Iikka Hämäläinen

Työvuorojen aikataulutuksen optimointi PuLP-kirjaston avulla

Tietotekniikan kandidaatintutkielma

3. toukokuuta 2021

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Iikka Hämäläinen

Yhteystiedot: iikka.a.j.hamalainen@student.jyu.fi

Ohjaaja: Tuomo Rossi

Työn nimi: Työvuorojen aikataulutuksen optimointi PuLP-kirjaston avulla

Title in English: Workforce scheduling optimisation using PuLP library

Työ: Kandidaatintutkielma

Opintosuunta: Kaikki opintosuunnat

Sivumäärä: 22+20

Tiivistelmä: Työvuorojen aikatauluttaminen on käytännön ongelma, jonka tekeminen käsin on työlästä. Ongelma voidaan ratkaista kirjoittamalla aikataulujen automatisointiin ja optimointiin soveltuva tietokoneohjelma. Tässä tutkielmassa selvitetään, kuinka lineaarisen kokonaislukuoptimoinnin menetelmiä voidaan hyödyntää työvuorojen aikatauluttamisessa. Optimointitehtävän ohjelmointiin käytetään Python-kieltä ja PuLP-kirjastoa.

Avainsanat: lineaarinen optimointi (LP), lineaarinen kokonaislukuoptimointi (ILP), lineaarinen sekalukuoptimointi (MILP), lineaarinen monitavoiteoptimointi (MOLP), Python, PuLP

Abstract: Workforce scheduling is a real life issue that is troublesome to perform manually. It's possible to automate and optimise the scheduling simultaneously by writing a computer program designed to resolve this issue. In this thesis it's examined how linear integer programming methods can be applied in workforce scheduling. Python and PuLP are used to write the optimisation program.

Keywords: linear optimisation/programming (LP), integer linear programming (ILP), mixed-integer linear programming (MILP), multi-objective linear programming (MOLP), Python, PuLP

Sisällys

1 Johdanto

Työvuorojen aikatauluttaminen käsin on vaivalloista ja aikataulun luomiseen voi mennä useita tunteja, kun työntekijöitä on paljon. Lisäksi käsin tehty aikataulu johtaa usein huonoihin lopputuloksiin. Molemmat ongelmat saadaan ratkaistua ohjelmoimalla aikatauluttamisen automatisoiva ratkaisu. Automatisoimalla voidaan ottaa huomioon työntekijöiden toiveet niin, etteivät kustannukset nouse liikaa.

Tässä tutkielmassa selvitetään, miten lineaarisen kokonaislukuoptimoinnin menetelmiä voidaan soveltaa työvuorojen aikatauluttamiseen PuLP-kirjaston avulla. Tutkielman tavoitteena on toteuttaa ohjelmakoodi, joka ratkaisee tosielämän työvuorojen aikataulutusongelman. Ohjelmakoodi toteutetaan lineaarisesta optimoinnista ja työvuorojen aikataulutuksesta tehdyn kirjallisuuden ja tutkimuksen avulla. Tutkielmassa yhdistetään kirjallisuuden ja tutkimuksen menetelmiä sekä arviointikriteereitä valituilla teknologioilla toteutetussa ohjelmakoodissa.

Tutkielman lukijan oletetaan tuntevan tietotekniikan ja tietojenkäsittelytieteen perustiedot, kuten termit olioparadigma, kirjasto, luokka, silmukka ja parametri. Näiden lisäksi lukijan oletetaan tuntevan lineaarialgebran perusteet.

Lineaarinen kokonaislukuoptimointi on matemaattisen optimoinnin osa-alue. Se on menetelmä, jolla etsitään lineaariselle funktiolle parasta mahdollista arvoa, kun funktion kuvajoukkoa on rajoitettu lineaarisella yhtälö- tai epäyhtälöryhmällä (Tang 1999, s. 1). Muut tutkielman avainsanoissa mainitut optimoinnin menetelmät ovat lineaarisen optimoinnin osa-alueita, jotka käydään läpi tarkemmin luvussa 2.

Python on korkean tason ohjelmointikieli ja PuLP sille tehty kirjasto, joka ratkaisee lineaarisia optimointitehtäviä. Tämän tutkielman ohjelmakoodi kirjoitetaan Pythonilla ja se käyttää PuLP-kirjastoa optimointitehtävän ratkaisemiseen.

Tutkielma alkaa lineaarisen optimoinnin määrittelyllä luvussa 2. Luvussa esitellään myös tarvittavat lineaarisen optimoinnin osa-alueet ja havainnollistetaan asiaa esimerkein. Luvussa 3 tarkastellaan työvuorojen optimoinnista tehtyä tutkimusta ja kirjallisuutta. Samalla pohditaan, mitä työvuoro-optimointiohjelman toteutuksessa on otettava huomioon. Luvussa 4

esitellään Python ja PuLP tarkemmin. Luvussa myös pohditaan, miksi nämä teknologiat soveltuvat tutkielman toteutukseen omien ominaisuuksiensa puolesta.

Luvussa 5 määritellään aluksi toteutettavan ohjelman vaatimukset. Sitten perustellaan miten ja miksi on tehty valinnat, joilla vaatimuket on pyritty täyttämään. Myöhemmin esitellään ohjelman rakenne ja mallinnetaan ohjelman toteuttama lineaarinen optimointimalli matemaattisesti. Lopuksi arvioidaan, miten toteutus vastaa ohjelmalle asetettuja vaatimuksia.

Toteutuksen tuloksia arvioidaan luvussa 6. Ohjelman testisuorituksista saatuja tuloksia analysoidaan ja samalla pohditaan, kuinka hyvin tulokset vastaavat ohjelman vaatimuksiin. Samalla pohditaan keinoja, joilla saatuja tuloksia voidaan parantaa. Lopuksi luvussa 7 pohditaan, mitä tuloksista ja toteutuksen ominaisuuksista voidaan päätellä sekä kuinka hyvä valinta Python ja PuLP ovat työvuorojen optimointiin.

2 Lineaarinen optimointi

Lineaarisella optimoinnilla tarkoitetaan matemaattisia malleja, joilla voidaan optimoida lineaarisen funktion arvoa rajoitefunktioista koostuvan joukon puitteissa (Tang 1999, s. 1; Borne ym. 2013, s. 1). Sen yleisiä sovelluksia ovat liiketalouden voittojen maksimointi sekä energiankulutuksen tai kulujen minimointi. Dantzig (2016, s. 1) kertoo, että lineaarinen optimointi sai alkunsa toisen maailmansodan aikana, kun monet valtiot halusivat optimoida muun muassa sotilasoperaatioiden, viljelyn ja laivareittien suunnittelua. Sodan jälkeen esitettiin ensimmäiset optimointitehtävien matemaattiset mallinnustavat ja simplex-algoritmi niiden ratkaisemiseen (Dantzig 2016, s. 1).

Lineaarinen optimointimalli muodostuu lineaarisesta tavoitefunktiosta ja joukosta lineaarisia rajoitteita. Tarkoituksena on optimoida tavoitefunktion arvo niin, että kaikki rajoitteet täyttyvät (Miller 2017, s. 47; Borne ym. 2013, s. 1). Dantzig (2016) huomauttaa, että tosielämän ilmiön voi harvoin tarkasti muotoilla matemaattiseksi malliksi. Esimerkiksi hän kertoo ruoka-aineiden ravintopitoisuudet, jotka vaihtelevat tosielämässä jatkuvasti. Mallin luojan vastuulle jää yksinkertaistaa tai jättää mallin ulkopuolelle liian monimutkaiset tosielämän ilmiöt (Dantzig 2016, s. 7).

Borne, Popescu, Filip ja Stefanoiu (2013, s. 1-3), Tang (1999, s. 1-2) sekä Brucker ja Knust (2012, s. 44) määrittelevät lineaarisen optimointimallin muodon seuraavasti:

$$\begin{cases} \text{min.} & z = c^T x \\ \text{s.e.} & Ax \geq b \\ & x \geq 0 \end{cases} \tag{2.1}$$

Ensimmäisellä rivillä on tavoitefunktio $z = c^T x$, jonka arvoa $z \in \mathbb{R}$ halutaan tässä tapauksessa minimoida. Vektori $x \in \mathbb{R}^n$ koostuu tehtävän päätösmuuttujista, joille etsitään optimaalisia arvoja. Vektori $c \in \mathbb{R}^n$ on vakiovektori, jonka alkiot painottavat päätösmuuttujia. Päätösmuuttujien arvoja rajoittavat tehtävän rajoitteet eli epäyhtälöt $Ax \geq b$ (Borne ym. 2013, s. 1-3; Tang 1999, s. 1-2). Lisäksi päätösmuuttujilla on usein epänegatiivisuusrajoite. Esimerkiksi hyödykkeiden tuotantomäärät eivät voi olla negatiivisia (Tang 1999, s. 2). Muodostettu tehtävä ratkaistaan esimerkiksi simplex-algoritmilla. Tässä tutkielmassa ei kuitenkaan tutki-

ta ratkaisijoiden toimintaa, joten simplexiä ei käydä läpi tarkemmin.

Optimointimallin rajoitteet saadaan esitettyä epäyhtälöiden sijaan yhtälöinä, kun lisätään rajoitefunktioihin puute- tai ylijäämämuuttujia (Dantzig 2016, s. 60). Kun mallin 2.1 rajoitefunktioihin lisätään ylijäämämuuttujat s, rajoitteet saavat muodon $Ax - s = b$. Millerin (2017) mukaan puutemuuttujien käyttö ei kuitenkaan ole aina tarpeellista, ja siitä voi olla jopa haittaa. Jos rajoitteita on paljon, ohjelman suoritusaika voi pidentyä merkittävästi muuttujien suuren määrän takia (Miller 2017, s. 73-74).

Monissa optimointitehtävissä vaaditaan, että osa päätösmuuttujista saa vain diskreettejä, kuten kokonaisluku- tai binääriarvoja (Tang 1999, s. 75). Tuotannon aikataulutusongelmat, joissa tuotteita on varastoitava kuljetusten välillä ovat hyvä esimerkki diskreettien muuttujien tarpeellisuudesta (Westerlund ym. 2007). Lisäksi on tehokasta käyttää binäärisiä päätösmuuttujia määrittämään, suoritetaanko jokin operaatio vai ei. Kun jatkuvia ja diskreettejä päätösmuuttujia käytetään samanaikaisesti, puhutaan sekalukuoptimoinnista (Tang 1999, s. 77-79).

