Sustav za optimizaciju proizvodnje i upravljanje proizvodnim procesom

Vugrinec, Matija

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:815347

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-18**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 343

SUSTAV ZA OPTIMIZACIJU PROIZVODNJE I UPRAVLJANJE PROIZVODNIM PROCESOM

Matija Vugrinec

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 343

SUSTAV ZA OPTIMIZACIJU PROIZVODNJE I UPRAVLJANJE PROIZVODNIM PROCESOM

Matija Vugrinec

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 343

Pristupnik:	**Matija Vugrinec (0036525885)**
Studij:	Računarstvo
Profil:	Programsko inženjerstvo i informacijski sustavi
Mentorica:	prof. dr. sc. Ljiljana Brkić

Zadatak: **Sustav za optimizaciju proizvodnje i upravljanje proizvodnim procesom**

Opis zadatka:

Optimizacija proizvodnog procesa ključna je za poboljšanje efikasnosti, smanjenje troškova i povećanje profitabilnosti. Odnosi se na širok spektar tema, uključujući optimizaciju korištenja resursa (ljudstvo, strojevi, materijali i energija), optimizaciju logistike i lanca opskrbe (transport, skladištenje i distribucija proizvoda) ili izračunavanje optimalne količine proizvoda koju treba proizvesti u jednoj seriji kako bi se minimizirali troškovi i maksimizirala profitabilnost. Potrebno je istražiti različite metode optimizacije proizvodnje u kontekstu industrijskih postrojenja. Cilj je istražiti kako različiti algoritmi i tehnike mogu biti primijenjeni za poboljšanje efikasnosti, smanjenje troškova i povećanje konkurentnosti proizvodnog procesa. Na temelju provedenog istraživanja, potrebno je primijeniti odabrani algoritam na optimizaciju rasporeda proizvodnje raznovrsnih proizvoda koji za proizvodnju dijele strojeve i sirovine. Pri tom je potrebno postići da sirovine za proizvodnju proizvoda kao i proizvedeni proizvodi što kraće borave na skladištu. Potrebno je izraditi programsku potporu koja će korisniku omogućiti upravljanje podacima u svim fazama proizvodnog procesa počevši od evidencije kataloških podataka (sirovine, proizvodi, strojevi, skladišna mjesta, dobavljači, kupci, itd.) do evidencije narudžbi kupaca te pokretanje postupka optimizacije i evidenciju rezultata optimizacije. Donijeti ocjenu ostvarenog optimizacijskog postupka, navesti glavne izazove u radu te smjernice za budući razvoj.

Rok za predaju rada: 28. lipnja 2024.

Sadržaj

Uvod 1

1. Opis problema 3

1.1. Opis predviđenog sustava 3

1.1.1. Upravljanje korisnicima 3

1.1.2. Upravljanje skladištima 3

1.1.3. Upravljanje podacima o proizvodima 4

1.1.4. Upravljanje narudžbama 4

1.1.5. Upravljanje radnim nalozima 4

1.2. Optimizacijski algoritmi 4

1.2.1. Osnovni pojmovi optimizacijskih algoritama 4

1.2.2. Neki optimizacijski algoritmi 5

2. Arhitektura sustava i tehnologije 7

2.1. Tehnologije implementacije 7

2.2. Arhitektura sustava 7

3. Implementacija sustava 8

3.1. Implementacija upravljanja podacima 8

3.1.1. Straničenje, sortiranje i filtriranje tabličnih podataka 8

3.1.2. Dodavanje i uređivanje zapisa tabličnih podataka 11

3.1.3. Grafičko uređivanje skladišnih mjesta 12

4. Optimizacijski algoritam za raspoređivanje radnih naloga 19

4.1. Multistart local search 22

4.1.1. Generiranje nezavisnih inicijalnih rješenja 23

4.1.2. Lokalna pretraga 28

4.1.3. Funkcija dobrote 32

4.2. Simulirano kaljenje 33
4.2.1. Generiranje slučajnog susjeda 36
4.2.2. Određivanje parametara algoritma za simulirano kaljenje 37
4.3. Prikaz rješenja na sučelju 40
4.4. Testiranje 41
Zaključak 44
Literatura 45
Sažetak 46
Summary 47

Uvod

Proizvodni procesi u tvrtkama obuhvaćaju brojne elemente kao što su korištenje resursa poput ljudi i strojeva, logistiku, upravljanje lancem opskrbe te koordinaciju različitih odjela ili vanjskih partnera kako bi se ostvarila efikasna i usklađena proizvodnja. Zbog kompleksnosti, mnoge tvrtke se oslanjaju na modernu programsku podršku koja im olakšava rad. Jedan od ključnih elemenata moderne informatičke podrške je i optimizacija rasporeda. Raspoređivati se može raspored proizvodnje, raspored skladištenja, raspored otpreme i mnogi drugi rasporedi. Optimizacija rasterećuje ljude zamornog posla kombinatorike i omogućava veću kvalitetu rasporeda. Cilj ovog rada je prikazati kako jedna takva optimizacija rasporeda proizvodnje olakšava rad proizvodne tvrtke.

Tijekom života susrećemo se s mnogim zadacima koje moramo obaviti. Ti zadaci mogu biti jednostavni kao prelazak ceste. U takvim jednostavnim zadacima, intuicija nam govori da je ravna linija najbrži put i ne moramo puno razmišljati koji put odabrati. Takvi problemi se često sastoje od samo par mogućih odabira i za njih ljudi ne trebaju pomoć računala i optimizacijskih algoritama. Pogledajmo sada i jedan teži problem koji se često javlja u akademskoj zajednici. Riječ je o problemu trgovačkog putnika (Hall, 2012). On je definiran kao postupak za pronalazak redoslijeda kojim trgovački putnik posjećuje gradove s tim da ne posjeti niti jedan grad više od 1 puta, prijeđe najmanje kilometara odnosno uzme najkraći mogući put i na kraju se vrati u početni grad. Recimo da imamo farmaceuta koji prodaje lijekove i 80000 ljekarna u SAD-u koje on mora obilaziti. Na temelju intuicije vidimo da naš mozak ne bi mogao napraviti taj raspored na smislen način zbog prevelikog broja izbora.

Sljedeća opcija koja bi nam mogla pasti na pamet je izračunati put za svaki izbor i uzeti opciju s najmanjim troškom. Takav postupak se zove iscrpna pretraga grubom silom (White & Yen, 2004). Ovaj problem kod pretrage grubom silom ima kompleksnost O(N!). gdje je N broj gradova. Faktorijelna složenost proizlazi iz toga da svaki redoslijed gradova proglasimo kao jednu kombinaciju. Ako imamo dovoljno jako računalo, ovaj postupak bi bio moguć, no vrlo je vjerojatno da nemamo takvo računalo pri ruci osim ako smo američka vojska. Zbog toga želimo pronaći algoritam koji neće ispitivati svaku moguću opciju nego se usmjeriti prema boljim opcijama i istražiti njih. Prva ideja koju možemo

uzeti u obzir je da krenemo od početne ljekarne i krenemo u prvu najbližu, zatim iz nje u sljedeću najbližu itd., uzimajući u obzir da se ne vraćamo u već posjećene. Takav algoritam jako brzo pronalazi rješenje, ali često ne pronalazi dovoljno dobro rješenje.

On je primjer algoritma naziva algoritam najbližeg susjeda. Dio je skupine algoritama koje nazivamo približni algoritmi ili heurističke metode. Oni ponekad nađu zadovoljavajuća rješenja, ali ne nude nikakvu garanciju da će pronaći optimalno ili približno optimalno rješenje. Sljedeća nadogradnja koju bi mogli iskoristiti je da koristimo neku dodatnu metriku koja nam pomaže kod odabira tipa broj ljekarni koje su udaljene manje od 10km od potencijalne sljedeće. Cilj nam je čim više ljekarni grupirati pa bi ovu metriku uzeli u obzir kod funkcije pogreške. Takav algoritam također bi se svrstao u heurističke metode.

U prethodnom dijelu razmatrali smo jednu problemski specifičnu heuristiku. Što ako bi mogli napraviti neki algoritam koji nam usmjerava postupak optimizacije za bilo koji problem. Takav skup algoritama se naziva metaheuristike i radi se o heuristici opće primjene koja usmjerava problemski specifičnu heuristiku prema području u prostoru rješenja gdje se nalaze dobra rješenja (Yang, 2009). Neki primjeri su lokalna pretraga i njene varijante, simulirano kaljenje, tabu pretraga i genetski algoritam.

Prema David S. Johnson, Christos H. Papadimitriou i Mihalis Yannakakis, jedan od rijetkih općih pristupa teškoj kombinatornoj optimizaciji koji je imao empirijski uspjeh je i lokalna pretraga (David S. Johnson, Christos H. Papadimitriou i Mihalis Yannakakis, 1988.).

Said, Mahmoud i El-Horbaty su 2014. su nizom eksperimenata otkrili da genetski algoritmi češće dovode do boljeg rješenja od tabu pretrage dok tabu pretraga ima brže vrijeme izvođenja na *Quadratic assignment* problemima koji su jedni od osnovnih problema raspoređivanja (Said, Mahmoud, & El-Horbaty, 2014).

Kod razvoja samog algoritma moramo biti svjesni i "Nema besplatnog ručka" teorema. Teorem "Nema besplatnog ručka" kaže da, u prosjeku za sve probleme optimizacije, bez ponovnog uzorkovanja, svi algoritmi optimizacije rade jednako dobro (Adam,S.P., Alexandropoulos, S.A.N.,Pardalos, P.M.,&Vrahatis, M.N., 2019.). Kako bi algoritam dobro rješavao neki konkretan problem, potrebno je dobro odabrati prikladni algoritam čime se prilagođavamo problemu (Čupić,2013.).

1. Opis problema

Cilj ovog rada je osmisliti cjelokupan sustav za primanje narudžbe, upravljanje skladištem, pretvorbu narudžbe u radni nalog i prikazati kako optimizacija rasporeda radnih naloga olakšava proizvodni proces. U ovom poglavlju se predstavlja detaljni opis ovakvog sustava i daje pregled optimizacijskih algoritama.

1.1. Opis predviđenog sustava

1.1.1. Upravljanje korisnicima

Sustav mora podržati prijavu korisnika. Korisnici imaju različite uloge unutar sustava i na temelju uloga imaju različite ovlasti oko upravljanja podacima i sustavom. Admin ima mogućnost upravljanja svim podacima. Ostatak rada treba biti fokusiran na temelju njegovog pregleda sustava. Ostale uloge se mogu stvoriti na temelju organizacijske strukture konkretnog sustava i njihovo određivanje nije dio teme ovog rada, već je potrebno samo podržati njihov rad.

1.1.2. Upravljanje skladištima

Potrebno je podržati rad sa skladištima. Jedna tvrtka može imati više lokacija i na svakoj lokaciji može imati više skladišta. Svako skladište može imati više katova. Svaki kat ima više skladišnih mjesta.

Svako skladišno mjesto može blokirati neka druga skladišna mjesta. Ako skladišno mjesto blokira neko drugo skladišno mjesto onda se roba mora pomaknuti s njega svaki puta kada se pristupa blokiranome mjestu. Potrebno je omogućiti grafičko označivanje koje skladišno mjesto blokira koja druga skladišna mjesta.

Na svako skladišno mjesto se može skladištiti više različitih kombinacija vrsta pakiranja. Potrebno je omogućiti grafičko određivanje mogućih mjesta za kombinaciju vrsta pakiranja.

Sustav mora podržati pregled skladišta na zadani datum.

1.1.3. Upravljanje podacima o proizvodima

Sustav mora voditi evidenciju svih poslovnih partnera. Sustav mora podržati rad s proizvodima. Proizvod može biti sirovina koja je kupljena od strane poslovnog partnera i može biti gotov proizvod koji je proizveden od strane organizacije koja je korisnik sustava. Svaki proizvod mora biti sortiran unutar neke grupe proizvoda. Sustav mora omogućiti da proizvod bude spremljen unutar nekog pakiranja proizvoda. Pakiranje proizvoda pamti koje je vrste pakiranja i time omogućuje kasniji raspored po skladištu na temelju već opisanih kombinacija vrsta pakiranja za svako skladišno mjesto.

1.1.4. Upravljanje narudžbama

Sustav mora podržati evidenciju naručenih pakiranja proizvoda prema dobavljačima. Mora se evidentirati datum i vrijeme dolaska robe te pakiranja proizvoda koji dolaze s kojom količinom.

Sustav mora podržati evidenciju narudžbe kupaca prema proizvodnji korisnika sustava. Narudžba ima svoj željeni datum isporuke i pakiranja proizvoda koji su naručeni s kojom količinom.

1.1.5. Upravljanje radnim nalozima

Sustav mora omogućiti stvaranje radnog naloga iz narudžbe kupca. Za svako pakiranje proizvod unutar narudžbe kupca, radni nalog određuje datum proizvode, radne stanice na kojima se to pakiranje proizvoda proizvodi te sastojci odnosno pakiranja proizvoda sirovina koji su potrebni za proizvodnju tog naloga. Sustav mora omogućiti automatizirano raspoređivanje radnih naloga po strojevima.

