# Sustav za optimiranje akademskih rasporeda temeljenih na ponavljajućim pravilima norme iCalendar

**Jurinčić, Jurica**

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

Diplomski studij informatike

**JURICA JURINČIĆ**

# SUSTAV ZA OPTIMIRANJE AKADEMSKIH RASPOREDA TEMELJENIH NA PONAVLJAJUĆIM PRAVILIMA NORME ICALENDAR

Diplomski rad

Pula, rujan 2022.

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

Diplomski studij informatike

# SUSTAV ZA OPTIMIRANJE AKADEMSKIH RASPOREDA TEMELJENIH NA PONAVLJAJUĆIM PRAVILIMA NORME ICALENDAR

Diplomski rad

**JMBAG:** 0303075593, redovan student

**Studijski smjer:** Informatika

**Predmet:** Izrada informatičkih projekata

**Znanstveno područje:** Društvene znanosti

**Znanstveno polje:** Informacijske i komunikacijske znanosti

**Znanstvena grana:** Informacijski sustavi i informatologija

**Mentor:** doc. dr. sc. Nikola Tanković

**Komentor:** doc. dr. sc. Siniša Miličić

Pula, rujan 2022.

**IZJAVA O AKADEMSKOJ ČESTITOSTI**

Ja, dolje potpisani Jurica Jurinčić, kandidat za magistra informatike ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljeni način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

Jurica Jurinčić

U Puli, 11. rujna 2022.

**IZJAVA O KORIŠTENJU AUTORSKOG DJELA**

Ja, Jurica Jurinčić dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom SUSTAV ZA OPTIMIRANJE AKADEMSKIH RASPOREDA TEMELJENIH NA PONAVLJAJUĆIM PRAVILIMA NORME ICALENDAR koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 11. rujna 2022.

Potpis

Jurica Jurinčić

# Sadržaj

# Uvod

Određivanje rasporeda izvođenja nastave čest je izazov u akademskim institucijama. Optimalno rješenje znači uskladiti različite oblike nastave, satnice profesora i studenata te pritom udovoljiti svim kriterijima koje dvorana mora posjedovati. Dodatan izazov predstavljaju razna ograničenja poput preferencija profesora, prethodna nedostupnost dvorana, te održavanje predmeta po proizvoljnim ponavljajućim pravilima. Radi se o problemu kombinatorne optimizacije jer se svodi na pronalazak optimalnog rješenja u konačnom diskretnom skupu. Znanstvenici su pokušavali pronaći definitivno računalno rješenje za probleme ovakvog tipa praktički od kad postoje računala, no s ograničenim uspjehom. Zato je problem optimizacije akademskog rasporeda i dalje klasificiran kao NP-težak [1], odnosno ne postoji polinomni deterministički algoritam koji ga optimalno rješava. Pristup grube sile (*engl. brute force*) je prespor za gotovo sve praktične primjene. To nas ostavlja s aproksimacijskim algoritmima i heuristikama koje se izvode u polinomnom vremenu, ali ne garantiraju optimalnost. Aproksimacijski algoritmi garantiraju da je rješenje blizu optimalnosti po nekom predefiniranom konceptu blizine, dok heuristike daju rješenja koja mogu biti po volji loša. To ne znači da su heuristike uvijek loše, naprotiv dobra heuristika može biti izrazito brza i zadovoljavajuća za praktične potrebe. U istu ruku, aproksimacijski algoritmi se mogu pokazati nepraktičnim zbog težine svođenja problema na rigorozniji model koji oni zahtijevaju. U literaturi za rješavanje problema akademskog rasporeda dominiraju heuristike kao što su genetski algoritam, lokalno traženje, i sl. Problem akademskog rasporeda nije samo teško riješiti, već ga je teško i univerzalno definirati. Gotovo svako sveučilište ima svoja proizvoljna ograničenja koja želi da budu uklopljena u problem. Štoviše, česta je pojava da autori imaju proizvoljne definicije problema bazirane na svojim sveučilištima gdje onda rješavaju taj problem u nadi da će se rješenje generalizirati na širi kontekst.

Ovaj rad se sastoji od pet poglavlja. U prvom poglavlju su predstavljene tipične sličnosti i različitosti u definicijama problema akademskog rasporeda koje se nalaze u literaturi. Predstavljena je ukratko i definicija problema koja se rješava u ovom radu. U drugom poglavlju su predstavljeni tipični načini rješavanja problema akademskog rasporeda

(bojanje grafa, genetski algoritam, i slično). U trećem poglavlju su teorijski pokriveni algoritmi iz obitelji lokalnog traženja koji su implementirani na problemu rasporeda u četvrtom poglavlju. Specifično, to su sljedeći algoritmi: jednostavno lokalno traženje, simulirano kaljenje, pretraživanje promjenjivim susjedstvom, pohlepno nasumično adaptivno pretraživanje, i iterirano lokalno traženje. U četvrtom poglavlju je predstavljen sustav za rješavanje problema akademskog rasporeda gdje se svaka grupa predmeta može održavati po proizvoljnom ponavljajućem pravilu. Objašnjene su sljedeće stvari: definicija problema (rigoroznije), alat RRULE za ponavljajuća pravila, struktura ulaznih podataka, način praćenja kršenja ograničenja, i pet optimizacijskih algoritama. U petom poglavlju su uspoređene performanse prethodno obrađenih optimizacijskih algoritama na skupovima podataka koji su kategorizirani po težini rješavanja.

# 1. Definicija problema

Teško je univerzalno definirati problem akademskog rasporeda jer većina akademskih institucija ima proizvoljne zahtjeve. Autori koji se bave ovim problemom ga najčešće definiraju u skladu s realnim potrebama svojih sveučilišta ili čak apstraktno [2,4,6,10,18]. To dovodi do toga da im se definicije u nekoj mjeri razlikuju, ali gotovo uvijek postoje sličnosti između njih. U nastavku će biti prikazane tipične sličnosti i različitosti definicija problema u literaturi, i na kraju kratka definicija problema obrađivana u ovom radu.

## 1.1. Sličnosti u definicijama

Najčešći elementi akademskog rasporeda su: predmeti, dvorane, profesori, studenti, i vrijeme. Predmete koje predaju neki profesori i slušaju neki studenti potrebno je smjestiti u neku dvoranu na neko vrijeme. Cilj je napraviti to tako da se prekrši što manje ograničenja.

Primjeri najčešćih ograničenja:

- „Dvorana ne smije imati dva ili više predmeta u isto vrijeme.“ [2,3,4,6]
- „Profesor ne smije predavati dva ili više predmeta u isto vrijeme.“ [2,4,6]
- „Student ne smije slušati dva ili više predmeta u isto vrijeme.“ [3,4,6]
- „Kapacitet dvorane mora biti veći ili jednak broju studenata u dvorani.“ [2,3,6]

Ograničenja se dijele na čvrsta i meka (*engl. hard and soft*). Čvrsta ograničenja ne smiju biti prekršena, u protivnom rješenje postaje nevažeće. Meka ograničenja smiju biti prekršena, ali je cilj minimizirati to. Odluka o tome koja ograničenja će biti čvrsta, a koja meka u potpunosti ovise o autoru.

Problem rasporeda se klasificira kao problem kombinatorne optimizacije. Različit poredak predmeta na dvorane i vremenske položaje će vrlo vjerojatno dati različit broj prekršenih ograničenja. Cilj je pronaći poredak predmeta koji krši što manje ograničenja.

## 1.2. Različitosti u definicijama

Različitosti u definicijama u literaturi se često pojavljuju zbog specifičnih zahtjeva akademskih institucija za koje autori rješavaju raspored.

Primjeri rjeđih ograničenja:

- „Zadovoljiti instruktorovo opterećenje na temelju *Ins* skupa.“ [2]
- „Kurs mora predavati instruktor koji ga je prije predavao.“ [2]
- „Na dan student mora slušati samo jedan kurs.“ [3]
- „Predavač može predavati najviše tri sesije u jednom danu.“ [7]
- „Minimizirati promjenu dvorana, osobito za studente nižih godina.“ [4]
- „Stopa popunjenosti svakog događaja ili ne smije biti manja od 75% ili mora biti jednaka 0%.“ [6]

Pored toga što su ograničenja različita, i terminologija je različita. Neki profesora zovu „instruktor“, ili „predavač“, neki predmet „kurs“, ili „događaj“, i sl. Iz ovoga se može zaključiti da autori često rješavaju tip problema koji se u određenoj mjeri razlikuje od tipa problema drugih autora i pritom koriste vlastitu terminologiju.

## 1.3. Definicija problema u ovom radu

Definicija problema korištena u ovom radu proizlazi iz potreba rasporeda Sveučilišta Jurja Dobrile u Puli. Analizirani su prijašnji načini izrade i identificirani zahtjevi. Jedan od temeljnih zahtjeva bio je da se svaka grupa predmeta može održavati po proizvoljnom ponavljajućem pravilu. Prijašnji način dozvoljavao je samo jedno ponavljajuće pravilo oblika „svaki tjedan isto“. Proizvoljno ponavljajuće pravilo podrazumijeva da korisnik može sam definirati tip ponavljanja. Neki od primjera su: „svaki drugi tjedan“, „samo osam dana počevši od 4. 10. 2021.“, „svaki zadnji četvrtak i petak u mjesecu“, i sl.

U ovom poglavlju nisu detaljno objašnjeni tehnički detalji definicije problema koja se obrađuje u ovom radu, ta definicija se nalazi u četvrtom poglavlju. Ovdje su samo iznesena ograničenja da čitatelj dobije dublji uvid u problem koji se rješava.

Pošto se u ograničenjima koristi termin „rasp“, bit će ukratko objašnjen. Rasp je jedinica predmeta koja ide u raspored. Formalnije, to je jedinica predmeta koja ima jedinstvenu trojku (identifikator predmeta, oblik izvođenja predmeta, oznaka grupe). Ako je predmet „Kriptografija“, onda raspovi mogu biti „Kriptografija Predavanje Grupa 1“, „Kriptografija Vježbe Grupa 1“, „Kriptografija Vježbe Grupa 2“ i sl.

**Ograničenja:**

1. Svaki rasp se mora održavati po ponavljajućem vremenskom pravilu (pr. svaki drugi tjedan, ili samo šest dana počevši od nekog datuma, i sl.)
2. Dvorana ne smije imati dva ili više raspa u isto vrijeme
3. Profesor ne smije predavati dva ili više raspa u isto vrijeme
4. Semestar ne smije imati dva ili više obaveznih raspova u isto vrijeme, osim ako nije riječ o grupi istog tipa (primjerice, na „Mat_P1“ i Mat_P2“ se ograničenje ne odnosi)
5. Semestar smije imati izborne raspove u isto vrijeme, osim ako nije riječ o raspovima istog predmeta, ali različitog tipa (primjerice, „Erp_P1“ i „Erp_V1“ se ne mogu održavati u isto vrijeme. Oba raspa imaju isti predmet „Erp“ i različit tip, prvi P=predavanje, drugi V=vježbe)
6. Kapacitet dvorane ne smije biti manji od broja studenata raspa
7. Raspovi koji zahtijevaju računala moraju biti postavljeni u računalnu dvoranu
8. Ako je rasp fiksiran na neku dvoranu, u tu dvoranu ga treba i postaviti
9. Ako je rasp fiksiran na neki dan, na taj dan ga treba i postaviti
10. Ako je rasp fiksiran na neki sat, na taj sat ga treba i postaviti

Izazov kod proizvoljnih ponavljajućih pravila je u praćenju kolizija vremena u dvoranama, profesorima, i semestrima. Primjerice, ako se jedan rasp održava samo pet dana u sredini semestra, a drugi rasp svaki tjedan od početka do kraja, kolizija između njih se može dogoditi tek negdje u sredini semestra, ali više o tome u četvrtom poglavlju.

# 2. Načini rješavanja

Pristup grube sile za problem akademskog rasporeda nije izvodljiv u realnom vremenu. Takav pristup enumerira sve moguće kombinacije predmeta i vremenskih položaja i svaku kombinaciju ubacuje u funkciju cilja. Ona kombinacija koja proizvede najbolju vrijednost predstavlja optimalno rješenje. Problem je u tome što je broj kombinacija za gotovo sve praktične situacije prevelik da bi algoritam terminirao u realnom vremenu. Pored toga, ne postoji polinomni deterministički algoritam koji rješava problem [1]. Zato se koriste heuristički algoritmi koji se na pametan način kreću kroz prostor traženja u nadi da će pronaći što bolje rješenje. Oni ne mogu garantirati optimalnost jer ne pokrivaju sve kombinacije, ali mogu dati zadovoljavajuća rješenja za praktične potrebe. U ovom poglavlju su objašnjeni neki od heurističkih algoritama koji se u literaturi koriste za rješavanje problema akademskog rasporeda.

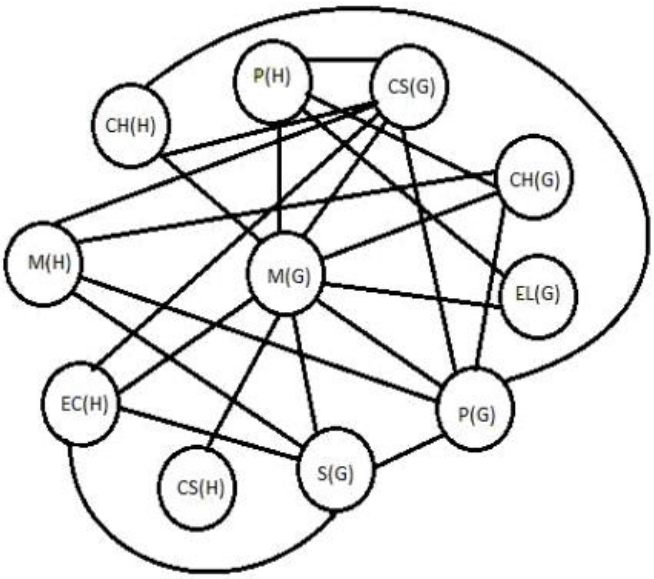
## 2.1. Bojanje grafa

**Teorija bojanja grafa** bavi se problemima u kojima je potrebno preko jednostavnog skupa pravila za svaki par objekata utvrditi jesu li u istoj klasi ili ne [8]. Objekti formiraju skup vrhova $V(G)$ grafa $G$ gdje su dva vrha povezana bridom ako im nije dozvoljeno biti u istoj klasi. Da bi se razlikovale klase koristi se skup boja $C$ s kojima se oboje vrhovi tako da svaki par vrhova povezan bridom nema istu boju. Najmanji broj boja s kojima se tako mogu obojiti vrhovi zove se kromatski broj grafa. Problem pronalaska kromatskog broja grafa je NP-težak [9]. Postoje pohlepni i heuristički algoritmi bojanja grafa koji se izvršavaju u polinomnom vremenu, ali ne garantiraju optimalnost.

Problem akademskog rasporeda je moguće riješiti bojanjem grafa [10,11]. Ideja iza ove reprezentacije je da predmeti budu vrhovi, bridovi ograničenja (tipično čvrsta ograničenja), a boje vremenski položaji. Dva susjedna vrha s istom bojom stvaraju koliziju. Ukoliko bi uspješno obojili graf, dobili bi rješenje. Optimalno rješenje bi bilo ono koje koristi najmanji broj boja.

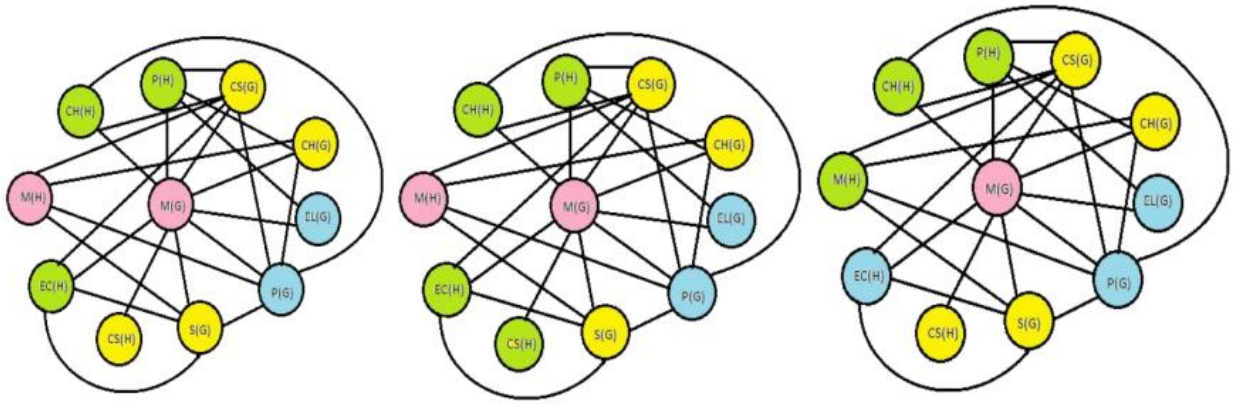
Ganguli i Roy [10] se u svom radu bave rješavanjem problema rasporeda koristeći algoritam bojanja grafa gdje je cilj zadovoljiti čvrsta ograničenja i minimizirati broj vremenskih položaja koji se koriste da bi se održali predmeti. Optimalno rješenje dobivaju pronalaskom najmanjeg broja boja za dani graf.

