

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IAȘI
FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

**Aplicație web pentru generarea și vizualizarea
orarului**

propusă de

Ioan-Paul Samson

Sesiunea: iunie, 2023

Coordonator științific

Lect. Dr. Frasinăru Cristian

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IAȘI
FACULTATEA DE INFORMATICĂ

**Aplicație web pentru generarea și
vizualizarea orarului**

Ioan-Paul Samson

Sesiunea: iunie, 2023

Coordonator științific

Lect. Dr. Frasinăru Cristian

Avizat,
Îndrumător lucrare de licență,
Lect. Dr. Frasinăru Cristian.

Data:

Semnătura:

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Samson Ioan-Paul** domiciliat în **România, jud. Neamț, Sat. Bâra (Com. Bâra)**, născut la data de **30 iunie 2001**, identificat prin CNP **5010630270823**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2018, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Aplicație web pentru generarea și vizualizarea orarului** elaborată sub îndrumarea domnului **Lect. Dr. Frasinăru Cristian**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data:

Semnătura:

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Aplicație web pentru generarea și vizualizarea orarului**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea "Alexandru-Ioan Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Ioan-Paul Samson**

Data:

Semnătura:

Rezumat

În această lucrare vom prezenta problema generării orarelor pentru universitate, folosind diferiți algoritmi pentru rezolvarea problemelor de colorare a grafurilor. Problema colorării unui graf este o problemă cunoscută, în care avem la dispoziție un graf $G = (V, E)$, ce conține o mulțime de noduri V și o mulțime de muchii între acestea E , și o mulțime de culori C . Rezolvarea problemei constă în atribuirea fiecărui nod $v \in V$, astfel încât oricare două noduri ce sunt legate printr-o muchie să aibă culori diferite (i. e. avem funcția $c : V \rightarrow C$, pentru care $c(v_1) \neq c(v_2)$, $\forall v_1, v_2$ astfel încât $v_1 v_2 \in E$) [1]. Problemele de generare a orarului sunt clasificate în două tipuri: probleme de generare a orarului pentru cursuri și probleme de planificare a examenelor. Problema pe care ne-o propunem să o rezolvăm este generarea orarului pentru cursuri, iar fiecare curs trebuie programat într-un interval de timp dint-o anumită zi a săptămânii. În problemă avem la dispoziție date despre studenți, profesori, săli disponibile și orele ce trebuiesc asignate. Pentru rezolvarea acestui tip de problemă, există mai multe constrângeri ce trebuiesc rezolvate, printre acestea fiind constrângeri între grupuri de studenți ce urmează același curs, constrângeri pentru profesori ce predau mai multe cursuri și constrângeri legate de disponibilitatea sălilor de curs. Pentru relațiile între cursurile ce trebuiesc asignate la diferite intervale de timp și constrângerile dintre aceste vom reduce problema generării orarului la o problemă cunoscută, cea de colorare a grafurilor. Vom folosi mai multe tipuri de reprezentări pentru aceste grafuri, și diferiți algoritmi de colorare pentru a rezolva problemele. Scopul principal al proiectului este generarea orarelor valide, cu cât mai puține coliziuni între cursurile asignate. Un alt scop al proiectului este implementarea unor algoritmi eficienți și a unei interfețe web în care să se poate atât vizualiza, cât și modifica anumite elemente ale orarului generat. Algoritmii ce sunt implementați vor fi detaliați ulterior, iar în final vom trage o concluzie legată de eficiența lor în generarea soluțiilor cât și o comparație între implementări. Deoarece problema colorării unui graf este NP-completă, din punct de vedere computațional sunt și niște limitări legate de numărul de cursuri pe care algoritmii îl vor putea procesa.

Cuprins

| | | |
|----------|--|-----------|
| 1 | Introducere | 2 |
| 1.1 | Problema | 2 |
| 1.2 | Motivație | 3 |
| 1.3 | Tehnologii folosite | 3 |
| 2 | Specificații funcționale | 4 |
| 2.1 | Problema, pe larg | 4 |
| 2.2 | Flux de lucru | 5 |
| 3 | Arhitectura sistemului | 8 |
| 3.1 | Schema bazei de date | 8 |
| 3.2 | Diagramele datelor | 11 |
| 4 | Implementare și testare | 14 |
| 4.1 | Implementare | 14 |
| 4.1.1 | Consecutive Room Coloring | 14 |
| 4.1.2 | Day-Time-Room Coloring Greedy | 16 |
| 4.1.3 | Day-Time-Room Coloring DSatur | 18 |
| 4.1.4 | Two-Step Day-Time Room Coloring Greedy | 19 |
| 4.1.5 | Two-Step Day-Time Room Coloring DSatur | 22 |
| 4.2 | PartialCol Optimisation | 23 |
| 4.3 | Testare | 24 |
| 5 | Manual de utilizare | 26 |
| 5.1 | Instalare | 26 |
| 5.2 | Utilizare | 27 |
| 6 | Concluzii și direcții viitoare | 29 |
| 6.1 | Concluzii | 29 |
| 6.2 | Direcții viitoare | 29 |
| 7 | Bibliografie | 29 |

1 Introducere

1.1 Problema

Originea teoriei grafurilor datează încă de la faimoasa problemă a podurilor din Königsberg, concepută de matematicianul Leonhard Euler, în 1736. [2]. De atunci și până în ziua de astăzi au aparut multe probleme a căror bază este teoria grafurilor, iar printre aplicațiile acestora se numără și domeniul Informaticii. Una dintre problemele din teoria grafurilor, pe care această lucrare o va aborda, este problema colorării unui graf pe noduri.

Problema colorării unui graf pe noduri reprezintă conceptul matematic de separare a unei mulțimi în diferite submulțimi, în funcție de un set de constrângeri. Aceste submulțimi conțin elementele din mulțimea inițială care, după rezolvarea problemei, au aceeași clasă (culoare). În aplicații, avem la dispoziție un graf G , unde $V(G) = \{v_1, v_2, \dots, v_n\}$ este o mulțime de noduri, iar $E(G) = \{\{x, y\} | x, y \in V(G) \text{ \& } x \neq y\}$ o mulțime de muchii, și o mulțime de culori $C = \{c_1, c_2, \dots, c_k\}$, și trebuie să găsim o **colorare** $\varphi : V(G) \rightarrow C$, unde $\varphi(x) \neq \varphi(y), \forall x, y \in E$ (i.e. pentru orice muchie, nodurile ce o compun vor avea culori diferite).

În problema generării orarului, avem o mulțime de cursuri / seminarii / laboratoare / examene care pot avea resurse comune unele cu altele, pe care le vom numi și **evenimente**, iar rezolvarea ei reprezintă asignarea lor la un număr cât mai mic de intervale de timp, astfel încât nicio resursă necesară unui eveniment să fie necesară altui eveniment în același interval de timp. Într-o universitate, tipurile de resurse de care pot avea nevoie, în același timp, mai multe evenimente sunt studenți (grupe de studenți), profesori și săli de curs.[3] Faptul că două evenimente nu pot fi asignate la aceiași studenți, profesori sau săli de curs impune constrângeri asupra asignării cursurilor.

În procesul de automatizare al generării orarului se definesc astfel două clase de constrângeri: *constrângeri tari* și *constrângeri slabe*. Constrângerile slabe sunt mai puțin necesare de satisfăcut decât cele tari care, în multe idei de rezolvare, sunt considerate obligatorii de satisfăcut. O soluție e reprezentată, de obicei, de faptul că toate constrângerile tari sunt satisfăcute. Constrângerile slabe sunt satisfăcute dacă este posibil, ele fiind necesare pentru anumite optimizări ale orarului. [4].

