RAJOITELASKENTA PYTHONILLA

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Tietojenkäsittelyn koulutus, Hämeenlinnan korkeakoulukeskus
kevät, 2021

Jesse Kopra

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutus — Tiivistelmä

Hämeenlinnan korkeakoulukeskus

Tekijä	Jesse Kopra	Vuosi 2021
Työn nimi	Rajoitelaskenta Pythonilla	
Ohjaajat	Lasse Seppänen	

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli tutkia tekoälyä, rajoitelaskentaa ja sen soveltuvuutta opetukseen. Työssä esitellään kolme eri tapaa ratkaista rajoitelaskentaongelmia käyttäen Python-ohjelmointikieltä. Opinnäytetyön toimeksiantaja oli Hämeenlinnan korkeakoulukeskuksen tietojenkäsittelyn opettaja Tommi Lahti, joka halusi lisätietoa siitä, miten rajoitelaskenta sopisi parhaiten opetettavaksi.

Tutkimuksellisen opinnäytetyön tietopohjana ovat tekoälyyn ja rajoitelaskentaan liittyvä kirjallisuus sekä työssä käytettyjen Python-kirjastojen tarjoamat dokumentaatiot. Rajoitelaskennan tekeminen esitellään kolmella eri tavalla, yksi käyttämällä klassisia Python-algoritmeja sekä kaksi aiheeseen tarkoitettua Python-kirjastoa. Rajoitelaskentaratkaisuja vertaillaan läpikäymällä saman ongelman ratkaiseminen eri tavoin.

Työn johtopäätöksenä todetaan, että Python-algoritmien käyttäminen on opettavaisin tapa, jos pääpaino opetuksessa on rajoitelaskennalla. Työssä esitellyt kaksi kirjastoa, pyDatalog ja PyCSP^3, tarjoavat tehokkaat ja helposti opittavat ratkaisumenetelmät, joiden käyttöä opetuksessa voidaan harkita, jos opetuksessa keskitytään tekoälyyn yleisemmällä tasolla.

Avainsanat	Tekoäly, rajoitelaskenta, Python
Sivut	32 sivua ja liitteitä 1 sivu

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology
Hämeenlinna University Centre

Abstract

Author Jesse Kopra
Year 2021
Subject Constraint satisfaction using Python
Supervisors Lasse Seppänen

ABSTRACT

The purpose of the thesis was to analyze artificial intelligence, constraint satisfaction and how it could be best taught. The thesis demonstrates three ways to solve constraint satisfaction problems using the Python programming language. The thesis was commissioned by Tommi Lahti, a Principal lecturer of Business Information Technology at HAMK, who wanted analysis about how to best teach constraint satisfaction.

The thesis is a literature review based on literature and lectures about artificial intelligence and constraint satisfaction in addition to the documentation and guides relating to the Python libraries used. The thesis shows and compares three different ways to solve a constraint satisfaction problem. One of the ways is to use classic Python algorithms, while the other two are Python libraries designed to solve constraint satisfaction problems.

Based on the analysis, it is recommended that this student-made library is the most educational, especially if constraint satisfaction is the focus of the study. The two other libraries, pyDatalog ja PyCSP^3, could also be useful in more general artificial intelligence studies, because they offer powerful and easy to learn ways to solve constraint satisfaction problems.

Keywords Artificial intelligence, constraint satisfaction, Python.
Pages 32 pages and appendices 1 page

Sisällys

Kuvat, ohjelmakoodit ja taulukot (nämä pitää päivittää kuten sisällyskin, poista tarpeettomat)

Liitteet

1 Johdanto

Tekoäly on monille tuttu sci-fi-elokuvien ihmisen kaltaisista roboteista, mutta harva ymmärtää kuinka paljon tekoälyä hyödynnetään jo tänä päivänä. Tietokoneiden nopean kehityksen myötä on pystytty aloittamaan todellisen koneälyn suunnittelu. Nykypäivän älykkäät koneet voivat rajallisesti toimia ja oppia kuten ihminen, ja niitä käytetään jo esimerkiksi teollisuudessa ratkomaan erilaisia ongelmia. Tässä opinnäytetyössä keskitytään yhteen näistä, rajoitelaskentaongelmiin, joita monet yleiset tosielämän ongelmat ovat. Rajoitelaskennalla etsitään joukolle muuttujia arvoja, jotka toteutuvat annettujen rajoituksien mukaan. Esimerkiksi tehokkaan aikataulutuksen tai työnjaon selvittäminen voidaan käsittää rajoitelaskentaongelmina.

Tekoälystä ja rajoitelaskennasta oli kiinnostunut myös opettajani Tommi Lahti, joka tämän aiheen minulle antoi. Hän halusi lisätietoja aiheesta, ja siitä miten sitä voisi käytännössä toteuttaa Python-ohjelmointikielellä. Päätimme että tähän työhön kuuluisi myös erilaisten toteutustapojen vertailu, mihin kuuluu selvitys, miten sama ongelma ratkaistaan eri tavoilla ja miten eri tavat soveltuisivat opetettavaksi. Hän suositteli aiheeseen liittyviä lukuja Stuart Russellin ja Peter Norvigin kirjasta Artificial Intelligence: A Modern Approach, jonka huomasin tiedonkeruuvaiheessa olevan erittäin monen käyttämä päälähde.

Minua tämä aihe kiinnosti, koska uskon, että tekoälyn rooli tulee kasvamaan valtavasti tulevan urani aikana. Halusin oppia aiheesta lisää, koska aihetta ei juurikaan ole käsitelty kursseilla. Käytännön käsitystä tekoälystä minulla ei juurikaan ollut työni aloittaessa, ja rajoitelaskennasta en ollut ikinä kuullut.

Opinnäytetyön tutkimuskysymykset ovat:

- Mitä on tekoäly ja miten se toimii?
- Mitä on rajoitelaskenta?
- Miten sitä käsitellään Pythonilla?
- Miten sitä olisi paras opettaa?

2 Tekoäly

Koska tekoäly on käsitteenä ja tutkimusalana laajakäsitteinen ja monimutkainen, esitellään siitä tässä luvussa opinnäytetyölle olennaisimmat käsitteet. Lyhyen historiikin jälkeen käsitellään eri näkökulmia tekoälyyn, joista tässä työssä syvennytään yhteen, päätyen lopulta työlle tärkeään rajoitelaskentaan.

Yksinkertaistettuna termillä tekoäly tarkoitetaan koneen kykyä ajatella ja oppia. Tässä luvussa käydään läpi eri tapoja, miten koneita on tehty toimimaan ja ajattelemaan. Tekoäly jaetaan käsitteenä yleensä kahteen, heikkoon ja vahvaan tekoälyyn. Näillä termeillä ilmaistaan tekoälyn itsenäisyyttä. Heikko tekoäly toimii aina sille annettujen käskyjen mukaan, kuten shakkiohjelma, joka osaa valita aina optimaalisen siirron siihen ohjelmoidun logiikan mukaan. Vahva tekoäly, jota ei ole vielä onnistuttu luomaan, kykenee ihmisen tavoin itsenäiseen ajatteluun ja tietoisuuteen. (Tekoälyinfo, n.d.)

2.1 Tekoälyn historia

Tekoäly ideana esiintyy ensi kertaa länsimaisessa muinaishistoriassa. Antiikin kreikan filosofit ja kirjailijat käsittelivät tekoälyä ja älyllisiä koneita jo kauan ennen, kun niitä pystyttiin toteuttamaan käytännössä. Homeros kuvailee jo vuonna 850 eaa. Iliaksessa kuinka Hefaistos, tulen ja taonnan seppäjumala, luo itselleen erilaisia ihmisen kaltaisia automaatteja, kuten avustajia ja vartijoita. (McCorduck, 2004, s. 4)

Kun ensimmäiset tietokoneet keksittiin vuonna 1940, tuli oikean koneellisen älyn luonnista huomattavasti saavutettavampaa. Kenties ensimmäinen tekoälyn luomista tietokoneella käsitellyt oli brittiläinen Alan Turing, joka julkaisi vuonna 1950 Computing Machinery and Intelligence -nimisen artikkelin, jossa hän kysyi voisiko kone ajatella ja ehdottaa testiä, jolla määriteltäisiin koneen älykkyys. Tästä testistä kerrotaan lisää luvussa 2.2.1. (Warwick, 2012, s. 2)

Ehkä tärkein tapahtuma tekoälyn alkuhistoriassa on vuonna 1956 Dartmouthin yliopistolla pidetty työpaja, Dartmouth Summer Research Project on Artificial Intelligence, johon kerääntyi tutkijoita ja asiantuntijoita useilta eri tieteen aloilta, kuten matematiikka,

sähkötekniikka ja psykologia. Dartmouthin konferenssiksikin kutsutussa työpajassa luotiin perusta ja tutkimuksen suuntaus tulevalle tekoälytutkimukselle. Yksi työpajan perustajista, John McCarthy, oli ensimmäinen, joka käytti termiä tekoäly (artificial intelligence) nimetessään tämän tutkimusalan. (Moor, 2006, s. 87)

Dartmouthin konferenssin läsnäolijoista monesta tuli tekoälytutkimuksen johtavia tutkijoita, jotka saivat projekteihinsa valtavasti rahoitusta. Ajattelevan koneen luominen oli kuitenkin vaikeampaa kuin he odottivat ja vuoteen 1973 mennessä rahoitus oli romahtanut. Tästä seurasi niin kutsuttu ”tekoälytalvi”, jolloin tutkimusalan kiinnostus ja rahoitus oli erittäin alhaista. 1980-luvun alussa yleinen kiinnostus palasi, ja tekoälytutkimukseen sijoitettiin miljardeja, ennen kuin sijoittajat kuitenkin pettyivät lopputuloksiin. Tekoälytutkimus oli tämän jälkeen vähäistä, ennen kuin 2000-luvulla ala lähti taas räjähdysmäiseen nousuun, kun tekoäly saatiin vihdoin sovellettua käytännössä. (Tekoälyinfo, n.d.)