Predstavljaju problem gdje postoje predmeti koji ne smiju biti na istim vremenskim položajima ako imaju zajedničkog studenta. Rješenje je da se stvori graf gdje su vrhovi predmeti između kojih postoji brid ako postoji zajednički student.




*Slika 1 - Graf zajedničkih studenata predmeta [10]*

Stvoreni graf se oboja s najmanjim brojem boja. Slika 2 prikazuje neke od mogućih načina bojanja grafa sa slike 1. Svaki vrh (predmet) obojen istom bojom moguće je smjestiti na isti vremenski položaj jer nemaju zajedničkih studenata.




*Slika 2 - Obojeni graf s najmanjim brojem boja [10]*

Mandal [11] je u svom radu predstavio interaktivni sustav za rješavanje problema rasporeda koji koristi bojanje grafa i lokalno traženje. Bojanje grafa koristi za konstrukciju inicijalnih rješenja, a lokalno traženje za optimizaciju. Specifično, bojanje grafa za zadovoljavanje čvrstih ograničenja, a lokalno traženje za zadovoljavanje mekih. Za graf koristi heuristiku stupnja zasićenja (*engl. saturation degree graph heuristic*), a za lokalno traženje postoje tri izbora: algoritam velikog potopa (*engl. great deluge algorithm*), simulirano kaljenje (*engl. simulated annealing*) i penjanje uzbrdo s kasnim prihvaćanjem (*engl. late acceptance hill climbing*). Korisnik može birati između algoritama i podesiti njihove parametre.

## 2.2. Cjelobrojno linearno programiranje

**Problem linearnog programiranja** (*engl. linear programming problem,* skraćeno LP) [12] je klasa problema matematičkog programiranja, optimizacije ograničenja, u kojoj je potrebno pronaći skup vrijednosti za kontinuirane varijable $(x_1, x_2, \dots, x_n)$ tako da se maksimizira ili minimizira objektivna funkcija $z$ i pritom zadovolji skup linearnih ograničenja. Matematička formula je prikazana ispod.

$$(LP) \quad Maksimiziraj\ z = \sum_j c_j x_j$$

$$Uz\ uvjete \sum_j a_{ij} x_j \leq b_i \quad (i = 1,2,\dots,m)$$

$$x_j \geq 0 \qquad (j = 1,2,\dots,n)$$

**Problem cjelobrojnog (linearnog) programiranja** (*engl. integer linear programming problem,* skraćeno IP) [12] je problem linearnog programiranja gdje je barem jedna varijabla ograničena na cjelobrojne vrijednosti. Ponekad se koristi i izraz „problem mješovitog cjelobrojnog programiranja“ (*engl. mixed integer programming problem,* skraćeno MIP) za isti pojam koji je nastao kao kontrast „problemu čistog cjelobrojnog programiranja“ (*engl. pure integer programming problem,* skraćeno *pure* IP) gdje su sve varijable ograničene na cjelobrojne vrijednosti.

**Problem binarnog cjelobrojnog (linearnog) programiranja** (*engl. binary integer linear programming problem,* skraćeno BIP) [12] je problem cjelobrojnog programiranja gdje su sve varijable ograničene na vrijednosti 0 ili 1.

Problem linearnog programiranja je u klasi složenosti P [13]. Problem cjelobrojnog programiranja je NP-težak [14]. Zato se često za rješavanje većih problema cjelobrojnog programiranja koriste heuristike.

Daskalaki, Birbas, i Housos [4] u svom radu predstavljaju model za problem rasporeda koji se temelji na binarnom cjelobrojnom programiranju i koji obuhvaća veliki broj tipičnih ograničenja. Podržava vremenski uzastopne predmete, kao i predmete koji se održavaju više od jedan put. Binarne varijable su definirane tako da su strukturni elementi: predmeti, studenti, profesori, dani i periodi sačuvani na svojstven (*engl. distinct*) način. To dovodi do velikog broja mogućih varijabli, ali autori napominju da se uvođenjem smislenih podskupova osnovnih skupova (dana, perioda, grupa studenata, profesora, predmeta, i dvorana) broj varijabli smanjuje na upravljivu razinu. Na kraju predstavljaju rezultate gdje im model rješava tri problema: prvi u 2.5 minute, drugi u 18.5 minuta, i treći u 95 minuta.

Perera i Lanel [5] u svom radu koriste bojanje grafa i cjelobrojno programiranje da bi riješili problem rasporeda. Bojanje grafa koriste kako bi odredili grupe predmeta bez zajedničkih studenata koje posljedično mogu biti smješteni na isto vrijeme. Binarno cjelobrojno programiranje koriste kako bi svakoj grupi odredili vremenski položaj. Na kraju predstavljaju rezultate gdje im model rješava raspored za tri godine studija.

## 2.3. Problem zadovoljenja ograničenja

**Problem zadovoljenja ograničenja** (*engl. constraint satisfaction problem,* skraćeno CSP) [15] se definira kao trojka $(Z, D, C)$ gdje je $Z =$ konačan skup varijabli $\{x_1, x_2, \dots, x_n\}$, $D =$ funkcija koja mapira varijable iz $Z$ na skup vrijednosti iz domene, i $C =$ konačan skup ograničenja na proizvoljnom podskupu od $Z$.

Rješenje se dobije dodjeljivanjem vrijednosti iz domene svim varijablama tako da sva ograničenja budu zadovoljena [15]. Ako postoji rješenje, kaže se da je problem zadovoljiv.

Problem zadovoljenja ograničenja je različit od linearnog programiranja jer ne zahtijeva optimizaciju funkcije cilja. Jedino što želi je dodijeliti vrijednosti varijablama tako da ne prekrši ograničenja.

Problem zadovoljenja ograničenja je NP-težak, ali svaki problem je različit i ponekad je moguće razviti specijalizirane tehnike za iskorištavanje njihovih individualnih svojstva [16].

Metode generiranja rješenja za probleme zadovoljenja ograničenja dijele se na tri klase [17]:

1. Varijante pretraživanja unazad (*engl. backtracking*)
2. Algoritmi propagiranja ograničenja (*engl. constraint propagation algorithms*)
3. Algoritmi vođeni strukturom (*engl. structure-driven algorithms*)

El-Sakka [18] u svom radu predstavlja model zadovoljenja ograničenja koji uz pomoć programiranja logike ograničenja (*engl. constraint logic programming,* skraćeno CLP) i optimizacijskog programskog jezika (*engl. optimization programming* language, skraćeno OPL) rješava problem rasporeda. OPL sadrži definiciju za model podataka, deklaraciju varijabli odluke, i iskaz ograničenja i cilja. Model podataka sadrži varijable domene od kojih su neke: *NumberOfDaysPerPeriod* (broj radnih dana u tjednu), *ClassRequirementSet* (edukacijski program), *Room* (skup dvorana), i *MorningClass* (skup jutarnjih predmeta). Postoji i skup ograničenja od C1 do C22 od kojih su neka: „Profesor je potreban samo jednom u svakoj vremenskoj točki“, „Dvorana je potrebna samo jednom u svakoj vremenskoj točki“, „Jutarnji predmet završava ujutro (8:00 do 14:00)“, i „Predmet završava nakon što započne“. Rješenje dobivaju pokretanjem CP rješavača podešenog na heuristički algoritam koji koristi „prvo u dubinu“ pretragu.

Ďuriš i Taylor [19] u svom radu verificiraju validnost i funkcionalnost metoda zadovoljenja ograničenja za rješavanje problema rasporeda. Definiraju problem i rješavaju ga algoritamski uz pomoć pretrage na bazi stabla (*engl. tree-based*). Osim što verificiraju da je moguće pouzdano riješiti problem rasporeda uz pomoć metoda zadovoljenja ograničenja, također i verificiraju vremensku složenost koristeći svoj program.

## 2.4. Genetski algoritam

**Genetski algoritam** je optimizacijski algoritam inspiriran prirodnom selekcijom. Bazira se na populaciji i konceptu „preživljavanja najsposobnijih“. Nove populacije se proizvode iterativno korištenjem genetskih operatora nad jedinkama populacije. Ključni elementi su: reprezentacija kromosoma, selekcija, križanje, mutacija, i funkcija cilja [20].

Procedura genetskog algoritma je sljedeća [20]:

1. Inicijaliziraj populaciju $Y$ od $n$ kromosoma.
2. Izračunaj vrijednost funkcije cilja svakog kromosoma iz $Y$.
3. Selektiraj dva kromosoma $C1$ i $C2$ iz $Y$ prema njihovoj vrijednosti funkcije cilja.

4. Apliciraj genetski operator križanja s vjerojatnošću križanja $C_p$ na $C1$ i $C2$ i time stvori potomka $O$.
5. Apliciraj genetski operator uniformne mutacije s vjerojatnošću mutacije $M_p$ na potomka $O$ i time stvori novog potomka $O'$.
6. Dodaj $O'$ u novu populaciju.
7. Ponovi operacije selekcije, križanja, i mutacije na tekućoj populaciji dok nova populacija nije potpuna.

**Inicijalizacija** [21]: Inicijalna populacija se tradicionalno generira slučajno, pokrivajući cijeli raspon mogućih rješenja. Međutim, da bi se poboljšala kvaliteta za specifične probleme, raspon se može fiksirati na područja gdje je veća vjerojatnost pronalaska optimalnih rješenja.

**Selekcija** [21]: Tijekom svake iteracije, dio postojeće populacije se selektira da stvori novu generaciju. Individualna rješenja su selektirana preko *fitness* baziranog procesa, gdje ona koja su više *fit* tipično budu prije izabrana. Neke metode ocjenjuju sva rješenja, a neka zbog brzine, samo slučajni uzorak populacije. Većina metoda su stohastičke i dizajnirane tako da selektiraju i malu proporciju manje *fit* rješenja. To pridonosi raznolikosti populacije koja smanjuje vjerojatnost preranih konvergencija s lošim rješenjima. Neke od popularnih metoda selekcije su: selekcija kola ruleta (*engl. roulette wheel selection*), turnir selekcija (*engl. tournament selection*), i slično.

**Genetski operatori** [21]: Nakon selekcije, slijedi stvaranje rješenja nove populacije uz pomoć genetskih operatora: križanja i mutacije. Selekcija odabire par „roditelj“ rješenja koji križanjem i mutacijom stvaraju novo „dijete“ rješenje. Novo rješenje tipično ima puno istih karakteristika kao i „roditelji“. Novi roditelji su odabrani za svako dijete i proces se ponavlja dok nova populacija ne dosegne dovoljnu veličinu.

**Terminiranje** [21]: Proces selekcije i primjenjivanja genetskih operatora se ponavlja sve dok se ne postigne uvjet zaustavljanja. Neki od popularnih uvjeta zaustavljanja su:

- Rješenje zadovoljava minimalne zahtjeve

- Fiksirani broj generacija je dostignut
- Alocirano vrijeme je iskorišteno
- Algoritam više ne proizvodi bolja rješenja
- Ručna intervencija

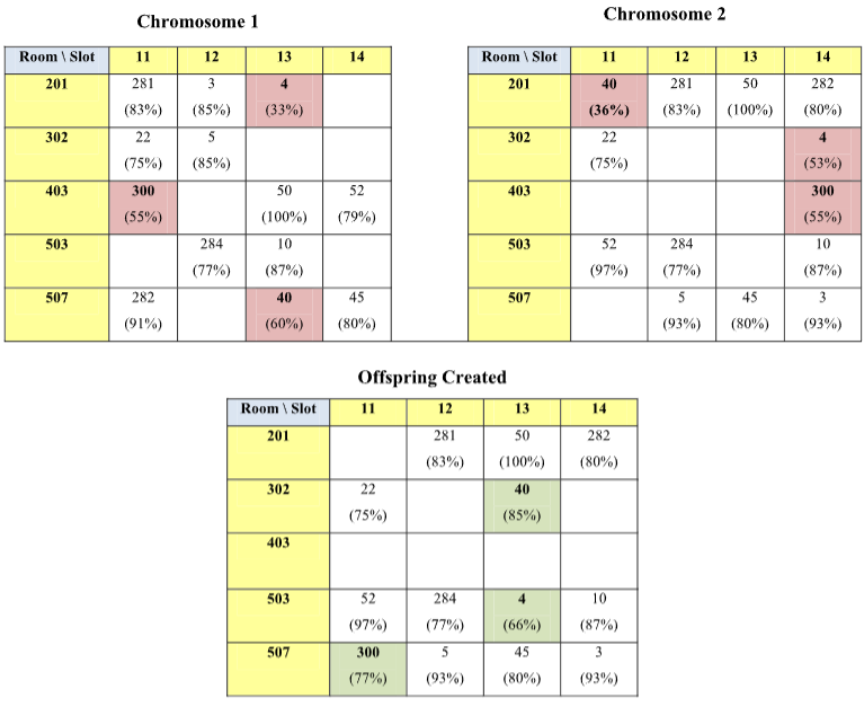
Genetski algoritam je po broju znanstvenih radova jedan od najpopularnijih algoritama za rješavanje problema akademskog rasporeda.

Aljarrah, Alsawalqah i Hamdan [2] u svom radu koriste genetski algoritam kako bi riješili problem rasporeda s čvrstim i mekim ograničenjima. Proces rješavanja im je podijeljen u dvije faze. Prva faza uključuje prikupljanje podataka i ograničenja od administratora, generaciju kromosoma (rješenja) i inicijalizaciju populacije. Druga faza uključuje apliciranje genetskih operatora: križanja i mutacije. Na svaki kromosom iz populacije se aplicira funkcija cilja koja mjeri kršenje čvrstih i mekih ograničenja. Neka čvrsta ograničenja su: „Svi predmeti moraju biti raspoređeni“, „Sesije istog predmeta ne smiju biti u isto vrijeme“, a neka meka ograničenja: „Kapacitet dvorane mora biti u skladu s očekivanim brojem studenata“, „Profesor mora predavati predmet koji želi predavati“, i sl. Riješili su problem rasporeda za odjel na svom sveučilištu nakon 32000 iteracija algoritma.

Abdelhalim i El Khayat [6] u svom radu koriste genetski algoritam da bi riješili problem rasporeda. Iznose tezu da su se mnogi prijašnji znanstveni radovi previše fokusirali na računalno vrijeme svojih algoritama (nerijetko na skupovima podataka s malim brojem ograničenja), a premalo na zadovoljavanje realnih ograničenja sveučilišta u praksi. Dalje iznose da računalno vrijeme ne bi trebalo biti fokus ovog tipa problema jer čak i da nekom algoritmu treba relativno više vremena, to bi bilo u redu ukoliko bi mogao: adaptirati se pravom problemu i njegovim ograničenjima, adresirati veću instancu problema, i uzimati manje vremena od ručnog procesa izrade rasporeda.

Predstavljaju pristup problemu rasporeda uz pomoć genetskog algoritma koji je u stanju boriti se s velikim i realnim rasporedima. Ograničenja su im podijeljena na čvrsta i meka.

Neka čvrsta ograničenja su: „Profesori se postavljaju na vremenske položaje koje preferiraju u skladu s mjerom prioriteta“, „Broj studenata je kompatibilan s kapacitetom dvorane“, a neka meka ograničenja: „Ne smije biti više od dva predavanja po danu za svakog profesora“, „Ne smije biti više od dva predavanja po danu za svaku grupu“, i sl. Kromosom reprezentiraju kao generičko dvodimenzionalno polje gdje su dvorane jedna dimenzija, a vremenski položaji druga dimenzija.




*Slika 3 - Operacija križanja na dva kromosoma [6]*

Nad kromosomima se vrše genetski operatori, primjer križanja se vidi na slici 3. Još jedna posebnost ovog rada je ta što se fokusiraju na iskorištenost (*engl. utilization*) prostora gdje traže balans između niske i visoke iskorištenosti.

## 2.5. Lokalno traženje

**Lokalno traženje** [22] je heuristički algoritam koji polazi od inicijalnog rješenja i kreće se iterativno kroz druga rješenja koja su „blizu“ tekućem rješenju. Funkcija susjedstva

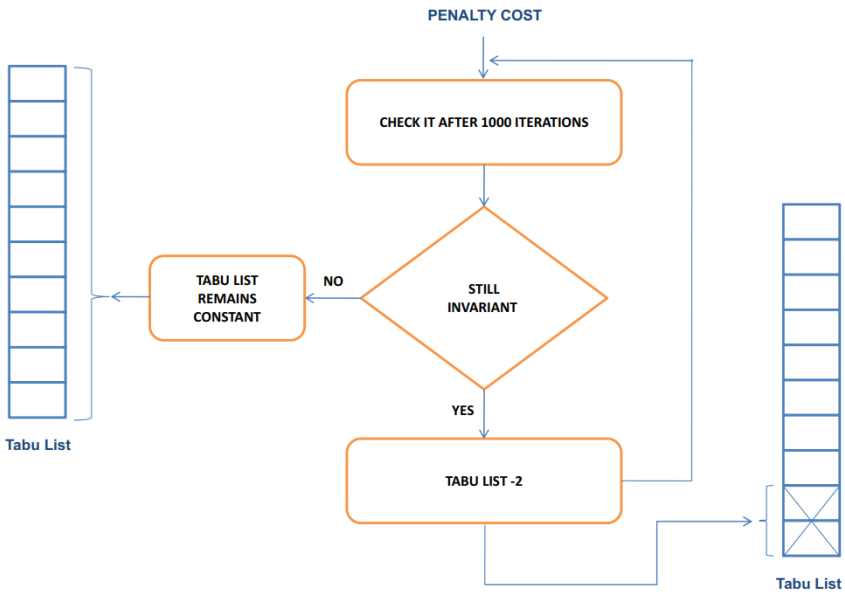
definira koja rješenja su blizu nekom rješenju. Tipične definicije funkcije susjedstva su lokalna preuređenja: razmjenjivanje, pomicanje, i zamjenjivanje objekata.