Problema generării orarului pentru universitate se separă în două mari categorii: problema asignării cursurilor (și a seminariilor/laboratoarelor) într-un mod cât mai eficient, și problema planificării examenelor. În cadrul acestei lucrări am tratat problema asignării cursurilor.

În continuare vom folosi termenul de **eveniment** pentru a ne referi la un curs/seminar/laborator ce trebuie asignat în orar.

Constrângerea fundamentală în generarea orarului este constrângerea binară în care două evenimente au o resursă comună. De aici putem deduce că dacă un student (sau orice alt tip de resursă, i.e. profesor, sală de curs) trebuie să fie prezentă pentru două evenimente diferite, acestea trebuie să fie asignate la intervale de timp diferite, căci altfel studentul (resursa) respectiv ar trebui să fie în două locuri în același timp. Aceste constrângeri sunt prezente în orice tip de problemă de generare a orarului, astfel putem reduce problema generării orarului la o problemă de colorare a nodurilor unui graf, în care nodurile sunt evenimentele ce trebuie asignate, muchiile dintre noduri reprezintă două evenimente ce

folosesc o resursă comună, iar nodurile sunt, de obicei intervalele de timp disponibile pentru asignare. [4].

În algoritmi ce vor fi prezentați ulterior, vom trata mai multe cazuri pentru colorare: un caz este când evenimentele au deja sălile asignate, astfel culorile vor fi reprezentate doar de săli, iar alt caz ar fi când evenimentele nu au nici sălile, nici intervalele de timp asignate, astfel culorile vor fi reprezentate de produsul cartezian între mulțimea sălilor disponibile R și mulțimea intervalelor de timp T : $R \times T$ (intervalele de timp sunt, la rândul lor, un produs cartezian între zilele în care putem pune cursuri și intervalele orare din fiecare zi).

1.2 Motivație

Generarea orarelor nu este folosită doar în domeniul educațional. Orarele pot fi folosite în vizualizarea programelor pentru transportul public, a programelor în fabrici, a programelor avioanelor. Acestea fiind spuse, în contextul universitar, generarea unei soluții de calitate pentru problema creării orarului este o sarcină dificilă și complexă din cauza numărului de constrângeri dintre evenimente.

Un orar bun este necesar pentru performanța oricărei universități. Astfel generarea orarului pentru a evide coliziunile între evenimente diferite este obligatorie. Satisfacerea diverselor constrângeri între evenimente trebuie tratată cât mai bine, astfel încât generarea să fie folositoare atât pentru studenții care participă la evenimente, cât și pentru profesorii care le predau.

Datorită constrângerilor ce pot exista în generarea unui orar, această problemă este NP-completă în multe din variantele în care e propusă. Cooper și Kingston (1996), de exemplu, au demonstrat faptul că această problemă este NP-completă pentru mai multe tipuri de implementări ce pot apărea. Ei au dovedit asta prin reducerea polinomială la diverse probleme NP-complete, printre care este și colorarea unui graf. [4]

Rezolvarea problemei colorării unui graf s-a dovedit necesară într-o multitudine de aplicații ale teoriei grafurilor dar și pentru aplicații în viața reală, cum ar fi colorarea unei hărți, în care oricare două țări învecinate au culori diferite.

Din acest motiv, vom aborda problema generării orarului reducând-o la o problemă de colorare a unui graf. Vom reprezenta evenimentele ca noduri în graf, iar muchiile vor fi adăugate în funcție de constrângerile dintre acestea.

1.3 Tehnologii folosite

Pentru implementarea algoritmilor pe care îi vom folosi pentru a rezolva problema, am ales să folosim Java. În limbajul de programare Java, programele sunt definite de clase. Clasele sunt o parte importantă a paradigmei Programării Orientată-Obiect, ele fiind ca și schițele după care sunt construite obiecte. [5]. Reprezentarea datelor cu care vom lucra drept obiecte ale unei clase este folositoare pentru vizualizarea datelor, salvarea lor într-o bază de date și transpunerea lor în diverse formate ce pot fi ulterior parsate (JSON, XML).

Pe lângă partea algoritmică necesară pentru generarea orarului, avem nevoie și de un mod de a salva, vizualiza și exporta datele.

Pentru salvarea datelor am ales să folosim o bază de date relațională, anume PostgreSQL. Folosim o bază de date relațională deoarece acest tip de reprezentare este folositor în cazul reprezentării datelor

orarului, ele fiind strâns legate între ele (evenimentele trebuie să folosească diferite resurse: studenți, profesori, săli, toate aceste componente fiind necesare să fie stocate).

În cazul aplicației Backend, am ales să folosim framework-ul Java Spring Boot datorită ușurinței cu care putem modela microserviciile. Microserviciile au devenit un stil arhitectural popular de proiectare a serviciilor oferite de o aplicație. Ele sunt independente una față de cealaltă, fiecare oferind o funcționalitate diferită și sunt folosite, în general, pentru a oferi multe soluții în rezolvarea unei probleme [6]. Astfel, o aplicație Spring Boot face folosirea unei paradigme derivate din MVC (model-view-controller) ușoară. De asemenea, în tehnologia Java, avem la dispoziție o serie de unelte cu care putem comunica ușor cu baza de date Postgres, cum ar fi framework-ul Hibernate, o unealtă pentru maparea obiectelor în baza de date, și librăria specifică pentru comunicarea cu baza de date Postgres.

În cazul aplicației Frontend, am ales să folosim framework-ul AngularJS. În AngularJS, aplicațiile web își creează codul HTML asamblându-se cu datele de pe serverul de Frontend, astfel rolul serverului este de a trimite resursele statice pentru templateuri și să ofere datele când sunt cerute [7]. Pentru a oferi resursele și a crea logica prelucrării lor, AngularJS se folosește de TypeScript, un superset al JavaScript. TypeScript, după cum sugerează numele, este un limbaj hard-typed (i.e. oblică ca variabilele să aibă un tip de date) și se folosește de clase. Astfel, datele ce sunt trimise de la aplicația Backend pot fi mapate ușor ca niște obiecte ale unei clase, și pot fi vizualizate în pagină, ulterior procesării.

2 Specificații funcționale

2.1 Problema, pe larg

După cum am spus, problema generării orarului pentru universitate, în cazul subproblemei asignării cursurilor, avem mai multe tipuri de constrângeri tari ce trebuie rezolvate obligatoriu, și o serie de constrângeri slabe. În problema generării orarului, evenimentele au ca participanți unul sau mai multe grupuri de studenți ce trebuie să le urmeze precum și unul sau mai mulți profesori ce le predau. De asemenea, avem la dispoziție o serie de resurse, adică sălile de curs/seminar în care se pot derula evenimentele. În cadrul unui eveniment se specifică și durata acestuia. Pentru a rezolva problema, trebuie asigurate pentru fiecare eveniment sălile în care acestea se vor ține, respectiv intervalele de timp (sau doar ziua și ora la care începe, deoarece pentru fiecare eveniment știm și durata acestuia), ce reprezintă mulțimea dată de produsul cartezian între mulțimea zilelor săptămânii și mulțimea orelor de start la care putem asigna evenimente.