2.2 Tekoälyn määritelmä

Tekoälylle ei ole yhtä oikeaksi sovittua määritelmää, vaan eri ihmiset omine näkökulmineen ovat ajatelleet siitä eri tavoin. Taulukossa 1 esitellään kahdeksan eri lähestymistapaa tekoälyyn, jotka eroavat vastauksessaan kahteen tärkeään kysymykseen: Kumpi on oleellisempaa, ajattelu vai toiminta? Käytetäänkö mallina ihmistä vai pyritäänkö ideaaliin standardiin? (Russell & Norvig, 2010, s. 1)

Ylemmät näkulmat keskittyvät tekoälyn sisäiseen toimintaan ja ajattelukykyyn, alemmat toimintaan ja käyttäytymiseen. Vasemmalla olevat tavat pyrkivät jäljittelemään ihmistä, oikealla olevat pyrkivät rationaalisuuteen ja ”ideaaliin” ratkaisuun. (Russell & Norvig, 2010, s. 2)

Taulukko 1 Tekoälyn neljä näkökulmaa (Russell & Norvig, 2010, s. 2)

Ihmismäinen ajattelu	**Rationaalinen ajattelu**
"The exciting new effort to make computers think ... machines with minds, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that makeit possible to perceive, reason, and act." (Winston, 1992)
Ihmismäinen toiminta	**Rationaalinen toiminta**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole et al., 1998) "AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

2.2.1 Ihmismäinen toiminta

Kenties tunnetuin tekoälytieteiden perustajista, Alan Turing, esitti vuonna 1950 matkimispeliksi (engl. imitation game) kutsumansa testin tekoälyn älykkyyden mittaamiseksi. Tässä testissä ihmistestaaja esittää kirjoitettuja kysymyksiä tietokoneelle ja jos ihminen ei pysty vastausten perusteella päättelemään onko vastaaja kone vai ihminen, katsotaan koneen olevan oikeasti älykäs. Turingin testi vaatii koneelta luonnollisen kielen ymmärtämistä, tietämyksen esittämistä, sekä päättely- ja oppimiskykyä. Alkuperäinen testi välttää suoraa kontaktia koneen ja testaajan välillä, mutta laajennettu totaalinen Turingin testi, jossa hyödynnetään videosignaalia, vaatii koneelta myös konenäköä ja robotiikkaa. (Hyvönen, 2004)

Turingin testi kattaa monia tekoälyn ominaisuuksia, mutta sen läpäiseminen ei ole merkittävä tavoite nykypäivän tekoälytutkijoille. Monet pitävät tärkeämpänä tutkia älykkyyden perusperiaatteita kuin toistaa ihmisen käyttäytymistä. Russell ja Norvig (2010) esittävät esimerkin Wright-veljeksistä, jotka keksivät ensimmäisen lentokoneen, kun he lopettivat lintujen matkimisen ja keskittyivät aerodynamiikkaan.

2.2.2 Ihmismäinen ajattelu

Toisen näkökulman lähtökohtana on ihmisen ajattelun mallintaminen, josta nousee väistämättä kysymys: miten ihminen ajattelee? Tapoja vastata kysymykseen on monia, kuten introspektio, psykologiset kokeet ja aivojen kuvantaminen. Ajatuksena on, että tarpeeksi kattavalla teorialla ihmisen ajatusmallia voitaisiin soveltaa koneellisen tekoälyn luonnissa. Ihmisaivojen toiminnan täydellinen mallintaminen on kuitenkin nykymenetelmin mahdotonta, joten on pakko käyttää abstraktiota. (Russell & Norvig, 2010, s. 3)

Tämä tekoälyn näkökulma on tärkeä osa monialaista kognitiotiedettä, jossa sitä käytetään yhdessä mm. psykologian, filosofian ja neurotieteen kanssa mallintamaan ajattelua ja tietoilmiöitä kuten havaitsemista, oppimista ja muistamista. (Boden, 2006, s. 282)

2.2.3 Rationaalinen ajattelu

Rationaalisen ajattelun isänä voidaan pitää antiikin Kreikan filosofia Aristotelesta, joka yritti syllogismeillaan systematisoida ”oikeaa” ajattelutapaa. Syllogismeissa kahden todeksi ymmärretyn lauseen avulla päätellään looginen johtopäätös. Klassinen esimerkki on ”Sokrates on ihminen; kaikki ihmiset ovat kuolevaisia; siis Sokrates on kuolevainen.” 1800-luvun loogikot loivat tarkemman merkintätavan ja vuoteen 1965 mennessä oli saatu luotua ohjelmia, jotka pystyivät teoriassa ratkomaan minkä tahansa ratkottavissa olevan ongelman. (Russell & Norvig, 2010, s. 4)

Perusteelliset ongelmat kuitenkin nousevat esiin, kun tätä menettelytapaa sovelletaan käytännössä tietokoneilla. Epämuodollisen, puutteellisen tai epävarman tiedon muuntaminen loogiseen muotoon voi olla erittäin vaikeaa. Jotkut päättelyt voivat toimia

teoriassa, mutta tarvitsevat liikaa laskentavoimaa ollakseen realistisia ratkaisuja, ilman että konetta ohjataan päättelyn suuntaamisessa. (Russell & Norvig, 2010, s. 4)

2.2.4 Rationaalinen toiminta

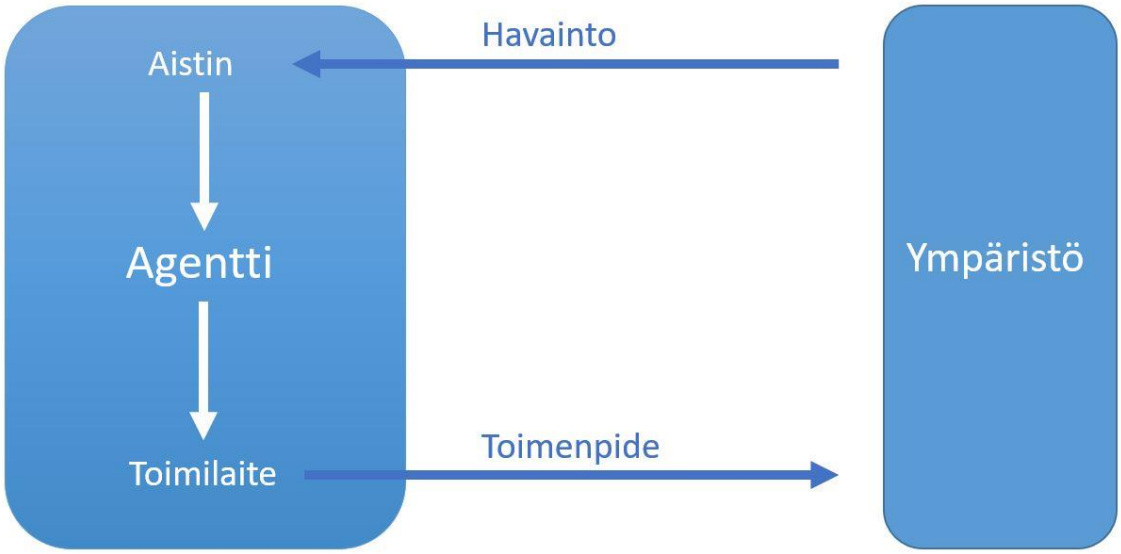
Rationaalisen toiminnan näkökulmaan sisältyy paljon rationaalisen ajattelun piirteitä, mutta se tähtää toimimaan myös tilanteissa, joissa loogiseen ratkaisuun ei ole mahdollista päästä. Rationaaliseen toimintaan kuuluvat loogiseen päättelyyn perustuva teot, mutta useissa tilanteissa ei ole yhtä tai ainuttakaan oikeaa ratkaisua, mutta jotain pitäisi tehdä. Etenkin robotiikassa on myös tärkeää jättää mahdollisuus refleksiiviselle, mahdollisesti ei-loogiselle toiminnalle. Esimerkiksi koskiessa kuumaan lieteen, usein parempi vaihtoehto on refleksiivisesti irrottaa ote nopeasti liedestä kuin harkita loogisesti useita vaihtoehtoja. (Russell & Norvig, 2010, s. 4)

Tähän näkökulmaan kuuluu ajatus tekoälystä rationaalisena agenttina. Tässä kontekstissa agentti tarkoittaa jotain, joka tekee jotain. Kaikki tietokoneohjelmat tekevät jotain, joten tietokoneagentilla tarkoitetaan koneita, jotka toimivat omatoimisesti, havaitsevat ympäristöään, reagoivat muutoksiin sekä luovat itselleen tavoitteita. Rationaalinen agentti toimii saadakseen parhaan lopputuloksen omaavansa tiedon perusteella. Tämä opinnäytetyö ja sen perusteena oleva materiaali perustuu näihin rationaalisiin agentteihin. (Russell & Norvig, 2010, s. 4)

2.3 Älykkäät agentit

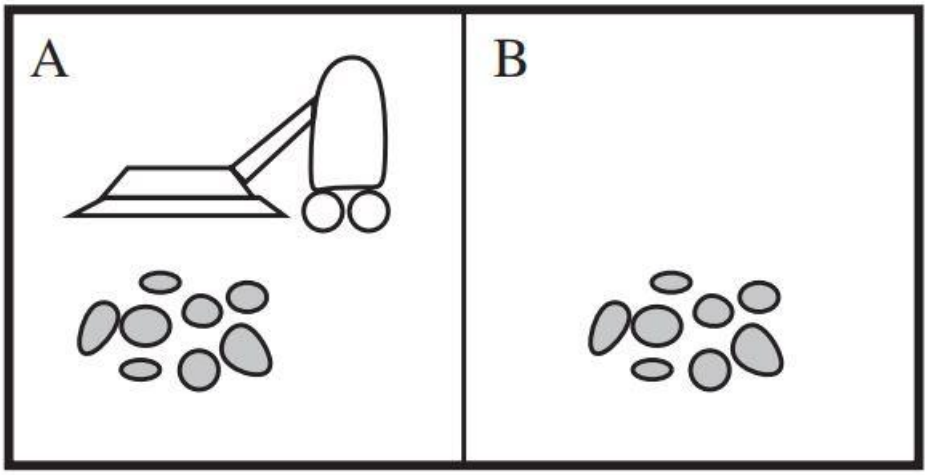
Agentit, kuten esimerkiksi ihmiset, robotit tai älyvalaisimet, havaitsevat **ympäristöään aistimilla** ja vaikuttavat siihen **toimilaitteilla**. Ihminen havaitsee monilla aisteillaan, robotti voi saavuttaa saman esimerkiksi kameroin tai infrapuna-anturein. Ympäristöllä ei kuitenkaan tarkoiteta välttämättä fyysistä sijaintia; ohjelmistoagentin suorittamat tiedoston muokkaukset ovat yhtä lailla vaikuttamista ympäristöön kuin ihmisen käsin suorittamat teot. Tässä opinnäytetyössä käytämme termejä **havainto** ja **toimenpide** kuvaamaan agentin senhetkistä havaittua ympäristöä ja niiden perusteella suoritetuista teoista. Kuva 1 esittää agentin suhdetta ympäristöönsä. (Russell & Norvig, 2010, s. 34)

Kuva 1 Agentin suhde ympäristöön



Agentin suorittamat toimenpiteet perustuvat usein yksittäisen havainnon sijasta havaintohistoriaan, mikä käsittää kaikki agentin havainnot. Agentin funktio määrittelee tehtävät toimenpiteet tietyn havaintosarjan esiintyessä. Russell ja Norvig (2010) havainnollistavat tätä kuvan 2 esimerkillä. Siinä esitettävä imuri on agentti, joka saa ympäristöstään kaksi havaintoa: Onko imuri ruudussa A vai B ja onko ruutu likainen. Agentilla on neljä mahdollista toimenpidettä: Liikkua vasemmalle tai oikealle, siivota ruutu sekä olla tekemättä mitään. Imurin agenttifunktio (kuva 3) on tässä yksinkertainen: Jos ruutu on likainen, siivoa; muuten siirry toiseen ruutuun. Jos imuriagentti saa havaintosarjan ”A, Puhdas”, se liikkuu oikealle ruutuun B, mutta jos sarja on ”A, Likainen”, imuri siivoaa ruudun. (Russell & Norvig, 2010, s. 35)

Kuva 2 Imuriagentti, jolla on kaksi mahdollista sijaintia (Russell & Norvig, 2010 s. 36)



Kuva 3 Esimerkki kuva 2:n imuriagentin funktiosta (Russell & Norvig, 2010 s. 36)

Percept sequence	Action
$[A, Clean]$	$Right$
$[A, Dirty]$	$Suck$
$[B, Clean]$	$Left$
$[B, Dirty]$	$Suck$
$[A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Dirty]$	$Suck$
⋮	⋮
$[A, Clean], [A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Clean], [A, Dirty]$	$Suck$
⋮	⋮

Rationaalinen agentti tekee aina järkevimmän mahdollisen valinnan. Agentin järkevyyttä arvioidaan sen perusteella, kuinka haluttuja sen vaikutukset ympäristöönsä ovat. Tätä kutsutaan agentin **suoritusarvoksi**. Suoritusarvon kriteerinä on tärkeää olla ympäristö eikä agentti tai sen toiminta. Jos edellisen esimerkin imurin suoritusta arvioitaisiin sen suorittaman imuroinnin pohjalta, se voisi nostaa suoritusarvoaan päästämällä imuroimansa pölyn takaisin lattialle, jolloin sen ei tarvitsisi liikkua ruutujen välillä tai odottaa pölyn kerääntyvän. Toivotumman lopputuloksen saa suoritusarvolla, joka mittaa kuinka puhdas lattia on tietyin aikavälein. (Russell & Norvig, 2010, s. 36)

Agentin rationaalisuuden katsotaan riippuvan neljästä asiasta.