Najjednostavnija varijanta lokalnog traženja je ona gdje se iterativno odabire susjed s boljom vrijednosti funkcije cilja od tekućeg rješenja dok takav susjed postoji [22]. Ta varijanta se zove penjanje uzbrdo i jednostavno pronalazi lokalni optimum u skladu s definiranim susjedstvom i funkcijom cilja.

Da bi se izbjegao lokalni optimum (koji nije ujedno i globalni optimum) apliciraju se raznorazne strategije i modifikacije jednostavnog lokalnog traženja. Neke od njih su: simulirano kaljenje, tabu pretraživanje, iterirano lokalno traženje, i slično [38]. Lokalno traženje uživa široku primjenu u praksi na problemima kao što su [23]: trgovački putnik, strojno raspoređivanje, Steinerov problem stabla u grafovima, bojanje grafa, i slično.

Schuurmans i Southey [24] postavljaju hipotezu da algoritmi lokalnog traženja funkcioniraju dobro, ne zato što razumiju prostor traženja već zato što se brzo kreću prema obećavajućim regijama te pretražuju prostor traženja na niskim dubinama na brz, širok, i sustavan način.

Awad, Al-kubaisi, i Mahmood [3] u svom radu predstavljaju adaptivno tabu pretraživanje za rješavanje problema rasporeda. Imaju čvrsta i meka ograničenja. Neka čvrsta ograničenja su: „Studentu treba biti dodijeljen samo jedan predmet u isto vrijeme“, „Ne može biti više studenata na predmetu nego što je kapacitet dvorane“, a neka meka ograničenja: „Barem dva uzastopna predmeta trebaju biti dodijeljena studentu“, i „Na neki dan, student mora imati samo jedan predmet“. Korišteni skupovi podataka su podijeljeni na pet malih, pet srednjih, i jedan veliki. Algoritam se sastoji od dva dijela: konstrukcija i poboljšanje. Konstrukciju provode uz pomoć algoritma „najmanjeg stupnja zasićenja“ (*engl. least saturation degree*) s kojim dobivaju inicijalno rješenje koje ne krši čvrsta ograničenja. Algoritam poboljšanja, tzv. „adaptivno tabu pretraživanje“ djeluje na inicijalnom rješenju u cilju da smanji kršenja mekih ograničenja. Rad adaptivnog tabu pretraživanja se vidi na slici 4.




*Slika 4 – Adaptivno tabu pretraživanje [3]*

Bendi i Junaidi [25] predstavljaju algoritam simuliranog kaljenja koji rješava problem rasporeda. Inicijalno rješenje generiraju uz pomoć algoritma bojanja grafa koji je popularan odabir za zadovoljavanje čvrstih ograničenja. Na to inicijalno rješenje apliciraju algoritam simuliranog kaljenja koji se sastoji od tipičnih dijelova: temperature, funkcije smanjivanja temperature, vjerojatnosne funkcije prihvaćanja, i sl. Algoritam su testirali na vlastitim podacima prikupljenima iz parnih semestara 2017/2018. godine i rezultati im pokazuju da veća temperatura i broj iteracija uzrokuju manji broj kolizija.

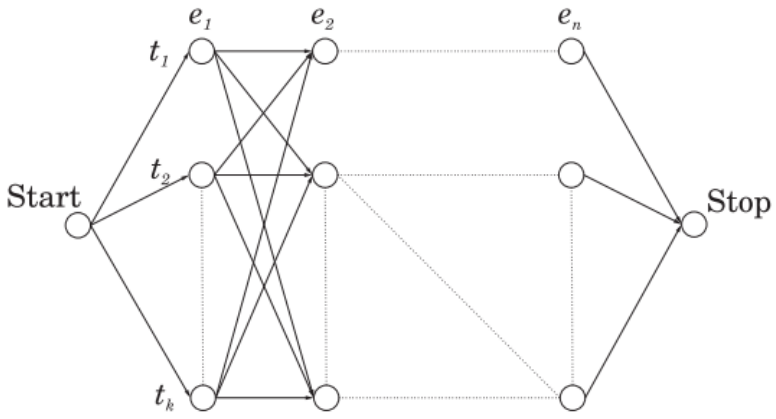
## 2.6. Algoritmi bazirani na roju

Algoritmi bazirani na roju (*engl. swarm based algorithms*) [26] su inspirirani prirodom. Sastoje se od velikog broja jedinki koje individualno imaju vrlo limitiranu računalnu moć, ali surađujući stvaraju moćan sustav procesiranja. Njihovo kolektivno inteligentno ponašanje proizlazi iz interakcije velikog broja neinteligentnih jedinki.

**Optimizacija kolonijom mrava** (*engl. ant colony optimization*) [26] je algoritamski okvir inspiriran prirodnom sposobnošću mrava da pronađu najkraći put do svojih ciljeva. Funkcionira tako da pošalje umjetne mrave koji istražuju puteve grafa čiji vrhovi tipično

reprezentiraju komponente rješenja. Krećući se od jednog vrha do drugog, mravi konstruiraju rješenje za problem. Bridovi grafa imaju numeričku vrijednost koja reprezentira koncentraciju feromona. Feromon je inače izlučevina kemijskih faktora u prirodi koja izaziva društveni odgovor kod članova iste vrste. Bridovi mogu također sadržavati dodatne heurističke informacije koje usmjeruju pretragu. Bridovi koji su dio puteva s većom kvalitetom rješenja dobivaju više feromona. Na ovaj način pretraga se usmjerava na regije s obećavajućim rješenjima. Optimizacija kolonijom mrava je poprilično generalan model, te ga je potrebno prilagoditi za specifičan problem koji se rješava.

Socha, Knowles i Sampels [27] u svom radu koriste *max-min* sustav mrava da bi riješili problem rasporeda. Fokusirali su se na simplifikaciju rasporeda koja koristi tri čvrsta i tri meka ograničenja. Čvrsta ograničenja su im: „Ni jedan student ne prisustvuje više od jednom događaju u isto vrijeme“, „Dvorana je dovoljno velika da primi sve studente i zadovoljava sve značajke događaja“, i „Samo jedan događaj je dozvoljen u dvorani na istom vremenskom položaju“. Meka ograničenja su im: „Student ne smije imati nastavu na zadnji sat u danu“, „Student ne smije imati više od dva predmeta za redom“, i „Student ne smije imati točno jedan predmet u danu“. Optimizacija kolonijom mrava počinje izradom konstrukcijskog grafa (*engl. construction graph*) tako da put kroz graf (od početka do kraja) reprezentira raspored.




*Slika 5 – Konstrukcijski graf [27]*

Na slici 5 je vidljiv konstrukcijski graf gdje se mravi kreću po događajima (*engl. event*) i odabiru vremenski položaj (*engl. time slot*) za svaki događaj. Odabir vremenskog položaja za događaj ovisi o matrici feromona i dodatnim heurističkim informacijama.

*Max-min* sustav mrava kreće s kolonijom od $m$ mrava i u svakoj iteraciji, svaki mrav konstruira kompletno rješenje tako da svaki događaj postavi na vremenski položaj. Na kraju iterativne konstrukcije, rješenje se pretvara u kandidat rješenje s algoritmom podudaranja (*engl. matching algorithm*). Nakon što je svih $m$ mrava generiralo svoje kandidat rješenje, jedno se odabire na temelju vrijednosti funkcije cilja. Odabrano rješenje se dalje poboljšava uz pomoć lokalnog traženja. Na kraju se ažurira dosad najbolje rješenje i matrica feromona.

**Optimizacija roja čestica** (*engl. particle swarm optimization*) je jedan od popularnijih algoritama baziranih na roju za kontinuirane probleme. Njegov razvoj potječe iz simulacija roja agenata pretrage gdje su rezultirajući modeli pokazali sličnosti s modelima iz područja fizike čestica (*engl. particle physics*) iz koje su posuđene ideje i notacije [26]. Algoritam rješava problem tako što prvo izgradi populaciju kandidat rješenja (čestica) i nakon toga pomiče te čestice kroz prostor traženja prema matematičkoj formuli ovisno o njihovoj poziciji i brzini [28]. Na kretanje čestica utječe njihova najbolja lokalna pozicija, ali su i usmjereni prema najboljim poznatim pozicijama u prostoru traženja koje se ažuriraju u skladu sa sve boljim pozicijama koje pronalaze druge čestice. Očekuje se da će se tako roj kretati prema najboljem rješenju.

Mudjihartono [7] u svom radu predstavlja algoritam optimizacije roja čestica za problem rasporeda. Ima samo meka ograničenja koja dijeli na dvije kategorije. Prva kategorija su negativna ograničenja čija kršenja degradiraju vrijednost funkcije cilja, a druga pozitivna ograničenja čija ispunjenja poboljšavaju vrijednost funkcije cilja. Neka od ograničenja su: „Profesor može predavati najviše tri sesije u danu“, „Profesor mora predavati barem tri dana u tjednu“, „Preferencije profesora se moraju poštivati“. U algoritmu optimizacije roja čestica, svaka čestica predstavlja raspored. Čestice se prema matematičkoj formuli kreću po prostoru traženja dok ne zadovolje terminirajući uvjet. Testirao je algoritam na generiranim rasporedima, nad kojima je 10 puta pokrenuo klasični i modificirani

algoritam optimizacije roja čestica (ispod 20 čestica i 100 iteracija). Zaključio je da je moguće koristiti predstavljeni algoritam za rješavanje problema rasporeda, te da je modificirani algoritam ostvario bolje rezultate od klasičnog.

## 2.7. Drugi algoritmi

Pored navedenih, postoje i brojni drugi algoritmi koji su korišteni za rješavanje problema rasporeda. Neki od njih su navedeni ispod:

**FET rješenje** [29]: FET je aplikacija otvorenog koda pod GNU/GPL licencom koja podržava automatsku generaciju rasporeda za akademske institucije kao što su škola i sveučilište. Koristi heuristički algoritam baziran na rekurzivnom zamjenjivanju aktivnosti da bi riješio teške instance rasporeda. Uzima u obzir razrede, predmete, studente i može prikazati rezultate u XHTML formi.

**Rudarenje podataka** [30]: Autori prikazuju kako je moguće koristiti rudarenje podataka za problem rasporeda. Rudarenje podataka je proces otkrivanja raznih modela, sažetaka, i deriviranih vrijednosti iz kolekcije podataka. Predstavljaju hibridnu metodu koja kombinira klastering algoritme i algoritme umjetne neuronske mreže za optimizaciju rasporeda.

# 3. Lokalno traženje

U ovom poglavlju su teorijski opisani algoritmi iz obitelji lokalnog traženja koji su u četvrtom poglavlju implementirani. Cilj je da čitatelj dobije dublji uvid u njihovo funkcioniranje da bi mogao lakše pratiti kada isti budu implementirani na problemu akademskog rasporeda.

## 3.1. Jednostavno lokalno traženje

**Jednostavno lokalno traženje** [31] je još poznato po nazivu „penjanje uzbrdo“ (*engl. hill climbing*). Algoritam je petlja koja se kreće u smjeru poboljšanja vrijednosti. Terminira kada dosegne vrh, odnosno kada niti jedan susjed više nema bolju vrijednost. Drugi način za opisati ovaj algoritam je da pronalazi lokalni optimum u skladu s definiranim susjedstvom. Odličan je u brzom pronalasku lokalnog optimuma, ali često ne uspije pronaći globalni optimum [32]. Ni na koji način se ne trudi izbjeći lokalni optimum, već ga jednostavno pronađe i stane. To nije problem kod konveksnih funkcija, ali je kod onih koje nisu konveksne. U praksi se često nadograđuje da bi se povećala šansa pronalaska globalnog optimuma. Algoritam je prikazan na pseudokodu 1.

```
1. Initialization: Find an initial solution x.
2. Repeat:
3.   x′ = best_neighbor(x)
4.   If x′ is not better than x: break
5.   x = x′
```

*Pseudokod 1 - Jednostavno lokalno traženje [31]*

Postoje razne varijante algoritma od kojih su neke [31]:

- **Najstrmiji uspon:** svi susjedi su uspoređeni po kvaliteti i uzima se najbolji.
- **Stohastika:** jedan susjed se slučajno odabere i na temelju njegove kvalitete se odluči da li ga treba uzeti.
- **Prvi izbor:** implementacija stohastičke varijante gdje se nasljednici generiraju slučajno dok jedan nije bolji od trenutnog stanja.
- **Slučajno ponovno pokretanje:** algoritam se ponavlja, ali svaki put s drugim slučajno generiranim ulazom.

## 3.2. Simulirano kaljenje

**Simulirano kaljenje** (*engl. simulated annealing*) je vjerojatnosna metoda za aproksimaciju globalnog optimuma dane funkcije koja može imati više lokalnih optimuma [33]. Često se koristi za vrlo teške optimizacijske probleme gdje su egzaktni algoritmi prespori, iako u pravilu daje samo aproksimaciju globalnog optimuma.

Metoda se bazira na analogiji simulacije kaljenja čvrstih tvari i rješavanja teških kombinatornih problema [34]. Po tome je i dobila ime „simulirano kaljenje“. U fizici, kaljenje predstavlja fizički proces u kojemu se čvrsta tvar zagrijava do tekuće faze gdje se čestice slučajno raspoređuju, nakon čega slijedi sporo hlađenje smanjivanjem temperature. Tako se sve čestice rasporede u području niske energije pod uvjetom da je maksimalna temperatura dovoljno visoka i da je hlađenje dovoljno sporo.

Simulirano kaljenje se razlikuje od jednostavnog lokalnog traženja po tome što ponekad prihvaća gore pomake ovisno o vjerojatnosnoj funkciji. Russell, Norvig i Davis [31] daju odličan primjer za razumijevanje algoritma: zamislite da vam je cilj ubaciti ping pong lopticu u najdublju pukotinu na grbavoj površini. Ako samo pustite lopticu da se odvalja, najvjerojatnije će završiti u lokalnom optimumu. Ako protresete površinu, možete lopticu izbaciti iz lokalnog optimuma. Trik je zatresti dovoljno jako da ju izbacite iz lokalnog optimuma, ali ne toliko jako da ju otjerate od globalnog optimuma. Simulirano kaljenje počinje tresti jako (na velikoj temperaturi) i postepeno smanjuje intenzitet (smanjivanjem temperature).

Specifičnije, algoritam simuliranog kaljenja funkcionira na ovaj način [31]: uzima slučajan pomak te ako pomak poboljšava stanje, uvijek se prihvaća. Inače algoritam prihvaća pomak s nekom vjerojatnošću manjom od jedan koja se smanjuje eksponencijalno što je pomak gori. Također se smanjuje kako se temperatura smanjuje što znači da su gori pomaci vjerojatniji kada je temperatura visoka, a manje vjerojatni kada je temperatura niska. Algoritam je prikazan na pseudokodu 2.

```
1. Initialization: Define temperature t. Define a stopping condition. Find an
     initial solution x.
2. Repeat until stopping condition is met:
3.   t = update_temperature(t)
4.   If t == 0: return x
5.   x' = random_neighbor(x)
6.   ΔE = f(x') - f(x)
7.   If ΔE > 0: x = x'
8.   Else: x = x' only with probability e^(ΔE/t)
```

*Pseudokod 2 - Simulirano kaljenje [31]*

## 3.3. Pretraživanje promjenjivim susjedstvom

**Pretraživanje promjenjivim susjedstvom** (*engl. variable neighborhood* search, skraćeno i dalje u tekstu: VNS) je metaheuristički okvir za razvijanje heuristika predložen 1997. godine [35]. Sustavno provodi proceduru promjene susjedstva, oboje u spuštanju do lokalnog minimuma i u izbjegavanju doline koja ju sadrži.

Okvir je izgrađen na sljedećim percepcijama [35]:
1. Lokalni minimum za jednu strukturu susjedstva nije nužno lokalni minimum za drugu strukturu susjedstva.
2. Globalni minimum je lokalni minimum s obzirom na sve moguće strukture susjedstva.
3. Za mnoge probleme, lokalni minimumi su relativno blizu jedan drugome s obzirom na jedno ili nekoliko susjedstva.

U usporedbi s drugim metaheuristikama, VNS i njegova proširenja zahtijevaju malo (ili ništa) parametara. Pored toga što daje dobra rješenja na relativno jednostavan način, daje i uvid u razloge takvih performansi, što može dovesti do učinkovitijih i sofisticiranijih implementacija [35].

### Silazak promjenjivim susjedstvom

Silazak promjenjivim susjedstvom (*engl. variable neighborhood descent*; skraćeno i dalje u tekstu: VND) [35] je algoritam lokalnog traženja koji pronalazi lokalni minimum u skladu s više susjedstva. Algoritam je prikazan na pseudokodu 3.

```
1. Initialization: Select the set of neighborhood structures N_l for
   l=1,…,,l_max; Find an initial solution x.
2. Repeat until no improvement is obtained:
3.   l = 1
4.   Repeat until l != l_max:
5.     Find the best neighbor x' od x (x' ∈ N_l(x))
6.     If x' is better than x, then x=x', l=1, else l=l+1
```

*Pseudokod 3 – Silazak promjenjivim susjedstvom [35]*

**Primjer:** Recimo da je $x = \{42, 13, -5, 9, 0, 21\}$ i da je potrebno definirati $l = 3$ susjedskih struktura nad $x$.