Din acest motiv putem defini constrângerile tari. Prima dintre aceste constrângeri este faptul că trebuie să asignăm aceste intervalele de timp la evenimente, astfel încât pentru orice două evenimente diferite care au un grup de studenți comun, acestea să aibă asignate intervale de timp diferite. [4]. De asemenea, grupurile de studenți pot fi incluse în alte grupuri de studenți, de exemplu o grupă '2A1' (anul 2, seria A, grupa 1) este inclusă în grupa '2A' (anul 2, seria A). O a doua constrângere tare este și ea la rândul ei o subproblemă, anume asignarea evenimentelor la diferite intervale orare ținând cont de profesorii care le predau (i.e. două evenimente diferite ce au un profesor comun trebuie asigurate la intervale de timp diferite). [8]

A treia constrângere constă în faptul că fiecare eveniment are o categorie, adică fiecare eveniment

este un anumit tip de eveniment, iar fiecare categorie poate necesita diferite resurse. Sălile pe care le vom asigna au și ele o categorie, astfel când vom asigna sălile va trebui să ținem cont să asociem tipul evenimentului cu tipul sălii. De exemplu, am considerat evenimentele de două categorii: cursuri și seminarii/laboratoare. Sălile și ele sunt sali de curs sau de seminar/laborator. Această constrângere este necesară, deoarece cursurile au, de obicei, mai mulți participanți decât seminariile, iar o sală cu atributul 'sală de curs' este mai mare, astfel încât poate accepta un eveniment curs.

A patra constrângere este similară cu primele două, doar că aceasta se ocupă cu asignarea sălilor. Față de grupuri de studenți/profesorii, evenimentele nu au asignate din start sălile de curs. Pe acestea va trebui să le asignăm noi, ținând cont de faptul că o sală de curs/seminar nu poate fi asignată la două sau mai multe evenimente ce se petrec simultan.

Mai formal, combinăm aceste constrângeri pentru a defini mulțimile de obiecte cu care vom lucra:

- $E = \{e_1, e_2, \dots, e_n\}$, $n = |E|$ este mulțimea evenimentelor ce trebuiesc asignate
- $S = \{s_1, s_2, \dots, s_{|S|}\}$ este mulțimea tuturor grupurilor de studenți
- $P = \{p_1, p_2, \dots, p_{|P|}\}$ este mulțimea profesorilor
- $R = \{r_1, r_2, \dots, r_{|R|}\}$ este mulțimea resurselor (a sălilor) disponibile
- $T = \{t_1, t_2, \dots, t_{|T|}\}$ este mulțimea intervalelor de timp la care se pot asigna evenimente (mai exact, $T = D \times O$, unde $D = \{d_1, d_2, \dots, d_{|D|}\}$, $0 \leq |D| \leq 6$ și $O = \{o_1, o_2, \dots, o_k\}$, $k = |O|$, iar relația între o_i, o_j este $o_j = o_i + durata$, $0 \leq i \leq j$, *durata* este durata generală a unui eveniment

Pe lângă constrângerile tari specificate, pe care le vom încerca să le rezolvăm, putem să mai specificăm alte constrângeri tari cum ar fi: pentru fiecare eveniment, putem avea unele intervale de timp în care putem plasa evenimentul; și precedentă evenimentelor: faptul că unele evenimente trebuiesc puse înaintea altora.

De asemenea, putem avea și constrângeri slabe:

1. Evenimentele nu ar trebui asignate în ultimul interval de timp din fiecare zi;
2. Pentru o grupă de studenți, nu ar trebui să asignăm mai mult de trei evenimente într-o zi;
3. Pentru o grupă de studenți, nu ar trebui să asignăm doar un eveniment într-o zi [4]

Ultimele constrângeri slabe nu le vom trata, astfel considerăm că orice eveniment poate fi asignat oricărui interval de timp disponibil. Constrângerile slabe menționate aduc o optimizare la generarea orarului, dar nu sunt neapărat necesare în a fi tratate.

2.2 Flux de lucru

Aplicația are 3 module:

1. Modulul algoritmic: Aici sunt implementați algoritmi ce generează orarul
2. Modulul backend: Acest modul are importat modulul algoritmic, și comunică cu modulul frontend pentru a genera și transmite datele

3. Modulul frontend: Acest modul este interfața cu care utilizatorul va interacționa pentru a genera și vizualiza orarele. Comunică cu aplicația backend când cere date

Modulul algoritmic Acest modul expune un API pentru modulul backend, pentru a folosi diferiți algoritmi în generarea orarelor. Ca input, toate funcțiile ce rezolvă problema generării orarului primesc conținutul unui fișier XML cu diverse date despre ce trebuie să apară în orar. Cele mai importante date din fișier sunt:

- **group** (grupuri de studenți): conțin detalii despre fiecare grupă cum ar fi numele, numărul de membri, grupul de studenți în care e inclus (dacă este cazul)
- **prof** (profesori): conțin detalii despre nume, titlul profesorilor
- **resource** (săli de curs/seminar): conțin detalii despre nume, capacitate, categoria sălii
- **event** (evenimentele): aici găsim numele evenimentelor, actorii (grupurile de studenți și profesorii) ce participă, durata și categoria
- **assignment** (evenimentele împreună cu intervalele orare, respectiv camerele asiguate): sunt opționale la input, deoarece generarea orarului nu ține cont de acestea, dar prin ele se pot exporta datele tot ca fișier XML

Modulul conține o serie de algoritmi diferiți, ce vor fi detaliați în secțiunea 'Implementare și '. Astfel, primind ca input conținutul fișierului XML și tipul de algoritm ce se dorește a fi folosit, modulul returnează datele într-un format ce este mai apoi parsat în aplicația backend și salvat în baza de date.

Modului backend Dupa cum am menționat, în aplicația backend primim cereri legate de algoritmul cu care dorim să generăm orarul, și în funcție de cerere, apelăm una dintre metodele expuse de modulul algoritmic. Datele statice (cele de care avem nevoie din fișierul XML) sunt prelucrate direct după parsarea fișierului. Algoritmii vor returna un map, o serie de asocieri, între fiecare eveniment și datele asiguate acestuia (sala, ziua și ora la care începe). Aceste asocieri sunt după prelucrate în modulul backend și salvate în baza de date.

Prin modulul backend expunem mai multe endpoint-uri spre aplicația frontend, prin care putem selecta generarea orarului cu unul din algoritmi, putem vizualiza orarul pentru o anumită grupă de studenți, profesor sau sală, putem schimba parametrii legați de crearea intervalelor orare disponibile, putem muta unele evenimente la alte intervale orare și putem schimba template-ul după care se generează orarul (i.e. fișierul XML).

Tot în aplicația backend modelăm datele primite de la modulul algoritm în mai multe entități pentru a le salva mai apoi în baza de date. Baza de date este necesară pentru a reține date ce nu se pot schimba direct, ci doar prin schimbarea fișierului XML de input (date despre studenți, profesori, săli și evenimentele neasiguate), dar și pentru a reține date legate de evenimentele asiguate, pentru eventualitatea în care se folosește același algoritm la generare, datele se vor prelua direct din baza de date.

Entitățile ce se vor salva în baza de date sunt:

- **StudentGroupEntity** - entitate ce reține datele despre o grupă de studenți;
- **ProfessorEntity** - entitate ce reține datele despre un profesor;
- **ResourceEntity** - entitate ce reține datele despre o sală;
- **EventEntity** - entitate ce reține datele despre un eveniment care poate sau nu poate fi asignat;
- **AssignedEventEntity** - entitate ce reține datele despre eveniment, sala asociată, respectiv ziua și ora la care începe evenimentul);
- **TimetableFileEntity** - entitate ce reține conținutul unui fișier XML ce a fost trimis din frontend;

Fiecare entitate are asociat un repository, un serviciu și un controller. În servicii, se face apel la repository pentru a prelua datele necesare din baza de date, date ce sunt prelucrate în obiecte de transfer de date (DTO) și peste care se mai pot face operații, dacă este necesar. În controller se preiau datele din serviciu și se expun endpointuri pentru a trimite datele la modulul frontend, pentru vizualizare.

Modulul frontend În acest modul utilizatorii pot vizualiza orarul generat într-un format tabelar, cu orele de start ca și linii și zilele ca și coloane. Aplicația comunică cu modulul backend printr-un API REST, primește sau trimite datele necesare, le prelucrează, dacă este cazul și le afișează în interfața web.