Millerin (2017) määritelmän mukaan monitavoiteoptimointissa optimoidaan montaa tavoitefunktiota kerralla. Tehtävän ratkaisemiseksi määritellään tavallisesti funktioille epänegatiiviset painot $w_1, \ldots, w_n$ siten, että $\sum_{i=1}^{n} w_i = 1$. Painot lisätään mallin 2.1 mukaisiin tavoitefunktioihin, jolloin saadaan funktio $w_1 {c_1}^T x + \cdots + w_n {c_n}^T x$. Kun painot ja vektorit $c_1, \ldots, c_n$ esitetään lineaarikombinaationa $c_w = w_1 c_1 + \cdots + w_n c_n$, voidaan tavoitefunktio palauttaa tavallisen lineaarisen optimointitehtävän mukaiseksi muotoon ${c_w}^T x$ (Miller 2017, s. 153-154). Malli 2.1 monitavoitteiseksi kokonaislukutehtäväksi muutettuna olisi siten seuraavanlainen:

$$\begin{cases} \text{min.} & z = {c_w}^T x \\ \text{s.e.} & Ax \geq b \\ & x \in \mathbb{Z}^n \end{cases} \tag{2.2}$$

Monitavoitteisen tehtävän painot määritellään ja priorisoidaan sen perusteella, mitkä alkuperäisistä tavoitefunktioista ovat päätöksentekijälle tärkeimpiä (Miller 2017, s. 154). Kun painot on määritelty, lopullinen tavoitefunktio on tavoitellussa muodossa ja rajoitteet pysyvät ennallaan. Lopuksi rajoitetaan päätösmuuttujien arvot positiivisiin kokonaislukuihin. Malli 2.2 sisältää lopulta malliin 2.1 nähden vain ylimääräisen kokonaislukurajoitteen, mutta on muuten samassa muodossa. Näin malli voidaan ratkaista lineaarisella ratkaisijalla.

3 Työvuorojen optimointi

Työvuorojen aikataulutus on hyödyllistä automatisoida. Dantzig (2016) toteaa, että tehtävien automatisointi helpottaa ihmisen taakkaa ja voi olla suureksi avuksi päätösten tekemisessä. Labidi, Mrad, Gharb ja Louly (2014) esittävät automatisoitua ratkaisua ongelmaansa tulosten parantamiseksi. Aikataulutuksen automatisoinnista ja optimoinnista voi siis olla hyötyä sekä ajankäytön että tulosten suhteen.

Labidin ym. (2014) mukaan työvuorojen optimointi voidaan tavallisesti toteuttaa lineaarisena optimointitehtävänä. Työvuoro-optimoinnille on löydetty sovelluksia monilla aloilla, kuten sairaaloissa tai lentokentillä (Labidi ym. 2014). Optimointimallit ovat kuitenkin monesti monimutkaisia ja vaikeita muodostaa esimerkiksi osa-aikatyön, useamman sijainnin tai laissa säädettyjen rajoitteiden vuoksi (Shahnazari-Shahrezaei, Tavakkoli-Moghaddam ja Kazemipoor 2013; Soukour ym. 2012; Mohan 2008).

Kun otetaan sekä työnantajan että -tekijöiden mieltymykset huomioon työvuorolistan laatimisessa, työn tuottavuus paranee (Shahnazari-Shahrezaei, Tavakkoli-Moghaddam ja Kazemipoor 2013). Myös Labidin ym. (2014) määritelmän mukaan hyvä työvuoroaikataulutus muun muassa tasapainottaa osapuolten toiveita. He esittävät mallia, joka tuottaa käsin aikataulutettuja parempia työvuorolistoja. Lisäksi ratkaisu saavutetaan huomattavasti käsin tehtävää nopeammin (Labidi ym. 2014). Molemmat lähteet ovat yhtä mieltä siitä, että työntekijöiden toiveita huomioonottava työvuorojen aikataulutus on myös yritykselle hyväksi.

Osa-aikaisten työntekijöiden mieltymykset tulisi siis huomioida optimointimallin suunnittelussa. Mohanin (2008) mukaan suurin osa tutkimuksesta keskittyy kokoaikaisten työntekijöiden aikatauluoptimointiin. Myös osa-aikatöitä koskevasta tutkimuksesta merkittävä osa on suunniteltu minimoimaan osa-aikatyöntekijöiden määrä eikä tutkimuksissa välttämättä oteta riittävästi huomioon työntekijöiden omia toiveita (Mohan 2008). Työntekijöiden saatavuus voi lisäksi vaihdella ajanjaksojen välillä huomattavasti (Glover ja McMillan 1986). Mohan (2008) painottaa, että vaihtelu on erityisen suurta osa-aikaisilla työntekijöillä. Optimointimallia suunnitellessa on siis otettava saatavuusrajoitteet tarkasti huomioon ja varmistettava mallin soveltuvuus osa-aikaisille työntekijöille.

4 Python ja PuLP

Python on suosittu, luettavuuteen keskittyvä ohjelmointikieli. Pythonin syntaksi on selkeä ja kieli on helppo oppia. Se korostaa ylläpidettävyyttä ja yksinkertaista, oliopohjaista lähestymistapaa (Mitchell, O'Sullivan ja Dunning 2011). Pythonia ei siksi selitetä tai tarkastella tässä tutkielmassa syvemmin. Tutkielmassa käytetään Pythonin versiota 3.8.

Optimointimallin ohjelmointiin on valittu tässä tutkielmassa Python-kielen kirjasto PuLP, josta käytetään versiota 2.4. Mitchellin, O'Sullivanin ja Dunningin (2011) mukaan se on täysin Pythonilla kirjoitettu avoimen lähdekoodin kirjasto. PuLP keskittyy pysymään lähellä Pythonin syntaksia ja konventioita. Ohjelmoitu malli muistuttaa matemaattista kaavaa erittäin paljon, ja Pythonin syntaksista poikkeavaa toiminnallisuutta on vältetty mahdollisuuksien mukaan. Syntaksin yksinkertaisuuden varmistamiseksi PuLP on luotu tukemaan vain lineaarisia- ja sekalukuoptimointimalleja. Epälineaaristen mallien tukemisen olisi katsottu monimutkaistavan kirjastoa liikaa (Mitchell, O'Sullivan ja Dunning 2011).

PuLP ei anna käyttäjänsä luoda täysin abstrakteja optimointitehtäviä. Abstrakteilla tehtävillä tarkoitetaan malleja, joiden komponentteja ei ole etukäteen määritelty. Vaikka Mitchell ym. (2011) arvioivat tämän teoriassa rajoittavan käyttäjää, he pitävät Pythonia ratkaisuna ongelmaan. Monimutkaisten mallien ohjelmointi PuLP-kirjastolla on mahdollista Pythonin ominaisuuksien avulla, joten PuLP:in ei tarvitse erikseen sallia abstrakteja optimointimalleja (Mitchell, O'Sullivan ja Dunning 2011). Esimerkkinä Mitchell ym. (2011) mainitsevat mahdollisuuden korvata muuttuja funktiolla, jonka paluuarvoa lopulta käytetään ohjelman suorituksen aikana. Tämä mahdollistaa siten käytännössä abstraktien mallien muodostamisen (Mitchell, O'Sullivan ja Dunning 2011).

Sekä Python että PuLP ovat Mitchellin ym. (2011) mukaan erittäin sallivasti lisensoituja. PuLP on myös käytettävissä erittäin laajasti, koska Python toimii monella alustalla, ja molemmat työkalut ovat ilmaisia. PuLP on rakennettu ylimääräisiä riippuvuuksia välttäen, jotta pelkkä Python-tulkki riittäisi kirjaston käyttöön (Mitchell, O'Sullivan ja Dunning 2011). Python ja PuLP ovat siten monipuolisuutensa ja helppolukuisuutensa vuoksi hyvä vaihtoehto tutkielman toteutuksessa käytettäväksi teknologiaksi.

PuLP on valittu tämän tutkielman teknologiaksi myös laajennettavuutensa ansiosta. Se sisältää valmiiksi sisäänrakennettuja ratkaisijoita ja uusien integroiminen on tehty käyttäjälle helpoksi (Mitchell, O'Sullivan ja Dunning 2011). Kirjaston avulla ohjelmoitu optimointimalli on helposti ylläpidettävä Pythonin tyylikonventioiden (Mitchell, O'Sullivan ja Dunning 2011) ja matemaattista mallia muistuttavan esitystapansa ansiosta.

Seuraavaksi esitellään mallin 2.1 mukainen kahden päätösmuuttujan esimerkkiohjelma:

```
1  from pulp import *
2
3  # Luodaan tehtävä ja muuttujat
4  problem = LpProblem(name='ExampleProblem', sense=LpMinimize)
5  x1 = LpVariable('x1', lowBound=0)
6  x2 = LpVariable('x2', lowBound=0)
7
8  # Lisätään tavoitefunktio ja kaksi rajoitetta
9  problem += 5 * x1 + 2 * x2
10 problem += x1 + 2 * x2 >= 13
11 problem += 5 * x1 + x2 >= 30
12
13 # Ratkaistaan tehtävä
14 problem.solve()
15
16 # Tavoitefunktion ja päätösmuuttujien arvot nähdään
17 # niiden ominaisuuksista
18 print('z =', value(problem.object))
19 print(f'{x1.name}={x1.varValue}, {x2.name}={x2.varValue}')
```

Nähdään, että rivit 9-11 muistuttavat ulkoasultaan mallin 2.1 tavoitefunktiota ja rajoitteita. Ohjelmakoodin mallia vastaavat komponentit ovat tällöin $c = \begin{pmatrix} 5 & 2 \end{pmatrix}$, $x = \begin{pmatrix} x1 & x2 \end{pmatrix}$, $A = \begin{pmatrix} 1 & 2 \\ 5 & 1 \end{pmatrix}$ ja $b = \begin{pmatrix} 13 & 30 \end{pmatrix}$. Epänegatiivisuusrajoitteet on lisätty muuttujien määritelmissä riveillä 5-6. Monipuolisempien ohjelmien tavoitteet ja rajoitteet voivat kuitenkin olla dynaamisesti ajon aikana täytettyjä listoja, eivätkä silloin täysin vastaa matemaattista esitystapaa edellisen esimerkin mukaan.

5 Toteutus

5.1 Ohjelman vaatimukset

Hyvän optimointimallin piirteitä kuvaillaan kirjallisuudessa monin tavoin. Glover ja McMillan (1986) huomauttavat, että työntekijöiden taidot ja ominaisuudet muuttuvat, joten ohjelman täytyy sallia työntekijän ominaisuuksien muokkaaminen jälkikäteen. Tavoite saavutetaan, kun ohjelman tietorakenteet suunnitellaan niin, että työntekijöiden muokkaaminen, lisääminen ja poistaminen on helppoa. Ohjelmakoodi kykenee myös luomaan satunnaisen joukon työntekijöitä tarvittaessa, joten tämän tutkielman kannalta on tarpeetonta toteuttaa työntekijöiden tallentaminen kiintolevylle.