Podsjetimo se da je susjedstvo $N(x)$ skup čiji su elementi (relativno) malo različiti od elemenata od $x$.

Jedan od načina definiranja $l = 3$ susjedstva je sljedeći:
$N_1 = \{42.5, 13.5, -4.5, 9.5, 0.5, 21.5\}$ (svaki element je uvećan za 0.5)
$N_2 = \{41.6, 12.6, -5.4, 8.6, -0.4, 20.6\}$ (svaki element je umanjen za 0.4)
$N_3 = \{45, 16, -2, 12, 3, 24\}$ (svaki element je uvećan za 3)
Susjedstva su definirana nekom proizvoljnom i smislenom funkcijom nad $x$.

VND počinje od prvog susjedstva ($l = 1$) i završava sa zadnjim ($l = 3$). U svakoj iteraciji pronalazi susjeda $x'$ u susjedstvu $N_l(x)$ koji poboljšava funkciju cilja. Ako takav susjed ne postoji, $l$ se povećava za jedan, što znači da se prebacuje na iduće susjedstvo gdje ponavlja proces pronalaska susjeda. Ako je u novom susjedstvu pronađen susjed koji poboljšava funkciju cilja, $l$ se postavlja na jedan, što znači da proces pronalaska susjeda kreće ponovno iz prvog susjedstva. Ako prođe kroz sva susjedstva bez da pronađe bolju vrijednost funkcije cilja, terminira.

### Pretraživanje promjenjivim susjedstvom (VNS)

VNS je shema traženja koja sadrži algoritam trešnje (*engl. shake*) i algoritam lokalnog traženja. Razlika između VNS-a i VND-a je u tome što je VND samo varijanta lokalnog

traženja, dok VNS sadrži i varijantu lokalnog traženja (pr. VND, jednostavno lokalno traženje…), ali i algoritam trešnje koji slučajno „trese“ rješenje u skladu s vlastitim skupom susjedstva.

Ukoliko bi se implementirao VNS tako da kao algoritam lokalnog traženja koristi VND, onda bi trebalo definirati dva odvojena skupa susjedstva, jedan za trešnju, i jedan za VND. Takva shema se zove generalno pretraživanje promjenjivim susjedstvom (*engl. general variable neighborhood search)* [35].

```
1. Initialization: Select the set of neighborhood structures N_k, for k =
   1,…,k_max for „shaking“ phase; Select the set of neighborhood structures N_l,
   for l = 1,…,l_max for local search. Select initial solution x, Select
   stopping condition.
2. Repeat until stopping condition is met:
3.   (1) k = 1
4.   (2) Repeat until k != k_max
5.     (a) Shake. Generate random point x' from kth neighborhood N_k(x) of x.
6.     (b) Local search by VND.
7.       (b1) l = 1
8.       (b2) Repeat until l != l_max
9.         Find the best neighbor x'' of x' in N_l(x')
10.        If x'' is better than x', then x'=x'', l=1, else l=l+1
11.    (c) If this local optimum is better than x, then x=x'', k=1, else k=k+1
```

*Pseudokod 4 – Generalno pretraživanje promjenjivim susjedstvom [35]*

Na pseudokodu 4 je prikazan rad generalnog pretraživanja promjenjivim susjedstvom.
Da bi algoritam radio, potrebno je definirati sljedeće:
1. Skup susjedstva $N_k$ za $k = 1, \dots, k_{max}$ za proces trešnje.
2. Skup susjedstva $N_l$ za $l = 1, \dots, l_{max}$ za proces lokalnog traženja.
3. Inicijalno rješenje $x$
4. Uvjet zaustavljanja

Algoritam će prilikom svake iteracije napraviti trešnju, što znači da će dohvatiti slučajnog susjeda $x'$ u skupu $N_k(x)$. Zatim će pronaći lokalni optimum $x''$ od $x'$ uz pomoć proizvoljnog algoritma lokalnog traženja (u ovom slučaju to je VND). Na kraju će usporediti najbolji lokalni optimum $x''$ s do sad najboljim rješenjem $x$. Ako je taj $x''$ bolji

od $x$ onda $x$ postaje $x''$, i $k$ se postavlja na jedan (što znači da će trešnja generirati iz prvog susjedstva), inače se $k$ uvećava za jedan (što znači da će radi neuspjeha generirati iz idućeg susjedstva).

Na pseudokodu 4 se vidi da se VNS zapravo vrti u nedogled jer čak i kad $k$ postane $k_{max}$, proces kreće iz početka $(k = 1)$. Zato je potrebno definirati neki smisleni uvjet zaustavljanja (pr. nakon 30 minuta računanja).

## 3.4. Pohlepno nasumično adaptivno pretraživanje

Pohlepno nasumično adaptivno pretraživanje (*engl. greedy randomized adaptive search procedure*; skraćeno i dalje u tekstu: GRASP) [36] je iterativna tehnika slučajnog uzorkovanja gdje svaka iteracija proizvodi rješenje. Najbolje rješenje od svih iteracija se zadržava kao završno. Postoje dvije faze u svakoj GRASP iteraciji: prva je inteligentna konstrukcija inicijalnog rješenja uz pomoć adaptivne nasumične pohlepne funkcije, a druga je aplikacija lokalnog traženja na dobiveno rješenje.

```
1. grasp():
2.   for GRASP stopping condidion not satisfied:
3.     ConstructGreedyRandomizedSolution()
4.     LocalSearch(Solution)
5.     UpdateSolution(Solution, BestSolutionFound)
```

*Pseudokod 5 – Pohlepno nasumično adaptivno pretraživanje [36]*

GRASP iterira do proizvoljnog uvjeta zaustavljanja. U svakoj iteraciji konstruira cjelokupno rješenje (*ConstructGreedyRandomizedSolution*) i zatim aplicira lokalno traženje (*LocalSearch*) na to rješenje u cilju da ga poboljša. Funkcija *UpdateSolution* jednostavno postavlja *BestSolutionFound* na *Solution* ako je *Solution* bolji od *BestSolutionFound*.

```
1. ConstructGreedyRandomizedSolution():
2.   Solution = {}
3.   for Solution construction not done:
4.     MakeRCL(RCL)
5.     s = SelectRandomElement(RCL)
6.     Solution = Solution union s
7.     AdaptGreedyFunction(s)
```

*Pseudokod 6 – GRASP konstruiranje rješenja [36]*

Procedura *ConstructGreedyRandomizedSolution* iterativno konstruira rješenje, jedan po jedan element, sve dok svi elementi nisu uključeni. U proceduri *MakeRCL*, prvo se izrađuje lista kandidata (*engl. candidates list*) koja predstavlja listu potencijalnih komponenti rješenja za trenutnu iteraciju. Zatim se svi elementi iz liste kandidata sortiraju po pohlepnoj funkciji koja mjeri (kratkovidnu) korist selektiranja pojedinog kandidata. Iz sortirane liste se odabire samo N najboljih kandidata da bi se stvorila ograničenja lista kandidata (*engl. restricted candidates list,* skraćeno RCL). Funkcija *SelectElementAtRandom* odabire slučajan element iz ograničene liste kandidata. Dakle, ne mora se odabrati striktno najbolji kandidat, već jedan od N najboljih, koji se dalje ukomponira u rješenje. Nakon toga se pohlepna funkcija adaptira s obzirom na odabranog kandidata što znači da će biti ažurirano buduće računanje (kratkovidne) koristi.

Nije garantirano da će rješenje iz funkcije *ConstructGreedyRandomizedSolution* biti lokalno optimalno pa se zato aplicira lokalno traženje.

Svaka GRASP iteracija proizvodi rješenje iz nepoznate distribucije svih dostižnih rezultata [36]. Srednja vrijednost i varijanca su funkcije restriktivne prirode liste kandidata. Ako je kardinalnost ograničene liste kandidata jednaka jedan, onda će se proizvesti samo jedno rješenje i varijanca će biti nula. Ako se poveća kardinalnost ograničene liste kandidata, onda će više različitih rješenja biti proizvedeno, što implicira veću varijancu.

Srednja vrijednost funkcije cilja rješenja će u pravilu biti veće što je varijanca manja, i obratno, srednja vrijednost u pravilu manja što je varijanca veća [36]. Intuitivno je moguće zaključiti da će kod manje varijance rješenje (iako kvalitetnije u prosjeku) biti vjerojatnije neoptimalno, dok će kod veće varijance rješenje (iako manje kvalitetno u prosjeku) imati veću vjerojatnost da bude optimalno.

Drugim riječima, kod GRASP-a je uvijek pitanje u kojem omjeru postaviti mjeru pohlepe i slučajnosti. Više pohlepe i manje slučajnosti znači kvalitetnija rješenja, ali uža distribucija, dok više slučajnosti i manje pohlepe znači manje kvalitetna rješenja, ali šira distribucija.

## 3.5. Iterirano lokalno traženje

Iterirano lokalno traženje polazi od ideje: „da li je moguće poboljšati lokalno traženje korištenjem iteracija?“ i odgovor je da [37].

**Slučajno ponavljanje** [37]: najjednostavniji slučaj iteriranog lokalnog traženja gdje se u svakoj iteraciji lokalno traženje poziva s drugim slučajnim ulazom. Korisno je u situacijama gdje ostale varijante nisu moguće, ali je poznato da slučajno uzorkovanje ima sve manju i manju vjerojatnost pronalaska optimalnog rješenja kako se povećava veličina ulaza. Da bi se povećala vjerojatnost pronalaska optimalnog rješenja, koriste se varijante pristranog uzorkovanja. Iterirano lokalno traženje sa slučajnim ponavljanjem se još zove i **ponavljano lokalno traženje** (*engl. repeated local search*).

**Iterirano lokalno traženje** [37] se sastoji od perturbacijskog algoritma i lokalnog traženja. Perturbacijski algoritam se može dizajnirati tako da perturbacija bude jača ili slabija. Prejaka perturbacija znači slučajno ponavljanje, a preslaba stvaranje gotovo istih rješenja iz uske distribucije. Potrebno je pažljivo balansirati jačinu perturbacije da bi dobili ciljanu razinu pristranosti.

```
1. s0 = GenerateInitialSolution()
2. s* = LocalSearch(s0)
3. Repeat until termination condition is met:
4.   s' = Perturbation(s*, history)
5.   s*' = LocalSearch(s')
6.   s* = AcceptanceCriterion(s*, s*', history)
```

*Pseudokod 7 - Iterirano lokalno traženje [37]*

Popularan način pojačavanja perturbacije je korištenje znanja iz prijašnjih iteracija da bi se generiralo bolje inicijalno rješenje [37]. Taj pristup zahtijeva dodatnu memoriju za pohranu povijesti, ali se pokazao kao puno bolji izbor za mnoge probleme.

# 4. Sustav za rješavanje RRULE rasporeda

U ovom poglavlju je predstavljen sustav za rješavanje problema akademskog rasporeda baziranog na ponavljajućim pravilima norme iCalendar [39]. Implementirani sustav se može pronaći na poveznici: https://github.com/jjurinci/scheduler_system.

## 4.1. Definicija problema

Za dani skup predmeta, dvorana, profesora, semestara, vremena, i ograničenja nad njima, potrebno je smjestiti sve grupe predmeta (tj. raspove) u dvorane na vremenske položaje tako da se minimizira broj prekršenih ograničenja. Grupe predmeta se mogu održavati po proizvoljnom ponavljajućem pravilu.

### 4.1.1. Rasp - definicija

Rasp je jedinica predmeta koja ide u raspored. Predmet može imati više raspova gdje je svaki rasp jedinstven po uređenoj trojci (identifikator predmeta, oblik izvođenja predmeta, oznaka grupe).

**Primjer 1:** ako je predmet „Matematika“, onda raspovi mogu biti „Matematika Predavanje Grupa 1“, „Matematika Vježbe Grupa 1“, „Matematika Vježbe Grupa 2“, i sl.
**Primjer 2:** ako je predmet „Programiranje“, onda raspovi mogu biti „Programiranje Predavanje Grupa 1“, „Programiranje Predavanje Grupa 2“, „Programiranje Seminar Grupa 1“, „Programiranje Seminar Grupa 2“, i sl.
**Primjer 3:** ako je predmet „Poslovni informacijski sustavi“, onda raspovi mogu biti „Poslovni informacijski sustavi Predavanje Grupa 1“, „Poslovni informacijski sustavi Predavanje Grupa 2“, „Poslovni informacijski sustavi Vježbe Grupa 1“, „Poslovni informacijski sustavi Vježbe Grupa 2“, i sl.

Rasp se može shvatiti i kao „grupa predmeta“ jer predstavlja upravo tu razinu granularnosti u rasporedu.

### 4.1.2. Ograničenja problema

Ograničenja problema su:
1. Svaki rasp se mora održavati po ponavljajućem vremenskom pravilu (pr. svaki drugi tjedan, ili samo šest dana počevši od nekog datuma, i sl.)
2. Dvorana ne smije imati dva ili više raspa u isto vrijeme
3. Profesor ne smije predavati dva ili više raspa u isto vrijeme
4. Semestar ne smije imati dva ili više obaveznih raspova u isto vrijeme, osim ako nije riječ o grupi istog tipa (primjerice, na „Mat_P1“ i Mat_P2“ se ograničenje ne odnosi)
5. Semestar smije imati izborne raspove u isto vrijeme, osim ako nije riječ o raspovima istog predmeta, ali različitog tipa (primjerice, „Erp_P1“ i „Erp_V1“ se ne mogu održavati u isto vrijeme. Oba raspa imaju isti predmet „Erp“ i različit tip, prvi P=predavanje, drugi V=vježbe)
6. Kapacitet dvorane ne smije biti manji od broja studenata raspa
7. Raspovi koji zahtijevaju računala moraju biti postavljeni u računalnu dvoranu
8. Ako je rasp fiksiran na neku dvoranu, u tu dvoranu ga treba i postaviti
9. Ako je rasp fiksiran na neki dan, na taj dan ga treba i postaviti
10. Ako je rasp fiksiran na neki sat, na taj sat ga treba i postaviti

Pošto su ograničenja četiri i pet malo kompliciranija, bit će dodatno pojašnjena. Rasp može biti obavezan u jednom skupu semestara, i izborni u drugom. Primjerice, rasp „Kriptografija_P1“ se može održavati kao obavezan u semestrima „FIPU preddiplomski semestar 5“, „TFPU preddiplomski semestar 3“ i izborni u semestrima „FIPU diplomski semestar 1“, „TFPU diplomski semestar 1“.

Ograničenje četiri se odnosi na obavezne raspove u nekom semestru. Ono ne dozvoljava kolizije između njih, osim u slučaju grupa istog tipa (pr. „Mat_P1“, i „Mat_P2“) jer takve grupe neće imati zajedničkih studenata pa ih se može staviti na isto vrijeme (što se studenata tiče).

Ograničenje pet se odnosi na izborne raspove u nekom semestru. Ono dozvoljava da se izborni predmeti održavaju u isto vrijeme, osim u slučaju istog predmeta i različitog tipa

(pr. „Erp_P1“, i „Erp_V1“). Ovakav pristup proizlazi iz činjenice da u trenutku kreiranja rasporeda ne znamo koji studenti su upisali koje izborne predmete pa posljedično ne znamo koji izborni imaju zajedničke studente (kada bi znali, onda bi im zabranili održavanje u isto vrijeme). Zato pretpostavljamo da se svi mogu održavati u isto vrijeme i pustimo algoritmu da ih rasporedi na pogodna vremena. Kada korisnik dobije informaciju o tome koji studenti su upisali koje izborne, može ručno (ili uz pomoć računala) podesiti raspored.

## 4.2. Gramatika RRULE

Postavlja se pitanje: „koji alat koristiti za reprezentaciju ponavljajućeg pravila raspova?“. U ovom radu je korišten alat RRULE iz specifikacije iCalendar RFC 5455 [39] koji definira gramatiku za ponavljajuća pravila.

Za lakše razumijevanje, gramatika će biti objašnjena kroz Python biblioteku dateutil [40] koja sadrži modul „rrule“ implementiran po specifikaciji iCalendar RFC 5455.