În modulul frontend avem entități în care putem salva obiectele de transfer primite de la aplicația backend, și o serie de servicii prin care putem face cereri către endpointurile expuse în backend.

Partea de interfață a aplicației conține mai multe componente:

- **Timetable** - este componenta principală, ce conține tabelul în care se pot vizualiza evenimentele;
- **Sidebar** - este componenta în care putem vizualiza evenimentele neasignate, selecta tipul de algoritm dorit pentru generare și selecta diferite tipuri de vizualizare (în funcție de grup de studenți, profesor, sală).
- **EventCard** - este componenta în care putem vizualiza atât evenimente asignate, cât și neasignate, în detaliu (putem vedea toate datele despre ele), și putem cere să vedem unde putem să mutăm/asignăm evenimentul
- **AssignEventForm** - este componenta prin care putem muta un eveniment deja asignat sau să asignăm unul neasignat
- **FileTransfer** - este componenta prin care putem încărca un fișier XML diferit, putem selecta template-ul dorit pentru generare, putem exporta orarul curent și putem schimba parametrii de creare pentru intervalele orare (numărul de zile, ora la care începe ziua, ora la care se termină ziua, cât durează, în general, un curs/seminar)

3 Arhitectura sistemului

3.1 Schema bazei de date

Pentru a vizualiza schema bazei de date, vom analiza scripturile de CREATE pentru fiecare tabel din schemă:

- Tabelul **assigned_events**:

```
CREATE TABLE IF NOT EXISTS assigned_events
(
    id bigint NOT NULL DEFAULT nextval('assigned_events_id_seq'::regclass),
    day integer,
    "time" time(6) without time zone,
    event_id bigint,
    resource_id bigint,
    CONSTRAINT assigned_events_pkey PRIMARY KEY (id),
    CONSTRAINT uk_7ntvb2eko21vxcdn62lk2l4nj UNIQUE (event_id),
    CONSTRAINT fk1nr13v8j3s2189hd8kby6ilxc FOREIGN KEY (event_id)
        REFERENCES public.events (id) MATCH SIMPLE,
    CONSTRAINT fkblx4brmwr2vjcms7m6fsd1pwc FOREIGN KEY (resource_id)
        REFERENCES public.resources (id) MATCH SIMPL
)
```

În acest tabel avem date despre evenimente asignate cum ar fi ziua (day), ora de start (time) și id-ul, ce are rol de cheie primară. Celelalte două coloane, 'event_id' și 'resoruce_id' sunt chei străine către tabelele 'event' și 'resource'. Prin aceste chei străine obținem evenimentul ce a fost asignat, respectiv sala care i-a fost asignată.

Tabelul 'assigned_events' este în relație unu-la-unu cu tabelul 'events', și în relție multi-la-unu cu tabelul 'resources'. Astfel un 'event' poate avea doar o singură asignare, iar acea asignare poate avea doar o singură cameră asociată (dar o cameră poate fi asignată la mai multe evenimente, în intervale orare diferite).

- Tabelul **resources**:

```
CREATE TABLE IF NOT EXISTS public.resources
(
    id bigint NOT NULL DEFAULT nextval('resources_id_seq'::regclass),
    abbr character varying(255),
    capacity integer,
    name character varying(255),
    notes text,
    quantity integer,
```

```

        type character varying(255),
    CONSTRAINT resources_pkey PRIMARY KEY (id)
)

```

Tabelul 'resources' are date despre sălile disponibile în universitate: 'id' este cheia primară, 'abbr' reprezintă numele prescurtat al sălii, 'capacity' reprezintă capacitatea (numărul maxim de persoane care încap în sală), 'name' este numele complet al sălii și 'type' este categoria ei (de exemplu, sala de curs sau sala de seminar/laborator). Acest tabel este în relație de tip 'unu-la-multi' cu tabelul 'assigned_events', astfel o sală poate fi asignată la mai multe evenimente diferite.

- Tabelul **events**:

```

CREATE TABLE IF NOT EXISTS events
(
    id bigint NOT NULL DEFAULT nextval('events_id_seq'::regclass),
    abbr character varying(255),
    actors character varying(255),
    duration integer,
    event_group character varying(255),
    frequency integer,
    name character varying(255),
    notes text,
    type character varying(255),
    CONSTRAINT events_pkey PRIMARY KEY (id)
)

```

În acest tabel avem datele despre evenimentul în sine: 'id' are rol de cheie primară, 'abbr' reprezintă numele prescurtat al evenimentului, iar acesta trebuie să fie unic, 'actors' reprezintă un șir de caractere în care apar abrevierile numelor tuturor grupelor de studenți și a tuturor proferorilor participanți, 'duration' reprezintă durata evenimentului, 'name' este numele complet al acestuia și 'type' este categoria evenimentului (după cum am menționat mai sus, de obicei curs sau laborator/seminar).

Acest tabel este în relație 'multi-la-multi' cu tabelele 'student_groups' (tabelul de studenți) și 'professors' (tabelul de profesori), de aceea corelarea între acestea trebuie făcută într-un tabel separat.

- Tabelul **student_groups**:

```

CREATE TABLE IF NOT EXISTS public.student_groups
(
    id bigint NOT NULL DEFAULT nextval('student_groups_id_seq'::regclass),
    abbr character varying(255),

```

```

        member_count integer ,
        name character varying(255) ,
        notes text ,
        parent character varying(255) ,
        CONSTRAINT student_groups_pkey PRIMARY KEY (id)
    )

```

În acest tabel sunt informațiile despre grupele de studenți: 'id' este cheia primară, 'abbr' este prescurtarea numelui grupei de studenți, 'member_count' reprezintă numărul de studenți din grupă, 'name' este numele complet al grupei, iar 'parent' este un șir de caractere ce conține abrevierea numelui primei grupe (în mod ierarhic) în care este inclusă grupa curentă. După cum am menționat, acest tabel este în relație 'multi-la-multi' cu tabelul 'events'.

- Tabelul **event_student_groups**:

```

CREATE TABLE IF NOT EXISTS public.event_student_groups
(
    student_group_id bigint NOT NULL,
    event_id bigint NOT NULL,
    CONSTRAINT fkl5yyaddy1uf32asvvvppyqg5s FOREIGN KEY (event_id)
        REFERENCES public.events (id) MATCH SIMPLE,
    CONSTRAINT fkmtuduqxravdn4lahbg67hqbo8 FOREIGN KEY (student_group_id)
        REFERENCES public.student_groups (id) MATCH SIMPLE
)

```

Rolul acestui tabel este de a corela tabelele 'events' și 'student_groups'. Astfel, în tabel există coloana 'student_group_id' ce reprezintă cheia străină către tabelul 'student_groups', și coloana 'event_id', ce reprezintă cheia străină către tabelul 'events'.

- Tabelul **professors**:

```

CREATE TABLE IF NOT EXISTS public.professors
(
    id bigint NOT NULL DEFAULT nextval('professors_id_seq'::regclass),
    abbr character varying(255),
    email character varying(255),
    name character varying(255),
    notes text,
    parent character varying(255),
    prefix character varying(255),
    CONSTRAINT professors_pkey PRIMARY KEY (id)
)

```

)

În acest tabel se rețin datele profesorilor: 'id' este cheia primară, 'abbr' reprezintă prescurtarea numelui profesorului, 'email' și 'name' sunt email-ul și numele profesorului, iar 'prefix' reprezintă titlul acestuia. După cum am menționat, tabelul 'professors' este în relație 'multi-la-multi' cu tabelul 'events', astfel avem nevoie de o tabelă separată pentru a reține asocierile.

