* Orlando, Florida USA, December 2001.

[Knowles & Corne, 1999] Joshua Knowles & David Corne. The Pareto archived evolution strategy: a new baseline algorithm for Pareto multiobjective optimisation. *CEC 99. Proceedings of the 1999 Congress on Evolutionary Computation* 98-105.

[Knowles & Corne, 2001] Joshua Knowles & David Corne. On Metrics for Comparing Non-Dominated Sets. In Congress on Evolutionary Computation (CEC 2002). 2001.

[Koskimies & Mikkonen, 2005] Kai Koskimies & Tommi Mikkonen. *Ohjelmistoarkkitehtuurit.* Talentum, 2005.

[Lucas, 2006] Chris Lucas. *Practical Multiobjective Optimisation.* Available at url: http://www.calresco.org/lucas/pmo.htm. Checked 29.4.2014.

[Luke, 2009] Sean Luke. *Essentials of Metaheuristics.* Lulu, Available at url: http://cs.gmu.edu/ sean/book/metaheuristics/ , 2009. Checked 1.5.2014.

[Reeves, 2002] Colin Reeves, R. Rowe & E. Jonathan. *Genetic Algorithms - Principles and Perspectives: A Guide to GA Theory*. Kluwer Academic Publishers, Secaucus, NJ, USA

[Räihä, 2011] Outi Räihä. *Genetic Algorithms in Software Architecture Synthesis*. Ph.D. thesis, University of Tampere, 2011.

[Räihä et al., 2011] Outi Räihä, Kai Koskimies & Erkki Mäkinen. Generating Software Architecture Spectrum with Multi-Objective Genetic Algorithms. In: Proc. of the Third World Congress on Nature and Biologically Inspired Computing (NaBIC'11). Salamanca, Spain. October 2011, IEEE Press, 29-36.

[Zitzler et al., 2001] Eckart Zitzler, Marco Laumanns & Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. TIK-Report 103, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH), Zurich, May 2001.