Funkcija „rrule“ prima određen broj argumenata i kao rezultat vraća listu datuma koji odgovaraju ponavljajućem pravilu.

lista_datuma = rrule(…argumenti…)

Argumenti su sljedeći:

| Argument | Značenje |
|---|---|
| **freq** | Frekvencija ponavljanja. Pr. YEARLY, MONTHLY, WEEKLY, DAILY, … |
| **dtstart** | Datum početka |
| **interval** | Interval između frekvencija. Pr. freq=WEEKLY i interval=2 znači svaki drugi tjedan. |
| **wkst** | Dan početka tjedna (*engl. week start*). Pr. MO za Monday. |

| | |
|---|---|
| **count** | Ako zadan, određuje koliko ponavljanja će biti. Pr. count=5 znači ukupno 5 datuma. |
| **until** | Ako zadan, određuje datum kojeg zadnji datum u listi ne smije prekoračiti. |
| **bysetpos** | Ako zadan, određuje n-to pojavljivanje u frekvencijskom periodu. |
| **bymonth** | Ako zadan, određuje mjesece na kojima će se aplicirati ponavljanje. |
| **bymonthday** | Ako zadan, određuje dane u mjesecu na koje će se aplicirati ponavljanje. |
| **byyearday** | Ako zadan, određuje dane u godini na koje će se aplicirati ponavljanje. |
| **byeaster** | Ako zadan, određuje ofset do uskršnje nedjelje. |
| **byweekno** | Ako zadan, određuje brojeve tjedana u godini na koje će se aplicirati ponavljanje. |
| **byweekday** | Ako zadan, određuje dane u tjednu na koje će se aplicirati ponavljanje. |
| **byhour** | Ako zadan, određuje sate na koje će se aplicirati ponavljanje. |
| **byminute** | Ako zadan, određuje minute na koje će se aplicirati ponavljanje. |
| **bysecond** | Ako zadan, određuje sekunde na koje će se aplicirati ponavljanje. |
| **cache** | Ako zadan, sprema rrule u memoriju za brži dohvat. |

*Tablica 1 - Značenje RRULE argumenata*

Slika 6 prikazuje 10 dnevnih datuma počevši od 02.09.1997 u 9:00.

```
>>> list(rrule(DAILY, count=10,
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 3, 9, 0),
 datetime.datetime(1997, 9, 4, 9, 0),
 datetime.datetime(1997, 9, 5, 9, 0),
 datetime.datetime(1997, 9, 6, 9, 0),
 datetime.datetime(1997, 9, 7, 9, 0),
 datetime.datetime(1997, 9, 8, 9, 0),
 datetime.datetime(1997, 9, 9, 9, 0),
 datetime.datetime(1997, 9, 10, 9, 0),
 datetime.datetime(1997, 9, 11, 9, 0)]
```

*Slika 6 - Primjer RRULE-a – Dnevno 10 ponavljanja [40]*

Slika 7 prikazuje osam datuma, svaki drugi tjedan na utorak i četvrtak počevši od 02.09.1997 u 9:00.

```
>>> list(rrule(WEEKLY, interval=2, count=8,
...            wkst=SU, byweekday=(TU,TH),
...            dtstart=parse("19970902T090000")))
[datetime.datetime(1997, 9, 2, 9, 0),
 datetime.datetime(1997, 9, 4, 9, 0),
 datetime.datetime(1997, 9, 16, 9, 0),
 datetime.datetime(1997, 9, 18, 9, 0),
 datetime.datetime(1997, 9, 30, 9, 0),
 datetime.datetime(1997, 10, 2, 9, 0),
 datetime.datetime(1997, 10, 14, 9, 0),
 datetime.datetime(1997, 10, 16, 9, 0)]
```

*Slika 7 - Primjer RRULE-a – Svaki drugi tjedan na utorak i četvrtak [40]*

RRULE je također moguće pretvoriti u *string* format. Primjerice, RRULE sa slike 7 je prikazan u *string* formatu na slici 8.

```
DTSTART:19970902T090000\nRRULE:FREQ=WEEKLY;INTERVAL=2;WKST=SU;COUNT=8;BYDAY=TU,TH
```

*Slika 8 - Primjer string reprezentacije RRULE-a [40]*

### 4.2.1. Akademski RRULE

U kontekstu akademskog rasporeda RRULE raspa ne može biti definiran na bilo koji način. On se sastoji od liste datuma i ne smije se dogoditi da se neki datum održava:

1. Izvan vremenskog razdoblja definiranog u akademskom kalendaru. (pr. u 9. mjesecu, a akademska godina počinje tek u 10. mjesecu)
2. Na nedozvoljene dane (pr. subota ili nedjelja).
3. Tako da počinje na sat koji nije akademski početni sat. Ako imamo početne akademske satove: „8:00, 8:50, 9:35, …, 20:50“ onda početni sat RRULE-a (odnosno njegovog „dtstart“ argumenta) mora biti jedan od njih.

RRULE koji zadovoljava navedena ograničenja zvat ćemo **akademski RRULE**. Sustav sadrži pomoćni algoritam koji provjerava da li su svi RRULE-ovi raspova akademski RRULE-ovi.

## 4.3. Ulazni podaci

U ovom poglavlju je opisan cjelokupni ulaz u sustav koji se sastoji od skupa .csv datoteka. Za svaku datoteku su u tablici prikazane kolone i njihovo značenje.

**start_end_year.csv:**

| Kolona | Značenje |
|---|---|
| **start_semester_date** | Datum početka semestra. |
| **end_semester_date** | Datum završetka semestra. |

*Tablica 2 - Značenje kolona iz datoteke start_end_year.csv*

**day_structure.csv:**

| Kolona | Značenje |
|---|---|
| # | Redni broj akademskog vremenskog bloka. |
| **timeblock** | Akademski vremenski blok u formatu „hr:min-hr:min“. |

*Tablica 3 - Značenje kolona iz datoteke day_structure.csv*

**faculties.csv:**

| Kolona | Značenje |
|---|---|
| **Id** | Identifikator fakulteta. |
| **name** | Ime fakulteta. |

*Tablica 4 - Značenje kolona iz datoteke faculties.csv*

**study_programmes.csv**

| Kolona | Značenje |
|---|---|
| **id** | Identifikator studijskog programa. |
| **name** | Ime studijskog programa. |
| **faculty_id** | Identifikator fakulteta kojemu studijski program pripada. |

*Tablica 5 - Značenje kolona iz datoteke study_programmes.csv*

**semesters.csv:**

| Kolona | Značenje |
|---|---|
| **id** | Identifikator semestra. |
| **num_semester** | Redni broj semestra u studijskom programu. |
| **season** | Zimski ili ljetni semestar. |

| | |
|---|---|
| **num_students** | Broj studenata u semestru. |
| **study_programme_id** | Identifikator studijskog programa kojemu semestar pripada. |

*Tablica 6 - Značenje kolona iz datoteke semesters.csv*

**subjects.csv:**

| Kolona | Značenje |
|---|---|
| **id** | Identifikator predmeta. |
| **name** | Ime predmeta. |
| **mandatory_in_semester_ids** | Zarezom odvojena lista identifikatora semestara u kojima je predmet obavezan. |
| **optional_in_semester_ids** | Zarezom odvojena lista identifikatora semestara u kojima je predmet izborni. |

*Tablica 7 - Značenje kolona iz datoteke subjects.csv*

**rasps.csv:**

| Kolona | Značenje |
|---|---|
| **id** | Identifikator raspa. |
| **professor_id** | Identifikator profesora koji predaje rasp. |
| **subject_id** | Identifikator predmeta kojemu pripada rasp. |
| **type** | Tip predmeta raspa. Pr. P=Predavanja, V=Vježbe, S=Seminar, … |
| **group** | Grupa raspa. |
| **duration** | Trajanje raspa u broju akademskih sati. |
| **needs_computers** | Ako je 1: raspu treba računalna učionica. Ako je 0: ne treba. |

| | |
|---|---|
| **fix_at_room_id** | Ako zadan, određuje dvoranu u koju će rasp biti smješten. |
| **random_dtstart_weekday** | Ako je 1: određuje algoritmu da odabere slučajan početni dan u dtstart tjednu. Pr. „odi na tjedan u kojemu je 04.10.2021. i umjesto 04.10.2021. (koji je ponedjeljak) odaberi slučajan datum iz skupa {ponedjeljak, utorak, srijeda, četvrtak, petak}“. Ako je 0, dan će biti fiksiran na dtstart od rrule vrijednosti. |
| **rrule** | Standardan rrule string omotan u navodne znakove da bi se izbjeglo automatsko .csv formatiranje. Ako se u dtstart-u postavi početni sat onda će algoritam fiksirati rasp na taj sat (pr. „DTSTART:20211004T080000“ će fiksirati rasp na 08:00). Ako se postavi 00:00 kao početni sat (pr. „DTSTART:20211004T000000“) onda će algoritam sam birati početni sat iz skupa svih početnih akademskih satova. |

*Tablica 8 - Značenje kolona iz datoteke rasps.csv*

**classrooms.csv:**

| Kolona | Značenje |
|---|---|
| **id** | Identifikator dvorane. |
| **name** | Ime dvorane. |
| **capacity** | Kapacitet dvorane. |
| **has_computers** | Da li dvorana ima računala. |

*Tablica 9 - Značenje kolona iz datoteke classrooms.csv*

**professors.csv:**

| Kolona | Značenje |
|---|---|
| **id** | Identifikator profesora. |
| **name** | Ime profesora. |

*Tablica 10 - Značenje kolona iz datoteke professors.csv*

**classroom_available.csv:**

| Kolona | Značenje |
|---|---|
| **room_id** | Identifikator dvorane. |
| **monday** | Vrijeme kada je dvorana ponedjeljkom slobodna. T znači cijeli dan, F znači nikad, a zarezom odvojena lista predstavlja listu parova koji određuju vrijeme. Pr. „1,6,11,16“ znači da je dvorana slobodna od 1. do 6. akademskog sata (uključivo), i od 11. do 16. akademskog sata (uključivo), dok ostala vremena nisu slobodna. Dakle, u ovom slučaju akademski satovi 7,8,9,10 nisu slobodni. |
| **tuesday** | Ista logika kao za monday, samo što vrijedi za utorak. |
| **wednesday** | Ista logika kao za monday, samo što vrijedi za srijedu. |
| **thursday** | Ista logika kao za monday, samo što vrijedi za četvrtak. |
| **friday** | Ista logika kao za monday, samo što vrijedi za petak. |

*Tablica 11 - Značenje kolona iz datoteke classroom_available.csv*

**professor_available.csv:**

| Kolona | Značenje |
|---|---|
| **professor_id** | Identifikator profesora. |
| **monday** | Vrijeme kada je profesor ponedjeljkom slobodna. T znači cijeli dan, F znači nikad, a zarezom odvojena lista predstavlja listu parova koji određuju vrijeme. Pr. „1,6,11,16“ znači da je dvorana slobodna od 1. do 6. akademskog sata (uključivo), i od 11. do 16. akademskog sata (uključivo), dok ostala vremena nisu slobodna. Dakle, u ovom slučaju akademski satovi 7,8,9,10 nisu slobodni. |
| **tuesday** | Ista logika kao za monday, samo što vrijedi za utorak. |
| **wednesday** | Ista logika kao za monday, samo što vrijedi za srijedu. |
| **thursday** | Ista logika kao za monday, samo što vrijedi za četvrtak. |
| **friday** | Ista logika kao za monday, samo što vrijedi za petak. |

*Tablica 12 - Značenje kolona za datoteku professor_available.csv*

## 4.4. Strukture podataka i algoritmi

Da bi se uspješno koristili ulazni podaci prvo ih je potrebno spremiti u pogodne strukture podataka. U implementiranom projektu svi podaci su spremljeni u jednu *State* strukturu koja sadrži raspored, ocjenu, stanje kršenja ograničenja i sve što je potrebno da bi optimizacijski algoritmi mogli manipulirati rješenjem. Tablica 13 prikazuje *State* strukturu.

| Atribut | Kratko objašnjenje |
|---|---|
| **timetable** | Rječnik gdje je ključ rasp, a vrijednost (dvorana, vremenski položaj). |
| **grade** | Ocjena rasporeda. |
| **is_winter** | True ako je semestar zimski inače False. |

| | |
|---|---|
| **semesters** | Rječnik gdje je ključ identifikator semestra, a vrijednost semestar. |
| **rooms** | Rječnik gdje je ključ identifikator dvorane, a vrijednost dvorana. |
| **rasps** | Lista svih raspova. |
| **students_per_rasp** | Rječnik gdje je ključ identifikator raspa, a vrijednost broj studenata raspa. |
| **time_structure** | Struktura za vremenske okvire. |
| **initial_constraints** | Struktura koja sadrži početna ograničenja za dvorane, profesore, i semestre. Ne mijenja se. |
| **mutable_constraints** | Struktura koja sadrži stanje kršenja ograničenja za dvorane, profesore, i semestre. Mijenja se. |
| **groups** | Rječnik gdje je ključ (identifikator predmeta + tip predmeta), a vrijednost lista raspova koja reprezentira grupe ključa. |
| **subject_types** | Rječnik gdje je ključ identifikator predmeta, a vrijednost lista tipova predmeta. |
| **rasp_rrules** | Rječnik gdje je ključ identifikator raspa, a vrijednost rječnik koji sadrži RRULE podatke o raspu. |
| **rrule_table** | Lista koja sadrži memoizirane moguće RRULE puteve raspova. |

*Tablica 13 - State struktura*

### 4.4.1. Praćenje ograničenja

Svaki rasp ima svoj RRULE koji treba pratiti kroz dvorane, profesore, i semestre. Postavlja se pitanje kako to napraviti pošto se kolizije između RRULE-ova mogu dogoditi bilo kada između njihovih datuma održavanja? Recimo da se jedan rasp održava samo šest dana od 08:50-10:25, a drugi rasp svaki tjedan u utorak od 09:35-11:15. Kako saznati kolika je kolizija između njihovih vremena održavanja u dvoranama, profesorima, i semestrima?

Da bi čitatelj bolje razumio rješenje tog problema, u nastavku će biti predstavljen niz opservacija na kojima se ono temelji. Svaka opservacija logički slijedi iz prethodnih i na kraju je predstavljeno rješenje.

**Opservacija 1:** Semestri imaju početni i završni datum.

**Primjer:** FIPU zimski semestar u 2021. godini je počeo 4.10.2021. i završio 28.1.2022.
Ne može se dogoditi da se neki predmet (zimskog semestra) održava izvan tog datumskog raspona. To implicira da RRULE svakog raspa mora biti unutar vremenskog bloka od 4.10.2021 do 28.1.2022.

**Opservacija 2:** Između početnog i završnog datuma u semestru postoji određen broj tjedana. U svakom tjednu postoji određen broj radnih dana. U svakom radnom danu postoji određen broj akademskih sati.

**Primjer:** Uzmimo početni datum 4.10.2021. i završni datum 28.1.2022. Taj raspon sadrži točno 17 tjedana (uključivo). Svaki tjedan ima određen broj dana na koje će se raspoređivati raspovi. Pretpostavljena vrijednost je pet dana u tjednu koji znače „ponedjeljak, utorak, srijeda, četvrtak, petak“. Svaki dan ima određen broj akademskih sati. Primjer svih (početnih) akademskih sati u danu može biti: 08:00, 08:50, 09:40, 10:30, 11:20, 12:10, 13:00, 14:00, 14:45, 15:30, 16:15, 17:00, 17:45, 18:30, 19:15, 20:05.

Kada bi ih izbrojili, dobili bi da ih ima 16. Dakle, imamo 17 tjedana, 5 dana (u svakom tjednu), i 16 sati (u svakom danu).

Definirajmo: $NUM\_WEEKS = 17$, $NUM\_DAYS = 5$, i $NUM\_HOURS = 16$

**Opservacija 3:** Svaki akademski datum u intervalu [početni datum semestra, završni datum semestra] može se zapisati kao uređena trojka (tjedan, dan, sat).

**Primjer:** Akademski datum je datum koji počinje na neki tjedan iz intervala [početni datum semestra, završni datum semestra], na neki radni dan iz tog tjedna, i na neki akademski sat iz tog dana. Akademskih datuma ima točno $NUM\_WEEKS * NUM\_DAYS * NUM\_HOURS$ gdje su spomenute varijable one iz opservacije 2.

Ako je $NUM\_WEEKS = 17$, $NUM\_DAYS = 5$, i $NUM\_HOURS = 16$ onda će skup dopuštenih datuma biti:
$\{tjedan\ 1, tjedan\ 2, \dots, tjedan\ 17\} \times \{ponedjeljak, utorak, srijeda, četvrtak, petak\} \times \{sat\ 1, sat\ 2, \dots, sat\ 16\}$ gdje je × kartezijev produkt.

Vratimo se na uređenu trojku (tjedan, dan, sat). Ako je početni datum 4.10.2021. i završni 28.1.2022. onda se „6.10.2021. u 8:50“ može zapisati kao (0, 2, 1) gdje prvi element (0) označava prvi tjedan, drugi element (2) treći dan - srijedu, i treći element (1) drugi sat 08:50. Još jedan primjer: „20.12.2021. u 14:00“ može se zapisati kao (11, 0, 7) gdje prvi element (11) označava 12. tjedan, drugi element (0) prvi dan (ponedjeljak), i treći element (7) osmi sat 14:00.

Primijetite da „19.12.2021. u 14:00“ ne bi bio dopušten datum jer se radi o nedjelji, a NUM_DAYS je 5 što ne obuhvaća nedjelju.

**Opservacija 4:** RRULE svakog raspa se može zapisati kao lista uređenih trojki (tjedan, dan, sat).

**Primjer 1:** Recimo da imamo RRULE „svaki drugi tjedan od 4.10.2021 do 28.1.2022. u 08:50“.

Konkretno to bi bili sljedeći datumi (svi u 08:50): [4.10.2021., 18.10.2021., 1.11.2021., 15.11.2021., 29.11.2021., 13.12.2021., 27.12.2021., 10.1.2022., 24.1.2022.]
Tu listu datuma možemo zapisati kao listu uređenih trojki (tjedan, dan, sat) ako izračunamo na koji se tjedan, dan i sat svaki datum održava:
[(0,0,1), (2,0,1), (4,0,1), (6,0,1), (8,0,1), (10,0,1), (12,0,1), (14,0,1), (16,0,1)]

Tjedan se svaki put povećava za dva, dok su dan (0 = ponedjeljak) i sat (1 = 08:50) uvijek isti, što je u skladu s očekivanjima.