- Tabelul **event_professors**:

```
CREATE TABLE IF NOT EXISTS public.event_professors
(
    professor_id bigint NOT NULL,
    event_id bigint NOT NULL,
    CONSTRAINT fk9kx6amkrpimkjjiy07snf5j3f FOREIGN KEY (professor_id)
        REFERENCES public.professors (id) MATCH SIMPLE,
    CONSTRAINT fkrvyo61nxlpqopci9ro9fkjhhs FOREIGN KEY (event_id)
        REFERENCES public.events (id) MATCH SIMPLE
)
```

Rolul acestui tabel este de a corela tabelele 'professors' și 'events'. 'professor_id' este cheia străină ce face legatura cu tabelul 'professors', iar 'event_id' este cheia străină ce face legatura cu tabelul 'events'.

- Tabelul **timetable_files**:

```
CREATE TABLE IF NOT EXISTS public.timetable_files
(
    id bigint NOT NULL DEFAULT nextval('timetable_files_id_seq'::regclass),
    file bytea,
    name character varying(255),
    timestamp_added bigint,
    CONSTRAINT timetable_files_pkey PRIMARY KEY (id),
    CONSTRAINT uk_sra4w7w9imupc33ew9h2cjs7w UNIQUE (name)
)
```

În acest tabel se vor reține datele pentru orar încărcate precedent (i.e. conținutul fișierelor XML), în caz că se dorește refolosirea lor. 'id' este cheia primară, 'file' reprezintă conținutul fișierului (în bytes), iar 'name' este numele acestuia (constrângerea este ca numele să fie unic).

3.2 Diagramele datelor

Fișierul XML conține datele sub formă de tag-uri cu atribute în care se specifică datele. Acesta este parsat în modulul algoritmic și sunt create clasele POJO (Plain Old Java Object). Format-ul de

XML valid pentru parsare este următorul:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<timetable beginDate="dd.mm.yyyy" endDate="dd.mm.yyyy" homePage="<url>"
hourLength="120" hoursPerDay="6" implementers="Firstname_Lastname"
name="Orar" notes="" startTime="08:00" title="Title">
  <assignment>
    <assignment day="0" end="2" endTime="10:00" event="Event1" notes=""
resources="Room1" start="0" startTime="8:00" week="0"/>
  </assignments>
<events>
  <event abbr="Event1" actors="group1A ,_group2A ,_prof1" duration="2"
frequency="1" group="" name="Event1_Name" notes="" type="C"/>
</events>
<resources>
  <resource abbr="Room1" capacity="300" name="Room1" notes=""
quantity="1" type="curs"/>
</resources>
<students>
  <group abbr="group1A" memberCount="30" name="group1A_name" notes=""
parent="groupA"/>
  <group abbr="group2A" memberCount="30" name="group2A_name" notes=""
parent="groupA"/>
  <group abbr="groupA" memberCount="60" name="groupA_name" notes=""
parent="" />
</students>
<profs>
  <prof abbr="prof1" email="" name="prof1_name" notes="" parent=""
prefix="" />
</profs>
</timetable>
```

De asemenea, fișierul XML poate conține alte date salvate în POJO-uri, ce nu sunt necesare pentru algoritmi:

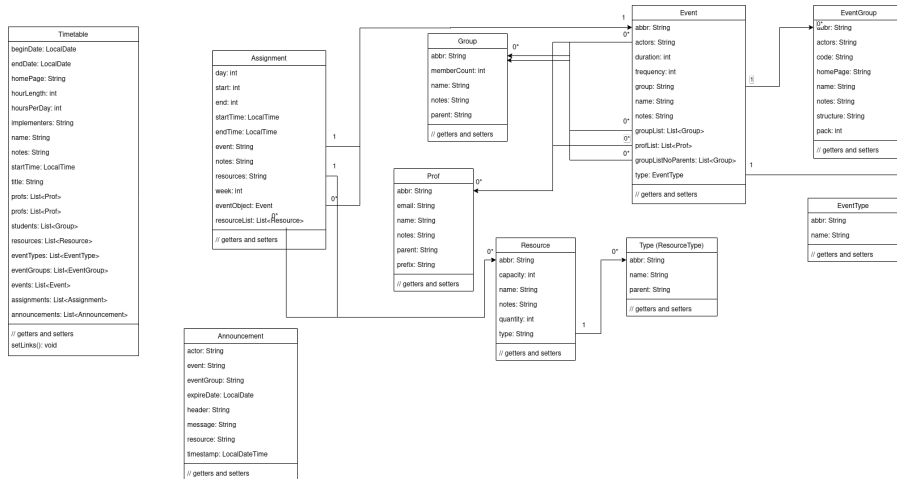
```
<resourceTypes>
  <type abbr="resType1" name="resType1_name" parent="" />
</resourceTypes>
<eventGroups>
  <eventGroup abbr="eventGroup1" actors="group1" code="code"
homePage="<url>" name="eventGroup1_name" notes="" structure="2C+2S"/>
</eventGroups>
<announcements>
```

```

<announcement actor="group1" event="" eventGroup="eventGroup1"
expireDate="" header="header" message="message" resource=""
timestamp="dd.mm.yyyy.HH-mm" />
</announcements>