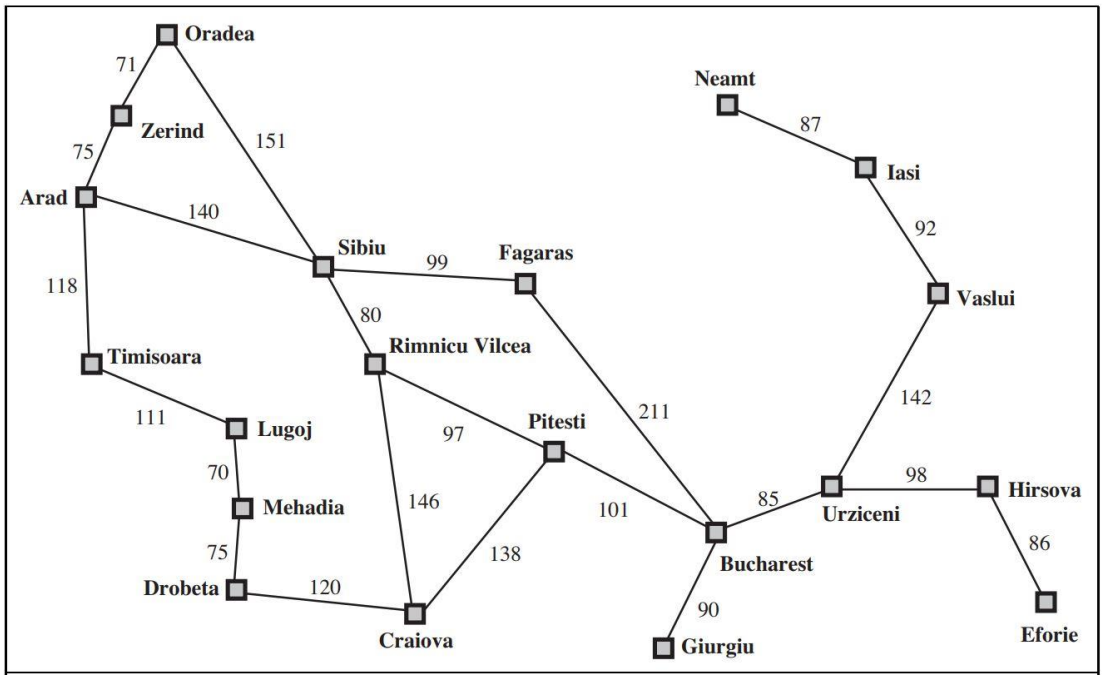
- Suoritusarvosta
- Agentin alustavasta tiedosta ympäristöstä
- Agentin mahdollisista toimenpiteistä
- Agentin havaintohistoriasta

Russell ja Norvig (2010) määrittelevät rationaalisen agentin valitsevan aina toimenpiteen, joka maksimoi suoritusarvon perustuen havaintohistoriaan ja alustavaan tietoon. Rationaalisuus ei kuitenkaan tarkoita kaikkitietävyyttä tai selvänäköisyyttä. Agentin havainnot voivat olla puutteellisia tai virheellisiä ja sen toimenpiteistä voi seurata yllättäviä lopputuloksia. Tämän takia rationaalisimmatkaan agentit eivät välttämättä pääse parhaaseen tulokseen. Rationaalisuus ei takaa täydellisyyttä, mutta sillä tavoitellaan suorituksen parasta mahdollista odotettua arvoa. Rationaalisen agentin tulee täten tutkia ympäristöään ja oppia havainnoimastaan, pystyen toimimaan itsenäisesti keräämänsä tiedon pohjalta.

2.4 Haku

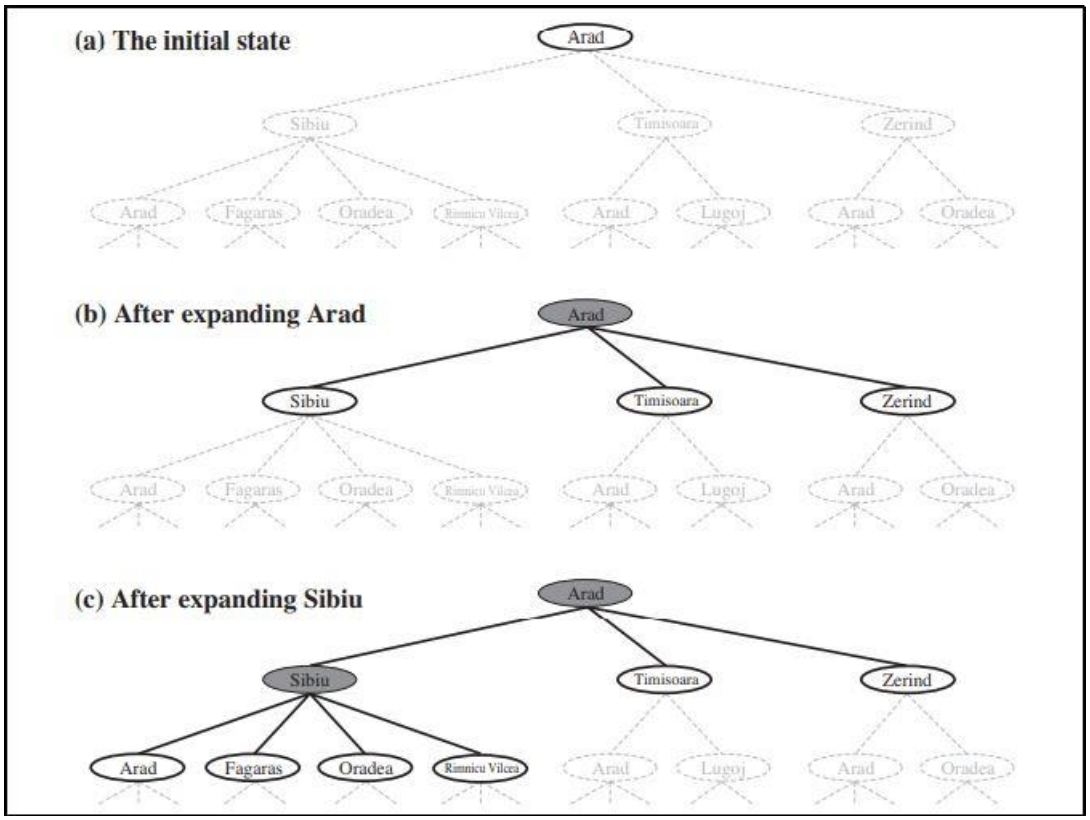
Tekoälyt käyttävät erilaisia hakualgoritmeja ratkaistakseen ongelmia, jotka vaativat tulevien toimien suunnittelua ja ratkaisun optimointia. Näitä ongelmia ratkovia agentteja kutsutaan tavoiteperustaisiksi agenteiksi. Agenteille annettavat ongelmat muotoillaan tila-avaruutena, jossa erilaisilla toimenpiteillä agentti voi siirtyä tilasta toiseen. Yksinkertaisessa tila-avaruusongelmassa on alkutila ja tavoitetila, ja haun tuloksena syntyy toimenpidejono näiden kahden väliltä. Russell ja Norvig (2010) havainnollistavat tätä kuvan 4 kartalla, missä kuvataan Romanian kaupunkeja ja niiden välisiä etäisyyksiä. Heidän esimerkissään on tavoitteena löytää kustannustehokkain, eli lyhyin, matka Aradista Bukarestiin, mitkä toimivat alkutilana ja tavoitetilana.

Kuva 4 Esimerkki hakuongelman tila-avaruudesta (Russell & Norvig, 2010, s. 68)



Haun mahdolliset toimintapolut esitetään hakupuuna (kuva 5), jossa juurena on tila-avaruuden alkutila. Hakupuun yhteydessä tiloja kutsutaan solmuiksi, joita laajennetaan soveltamalla mahdollisia toimenpiteitä, mikä vastaa siirtymistä seuraaviin tiloihin. Solmuja laajennetaan niin kauan, kunnes lopputilaa vastaava solmu löytyy. (Russell & Norvig, 2010, s. 75; ks. myös Hyvönen, 2004)

Kuva 5 Hakupuu, jossa etsitään reittiä Aradista Bukarestiin. (a) Lähtötila. (b) Arad-solmun laajentamisen jälkeen. (c) Sibiu-solmun laajentamisen jälkeen. (Russell & Norvig, 2010, s. 76)



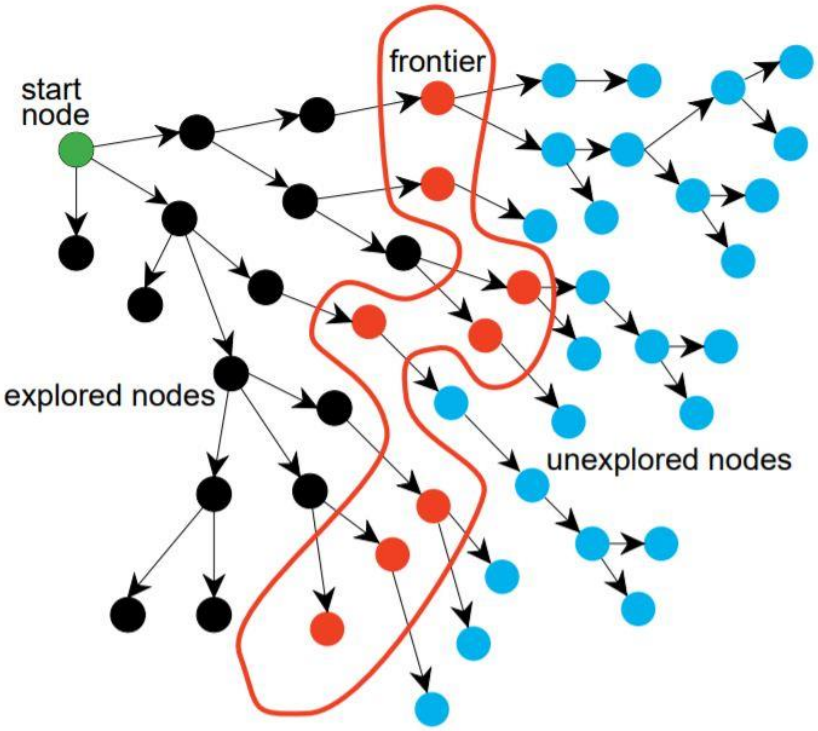
2.4.1 Hakualgoritmit

Hakualgoritmi etenee alkutilasta erilaisin tavoin riippuen käytetystä hakustrategiasta. Kuvassa 6 havainnollistetaan tätä prosessia, missä mustat pallot ovat tutkittuja ja siniset tutkimattomia solmuja. Näiden välissä on niin kutsuttu rintama (engl. frontier), joka sisältää algoritmin mahdolliset seuraavat toimenpiteet. Tavalliset hakustrategiat toimivat niin sanotusti sokkona, ne eivät saa alustavaa tietoa eivätkä ne ota huomioon tavoitetta päätöksenteossa. Ne tutkivat kaikki reitit ja valitsevat optimaalisen ratkaisun. (Poole ym., 1998; Artasanchez & Joshi, 2020, s. 222)