Labidi ym. (2014) saavuttavat mallissaan vähemmän aikataulutettua ylityötä käsin tehtyyn aikatauluun nähden ja ylityö on jaettu tasaisemmin työntekijöiden välillä. Heidän mallinsa ei myöskään tuota yli viiden työpäivän jaksoja kenellekään työntekijöistä, joten työntekijöiden tyytyväisyys pysyy hyvänä. Heidän tavoitteensa suhteen malli tuottaa käsin tehtyjä aikatauluja parempia ratkaisuja. Soukour, Devendeville, Lucet ja Moukrim (2012) arvioivat mallinsa laatua monilla kriteereillä, joista eräitä tärkeimpiä ovat palkkakustannukset ja työntekijöiden tyytyväisyys. Myös tämän tutkielman ohjelmassa pyritään saavuttamaan korkea työntekijöiden tyytyväisyys nostamatta palkkakustannuksia liikaa.

Toteutus pyritään pitämään laadukkaana myös ohjelmoinnin näkökulmasta. Tutkielman ohjelmakoodi kirjoitetaan englanniksi Python-kielen käytäntöjen mukaisesti (“PEP 8 – Style Guide for Python Code” 2001). Tutkielman muotoilun takia liitteenä A olevan ohjelmalistauksen rivien pituus ei voi olla yli 73 merkkiä. Lisäksi otetaan huomioon Mitchellin ym. (2011) mainitsema laajennettavuus ja ylläpidettävyys. Ohjelmakoodin toteutus pyritään pitämään helppona sisällyttää esimerkiksi graafiseen käyttöliittymään tai muuhun ohjelmistokokonaisuuteen. PuLP-kirjaston tavoin Pythonin standardikirjaston ulkopuolisia kirjastoja ei käytetä ohjelman siirrettävyyden parantamiseksi.

5.2 Ohjelman ominaisuudet

Ohjelmakoodi on kehitetty ratkaisemaan aikataulutusongelma, jolla on seuraavat ominaisuudet:

- Työntekijät ovat joko osa- tai kokoaikaisia.
- Työntekijät tekevät enintään yhden työvuoron päivässä.
- Työntekijät määrätään työvuoroon enintään seitsemänä päivänä peräkkäin.
- Työntekijöillä on vaihtelevat vähimmäis- ja enimmäismäärät viikoittaisia työtunteja.
- Työntekijöillä on vaihteleva enimmäismäärä viikoittaisia työvuoroja.
- Työntekijöille on varmistettava kaksi peräkkäistä vapaapäivää perjantain ja sunnuntain väliltä riittävän usein.
- Työntekijät voivat lisätä poissaoloja tai mieltymyksiä, jotka otetaan aikataulutuksessa huomioon.
- Työntekijöillä voi olla ominaisuuksia, jotka vaikuttavat aikataulutukseen.
- Työpaikalla on oltava jatkuvasti vähintään ennalta määrätty määrä työntekijöitä.

Tavoitteiden saavuttamiseksi ohjelmassa jaetaan Gloverin ja McMillanin (1986) mallista hieman poiketen työpäivä puolen tunnin mittaisiin jaksoihin. Lisäksi jokaiselle työntekijälle luodaan mahdollisten työvuorojen joukko (Glover ja McMillan 1986, s. 570). Tämän tutkielman tapauksessa työvuorojen joukko ei sisällä vuoroja, joiden aikana työntekijä on poissaolevana. Jokaista mahdollista työvuoroa vastaa päätösmuuttuja, joka saa kokonaislukuarvon nolla tai yksi. Työntekijä määrätään niihin työvuoroihin, joita vastaavat päätösmuuttujat saavat arvon yksi. Työntekijöille lisätään viikoittaiset vähimmäis- ja enimmäistyötunnit, työvuorojen enimmäismäärä ja muut edellä mainittuja ominaisuuksia vastaavat rajoitteet.

Monitavoitemallin tavoitefunktio muodostetaan Millerin (2017, s. 154) määritelmän mukaan yksittäisten tavoitefunktioiden painotettuna summana. Labidi ym. (2014), Mohan (2008) ja Soukour ym. (2012) käyttävät monitavoitteisia optimointimalleja, joissa painotetuilla tavoitefunktioilla maksimoidaan työntekijöiden tyytyväisyys ja minimoidaan ylimääräisten työtuntien määrä. Tämän tutkielman mallin tavoitefunktioilla maksimoidaan samoin työntekijöiden tyytyväisyys ja minimoidaan ylimääräiset työntekijät työpaikalla. Tavoitefunktioihin ei kuitenkaan tarvitse lisätä työntekijöiden merkitsemiä poissaoloja, koska poissaoloja vas-

taavia työvuoroja ei lisätä päätösmuuttujiin.

Ongelman luonteen vuoksi päätösmuuttujat rajoitetaan kokonaislukuarvoisiksi. Lisäksi suurin osa päätösmuuttujista on binäärilukurajoitteisia, mikä osaltaan helpottaa päätösmuuttujien kytkemistä toisiinsa lineaaristen rajoitteiden avulla. Bisschop (2021) esittelee tilanteen, jossa binäärimuuttuja asetetaan saamaan kahden muun binäärimuuttujan tulon arvo. Hän käyttää esimerkissään neljää lineaarista rajoitetta (Bisschop 2021, s. 84). Tämä vastaa loogista ”ja” -operaatiota. Esimerkistä saadaan helposti johdettua myös looginen ”tai” -operaatio.

5.3 Ohjelman mallinnus ja rakenne

Ohjelmassa toteutetaan työntekijöiden luomiseen, poistamiseen ja ylläpitoon tarvittavat luokat Employee ja Employees. Lisäksi toteutetaan työntekijöiden ominaisuuksien määrittelyyn Contract, PropertyFlag ja Preference -luokat. Ominaisuuksien määrittely tarkasti luokkarakenteen avulla tukee Pythonin aforismeja (“PEP 20 – The Zen of Python” 2004) ja johtaa hyvään luettavuuteen. Lopuksi luodaan luokka Schedule, joka suorittaa varsinaisen aikataulutuksen sille annetuilla parametreilla. Tarkastellaan seuraavaksi luokan matemaattisia ominaisuuksia.

Määritellään aluksi muuttujat ja ja käytettävät lyhenteet. Kaikki päätösmuuttujat ovat kokonaislukurajoitteisia ja suurin osa lisäksi binäärirajoitteisia.

E: työntekijöiden joukko
D: päivien joukko
K: viikkojen joukko
D_k: päivien joukko viikossa k
P_d: puolen tunnin jaksojen joukko päivänä d
S_{ed}: työntekijän e mahdollisten työvuorojen joukko päivänä d
Z_{edsp}: työntekijän e puolen tunnin jakson p sisältävien mahdollisten työvuorojen joukko päivänä d
o_{eds}: puolen tunnin jaksojen määrä vuorossa s
r_{edp}: työntekijän d puolen tunnin jaksolle p merkitty mieltymyskerroin, missä suurempi kerroin on huonommin sopiva vuoro

x_{eds}: binäärinen päätösmuuttuja, jonka ollessa 1 työntekijä e määrätään päivän d työvuoroon s

x_{edsp}: niiden muuttujien x_{eds} joukko, joita vastaavat työvuorot sisältävät puolen tunnin jakson p

x^o_{eds}: sisältää päivän d ensimmäisen puolen tunnin jakson sisältävät työvuorot s niiltä työntekijöiltä e, jotka voivat avata liikkeen

x^c_{eds}: sisältää päivän d viimeisen puolen tunnin jakson sisältävät työvuorot s niiltä työntekijöiltä e, jotka voivat sulkea liikkeen

y_{dp}: päätösmuuttuja, joka esittää ylimääräisten työntekijöiden määrää työpaikalla päivän d puolituntisjaksona p

f_{ed}: binäärinen päätösmuuttuja, jonka ollessa 1 työntekijällä e on vapaapäivä päivänä d

g_{ed}: binäärinen päätösmuuttuja, jonka ollessa 1 työntekijällä e on vapaapäivä päivinä d ja $d+1$

g^p_{ek}: g, joka koskee viikon k perjantaita ja lauantaita

g^l_{ek}: g, joka koskee viikon k lauantaita ja sunnuntaita

g^r_{ek}: g, joka koskee satunnaista muuttujaa g_{ed} viikolla k

h_{ek}: binäärinen päätösmuuttuja, jonka ollessa 1 työntekijällä on kaksi peräkkäistä vapaapäivää perjantain ja sunnuntain välillä aikataulutettavan ajanjakson viikolla k

H^v_e: jokainen h^v_e sisältää ne indeksit k, joita vastaavista muuttujista h_{ek} osan on saatava arvo 1

h^m_e: määrittää, kuinka monen h^v_e sisältämän muuttujan h_{ek} on saatava arvo 1

l_e: työntekijän e viikoittaisten puolen tunnin jaksojen vähimmäismäärä

u_e: työntekijän e viikoittaisten puolen tunnin jaksojen enimmäismäärä

m_e: työntekijän e viikoittaisten työvuorojen enimmäismäärä

q_e: työntekijän e meneillään oleva yhtäjaksoisten työpäivien määrä

v_{dp}: käynnissä olevien työvuorojen vähimmäismäärä päivän d puolen tunnin jaksona p

t: yhtäjaksoisten työpäivien enimmäismäärä

w_c: tavoitefunktion c painokerroin

Edellä luetellut muuttujat luodaan, jotta kaikki alaluvun 5.2 ominaisuudet voidaan ottaa huomioon. Ohjelmakoodissa päätösmuuttujien luominen tapahtuu kahden tai useamman sisäkkäisen silmukan sisällä. Silmukoissa luodaan PuLP-kirjaston LpVariable-luokasta päätös-

muuttujaoliot, jotka lisätään sopivaan tietorakenteeseen.

Seuraavaksi määritellään tavoitefunktiot, joita painotetaan kertoimilla $w_1, \ldots, w_4$. Tavoitefunktiot c_1 ja c_2 minimoivat työntekijöiden mieltymysten vastaisesti määrättyjen työvuorojen määrää ja ylimääräisten työntekijöiden määrää työpaikalla. Tavoitefunktiot c_3 ja c_4 saavat negatiivisia arvoja ja siten maksimoivat peräkkäisten vapaapäivien ja vapaiden viikonloppujen määrää, jotta työntekijöiden vapaat olisivat tasaisempia ja yhtäjaksoisempia. Esitellään tavoitefunktiot ja lopuksi niiden kaavan 2.2 mukainen summa c_{tot}:

$$c_1 = w_1 \sum_{i=1}^{e} \sum_{j=1}^{d} \sum_{k=1}^{s} r_{ijk} x_{ijk}$$
$$c_2 = w_2 \sum_{i=1}^{d} \sum_{j=1}^{p} y_{ij}$$
$$c_3 = -w_3 \sum_{i=1}^{e} \sum_{j=1}^{k} g^r_{ij}$$
$$c_4 = -w_4 \sum_{i=1}^{e} \sum_{j=1}^{k} h_{ij}$$
$$c_{tot} = \sum_{i=1}^{4} c_i$$

Tavoitefunktioiden osalta luvun 5.1 vaatimukset on toteutettu, eli halutut seikat on otettu huomioon jossakin tavoitteessa. Päätöksentekijän vastuulle jää asettaa sopivat painokertoimet kullekin tavoitefunktiolle.