1.2. Optimizacijski algoritmi

1.2.1. Osnovni pojmovi optimizacijskih algoritama

- Prostor rješenja – Prostor rješenja sadrži sva moguća rješenja. Rješenja su točke unutar prostora rješenja. One mogu biti diskretne ili kontinuirane. Primjer kontinuiranog prostora rješenja je [-100, 200] gdje su rješenja sve što se nalazi unutar prostora rješenja. Primjer diskretnog prostora rješenja je uređena četvorka

točaka A.B.C.D koja može predstavljati nekakav put. Primjer rješenja unutar takvog prostora rješenja bi bio A - C - D – B, dok bi drugi bio C – D – A – B.

- Ograničenja – Problem može imati ograničenja na prostor rješenja kao što je to u prošlom primjeru na [-100, 200]. Postoje tvrda i meka ograničenja. Tvrda ograničenja dijele prostor stanja na prostor prihvatljivih rješenja i na prostor neprihvatljivih rješenja. Ako neko tvrdo ograničenje nije zadovoljeno, onda to rješenje nije prihvatljivo. Meka ograničenja definiraju poželjna svojstva i definiraju spektar zadovoljstva rješenjem od jako do slabo zadovoljeno. Meka ograničenja se mjere pomoću funkcije dobrote.
- Funkcija dobrote – funkcija dobrote kao ulaz prima neko rješenje, na temelju tog rješenja izračunava koliko je to rješenje dobro/loše odnosno koliko su zadovoljena meka ograničenja i to prosljeđuje na izlaz. Funkcije dobrote su izrazito korisne kako bi se mogli kvalitetno i efektivno kretati po prostoru rješenja i usmjeravati se u prostore boljih rješenja.
- Lokalni optimum - Rješenje x* naziva se lokalnim optimumom ako i samo ako rješenje x* pripada prostoru prihvatljivih rješenja i ako za svako drugo rješenje x iz delta-okoline od x*, tj. uz delta > 0 i $|x-x*| \leq$ delta vrijedi $f(x*) \geq f(x)$, gdje je f funkcija dobrote rješenja (Čupić, 2013.).
- Globalni optimum – Rješenje x* za koje vrijedi isto što i za lokalni optimum, ali na cijelome prostoru stanja.

1.2.2. Neki optimizacijski algoritmi

Optimizacijske algoritme dijelimo na 2 grane a to su egzaktni algoritmi i približni algoritmi. Egzaktni algoritmi garantiraju pronalazak najboljeg rješenja, ali zato nisu primjenjivi na zahtjevne probleme. Neke od najpopularnijih egzaktnih metoda su : A*, *Simplex*, Dinamičko programiranje i *Branch and bound.* Oni nisu primjenjivi na problem optimizacije rasporeda proizvodnje. Glavna grana unutar približnih algoritama su heuristički algoritmi. Heuristika je pristup rješavanju problema koji rangira različite alternative u algoritmima pretraživanja u svakome koraku grananja kako bi se moglo odlučiti na koju granu krenuti.

Jedan od osnovnih heurističkih algoritama je *greedy* algoritam. On pripada skupini konstruktivnih algoritama koji gradi jedinstveno rješenje unutar prostora rješenja korak po

korak donoseći odluke na temelju heuristike. Kod svakog grananja on uzima što se u tome trenutku čini kao najbolji gradivni blok rješenja. *Greedy* algoritam je pogodan za jednostavne probleme. Često su takvi problem rješivi i egzaktnim algoritmima. Dodatna prednost *greedy* algoritma je i brzina izvođenja. Ako nam je brzina izvođenja bitnija od kvalitete rješenja onda je potencijalno dobar odabir *greedy* algoritam.

Druga velika skupina približnih algoritama su i metaheuristike. Metaheuristike su algoritmi koji rade na principu inkrementalnog poboljšavanja rješenja. One se najčešće koriste na NP-teškim problemima. Jedan od ključnih koncepata metaheuristika je lokalna pretraga. Ona omogućava brz pronalazak najboljeg susjeda iz susjedstva nekog rješenja. Dodatne komplikacije oko lokalne pretrage su izrodile sljedeće korisne algoritme: *Multistart local search, Iterative local search, Varible local search, Large neighborhood local search.* Tijekom godina se razvio i veliki broj prirodom inspiriranih metaheuristika koje se zajedno često nazivaju i evolucijski algoritmi. Prvi takav algoritam je simulirano kaljenje dok su kasnije dobre rezultate pokazali i tabu pretraga, genetski algoritam, mravlja kolonija, roj čestica itd.

U ovome radu će biti implementiran *Multistart local search* i simulirano kaljenje kao predstavnici iz svake od povijesno značajnih skupina metaheuristika pogodnih za algoritme optimizacije rasporeda.

2. Arhitektura sustava i tehnologije

2.1. Tehnologije implementacije

Aplikacija koristi moderne web tehnologije koje omogućavaju kvalitetnu implementaciju, dugoročno lako održavanje i nadogradnju.

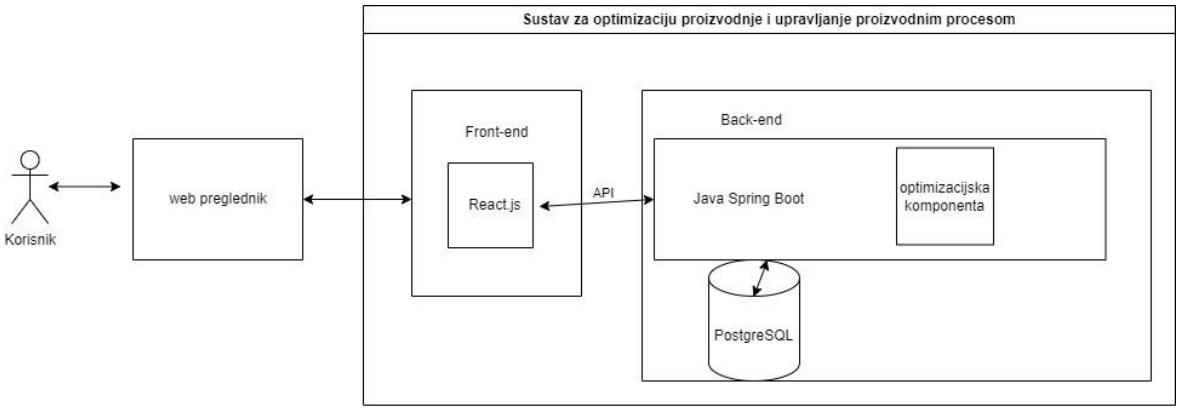
Tehnologije korištene u izradi aplikacije su:

- Spring Boot – backend tehnologija
- Java – unutar spring boota za sve algoritme optimizacije
- React.js – SPA library za moderno sučelje
- PostgreSQL – za bazu podataka
- DaisyUI – za moderan i elegantan izgled sučelja
- JWT – za autorizaciju

2.2. Arhitektura sustava

Arhitektura aplikacije se sastoji od klasične moderne raspodijele na front-end i back-end.

Korisnik aplikaciji pristupa pomoću web preglednika npr. firefox na kojem se izvodi front-end. Korisnikove zahtjeve preglednik šalje na back-end gdje se oni obrađuju i vraćaju rezultat. To je prikazano i na slici (Slika 2.1).



Slika 2.1. Opis arhitekture sustava

3. Implementacija sustava

3.1. Implementacija upravljanja podacima

Za kvalitetno upravljanje podacima, potrebno je osigurati straničenje, sortiranje, filtriranje, promjenu postojećih zapisa, *soft* brisanje zapisa i dodavanje novih zapisa.

3.1.1. Straničenje, sortiranje i filtriranje tabličnih podataka

Aplikacija koristiti spring boot JpaRepository i @Repository te joj to omogućuje da standardizira straničenje, sortiranje i filtraciju tabličnih podataka. Pogledajmo to na primjeru tablice korisnika. REST api prima 3 polja: Pagable pagable, String property i String value. Pagable sadržava 4 podatka potrebna za straničenje i sortiranje, a to su: current page, size, sort property i sort direction. Property je atribut po kojem će se tablični podaci filtrirat, a value je vrijednost koja se traži unutar toga atributa (Slika 3.1.).

```
@Override
public Page<UserDto> pageUsers(Pageable pageable, String property, String value) {
    if (value.isBlank()) {
        return userRepository.findAll(pageable).map(UserDto::new);
    }
    switch (property) {
        case "id":
            long id;
            try {
                id = Long.parseLong(value);
            } catch (Exception e) {
                return userRepository.findAll(pageable).map(UserDto::new);
            }
            return userRepository.findByIdEquals(id, pageable).map(UserDto::new);
        case "name":
            return userRepository.findByNameContainsIgnoreCase(value, pageable).map(UserDto::new);
        case "surname":
            return userRepository.findBySurnameContainsIgnoreCase(value, pageable).map(UserDto::new);
        case "email":
            return userRepository.findByEmailContainsIgnoreCase(value, pageable).map(UserDto::new);
        default:
            return userRepository.findAll(pageable).map(UserDto::new);
    }
```

Slika 3.1. Straničenje na backendu

Na front-endu su implementirane vlastite react hook komponente naziva „usePageCommand“ i „usePageCommandDesc“. One omogućavaju da si bilo koja komponenta pozove osnovnu verziju pagination objekta (Slika 3.2.).

```
import { useState } from 'react';

export default function usePageCommand(initialState = {
  "currentPage": 0,
  "size": 14,
  "sortProperty": "id",
  "sortDirection": "ASC"
}) {
  const [pageCommand, setPageCommand] = useState(initialState);

  return [pageCommand, setPageCommand];
}
```

Slika 3.2. Vlastiti react hook

Jednostavan primjer korištenja je prikazan slijedećim kodom

„const [pageCommand,setPageCommand] = usePageCommand()„

Sort property i sort direction je upravljan pomoću <select> </select> html objektom (slika 3.3.).

```
<div className="flex flex-row">
    <div className="text-lg m-2">
        Sort:
    </div>
    <select selected={pageCommand.sortProperty} onChange={(e) => setPageCommand({...pageCommand,sortProperty : e.target.value})}
    className="select select-sm select-info w-full max-w-xs m-2">
        <option value={"id"}>ID</option>
        <option value={"name"}>Ime</option>
        <option value={"surname"}>Prezime</option>
        <option value={"email"}>Email</option>
    </select>
    <select selected={pageCommand.sortDirection} onChange={(e) => setPageCommand({...pageCommand,sortDirection : e.target.value})}
    className="select select-sm select-info w-full max-w-xs m-2">
        <option value={"ASC"}>uzlazno</option>
        <option value={"DESC"}>silazno</option>
    </select>
</div>
```

Slika 3.3. Front end sortiranje

Property i value vrijednosti filtracije su također upravljanje <select> html objektom (slika 3.4.).

```
<div className="flex flex-row">
    <div className="text-lg m-2">
        Filter:
    </div>
    <select selected={property} onChange={(e) => setProperty(e.target.value)}
    className="select select-sm select-info w-full max-w-xs m-2">
        <option value={"id"}>ID</option>
        <option value={"name"}>Ime</option>
        <option value={"surname"}>Prezime</option>
        <option value={"email"}>Email</option>
    </select>
    <label className="input input-sm m-2 input-bordered input-info flex items-center gap-2">

        <input type="text" className="grow" placeholder="unesite vrijednost" onBlur={(e) => setValue(e.target.value)} />
    </label>
```

Slika 3.4. Front end filtriranje

UseEffect je react-hook koji poziva funkciju kod svake promjene njemu zadanih vrijednosti. Vidimo da kod svake promjene pageCommand, property, value i changed varijable on zove api gdje dohvaća korisnike koje zadovoljavaju uvjetima (Slika 3.5.).

```
useEffect(() => {
    console.log(property)
    console.log(value)
    api.post("/users/page?property=" + property + "&value=" + value,pageCommand)
    .then(res => res.data)
    .then(data => {
        console.log(data)
        setData(data.content)
        setPagination(Array.from({ length: data.totalPages }, (_, index) => index))
    })
},[pageCommand,property,value,changed])
```

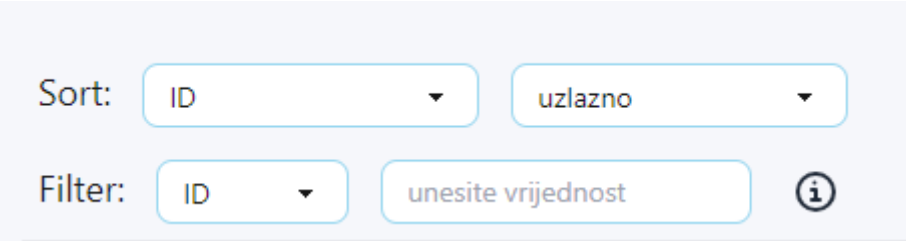
Slika 3.5. Pozivi api-ja za dohvat podataka na temelju uvjeta

Sam prikaz straničenja na dnu tablice je omogućen pomoću funkcije prikazane na slici 3.6.

```
</div>
        <hr/>
        <div className="flex justify-center mt-5">
            <div className="join">
                {
                    numbersArray.map(num => <div key={num}>{
                        pageCommand.currentPage === num ?
                        <button className="join-item btn btn-active">
                        {num}
                    </button> :
                    <button className="join-item btn" onClick={() => setPageCommand({...pageCommand,currentPage : num})}>
                        {num}
                    </button>
                    }</div>)
                }
            </div>
        </div>

    </div>
}
```

Slika 3.6. Prikaz straničenja na dnu stranice

Na slici 3.7. vidimo prikaz sortiranja i filtriranja u sučelju, a na slici 3.8. vidimo prikaz straničenja na dnu stranice u sučelju.