Yksinkertaisimpia hakustrategioita ovat muun muassa Syvyys ensin- ja Leveys ensin -haut. Syvyys ensin -haku valitsee aina rintaman uusimman mahdollisimman solmun, joten se etenee ensin yhden valitsemansa polun loppuun, jolloin se palaa takaisinpäin ja etenee lähintä tutkimatonta haaraa. Tämän hakustrategian selvänä ongelmana on rajattomat tai sykliset tila-avaruudet, joissa se ei välttämättä pysähdy ikinä. Leveys ensin -haku valitsee

aina rintaman vanhimman tilan tutkittavaksi, edeten näin kaikkia mahdollisia haaroja samaan aikaan. Leveys ensin -haku saavuttaa aina lopputilan, jos tiloilla on rajallinen määrä haaroja, ja löytää aina suorimman reitin lopputilaan. Tämä perusteellinen hakutapa johtaa kuitenkin vaaditun laskentatehon ja -ajan räjähdysmäiseen kasvuun tila-avaruuden kasvaessa ja monimutkaistuessa. (Poole ym., 1998)

Kuva 6 Hakualgoritmin eteneminen tila-avaruudessa, tässä leveys ensin -haulla (Poole ym., 1998)



2.4.2 Heuristinen haku

Edellä esitellyt Syvyys ensin- ja Leveys ensin -hakustrategiat etenivät tila-avaruudessa harhaillen, valiten reitin ennalta määritetyn etenemistavan mukaan. Tähän vastakohtana on heuristinen haku, jossa käytetään sääntöjä ja tila-avaruuden ulkopuolista tietoa poistamaan turhia ja epäoptimaalisia vaihtoehtoja. Kun parhaan mahdollisen ratkaisun löytäminen on mahdotonta tai liian aikaa vievää, heuristisella haulla voidaan karsia toimenpiteitä, jotka ovat selvästi huonoja ja priorisoida vaihtoehtoja, joilla on parhaat mahdollisuudet hyvään lopputulokseen. Tämä nopeuttaa hyvän ratkaisun löytämistä, mutta se ei kuitenkaan käy läpi

kaikki mahdollisia ratkaisuja. Heuristista hakua on paras käyttää, kun tavoitteena on saada hyvä ratkaisu kohtuullisessa ajassa. (Artasanchez & Joshi, 2020, s. 222)

Heuristinen haku käyttää arvofunktiota arvioimaan solmujen kautta kulkevan ratkaisupolun kustannuksen. Arvion tekemiseen käytetään ongelman ulkopuolelta tullutta lisätietoa eli heuristiikkaa. Tästä esimerkkinä toimii Romanian kaupunkien suora etäisyys pääkaupunki Bukarestiin (kuva 7), joka on tietoa, jota ei saa selville ongelmasta itsestään. Suora etäisyys tavoitteeseen korreloi hyvin tietä pitkin ajetun etäisyyden kanssa, joten se on käytännöllinen heuristiikka. (Russell & Norvig, 2010, s. 92; ks. myös Hyvönen, 2004)

Kuva 7 Romanian kaupunkien suorat etäisyydet Bukarestiin. (Russell & Norvig, 2010, s. 93)

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Heuristiikkaakin käytettäessä on tarpeen valita erilaisista hakustrategioista ongelmaan sopivin. Yksi käytetympiä on Paras ensin -haku, mikä laajentaa aina parhaan kustannusarvion saaneen solmun. Ahne paras ensin -haku valitsee aina lähimpänä loppuratkaisua olevan solmun. A*-haku (A tähti) ottaa huomioon myös jo kuljetun matkan, välttäen polkuja, joilla on korkeammat kustannusarviot. Tämä hakutyyppi onkin erittäin paljon käytetty, kun etsitään lyhyintä mahdollista reittiä loppuratkaisuun. (Russell & Norvig, 2010, s. 92; ks. myös Hyvönen, 2004)

2.5 Rajoitelaskenta

Rajoitelaskentaongelmat (engl. Constraint Satisfaction Problem, CSP) ovat matemaattisia ongelmia, joissa on äärellinen määrä **muuttujia**, joilla on äärellinen määrä **tiloja**, joiden tulee noudattaa määrättyjä **rajoitteita** (Tsang, 1993, s. 1). Rajoitteet toimivat heuristisena apuna, karsien pois toimimattomia ratkaisuja. Esimerkkinä toimii Sudoku-numeropeli (kuva 8), jossa

yksi laatikko on muuttuja, jolla voi olla yhdeksän erilaista tilaa, numerot 1-9. Näitä koskevat kolme eri rajoitetta: Numero ei saa toistua samalla rivillä, samassa sarakkeessa eikä samassa pienemmässä neliössä. Jokaisen mahdollisen numeroyhdistelmän yrittäminen olisi aikaa vievää, joten rajoitteita käyttäen päästään toimivaan loppuratkaisuun paljon nopeammin.

Rajoitteiden toimintaa voidaan havainnollistaa laskemalla punaisella merkityn ruudun numero. Aluksi voidaan poistaa numerot 1, 6, 8, ja 9, jotka sijaitsevat samassa neliössä. Numerot 2 ja 7 voidaan poistaa, koska ne sijaitsevat samalla rivillä ja numerot 3 ja 4, koska ne sijaitsevat samassa sarakkeessa. Ruudun numeron on siis oltava 5. (Artasanchez & Joshi, 2020, s. 224)

Kuva 8 Sudoku-peli (Artasanchez & Joshi, 2020, s. 223)

			2	6		7		1
6	8			7			9	
1	9				4	5		
8	2		1				4	
		4	6		2	9		
	5				3		2	8
		9	3				7	4
	4			5			3	6
7		3		1	8			

Rajoitelaskentaongelman jokaista mahdollista tilaa, jossa joillain tai kaikilla muuttujilla on arvoja, kutsutaan asetukseksi (engl. assignment). Asetus, jossa jokaisella muuttujalla on arvo, joka toteutuu rajoitusten mukaan, on toimiva loppuratkaisu. Ratkaisu, tai ongelman ratkaisemattomuus, löydetään soveltamalla ongelmaan heuristiikkaa ja muita tilahakutekniikoita. Rajoitelaskennan ongelmissa on yleensä enemmän kuin yksi oikea ratkaisu. (Russell & Norvig, 2010, s. 203)

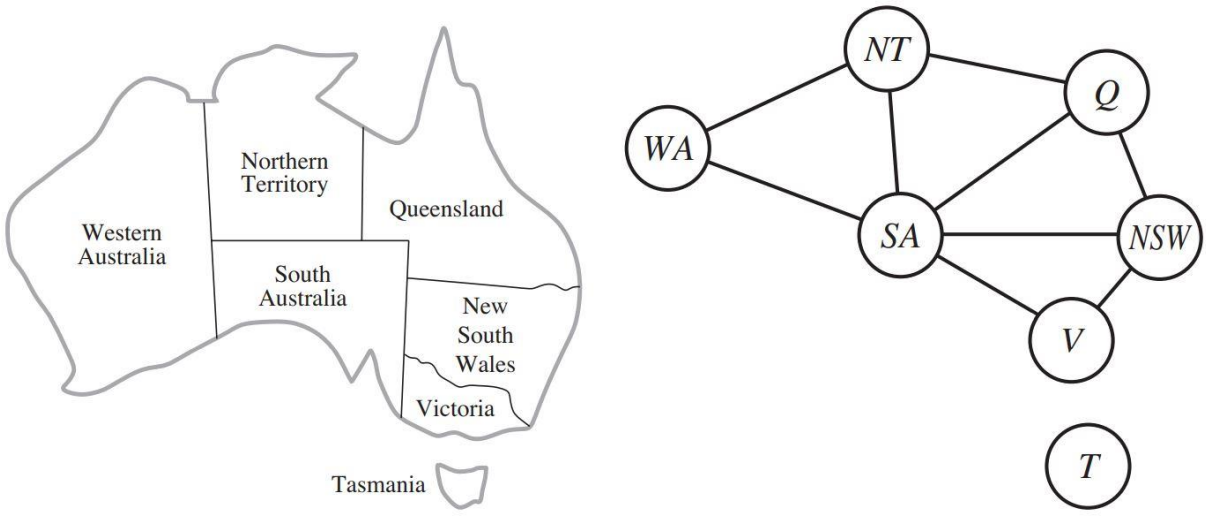
Rajoitelaskenta on käytännöllinen ongelmanratkontatapa, koska monet tosielämän ongelmat voidaan käsittää ja ratkaista sen avulla. Työnjako ja aikataulutus ovat yleisiä esimerkkejä tästä, koska niissä yksittäiset muuttujat ovat usein riippuvaisia toisistaan. Rajoitelaskentamenetelmät kehitetään ongelmakohtaisen suunnittelun sijasta

yleiskäyttöiseksi ratkaisuksi, niin että niitä voidaan soveltaa monimutkaisiinkin ongelmiin. (Russell & Norvig, 2010, s. 202; ks. myös Hyvönen, 2004)

Rajoitelaskentaongelmien rakenne ei ole samanlainen tila-avaruus kuin tavallisissa hakumenetelmissä, vaan se voidaan käsittää rajoiteverkkona, jossa ei ole tiettyä alku- ja loppukohtaa. Haku voidaan aloittaa mistä tahansa kohtaa, ja sitä voidaan suunnata erilaisilla rajoitteilla. Kuva 9 havainnollistaa miten Australian osavaltiot ovat yhteydessä toisiinsa rajoiteverkossa. Verkko voidaan myös jakaa aliongelmiin, tässä Tasmania on riippumaton muista muuttujista. (Russell & Norvig, 2010, s. 203; ks. myös Hyvönen, 2004)

Rajoiteverkkorakenteen ansiosta voidaan karsia runsaasti toimimattomia asetuksia kerralla, mikä nopeuttaa ratkaisun löytymistä verrattuna tila-avaruushakuun. Esimerkkinä ongelma, jossa kuvan 9 osavaltiot pitäisi värittää kolmea väriä käyttäen, ilman että kaksi vierekkäistä osavaltiota käyttää samaa väriä. Muuttujalla South Australia (SA) on eniten vierekkäisiä muuttujia, joten jos sille annetaan arvoksi sininen, tiedetään että näillä viidellä muuttujalla on enää kaksi mahdollista arvoa. Tämä karsii laskettavia asetuksia $3^5 = 243$ mahdollisesta asetuksesta $2^5 = 32$ asetukseen. (Russell & Norvig, 2010, s. 203)