Lopuksi luodaan optimointimallin rajoitteet:

1. $\sum_{i=1}^{e} \sum_{j=1}^{s} x_{idjp} - y_{ip} = v_{ip}$, kaikille d ja p
2. $\sum_{i=1}^{e} \sum_{j=1}^{s} x^o_{idj} \geq 1$, kaikille d
3. $\sum_{i=1}^{e} \sum_{j=1}^{s} x^c_{idj} \geq 1$, kaikille d
4. $\sum_{i=1}^{s} x_{edi} + f_{ed} = 1$, kaikille e ja d
5. $\sum_{i=1}^{d_k} \sum_{j=1}^{s} o_{eij} x_{eij} \geq l_e$, kaikille e ja viikoille k

6. $\sum_{i=1}^{d_k} \sum_{j=1}^{s} o_{eij} x_{eij} \leq u_e$, kaikille e ja viikoille k
7. $\sum_{i=1}^{d_k} \sum_{j=1}^{s} x_{eij} \leq m_e$, kaikille e ja viikoille k
8. $\sum_{i=d-t}^{d} f_{ei} \geq 1 : i \geq 1$, kaikille e ja $d \geq t - q_e$
9. $g_{ed} \leq f_{ed}$, kaikille e ja $d_1, \ldots, d_{n-1}$
10. $g_{ed} \leq f_{e(d+1)}$, kaikille e ja $d_1, \ldots, d_{n-1}$
11. $g_{ed} \geq f_{ed} + f_{e(d+1)} - 1$, kaikille e ja $d_1, \ldots, d_{n-1}$
12. $h_{ek} \geq g^p_{ek}$, kaikille e ja viikoille k
13. $h_{ek} \geq g^l_{ek}$, kaikille e ja viikoille k
14. $h_{ek} \leq g^p_{ek} + g^l_{ek}$, kaikille e ja viikoille k
15. $\sum_{i=1}^{k} h_{ei} \geq g^l_{ek}$, kaikille e ja viikoille k
16. $\sum_{j=1}^{h^v_{ei}} h_{eh^v_{eij}} \geq h^m_{ei}$, kaikille e ja $h^v_{ei} : i = 1, \ldots, n$
17. $x_{eds}, x_{edsp}, x^o_{eds}, x^c_{eds}, f_{ed}, g_{ed}, g^p_{ek}, g^l_{ek}, h_{ek} \in \{0,1\}$
18. $y_{dp} \in \mathbb{Z}$

Rajoitteet 1 varmistavat, että työpaikalla on jatkuvasti vaadittu määrä työntekijöitä ja asettavat samalla ylijäämämuuttujat y_{dp}. Rajoitteet 2 sekä 3 takaavat, että avatessa ja sulkiessa paikalla on ainakin yksi soveltuva työntekijä. Rajoitteet 4 määrittävät työntekijöille enintään yhden työvuoron päivää kohti ja asettavat samalla jäännösmuuttujan f_{ed}. Rajoitteet 5-7 rajoittavat työntekijöiden viikoittaisia työtunteja ja -vuoroja. Rajoitteet 8 varmistavat, ettei työntekijöille määrätä liian montaa peräkkäistä työpäivää. Rajoitteet 9-14 toimivat loogisina ”ja” ja ”tai” -operaattoreina vapaapäivien, vapaapäiväparien ja vapaiden viikonloppujen välillä. Rajoitteet 15 ja 16 varmistavat, että työntekijät saavat vapaaksi tarvittavat viikonloput. Lopuksi lisätään binääri- ja kokonaislukurajoitteet rajoitteissa 17-18.

Kaikki luvun alussa määritellyt muuttujat voidaan luoda tosielämän tilanteen mukaisiksi muuttamalla olioille annettavia parametreja. Rajoitteet on ohjelmoitu toimimaan kaikilla sellaisilla syötteillä, joilla aikataulutettavien päivien määrä on täysi viikko, eli seitsemän moninkerta. Jos epämääräisen mittaisten jaksojen aikatauluttaminen sallittaisiin, se vaatisi suuren määrän lisää muuttujia tallennettavaksi levylle suoritusten välillä. On suoraviivaisempaa

rajoittaa suoritus viikon mittaisiin ajanjaksoihin.

Työntekijöitä vastaavilla Employee-olioilla on ominaisuuksia, joista osa on enimmäkseen pysyviä ja toiset muutetaan ennen ohjelman suoritusta viikkoa tai muuta aikataulutusjaksoa vastaaviksi. Employee-oliolle on annettava ainakin tunniste, nimi, työsopimuksen tyyppi ja viikottaiset vähimmäistunnit. Loput ominaisuudet voidaan jättää oletusarvoisiksi. Käytännön tilanteessa on kuitenkin syytä asettaa ainakin suurin osa ominaisuuksista vastaamaan tosielämää.

Scheduler-oliolla on myös ominaisuuksia ja parametreja, jotka vaikuttavat lopulliseen aikatauluun. Niistä Employees-olio ja työvuorovaatimuksia esittävä matriisimuotoinen tietorakenne on asetettava aikatauluoliota luotaessa. Muut ominaisuudet voidaan jättää oletusarvoisiksi tai vaihtaa tilanteeseen sopiviksi. Näitä ovat tavoitefunktioiden painot, aikataulutusjakson ensimmäinen viikonpäivä, optimointitarkkuus ja aikaraja ratkaisijalle. Optimointitarkkuus on se parhaan tunnetun ja parhaan mahdollisen tavoitefunktion arvon suhteellinen ero, jolla malli palauttaa tuloksen. Lisäksi voidaan asettaa työvuorot alkamaan eri intervalleilla. Esimerkiksi intervalli 1 tarkoittaa, että työvuorojen on mahdollista alkaa minä tahansa puolen tunnin jaksona ja 2 joka toisena.

Tutkielman ohjelmakoodi on kuitenkin vain suppein mahdollinen toteutus, jolla saavutetaan tarvittavat tietorakenteet ja olioparadigmaa noudattava, aikataulutukseen kykenevä optimointimalli. Se ei ole tarkoitettu käytettäväksi sellaisenaan, vaan liitettäväksi vähintään komentorivipohjaiseen käyttöliittymään. Tarkka ja lopullinen ohjelman toimintalogiikka riippuu siitä, millaiseen tietojärjestelmään se sisällytetään. Seuraavassa luvussa tarkastellaan, millaisia testituloksia ohjelma tämänhetkisessä muodossaan tuottaa.

6 Tulokset

Ohjelmalle tehtiin kaksi testisuoritusta 3,4 gigahertsin Intel® Core™ i5-7500 -neliydinprosessorilla, kun käytettävisä oli 7,6 gigatavua vapaata keskusmuistia. Molemmissa suorituksissa painokertoimet olivat samat ja painottivat selkeästi eniten ylimääräisen työvoiman minimointia. Ensimmäisen testisuorituksen parametreina oli yhteensä 364 viikoittaista työtuntia työpaikan vaatimuksissa, kolmen viikon ajanjakso, vuorojen aloitusintervalli 1, optimointitarkkuus 0,1 ja 12 satunnaisesti luotua työntekijää, joiden keskimääräisten viikoittaisten työtuntien summa oli 395,5. Ratkaisijalle annettiin aikarajaksi 400 sekuntia. Toisen testisuorituksen parametreina oli yhteensä 1764 viikoittaista työtuntia vaatimuksissa, yhden viikon ajanjakso, vuorojen aloitusintervalli 1, optimointitarkkuus 0,15 ja 60 satunnaisesti luotua työntekijää, joiden keskimääräisten viikoittaisten työtuntien summa oli 1934,5. Aikarajaksi annettiin toisella testisuorituksella 1800 sekuntia.

Ensimmäinen testisuoritus tuotti työntekijöiden poissaolot huomioivan aikataulun, jossa työpaikalle määrättiin ylimääräisiä työntekijöitä yhteensä 45 tunnin ajalle. Ohjelma loi satunnaiset työntekijät, muodosti optimointimallin ja saavutti ratkaisun 4 minuutissa ja 42 sekunnissa. Työntekijöille aikataulutettiin vähimmäismäärä työtunteja ja vapaapäivät olivat usein ainakin kahden päivän peräkkäisiä jaksoja. Tulos oli työnantajan kannalta optimaalinen ja työntekijöiden kannalta sekä sallittu että toiveita huomioiva.

Toisen testisuorituksen aikataulu huomioi myös työntekijöiden toiveita, mutta työntekijöiden mieltymyksiä rikottiin usein. Työpaikalle määrättiin tällä suorituksella ylimääräisiä työntekijöitä yhteensä 75,5 tunnin ajalle ja työntekijät tekevät vähimmäismäärän työtunteja. Tulos olisi siis tasapainotettavissa paremmin asetetuilla painokertoimilla. Ohjelma loi satunnaiset työntekijät, muodosti optimointimallin ja saavutti ratkaisun 7 minuutissa ja 16 sekunnissa. Gloverin ym. (1986) mukaan testisuoritusta hieman suuremman aikataulun luomiseen voi kulua käsin 8-14 tuntia. Ohjelma siis suoriutuu tehtävästä merkittävästi nopeammin tuottaen silti riittävän hyviä aikatauluja, kun parametrit valitaan hyvin.

Millerin (2017, s. 153) mukaan lineaarisella optimointimallilla päästään erittäin lähelle parasta mahdollista ratkaisua kohtuullisessa ajassa. Ohjelman tuottama malli minimoi ylimää-

räiset työntekijät työpaikalla, kaikille työntekijöille aikataulutettiin vähimmäismäärä tunteja ja jokaisen puolen tunnin jakson ajalle aikataulutettiin riittävän monta työntekijää. Malli myös saavutti ratkaisun huomattavasti nopeammin, kuin käsin tehtynä olisi mahdollista. Voidaan päätellä, että malli toteuttaa aikataulun osalta laadukkaan tuloksen.

Labidin ym. (2014) mukaan hyvällä optimointimallilla voidaan tehdä aikataulu vaihteleville ajanjaksoille, mutta tämän tutkielman ohjelmalla voi käytännön syistä luoda aikataulun vain kaikille seitsemän päivän moninkerroille. Rajoittamaton aikataulutettavien viikkojen määrä tekee ohjelmasta kuitenkin riittävän joustavan. Vaikka ohjelmassa on ajanjaksojen suhteen pieni heikkous, ohjelman voidaan katsoa olevan riittävän laadukas myös tällä mittarilla.

Alaluvussa 5.1 mainitut laadukkaan optimointimallin ominaisuudet joko saavutetaan tai niiden saavuttamiseen tarvittavien komponenttien ohjelmointi on suoraviivaista. Mallin tavoitefunktio maksimoi työntekijöiden tyytyväisyyden sekä minimoi työmäärän ylimitoitusta, joten palkkakustannukset eivät kasva liikaa. Tavoitefunktioiden painotuksilla voidaan muokata tavoitteiden painoarvoja tarpeen mukaan, jotta lopputulokseen voidaan olla täysin tyytyväisiä.