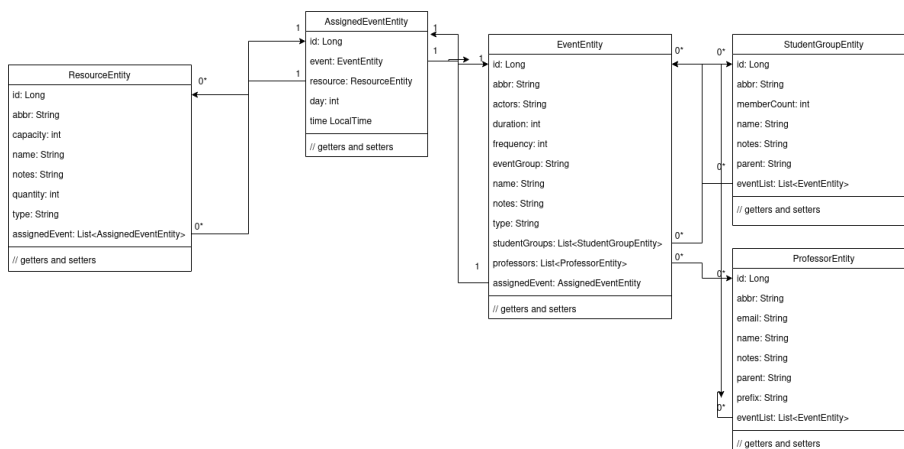
```

În funcție de aceste date, generăm clasele POJO. Datele pe care le conțin și relațiile dintre acestea sunt prezentate în diagrama următoare:



Modulul algoritmic se va folosi de aceste tipuri de date și de relațiile existente pentru a crea grafurile, peste care va aplica mai apoi o versiune de algoritm de colorare (în funcție de opțiunea selectată), și atât rezultatul algoritmului, cât și datele statice sunt prelucrate de modulul backend și transformate în entități pentru maparea în baza de date.

Următoarea imagine reprezintă diagrama de clase a entităților:



În această diagramă putem observa tipurile de date ce sunt salvate în baza de date. O entitate

'AssignedEventEntity' este astfel corelată cu o entitate de tip 'ResourceEntity' și cu una de tip 'EventEntity', iar 'EventEntity' este la rândul ei asociată cu o listă de 'StudentGroupEntity' și cu o listă de 'ProfessorEntity'.

Entitățile sunt mapate în baza de date cu ajutorul mapării obiect-relaționale (ORM) din framework-ul Hibernate. Maparea obiect-relațională rezolvă problema mapării unui obiect cu diferite date și cu relații cu alt obiect la un obiect relațional, ce poate fi persistat în baza de date (și după preluat exact cum a fost salvat). [9]

4 Implementare și testare

4.1 Implementare

În secțiunea implementare vom observa, la fiecare capitol, detaliile de implementare pentru fiecare din algoritmi (modul de generare al grafului, aplicarea algoritmului) și diferențele față de ceilalți algoritmi.

4.1.1 Consecutive Room Coloring

Acest algoritm își propune să rezolve problema generării orarului folosind sălile drept culori, astfel prin rezolvarea colorării grafului se poate rezolva constrângerea tare ca aceeași cameră să nu fie asignată la evenimente din același interval de timp. De asemenea, ne propunem să mai punem o constrângere asupra generării, anume că dacă un grup de studenți are mai multe ore în aceeași zi, acestea să fie una după alta și, în funcție de disponibilitate și de tipul sălii, acestea să se petreacă în aceeași sală. Datorită acestei noi constrângeri pe care o impunem, va trebui să transformăm una dintre constrângerile slabe menționate anterior, cea care spune că un grup de studenți să nu aibă mai mult de trei evenimente pe zi [4], în constrângere tare (altfel, datorită constrângerii noi, am putea avea mai mult de trei evenimente pentru o grupă în aceeași zi). Noua constrângere pe care o impunem duce la generarea unui orar mai răspândit din punct de vedere al zilelor, dar mai compact din punct de vedere al evenimentelor/zi. De asemenea, orarul mai răspândit duce la mai multe evenimente care nu vor putea fi asignate, lucru pe care îl vom vedea în secțiunea 'Testare'.

Vom avea două tipuri de culori, pentru fiecare tip de sală, și anume culoare cu sala tip curs și culoarea cu sala tip seminar/laborator. La fiecare pas, înainte să vizualizăm lista de culori disponibile pentru un eveniment, alegem lista de culori posibile pentru tipul de eveniment, după care o filtrăm și obținem setul adevărat de culori disponibile pentru eveniment.

Pentru a rezolva noua constrângere, dorim o structură de date în care să putem reține ultimele noduri pentru care au fost asignate camere. Astfel, generarea orarului se va face doar în funcție de profesori, coliziunile pe grupurile de studenți le putem verifica ulterior.

Graful pe care îl creăm va avea setul de evenimente $E = \{e_1, e_2, \dots, e_n\}$, $n = |E|$ ca mulțime a nodurilor, muchiile dintre acestea vor fi generate în funcție de conflictele pe profesori (dacă două evenimente au cel puțin un profesor comun, creăm o muchie între nodurile asociate). Graful $G = (N, M)$, unde N este mulțimea nodurilor și M este mulțimea muchiilor este generat în felul următor:

```

public TimetableGraph<TimetableNode , TimetableEdge> createGraph(Timetable
timetable) {
    List<TimetableNode> nodes = new ArrayList<>();
    List<TimetableEdge> edges;

    // Add nodes
    for (Event event : timetable.getEvents()) {
        if (event.getType().equals("C") || event.getType().equals("L") ||
            event.getType().equals("S")) {
            TimetableNode node = new TimetableNode(event, false, null);
            nodes.add(node);
        }
    }

    edges = new ArrayList<>(addEdgesBasedOnProfs(timetable , nodes));
    return new TimetableGraph<>(nodes , edges);
}

```

În secțiunea de cod de mai sus observăm cum, din mulțimea tuturor evenimentelor din orar, le selectăm doar pe cele care au tipurile cu care lucrăm (cursuri, seminarii sau laboratoare), după care generăm muchiile în funcție de coliziunile pe profesori.

Pentru algoritm ne mai trebuie două mulțimi de culori, anume mulțimea culorilor pentru cursuri și mulțimea culorilor pentru seminarii/laboratoare. Această filtrare se face similar cu cea făcută pentru adăugarea nodurilor.

Următoarele metode reprezintă esența acestui algoritm:

```

public void colorGraph(int algorithmOption , boolean useSorting , boolean
shuffle) {
    applyOptimisations(useSorting , shuffle);
    for (int day = 0; day < AlgorithmConstants.NUMBER_OF_DAYS; day++) {
        for (LocalTime time = AlgorithmConstants.START_TIME; time.isBefore(
            AlgorithmConstants.END_TIME);
            time = time.plusHours(AlgorithmConstants.GENERAL_DURATION)) {
            colorNodesAtDayAndTime(day , time);
        }
    }
}

private void colorNodesAtDayAndTime(int day, LocalTime time) {
    Set<TimetableColorRoom> availableColorsLab = new HashSet<>(
        timetableLaboratoryColors);
    Set<TimetableColorRoom> availableColorsCourse = new HashSet<>(
        timetableCourseColors);
}

```

```

Set<Group> assignedGroups = new HashSet<>();
// Try to assign rooms to groups that have already been assigned
List<TimetableNode> previouslyAssignedNodes =
    getPreviouslyAssignedNodes(day, time);
previouslyAssignedNodes = filterNodesByMaxSuccessiveAssignments(
    previouslyAssignedNodes, day, time);
List<TimetableNode> otherNodes = new ArrayList<>(graph.getNodes());
otherNodes.removeAll(previouslyAssignedNodes);

assignedGroups = assignColorToNode(day, time, availableColorsLab,
    availableColorsCourse,
    assignedGroups, previouslyAssignedNodes);
assignedGroups = assignColorToNode(day, time, availableColorsLab,
    availableColorsCourse,
    assignedGroups, otherNodes);
}

```

Prima metodă apelează la opțiunile algoritmului (i.e. sortarea descrescătoare în funcție de gradul nodului sau amestecarea evenimentelor. După aceasta, într-o buclă peste numărul de zile și peste intervalele orare pentru fiecare zi, se apelează colorarea nodurilor.

În a doua metodă, preluăm culorile disponibile pentru ambele tipuri de evenimente. În cadrul algoritmului avem o structură de date (un map de la student la o combinație între cameră, zi și oră) în care reținem, pentru fiecare zi, grupele pentru care au fost asignate evenimente în trecut, ca mai apoi în metoda 'assignColorToNode' să vedem ce tip de culoare trebuie să folosim, și să încercăm să asignăm culoarea care a fost asignată precedent pentru a cea grupa (i.e. dacă o grupă a avut deja un eveniment în acea sală, urmatorul eveniment să vină imediat după și să folosească aceeași sală). Înainte de asignare trebuie să filtrăm nodurile în funcție de numărul de evenimente pe care le-au avut în acea zi. Astfel, folosindu-ne de map-ul menționat anterior, putem obține grupele care au avut deja trei evenimente în acea zi, și să scoatem nodurile asociate lor din grupul nodurilor ce urmează a fi asociate în acea zi.

În metoda 'assignColorToNode' filtrăm nodurile după profesori comuni (verificăm muchiile), intervalul de timp, grupul de studenți și camerele disponibile, și îi asignăm o cameră disponibilă (culoarea), împreună cu ziua și ora pentru asignare.

La finalizarea algoritmului, asocierea între noduri (evenimente) și obiecte 'ColorDayTimeWrap' (obiecte ce conțin culoarea - sala - folosită, ziua și timpul) este creată, și este returnată pentru folosire.

4.1.2 Day-Time-Room Coloring Greedy

În acest algoritm am folosit o altă abordare pentru generarea grafului și a culorilor, și scoatem constrangerea de a avea evenimentele unul după altul în aceeași zi.

Generarea grafului se face similar cu cea de la algoritmul precedent: mulțimea nodurilor, M , va fi aceeași cu mulțimea evenimentelor, iar generarea muchiilor se schimbă astfel:

```

private List<TimetableEdge> addEdgesBasedOnProfsAndGroups(Timetable
timetable , List<TimetableNode> nodes) {
    List<TimetableEdge> edges = new ArrayList<>();

    // Add edges
    for (TimetableNode node : nodes) {
        Set<Prof> nodeProfs = new HashSet<>(node.getEvent().getProfList());
        Set<Group> nodeGroups = new HashSet<>(node.getEvent().getGroupList
());
        for (TimetableNode otherNode : nodes) {
            Set<Prof> otherNodeProfs = new HashSet<>(otherNode.getEvent().
getProfList());
            Set<Group> otherNodeGroups = new HashSet<>(otherNode.getEvent()
.getGroupList());
            if (node.equals(otherNode)) {
                continue;
            }

            Set<Prof> nodeProfsCopy = new HashSet<>(nodeProfs);
            nodeProfsCopy.retainAll(otherNodeProfs);

            Set<Group> nodeGroupsCopy = new HashSet<>(nodeGroups);
            nodeGroupsCopy.retainAll(otherNodeGroups);

            if (!nodeProfsCopy.isEmpty() || !nodeGroupsCopy.isEmpty()) {
                TimetableEdge edge = new TimetableEdge(node, otherNode);
                edges.add(edge);
            }
        }
    }

    return edges;
}

```

Practic, în loc să adăugăm muchiile doar în funcție de profesor și după care să verificăm separat coliziunile pe studenți, vom adauga o muchie între două evenimente dacă ele au cel puțin un grup de studenți comun sau un profesor comun. Acest lucru va eficientiza rularea, deoarece nu va trebui să verificăm separat coliziunile între grupuri de studenți.

În generarea celor două mulțimi de culori (pentru curs și pentru seminar/laborator), folosim produsul cartezian între mulțimea sălilor și mulțimea intervalelor de timp (zi + interval orar). Astfel, putem verifica și conflictele pe cameră / interval de timp simultan.

Pentru un graf $G = (N, M)$ ($N = \{n_1, n_2, \dots, n_{|N|}\}$, și o permutare a nodurilor π un algoritm greedy pentru colorarea grafului funcționează în următorii pași:

1. Colorăm n_{π_0}
2. Pentru $i = 1, |N|$
3. Presupunem că s-au folosit deja k culori
4. Colorăm v_{π_i} cu prima culoare disponibilă (care nu crează conflict) din $\{0, 1, \dots, k\}$ [10]

Metoda prin care implementăm acest algoritm este 'greedyColoringMethod()', în care, pentru fiecare nod, scoatem din setul culorilor disponibile culorile ce au fost asignate vecinilor (i.e. nodurile ce au conflicte de tip grup de studenți/profesor).

```
Set<TimetableNode> neighbors = graph.getNeighbors(node);
Set<TimetableColorIntervalRoom> usedColors = new HashSet<>();
Set<TimetableColorIntervalRoom> availableColors;
if (node.getEvent().getType().equals("C")) {
    availableColors = new HashSet<>(availableCourseColors);
} else {
    availableColors = new HashSet<>(availableLaboratoryColors);
}

for (TimetableNode neighbor : neighbors) {
    if (nodeColorMap.containsKey(neighbor)) {
        ColorDayTimeWrap wrap = nodeColorMap.get(neighbor);
        for (TimetableColorIntervalRoom color : availableColors) {
            if (color.getDay() == wrap.getDay() && color.getTime().getHour()
                == wrap.getTime().getHour()) {
                usedColors.add(color);
            }
        }
    }
}
}
```

În final, selectăm din setul culorilor disponibile culoarea potrivită în funcție de sală / interval de timp și o asignăm evenimentului curent. Acest lucru e repetat până sunt asignate toate nodurile sau până nu mai putem asigna culori.

La final, rezultatul este întors la fel ca la algoritmul precedent, anume un map între evenimente și 'ColorDayTimeWrap', care sunt, în acest caz, direct culorile asignate de algoritm.

4.1.3 Day-Time-Room Coloring DSatur

Acest algoritm folosește o generare a grafului identică cu cea a algoritmului 'Day-Time-Room Coloring Greedy', aici schimbăm doar tehnica de asignare a nodurilor, anume vom folosi tehnica 'DSatur'.

DSatur este o euristică cunoscută în rezolvarea problemei de colorare a grafului. În această euristică, nodurile nu sunt asignate pe rând la colorare, în schimb se selectează pentru colorare nodul cu cel mai mare număr de vecini deja colorați. În caz de egalitate, se pot aplica și alte euristici, cum ar fi selectarea nodului cu gradul mai mare. [11]

Numărul de vecini deja colorați se numește saturația nodului, iar pe aceasta o vom salva în cadrul obiectelor de tip nod, alături de evenimentul în sine. Pentru a asigna nodurile în ordine, în funcție de saturație vom folosi o coadă. Inițial, punem în coadă toate nodurile, și după le scoatem și readăugăm în funcție de saturația lor la iterația curentă.

```
for (T timetableNode neighbor : neighbors) {
    if (nodeColorMap.containsKey(neighbor)) {
        continue;
    }
    Optional<T timetableNode satur> nodeSaturOpt = nodeSaturList.stream()
        .filter(saturNode -> saturNode.getNode().equals(neighbor))
        .findFirst();

    if (nodeSaturOpt.isEmpty()) {
        continue;
    }
    T timetableNodeSatur nodeSatur = nodeSaturOpt.get();
    queue.remove(nodeSatur);
    nodeSatur.setSaturation(nodeSatur.getSaturation() + 1);
    queue.add(nodeSatur);
}
```

În secvența de cod de mai sus observăm cum selectăm dintre vecinii nodului asignat curent vecinul cu saturația cea mai mare, și îl adăugăm în coada cu saturația crescută (deoarece este vecin cu nodul asignat curent). Algoritmul se oprește când coada este goală (nu mai avem noduri de asignat) sau nu mai putem asigna culori, iar datele sunt returnate în același format ca la algoritmii precedenți.

4.1.4 Two-Step Day-Time Room Coloring Greedy

Dupa cum menționează și numele, în acest tip de algoritm generarea orarului se va face în doi pași (ulterior generării grafului): Pasul 1 presupune crearea culorilor în funcție de intervalele de timp, și rezolvarea colorării grafului folosind strict această colorare. Pasul 2 presupune crearea unui graf bipartit $G = ((N1, N2), M)$ unde $N1$ este mulțimea evenimentelor dintr-un anumit interval de timp, iar $N2$ este mulțimea sălilor. Mulțimea M a muchiilor este generată astfel încât muchiile să ducă de la un eveniment de un anumit tip, la o sală de același tip (e.g. să avem muchii între noduri cu eveniment de tip 'curs' și noduri cu camere de tip 'curs'). Rezolvarea problemei de cuplaj maxim pe acest graf bipartit rezultă în asignarea camerelor la evenimente.

Rezolvarea problemei de cuplaj maxim pentru un graf bipartit presupune găsirea cuplajului cu cardinalitatea cea mai mare. Pentru rezolvare am folosit algoritmul Hopcroft-Karp, ce presupune găsirea

căilor de augmentare prin operații de breadth-first search urmate de depth-first search, astfel rezultând cu creșterea cardinalității cuplajului la fiecare pas. [12]

Pentru generarea grafului inițial (cel ce trebuie rezolvat de algoritmul de colorare), se definesc o serie de matrici:

- $M_{ij}^{(1)} = \{1, \text{daca grupul de studenti } s_i \text{ are asociat evenimentul } e_j, 0, \text{altfel} \}$
- $M_{ij}^{(2)} = \{1, \text{daca profesorul } p_i \text{ are asociat evenimentul } e_j, 0, \text{altfel} \}$
- $M_{ij}^{(3)} = \{1, \text{daca sala } r_i \text{ are asociata categoria } c_j, 0, \text{altfel} \}$
- $M_{ij}^{(4)} = \{1, \text{daca evenimentul } s_i \text{ are asociata categoria } c_j, 0, \text{altfel} \}$
- $R_{ij} = \{1, \text{daca evenimentul } e_i \text{ si camera } r_j \text{ au aceeasi categorie, } 0, \text{altfel} \}$
- $C_{ij} = \{1, \text{daca evenimentul } e_i \text{ este in conflict cu evenimentul } e_j, 0, \text{altfel} \}$ [4]

Generarea grafului inițial $G_1 = (N, M_1)$ se face inițializând mulțimea nodurilor cu mulțimea evenimentelor, și adăugând muchii între evenimentele e_i și e_j dacă $C_{ij} = 1$ (în matricea de conflicte, vom considera conflicte următoarele: evenimente cu grupuri de studenți comune, evenimente cu profesori comuni și evenimente ce sunt obligate să împartă o cameră - în cazul în care nu mai este niciuna de același tip disponibilă).

După cum am menționat, culorile sunt generate în funcție de intervalele de timp (zi + interval orar). Rezolvarea problemei colorării folosește același tip de metodă greedy ca la algoritmul 'Day-Time-Room Coloring Greedy', doar că nu se mai preocupă cu asignarea sălilor.