Kuva 9 Australian osavaltiot karttana ja rajoiteverkkona (Russell & Norvig, 2010, s. 204)



3 Pythonin esittely

Python on alankomaalaisen Guido van Rossumin 1980-luvun alkupuolella kehittämä ohjelmointikieli. Se on tunnettu yhtenä helpoimmin opittavista ja selkeimmin ymmärrettävistä ohjelmointikielistä, ja sen kehitysfilosofiaan kuuluu koodin kauneus ja yksinkertaisuus. Tämä heijastuu koodin ulkoasuun, joka perustuu sisennyksiin, jotka ilmaisevat ohjelman rakennetta (ohjelmakoodi 1). Tämän ansiosta koodissa ei tarvita monen muun kielen käyttämiä sulkulohkoja, mikä tekee Pythonista monen käyttäjän mielestä helpommin luettavaa. Yksi Pythonin tunnusmerkkejä on dynaaminen tyypittäminen. Tämä tarkoittaa, että muuttujan tyyppiä ei tarvitse esitellä, vaan se määritellään automaattisesti muuttujan arvon mukaan. Esimerkiksi, jos muuttujalle `nimi` annetaan tekstiarvo ”Jesse”, osaa Python antaa tälle muuttujalle merkkijonotyypin. (Python Land, 2021)

Ohjelmakoodi 1 Esimerkki Python-kielen sisennyksistä. (Python Land, 2021)

```
def bigger_than_five(x):
  # The contents of a function are indented
  if x > 5:
    # This is another, even more indented block of code
    print("X is bigger than five")
  else:
    # And one more!
    print("x is 5 or smaller")
```

Pythonia voidaan käyttää proseduraalisena, oliopohjaisena sekä funktionaalisena ohjelmointikielenä. Nämä tarkoittavat, että sitä käyttävät ohjelman toiminnallisuutta voidaan jakaa itsenäisiin aliohjelmiin (proseduraalinen), ohjelmassa olevat tiedot ja niitä koskevat toiminnallisuudet jäsennetään omiksi tietorakenteiksi (oliopohjainen) ja ohjelmat voidaan perustaa matemaattisiin funktioihin (funktionaalinen). Python sisältää myös suuren määrän kirjastoja, jotka tarjoavat erilaisia toiminnallisuuksia, joiden ansiosta sillä ohjelmointi voi olla huomattavasti nopeampaa ja tiiviimpää kuin muilla ohjelmointikielillä, kuten C tai Java. Se tarjoaa myös tuen muiden tekemille kirjastoille, kuten seuraavaksi esiteltävät PyCSP[3] ja pyDatalog, jotka tarjoavat toiminnallisuuksia rajoitelaskentaongelmien ratkaisemiseksi. (Summerfield, 2010, s. 1)

3.1 PyDatalog

PyDatalog on rajoitelaskentaongelmia varten suunniteltu Python-kirjasto, joka on kehitetty avoimen lähdekoodin projektina Belgiassa. Sen periaatteena on deklaratiivinen koodi, jossa kuvataan haluttua lopputulosta, jonka tietokoneen pitää saada tuotettua. Oppimisen ja käytön helpottamiseksi pyDatalog sisältää pienen määrän erilaisia lauseita, joista suurin osa sisältyy jo Pythoniin. Se on suunniteltu monimutkaisten ongelmien ratkontaa varten, se esimerkiksi pitää keskeneräisiä hakutuloksia välimuistissa välttääkseen tekemästä samoja prosesseja turhaan. (pyDatalog, n.d.)

Muuttujat ja logiikkafunktiot luodaan kummatkin pyDatalogia käytettäessä `create_terms()`-funktiolla, muuttujat isolla ja funktiot pienellä alkukirjaimella. Ohjelmakoodissa 2 on lyhyt esimerkki pyDatalogin käytöstä. Siinä kuvastuu kuinka pyDatalogilla luodut logiikkafunktiot toimivat samalla tavalla kuin Pythonin sanakirjatyypit. (pyDatalog, n.d.)

Ohjelmakoodi 2 Esimerkki pyDatalogia käyttävästä koodista. (pyDatalog, n.d.)

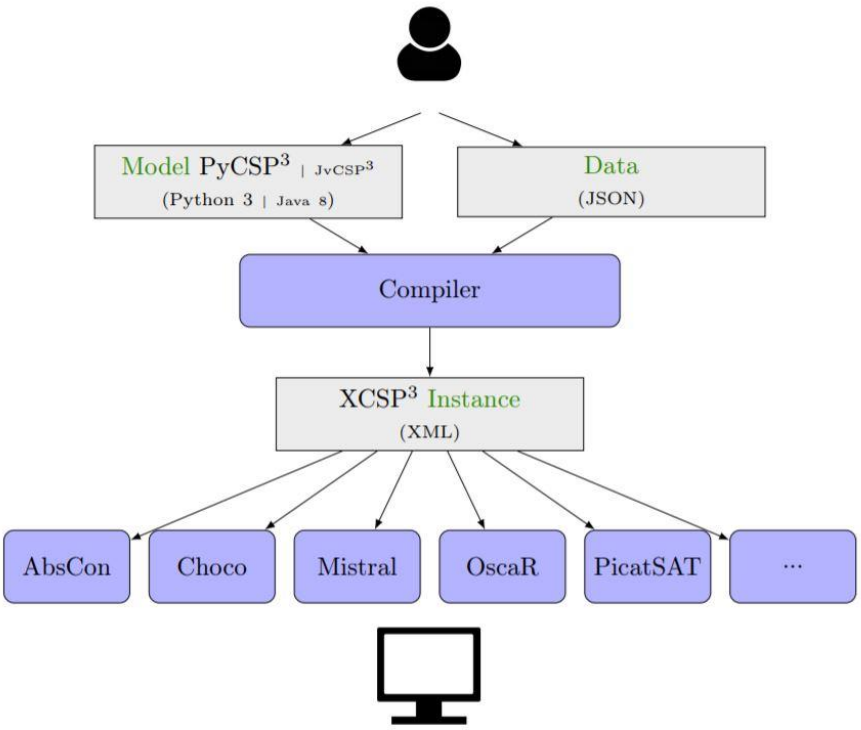
```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z, salary, tax_rate,
tax_rate_for_salary_above, net_salary')
salary['foo'] = 60
salary['bar'] = 110
# give me all the X and Y so that the salary of X is Y
print(salary[X]==Y)
X   | Y
----|----
bar | 110
foo | 60
```

3.2 PyCSP^3

PyCSP^3 on rajoitelaskentaan tarkoitettu Python-kirjasto, jonka ensimmäinen versio julkaistiin vuonna 2019 ja jonka ovat kehittäneet Christophe Lecoutre ja Nicolas Szczepanski. Sen tarkoituksena on rajoitelaskentaongelmien mallintaminen yksinkertaisella ja deklaratiivisella koodilla. Siitä on myös Java-kielinen versio. Tätä kirjastoa käytettäessä ongelmien mallintaminen ja ratkaiseminen on jaettu kahteen eri osaan. Mallintaminen kirjoitetaan Python-tiedostoon, josta PyCSP^3-kirjasto tuottaa annetun datan kanssa XCSP^3-tyyppisen XML-tiedoston, joka voidaan syöttää rajoitelaskijaohjelmalle ongelman ratkaisemiseksi.

Kuvassa 10 esitetään tämä prosessi sekä muutamia yhteensopivia rajoitelaskentaohjelmia. (Lecoutre & Szczepanski, 2020)

Kuva 10 Kaava PyCSP^3-kirjaston käytöstä. (Lecoutre & Szczepanski, 2020)



4 Rajoitelaskenta Pythonilla

Tässä luvussa esitellään ja vertaillaan kolmea eri tapaa ratkaista rajoitelaskentaongelmia Python-kielellä. Ensimmäiseksi ja tarkimmin läpikäydään klassisia Python-algoritmeja, joita voitaisiin hyödyntää opetuksessa, jolloin rajoitelaskennan rakenne ja toiminnot tulisivat selkeästi esille. Algoritmeja havainnollistetaan kahdella rajoitelaskentaongelmalla, joista toinen ratkaistaan myös käyttäen rajoitelaskentaan erikoistuneita Python-kirjastoa, pyCSP[3] ja pyDatalog. Näin saadaan vertailtua, kuinka erilaisin tavoin rajoitelaskentaa voidaan toteuttaa ja mikä sopisi parhaiten opetukseen.

4.1 Klassiset Python-algoritmit

Tässä luvussa esitellään rajoitelaskentaongelmien ratkomiseen tarkoitettuja Python-kielisiä algoritmeja, jotka ovat alkuperäisin kirjasta Classic Computer Science Problems in Python (Kopec, 2019). Aluksi luodaan muuttujia, tiloja ja rajoitteita koskevat perustoiminnallisuudet yhteen csp.py-nimiseen tiedostoon, jota voidaan soveltaa ja mukauttaa erilaisiin ongelmiin. Seuraavissa alaluvuissa esitellään tätä soveltamista.

Aluksi csp.py-tiedostoon luodaan rajoitteiden (engl. constraint) perusluokka (ohjelmakoodi 3), joka on abstrakti luokka tarkoittaen, että sitä ei itsessään käytetä valmiina koodina, vaan siitä periytyvien alaluokkien täytyy määritellä ja toteuttaa se annettujen vaatimusten mukaan. Rajoiteluokassa määritellään ensiksi muuttujat, joita rajoitteet koskevat. Rajoitteiden toteutumisen tarkistaa abstrakti funktio `satisfied()`, joka määritellään aina ongelmakohtaisesti.

Ohjelmakoodi 3 csp.py: Koodin alku ja rajoiteluokka (Kopec, 2019, Building a constraint-satisfaction problem framework -luku)

```
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod

V = TypeVar('V') # Muuttuja (Variable)
D = TypeVar('D') # Tila (Domain)


# Rajoitteiden perusluokka
class Constraint(Generic[V, D], ABC):
    # Muuttujat, joita rajoitteet koskevat
    def __init__(self, variables: List[V]) -> None:
```

```
        self.variables = variables

    # Abstrakti funktio, jota alaluokkien täytyy toteuttaa
    @abstractmethod
    def satisfied(self, assignment: Dict[V, D]) -> bool:
        ...
```

Csp.py-tiedoston ytimessä on `CSP`-luokka, joka käsittelee muuttujien, tilojen ja rajoitteiden yhdistämisen (ohjelmakoodi 2). `CSP`-luokkaa varten muuttujat ja tilat on määritelty geneerisesti, mahdollistaen toimivuuden erilaisten arvotyyppien kanssa. Tässä luokassa käytetyt muuttujat kootaan `list`-tyyppiseen `variables`-kokoelmaan. Muuttujat ja niiden mahdolliset arvot eli tilat kootaan `dict`-tyyppiseen `domains`-kokoelmaan. `Constraints` on `dict`-tyyppinen kokoelma, joka yhdistää muuttujat listaan niitä koskevia rajoitteita. Kokoelmat täytetään `__init__()`-rakentajassa. Muuttujien rajoitteisiin yhdistämistä varten luodaan `add_constraint()`-funktio. Tämä funktio tarkistaa muuttujalistasta koskeeko käsittelyssä oleva rajoite kutakin muuttujaa. Rajoitteen ja muuttujan yhteys lisätään `constraints`-kokoelmaan, mikäli rajoite koskee muuttujaa. Kummassakin funktiossa hyödynnetään yksinkertaista virheentarkistusta.