7 Yhteenveto

Ohjelmakoodin ominaisuuksista ja tuloksista nähdään, että Python ja PuLP sopivat työvuoroaikataulutuksen automatisointiin ja optimointiin käytettäviksi teknologioiksi hyvin. Python antaa paljon vapauksia ohjelmakoodin laajentamiseen. Jos esimerkiksi halutaan alustariippumaton ohjelmisto, tutkielman ohjelmakoodia voidaan laajentaa alustariippumattomilla ratkaisuilla.

Tutkielman toteutus noudattaa kirjallisuudessa mainittuja laadukkaan mallin piirteitä ja päätyy kohtuullisessa ajassa hyvään lopputulokseen. Ohjelma on skaalautuva, mutta skaalautuvuuteen vaikuttaa käytettävissä oleva keskusmuistin määrä ja aikataulutukseen kuluva aika.

Mallin parametrien asettaminen voidaan nähdä heikkoutena, koska useiden minuuttien suoritusajan takia parametrien uudelleenasettaminen ja testaaminen on hidasta. Lisäksi päätösmuuttujien suuren määrän aiheuttama mahdollisuus muistin loppumiseen kuluttajatason tietokoneessa voi olla merkittävä ongelma joissain tilanteissa. Pitkien suoritusaikojen ja muistin rajallisuuden takia myös ohjelman testaus rajoittui vain kuuteenkymmeneen työntekijään.

Kuten aiemmin on todettu, ohjelma täyttää silti sille asetetut vaatimukset ja tuottaa heikkouksista huolimatta hyviä tuloksia aina, kun resursseja on tarpeeksi käytössä. Aikataulut ottavat koko- ja osa-aikaiset työntekijät huomioon ja työntekijöille aikataulutetaan usein monta peräkkäistä vapaapäivää yhden sijaan.

Python-pohjaisuus mahdollistaa ohjelman käytön monilla alustoilla erittäin helposti. Ohjelman oliopohjaisen rakenteen ansiosta se on käytettävissä sellaisenaan suuremmassa ohjelmistokokonaisuudessa. Ohjelmalle voidaan esimerkiksi luoda helposti graafinen käyttöliittymä ja tietojen tallentaminen levylle. Toteutus on siten hyödynnettävissä erittäin laajasti. Lopputuloksena voidaan todeta, että teknologiavalinta oli onnistunut ja lineaarisen kokonaislukuoptimoinnin keinoja voidaan soveltaa PuLP-kirjastolla työvuoroaikataulutuksessa helposti ja tehokkaasti.

Lähteet

Bisschop, Johannes. 2021. *AIMMS Optimization Modeling.* https://manual.aimms.com/_downloads/AIMMS_modeling.pdf.

Borne, Pierre, Dumitru Popescu, Florin Gheorghe Filip ja Dan Stefanoiu. 2013. *Optimization in Engineering Sciences : Exact Methods.* Toimittanut Pierre Borne. Hoboken, N.J.: ISTE Ltd/John Wiley & Sons Inc. https://doi.org/10.1002/9781118577899.

Brucker, Peter, ja Sigrid Knust. 2012. *Complex Job-Shop Scheduling,* 239–317. ISBN: 9783642239281. https://doi.org/10.1007/978-3-642-23929-8_4.

Dantzig, George. 2016. *Linear Programming and Extensions.* Princeton Landmarks in Mathematics and Physics. Princeton University Press. ISBN: 9780691059136. https://doi.org/10.7249/R366.

Glover, Fred, ja Claude McMillan. 1986. "The general employee scheduling problem. An integration of MS and AI". *Computers & Operations Research* 13 (5): 563–573. ISSN: 0305-0548. https://doi.org/10.1016/0305-0548(86)90050-X.

Labidi, M., M. Mrad, A. Gharbi ja M. A. Louly. 2014. "Scheduling IT staff at a Bank: A mathematical programming approach". *The Scientific World Journal* 2014. ISSN: 1537744X. https://doi.org/10.1155/2014/768374.

Miller, Steven J. 2017. *Mathematics of Optimization: How to do Things Faster.* Pure and Applied Undergraduate Texts v.30. AMS. ISBN: 9781470441142. http://search.ebscohost.com.ezproxy.jyu.fi/login.aspx?direct=true&db=nlebk&AN=1671153&site=ehost-live.

Mitchell, Stuart, Michael O'Sullivan ja Iain Dunning. 2011. "PuLP: A Linear Programming Toolkit for Python". *Department of Engineering Science, The University of Auckland,* http://www.optimization-online.org/DB_FILE/2011/09/3178.pdf.

Mohan, Srimathy. 2008. "Scheduling part-time personnel with availability restrictions and preferences to maximize employee satisfaction". *Mathematical and Computer Modelling* 48 (11-12): 1806–1813. ISSN: 08957177. https://doi.org/10.1016/j.mcm.2007.12.027.

"PEP 20 – The Zen of Python". 2004. Viitattu 26. huhtikuuta 2021. https://www.python.org/dev/peps/pep-0020/.

"PEP 8 – Style Guide for Python Code". 2001. Viitattu 7. huhtikuuta 2021. https://www.python.org/dev/peps/pep-0008/.

Shahnazari-Shahrezaei, Parisa, Reza Tavakkoli-Moghaddam ja Hamed Kazemipoor. 2013. "Solving a multi-objective multi-skilled manpower scheduling model by a fuzzy goal programming approach". *Applied Mathematical Modelling* 37 (7): 5424–5443. ISSN: 0307904X. https://doi.org/10.1016/j.apm.2012.10.011.

Soukour, Anas Abdoul, Laure Devendeville, Corinne Lucet ja Aziz Moukrim. 2012. *Staff scheduling in airport security service,* 14:1413–1418. PART 1. IFAC. ISBN: 9783902661982. https://doi.org/10.3182/20120523-3-RO-2023.00169.

Tang, S L. 1999. *Linear Optimization in Applications.* Hong Kong University Press. ISBN: 9789622094833. http://search.ebscohost.com.ezproxy.jyu.fi/login.aspx?direct=true&db=nlebk&AN=321900&site=ehost-live.

Westerlund, Joakim, Mattias Hästbacka, Sebastian Forssell ja Tapio Westerlund. 2007. "Mixed-time mixed-integer linear programming scheduling model". *Industrial and Engineering Chemistry Research* 46 (9): 2781–2796. ISSN: 08885885. https://doi.org/10.1021/ie060991a.