**Primjer 2:** Recimo da imamo RRULE „četiri dana počevši od 16.11.2021. do 19.11.2021. (uključivo) u 11:20“

Konkretno to bi bili sljedeći datumi (svu u 11:20):
[16.11.2021., 17.11.2021., 18.11.2021., 19.11.2021.]
Tu listu datuma možemo zapisati kao listu uređenih trojki (tjedan, dan, sat):
[(6,1,4), (6,2,4), (6,3,4), (6,4,4)]

Vidimo da su svi datumi u sedmom tjednu, i to na utorak (1), srijedu (2), četvrtak (3), i petak (4) u 11:20 (4).

**Opservacija 5:** U cjelobrojnom 3D polju [NUM_WEEKS][NUM_DAYS][NUM_HOURS] moguće je pratiti sve kolizije RRULE-ova.

**Primjer:** Recimo da je $NUM\_WEEKS = 17$, $NUM\_DAYS = 5$, i $NUM\_HOURS = 16$.
Definirajmo cjelobrojno 3D polje: int polje[17][5][16]

Iz opservacije 4 znamo da možemo pretvoriti svaki akademski RRULE u listu uređenih trojki (tjedan, dan, sat). To nam dopušta da pratimo kolizije RRULE-ova na sljedeći način:

1. Pretvori sve RRULE-ove u liste uređenih trojki
2. Iteriraj po (tjedan, dan, sat) iz lista:
    - Inkrementiraj polje[tjedan][dan][sat]

Nakon izvršenja gornjeg algoritma polje će biti ažuriramo s informacijama o kolizijama. Preciznije, ako je za neki (tjedan, dan, sat) polje[tjedan][dan[sat] jednako:

- 0, onda nema kolizija na taj (tjedan, dan, sat).
- 1, onda nema kolizija na taj (tjedan, dan, sat), ali postoji 1 rasp koji se tada izvodi.
- N>1, onda (tjedan, dan, sat) ima točno N kolizija, jer se N raspova tada održava.

Za kraj, moguće je znatno uštediti na memoriji ako se umjesto *int* (četiri bajta) tip elementa polja postavi na osam bitni *unsigned integer*, tzv. „*uint8*“ (jedan bajt). *Unsigned* ima smisla jer algoritam samo inkrementira polje, negativne vrijednosti u ovom kontekstu nemaju nikakvo značenje. Tip uint8 ima raspon od [0,255] što je u većini slučajeva dovoljno.

Gornje 3D polje ima $17 * 5 * 16 = 1360$ elemenata.
S *int* tipom polje bi zauzelo $1360 * 4 = 5440$ bajtova.
S *uint8* tipom polje bi zauzelo $1360 * 1 = 1360$ bajtova.
Dakle četiri puta manje sa *uint8*, što je značajno, pogotovo kada se stvori N takvih trodimenzionalnih polja.

**Zaključak:** Ako svakoj dvorani, profesoru, i semestru dodijelimo po jedno cjelobrojno polje[NUM_WEEKS][NUM_DAYS][NUM_HOURS], moći ćemo pratiti RRULE kolizije za svakog od njih zasebno.

Kao što je već objašnjeno u prethodnim poglavljima, svaki rasp predaje neki profesor nekim semestrima u nekoj dvorani i posljedično treba zasebno pratiti RRULE kolizije za svakog od njih. Za to će biti korištene ispod navedene strukture podataka.

Definirajmo:
**rooms_occupied** = rječnik gdje je ključ identifikator dvorane, a vrijednost uint8 polje[NUM_WEEKS][NUM_DAYS][NUM_HOURS] inicijalizirano na nula
**profs_occupied** = rječnik gdje je ključ identifikator profesora, a vrijednost uint8 polje[NUM_WEEKS][NUM_DAYS][NUM_HOURS] inicijalizirano na nula
**sems_occupied** = rječnik gdje je ključ identifikator semestra, a vrijednost uint8 polje[NUM_WEEKS][NUM_DAYS][NUM_HOURS] inicijalizirano na nula

**Taksing funkcije** su funkcije koje dodaju prisutnost RRULE-a raspa u dvorani, profesoru, i semestru.
**Untaksing funkcije** su funkcije koje uklanjaju prisutnost RRULE-a raspa iz dvorane, profesora, i semestra.

Optimizacijski algoritam funkcionira tako da isprobava dvorane i početne datume raspova s ciljem da pronađe što bolju konfiguraciju, odnosno onu sa što manje kolizija. Mijenjanje dvorane ili početnog datuma raspa ne mijenja njegov tip ponavljanja (već samo dvoranu i datum kada počinje). Primjerice, umjesto „svaki tjedan *četvrtkom u 8:00* u dvorani *403*“ može se promijeniti na „svaki tjedan *utorkom u 11:20* u dvorani *202*“. Da bi se ta promjena uspješno obavila, prvo je potrebno untaksirati stari RRULE „svaki tjedan četvrtkom u 8:00 dvorana 403“ iz dvorane, profesora, i semestara. Zatim je potrebno taksirati novi RRULE „svaki tjedan utorkom u 11:20 dvorana 202“ u dvorani, profesoru, i semestrima.

Taksing i untaksing funkcije direktno manipuliraju sa rooms_occupied, profs_occupied, i sems_occupied rječnicima. Njihova zadaća je pobrinuti se da se svaka promjena dvorane i početnog datuma RRULE-a ispravno ažurira u relevantnim trodimenzionalnim poljima. Ispod se nalazi pseudokod taksing i untaksing funkcija za dvorane, profesore, i semestre. Nakon svake taksing i untaksing operacije poziva se funkcija „update_grade“ koja ažurira ocjenu u skladu s izračunatim kolizijama.

```
1. tax_rooms(rooms_occupied, room_id, rasp):
2.   for (week, day, hour) in rasp_rrule:
3.     rooms_occupied[room_id][week][day][hour] += 1
4.   update_grade(num_collisions)
```

*Pseudokod 8 - Taksiranje dvorane*

```
1. untax_rooms(rooms_occupied, room_id, rasp):
2.   for (week, day, hour) in rasp_rrule:
3.     rooms_occupied[room_id][week][day][hour] -= 1
4.   update_grade(-num_collisions)
```

*Pseudokod 9 - Untaksiranje dvorane*

```
1. tax_profs(profs_occupied, prof_id, rasp):
2.   for (week, day, hour) in rasp_rrule:
3.     profs_occupied[prof_id][week][day][hour] += 1
4.   update_grade(num_collisions)
```

*Pseudokod 10 - Taksiranje profesora*

```
1. untax_profs(profs_occupied, prof_id, rasp):
2.   for (week, day, hour) in rasp_rrule:
3.     profs_occupied[prof_id][week][day][hour] -= 1
4.   update_grade(-num_collisions)
```

*Pseudokod 11 - Untaksiranje profesora*

Semestri su kompleksniji od dvorana i profesora jer imaju dodatne zahtjeve koji su opisani u poglavlju 4.1.2.

```
1. tax_sems(sems_occupied, rasp):
2.   for sem_id in rasp.semesters:
3.     if rasp is mandatory in semester:
4.       tax_sem_mandatory(sems_occupied[sem_id], rasp)
5.     else:
6.       tax_sem_optional(sems_occupied[sem_id], rasp)
```

*Pseudokod 12 - Taksiranje semestara*

```
1. untax_sems(sems_occupied, rasp):
2.   for sem_id in rasp.semesters:
3.     if rasp is mandatory in semester:
4.       untax_sem_mandatory(sems_occupied[sem_id], rasp)
5.     else:
6.       untax_sem_optional(sems_occupied[sem_id], rasp)
```

*Pseudokod 13 - Untaksiranje semestara*

```
1. tax_sem_mandatory(sem_occupied, rasp):
2.   for (week, day, hour) in rasp_rrule:
3.     if no group of same type is at sem_occupied[week][day][hour]:
4.       sem_occupied[week][day][hour] += 1
5.   update_grade(num_collisions)
```

*Pseudokod 14 - Taksiranje obaveznog semestra*

```
1. tax_sem_optional(sem_occupied, rasp):
2.   for (week, day, hour) in rasp_rrule:
3.     if no group of same type is at sem_occupied[week][day][hour] AND
       (no optional rasps at sem_occupied[week][day][hour] OR
       group of different type is at sem_occupied[week][day][hour]):
4.       sem_occupied[week][day][hour] += 1
5.   update_grade(num_collisions)
```

*Pseudokod 15 - Taksiranje izbornog semestra*

```
1. untax_sem_mandatory(sem_occupied, rasp):
2.   for (week, day, hour) in rasp_rrule:
3.     if no group of same type is at sem_occupied[week][day][hour]:
4.       sem_occupied[week][day][hour] -= 1
5.   update_grade(-num_collisions)
```

*Pseudokod 16 - Untaksiranje obaveznog semestra*

```
1. untax_sem_optional(sem_occupied, rasp):
2.   for (week, day, hour) in rasp_rrule:
3.     if no group of same type is at sem_occupied[week][day][hour] AND
4.     (exactly 1 optional rasp is at sem_occupied[week][day][hour] OR
5.     group of different type is at sem_occupied[week][day][hour]):
6.       sem_occupied[week][day][hour] -= 1
7.   update_grade(-num_collisions)
```

*Pseudokod 17 - Untaksiranje izbornog semestra*

```
1. tax_capacity(room_id, rasp):
2.   if number of rasp students > room capacity:
3.     update_grade(penalty)
```

*Pseudokod 18 - Taksiranje kapaciteta dvorane*

```
1. untax_capacity(room_id, rasp):
2.   if number of rasp students > room capacity:
3.     update_grade(-penalty)
```

*Pseudokod 19 - Untaksiranje kapaciteta dvorane*

```
1. tax_computers(room_id, rasp):
2.   if rasp needs computers AND room doesn't have computers:
3.     update_grade(penalty)
```

*Pseudokod 20 - Taksiranje zadovoljenja računalnosti*

```
1. untax_computers(room_id, rasp):
2.   if rasp needs computers AND room doesn't have computers:
3.     update_grade(-penalty)
```

*Pseudokod 21 - Untaksiranje zadovoljenja računalnosti*

Nakon što su definirani algoritmi taksiranja i untaksiranja za dvorane, profesore, i semestre, moguće je definirati i pseudokod opće funkcije taksiranja i untaksiranja.

```
1. tax_new_slot(rasp, slot):
2.   tax_rooms(rooms_occupied, room_id, rasp)
3.   tax_profs(profs_occupied, prof_id, rasp)
4.   tax_sems(sems_occupied, rasp)
5.   tax_capacity(room_id, rasp)
6.   tax_computers(room_id, rasp)
```

*Pseudokod 22 - Taksiranje novog položaja*

```
1. untax_old_slot(rasp, slot):
2.   untax_rooms(rooms_occupied, room_id, rasp)
3.   untax_profs(profs_occupied, prof_id, rasp)
4.   untax_sems(sems_occupied, rasp)
5.   untax_capacity(room_id, rasp)
6.   untax_computers(room_id, rasp)
```

*Pseudokod 23 - Untaksiranje starog položaja*

**Ocjena**

Nakon taksing operacije, od broja kolizija je moguće formirati ocjenu. Primjerice, tako da se doda -30 svaki puta kada je element 3D polja veći od jedan. Manja ocjena u ovom slučaju znači gore rješenje. Pošto taksiramo odvojeno dvorane, profesore, semestre, kapacitet, i računalnost, ima smisla da i odvojeno pratimo njihove ocjene. Posljedično imamo ocjenu dvorane, profesora, semestra, kapaciteta, i računalnosti. Zbroj svih njih tvori ukupnu ocjenu.

Sada imamo sav potreban alat da pratimo i ažuriramo kolizije dvorana, profesora, semestara, kapaciteta, i računalnosti kada mijenjamo dvorane i početne datume raspova. Jedino što preostaje je optimizacijski algoritam koji će na pametan način odabrati rasp i pogodnu promjenu ovisno o situaciji u kojoj se nalazi da postigne što bolji raspored.

## 4.5. Optimizacijski algoritmi - implementacije

U ovom poglavlju bit će objašnjene implementacije algoritama iz trećeg poglavlja. Naglasak je na njihovoj konkretnoj primjeni na problemu akademskog rasporeda preko prethodno objašnjenog sustava.

### 4.5.1. Perturbacijski algoritmi

Perturbacijski algoritmi u ovom kontekstu se koriste za generiranje početnih rješenja optimizacijskih algoritama. U ovom slučaju to znači generiranje početnog rasporeda kojeg će trebati u daljnjim koracima optimizirati. Korištena su dva perturbacijska algoritma:

1. **Slučajno generiran raspored:** Svakom raspu se slučajno odabere položaj (*engl. slot*) iz skupa dopuštenih položaja.

```
1. random_timetable(rasps):
2.   timetable = {}
3.   for rasp in rasps:
4.     slot = random_slot(rasp)
5.     tax_new_slot(rasp, slot)
6.     timetable[rasp] = slot
```

*Pseudokod 24 - Slučajan raspored*

2. **Semi-slučajno generiran raspored:** Prvi put se generira slučajan raspored, a svaki idući put (nakon što je optimizacijski algoritam vratio rješenje) se odabere pet posto najproblematičnijih raspova i prvo se njima odabere položaj. Pokrene se lokalno traženje nad parcijalnim rasporedom i tek nakon toga preostali raspovi dobiju slučajan položaj.

```
1. semi_random_timetable(rasps):
2.   if first iteration:
3.     random_timetable(rasps)
4.   else:
5.     problematic_rasps = most_problematic(percent)
6.     timetable = random_timetable(problematic_rasps)
7.     local_search(timetable)
8.     leftover_rasps = rasps without problematic_rasps
9.     random_timetable(leftover_rasps)
```

*Pseudokod 25 - Semi slučajan raspored*

### 4.5.2. Jednostavno lokalno traženje

Započnimo od najjednostavnijeg algoritma koji ima vrlo široku primjenu: jednostavno lokalno traženje. Kao što je već objašnjeno u trećem poglavlju, algoritam jednostavnog lokalnog traženja kreće od inicijalnog rješenja i pronalazi lokalni optimum u skladu s definiranim susjedstvom.

Zanimaju nas četiri pojma kod problema rasporeda: način generiranja inicijalnog rješenja, definicija susjedstva rasporeda, način kretanja kroz susjede, i funkcija cilja.

**Generiranje inicijalnog rješenja:** Svodi se na odabir perturbacijskog algoritma, primjerice „slučajno generiran raspored“.

**Susjedstvo rasporeda:** Raspored je mapiranje od raspa do položaja. Problematičan rasp u rasporedu $X$ je onaj rasp koji ima barem jedno od navedenog: koliziju dvorane, profesora, semestra, prekoračen kapacitet, ili nezadovoljeno računalstvo. Susjedstvo od rasporeda $X$ je skup rasporeda $N(X)$ koji se dobiju tako što se nekom problematičnom raspu promijeni položaj.

**Način kretanja kroz susjede:** Slučajni odabir susjeda $X'$ iz $N(X)$. Time se odabire raspored $X'$, odnosno slučajan problematičan rasp od $X$ kojemu treba mijenjati položaj. Za rasp se odabire prvi položaj koji poboljšava ocjenu rasporeda (moguće je odabrati i položaj koji najviše poboljšava ocjenu, ali je to znatno skuplje). Ako ne postoji takav položaj, zabranjuje se ponovna posjeta tekućeg problematičnog raspa i pregledavaju se

ostali susjedi. Zabranjivanje se može implementirati uz pomoć tabu liste. Ako je algoritam prošao kroz sve susjede $X'$ iz $N(X)$ bez da je poboljšao ocjenu rasporeda, onda je raspored $X$ lokalni optimum.

**Funkcija cilja:** ocjena dobivena sumiranjem kolizija svih dvorana, profesora, semestara, prekoračenog kapaciteta, i prekršenih potreba računala u rasporedu.

```
1. local_search(timetable):
2.   while there are improving neighbors of timetable:
3.     problematic_rasp = random_neighbor(timetable)
4.     new_slot = first_better_slot(problematic_rasp)
5.     if not new_slot:
6.       tabu problematic_rasp
7.     else:
8.       tax_new_slot(problematic_rasp, new_slot)
```

*Pseudokod 26 - Jednostavno lokalno traženje (raspored)*

### 4.5.3. Ponavljano lokalno traženje

Algoritam lokalnog traženja često zapne u optimumu koji je lokalni, ali ne i globalni. Postoje razni načini borbe protiv tog svojstva, a jedan od njih je ponovno pokretanje algoritma s novim slučajno generiranim rješenjem. To se zove ponavljano lokalno traženje i sastoji se od tri koraka:

1. Slučajno generiranje novog početnog rasporeda (perturbacijski korak)
2. Apliciranje lokalnog traženja (optimizacijski korak)
3. Zadržavanje najboljeg rasporeda

```
1. repeated_local_search():
2.   best_timetable = {}
3.   while stopping condition is not met:
4.     timetable = random_timetable(rasps)
5.     local_search(timetable)
6.     update_timetable(timetable, best_timetable)
```

*Pseudokod 27 – Ponavljano lokalno traženje (raspored)*

### 4.5.4. Iterirano lokalno traženje

Iterirano lokalno traženje je slično ponavljanom lokalnom traženju u smislu da ponavljano vrši perturbaciju i lokalno traženje. Razlika je u tome što ponavljano lokalno traženje ne koristi znanje iz prethodnih iteracija da stvori novo početno rješenje u perturbacijskom koraku, dok iterirano lokalno traženje to znanje koristi.