Slika 3.7. Prikaz sortiranja i filtriranja u sučelju

Slika 3.8. Prikaz straničenja na dnu stranice

3.1.2. Dodavanje i uređivanje zapisa tabličnih podataka

Za dodavanje i uređivanje zapisa sustav koristi istu komponentu. Komponenta prima „prop“ koji označava objekt koji se uređuje, a ukoliko je taj „prop“ undefiend onda komponenta zna da se radi o novom unosu (slika 3.9.).

```
useEffect(() => {
    setError("")
    setSuccess("")
    if(specific === undefined) {
        reset()
        return
    }
    Object.keys(specific).forEach(field => {
      setValue(field, specific[field]);
    });
  }, [specific, setValue,reset]);
```

Slika 3.9. prikaz inicijalizacije komponente za uređivanje/dodavanje zapisa

Za formu za uređivanje/dodavanje podataka sustav koristi react-hook-form koja omogućava lakšu manipulaciju formom. Sljedećim kodom inicijaliziramo formu u toj komponenti.

„const { register, handleSubmit, formState: { errors },setValue,reset } = useForm();“

Register koristimo u jsx kodu za definiranje polja, setValue koristimo kod inicijalog postavljanja polja kao kod slike 3.9.. HandleSubmit koristimo dok korisnik preda formu na izvršavanje odnosno na slanje podataka na back-end (Slika 3.10.).

```
const onSubmit = data => {
        setLoading(true)
        api.post("/users",data)
        .then(() => {
         onSuccess()
         setSuccess("Uspjeh!")
         reset()
        })
        .catch(ex => {
            setError(ex)
        })
        .finally(() => {
         setLoading(false)
        })
    }
```

Slika 3.10. Prikaz slanja obrađenog zapisa na back-end

3.1.3. Grafičko uređivanje skladišnih mjesta

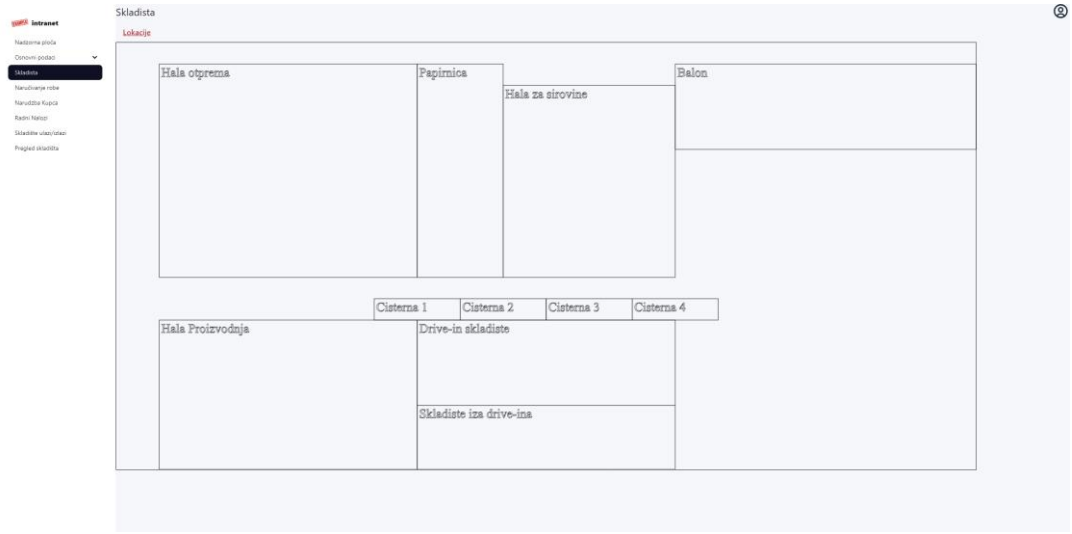
Za grafički prikaz vezani uz skladišta aplikacija koristi canvas iz html5. HTML element canvas koristi se za crtanje grafike, u hodu, putem JavaScripta (w3schools,2024.). Za prikaz svih skladišta unutar lokacije, prvo dohvatimo sva skladišta iz baze podataka. Svako skladište osim osnovnih podataka poput imena, ima i 4 podatka koji su potrebni za uspješno ocrtavanje pomoću canvasa. Oni su : x,y,width i height. Pretpostavka je da se svako skladište može prikazati pravokutnikom. Koordinate jedne točke su x i y dok visina i širina onda određuju izgled pravokutnika. Pravokutnik se crta sa canvas.getContext(„2d“).rect(…), a ime skladišta se centrira unutar pravokutnika i upisuje pomoću canvas.getContext(„2d“).stroketText(…). Svi stvoreni pravokutnici se spremaju u listu i nadodaje im se listener na click unutar njihovog ocrtanog područja (Slika 3.11.).

Kada se klikne na neki ocrtani pravokutnik unutar sučelja, on se odabere i ulazimo u to skladište.

```
useEffect(() => {
    if(selected != "") {
        return;
    }
    api.get("/skladista/"+adresa)
    .then(res => res.data)
    .then(data => {
        const canvas = document.getElementById("myCanvas");
        if(window.innerWidth > 768) {
            canvas.style.width = "" + window.innerWidth * 0.8+ "px"
            canvas.style.height = "" + window.innerHeight * 0.8 + "px"
            canvas.width = window.innerWidth * 0.8
            canvas.height = window.innerHeight * 0.8
        } else {
            canvas.style.width = "" + window.innerWidth + "px"
            canvas.style.height = "" + window.innerHeight * 0.8 + "px"
            canvas.width = window.innerWidth
            canvas.height = window.innerHeight *0.8
        }
        const ctx = canvas.getContext("2d");
        let elements = []
        data.forEach(element => {
            console.log(element)
            const rect1 = new Path2D()
            rect1.ime = element.ime
            rect1.brojKatova = element.brojKatova
            rect1.xi = element.xskladisnihMjesta
            rect1.yj = element.yskladisnihMjesta
            rect1.rect(element.xmax * canvas.width / 100, element.ymax * canvas.height / 100,element.width * canvas.width / 100,element.height * canvas.height / 100);
            elements.push(rect1)
            ctx.stroke(rect1)

            ctx.font = "30px Ariel";
            ctx.strokeText(element.ime, (element.xmax * canvas.width / 100) + 5, (element.ymax * canvas.height / 100) + 30,(element.width * canvas.width / 100) - 5);
        });

        canvas.addEventListener('click', function(event) {

            elements.forEach(element => {
                if (ctx.isPointInPath(element, event.offsetX, event.offsetY)) {
                    setSelected(element)
                    ctx.strokeStyle = 'green';
                    ctx.stroke(element);
                  }
                  else {
                    ctx.strokeStyle = 'black';
                    ctx.stroke(element);
                  }
            });

          });

    })
},[adresa,selected])
```

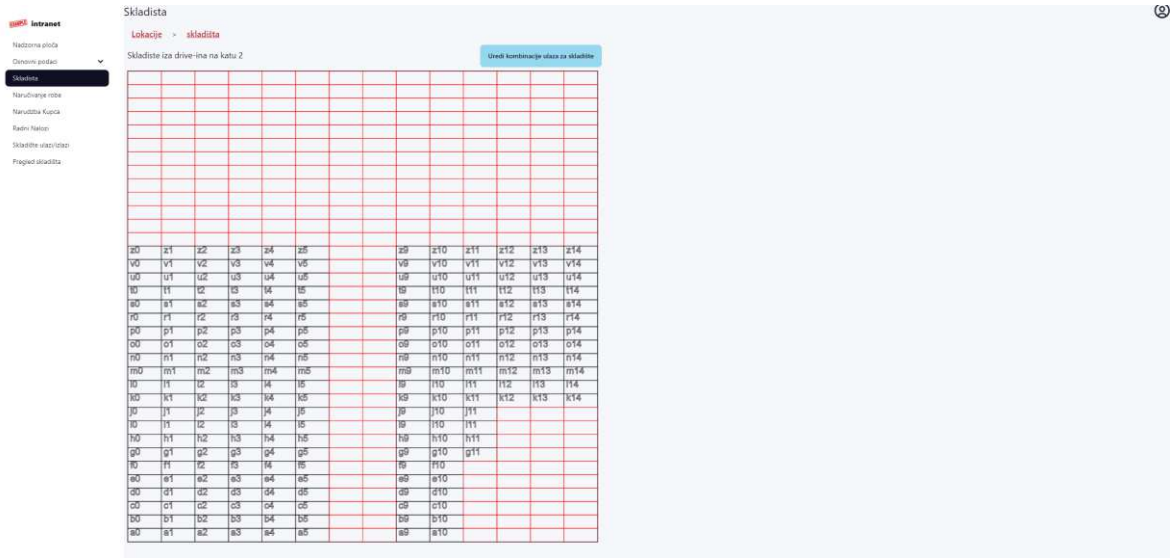
(parameter) element: any

Slika 3.11. Kod koji prikazuje crtanje prikaza svih skladišta

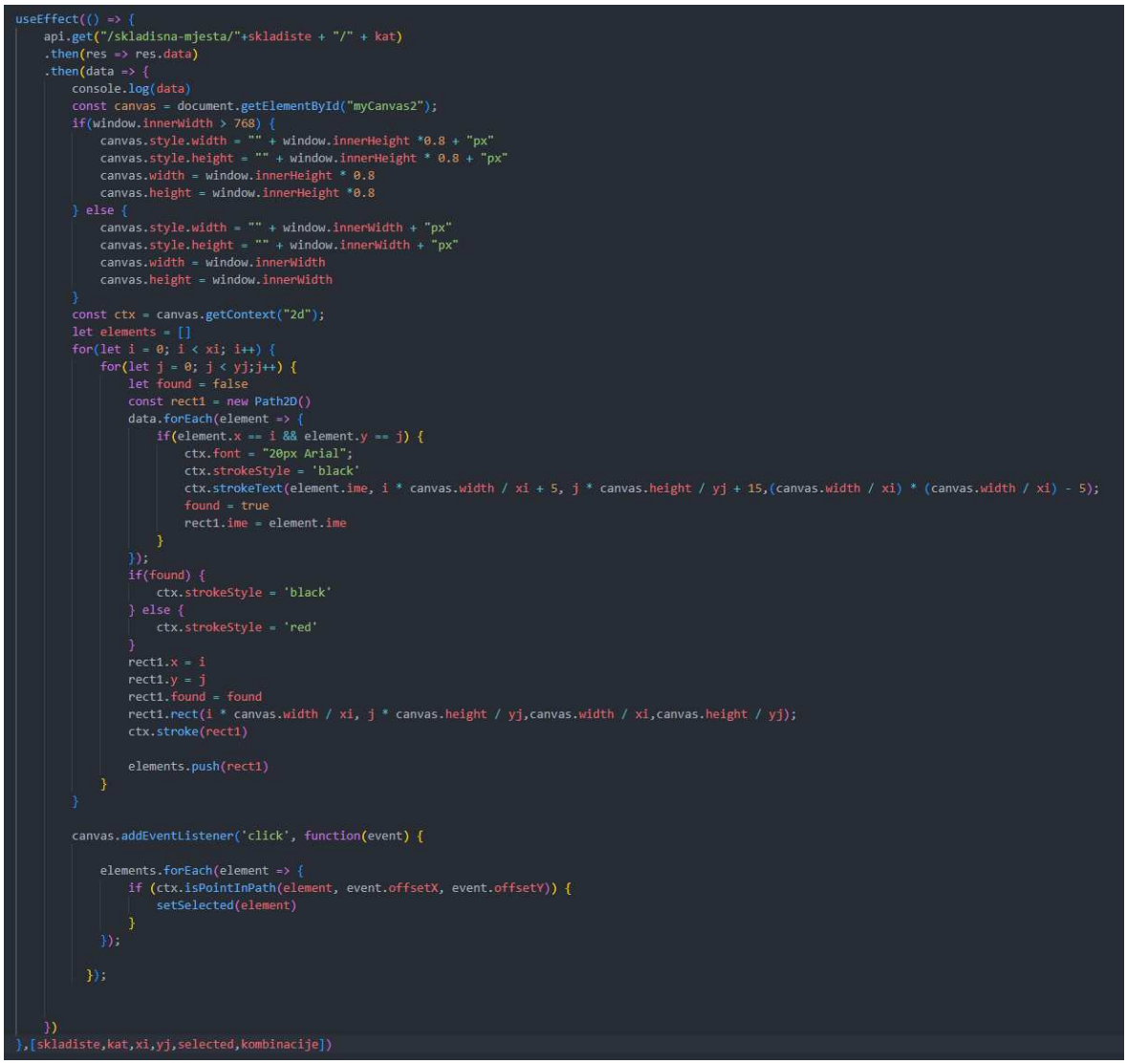


Slika 3.12. Kod koji prikazuje crtanje prikaza svih skladišta

Nakon klika na neko skladište koje se želi uređivati, dolazi se na prikaz svih skladišnih mjesta u odabranome skladištu na odabranome katu (Slika 3.13.). Određeno skladište ima svoju x-os i y-os kao na slici 3.13. x = 15, a y =35 koje omogućavaju jednoznačni prikaz skladišnih mjesta. Skladišna mjesta se prikazuju sličnim kodom kao i kod prikaza skladišta (Slika 3.14.). Skladišna mjesta koja su u funkciji skladištenja imaju crne rubove i imena, dok je prazan prostor odnosno skladišna mjesta koja nisu u funkciji obrubljena crvenom bojom.