Ohjelmakoodi 4 csp.py: `CSP`-luokka (Kopec, 2019, Building a constraint-satisfaction problem framework -luku)

```
class CSP(Generic[V, D]):
    def __init__(self, variables: List[V], domains: Dict[V, List[D]])
-> None:
        self.variables: List[V] = variables # Muuttujat joita
rajoitteet koskevat
        self.domains: Dict[V, List[D]] = domains # Muuttujien tilat
        self.constraints: Dict[V, List[Constraint[V, D]]] = {}
        for variable in self.variables:
            self.constraints[variable] = []
            if variable not in self.domains:
                raise LookupError("Jokaisella muuttujalla pitäisi olla
tila.")

    # Rajoitteiden yhdistäminen muuttujiin
    def add_constraint(self, constraint: Constraint[V, D]) -> None:
        for variable in constraint.variables:
            if variable not in self.variables:
                raise LookupError("Muuttujaa ei löytynyt CSP:stä")
            else:
                self.constraints[variable].append(constraint)
```

Seuraavaksi tarvitaan toiminto, joka tarkistaa toteuttavatko muuttujat ja niiden arvot jokaista niitä koskevaa rajoitusta. Tätä varten `CSP`-luokkaan luodaan `consistent()`-funktio

(ohjelmakoodi 5). Tilannekuvaa muuttujista ja niiden nykyisestä tila-arvosta kutsutaan `assignmentiksi` (suom. asetus). `Consistent()`-funktio tarkastaa `assignmentin` jokaisen muuttujan kohdalla toteutuvatko niiden tila-arvot rajoitteiden mukaan. Mikäli jokainen muuttuja toteuttaa rajoitteita onnistuneesti, funktio palauttaa arvon `True`, muutoin se palauttaa arvon `False`.

Ohjelmakoodi 5 csp.py: Rajoitteiden toteutumisen tarkistus (Kopec, 2019, Building a constraint-satisfaction problem framework -luku)

```
    def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:
        for constraint in self.constraints[variable]:
            if not constraint.satisfied(assignment):
                return False
        return True
```

Toimivien ratkaisumallien etsimisessä hyödynnetään tässä esimerkissä peräytyvää hakumallia (engl. backtracking search). Tässä mallissa edetään niin kauan, kunnes haussa ei ole enää mahdollista edetä, jolloin palataan taaksepäin ja yritetään toista reittiä. Tätä hakumallia kutsutaan myös syvyys ensin -hauksi. Tätä varten `CSP`-luokkaan luodaan `backtracking_search()`-niminen funktio (ohjelmakoodi 6). Tämä funktio etsii muuttujille tila-arvoja niin kauan, kunnes kaikilla on rajoitteet toteuttavat arvot, jolloin se palauttaa edellisessä funktiossa esitellyn `assignmentin`. Käsitellessään muuttujia funktio etsii muuttujia, jotka eivät ole vielä `assignmentissa`, ja lisää ne `unassigned`-nimiseen listaan. Tämän listan ensimmäistä käsitellään `first`-nimisenä muuttujana. Tälle muuttujalle määritellään kaikki mahdolliset tila-arvot ja tuloksena syntyvät `assignmentit` kootaan `dict`-tyyppiseen `local_assignmentiin`.

Tämän jälkeen kutsutaan `consistent()`-funktiota tarkastamaan `local_assignment`. Mikäli `local_assignment` toteuttaa annettuja rajoitteita, hakua jatketaan rekursiivisesti sen pohjalta niin kauan, kunnes kaikille muuttujille on saatu hyväksytyt arvot. Kun yhtäkään toimivaa ratkaisua ei löydy, `consistent()` palauttaa arvon None, ja rekursiivisessa haussa palataan askel taaksepäin.

Ohjelmakoodi 6 csp.py: Peräytyvä haku (Kopec, 2019, Building a constraint-satisfaction problem framework -luku)

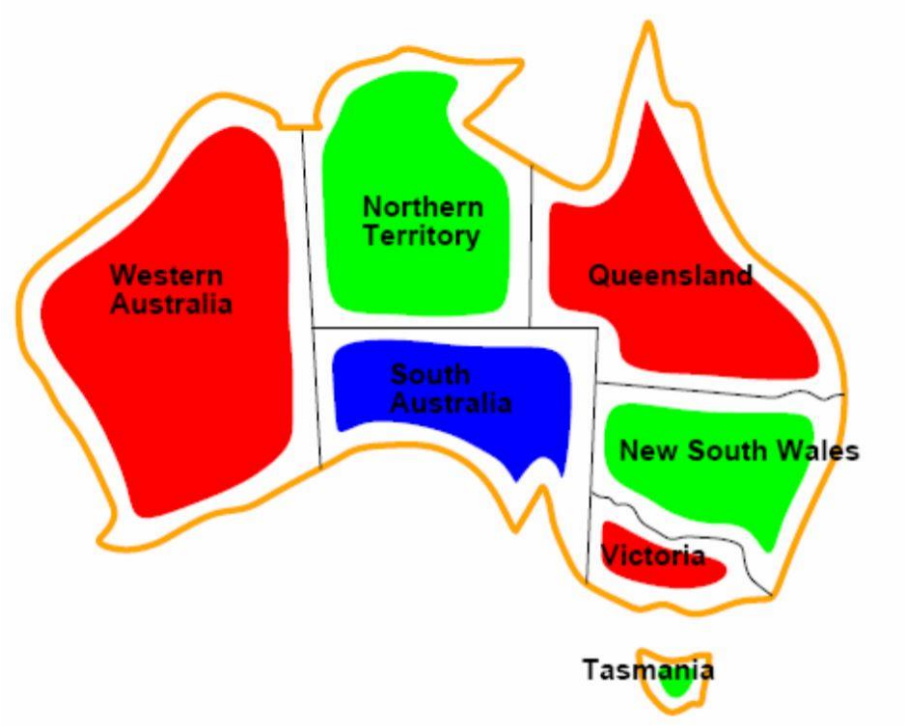
```
    def backtracking_search(self, assignment: Dict[V, D] = {}) ->
Optional[Dict[V, D]]:
```

```
        # Assignment on valmis kun kaikki muuttujat on määritelty
        if len(assignment) == len(self.variables):
            return assignment

        # Hakee muuttujat jotka ovat CSP:ssä, mutta eivät
assignmentissa
        unassigned: List[V] = [v for v in self.variables if v not in
assignment]

        # Hakee jokaisen mahdollisen tila-arvon ensimmäiselle
käsittelemättömälle muuttujalle
        first: V = unassigned[0]
        for value in self.domains[first]:
            local_assignment = assignment.copy()
            local_assignment[first] = value
            # Toimintoa jatketaan niin kauan kun consistent()
palauttaa True
            if self.consistent(first, local_assignment):
                result: Optional[Dict[V, D]] =
self.backtracking_search(local_assignment)
                # Jos ratkaisua ei löytynyt, palataan taaksepäin
                if result is not None:
                    return result
        return None
```

4.2 Australia-ongelma

Valmista csp.py-tiedostoa voidaan käyttää apuna ratkomaan luvussa 2.5 esiteltyä rajoitelaskentaongelmaa, jossa Australian osavaltiot pitäisi värittää kolmella värillä niin, että kahdella vierekkäisellä osavaltiolla ei ole samaa väriä. Kuvassa 11 näkyy yksi mahdollinen ratkaisu. Ongelma on ihmismielelle suhteellisen yksinkertainen ja nopeasti ratkaistavissa, mutta sillä voi hyvin havainnollistaa CSP-ongelman ratkomista käytännön kooditasolla. Tässä esimerkkiongelmassa muuttujina toimivat osavaltiot, joilla on kolme mahdollista tilaa: punainen, vihreä ja sininen. Rajoitteina toimivat vierekkäiset osavaltioparit, joilla ei voi olla kahta samaa arvoa. Kun kaikki rajoitteet toteutuvat onnistuneesti, saadaan toimiva ratkaisu.

Kuva 11 Mahdollinen ratkaisu Australia-aiheiseen ongelmaan (Hyvönen, 2004)



Tätä ongelmaa varten voidaan luoda oma australia.py-niminen tiedosto, johon tuodaan csp.py-tiedostosta `Constraint`- ja `CSP`-luokat (ohjelmakoodi 7). Rajoitteet käsitellään `MapColoringConstraint`-nimisessä luokassa, mikä on abstraktin `Constraint`-luokan alaluokka ja jonka tulee rakentaa ja toteuttaa sen antama `satisfied()`-funktio. Tässä ongelmassa muuttujat ja tilat ovat tekstimuodossa, joten rajoiteluokka käsittelee niitä merkkijonoina. Muuttujia verratessa ne nimetään `place1` ja `place2`.

Ohjelmakoodi 7 australia.py: Rajoiteluokka (Kopec, 2019, The Australian map-coloring problem -luku)

```
from csp import Constraint, CSP
from typing import Dict, List, Optional


class MapColoringConstraint(Constraint[str, str]):
    def __init__(self, place1: str, place2: str) -> None:
        super().__init__([place1, place2])
        self.place1: str = place1
        self.place2: str = place2

    def satisfied(self, assignment: Dict[str, str]) -> bool:
        # Palauttaa True jos jommalla kummalla ei ole vielä arvoa
        if self.place1 not in assignment or self.place2 not in
assignment:
            return True
        # Vertaa kahden osavaltion arvoja, palauttaa True jos eri
```

```
        return assignment[self.place1] != assignment[self.place2]
```

Abstrakti `satisfied()`-funktio toteutetaan tässä niin, että ensin tarkistetaan että kummallekin muuttujalle on annettu tila-arvo, mikä on tässä tapauksessa osavaltion väri. Muuttujien arvot eivät voi rikkoa rajoitteita, jos toinen tai kummatkin puuttuvat, joten rajoite toteutuu. Kummankin arvon ollessa annettuna, tarkistetaan ovatko arvot erilaisia. Funktio palauttaa `True` jos ne ovat. Ongelman rajoitteiden määrittelyn jälkeen on vuorossa muuttujien ja tilojen sekä niitä koskevien rajoitteiden asettaminen (ohjelmakoodi 8). Lopuksi kutsutaan peräytyvää hakua ongelman ratkaisemiseksi.