Liitteet

A Ohjelmalistaus

```
"""Program to automate and optimise a workforce schedule."""


import sys
import random
import time
from math import isclose
from string import ascii_lowercase
from enum import Enum, IntFlag, auto
from pulp import *


START_TIME = time.time()
DEFAULT_OPTIMISATION_ACCURACY = .15
ID_LOWER_BOUND = 10000000
ID_UPPER_BOUND = 99999999
NUMBER_OF_WORKDAYS = 7
MAXIMUM_CONSECUTIVE_WORKDAYS = 7
PERIODS_PER_HOUR = 2
SHIFT_START_INTERVAL = 1
DEFAULT_SHIFT_IN_PERIODS = 8 * PERIODS_PER_HOUR
MINIMUM_SHIFT_IN_PERIODS = 4 * PERIODS_PER_HOUR
MAXIMUM_SHIFT_IN_PERIODS = DEFAULT_SHIFT_IN_PERIODS
DEFAULT_WEEKLY_MAXIMUM_SHIFTS = 5
PREFERENCE_MULTIPLIER = 4
DEFAULT_WEIGHTS = {'preference': .25, 'day_pairs_off': .25,
                   'weekends_off': .25, 'excess_workforce': .25}
WEEKDAY_FRI = 4
WEEKDAY_SAT = 5
WEEKDAY_SUN = 6
RANDOM_CHANCES = {'absence': .05, 'preference': .06,
                  'open_and_close': .87, 'weekend': .1}


class Contract(Enum):
    """Represents contract types."""

    FULLTIME = 1
    PARTTIME = 2


class PropertyFlag(IntFlag):
    """Represents all special properties employees can have."""

    NONE = 0
    CAN_OPEN = auto()
    CAN_CLOSE = auto()
```

```
48      IS_STUDENT = auto()
49      IS_IN_SCHOOL = auto()
50      HAS_KEYS = auto()
51
52
53  class Preference(IntFlag):
54      """Represents employee preference or availability for a shift.
55
56      Undesirable flag also works as a dissatisfaction
57      factor in the objective function.
58      """
59
60      NORMAL = 0
61      UNAVAILABLE = 1
62      UNDESIRABLE = 8
63
64
65  class Employee:
66      """Employee class with required properties.
67
68      Attributes:
69          id:
70              Unique ID for every employee.
71          name:
72              A string representing the name of the employee.
73          type_of_contract:
74              An enumerator representing full-time or part-time contracts.
75          min_hours:
76              An integer defining the employee's minimum weekly hours.
77          max_hours:
78              An integer defining the employee's maximum weekly hours.
79              Defaults to match minimum hours.
80          max_shifts:
81              An integer representing the maximum amount of shifts per
82              week for the employee. Defaults to 5.
83          seniority:
84              A float representing employee's seniority. Defaults to 0.
85          special_properties:
86              PropertyFlag flags for special properties the employee
87              satisfies. For example can open or close business.
88          current_workday_streak:
89              An integer representing the length of the streak of days the
90              employee has worked at the end of the previous schedule.
91          weekends_config:
92              A dictionary where 'single' is a list of all weekend indices
93              that the employee must be have off. The item for the key
94              'groups' is a list of lists, where the first items of the
95              innermost lists define the minimum weekends off selected from
96              the list. The rest of the items are the weekend indices that
97              belong to the group.
98          preferences:
99              A dictionary for setting special preferences for shifts.
100             Defaults to an empty dictionary.
```

```
    """

    def __init__(self, new_id, name, type_of_contract, min_hours,
                 max_hours=None, max_shifts=None, seniority=None,
                 special_properties=None, current_workday_streak=None,
                 weekends_config=None, preferences=None):
        """Initialise employee."""
        self.id = new_id
        self.name = name
        self.type_of_contract = type_of_contract
        self.min_hours = min_hours
        self.max_hours = min_hours if (max_hours is None) else max_hours
        self.max_shifts = DEFAULT_WEEKLY_MAXIMUM_SHIFTS if (
            max_shifts is None) else max_shifts
        self.seniority = 0 if (seniority is None) else seniority
        self.special_properties = PropertyFlag.NONE if (
            special_properties is None) else special_properties
        self.current_workday_streak = 0 if (
            current_workday_streak is None) else current_workday_streak
        self.weekends_config = {} if (
            weekends_config is None) else weekends_config
        self.preferences = {} if (preferences is None) else preferences

    def to_text(self):
        """Return a text representation of employee."""
        min_h = int(self.min_hours / PERIODS_PER_HOUR)
        max_h = int(self.max_hours / PERIODS_PER_HOUR)
        hour_range = f'{min_h}-{max_h}'
        padding = ''
        if self.min_hours == self.max_hours and False:
            hour_range = str(int(self.min_hours / PERIODS_PER_HOUR))
            padding = '   '
        preferences_text = {}
        for day, day_preference in self.preferences.items():
            preferences_text[day] = {}
            for shift, flag in day_preference.items():
                preferences_text[day][shift] = int(flag)
        return str(f'ID: {self.id}, Name: {self.name}, ' +
                   f'Contract: {self.type_of_contract.name}, ' +
                   f'Hours: {hour_range},{padding} ' +
                   f'Max shifts: {self.max_shifts}, ' +
                   f'Seniority: {self.seniority}, ' +
                   f'Properties: {self.special_properties}, ' +
                   f'Streak: {self.current_workday_streak},   ' +
                   f'Weekends: {self.weekends_config}, ' +
                   f'Preferences: {preferences_text}')

    def set_employee_shifts(self, work_site_demands):
        """Find all employee's plausible shifts.

        Assign the list of shifts as an attribute to the employee.

        Args:
```

```
            work_site_demands:
                A list of tuples defining work site demands. Shifts
                will be generated in respect to opening hours.
        """
        all_shifts = []
        for day_index in range(len(work_site_demands)):
            todays_shifts = []
            days_preferences = None
            try:
                # Try to set preferences set for today.
                days_preferences = self.preferences[day_index]
            except KeyError:
                pass
            minimum_shift_length = MINIMUM_SHIFT_IN_PERIODS
            if self.special_properties & PropertyFlag.IS_IN_SCHOOL:
                minimum_shift_length = 2 * PERIODS_PER_HOUR
            for shift_length in range(minimum_shift_length,
                                      MAXIMUM_SHIFT_IN_PERIODS + 1):
                workday_periods = len(work_site_demands[day_index])
                todays_shifts += self.get_possible_shifts_for_day(
                    workday_periods, shift_length, days_preferences)
            all_shifts.append(todays_shifts)
        self.shifts = all_shifts

    def get_possible_shifts_for_day(self, number_of_periods,
                                    shift_length=None,
                                    days_preferences=None):
        """Get all consecutive defined-length sets of periods from a
        given set of periods.

        Args:
            number_of_periods:
                An integer representing the number
                of total periods on given day.
            shift_length:
                An integer defining the length of shift in periods.
            days_preferences:
                Employee's preferences for today.

        Returns:
            A list of possible shifts, unavailabilities factored in.
        """
        if shift_length is None:
            shift_length = DEFAULT_SHIFT_IN_PERIODS
        if days_preferences is None:
            days_preferences = {}
        possible_shifts = []
        for i in range(0, number_of_periods - shift_length + 1,
                       SHIFT_START_INTERVAL):
            eligible_shift = True
            try:
                for shift_index in days_preferences:
                    if (
```

```
                                (shift_index >= i) and
                                (shift_index < i + shift_length) and
                                (days_preferences[shift_index] ==
                                    Preference.UNAVAILABLE)
                        ):
                            eligible_shift = False
                except (AttributeError):
                    # Expected if preferences is not defined
                    # or it's an empty dictionary.
                    pass
                if eligible_shift:
                    shift_as_periods = [x for x in range(i,
                                                         i + shift_length)]
                    possible_shifts.append(shift_as_periods)
            return possible_shifts


class Employees:
    """Maintains a list of employees.

    Capable of creating random lists for testing purposes.

    Attributes:
        list:
            A dictionary of current employees. Keys are employee IDs and
            items instances of Employee class. Named in a potentially
            confusing way. Might need renaming.
    """

    def __init__(self, employee_list=None):
        """Initialise the object with an existing dictionary of
        employees or empty dictionary."""
        self.list = {} if (employee_list is None) else employee_list

    def count(self):
        """Return the number of current employees."""
        return len(self.list)

    def add(self, employee):
        """Add employee to dictionary.

        Args:
            employee: Employee instance to add to the dictionary.

        Returns:
            A boolean value. True if adding successful, False if not.
        """
        if not isinstance(employee, Employee):
            return False
        self.list[employee.id] = employee
        return True

    def remove(self, employee):
```

```
260             """Remove employee from dictionary.
261
262             Args:
263                 employee:
264                     Employee instance to remove from dictionary.
265             """
266             self.list.pop(employee.id, None)
267
268         def generate_employee_id(self):
269             """Create ID for employee randomly.
270
271             Return None if no unique ID is available after a
272             pre-defined maximum amount of tries."""
273             maximum_iterations = 2500
274             for _ in range(maximum_iterations):
275                 new_id = random.randint(ID_LOWER_BOUND, ID_UPPER_BOUND)
276                 if not self.id_exists(new_id):
277                     return new_id
278             return None
279
280         def id_exists(self, checked_id):
281             """Check if ID exists within existing employees.
282
283             Args:
284                 checked_id: ID whose value is checked for uniqueness.
285             """
286             for _, employee in self.list.items():
287                 if checked_id == employee.id:
288                     return True
289             return False
290
291         def create_dummy_employees(self, count_of_employees,
292                                    work_site_demands, fixed_hours=False,
293                                    start_day=0):
294             """Create a random list of employees for testing purposes.
295
296             Args:
297                 count_of_employees:
298                     Number of employees to be created. If None, employees are
299                     created as long as needed to fulfill the total hours in
300                     work site demands.
301                 work_site_demands:
302                     List of tuples defining future work site schedules.
303                     Employee preferences are created in respect to
304                     opening hours.
305                 fixed_hours:
306                     A boolean defining if random employees will have a fixed
307                     number of hours in their contracts instead of a range.
308                     Defaults to False.
309                 start_day:
310                     Index of the weekday from where scheduling starts.
311                     Affects the number of full weekends.
312                     Defaults to 0 i.e. Monday.
```

```
313
314             Returns:
315                 Boolean value. True if employees' total hours are above
316                 the first week's demands.
317             """
318             fulfill_hours = False
319             if not count_of_employees:
320                 fulfill_hours = True
321                 count_of_employees = sys.maxsize
322             weeks_in_schedule = len(work_site_demands) / 7
323             total_weekly_hours = sum([sum(x) for x in work_site_demands])
324             total_weekly_hours /= weeks_in_schedule
325             employee_hours_currently = 0
326             seniors_created = 0
327             print('total needed hours:',
328                   total_weekly_hours / PERIODS_PER_HOUR)
329             extras = 0
330             needed_extras = 0
331             for i in range(count_of_employees):
332                 new_employee = self.create_random_employee(work_site_demands,
333                                                            fixed_hours,
334                                                            start_day)
335                 if new_employee is None:
336                     break
337                 if new_employee.seniority != 0:
338                     seniors_created += 1
339                 self.list[new_employee.id] = new_employee
340                 employee_hours_currently += (
341                     (new_employee.min_hours + new_employee.max_hours) / 2)
342                 # Add ~7% extra employees for more probable feasibility.
343                 if i % 15 == 0:
344                     needed_extras += 1
345                 if extras >= needed_extras:
346                     break
347                 if (
348                         (employee_hours_currently > total_weekly_hours) and
349                         fulfill_hours
350                 ):
351                     extras += 1
352             print('total (avg) hours :',
353                   employee_hours_currently / PERIODS_PER_HOUR)
354             if not seniors_created:
355                 _, random_employee = random.choice(list(self.list.items()))
356                 random_employee.seniority = 1
357             return employee_hours_currently >= total_weekly_hours
358
359         def create_random_employee(self, work_site_demands,
360                                    fixed_hours=False, start_day=0):
361             """Create random employee and return the instance.
362
363             Args:
364                 work_site_demands:
365                     List of tuples defining work site schedule. Preferences
```

```
            are created in respect to opening hours.
        fixed_hours:
            A boolean defining if the employee will have a fixed
            number of hours in their contract instead of a range.
            Defaults to False.
        start_day:
            Index of the weekday from where scheduling starts.
            Affects the number of full weekends.
            Defaults to 0 i.e. Monday.
    """
    contract_type = random.choice((Contract.FULLTIME,
                                   Contract.PARTTIME))
    if contract_type == Contract.FULLTIME:
        min_hours = 38 * PERIODS_PER_HOUR
        max_hours = (38 if fixed_hours else 40) * PERIODS_PER_HOUR
    else:
        min_hours = random.choice(range(15 * PERIODS_PER_HOUR,
                                        30 * PERIODS_PER_HOUR, 2))
        max_hours = random.choice(range(min_hours,
                                        30 * PERIODS_PER_HOUR, 2))
    if fixed_hours:
        min_hours = max_hours
    max_shifts = None
    if max_hours is None:
        pass
    elif max_hours < 20 * PERIODS_PER_HOUR:
        max_shifts = 4
    elif max_hours < 15 * PERIODS_PER_HOUR:
        max_shifts = 3