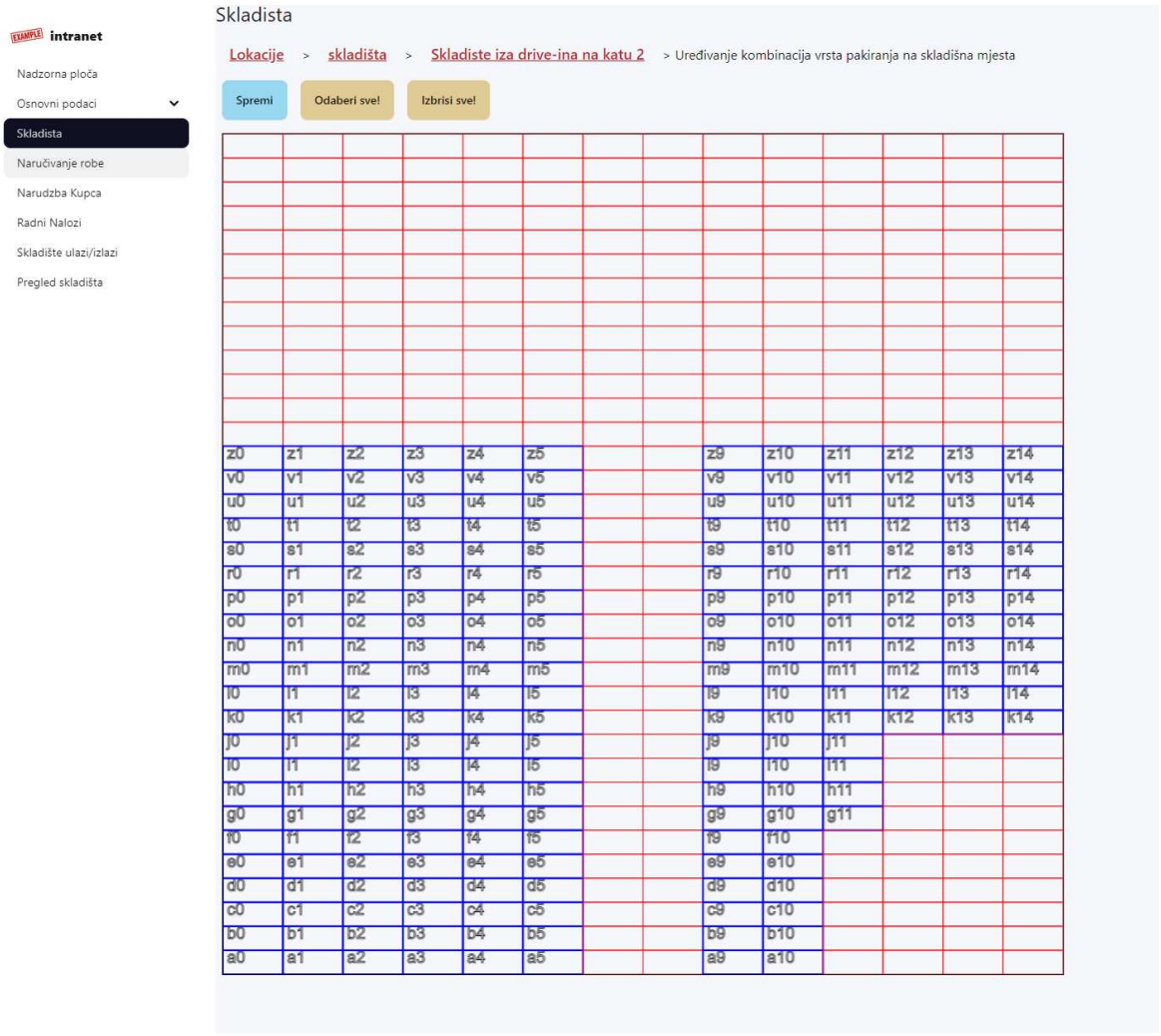
Slika 3.13. Prikaz svih skladišnih mjesta u odabranome skladištu na odabranome katu.



```
useEffect(() => {
    api.get("/skladisna-mjesta/"+skladiste + "/" + kat)
    .then(res => res.data)
    .then(data => {
        console.log(data)
        const canvas = document.getElementById("myCanvas2");
        if(window.innerWidth > 768) {
            canvas.style.width = "" + window.innerHeight *0.8 + "px"
            canvas.style.height = "" + window.innerHeight * 0.8 + "px"
            canvas.width = window.innerHeight * 0.8
            canvas.height = window.innerHeight *0.8
        } else {
            canvas.style.width = "" + window.innerWidth + "px"
            canvas.style.height = "" + window.innerWidth + "px"
            canvas.width = window.innerWidth
            canvas.height = window.innerWidth
        }
        const ctx = canvas.getContext("2d");
        let elements = []
        for(let i = 0; i < xi; i++) {
            for(let j = 0; j < yj;j++) {
                let found = false
                const rect1 = new Path2D()
                data.forEach(element => {
                    if(element.x == i && element.y == j) {
                        ctx.font = "20px Arial";
                        ctx.strokeStyle = 'black'
                        ctx.strokeText(element.ime, i * canvas.width / xi + 5, j * canvas.height / yj + 15,(canvas.width / xi) * (canvas.width / xi) - 5);
                        found = true
                        rect1.ime = element.ime
                    }
                });
                if(found) {
                    ctx.strokeStyle = 'black'
                } else {
                    ctx.strokeStyle = 'red'
                }
                rect1.x = i
                rect1.y = j
                rect1.found = found
                rect1.rect(i * canvas.width / xi, j * canvas.height / yj,canvas.width / xi,canvas.height / yj);
                ctx.stroke(rect1)

                elements.push(rect1)
            }
        }

        canvas.addEventListener('click', function(event) {

            elements.forEach(element => {
                if (ctx.isPointInPath(element, event.offsetX, event.offsetY)) {
                    setSelected(element)
                }
            });

         });


    })
},[skladiste,kat,xi,yj,selected,kombinacije])
```

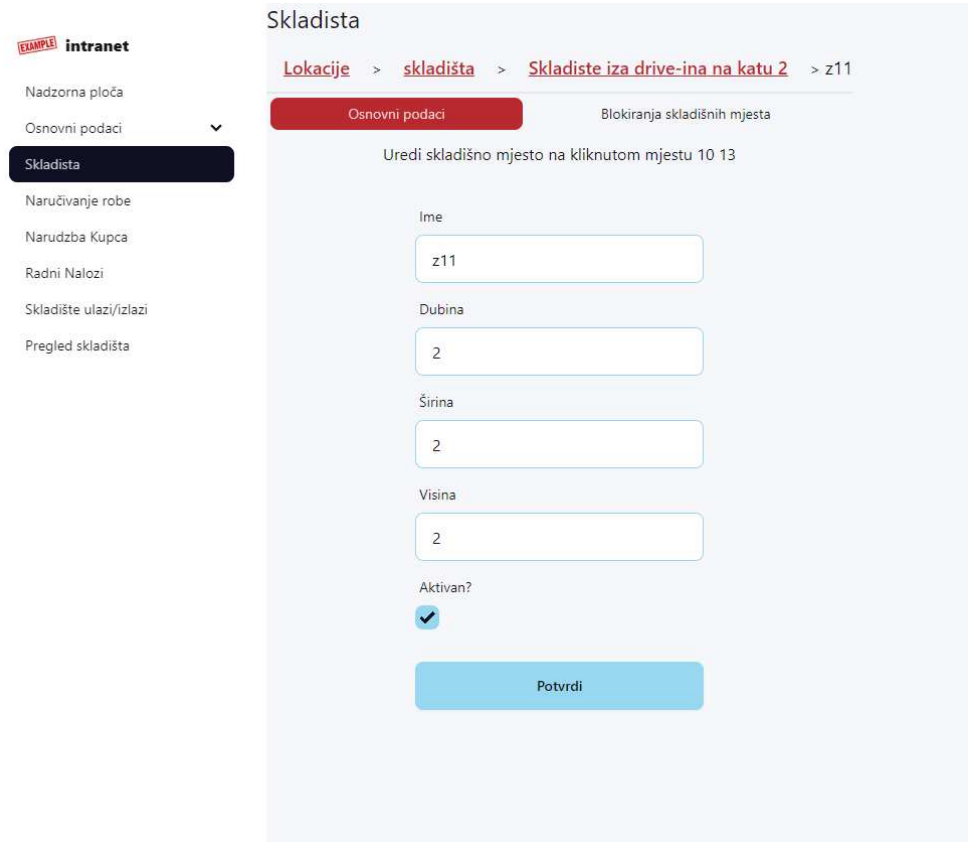
Slika 3.14. Kod za prikaz skladišnih mjesta unutar skladišta

Na slici 3.13. se vidi u gornjem desnom kutu skladišnih mjesta gumb „Uredi kombinacije ulaza za skladište“. Klikom na taj gumb i odabirom kombinacije vrsta pakiranja koje se želi uređivati, dolazi se na ekran gdje se mogu odabrati/izbrisati mjesta na koje se ta kombinacije može skladištiti. Odabrana mjesta su obrubljena plavom bojom, neodabrana su obrubljena crnom bojom dok je prazan prostor odnosno skladišna mjesta koja nisu u funkciji obrubljena crvenom bojom (Slika 3.15.). Radi lakšeg korištenja aplikacija pruža i gumbe „odaberite sve“ i „izbriši sve“, kako bi se lako moglo odabrati ili izbrisati sva mjesta iz odabira.



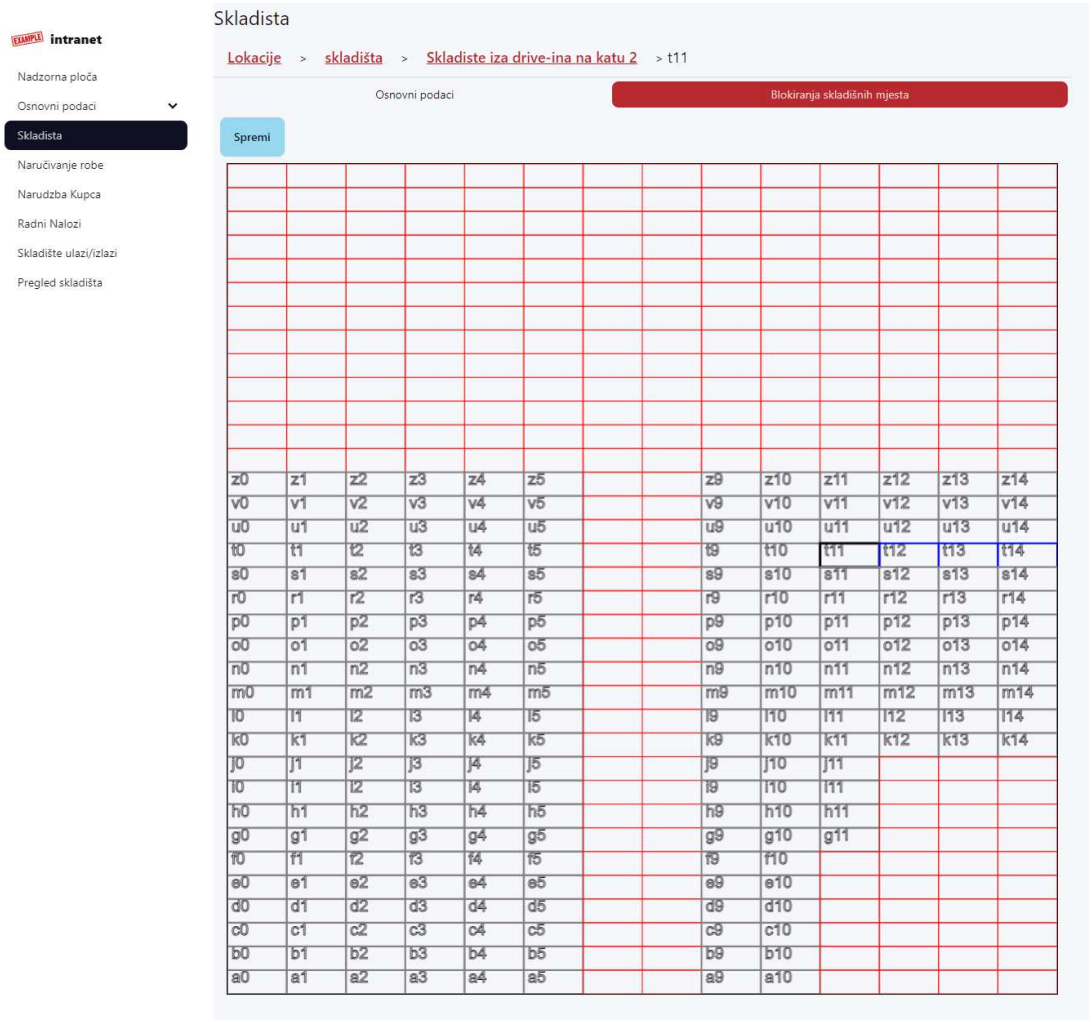
Slika 3.15. Prikaz uređivanja kombinacija skladištenja po vrstama pakiranja

Klikom na neko postojeće skladišno mjesto na slici 3.15. otvara se forma za uređivanje podataka o tome skladišnome mjestu odnosno mogućnost brisanja toga skladišnoga mjesta. Klikom na neko ne postojeće skladišno mjesto na slici 3.15. otvara se forma za unos novog skladišnog mjesta na tome mjestu (Slika 3.16.).



Slika 3.16. Uređivanje osnovnih podataka o skladišnome mjestu.