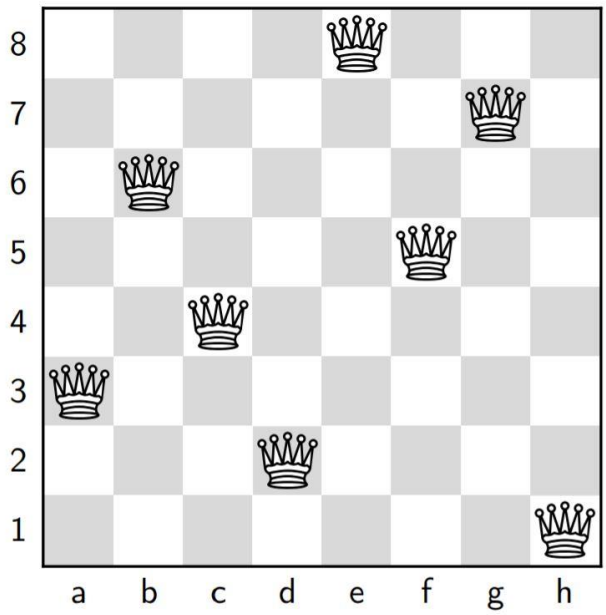
Ohjelmakoodi 8: australia.py: Muuttujien, tilojen ja rajoitteiden asettaminen. (Kopec, 2019, The Australian map-coloring problem -luku)

```
if __name__ == "__main__":
    variables: List[str] = ["Western Australia", "Northern Territory",
"South Australia",
                            "Queensland", "New South Wales",
"Victoria", "Tasmania"]
    domains: Dict[str, List[str]] = {}
    for variable in variables:
        domains[variable] = ["puna", "viher", "sini"]
    csp: CSP[str, str] = CSP(variables, domains)
    csp.add_constraint(MapColoringConstraint("Western Australia",
"Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Western Australia",
"South Australia"))
    csp.add_constraint(MapColoringConstraint("South Australia",
"Northern Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland", "Northern
Territory"))
    csp.add_constraint(MapColoringConstraint("Queensland", "South
Australia"))
    csp.add_constraint(MapColoringConstraint("Queensland", "New South
Wales"))
    csp.add_constraint(MapColoringConstraint("New South Wales", "South
Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "South
Australia"))
    csp.add_constraint(MapColoringConstraint("Victoria", "New South
Wales"))
    csp.add_constraint(MapColoringConstraint("Victoria", "Tasmania"))
    solution: Optional[Dict[str, str]] = csp.backtracking_search()
    if solution is None:
        print("Ei ratkaisua")
    else:
        print(solution)
```

4.3 Kuningatarongelma

Tässä luvussa vertaillaan edellä esiteltyä ratkaisutapaa ja kahta valmista Python-kirjastoa, PyCSP[3] ja pyDatalog, käyttäen shakin kuningatarnappuloihin liittyvää ongelmaa. Ongelman ympäristönä toimii shakkipöytä, joka on ruudukko, jossa on kahdeksan riviä ja kahdeksan saraketta. Kuningatar on shakkinappula, joka voi liikkua viistoon sekä pysty- ja vaakasuoraan niin pitkälle kunnes se saavuttaa toisen nappulan tai shakkipöydän rajan. (Montonen, n.d.) Kahdeksan kuningattaren ongelmaksi (The eight queens problem) kutsutussa ongelmassa on tarkoitus saada selville, miten shakkipöydälle asetetaan kahdeksan kuningatarta niin, että ne eivät voi hyökätä toisiaan kohtaan. (Kopec, 2019) Kuvassa 12 yksi mahdollinen ratkaisu.

Kuva 12 Yksi mahdollinen ratkaisu kahdeksan kuningattaren ongelmaan. (Lecoutre & Szczepanski, 2020, s. 15)



Australia-ongelman tapaan ongelma ratkaistaan omassa queens.py-nimisessä tiedostossa (ohjelmakoodi 9), johon tuodaan csp.py-tiedoston toiminnallisuudet. Shakkipöydän ruuduille annetaan koodin pääluokassa kaksi kokonaislukuarvoa (`int`), jotka kuvaavat ruudun riviä ja saraketta. Jokaiselle kuningattarelle annetaan oma sarakkeensa, joka toimii rajoitelaskennan muuttujana. Muuttujan tilaksi haetaan kuningattaren riviä, jota rajoittavat pöydän toiset kuningattaret.

Ohjelmakoodi 9 queens.py (Kopec, 2019, The eight queens problem -luku)

```
from csp import Constraint, CSP
from typing import Dict, List, Optional


class QueensConstraint(Constraint[int, int]):
    def __init__(self, columns: List[int]) -> None:
        super().__init__(columns)
        self.columns: List[int] = columns

    def satisfied(self, assignment: Dict[int, int]) -> bool:
        # q1c = kuningatar (queen) 1 sarake (column), q1r = kuningatar
1 rivi (row)
        for q1c, q1r in assignment.items():
            # q2c = kuningatar 2 sarake
            for q2c in range(q1c + 1, len(self.columns) + 1):
                if q2c in assignment:
                    q2r: int = assignment[q2c] # q2r = kuningatar 2
rivi
                    if q1r == q2r: # sama rivi?
                        return False
                    if abs(q1r - q2r) == abs(q1c - q2c): # sama
viistoon?
                        return False
        return True # rajoitteet toteutuvat


if __name__ == "__main__":
    columns: List[int] = [1, 2, 3, 4, 5, 6, 7, 8]
    rows: Dict[int, List[int]] = {}
    for column in columns:
        rows[column] = [1, 2, 3, 4, 5, 6, 7, 8]
    csp: CSP[int, int] = CSP(columns, rows)
    csp.add_constraint(QueensConstraint(columns))
    solution: Optional[Dict[int, int]] = csp.backtracking_search()
    if solution is None:
        print("Ei ratkaisua")
    else:
        print(solution)
```

Sarake ratkaistaan aluksi ja rivi voidaan tarkastaa helposti, mutta viistoon tarkistaminen on monimutkaisempaa. Jos kuningattaret ovat samassa linjassa viistosuunnassa, voidaan laskea, että kummankin kuningattaren riviarvon ja sarakearvon erotus on sama. Rajoitteet lasketaan csp.py-tiedosta tulleessa `satisfied()`-metodissa. `CSP`-luokasta tuodaan rajoitteiden lisäys ja peräytyvä haku, joka saa vastaukseksi ohjelmakoodin 10, jossa esitetään kuningattarien sarakkeet ja niille löydetyt rajoitteita toteuttavat rivit.

Ohjelmakoodi 10 haun tuloksena syntyvä toimiva ratkaisu. (Kopec, 2019, The eight queens problem -luku)

```
{1: 1, 2: 5, 3: 8, 4: 6, 5: 3, 6: 7, 7: 2, 8: 4}
```

4.3.1 PyCSP³

Kuningatarongelman ratkaiseminen PyCSP³-kirjastoa käyttäen jättää opiskelijan itse kirjoitettavan Python-koodin selvästi lyhyemmäksi. Aluksi määritellään muuttuja `n`, joka kuvaa ongelman kokoa, tässä tapauksessa kuinka monta riviä, saraketta ja kuningatarta ongelmassa on, ja se saa arvon PyCSP³-kirjaston `data`-nimisestä muuttujasta joka hakee arvon koodin suorituskomennossa. Arvo voidaan ilmaista suoraan tai viitata JSON-tiedostoon, jossa on `n`-niminen muuttuja. Kun arvon halutaan olevan 8, suoritetaan koodi komennolla 1:

Komento 1 PyCSP³-kirjastoa käyttävän Python-koodin suorittamiskomento (Lecoutre & Szczepanski, 2020, s. 16)

```
python3 Queens.py -data=8
```

Kuningattaret kuvataan taulukkomuuttujana (`VarArray`) `q`, joka on yhtä suuri kuin muuttuja `n`, ja jossa on muuttujan tilaa kuvaava arvo `dom`. Tässä taulukossa indeksi kuvaa jokaisen kuningattaren riviä. Rajoitteiden toteutuminen tarkistetaan `satisfy()`-funktiossa, jossa voidaan käyttää useita erilaisia valmiita rajoitetyyppejä. Paljon käytetty ja tähän ongelmaan loogisesti sopiva tarkistustapa on `AllDifferent()`, joka tarkistaa että kaikki q-taulukon arvot ovat erilaisia toisistaan. Ensimmäisellä funktiolla tarkastetaan, että jokaisella taulukon kuningattarella on eri arvo, eli ne ovat eri sarakkeissa. Kahdella toisella funktiolla tarkastetaan sama viistoon sekä ylös- että alasuuntaan (ohjelmakoodi 11). (Lecoutre & Szczepanski, 2020, s. 15)

Ohjelmakoodi 11 Queens.py: PyCSP³-kirjastoa käyttävä Python-koodi, jolla ratkaistaan kuningatarongelma. (Lecoutre & Szczepanski, 2020, s. 17)

```
from pycsp3 import *

n = data

# q[i] is the column of the ith queen (at row i)
q = VarArray (size =n, dom = range(n))

satisfy (
    # all queens are put on different columns
    AllDifferent(q),
    # no two queens on the same upward diagonal
    AllDifferent(q[i] + i for i in range(n)),
    # no two queens on the same downward diagonal
```

```
        AllDifferent(q[i] - i for i in range(n))
    )
```

Kun yllä oleva koodi suoritetaan Komennon 1 mukaisesti, saadaan XML-tiedosto (ohjelmakoodi 12), jota voidaan käyttää kahdeksan kuningattaren ongelman ratkaisemiseksi erityistä rajoitelaskijasovellusta käyttäen.

Ohjelmakoodi 12 Python-tiedoston tuottama XML-tiedosto, jota voidaan käyttää rajoitelaskijasovelluksessa. (Lecoutre & Szczepanski, 2020, s. 17)

```
<instance format ="XCSP3" type ="CSP">
    <variables>
        <array id="q" note ="q[i] is the column of the ith queen (at
row i)" size ="[8]">
        0..7
        </array>
    </variables>
    <constraints>
        <allDifferent note ="all queens are put on different columns">
        q[]
        </allDifferent>
        <allDifferent note ="no two queens on the same upward
diagonal">
        add(q[0],0) add(q[1],1) add(q[2],2) add(q[3],3) add(q[4],4)
add(q[5],5) add(q[6],6) add(q[7],7)
        </allDifferent>
        <allDifferent note ="no two queens on the same downward
diagonal">
        sub(q[0],0) sub(q[1],1) sub(q[2],2) sub(q[3],3) add(q[4],4)
add(q[5],5) add(q[6],6) add(q[7],7)
        </allDifferent>
    </constraints>
</instance>
```

4.3.2 PyDatalog

Käytettäessä pyDatalog-kirjastoa kahdeksan kuningattaren ongelman ratkaisemiseksi aloitetaan määrittelemällä muuttujat `create_terms`-funktiolla. Muuttujat `X0`, `X1`… esittävät kuningattaria, joiden indeksinumero vastaa niiden saraketta. Muuttujilla `ok`, `queens` ja `next_queen` ratkotaan rajoitteiden toteutumista. (ohjelmakoodi 13). (PyDatalog, n.d.)

Ohjelmakoodi 13 Python-koodin alustus pyDatalogia käyttäen. (pyDatalog, n.d.)

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('N,X0,X1,X2,X3,X4,X5,X6,X7')
pyDatalog.create_terms('ok,queens,next_queen')
```

Kun muuttujat on alustettu, asetetaan kuningattarille rivit (ohjelmakoodi 14). Ensimmäinen kuningatar voi olla millä tahansa rivillä, mutta seuraavan kuningattaren on oltava aina sitä seuraavalla rivillä. Tämä prosessi toistetaan, kunnes kaikilla kuningattarilla on rivi. `Next_queen`-funktio tarkistaa kuningattarien yhteensopivuuden ja varmistaa että ne ovat eri riveillä (ohjelmakoodi 15). Se käyttää tähän `ok`-funktiota, joka sallii kuningattarien sijoitukset, kunhan niiden sarakkeiden välillä on sama etäisyys kuin muuttujan `N` arvo. Kun ohjelma on saanut haettua rajoitteita toteuttavat arvot, se tulostaa ne listana.

Ohjelmakoodi 14 pyDatalogia käyttävän koodin jatkoa. <=-merkki vastaa ohjelmoinnissa usein käytettyä if-lauseketta. (pyDatalog, n.d.)