    random_id = self.generate_employee_id()
    if random_id is None:
        return None
    random_name = ''
    for _ in range(8):
        random_name += random.choice(ascii_lowercase)
    random_seniority = 1 if (random.random() < .05) else 0
    random_properties = PropertyFlag.NONE
    if random.random() < RANDOM_CHANCES['open_and_close']:
        random_properties += PropertyFlag.CAN_OPEN
        random_properties += PropertyFlag.CAN_CLOSE
    random_streak = random.choice([6] + 2 * [5] + 3 * [4] + (
        4 * [3]) + 5 * [2] + 6 * [1] + 7 * [0])
    random_weekends = {}
    if random.random() < RANDOM_CHANCES['weekend']:
        s = 1 if (start_day == WEEKDAY_SUN) else 0
        weekend_range = range(int(len(work_site_demands) / 7) - s)
        random_weekends['single'] = [random.choice(weekend_range)]
    random_weekends['groups'] = []
    if len(work_site_demands) / 7 > 3:
        week_count = int(len(work_site_demands) / 7)
        weekend_list = list(range(0, week_count))
        slice_length = 5
        groups = [weekend_list[i:i + slice_length] for i in (
```

```
                range(0, len(weekend_list), slice_length))]
            for split_group in groups:
                if random.random() < RANDOM_CHANCES['weekend']:
                    weekends_off = random.choice((1, 2))
                    random_weekends['groups'].append([weekends_off] + (
                        split_group))
        random_preferences = {}
        for i in range(len(work_site_demands)):
            rand = random.random()
            if rand < RANDOM_CHANCES['absence']:
                unavailable_period_index = random.choice(
                    range(len(work_site_demands[i])))
                random_preferences[i] = {unavailable_period_index: (
                    Preference.UNAVAILABLE)}
            elif (
                rand <
                RANDOM_CHANCES['absence'] + RANDOM_CHANCES['preference']
            ):
                undesirable_period_index = random.choice(
                    range(len(work_site_demands[i])))
                random_preferences[i] = {undesirable_period_index: (
                    Preference.UNDESIRABLE)}

        return Employee(random_id, random_name, contract_type, min_hours,
                        max_hours, max_shifts, random_seniority,
                        random_properties, random_streak,
                        random_weekends, random_preferences)


class Scheduler:
    """Scheduler class.

    Maximises employee preferences while minimising costs.

    Attributes:
        employees:
            Employees object from which the schedule will be generated.
        work_site_schedule:
            A list of tuples representing work site needs. Elements of
            the list are workdays and elements of the tuples are the
            required amount of employees for every period of the day.
            The length of the tuples are the total periods that the site
            is open for that day.
        weights:
            A length 3 tuple representing objective function weights.
            Format: (preference, shift lengths, excess workforce)
        start_day:
            An integer between 0-6, representing the weekday from where
            scheduling is started. Affects weekends allocation in the
            model.
        shift_start_interval:
            An integer defining the number of time periods between
            every new starting shift.
```

```
472             accuracy:
473                 A float representing the desired solver accuracy. As soon as
474                 the best known solution is within this fraction of the best
475                 possible solution, the solver will return.
476             time_limit:
477                 Maximum allowed running time in seconds.
478             debug:
479                 A boolean to define whether to print debug messages.
480                 Defaults to False
481         """
482 
483         def __init__(self, employees, work_site_demands, weights=None,
484                      start_day=None, shift_start_interval=None,
485                      accuracy=None, time_limit=None, debug=False):
486             """Initialise scheduler with list of employees."""
487             self.employees = employees
488             self.work_site_demands = work_site_demands
489             self.workday_count = len(work_site_demands)
490             self.workdays_period_demand = [(
491                 len(all_periods_needs)) for all_periods_needs in (
492                 work_site_demands)]
493             self.weights = weights
494             if (weights is None) or not isclose(sum(weights.values()), 1):
495                 print('No objective weights provided or their sum is not 1.',
496                       'Using defaults.')
497                 self.weights = DEFAULT_WEIGHTS
498             self.start_day = start_day if start_day else 0
499             self.shift_start_interval = SHIFT_START_INTERVAL if (
500                 shift_start_interval is None) else shift_start_interval
501             self.accuracy = accuracy if accuracy else (
502                 DEFAULT_OPTIMISATION_ACCURACY)
503             self.time_limit = time_limit
504             self.debug = debug
505             [print(x.to_text()) for _, x in self.employees.list.items()]
506 
507         def run(self, time_limit=None):
508             """Create LP problem from employees.
509 
510             Solve the created problem and return a schedule.
511 
512             Args:
513                 time_limit:
514                     Optional time limit in seconds. This overrides
515                     the time limit property for this run.
516             """
517             print('workdays:', self.workday_count)
518             decision_variables = self.create_lp_problem()
519             if not time_limit:
520                 time_limit = self.time_limit
521             self.problem.solve(PULP_CBC_CMD(gapRel=self.accuracy,
522                                             timeLimit=time_limit))
523             print(f'Solved in {time.time() - START_TIME}s')
524             self.print_results(decision_variables, self.workday_count / 7,
```

```
                            self.problem.status)

    def create_lp_problem(self):
        """Create the LP problem for this scheduler."""
        self.problem = LpProblem('schedule', LpMinimize)
        for _, employee in self.employees.list.items():
            employee.set_employee_shifts(self.work_site_demands)
        print(f'All shifts created in {time.time() - START_TIME}s')
        decision_variables = self.create_decision_variables()
        time_passed = time.time() - START_TIME
        print(f'Decision variables created in {time_passed}s')
        self.create_objective(decision_variables)
        print(f'Objective created in {time.time() - START_TIME}s')
        self.create_constraints(decision_variables)
        print(f'Constraints created in {time.time() - START_TIME}s')

        print('decision variables:', len(self.problem.variables()))
        print('constraints:', len(self.problem.constraints))
        return decision_variables

    def print_results(self, decision_variables, number_of_weeks=None,
                      status=None, print_daily=False):
        """Print out the results given by the solved model.

        Args:
            decision_variables:
                Problem decision variables
            number_of_weeks:
                Number of weeks scheduled. Defaults to None.
                If not provided, employee hours in console show
                the total for the whole schedule.
            status:
                Problem status.
            print_daily:
                If daily excess hours should be printed or not.
                Defaults to False.
        """
        x = decision_variables['shifts']
        y = decision_variables['workforce']
        d = decision_variables['days']
        w = decision_variables['weekends']
        if not number_of_weeks:
            number_of_weeks = 1
        for key, employee in x.items():
            employee_hours = 0
            for day in employee:
                for shift in day:
                    if shift.value() != 0:
                        employee_id, day_index, shift_index = (
                            self.get_decision_var_ids(shift))
                        print(shift, '->', shift.value(), '->',
        self.employees.list[employee_id].shifts[day_index][shift_index])
                        employee_hours += len(
```

```
578             self.employees.list[employee_id].shifts[day_index][shift_index])
579
580                 min_h = self.employees.list[key].min_hours / PERIODS_PER_HOUR
581                 max_h = self.employees.list[key].max_hours / PERIODS_PER_HOUR
582                 raw_h = employee_hours / PERIODS_PER_HOUR / number_of_weeks
583                 print(f'Employee {key} hours:', round(raw_h, 2),
584                       f'{min_h}-{max_h}')
585                 days_off_list = []
586                 for day_off_var in d[key]:
587                     if day_off_var.value() == 1:
588                         days_off_list.append(self.get_decision_var_ids(
589                             day_off_var)[1])
590                 print('Days off:', days_off_list)
591             total_excess_hours = 0
592             for day in y:
593                 for lpvariable in day:
594                     if print_daily:
595                         print(lpvariable.name, '->', lpvariable.value())
596                     total_excess_hours += lpvariable.value()
597             print('Weekends off:')
598             for key, employee in w.items():
599                 print(key, [weekend[0].value() for weekend in employee])
600
601             print('obj value:', self.problem.objective.value())
602             print('excess hours:', total_excess_hours / PERIODS_PER_HOUR)
603             print('problem status (1=opt):', status)
604
605         def get_decision_var_ids(self, variable):
606             """Return parsed variables's IDs as integers.
607
608             Args:
609                 variable: A decision variable to process.
610
611             Returns:
612                 A tuple of IDs. Length depends on the type
613                 of the decision variable being processed.
614             """
615             return [int(x) for x in variable.name[1:].split(':')]
616
617         def create_decision_variables(self):
618             """Create decision variables for the LP model.
619
620             Returns:
621                 A dictionary of different kinds of decision variables:
622                 shifts:
623                     Dictionary of lists of lists. Defines if
624                     employee i is assigned to day j:s shift k.
625                 workforce:
626                     List of lists. Keeps track of excess
627                     employees on day i:s period j.
628                 days_off:
629                     Dictionary of lists. Defines if
630                     employee i:s day j is off duty.
```

```
            day_pairs:
                Dictionary of lists. Defines if employee
                i has days j and j+1 both off duty.
            weekends:
                Dictionary of lists of tuples. Defines if
                either Fri-Sat or Sat-Sun of employee i:s
                week j is off.
                Innermost tuple consists of:
                (decision_variable, [Fri and/or Sat indices])
        """
        # 1. Create decision variables in format:
        #   x{employee_id: [day_index][shift_index]}
        #   These determine if a shift is assigned to employee.
        # 2. Create surplus variables representing days off
        #   for all employees.
        # 3. Create binary variables for every subsequent two days off.
        #   The variables will "overlap".
        x_eds = {}
        days_off = {}
        subsequent_days_off = {}
        weekends_off = {}
        recent_days_off = {}
        for _, employee in self.employees.list.items():
            x_eds[employee.id] = []
            days_off[employee.id] = []
            subsequent_days_off[employee.id] = []
            recent_days_off[employee.id] = []
            weekend_indices = []
            for day_index in range(len(employee.shifts)):
                # Add employee-shift -assignment variables.
                x_eds[employee.id].append([])
                day_shift_count = len(employee.shifts[day_index])
                for shift_index in range(day_shift_count):
                    lp_var_name = str(f'x{employee.id}:{day_index}:' +
                                      f'{shift_index}')
                    x_eds[employee.id][day_index].append(
                        LpVariable(lp_var_name, 0, 1, 'Integer'))

                # Add days off variables.
                days_off[employee.id].append(
                    LpVariable(f'd{employee.id}:{day_index}', 0, 1,
                               'Integer'))

                # Add binary variables to define if a consecutive
                # pair of days is off-duty for the employee.
                if (day_index + 1 < len(employee.shifts)):
                    subsequent_days_var = LpVariable(
                        f'p{employee.id}:{day_index}-{day_index + 1}',
                        0, 1, 'Integer')
                    subsequent_days_off[employee.id].append(
                        subsequent_days_var)
                    if (self.start_day + day_index) % 7 in (WEEKDAY_FRI,
                                                            WEEKDAY_SAT):
```

```
                        weekend_indices.append(day_index)

            # Combine same weekend's indices to pairs. If start day is
            # Saturday, start splitting to pairs from index 1. The first
            # item will then be a single item list because the first
            # weekend only has Sat-Sun pair but not a Fri-Sat pair.
            weekends_off[employee.id] = []
            split_start_idx = 1 if (
                self.start_day == WEEKDAY_SAT) else 0
            weekends_split = [weekend_indices[i:i + 2] for i in range(
                split_start_idx, len(weekend_indices), 2)]
            for pair in weekends_split:
                weekend_variable_idx = len(weekends_off[employee.id])
                weekend_variable = LpVariable(
                    f'w{employee.id}:{weekend_variable_idx}', 0, 1,
                    'Integer')
                weekends_off[employee.id].append((weekend_variable,
                                                  pair))

        # Create more decision variables in format:
        # y[day_index][period_index]
        # These represent the excess employees working
        # during every period of day.
        y_dp = []
        for i, day_length in enumerate(self.workdays_period_demand):
            y_dp.append([])
            for j in range(day_length):
                lp_var_name = f'y{i}:{j}'
                y_dp[i].append(LpVariable(lp_var_name, 0, cat='Integer'))
        return {'shifts': x_eds, 'workforce': y_dp, 'days': days_off,
                'pairs': subsequent_days_off, 'weekends': weekends_off}

    def create_objective(self, decision_variables):
        """Create objective function for LP model.

        Args:
            decision_variables:
                A dictionary of decision variables in correct format.
        """
        objective = []
        main_variables = decision_variables['shifts']
        period_surplus_variables = decision_variables['workforce']
        day_pairs = decision_variables['pairs']
        weekends_off = decision_variables['weekends']
        for _, employee in self.employees.list.items():
            shift_count = len(employee.shifts)
            for day_index in range(shift_count):
                for shift_index, shift in enumerate(
                        employee.shifts[day_index]):
                    preference_factor = 1
                    try:
                        if (