Dodatno za svako skladišno mjesto možemo i određivati skladišna mjesta koja ono blokira (Slika 3.17.). Plavom bojom na slici 3.17. su prikazana odabrana mjesta odnosno ona mjesta koja su blokirana trenutnim skladišnim mjestom. Trenutno skladišno mjesto je obrubljena duplim crnim okvirom dok su neodabrana obrubljena jednostrukim crnim okvirom. Klikom na bilo koje mjesto se ono može odabrati odnosno izbrisati iz odabira.

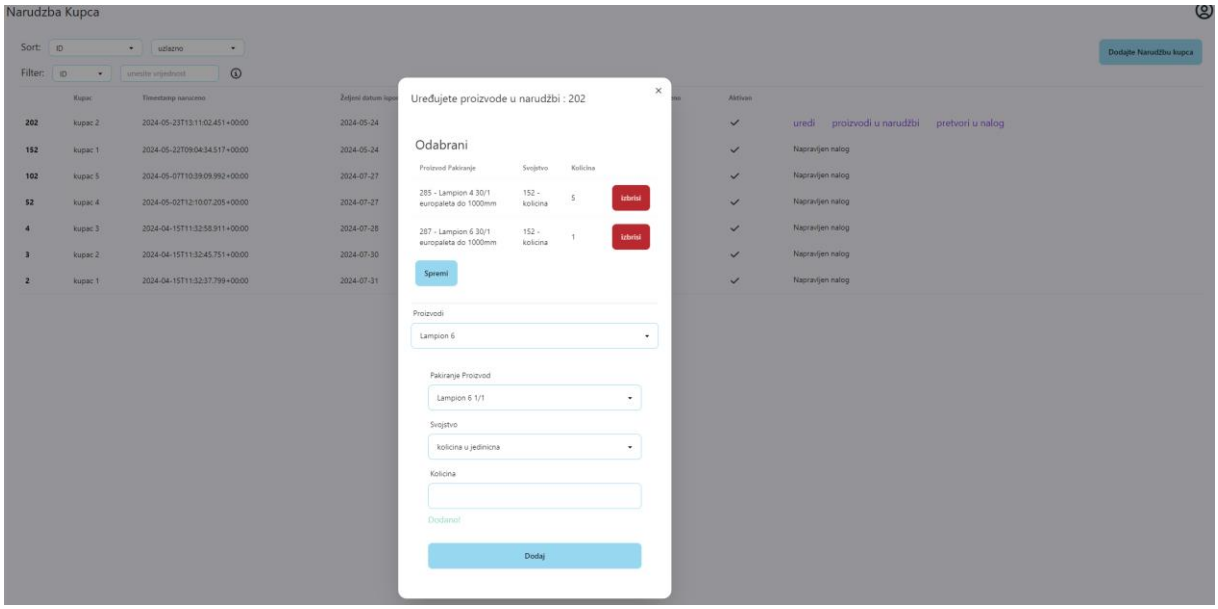


Slika 3.17. Prikaz uređivanja skladišnih mjesta koje se blokiraju

4. Optimizacijski algoritam za raspoređivanje radnih naloga

Cilj ovog optimizacijskog algoritma je stvoriti raspored proizvodnje za određeni datum za svaki stroj.

Sve kreće od narudžbi kupaca. Narudžba kupca se sastoji od određenih proizvoda i količina koje kupac naručuje. Pretpostavljamo da se proizvodnja radi po narudžbama.

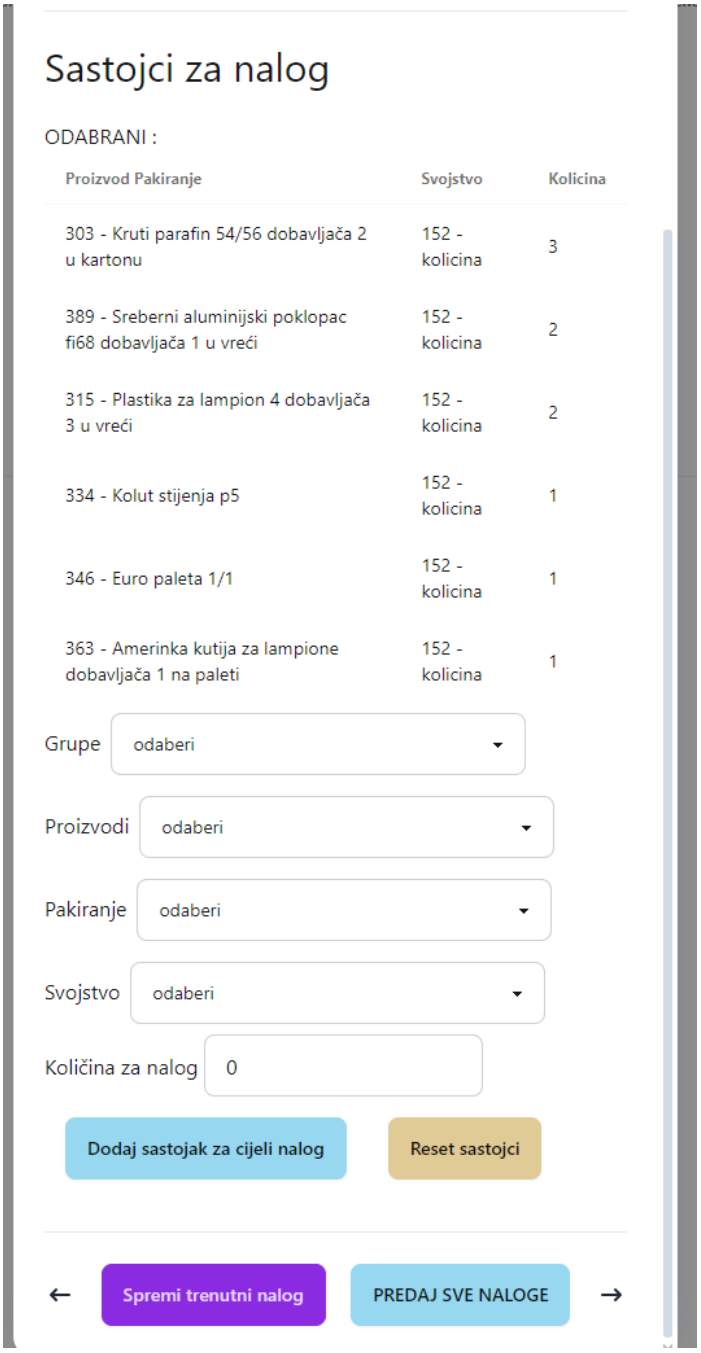


Slika 4.1. Primjer upisa proizvoda u narudžbu

Svaka narudžba se zatim pretvara u 1..n radnih naloga. Broj naloga zavisi o broju proizvoda unutar narudžbe. Svaki proizvod unutar narudžbe se pretvara u vlastiti radni nalog.

Potaknuto problematikom proizvodnje svijeća, gdje svaki kupac može imati drugačije sastojke za isti proizvod te dodatno različite sastojke od isporuke do isporuke, omogućen je ekran gdje se mogu lako naći i dodati sastojci za određeni proizvod za određeni radni nalog (slika 4.2.).

Svaki nalog se mora proizvoditi na 1…n strojeva koji postoje u tvornici. Putanja po strojevima se također može odrediti na razini naloga. Nalog se mora proizvoditi na svim strojevima, točno tim redom koji je određen kod stvaranja radnog naloga (Slika 4.3.)



Slika 4.2. Određivanje sastojaka za određeni nalog

Slika 4.3. Određivanje datuma proizvodnje i putanje po strojevima

Pretpostavka algoritma je da proizvodnja može raditi na svim strojevima u isto vrijeme. Dodatna pretpostavka je kako svi strojevi imaju jednak ciklus rada i kako mogu obraditi svu potrebnu količinu u tom ciklusu. Ciklusi rada mogu biti različiti od industrije do industrije npr. 10 minuta ili 2 sata. Jedan ciklus je vremenski period potreban najsporijem stroju u proizvodnom pogonu da obradi svu količinu koju obrađuje na temelju rasporeda. Jedan radni nalog mora slijediti odabrani redoslijed strojeva. Jedan radni nalog se ne smije u istome ciklusu proizvoditi na 2 stroja već se podrazumijeva da je potreban završeni izlaz iz jednog stroja kao ulaz u sljedeći stroj u rasporedu. Jedan radni nalog se ne mora proizvoditi na svim strojevima već samo na onima zadanima od poslovođe. Stroj 1, Stroj 2

i Stroj 3 u sljedećim primjerima su samo nazivi strojeva i nikako ne određuju implicitni put radnog naloga od stroja 1 preko stroja 2 do stroja 3.Tako da je mogući put proizvoda stroj 3 -> stroj 1 -> stroj 2 jer su brojevi samo dio naziva u ovome primjeru. Ako se na određeni datum može proizvesti samo određeni broj koraka, možemo to dodatno proglasiti kao tvrdu granicu prostora rješenja

4.1. Multistart local search

U okviru diplomskog rada implementirana su 2 algoritma. Prvi je „Višekratno pokretanje lokalne pretrage“ poznatiji kao *Multistart local search*.

Lokalna pretraga je koristan algoritam koji lako i brzo pronađe najbolje rješenje u zadanome susjedstvu. Problem je što pronađeno rješenje izrazito ovisi o kvaliteti susjedstva inicijalno slučajno generiranog rješenja.

Ideja algoritma *Multistart local search* je da više puta pokrećemo lokalnu pretragu na nezavisno generiranim inicijalnim rješenjima i samim time anuliramo ovisnost o kvaliteti susjedstva inicijalnog rješenja.

Multistart local search s dovoljnim brojem iteracija generira dovoljno različitih inicijalnih slučajnih rješenja da se pretraži široki prostor rješenja.

Sljedeći pseudo kod jasno dočarava generiranje inicijalnih rješenja i postupke lokalne pretrage.

```
1 Rješenje bestbest = null;
2 int bestbestdobrota = -1;
3 for (int i = 0; i < 100; i++) {
4   Rješenje init = generirajRandomRjesenje ();
5    Rješenje best = localSearch (init);
6   int dobrota = fjaDobrote (best);
7    if (dobrota > bestbestdobrota) {
8       bestbest = best;
9        bestbestdobrota = dobrota;
    }
}
```

Kod 4.1. Pseudokod generiranje inicijalnih rješenja

1. Inicijalizira se prazno najbolje rješenje

2. Kako se radi o problemu maksimiziranja, najbolja dobrota se inicijalizira na -1
3. Petlja se ponavlja zadano mnogo puta (ovdje je to 100 iteracija)
4. Generira se slučajno rješenje i postavlja se u inicijalno
5. Provodi se lokalna pretraga nad dobivenim inicijalnim rješenjem
6. Izračunava se dobrota rješenja dobivenog lokalnom pretragom
7. Provjerava se da li je dobivena dobrota bolja od najbolje dobrote u svim iteracijama
8. Postavlja se rješenje kao najbolje rješenje kroz sve iteracije jer je korak 7. bio pozitivan
9. Postavlja se najbolja dobrota na trenutnu dobrotu rješenja dobivenog lokalnom pretragom

Kako bi ovaj algoritam davao dobra rješenja, potrebno je prikladno osmisliti :

1. Generiranje nezavisnih inicijalnih rješenja
2. Lokalnu pretragu
3. Funkciju dobrote

4.1.1. Generiranje nezavisnih inicijalnih rješenja

Krenimo prvo od generiranja nezavisnih inicijalnih rješenja.

Jedna jedinka odnosno rješenje je predstavljena listom koraka u proizvodnji spomenutih prije.

Jedan korak u proizvodnji je lista akcija.

Jedna akcija je kombinacija stroja i radnog naloga.

Jedna akcija znači da se na tome stroju proizvodi taj nalog unutar toga koraka proizvodnje unutar toga rasporeda odnosno rješenja.

Primjer izvođenja na mini primjeru :

RN – kratica za radni nalog

S – kratica za stroj

Prvi korak je dohvatiti iz baze podataka listu svih radnih naloga i njihove pripadajuće puteve po strojevima. Ta lista je prikazana kako slijedi :

RN1 – S1->S2->S3

RN2 – S2->S1->S3

RN3 – S3->S2

RN4-S3->S1

RN5-S1->S2

RN6-S1

Algoritam zatim ulazi u WHILE petlju koja se izvodi dok se iznad prikazana lista ne isprazni.

Odabiremo slučajno RN iz liste i gledamo na kojem stroju se on mora izvesti. Recimo da smo odabrali RN5. On prvo mora ići na stroj 1. Zatim tu akciju (RN5 + S1) stavljamo u prvi korak.

Zatim odabiremo slijedeći slučajni RN iz liste. Recimo RN2. Vidimo da on mora ići na stroj 2. Tu akciju (RN2 + S2) dodajemo u prvi korak.

Recimo da zatim odaberemo RN1. Vidimo da on mora ići na stroj 1. Kako je u ovome koraku stroj 1 već zauzeti od RN5, RN1 ne možemo ubaciti u prvi korak i on mora čekati svoj red.

Recimo da zatim odabiremo RN4. Vidimo da on mora ići na stroj 3. Zatim tu akciju (RN4+S3) dodajemo u prvi korak.

Prvi korak proizvodnje bi zato izgledao ovako :

Lista akcija : (RN5 + S1) , (RN2 + S2) i (RN4 + S3)

Nakon prolaska po svim nalozima i stvaranja koraka proizvodnje odnosno liste akcija, moramo izbaciti te akcije iz Liste u kojoj čuvamo potrebne naloge i putove po strojevima.