Ako bi u prethodno definiranom algoritmu ponavljanog lokalnog traženja u perturbacijskom koraku umjesto „random_timetable“ postavili „semi_random_timetable“ (oba su objašnjenja u poglavlju 4.5.1), dobili bi iterirano lokalno traženje jer „semi_random_timetable“ koristi znanje iz prethodnih iteracija.

Algoritam se sastoji od tri koraka:
1. Generiranje novog početnog rasporeda koristeći znanje iz prethodne iteracije (perturbacijski korak)
2. Apliciranje lokalnog traženja (optimizacijski korak)
3. Zadržavanje najboljeg rasporeda

```
1. iterated_local_search():
2.   best_timetable = {}
3.   while stopping condition is not met:
4.     timetable = semi_random_timetable(rasps)
5.     local_search(timetable)
6.     update_timetable(timetable, best_timetable)
```

*Pseudokod 28 – Iterirano lokalno traženje (raspored)*

### 4.5.5. Pretraživanje promjenjivim susjedstvom

Drugi mogući način borbe protiv lokalnog optimuma je da se sustavno mijenja susjedstvo jednom kada algoritam dođe do lokalnog optimuma u tekućem susjedstvu. Taj algoritam se zove pretraživanje promjenjivim susjedstvom (*engl. variable neighborhood search*; skraćeno i dalje u tekstu: VNS). Postoji i algoritam lokalnog traženja „silazak promjenjivim susjedstvom“ (*engl. variable neighborhood descent*; skraćeno i dalje u tekstu: VND). Oba su objašnjena u poglavlju 3.3.

VNS se sastoji od četiri koraka [35]:

1. **Trešnja (shake):** Generiranje inicijalnog rješenja $X'$ iz skupa susjedstva $N_k(X)$ za $k = 1,..,k_{max}$.
2. **Lokalno traženje (VND):** Pronalazak najboljeg susjeda $X''$ od $X'$ iz susjedstva $N_l(X')$ za $l = 1,..,l_{max}$.
3. Ažuriranje brojeva $k$, i $l$ ovisno o rezultatima.
4. Ažuriranje najboljeg rješenja

Za problem rasporeda, zanimaju nas sljedeće informacije:

1. Definicija susjedstva $N_k(X)$ za $k = 1, \dots, k_{max}$ (za trešnju)
2. Definicija susjedstva $N_l(X')$ za $l = 1,..,l_{max}$ (za VND)

1. Definicija $k = 2$ susjedstva za funkciju trešnje:

- $N_1(X)$ je skup rasporeda koji se dobiju tako što se odabere slučajan rasp iz rasporeda $X$ te ga se postavi na slučajan položaj.
- $N_2(X)$ je skup rasporeda koji se dobiju tako što se odaberu dva slučajna raspa koji mogu zamijeniti položaje (u skladu sa svojim RRULE ograničenjima) i zatim im se zamijene položaji.

2. Definicija $l = 2$ susjedstva za VND funkciju:

- $N_1(X')$ je skup rasporeda koji se dobiju tako što se nekom problematičnom raspu iz rasporeda $X'$ promijeni položaj.
- $N_2(X')$ je skup rasporeda koji se dobiju tako što se odabere jedan problematičan rasp0 iz rasporeda $X'$, i jedan slučajan rasp1 iz rasporeda $X'$ (rasp0 != rasp1) takav da može zamijeniti položaj sa rasp0. Napomena: rasp1 ne mora biti problematičan.

VND će za problematičan rasp iz rasporeda $X'' \in N_1(X')$ tražiti prvi položaj koji poboljšava ocjenu. Ako takvog ne pronađe, zabranjuje ponovnu posjetu tekućeg problematičnog raspa. Ako svi problematični raspovi nemaju položaj koji im poboljšava ocjenu, algoritam je došao do lokalnog optimuma za susjedstvo $N_1(X')$ i $l$ se inkrementira za jedan, odnosno ide na iduće susjedstvo.

VND će za par (problematičan rasp0, rasp1) iz rasporeda $X'' \in N_2(X')$ prebaciti rasp0 na položaj od rasp1, a za rasp1 će tražiti prvi položaj na koji ga može prebaciti tako da poboljša ocjenu. Ako pronađe takav položaj, $l$ se postavlja na jedan, odnosno kreće opet iz prvog susjedstva. Ako se takav položaj ne pronađe za sve elemente iz $N_2(X')$, algoritam je došao do lokalnog optimuma za susjedstvo $N_2(X')$ i $l$ se inkrementira za jedan, odnosno ide na iduće susjedstvo. Pošto $N_3(X')$ nije definiran, VND staje i vraća raspored koji je optimizirao. Ažurira se najbolji raspored i pokreće nova iteracija trešnje i VND-a.

```
1. variable_neighborhood_search():
2.   k = 0, kmax = 2
3.   best_timetable = {}
4.   while k <= kmax:
5.     shake(timetable, k)
6.     variable_neighborhood_descent(timetable)
7.     if grade was improved:
8.       k = 1
9.       update_timetable(timetable, best_timetable)
10.    else:
11.      k += 1
```

*Pseudokod 29 – Pretraživanje promjenjivom okolinom (raspored)*

```
1. shake(timetable, k):
2.   if k==1:
3.     rasp0 = random_rasp(timetable)
4.     slot  = random_slot(rasp0)
5.     tax_new_slot(rasp0, slot)
6.     timetable[rasp0] = slot
7.   else if k==2:
8.     rasp0 = random_rasp(timetable)
9.     rasp1 = random_swappable_rasp(timetable, rasp0)
10.    rasp0_new_slot = timetable[rasp1]
11.    rasp1_new_slot = timetable[rasp0]
12.    swap_slots(rasp0, rasp1, rasp0_new_slot, rasp1_new_slot)
```

*Pseudokod 30 - VNS - Trešnja (raspored)*

```
1. variable_neighborhood_descent(timetable, l, lmax):
2.   l = 0, lmax = 2
3.   while l <= lmax:
4.     if l == 1:
5.       while there are problematic rasps:
6.         rasp0    = random_rasp(timetable)
7.         new_slot = first_better_slot(rasp0)
8.         if not new_slot:
9.           tabu rasp0
10.        else:
11.          tax_new_slot(rasp0, slot)
12.          timetable[rasp0] = slot
13.          break
14.    else if l == 2:
15.      while there are problematic rasps:
16.        rasp0 = random_rasp(timetable)
17.        rasp1 = random_swappable_rasp(timetable, rasp0)
18.        if not rasp1:
19.          tabu rasp0
20.          continue
21.        rasp0_new_slot = timetable[rasp1]
22.        rasp1_new_slot = first_better_slot(rasp1)
23.        if not rasp1_new_slot:
24.          tabu rasp1 for rasp0
25.        else:
26.          swap_slots(rasp0, rasp1, rasp0_new_slot, rasp1_new_slot)
27.          break
28.    l = 1 if grade was improved else l+1
```

*Pseudokod 31 – VND (raspored)*

### 4.5.6. Simulirano kaljenje

Simulirano kaljenje [33] je tehnika lokalnog traženja koja prihvaća gore rješenje uz određenu vjerojatnost koja se mijenja s vremenom. S time nastoji istražiti više različitih rješenja u nadi da će se približiti globalnom optimumu. Jednostavno lokalno traženje nikad ne prihvaća gore rješenje, već se jednostavno spusti do lokalnog optimuma po boljim rješenjima. To je temeljna razlika između njih.

Pošto je simulirano kaljenje zapravo varijanta lokalnog traženja, i dalje nas zanimaju četiri pojma koja su nas zanimala kod jednostavnog lokalnog traženja u poglavlju 4.5.2: generiranje inicijalnog rješenja, definicija susjedstva rasporeda, način kretanja kroz susjede, i funkcija cilja.

Uz to nas zanimaju i dodatni parametri koju su specifični za simulirano kaljenje:

- Temperatura
- Funkcija temperature (smanjivanje temperature kroz vrijeme)
- Vjerojatnosna funkcija (za prihvaćanje rješenja)

**Temperatura:** obično $10^4$ do $10^9$, ali nju treba mijenjati ovisno o problemu
**Funkcija temperature:** $T = T * 0.99$
**Vjerojatnosna funkcija prihvaćanja:** Metropolis-Hastings algoritam

**Generiranje inicijalnog rješenja:** Svodi se na odabir perturbacijskog algoritma, primjerice „slučajno generiran raspored“.

**Susjedstvo rasporeda:** skup rasporeda $N(X)$ koji se dobiju tako što se nekom problematičnom raspu iz rasporeda $X$ promijeni položaj.

**Način kretanja kroz susjede:** Slučajni odabir susjeda $X' \in N(X)$. Time se odabire raspored $X'$, odnosno slučajan problematičan rasp od $X$ kojemu treba mijenjati položaj. Za rasp se iterira kroz položaje i aplicira vjerojatnosna funkcija prihvaćanja. U slučaju da položaj poboljšava ocjenu, uvijek se prihvati. U slučaju da ne poboljšava ocjenu, prihvaćen je ili odbijen uz određenu vjerojatnost. Ako se ne prihvati niti jedan položaj, zabranjuje se ponovna posjeta tekućeg problematičnog raspa. Ako je algoritam prošao kroz sve problematične raspove bez da je barem jednome prihvatio novi položaj, onda je algoritam zapeo u lokalnom optimumu.

**Funkcija cilja:** ocjena dobivena sumiranjem kolizija svih dvorana, profesora, semestara, prekoračenog kapaciteta, i prekršenih potreba računala u rasporedu.

```
1. // Metropolis-Hastings
2. P(old_grade, new_grade, temperature):
3.   if new_grade < old_grade:
4.     return e^((old_grade-new_grade) / temperature)
5.   else:
6.     return 1.0
```

*Pseudokod 32 - Vjerojatnosna funkcija za simulirano kaljenje (raspored)*

```
1. simulated_annealing(timetable):
2.   while there are neighbors of timetable:
3.     problematic_rasp = random_neighbor(timetable)
4.     new_slot = first_accepted_slot(problematic_rasp, P)
5.     if not new_slot:
6.       tabu problematic_rasp
7.     else:
8.       tax_new_slot(problematic_rasp, new_slot)
9.       timetable[problematic_rasp] = new_slot
```

*Pseudokod 33 – Simulirano kaljenje (raspored)*

### 4.5.7. Pohlepno nasumično adaptivno pretraživanje

Pohlepno nasumično adaptivno pretraživanje (*engl. greedy randomized adaptive search procedure*, skraćeno i dalje u tekstu: GRASP) je algoritam koji nastoji kreirati kvalitetnija inicijalna rješenja koristeći mješavinu pohlepnog i slučajnog pristupa.

Kao što je objašnjeno u poglavlju 3.4., GRASP se odvija kroz dvije faze: konstrukcija rješenja i lokalno traženje. Zanima nas kako će se te dvije faze implementirati na problemu rasporeda.

**Konstrukcija rješenja:** Rješenje se konstruira od praznog rasporeda, rasp po rasp. U svakoj iteraciji se odabire jedan rasp za kojega se izrađuje lista kandidata. Lista kandidata se sastoji od položaja za promatrani rasp i ocjene koja označava jačinu kršenja ograničenja ako bi taj položaj bio odabran. Izračunavanje ocjene položaja za promatrani rasp na temelju jačine kršenja ograničenja predstavlja pohlepni dio ove faze. Veličina liste kandidata ovisit će o „num_candidates“ parametru. Jednom kada je lista kandidata napravljena, sortira se po ocjeni položaja tako da najbolje ocijenjeni budu prvi. Izrađuje se ograničenja lista kandidata tako da se iz sortirane liste kandidata zadrži

samo prvih „num_restrict“ elemenata („num_restrict“ je parametar). Iz ograničene liste kandidata se odabire jedan slučajan položaj koji postaje odabrani položaj za promatrani rasp. Odabir slučajnog položaja iz ograničene liste kandidata predstavlja slučajni dio faze. Iteracija prelazi na idući rasp gdje proces kreće iz početka.

**Lokalno traženje:** Na konstruirano rješenje se aplicira proizvoljni algoritam lokalnog traženja. Primjerice, jednostavno lokalno traženje iz poglavlja 4.5.2.

```
1. grasp(rasps):
2.   best_timetable = {}
3.   while stopping condition is not met:
4.     timetable = construct_solution(rasps, num_candidates, num_restrict)
5.     timetable = local_search(timetable)
6.     update_timetable(timetable, best_timetable)
```

*Pseudokod 34 - GRASP (raspored)*

```
1. construct_solution(rasps, num_candidates, num_restrict):
2.   timetable = {}
3.   for rasp in rasps:
4.     rcl = make_rcl(rasp, num_candidates, num_restrict)
5.     slot = select_random_element(rasp, rcl)
6.     timetable[rasp] = slot
7.   return timetable
```

*Pseudokod 35 - GRASP - Konstruiranje rješenja (raspored)*

```
1. make_rcl(rasp, num_candidates, num_restrict):
2.   candidate_list = apply_greedy(rasp, num_candidates)
3.   candidate_list = sort_by_grade(candidate_list)
4.   restricted_candidate_list = candidate_list[:num_restrict]
5.   return restricted_candidate_list
```

*Pseudokod 36 - GRASP - Izrada RCL-a (raspored)*

```
1. apply_greedy(rasp, num_candidates):
2.   candidate_list = []
3.   slot_pool = get_slot_pool(rasp)
4.   slot_pool = slot_pool[:num_candidates]
5.   for slot in slot_pool:
6.     grade = grade_slot(rasp, slot)
7.     candidate_list.add((grade, slot))
8.   return candidate_list
```

*Pseudokod 37 - GRASP - Apliciranje pohlepe (raspored)*

```
1. select_random_element(rasp, rcl):
2.   slot = random_slot(rcl)
3.   tax_new_slot(rasp, slot)
```

*Pseudokod 38 - GRASP - Odabir slučajnog elementa (raspored)*

# 5. Evaluacija

Evaluacija je provedena na operacijskom sustavu Arch Linux x86_64 u programskom jeziku C++ na AMD Ryzen 5 4500U procesoru. Postoji 12 skupova podataka od kojih se svaki može kategorizirati u jednu od tri kategorije: S (*small*), M (*medium*), i L (*large*). Skupovi podataka su slučajno generirani u skladu sa svojom kategorijom.

| | S | M | L |
|---|---|---|---|
| **Broj raspova** | 300 | 750 | 1500 |
| **Broj rač. raspova** | 150 | 300 | 750 |
| **Broj dvorana** | 20 | 30 | 50 |
| **Broj rač. dvorana** | 10 | 15 | 25 |
| **Broj profesora** | 25 | 40 | 60 |
| **Broj semestara** | 25 | 40 | 60 |

*Tablica 14 – Kategorije skupova podataka*

Kategorije primarno razdvaja jačina stegnutosti koja je prisutna, odnosno omjer broja raspova s brojem dvorana, profesora, semestara i sl. Što više raspova na manje dvorana, profesora, i semestara, to će borba s kolizijama biti teža. Primijetite da kategorija „S“ ima 300 raspova na 20 dvorana, dok kategorija „L“ ima 1500 raspova (5 puta više) na 50 dvorana (samo 2.5 puta više).

Podaci o profesorima su uniformno raspoređeni da predaju raspove, a ako ih nema dovoljno, onda ciklički: nakon što je zadnji profesor iskorišten, kreće se od prvog i on predaje idući rasp, i tako dalje. Na isti način su raspoređeni i semestri. To implicira sljedeće: što manje ima profesora i semestara, to će više raspova biti smješteno na iste profesore i semestre i borba s kolizijama će biti teža.

Od 12 generiranih skupova podataka, po četiri su u svakoj kategoriji. Kategorija „S“ ima skupove podataka „S1“, „S2“, „S3“, i „S4“, kategorija „M“ ima „M1“, „M2“, „M3“, i „M4“, i kategorija „L“ ima „L1“, „L2“, „L3“, i „L4“. Skupovi podataka poštuju svojstva svoje kategorije, ali se i dalje međusobno razlikuju u drugim (slučajno generiranim) svojstvima.

Cilj je provesti 12 skupova podataka kroz pet algoritama i vidjeti završni broj prekršenih ograničenja za svaki od njih. Cilj nije striktno doći do stanja od nula kolizija jer zbog prirode ograničenja nije garantirano da je to moguće. Teška ograničenja su namjerno postavljena da bi vidjeli koji algoritmi će ih najviše minimizirati. Također, kao što je već objašnjeno, kategorije primarno predstavljaju jačinu stegnutosti, a ne veličinu ulaza. Moguće je nakon 10 sekundi doći do stanja od nula kolizija za 1500 raspova i ne doći nakon 30 minuta do stanja od nula kolizija za 150 raspova. Sve ovisi o ograničenjima.

Svaki od pet algoritama je radio 30 minuta na svakom od 12 skupova podataka. To je 30 sati ukupnog računanja. Kratice imena algoritama znače sljedeće:

- VNS je *variable neighborhood search* (poglavlje 4.5.5.)
- GRASP je *greedy randomized adaptive search procedure* (poglavlje 4.5.7.)
- RLS je *repeated local search* (poglavlje 4.5.3.)
- ILS je *iterated local search* (poglavlje 4.5.4.)
- SA je *simulated annealing* (poglavlje 4.5.6.)