                            employee.preferences[day_index][shift_index]
```

```
                                & Preference.UNDESIRABLE
                            ):
                                # Violating a preference results in a hefty
                                # rise in the objective value. The multiplier
                                # needs to be big since preferences are
                                # relatively rare considering the total
                                # amount of terms in the objective function.
                                preference_factor = int(
                                    Preference.UNDESIRABLE)
                            except KeyError:
                                pass
                            # Add employee's dissatisfaction towards
                            # a certain shifts into the objective.
                            objective += [self.weights['preference'] * (
                                preference_factor) * (
                                main_variables[employee.id][day_index][(
                                    shift_index)])]

                        # Add one off-duty subsequent day pair to
                        # the objective each week.
                        if (day_index % 7 == 6):
                            # Default ending offset set to 1 due to range
                            # function behaviour. Set to 0 in case the
                            # current day is the last in the schedule.
                            offset = 0 if (day_index == shift_count - 1) else 1
                            indices = range(day_index - 6, day_index - offset)
                            random_index = random.choice(indices)
                            objective += [-self.weights['day_pairs_off'] * (
                                day_pairs[employee.id][random_index])]

                    # Add off-duty weekends to the objective.
                    objective += [(-self.weights['weekends_off'] * (
                        weekend_tuple[0])) for weekend_tuple in (
                        weekends_off[employee.id])]

            # Add excess workers for each shift to the
            # objective to minimise expenses.
            objective += [(self.weights['excess_workforce'] * (
                period_variable) for period_variable in day) for day in (
                period_surplus_variables)]
            self.problem += lpSum(objective)

        def create_constraints(self, decision_variables):
            """Create constraints to LP model.

            Args:
                decision_variables:
                    A dictionary of decision variables in correct format.
            """
            main_variables = decision_variables['shifts']
            period_surplus_variables = decision_variables['workforce']
            day_off_surplus_variables = decision_variables['days']
            day_pair_off_variables = decision_variables['pairs']
```

```
790             weekend_variables = decision_variables['weekends']
791             # Prepare first constraints of a kind to be
792             # added to debug messages.
793             db_msgs = []
794             first_constraint = 11 * [True]
795             # Add constraints for fulfilling all
796             # work site's time period needs.
797             # Iterate over all days in work site schedule.
798             for day_index in range(len(self.work_site_demands)):
799                 # Create vectors to hold all opening and closing shifts
800                 # from eligible employees.
801                 all_open_capable_employees_shifts = []
802                 all_close_capable_employees_shifts = []
803                 # Iterate over every period in every day.
804                 period_count = len(self.work_site_demands[day_index])
805                 for period_index in range(period_count):
806                     # Create a vector to hold all shifts
807                     # that contain said period.
808                     all_shifts_matching_period = []
809                     # Iterate over each employee.
810                     for _, employee in self.employees.list.items():
811                         # Iterate over each open shift for
812                         # the employee on the given day.
813                         shift_count = len(employee.shifts[day_index])
814                         for shift_index in range(shift_count):
815                             # If current processed shift contains current
816                             # period, add decision variable to vector.
817                             if period_index in (
818                                     employee.shifts[day_index][shift_index]):
819                                 all_shifts_matching_period.append(
820                                     main_variables[employee.id][day_index][(
821                                         shift_index)])
822 
823                                 # If current shift is also an opening shift
824                                 # and employee can open, add to vector.
825                                 # Do the equivalent for closing as well.
826                                 if (period_index == 0) and (
827                                     employee.special_properties & (
828                                         PropertyFlag.CAN_OPEN)):
829                                     all_open_capable_employees_shifts.append(
830                                         main_variables[employee.id][(
831                                             day_index)][shift_index])
832                                 last_period_index = period_count - 1
833                                 if (period_index == last_period_index) and (
834                                     employee.special_properties & (
835                                         PropertyFlag.CAN_CLOSE)):
836                                     (
837                 all_close_capable_employees_shifts.append(
838                     main_variables[employee.id][day_index][shift_index])
839                                     )
840 
841                 # Ensure all periods of the day have enough
842                 # shifts overlapping them.
```

```
                constraint = (lpSum(all_shifts_matching_period) - (
                    period_surplus_variables[day_index][(
                        period_index)])) == self.work_site_demands[(
                            day_index)][period_index]
                self.problem += constraint
                if first_constraint[0]:
                    db_msgs.append(constraint)
                    first_constraint[0] = False

            # For every first and last period per day, ensure
            # that employee who can open or close is at work.
            constraint = lpSum(all_open_capable_employees_shifts) >= 1
            self.problem += constraint
            if first_constraint[1]:
                    db_msgs.append(constraint)
                    first_constraint[1] = False
            constraint = lpSum(all_close_capable_employees_shifts) >= 1
            self.problem += constraint
            if first_constraint[2]:
                    db_msgs.append(constraint)
                    first_constraint[2] = False

        # Add multiple constraints employee by employee.
        # Iterate over employees.
        for _, employee in self.employees.list.items():
            employee_weekly_shifts = []
            streaks_start_index = MAXIMUM_CONSECUTIVE_WORKDAYS - (
                employee.current_workday_streak)
            # Iterate over every day for each employee.
            for day_index in range(len(employee.shifts)):
                # Any employee mustn't be assigned to
                # more than one shift per day.
                constraint = lpSum([x for x in main_variables[(
                    employee.id)][day_index]]) + (
                        day_off_surplus_variables[employee.id][(
                            day_index)]) == 1
                self.problem += constraint
                if first_constraint[3]:
                    db_msgs.append(constraint)
                    first_constraint[3] = False

                # Weekly working hours have lower and upper bounds.
                # Also any worker mustn't work more than the maximum
                # number of shifts defined for them. Resolve weekly shift
                # boundaries for every seven days passed.
                # Weekly shifts are (length, decision variable) -pairs.
                shift_count = len(employee.shifts[day_index])
                for shift_index in range(shift_count):
                    employee_weekly_shifts.append(
                        (len(employee.shifts[day_index][shift_index]),
                            main_variables[employee.id][day_index][(
                                shift_index)]))
                if (day_index % 7 == 6):
```

```
                    # Limit the number of periods
                    # (hours) in weekly shifts.
                    if employee.min_hours == employee.max_hours:
                        constraint = lpSum([l * x for l, x in (
                            employee_weekly_shifts)]) == (
                                employee.min_hours)
                        self.problem += constraint
                        if first_constraint[4]:
                            db_msgs.append(constraint)
                            first_constraint[4] = False
                    else:
                        self.problem += lpSum([l * x for l, x in (
                            employee_weekly_shifts)]) >= (
                                employee.min_hours)
                        constraint = lpSum([l * x for l, x in (
                            employee_weekly_shifts)]) <= (
                                employee.max_hours)
                        self.problem += constraint
                        if first_constraint[4]:
                            db_msgs.append(constraint)
                            first_constraint[4] = False

                    # Limit the number of weekly shifts.
                    constraint = lpSum([x for _, x in (
                        employee_weekly_shifts)]) <= employee.max_shifts
                    employee_weekly_shifts = []
                    self.problem += constraint
                    if first_constraint[5]:
                        db_msgs.append(constraint)
                        first_constraint[5] = False

                # For every day, ensure that the previous n days have
                # at least one day off. This prevents over n day-long
                # consecutive streaks. Some first days in schedule
                # get ignored.
                if day_index >= streaks_start_index:
                    first_streak_day = day_index - (
                        MAXIMUM_CONSECUTIVE_WORKDAYS)
                    if first_streak_day < 0:
                        first_streak_day = 0
                    # Use i+1 as the endpoint due to
                    # range function behaviour.
                    vars_list = [day_off_surplus_variables[(
                        employee.id)][i] for i in range(
                            first_streak_day, day_index + 1)]
                    constraint = lpSum(vars_list) >= 1
                    self.problem += constraint
                    if first_constraint[6]:
                        db_msgs.append(constraint)
                        first_constraint[6] = False

            # For each two-day pair, assign a binary variable
            # that takes the value of day1 * day2, i.e. works
```

```
            # as an AND logical operator.
            pair_count = len(day_pair_off_variables[employee.id])
            for pair_idx in range(pair_count):
                day1_off = day_off_surplus_variables[(
                    employee.id)][pair_idx]
                day2_off = day_off_surplus_variables[(
                    employee.id)][pair_idx + 1]
                pair_off_variable = day_pair_off_variables[(
                    employee.id)][pair_idx]
                self.problem += pair_off_variable <= day1_off
                self.problem += pair_off_variable <= day2_off
                constraint = pair_off_variable >= (
                    day1_off + day2_off - 1)
                self.problem += constraint
                if first_constraint[7]:
                    db_msgs.append(constraint)
                    first_constraint[7] = False

            # For each weekend per employee, assign a new binary
            # variable that takes the value of day1*day2, i.e.
            # create an AND logical operator. These will be later
            # combined to ensure enough weekends off for everyone.
            weekend_count = len(weekend_variables[employee.id])
            for weekend_idx in range(weekend_count):
                weekend_variable, day_indices = weekend_variables[(
                    employee.id)][weekend_idx]
                pair1_off = day_pair_off_variables[(
                    employee.id)][day_indices[0]]
                if len(day_indices) > 1:
                    pair2_off = day_pair_off_variables[(
                        employee.id)][day_indices[1]]
                    self.problem += weekend_variable >= pair1_off
                    self.problem += weekend_variable >= pair2_off
                    constraint = weekend_variable <= (
                        pair1_off + pair2_off)
                    self.problem += constraint
                    if first_constraint[8]:
                        db_msgs.append(constraint)
                        first_constraint[8] = False
                else:
                    # Weekend only has one pair because
                    # it was cut in half.
                    constraint = weekend_variable == pair1_off
                    self.problem += constraints
                    if first_constraint[8]:
                        db_msgs.append(constraint)
                        first_constraint[8] = False

            # Add constraints for ensuring the required weekends off.
            try:
                for obligatory_weekend_off_idx in (
                        employee.weekends_config['single']):
                    constraint = weekend_variables[employee.id][(
```

```
                        obligatory_weekend_off_idx)][0] == 1
                    self.problem += constraint
                    if first_constraint[9]:
                        db_msgs.append(constraint)
                        first_constraint[9] = False
                    print(f'free weekend {obligatory_weekend_off_idx}',
                          f'for {employee.id}')
            except KeyError:
                # Employee has no single weekend constraints.
                pass
            try:
                key = 'groups'
                for weekend_group_off in employee.weekends_config[key]:
                    minimum_weekends = weekend_group_off[0]
                    weekend_indices = weekend_group_off[1:]
                    constraint = lpSum(
                        [weekend_variables[employee.id][i][0] for i in (
                            weekend_indices)]) >= minimum_weekends
                    self.problem += constraint
                    if first_constraint[10]:
                        db_msgs.append(constraint)
                        first_constraint[10] = False
            except KeyError:
                # Employee has no multi weekend constraints.
                pass

        if self.debug:
            for msg in db_msgs:
                line_len = 50
                print(line_len * '-')
                print(msg)
                print(line_len * '-')
                print()


if __name__ == '__main__':
    # Testing code goes here.
    # Example process to test the program:
    # 1. Create a matrix (nested list or tuple) that holds workforce
    # demands for each day.
    # 2. Create employees.
    # 3. Create scheduler with desired parameters.
    # 4. Run scheduler.
    pass
```