RN1 – S1->S2->S3

RN2 – ~~S2~~->S1->S3

RN3 – S3->S2

RN4-~~S3~~->S1

RN5-~~S1~~->S2

RN6-S1

Tako da lista kod početka drugoga koraka izgleda ovako :

RN1–S1->S2->S3

RN2–S1->S3

RN3–S3->S2

RN4-S1

RN5-S2

RN6-S1

Drugi korak proizvodnje se isto tako popunjava akcijama kako je bilo opisano u prvome koraku.

Recimo da je generirani drugi korak imao sljedeću listu akcija :

Lista akcija : (RN1 + S1) , (RN5 + S2) i (RN3 + S3)

Zatim moramo izbaciti te akcije iz Liste u kojoj čuvamo potrebne naloge i putove po strojevima.

RN1–~~S1~~->S2->S3

RN2–S1->S3

RN3–~~S3~~->S2

RN4-S1

RN5-~~S2~~

RN6-S1

Kako je sada RN5 izveden do kraja, njega možemo izbaciti iz liste.

Tako da lista kod početka trećeg koraka izgleda ovako :

RN1–S2->S3

RN2–S1->S3

RN3–S2

RN4-S1

RN6-S1

Treći korak proizvodnje se isto tako popunjava akcijama kako je bilo opisano u prvome koraku.

Recimo da je generirani treći korak imao sljedeću listu akcija :

Lista akcija : (RN4 + S1) , (RN3 + S2)

Zatim moramo izbaciti te akcije iz Liste u kojoj čuvamo potrebne naloge i putove po strojevima.

RN1–S2->S3

RN2–S1->S3

RN3–~~S2~~

RN4-~~S1~~

RN6-S1

Kako su sada RN3 i RN4 izvedeni do kraja, njih možemo izbaciti iz liste.

Vidimo da je stroj 3 u ovome koraku prazan jer nema niti jedan radni nalog koji bi se mogao izvesti na njemu u ovome koraku.

Tako da lista kod početka četvrtog koraka izgleda ovako :

RN1–S2->S3

RN2–S1->S3

RN6-S1

Četvrti korak proizvodnje se isto tako popunjava akcijama kako je bilo opisano u prvome koraku.

Recimo da je generirani četvrti korak imao sljedeću listu akcija :

Lista akcija : (RN6 + S1) , (RN1 + S2)

Zatim moramo izbaciti te akcije iz Liste u kojoj čuvamo potrebne naloge i putove po strojevima.

RN1–~~S2~~->S3

RN2–S1->S3

RN6-~~S1~~

Kako je sada RN6 izveden do kraja, njega možemo izbaciti iz liste.

Vidimo da je stroj 3 u ovome koraku opet prazan jer nema niti jedan radni nalog koji bi se mogao izvesti na njemu u ovome koraku.

Tako da lista kod početka petog koraka izgleda ovako :

RN1–S3

RN2–S1->S3

Peti korak proizvodnje se isto tako popunjava akcijama kako je bilo opisano u prvome koraku.

Recimo da je generirani četvrti korak imao sljedeću listu akcija :

Lista akcija : (RN2 + S1) , (RN1 + S3)

Zatim moramo izbaciti te akcije iz Liste u kojoj čuvamo potrebne naloge i putove po strojevima.

RN1–~~S3~~

RN2–~~S1~~->S3

Kako je sada RN1 izveden do kraja, njega možemo izbaciti iz liste.

Vidimo da je stroj 2 u ovome koraku opet prazan jer nema niti jedan radni nalog koji bi se mogao izvesti na njemu u ovome koraku.

Tako da lista kod početka šestog koraka izgleda ovako :

RN2–S3

Šesti korak proizvodnje se isto tako popunjava akcijama kako je bilo opisano u prvome koraku.

Šesti korak ima sljedeću listu akcija :

Lista akcija : (RN2 + S3)

Nakon izbacivanja RN2–~~S3~~

Početna lista ostaje prazna i generiranje slučajne inicijalne jedinke je završeno.

Tako da je inicijalno generirana slučajna jedinka odnosno rješenje lista koraka koji unutar sebe sadrže akcije.

Lista koraka :

1. Korak : Lista akcija : (RN5 + S1) , (RN2 + S2) i (RN4 + S3)
2. Korak : Lista akcija : (RN1 + S1) , (RN5 + S2) i (RN3 + S3)
3. Korak : Lista akcija : (RN4 + S1) , (RN3 + S2)
4. Korak : Lista akcija : (RN6 + S1) , (RN1 + S2)
5. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)
6. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)

4.1.2. Lokalna pretraga

To rješenje zatim šaljemo u 2. korak *Multistart local search* algoritma odnosno lokalnu pretragu toga inicijalno generiranog rješenja.

Pseudo kod lokalne pretrage :

```
1 Metoda Lokalna pretraga( init : rješenje) {
2     Best = init;
3     int bestDobrota = fjaDobrote(best);
4     while(true) {
5           susjed = generirajNajboljegSusjeda(best)
6           if(susjed == null) {
7           break; // ako niti jedno rješenje iz susjedstva nije
            validno.
            }
8           int dobrota = fjaDobrote(susjed)
9           if(dobrota > bestDobrota) {
10                Best = susjed;
11                bestDobrota = dobrota;
            } else {
12          Break; // trenutni best je najbolji u susjedstvu.
            }
      } }
```

Kod 4.2. Pseudokod lokalne pretrage

1. Lokalna pretraga prima inicijalno rješenje na kojemu se radi lokalna pretraga

2. U trenutno najbolje rješenje se stavlja inicijalno rješenje

3. Trenutno najbolja dobrota je dobrota inicijalnog rješenja

4. Ponavlja se petlja dok se ne dođe do lokalnog optimuma

5. Na temelju trenutno najboljeg rješenja se generira najbolji susjed

6. i 7. Ako je najbolji susjed null onda niti jedno rješenje iz susjedstva nije validno i lokalna pretraga se nalazi u lokalnom optimumu

8. Računa se dobrota najboljeg susjeda

9. Provjerava se da li je dobrota najboljeg susjeda bolja od trenutnog najboljeg rješenja

10. Kako je najbolji susjed bolji od najboljeg rješenja, najbolji susjed postaje najbolje rješenje

11. Kao u koraku 10. samo se ažurira najbolja dobrota

12. Najbolji susjed nije bolji od trenutnog najboljeg rješenja, tako da je najbolje rješenje lokalni optimum pa petlja završava

Lokalna pretraga je poprilični jednostavan postupak gdje generiramo susjedstvo i uzimamo najboljeg susjeda tako dugo dok susjed postoji i ujedno je bolji od trenutno najboljeg rješenja.

Kako bi znali kako se izvodi ova lokalna pretraga potrebno je opisati metodu Generiraj Najboljeg Susjeda.

Susjedstvo je definirano kao zamjena 2 akcije na istome stroju unutar različitih koraka algoritma.

Opišimo to na prethodno generiranom primjeru :

Prvo generiramo slučajan broj od 1 do 6 i zatim slučajan broj od 1 do 3 zbog toga što imamo 6 radnih naloga i 3 stroja.

Recimo da se prvo generirao broj 3 i zatim broj 2.

Lista koraka :

1. Korak : Lista akcija : (RN5 + S1) , (RN2 + S2) i (RN4 + S3)
2. Korak : Lista akcija : (RN1 + S1) , (RN5 + S2) i (RN3 + S3)
3. Korak : Lista akcija : (RN4 + S1) , (RN3 + S2)
4. Korak : Lista akcija : (RN6 + S1) , (RN1 + S2)
5. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)
6. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)

Zatim pokrećemo petlju koja se ponavlja 6 puta.

U prvome prolazu prekopiramo cijelu jedinku u susjeda i zamijenimo akciju na S2 s akcijom na S2 iz 3. koraka koji je bio odabran slučajnim brojem.

Susjed bi u prvome prolazu bio:

Lista koraka :

1. Korak : Lista akcija : (RN5 + S1) , (RN3 + S2) i (RN4 + S3)
2. Korak : Lista akcija : (RN1 + S1) , (RN5 + S2) i (RN3 + S3)
3. Korak : Lista akcija : (RN4 + S1) , (RN2 + S2)
4. Korak : Lista akcija : (RN6 + S1) , (RN1 + S2)
5. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)
6. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)

Susjed bi u drugome prolazu bio:

Lista koraka :

1. Korak : Lista akcija : (RN5 + S1) , (RN2 + S2) i (RN4 + S3)
2. Korak : Lista akcija : (RN1 + S1) , (RN3 + S2) i (RN3 + S3)
3. Korak : Lista akcija : (RN4 + S1) , (RN5 + S2)
4. Korak : Lista akcija : (RN6 + S1) , (RN1 + S2)
5. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)
6. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)

Susjed bi u trećemu prolazu bio preskočen jer bi ponovo nastalo inicijalno rješenje.

Susjed bi u četvrtome prolazu bio:

Lista koraka :

1. Korak : Lista akcija : (RN5 + S1) , (RN2 + S2) i (RN4 + S3)
2. Korak : Lista akcija : (RN1 + S1) , (RN5 + S2) i (RN3 + S3)
3. Korak : Lista akcija : (RN4 + S1) , (RN1 + S2)
4. Korak : Lista akcija : (RN6 + S1) , (RN3 + S2)
5. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)

6. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)

Susjed bi u petome prolazu bio:

Lista koraka :

1. Korak : Lista akcija : (RN5 + S1) , (RN2 + S2) i (RN4 + S3)
2. Korak : Lista akcija : (RN1 + S1) , (RN5 + S2) i (RN3 + S3)
3. Korak : Lista akcija : (RN4 + S1) ,
4. Korak : Lista akcija : (RN6 + S1) , (RN1 + S2)
5. Korak : Lista akcija : (RN2 + S1) , (RN3 + S2) i (RN1 + S3)
6. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)

Susjed bi u šestome prolazu bio:

Lista koraka :

1. Korak : Lista akcija : (RN5 + S1) , (RN2 + S2) i (RN4 + S3)
2. Korak : Lista akcija : (RN1 + S1) , (RN5 + S2) i (RN3 + S3)
3. Korak : Lista akcija : (RN4 + S1) ,
4. Korak : Lista akcija : (RN6 + S1) , (RN1 + S2)
5. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)
6. Korak : Lista akcija : (RN2 + S1) , (RN3 + S2) i (RN1 + S3)

U petome i šestome koraku nismo imali ništa na stroju 2 tako da nismo imali akciju za vratiti na korak 3 već se samo iz 3. koraka prebacilo u 5. odnosno 6. korak.

Kako smo mijenjali redoslijed akcija potrebno je ponovno provjeriti jesu li zadovoljeni svi putovi po strojevima za sve radne naloge.

1. Susjed : ne zadovoljava jer se sada RN3 prvo izvodi na S2 i zatim na S3 dok bi trebalo biti obrnuto
2. Susjed : ne zadovoljava jer unutar koraka 2 postoje 2 akcije za RN3
3. Susjed : preskočen radi jednakosti
4. Susjed : zadovoljava tvrdu granicu prostora stanja
5. Susjed : zadovoljava tvrdu granicu prostora stanja
6. Susjed : zadovoljava tvrdu granicu prostora stanja

Nakon izbacivanja susjeda 1 i 2 zbog ne zadovoljavanja tvrdih granica prostora stanja, računamo funkciju dobrote za susjede 4, 5 i 6. i najboljeg susjeda vraćamo u lokalnu pretragu. Nakon završetka lokalne pretrage vraćamo ga u *multistart local search* kao najboljeg iz ove iteracije.

4.1.3. Funkcija dobrote

Funkcija dobrote izračunava kvalitetu rješenja i prilagođena je specifičnostima problema.

U ovome radu želimo da je broj koraka proizvodnje čim manji jer time osiguravamo brže izvođenje proizvodnje + želimo da slijedne akcije za svaki stroj dijele čim veći broj sastojaka jer time smanjujemo pauze za promjenu sastojaka između koraka proizvodnje.

Ova situacija je specifična za proizvodnju svijeća od voska koja je inspirirala ovaj rad, dok bi za neke druge strojeve i tip proizvodnje vjerojatno bila drugačija.

Zadavanje sastojka smo opisali na početku opisa ove optimizacije.

Funkcija dobrote prolazi po svakome koraku proizvodnje unutar jedinke rješenja.
Za svaki korak i svaki stroj uspoređuje proizvod koji se proizvodi u trenutnome koraku na tome stroju i u sljedećem koraku na tome istome stroju. Na temelju toga zbroj zajedničkih sastojaka se doda u sumu koja čini funkciju dobrote.

Nakon dobivanja sume od sastojaka želi se kazniti duge liste tako da se od dobivene sume oduzme broja korak proizvodnje unutar rješenja. Oduzima se od dobivene sume jer se funkcija dobrote maksimizira, a želi se kazniti duže liste. Alternativno moguće je odabrati i druge funkcije nad brojem koraka proizvodnje zavisno koliki utjecaj želimo da imaju sastojci a koji utjecaj odnosno značaj pridajemo dužini liste. Ova funkcija je odabrana na temelju empirijskog testiranja.

Ako se na određeni datum može provesti samo određeni broj koraka, možemo to također proglasiti kao tvrdu granicu prostora stanja.

4.2. Simulirano kaljenje

Drugi algoritam koji je implementiran kod ovog optimizacijskog problema je Simulirano Kaljenje.