Brojevi prekršenih ograničenja nakon 30 minuta računanja su prikazani u tablici 15.

| | VNS | GRASP | RLS | ILS | SA |
|---|---|---|---|---|---|
| **S1** | 0 | 0 | 0 | 0 | 0 |
| **S2** | 0 | 0 | 0 | 0 | 0 |
| **S3** | 0 | 0 | 0 | 0 | 0 |
| **S4** | 0 | 0 | 0 | 0 | 0 |
| **M1** | **20** | 50 | 37 | 38 | 31 |
| **M2** | **54** | 87 | 93 | 102 | 79 |
| **M3** | **50** | 82 | 66 | 78 | 75 |
| **M4** | **31** | 79 | 54 | 64 | 60 |
| **L1** | **565** | 658 | 628 | 600 | 597 |
| **L2** | **698** | 1044 | 904 | 939 | 849 |
| **L3** | **750** | 1056 | 969 | 843 | 934 |
| **L4** | **682** | 861 | 784 | 736 | 738 |

*Tablica 15 – Broj prekršenih ograničenja: pet algoritama*

VNS je konzistentno ostvario najbolje rezultate, najvjerojatnije zbog toga što pretražuje više susjedstva odjednom, a ostali algoritmi ne. Ostali algoritmi su varirali ovisno o skupu podataka. Zanimljivo je da je RLS (slučajan ulaz) nerijetko imao bolje performanse od ILS-a koji koristi znanje iz prethodne iteracije da generira ulaz. Jedan od razloga za to može biti taj što ILS uzorkuje ulaze iz distribucije koja nije dovoljno kvalitetna. GRASP je imao najgore performanse, što može značiti da i on uzorkuje ulaze iz nedovoljno kvalitetne distribucije. Oba (ILS i GRASP) su sumnjivi jer pohlepno sužavaju distribuciju ulaza, dok je RLS (koji ne sužava distribuciju) nerijetko imao bolje performanse od njih. Performanse SA algoritma su varirale od jednog skupa podataka do drugog, bio je gori od VNS-a, ali je bio među boljima od preostalih algoritama.

Uzeti ćemo algoritam s najboljim rezultatima (VNS) i izmijeniti mu perturbacijski algoritam. Prijašnji VNS je imao „random“ perturbacijski algoritam koji bi svaki put kao ulazni raspored poslao slučajni kada bi zapeo u lokalnom optimumu. Novi perturbacijski algoritam će biti „semi-random“ koji će kao ulazni raspored poslati onaj kojemu je prvo postavljeno 20% najproblematičnijih raspova iz prijašnje iteracije, a ostalih 80% slučajno. Oba perturbacijska algoritma su objašnjena u poglavlju 4.5.1. Brojevi prekršenih ograničenja nakon 30 minuta računanja su prikazani u tablici 16.

| | VNS random | VNS semi-random |
|---|---|---|
| **S1** | 0 | 0 |
| **S2** | 0 | 0 |
| **S3** | 0 | 0 |
| **S4** | 0 | 0 |
| **M1** | 20 | **18** |
| **M2** | 54 | **52** |
| **M3** | 50 | **43** |
| **M4** | 31 | **26** |
| **L1** | 565 | **510** |
| **L2** | **698** | 782 |
| **L3** | **750** | 773 |
| **L4** | 682 | **592** |

*Tablica 16 – Broj prekršenih ograničenja: VNS random i semi-random ulaz*

Zanimljivo je da je u većini slučajeva „semi-random“ VNS ostvario bolje rezultate. To može biti jer VNS zbog svoje prirode već dovoljno široko pretražuje rješenja i odgovara

mu kada se najproblematičniji raspovi smjeste prvi.

Uzeti ćemo algoritam simuliranog kaljenja (SA) i pokrenuti ga tri puta, jedan put s temperaturom 10000, jedan s 100000, i jedan s 1000000. Brojevi prekršenih ograničenja nakon 30 minuta računanja su prikazani u tablici 17.

| | SA – 10000 | SA – 100000 | SA – 1000000 |
|---|---|---|---|
| **S1** | 0 | 0 | 0 |
| **S2** | 0 | 0 | 0 |
| **S3** | 0 | 0 | 0 |
| **S4** | 0 | 0 | 0 |
| **M1** | **31** | **31** | 32 |
| **M2** | 76 | 79 | **73** |
| **M3** | **61** | 75 | 71 |
| **M4** | 53 | 60 | **46** |
| **L1** | 623 | **597** | 598 |
| **L2** | 963 | **849** | 905 |
| **L3** | 1000 | **934** | 956 |
| **L4** | 777 | **738** | 781 |

*Tablica 17 – Broj prekršenih ograničenja: simulirano kaljenje – temperature*

Temperatura 100000 je imala najbolje performanse na najjačim ograničenjima (kategorija L). U kategoriji M su ostale temperature (10000 ili 1000000) imale bolje ili iste performanse. Iz toga je moguće zaključiti da je potrebno eksperimentirati s temperaturom ovisno o skupu podataka.

# Zaključak

Problem optimizacije akademskog rasporeda gdje treba smjestiti predmete na vremenske položaje u dvorane pazeći na proizvoljna ograničenja i pritom optimizirati vrijednost funkcije cilja je NP-težak [1]. Ne postoji algoritam koji u polinomnom vremenu daje optimalna rješenja, a ulaz za tipičnu situaciju je prevelik da bi se problem riješio provjeravanjem svih kombinacija. Zato se pribjegava korištenju heurističkih algoritama koji daju zadovoljavajuća, ali ne nužno i optimalna rješenja.

U radu je predstavljen sustav uz pomoć kojega je moguće rješavati problem rasporeda temeljenog na ponavljajućim pravilima. Sastoji se od definicije ulaza, odnosno strukture koja opisuje kako spremiti podatke, i od optimizacijskih algoritama koji generiraju i poboljšavaju rješenje. Fokus kod optimizacijskih algoritama je stavljen na lokalno traženje i njegove varijacije, specifično: ponavljano lokalno traženje, iterirano lokalno traženje, pretraživanje promjenjivim susjedstvom, i simulirano kaljenje. Također je implementiran i algoritam pohlepnog nasumičnog adaptivnog pretraživanja. Svi oni kao cilj imaju boriti se protiv lokalnog optimuma koje je temeljno svojstvo običnog lokalnog traženja i svaki tu borbu sprovodi na svoj način što ih čini korisnima u različitim situacijama. Na kraju je provedena evaluacija gdje su uspoređene njihove performanse. Pretraživanje promjenjivim susjedstvom je ostvarilo najbolje rezultate što ga čini dobrim izborom za ovaj tip problema.

Za daljnji rad je moguće nadograditi ulaz, odnosno prihvatiti neku drugu strukturu ulaznih podataka koja bi bila pogodnija za druge slučajeve. Moguće je implementirati još optimizacijskih algoritama kao što su genetski algoritam, logičko programiranje, strojno učenje, i sl. Također, problem rasporeda je moguće podijeliti na više dijelova i svaki dio riješiti zasebnim algoritmom.

Implementacija sustava objašnjenog u ovom radu može se pronaći na sljedećoj poveznici: https://github.com/jjurinci/scheduler_system

# Literatura

[1] Sapru, V., Reddy, K. i Sivaselvan B. (2010) *Timetabling Using Genetic Algorithms Employing Guided Mutation*, IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), 1-4.

[2] Aljarrah, M. A., Alsawalqah, A. A. i Hamdan, S. F. (2017) *Developing A Course Timetable System for Academic Departments Using Genetic Algorithm*, Jordanian Journal of Computers and Information Technology, 3(1), 25-36.

[3] Awad, F. H., Al-Kubaisi, A. i Mahmood, M. (2022) *Large-scale timetabling problems with adaptive tabu search*, Journal of Intelligent Systems. 31(1), 168-176.

[4] Daskalaki, S., Birbas, T. i Housos, E. (2004) *An integer programming formulation for a case study in university timetabling*, European Journal of Operational Research. 153(1), 117-135.

[5] Perera, M.T.M. i Lanel, G.H.J. (2016) *A Model to Optimize University Course Timetable Using Graph Coloring and Integer Linear Programming*, IOSR Journal of Mathematics, 12(5), 13-18.

[6] Abdelhalim, E. A. i El Khayat, G. A. (2016) *A Utilization-based Genetic Algorithm for Solving the University Timetabling Problem (UGA)*, Alexandria Engineering Journal. 55(2), 1395-14.

[7] Mudjihartono, P. (2014) *Academic Timetable Generation Using Abandoned and Reborn Solution Mechanism of Particle Swarm Optimization.*

[8] Jensen, T. R. i Toft, B. (1995) Introduction to Graph Coloring: Basic Definitions. U: *Graph Coloring Problems*, New York: John Wiley & Sons, 1-3.

[9] Garey, M. R. i Johnson, D. S. (1979) NP-hardness. U: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York: W.H. Freeman, 109-118.

[10] Ganguli, R. i Roy, S. (2017) *A Study on Course Timetable Scheduling using Graph Coloring Approach*, International Journal of Computational and Applied Mathematics, 12(2), 469-485.

[11] Mandal, A. K. (2020) *Development of an Interactive Tool based on Combining Graph Heuristic with Local Search for Examination Timetable Problem*, International

Journal of Advanced Computer Science and Applications, 11(3).
[12] Chen, D., Batson, R. G., i Dang, Y. (2010) Modeling: Introduction: Integer Programming. U: *Applied Integer Programming: modeling and solution*. Hoboken, New Jersey: John Wiley & Sons, 3-5.
[13] Conforti, M., Cornuejols, G. i Zambelli, G. (2014) Getting Started: Complexity: Polynomial Algorithm. U: *Integer Programming*, London: Springer, 18-19.
[14] Conforti, M., Cornuejols, G. i Zambelli, G. (2014) Getting Started: Complexity: Complexity Class NP. U: *Integer Programming*, London: Springer, 19-20.
[15] Tsang, E. (1996) Introduction: Formal Definition of the CSP. U: *Foundations of Constraint Satisfaction*, London: Academic Press Limited, 5-10.
[16] Tsang, E. (1996) CSP solving – an overview: Characteristics of Individual CSPs. U: *Foundations of Constraint Satisfaction*, London: Academic Press Limited, 46-53.
[17] Ruttkay, Z. (1998) *Constraint satisfaction a survey*, CWI Quarterly, 11(1), 163–214.
[18] El-Sakka, T. (2015) *University Course Timetable using Constraint Satisfaction and Optimization*, International Journal of Computing Academic Research, 4(3), 83-95.
[19] Ďuriš, V. i Taylor, G. (2020) *Algorithmic Verification of Constraint Satisfaction Method on Timetable Problem*, Journal of Mathematics and Statistics, 8(6), 728-739.
[20] Katoch, S., Chauhan, S.S. i Kumar, V. (2021) *A review on genetic algorithm: past, present, and future*, Multimed Tools Appl. 80(5), 8091–8126.
[21] Kumar, M., Husain, M., Upreti i N., Gupta, D. (2010) *Genetic algorithm: Review and application*, Journal of Information & Knowledge Management, 2(1), 451-454.
[22] Michiels, W., Aarts E. H. L. i Korst, J. (2007) Introduction: Basics of Local Search. U: *Theoretical Aspects of Local Search*, Berlin: Springer, 3-9.
[23] Michiels, W., Aarts E. H. L. i Korst, J. (2007) Basic Examples. U: *Theoretical Aspects of Local Search*, Berlin: Springer, 11-36.
[24] Schuurmans, D. i Southey, F. (2001) *Local search characteristics of in- complete SAT procedures*. Artificial Intelligence, 132(2), 121–150.
[25] Bendi, R. K. J. i Junaidi, H. (2019) *Simulated Annealing Approach for University Timetable Problem*, Jurnal Ilmiah Matrik, 21(3).
[26] Parsopoulos, K. E. i Montes de Oca M. A. (2017) *Swarm-Based Optimization Algorithms*, Wiley Encyclopedia of Electrical and Electronics Engineering, J.G. Webster

(Ed.)
[27] Socha, K., Knowles J. i Sampels, M. (2002) *A MAX-MIN Ant System for the University Course Timetabling Problem*, International Workshop on Ant Algorithms. Berlin: Springer, 1-13.
[28] Zhang, Y., Wang, S. i Ji, G. (2015) *A Comprehensive Survey on Particle Swarm Optimization Algorithm and Its Applications*, Mathematical Problems in Engineering. 2015(1), 1-38.
[29] Fahmy, S., Ruhaida, I., Akhyari, N. i Nazri, I. (2014) *Open-Source Academic Timetabling System: FET Implementation at TATI University College*. Australian Journal of Basic and Applied Sciences, 8(4), 86-91.
[30] Ilić, M., Spalević, P., Ilić, S., Veinović, M., Milivojević i Z., Prlinčević, B. (2015) *Data mining techniques for student timetable optimization*, Infoteh-Jahorina, 14(1).
[31] Russell, S. J., Norvig, P., i Davis, E. (2010) Beyond Classical Search: Local Search Algorithms and Optimization Problems. U: *Artificial Intelligence: A Modern Approach* 3rd ed, New Jersey: Pearson, 120-129.
[32] Skiena, S. S. (2008) Combinatorial Search and Heuristic Methods: Heuristic Search Methods. U: *The Algorithm Design Manual* 2nd ed, London: Springer, 247-260.
[33] Bertsimas, D. i Tsitsiklis, J. (1993) *Simulated Annealing*, Statistical Science, 8(1), 10-15.
[34] Laarhoven, P. J. M., i Aarts, E. H. L. (1987) Simulated Annealing: Introduction of the algorithm. U: *Simulated annealing: Theory and applications*, Nizozemska: Springer, 7-12.
[35] Hansen, P. i Mladenović, N. (2003) *A tutorial on variable neighborhood Search*, Groupe d'études et de recherche en analyse des décisions, HEC Montréal.
[36] Feo, T. A. i Resende, M. G. C. (1995) *Greedy Randomized Adaptive Search Procedures*, Journal of Global Optimization, 6(2), 109-133.
[37] Gendreau, M., i Potvin, J. (2019) Iterated Local Search: Framework and Applications. U: *Handbook of metaheuristics* 3rd ed, Švicarska: Springer, 129-169.
[38] Gendreau, M., i Potvin, J. (2019) Computational Comparison of Metaheuristics. U: *Handbook of metaheuristics* 3rd ed, Švicarska: Springer, 581-601.
[39] Desruisseaux, B. (2009) *Internet Calendaring and Scheduling Core Object*

*Specification (iCalendar): RFC 5545*, Network Working Group. Dostupno na: https://www.rfc-editor.org/rfc/rfc5545 [11. rujna 2022.]

[40] rrule – dateutil 2.8.2. documentation. (2021) Dostupno na: https://dateutil.readthedocs.io/en/stable/rrule.html [11. rujna 2022.]

# Popis slika

# Popis tablica

# Popis pseudokoda

# Sažetak

Optimizacija akademskog rasporeda je NP-težak problem koji se često pojavljuje u praksi. Zbog nepostojanja boljih algoritama, obično se koriste heuristički algoritmi koji daju zadovoljavajuća rješenja za praktične potrebe. Različiti autori problem različito definiraju zbog proizvoljnosti ograničenja koja njihova sveučilišta imaju. U ovom radu je dana jedna moguća definicija koja omogućuje da svaka grupa predmeta ima ponavljajuće pravilo po kojemu će se vremenski održavati uz optimizaciju kršenja ograničenja kao što su kolizije vremena dvorana, profesora, semestara, kapaciteta, računala, i sl. Za ponavljajuća pravila se koristi RRULE iz specifikacije iCalendar RFC 5545. U praktičnom dijelu ulaz je analiziran, rastavljen na dijelove i pretvoren u rješenje uz pomoć optimizacijskih algoritama. Specifično, predstavljeni su sljedeći algoritmi: ponavljano lokalno traženje, iterirano lokalno traženje, pretraživanje promjenjivim susjedstvom, simulirano kaljenje, i pohlepno nasumično adaptivno pretraživanje. Algoritmi su na kraju evaluirani na različitim skupovima podataka gdje su im uspoređene performanse.

**Ključne riječi:** problem akademskog rasporeda, ponavljajuća pravila, RRULE, lokalno traženje

# Abstract

Academic timetable optimization is an NP-hard problem that often appears in practice. In the absence of better algorithms, heuristic algorithms are usually used, which provide satisfactory solutions for practical needs. Different authors define the problem differently due to the arbitrariness of the restrictions that their universities have. In this thesis, one possible definition is given that allows each group of subjects to have a recurring rule according to which it will be maintained in time while optimizing the violation of restrictions such as collisions of the time of rooms, professors, semesters, capacity, computers, etc. For recurring rules, RRULE from the iCalendar RFC 5545 specification is used. In the practical part, the input is analyzed, divided into parts, and turned into a solution with the help of optimization algorithms. Specifically, the following algorithms are presented: repeated local search, iterated local search, variable neighborhood search, simulated annealing, and greedy randomized adaptive search procedure. Algorithms were finally evaluated on different datasets where their performance was compared.



Keywords: academic timetable problem, recurring rules, RRULES, local search