```
# the queen in the first column can be in any row
queens(X0)                      <= (X0._in(range(8)))
# to find the queens in the first 2 columns, find the first one first,
then find a second one
queens(X0,X1)                   <= queens(X0)                   &
next_queen(X0,X1)
# repeat for the following queens
queens(X0,X1,X2)                <= queens(X0,X1)                &
next_queen(X0,X1,X2)
queens(X0,X1,X2,X3)             <= queens(X0,X1,X2)             &
next_queen(X0,X1,X2,X3)
queens(X0,X1,X2,X3,X4)          <= queens(X0,X1,X2,X3)          &
next_queen(X0,X1,X2,X3,X4)
queens(X0,X1,X2,X3,X4,X5)       <= queens(X0,X1,X2,X3,X4)       &
next_queen(X0,X1,X2,X3,X4,X5)
queens(X0,X1,X2,X3,X4,X5,X6)    <= queens(X0,X1,X2,X3,X4,X5)    &
next_queen(X0,X1,X2,X3,X4,X5,X6)
queens(X0,X1,X2,X3,X4,X5,X6,X7) <= queens(X0,X1,X2,X3,X4,X5,X6) &
next_queen(X0,X1,X2,X3,X4,X5,X6,X7)
```

Ohjelmakoodi 15 Rajoitteiden toteutumisen tarkastus. (pyDatalog, n.d.)

```
# the second queen can be in any row, provided it is compatible with
the first one
next_queen(X0,X1)                   <= queens(X1)
& ok(X0,1,X1)
# to find the third queen, first find a queen compatible with the
second one, then with the first# re-use the previous clause for
maximum speed, thanks to memoization
next_queen(X0,X1,X2)                <= next_queen(X1,X2)
& ok(X0,2,X2)
# repeat for all queens
next_queen(X0,X1,X2,X3)             <= next_queen(X1,X2,X3)
& ok(X0,3,X3)
next_queen(X0,X1,X2,X3,X4)          <= next_queen(X1,X2,X3,X4)
& ok(X0,4,X4)
next_queen(X0,X1,X2,X3,X4,X5)       <= next_queen(X1,X2,X3,X4,X5)
& ok(X0,5,X5)
next_queen(X0,X1,X2,X3,X4,X5,X6)    <= next_queen(X1,X2,X3,X4,X5,X6)
& ok(X0,6,X6)
next_queen(X0,X1,X2,X3,X4,X5,X6,X7) <=
next_queen(X1,X2,X3,X4,X5,X6,X7) & ok(X0,7,X7)
```

```
# it's ok to have one queen in row X1 and another in row X2 if they
are separated by N columns
ok(X1, N, X2) <= (X1 != X2) & (X1 != X2+N) & (X1 != X2-N)
# give me one solution to the 8-queen puzzle
print(queens(X0,X1,X2,X3,X4,X5,X6,X7).data[0])
(4, 2, 0, 6, 1, 7, 5, 3)
```

5 Johtopäätökset ja pohdinta

Tässä opinnäytetyössä oli tavoitteena vertailla erilaisia tapoja ratkaista rajoitelaskentaongelmia Python-kielellä ja arvioida niiden soveltuvuutta opetuskäyttöön. Työssä esitellään kolme eri tapaa, joista algoritmien käyttö on ainoa, jossa opiskelija tekee rajoitelaskennan toiminnallisuudet itse. Tämä tapa on aluksi työläämpi ja mahdollisesti myös aluksi vaikeampi ymmärtää, mutta sitä käytetään lopulta melko samalla tavalla kuin kahta Python-kirjastoa. Verrattuna näihin, koska opiskelija kirjoittaa algoritmit itse, on sen soveltamista myös suhteellisesti helpompi ymmärtää. Mikäli opetuksessa halutaan painottaa erityisesti rajoitelaskennan oppimista, eikä pelkästään osana laajempaa tekoälykokonaisuutta, on tämä opettavaisin vaihtoehto.

Kahdessa muussa esitellyssä tavassa käytettiin valmiita Python-kirjastoja, pyDatalog ja PyCSP^3. Näistä ensimmäinen painottaa helppoa opittavuutta ja koodin luettavuutta. Sen komennot ovat nopeasti opittavia ja sen koodi on usein huomattavasti lyhyempää kuin perus-Pythonilla, joten pääpaino opetuksessa olisi koodin logiikan ymmärtämisessä. Kirjasto kärsii kuitenkin hyvän dokumentaation puutteesta, ja siitä ei ole julkaistu päivitettyä versiota vuoden 2016 jälkeen. Sen käyttö opetuksessa voisi olla jossain tapauksissa perusteltua, mutta on erittäin epävarmaa, kuinka hyödyllistä tämän kirjaston käytön osaaminen olisi opiskelun jälkeen.

PyCSP^3 on verrattuna erittäin tuore ja hyvin dokumentoitu kirjasto. Se on tehokas tapa saada ratkottua rajoitelaskentaongelmia pienellä määrällä koodia. Sen soveltuminen opetukseen onkin riippuvainen siitä, mikä painoarvo Pythonilla on opetuksessa. Koska tätä kirjastoa käytettäessä vain ongelman mallinnus tapahtuu Pythonilla, tarvitaan ulkopuolinen ohjelma ratkaisemaan ongelma. PyCSP^3 on tehokas väline näiden ongelmien ratkontaan, mutta opetuskäytössä ei anna aina parasta kuvaa siitä, miten niitä ratkaistaan.

6 Yhteenveto

Opinnäytetyö vastasi tutkimuskysymyksiinsä mielestäni melko hyvin. Rajoitelaskenta ja etenkin tekoäly yleisemmin on niin monimutkainen aihe, että työtä olisi voinut jatkaa vielä paljon pidemmälle aikataulun salliessa. Tämän monimutkaisuuden takia opinnäytetyön teko oli usein hidasta, etenkin tiedonkeruuvaiheessa. Pyrin työssäni tiivistämään paljon tietoa, tavoitteena saada aiheesta kiinnostuneelle mutta tietämättömälle lukijalle ymmärrys oleellisimmasta tiedosta.

Opinnäytetyötä voisi kehittää jatkossa vertailemalla useampia eri ratkaisutapoja ja Python-kirjastoja. Toinen lähtökohta olisi syventyä enemmän rajoitelaskennan ja tekoälyn tarjoamien mahdollisuuksien soveltuvuudesta opetukseen. Tässä tapauksessa voitaisiin kartoittaa esimerkiksi mahdollista kurssiohjelmaa ja luoda opiskelijaryhmille malliprojekteja ja -tehtäviä.

Itselleni opinnäytetyön teko oli erittäin opettavaista. En tuntenut aihetta juuri ollenkaan aloittaessani, joten tekoprosessi oli paljon uuden tiedon käsittelemistä. Tein työni aiheesta koska tekoäly kiinnosti minua ja olin utelias oppimaan miten sellainen sisäisesti toimii. Tekoäly tulee varmasti olemaan tärkeä osa tulevaisuuttamme, joten tässä oppimani tieto voi hyvinkin olla avuksi tulevaisuuden työelämässäni. Opinnäytetyö on myös pisin kirjoittamani teksti, ja sen teko opetti valtavasti tieteellisestä tekstistä ja sen kirjottamisesta.

Lähteet

Artasanchez, A., & Joshi, P. (2020). *Artificial intelligence with Python : Your complete guide to building intelligent apps using Python 3.x*.

Boden, M. A. (2006). *Mind as Machine: A History of Cognitive Science*.

Hyvönen, E. (2004). *Tekoäly 2004*. Helsingin Yliopisto. https://www.cs.helsinki.fi/u/eahyvone/courses/tekoaly04/

Kopec, D. (2019). *Classic Computer Science Problems in Python*.

Lecoutre, C., & Szczepanski, N. (2020). *PyCSP 3 Modeling Combinatorial Constrained Problems in Python*. https://github.com/xcsp3team/pycsp3

McCorduck, P. (2004). *Machines Who Think*. https://doi.org/10.1201/9780429258985

Montonen, S. (n.d.). *Miten nappulat liikkuvat*. Haettu 6.3.2021 osoitteesta https://peda.net/p/samu.montonen/shakki/säännöt/mnl

Moor, J. (2006). The Dartmouth College Artificial Intelligence Conference: The Next Fifty Years. In *AI Magazine* (Vol. 27, Issue 4). https://doi.org/10.1609/AIMAG.V27I4.1911

Poole, D., Mackworth, A., & Goebel, R. (1998). *Computational Intelligence: A Logical Approach*. https://www.cs.ubc.ca/~poole/ci/lectures/lectures.html

pyDatalog. (n.d.). *Datalog logic programming in python*. Haettu 7.3.2021 osoitteesta https://sites.google.com/site/pydatalog/home

PyDatalog. (n.d.). *Datalog tutorial*. Haettu 7.3.2021 osoitteesta https://sites.google.com/site/pydatalog/Online-datalog-tutorial

Python Land. (2021). *What is Python: the major features and what it's used for*. https://python.land/python-tutorial/what-is-python

Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach, 3. painos*. https://doi.org/10.5555/1671238

Summerfield, M. (2010). *Programming in Python 3: A Complete Introduction to the Python Language*. e

Tekoäly.info. (n.d.). *Tekoälyn historia*. Haettu 1.3.2021 osoitteesta https://tekoäly.info/tekoaly_historia/

Tsang, E. (1993). *Foundations of Constraint Satisfaction: Computation in Cognitive Science*.

Warwick, K. (2012). *Artificial Intelligence: The Basics*. https://doi.org/10.4324/9780203802878

Liite 1: Aineistonhallintasuunnitelma

Tässä opinnäytetyössä kerättyä materiaalia syntyy varsin vähän, lähinnä rajoittuen tutkimusprosessin aikana tehtyihin muistiinpanoihin sekä Python-ohjeistuksen teossa syntyneisiin koodeihin. Opinnäytetyö voi myös sisältää havainnollistavia kuvia ja kaavioita. Valokuvia tässä työssä tuskin tullaan käyttämään. Myös ohjeistava Python-osio voi mahdollisesti sisältää havainnollistavia kuvakaappauksia koodinkirjoitusprosessista esimerkkikoodien lisäksi.

Kaikki opinnäytetyön teossa syntyvä aineisto säilytetään tekijän kannettavan tietokoneen C-asemalla työn tekoprosessin ajan. Aineistoa säilytetään ainakin 1 vuosi opinnäytetyön hyväksymispäivästä, jotta opinnäytetyön tulokset voidaan tarvittaessa varmistaa. Aineisto myös varmuuskopioidaan OneDrive-pilvipalveluun päivittäin sekä USB-muistitikulle viikoittain.

Tässä työssä ei tulla käsittelemään henkilötietoja tai muita luottamuksellisia tai arkaluontoisia tietoja, eikä aineistoa siten ole tarpeen tuhota säilytysjakson päätyttyä.