Inspiracija za razvoj algoritma simuliranog kaljenja je postupak kaljenja metala koji se koristi u metalurgiji i kojim se postižu bolja mehanička svojstva metala (poput promjene tvrdoće metala te elastičnosti). Prilikom postupka kaljenja, metal se zagrijava preko kritične temperature koja se neko vrijeme održava i potom se postupno hladi (sporo hlađenje posebno je važno kod materijala poput željeza). Prilikom ovog postupka materijal se najprije dovodi do točke vrlo visoke energije pri kojoj se povećava gibljivost atoma koji se mogu kretati kroz materijal, nakon čega se energija postupno spušta. Zahvaljujući ovom postupku hlađenja i povećanoj gibljivosti atoma, u metalu će se postupno stvoriti pravilne kristalne strukture koje nemaju deformacija i koje metal dovode do stanja minimalne energije. Ako bi se metal hladio prebrzo, nastale bi nepravilne kristalne strukture, pojavile bi se deformacije i naprezanje; sustav bi ostao u višem energetskom stanju - u tom stanju metal bi iskazivao svojstva veće tvrdoće i manje elastičnosti što bi lakše dovelo do kidanja odnosno oštećivanja materijala. Dobro proveden postupak kaljenja povećava elastičnost metala, smanjuje mu tvrdoću i unutarnja naprezanja i stvara pravilnu kristalnu strukturu (Čupić,2013.).

Ista ideja se primjenjuje kod ovog optimizacijskog algoritma. Ideja je da se pomoću prihvaćanja određenih lošijih rješenja koje algoritam generira pokuša izbjeći ulazak u lokalni optimum i pronalazak globalnog optimuma.

U *multistart local seach* algoritmu se generira u 100 iteracija 100 različitih lokalnih optimuma i nadamo se da je jedan on njih dovoljno dobar ili idealno globalni optimum.

Nasuprot tome, u simuliranome kaljenju u svakoj iteraciji se generira slučajni susjed (ne najbolji u susjedstvu kao u lokalnoj pretrazi). Ako je slučajan susjed bolji od trenutnog rješenja onda se uvijek prihvaća, a ako je lošiji onda se prihvaća s određenom vjerojatnošću. Kod početnih nekoliko iteracija je ta vjerojatnost jako velika jer je ideja da algoritam istraži čim širi prostor stanja, dok ta vjerojatnost u daljnjim iteracijama pada jer se želi da algoritam konvergira u određeni optimum. Taj postupak je bio inspiriran postupkom stvarnog simuliranog kaljenja i postupka zagrijavanja i polaganog hlađenja željeza.

Pseudo kod simuliranog kaljenja :

```
1 Rješenje bestbest = null;
2 int bestbestdobrota = -1;
3 double t = 100;
4 Rješenje current = generirajRandomRjesenje ();
5 int currentDobrota = fjaDobrote (current);

6 while (t > 0.0001) {
7   Rješenje randomSusjed = randomSusjed (current);
8    if (randomSusjed == null) { // jer nije zadovoljavao tvrde granica
prostora stanja
9       t = t * 0.99;
10        continue;
    }
11   int susjedDobrota = fjaDobrote (randomSusjed);
12    if (susjedDobrota >= currentDobrota) {
13        current = randomSusjed;
14        currentDobrota = susjedDobrota;
15  } else if (new Random().nextInt(0, 1) < Math.pow(2.71828,-
(currentDobrota-susjedDobrota)/t))   {
16        current = randomSusjed;
17       currentDobrota = susjedDobrota;
    }

 18   if (susjedDobrota > bestbestdobrota) {
 19       bestbest = randomSusjed;
 20       bestbestdobrota = susjedDobrota;
    }

 21   t = t * 0.85;
}
```

Kod 4.3. Pseudokod simuliranog kaljenja

1. Inicijalizira se najbolje rješenje

2. Kako se radi o problemu maksimiziranja, najbolja dobrota se postavlja na -1

3.Postavlja se početna temperatura

4.Generira se trenutno slučajno rješenje

5.Računa se dobrota trenutnog rješenja

6. Petlja se ponavlja dok se temperatura ne približi 0

7. Generira se slučajan susjed od trenutnog rješenja

8./9./10. Ako je generirani ne valjani slučajni susjed onda se samo minimalno smanjuje temperatura i nastavlja se petlja

11. Računa se dobrota susjeda

12. Provjerava se da li je susjedova dobrota veća od trenutne dobrote

13. Ukoliko je, onda se postavlja susjed kao trenutno rješenja

14. Postavlja se i trenutna dobrota na susjedovu dobrotu

15.Ukoliko susjedova dobrota nije bolje od trenutne dobrote onda s određenom vjerojatnošću radi se isto kao u koracima 13. i 14.

16. isti korak 13.

17. isti korak 14.

18. Provjerava se da li je susjedno rješenje bolje od trenutno najboljeg rješenja pronađenog

19. Ukoliko je, onda se postavlja susjed na najbolje rješenje

20. Postavlja se i susjedova dobrota na najbolju dobrotu.

21. Ažurira se temperatura

Korištena je početna temperatura 100 i minimalna temperatura 0.0001. Korišten je geometrijski plan hlađenja t = t * 0.85. To su parametri koji upravljaju algoritmom odnosno određuju koliko dugo će se algoritam izvoditi i s kolikom vjerojatnošću će se prihvaćati loša rješenja.

Ovi parametri su postavljeni radi lakšeg objašnjenja simuliranog kaljenja na okruglim brojkama. U sljedećim poglavljima će biti prikazano detaljno testiranje i izračun najboljih parametara simuliranog kaljenja.

Mali primjer opisa rada simuliranog kaljenja :

Početna temperatura = 100.

Recimo da je currentDobrota = 50.

Generirani susjed ima dobrotu = 40.

Za parametre :

T = 100

currentDobrota = 50

susjedDobrota = 40

računamo e^-(currentDobrota-susjedDobrota)/T odnosno e^-(50-40)/100 = e^-10/100 = 0.9048 To znači da toga susjeda koji je lošiji prihvaćamo s vjerojatnošću od preko 90%. To je i prikladno jer je tek početak algoritma i želimo ići u širinu.

Zatim temperaturu smanjimo t = t*0.85 i dobimo t = 85.

Za parametre :

T = 85

currentDobrota = 50

susjedDobrota = 40

računamo e^-(currentDobrota-susjedDobrota)/T odnosno e^-(50-40)/85 = e^-10/85 = 0.889 To znači da toga susjeda koji je lošiji prihvaćamo s vjerojatnošću od ~89%. Vidimo da geometrijsko upravljanje temperaturom smanjuje vjerojatnost za prihvaćanje lošijih rješenja.

I tako polako smanjujemo vjerojatnost sve dok ne konvergiramo ka nekome lokalnome optimumu.

U ovome algoritmu koristimo i 3 druge funkcije : generirajRandomRjesenje, randomSusjed i fjaDobrote.

GenerirajRandomRjesenje i fjaDobrote smo već upoznali kod opisivanja *multistart local search* i one su identične. Jedina nova metoda je RandomSusjed.

4.2.1. Generiranje slučajnog susjeda

Opišimo metodu randomSusjed na jedinki koju smo generirali u mini primjeru generiranju inicijalnog rješenja odnosno :

Lista koraka :

1. Korak : Lista akcija : (RN5 + S1) , (RN2 + S2) i (RN4 + S3)
2. Korak : Lista akcija : (RN1 + S1) , (RN5 + S2) i (RN3 + S3)
3. Korak : Lista akcija : (RN4 + S1) , (RN3 + S2)
4. Korak : Lista akcija : (RN6 + S1) , (RN1 + S2)
5. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)
6. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)

Odabiremo 2 slučajna koraka unutar liste koraka.

Recimo da su se odabrali korak 1 i 2.

Zatim odabiremo slučajan stroj od svih mogućih strojeva. Recimo stroj 2.

Zatim zamijenimo u odabranim koracima, akcije koje se izvode na odabranome stroju u ovome slučaju je to stroj 2.

Lista koraka :

1. Korak : Lista akcija : (RN5 + S1) , (RN5 + S2) i (RN4 + S3)
2. Korak : Lista akcija : (RN1 + S1) , (RN2 + S2) i (RN3 + S3)
3. Korak : Lista akcija : (RN4 + S1) , (RN3 + S2)
4. Korak : Lista akcija : (RN6 + S1) , (RN1 + S2)
5. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)
6. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)

Ukoliko neki korak nema akciju na odabranome stroju onda se ne upisuje ništa u susjedni stroj. Pogledajmo primjer.

Odabrani koraci iz inicijalnog rješenja su : 4 i 5. Odabrani stroj je 3.

Lista koraka :

1. Korak : Lista akcija : (RN5 + S1) , (RN2 + S2) i (RN4 + S3)
2. Korak : Lista akcija : (RN1 + S1) , (RN5 + S2) i (RN3 + S3)
3. Korak : Lista akcija : (RN4 + S1) , (RN3 + S2)
4. Korak : Lista akcija : (RN6 + S1) , (RN1 + S2) i (RN1 + S3)
5. Korak : Lista akcija : (RN2 + S1)
6. Korak : Lista akcija : (RN2 + S1) , (RN1 + S3)

Vidimo da se iz 5. koraka prebacila akcija na 4. korak dok se na 5. korak ništa nije prebacilo jer nema akcije na 3. stroju u 4. koraku.

Nakon takvog generiranja susjeda, provjeravamo njegovo zadovoljavanje tvrdih granica prostora stanja kao što sam opisao već prije. Ako zadovoljava ga vraćamo inače vraćamo null i postupamo po opisu u pseudo kodu.

4.2.2. Određivanje parametara algoritma za simulirano kaljenje

Posljedica No-free-lunch teorema jest da je ovaj algoritam, kao i svi ostali, u prosjeku jednako dobar kao i svi ostali. Kako bi algoritam simuliranog kaljenja dobro rješavao neki konkretan problem, potrebno je dobro postaviti parametre algoritma čime algoritam

prilagođavamo problemu (Čupić, 2013.). U algoritmu simuliranog kaljenja postoje 2 parametra koja utječu na izvedbu algoritma, a to su početna temperatura i plan hlađenja. U literaturi se najviše koristi geometrijski plan hlađenja s vrijednostima iz intervala od 0.5 do 0.99 (Čupić,2013.). Geometrijski plan hlađenja smanjuje temperaturu pomoću formule x = x* vrijednost iz intervala. Početne temperature su zavisne o problemu i dobro je isprobati široku paletu temperatura.

U prvome testu testirani su planovi hlađenja s vrijednostima 0.5,0.75,0.85,0.95 i temperature 100,200,300,400,500,600,700,800,900,1000,2000,3000,4000,5000,6000,7000, 8000, 9000 i 10000. Provedeni su testovi sa svim kombinacijama planova hlađenja i početnih temperatura (100,0.5),(100,0.75),(100,0.85),(100,0.95).(200,0.5),(200,0.75)…

Za svaku kombinaciju se testirala izvedba na 3 jednake postavke problema. Jedna mala s 5 radnih naloga, jedna srednja s 20 radnih naloga i jedna velika sa 60 radnih naloga. Dobivene dobrote na svim instancama su zbrojene. Dodatno zbrojena su i sva vremena izvođenja u milisekundama.

Izdvojeni zanimljivi rezultati :

Tablica 4.1. Rezultati prve iteracije testova

Početna temperatura	Plan hlađenja-vrijed.	Dobivena dobrota	Vrijeme izvođenja
7000	0.95	1613	150406
4000	0.95	1609	154360
8000	0.95	1607	136678
100	0.85	1607	43423
…	…	…	…
4000	0.5	1580	13280

Četiri najbolja rezultata i jedan najgori su prikazani u tablici. Najzanimljiviji su naravno najbolji parametri odnosno 7000 i 0.95. Također su zanimljivi i 100 i 0.85 jer daju dobre rezultate za 3x brže izvođenje.

U prvome testiranju su bila testirana različita područja vrijednosti parametara.

Kako bi se unutar područja vrijednosti koja su dala dobre rezultate pronašlo specifične kvalitetne vrijednosti potrebno je provesti još testiranja.

Druga iteracija testova je pokrenuta na sljedećim temperaturama 80,85,90,95,100,105,110,115,120,6500,6600,6700,6800,6900,7000,7100,7200,7300,7400, 7500 u kombinaciji s vrijednostima planova hlađenja : 0.85,0.90,0.95.

Izdvojeni zanimljivi rezultati :

Tablica 4.2. rezultati druge iteracije testova

Početna temperatura	Plan hlađenja-vrijed.	Dobivena dobrota	Vrijeme izvođenja
7200	0.95	1208	131677
7300	0.95	1204	112874
7400	0.9	1204	72134
7000	0.95	1203	149461
…	…	…	…
7400	0.85	1178	42368

Najbolji rezultati su se pokazali oni s visokom temperaturom i sporim planom hlađenja. Rezultati pokazuju da je prostor rješenja jako velik te da niska temperatura i brzi planovi hlađenja prebrzo konvergiraju ka lokalnome optimumu dok sporiji lakše istraže širi prostor stanja i nađu bolja rješenja.

Kako bi se zaključilo koji su najbolji parametri, provodi se i 3. odnosno zadnje testiranje na dobrotama između 7200 i 7300 s planom hlađenja oko 0.95.

. Pokrenuta je i treća iteracija testova na sljedećim temperaturama 7200,7220,7240,7260,7280,7300,7320,7340 u kombinaciji s vrijednostima planova hlađenja : 0.94,0.95,0.96.,0.97,0.98 i 0.99

Izdvojeni zanimljivi rezultati :

Tablica 4.3. Rezultati treće iteracije testova

Početna temperatura	Plan hlađenja	Dobivena dobrota	Vrijeme izvođenja

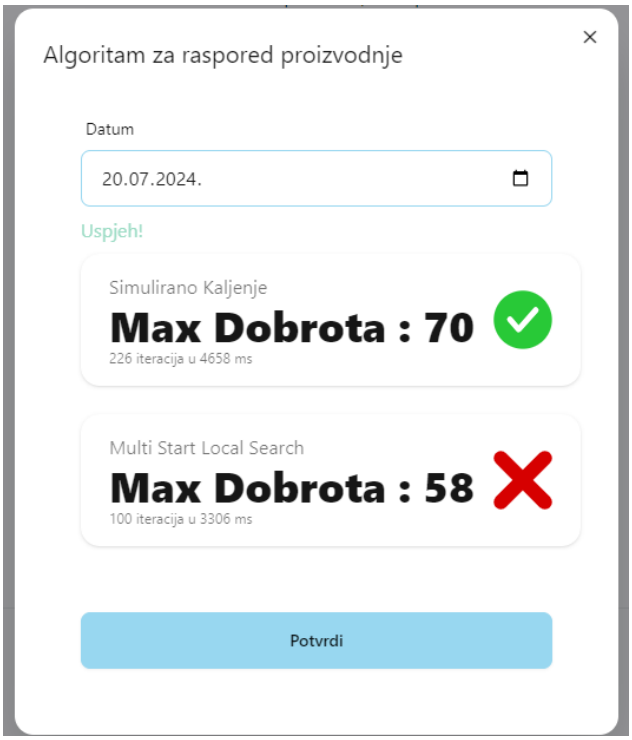
7300	0.94	1138	105955
7320	0.95	1134	95364
…	…	…	…
7280	0.96	1111	131244

Najbolji rezultat je postigla početna temperatura 7300 i plan hlađenja s faktorom 0.94.

Zaključak je da kako bi ovaj algoritam producirao najbolje rezultate, povoljno je postaviti početnu temperaturu na 7300 i plan hlađenja na faktor 0.94.

4.3. Prikaz rješenja na sučelju

Nakon što završe oba algoritma, i *Multistart local search* i simulirano kaljenje za najbolje pronađeno rješenje iz oba algoritma ispitamo funkciju dobrote i bolje rješenje zapišemo kao trenutni raspored. U sučelju je to prikazano kao na slici 4.4.



Slika 4.4. Prikaz rješenja algoritama za optimizaciju rasporeda proizvodnje

Rješenje algoritma služi za raspored naloga po strojevima koje je u sučelju prikazano kao na slici 4.5. Prvo je napisano ime stroja i u zagradi koji se po redu proizvodi taj nalog na tome stroju.

152	2024-07-20	Stroj 2 (6.)	Mix lampiona 1 i 2 4+4 u kartonu na europaleti do 1000mm
111	2024-07-20	Stroj 1 (1.) -Stroj 3 (3.) -Stroj 2 (8.) -Stroj 0 (9.)	Lampion 10 13/1 europaleta do 1000mm
110	2024-07-20	Stroj 2 (2.) -Stroj 3 (6.) -Stroj 0 (7.) -Stroj 1 (9.)	Lampion 9 15/1 europaleta do 1000mm
109	2024-07-20	Stroj 1 (8.)	Lampion 8 12/1 europaleta do 1000mm
108	2024-07-20	Stroj 0 (2.) -Stroj 1 (5.) -Stroj 2 (7.)	Lampion 7 33/1 europaleta do 1000mm
107	2024-07-20	Stroj 3 (1.) -Stroj 1 (2.)	Lampion 6 30/1 europaleta do 1000mm
106	2024-07-20	Stroj 3 (5.) -Stroj 0 (6.)	Lampion 5 28/1 europaleta do 1000mm
105	2024-07-20	Stroj 1 (7.) -Stroj 2 (9.)	Lampion 4 30/1 europaleta do 1000mm
104	2024-07-20	Stroj 2 (5.) -Stroj 1 (6.) -Stroj 3 (7.) -Stroj 0 (8.)	Lampion 3 9/1 europaleta do 1000mm
103	2024-07-20	Stroj 1 (3.) -Stroj 0 (4.)	Lampion 2 8/1 europaleta do 1000mm
102	2024-07-20	Stroj 1 (4.) -Stroj 0 (5.)	Lampion 1 9/1 europaleta do 1000mm

Slika 4.5. Prikaz rasporeda algoritama po strojevima.

4.4. Testiranje

Kako bi znali ocijeniti uspješnost implementiranih algoritama i odrediti koji je bolji na ovome algoritmu, potrebno je obaviti testiranje. Obavljeno je 200 iteracija testiranja pomoću sljedeće skripte opisane pseudo kodom.

Skripta emulira postupak stvaranja naloga kroz sučelje.

```
Funkcija „stvori naloge za neki dan“ () {
      broj_nalog = random(2,100)
      for ( i do broj_nalog) {
            stvori nalog za slučajno odabrani proizvod sa slučajno
      odabranom količinom.
            stvori raspored naloga po strojevima tako da svaki stroj
            dodaješ u raspored s vjerojatnošću od 50%
            stvori sastojke za nalog na temelju slučajnog odabira
            sastojka iz svih potrebnih grupa proizvoda, količine odaberi
            slučajno
      }
}
```

Prethodno opisana funkcija je bila pokrenuta 200 puta. Na računalu „Lenovo Thinkpad X1 Extreme“ iz 2019. godine se to stvaranje naloga i zatim optimizacija dvama optimizacijskim procesima 200 puta izvodilo oko 8 sati.

Statistika generiranih podataka :

Prosječan broj radnih naloga po generiranju : 54 što je očekivano za random funkciju

Prosječan broj strojeva u putanji po generiranju : 2.49

Dobiveni su sljedeći rezultati :

Tablica 4.4. Rezultati testova optimizacijskih algoritama

	Simulirano Kaljenje	*Multistart local search*
Prosječna funkcija dobrote	749.66	748,55
Pobjednik	65 puta	77 puta (izjednačeno 58 puta)
Prosječna razlika između dobrota kada je on pobijedio	6,12	2,27
Prosječno vrijeme izvođenja u ms	30 209	70 987

Kao što se vidi iz rezultata oba algoritma su skoro identična i oba daju dobre rezultate koji su skoro optimalni. To je rezultat njihove robusnosti i dobrog podešavanja parametara simuliranog kaljenja.

Prosječne funkcije dobrote preko svih iteracija su skoro jednake, razlikuju se tek za cca 0.01 % što je zanemarivo.

Unatoč tome, vidi se iz podataka da je *Multistart local search* bio bolji u 12 više slučajeva, ali je razlika kada je on pobijedio bila prosječno samo 2,27 dok je razlika u dobrotama kada je simulirano kaljenje pobijedilo bila puno veća i to 6,12.

Također vidimo da se *Multistart local search* izvodio više nego duplo duže u prosjeku od simuliranog kaljenja odnosno 70 987ms prema 30 209ms.

Može se zaključiti da je *multistart local search* robusniji jer pobjeđuje u većem brojem slučaju, ali kada simulirano kaljenje pobjedi onda zna imati poprilično bolje rezultate od *Multistart local searcha.* Unatoč tome, simulirano kaljenje ima drastično bolje vrijeme

izvođenja i to prosječno samo 40% dužine izvođenja *multistart local search.* Ako situacija zahtjeva brzinu izvođenja kao ključno svojstvo algoritma onda je povoljno odabrati simulirano kaljenje.

Zaključak

Sustavi za optimizaciju proizvodnje i upravljanje proizvodnim procesom predstavljaju ključan korak ka povećanju učinkovitosti, smanjenju troškova i poboljšanju kvalitete u suvremenoj industriji.

U ovome radu implementirana su 2 algoritma za optimizaciju rasporeda radnih naloga. Na temelju *no free lunch* teorema znamo da je potrebno pogoditi algoritam / parametre koji su prikladni za dati problem jer su svi algoritmi pretraživanja prosječno jednako dobri zbog *no free lunch* teorema. Zato su odabrani i uspješno implementirani povijesno relevantni algoritmi iz dvaju velikih područja : *multistart local search* i simulirano kaljenje.

Parametri simuliranog kaljenja su uspješno optimizirani nizom provedenih testova. Testiranjem je zaključeno da je prostor rješenja vrlo velik i da bolje rezultate daju više temperature i sporiji planovi hlađenja jer oni omogućuju prihvaćanje lošijih rješenja s većom vjerojatnošću i samim time sporije konvergiranje ka lokalnome optimumu. Niže temperature i brži planovi hlađenja brzo konvergiraju ka lokalnome optimumu i zato su pružali puno lošija rješenja.

Na temelju provedenih testiranja može se zaključiti da oba algoritma pružaju zadovoljavajuću razinu optimizacije rasporeda te da su prikladni za praktičnu primjenu. *Multistart local search* je robusniji i daje bolja rješenja u većem broju primjera, dok simulirano kaljenje ima brže vrijeme izvođenja.

Oba algoritma ubrzavaju poslovne procese unutar poslovnog sustava. Sustav je samim time efikasniji što zbog smanjenja radnog opterećenja radne snage, a što zbog efikasnijeg stvaranja rasporeda što sam čovjek ne bi mogao zbog prevelikog prostora rješenja

Iako je aplikacija zadovoljila sve funkcionalne zahtjeve, postoji puno mogućnosti za napredak. Jedna mogućnost bi bila implementirati dodatne algoritme za postojeće probleme raspoređivanja npr. Genetski algoritam i procijeniti jesu li bolji. Druga mogućnost bi bila podržati kompleksnije proizvodne procese npr. da koraci proizvodnje po strojevima nisu sinkroni i da nemaju jednako vrijeme izvođenja.

Literatura

Hall, S. N. (2012). *A Group Theoretic Tabu Search Approach to the Traveling Salesman Problem*. Preuzeto 4.6 2024 iz https://amazon.com/theoretic-approach-traveling-salesman-problem/dp/1249584426

Said, G. A.-N., Mahmoud, A. M., & El-Horbaty, E.-S. M. (2014). A Comparative Study of Meta-heuristic Algorithms for Solving Quadratic Assignment Problem. *International Journal of Advanced Computer Science and Applications, 5*(1). Preuzeto 4. 6 2024 iz https://arxiv.org/pdf/1407.4863

White, C., & Yen, G. G. (2004). *A hybrid evolutionary algorithm for traveling salesman problem*. Preuzeto 4.6 2024 iz http://cdn.intechopen.com/pdfs/12404/intech-a_fast_evolutionary_algorithm_for_traveling_salesman_problem.pdf

Yang, X.-S. (2009). Harmony Search as a Metaheuristic Algorithm. *arXiv: Optimization and Control*, 1-14. Preuzeto 5.6 2024 iz https://link.springer.com/chapter/10.1007/978-3-642-00185-7_1

Johnson, D. S., Papadimitriou, C. H., & Yannakakis, M. (1988). How easy is local search?. *Journal of computer and system sciences*, *37*(1), 79-100. Preuzeto 6.6.2024. iz https://www.sciencedirect.com/science/article/pii/0022000088900463

Adam, S. P., Alexandropoulos, S. A. N., Pardalos, P. M., & Vrahatis, M. N. (2019). No free lunch theorem: A review. *Approximation and optimization: Algorithms, complexity and applications*, 57-82. Preuzeto : 9.6.2024. iz https://link.springer.com/chapter/10.1007/978-3-030-12767-1_5

Čupić (2013). Prirodom inspirirani algoritmi, Preuzeto : 10.6.2024 iz http://java.zemris.fer.hr/nastava/pioa/knjiga-0.1.2013-12-30.pdf.

w3schools (2024.) Canvas html, Prezeto : 12.6.2024. iz https://www.w3schools.com/html/html5_canvas.asp

Sažetak

Sustav za optimizaciju proizvodnje i upravljanje proizvodnim procesom

Ovaj rad razmatra izradu sustava za optimizaciju proizvodnje i upravljanje proizvodnim procesima. Rad u prvome poglavlju predstavlja osnovne pojmove optimizacijskih algoritama, opseg razmatranog praktičnog rješenja i pregled nekih optimizacijskih algoritama. U drugome poglavlju opisana je korištena tehnologije i arhitektura sustava. U trećem poglavlju rad predstavlja zanimljive implementacijske detalje. U četvrtom poglavlju rad predstavlja detaljan prikaz optimizacijskih rješenja i njihovu evaluaciju. U zaključku rad zaključuje da su uspješno implementirana 2 kvalitetna i primjenjiva optimizacijska rješenja.

Ključne riječi: Optimizacija, simulirano kaljenje, višekratno pokretanje lokalne pretrage, proizvodnja

Summary

System for production optimization and production process management

This paper considers the creation of a system for production optimization and management of production processes. The paper in the first chapter presents the basic concepts of optimization algorithms, the scope of the considered practical solution and an overview of some optimization algorithms. In the second chapter, the used technologies and system architecture are described. In the third chapter, the paper presents interesting implementation details. In the fourth chapter, the paper presents a detailed presentation of optimization solutions and their evaluation. In conclusion, the paper concludes that 2 high-quality and applicable optimization solutions have been successfully implemented.

Keywords : Optimization, simulated annealing, multistart local search, production