```

for (TimeslotDataNode node : graph.getNodes()) {
    List<Timeslot> availableColors = new ArrayList<>(colorList);
    for (TimeslotDataNode neighbor : graph.getNeighbors(node)) {
        if (solution.containsKey(neighbor)) {
            availableColors.remove(solution.get(neighbor));
        }
    }
    if (!availableColors.isEmpty()) {
        solution.put(node, availableColors.get(0));
    }
}

```

După obținerea asignărilor eveniment-interval de timp, creăm câte un map în care asociem fiecare interval de timp la lista evenimentelor asignate pentru acesta. Pentru fiecare astfel de map generat, creăm graful bipartit $G_2 = ((N_1, N_2), E_2)$ cu primul set de noduri N_1 evenimentele din intervalul de timp curent, și al doilea set de noduri N_2 mulțimea sălilor disponibile.

```

public class RoomDataGraph {
    // set 1 of the bipartite graph
    private List<TimeslotDataNode> timeslotEvents = new ArrayList<>();
    // set 2 of the bipartite graph
    private List<RoomDataNode> rooms = new ArrayList<>();
    private TimeslotDataModel timeslotDataModel;

    private List<RoomDataEdge> edges = new ArrayList<>();
}

```

Mulțimea muchiilor E_2 este generată astfel: pentru fiecare eveniment, trasăm muchii de la acel eveniment la camerele care au aceeași categorie. Nu se pot trasa muchii între evenimente și alte evenimente, și nici între săli și alte săli, astfel graful este bipartit.

Pentru a rezolva problema cuplajului de cardinalitate maximă în graful bipartit creat, aplicăm algoritmul Hopcroft-Karp.

```

public void hopcroftKarpSolver() {
    for (TimeslotDataNode node : graph.getTimeslotEvents()) {
        eventPairs.put(node, null);
    }
    eventPairs.put(null, null);
    for (RoomDataNode node : graph.getRooms()) {
        roomPairs.put(node, null);
    }
    roomPairs.put(null, null);

    int matchingLength = 0;
    while (BFS()) {
        for (int i = 0; i < graph.getTimeslotEvents().size(); i++) {
            TimeslotDataNode node = graph.getTimeslotEvents().get(i);
            if (eventPairs.get(node) == null && DFS(node)) {
                matchingLength++;
            }
        }
    }
}

```

Funcția BFS parcurge graful în mod breadth-first-search și indică faptul că încă există căi de augmentare în graf, iar funcția DFS parcurge graful în mod depth-first-search începând cu un nod, și indică dacă există căi de augmentare de la nodul de start. La fiecare pas, se face diferența simetrică între muchiile din cuplajul curent și nodurile din calea de augmentare găsită, astfel dimensiunea noului cuplaj va crește cu 1. Algoritmul finalizează asignările când nu mai există căi de augmentare în graf. O soluție este perfect validă când fiecare nod din setul de evenimente are asociat o sală.

Rulăm acest algoritm pentru fiecare mulțime de evenimente asociată fiecărui interval de timp, formatăm rezultatele ca un map de la evenimente la 'ColorDayTimeWrap' și returnăm soluția.

4.1.5 Two-Step Day-Time Room Coloring DSatur

Majoritatea implementărilor acestui algoritm sunt similare cu implementările algoritmului 'Two-Step Day-Time Room Coloring Greedy'.

La pasul 1, în loc să folosim colorarea de tip greedy, folosim o colorare în ideea colorării dsatur, dar cu euristici diferite pentru alegerea nodului și a culorii.

Euristicile pe care le folosim sunt următoarele:

La alegerea nodului:

1. Selectăm evenimentul neassignat cu cele mai puține intervale de timp disponibile (cu cel mai mic număr de culori disponibil)
2. Selectăm evenimentul neassignat cu cele mai multe conflicte (cu cel mai mare grad)
3. Selectăm evenimentul neassignat în mod aleatoriu [4]

La alegerea culorii:

1. Selectăm culoarea validă pentru cel mai mic număr de noduri neassignate
2. Selectăm culoarea care a fost asignată de cele mai puține ori
3. Selectăm culoarea în mod aleatoriu [4]

Euristicile se aplică în ordine, în cazul în care după una din euristici avem mai mult de un nod pentru care euristica e adevărată.

```
private TimeslotDataNode chooseNode(List<TimeslotDataNode> unplacedEvents)
{
    List<TimeslotDataNode> chosenEvents = applyHeuristic1(unplacedEvents);
    if (chosenEvents.size() == 1) {
        return chosenEvents.get(0);
    }

    chosenEvents = applyHeuristic2(chosenEvents);
    if (chosenEvents.size() == 1) {
        return chosenEvents.get(0);
    }

    return applyHeuristic3(chosenEvents);
}
```

```
private Timeslot chooseColor(List<TimeslotDataNode> unplacedEvents,
    TimeslotDataNode event) {
```

```

List<Timeslot> availableColors = new ArrayList<>(colorList);
for (TimeslotDataNode neighbor : graph.getNeighbors(event)) {
    if (solution.containsKey(neighbor)) {
        availableColors.remove(solution.get(neighbor));
    }
}
List<Timeslot> chosenColor = applyHeuristic4(availableColors,
    unplacedEvents, event);
if (chosenColor.size() == 1) {
    return chosenColor.get(0);
}

chosenColor = applyHeuristic5(availableColors);
if (chosenColor.size() == 1) {
    return chosenColor.get(0);
}

return applyHeuristic6(chosenColor);
}

```

La fiecare pas, selectăm atât nodul, cât și culoarea în funcție de aceste euristici. Cu rezultatele de la acest algoritm sunt create grafurile bipartite menționate la algoritmul anterior, și rezolvarea cuplajului de cardinalitate maximă se face în același mod. Soluția este reprezentată de map-ul de la evenimente la 'ColorDayTimeWrap'.

4.2 PartialCol Optimisation

Algoritmul PartialCol este un algoritm evolutiv pe care îl putem aplica peste algoritmi 'Two-Step Day-Time Room Coloring Greedy' și 'Two-Step Day-Time Room Coloring DSatur'. Acest algoritm își propune să adauge evenimentele ce nu au putut fi asignate la intervale de timp.

Avem mulțimea evenimentelor neasignate S' și soluția inițială \mathcal{S} , din care considerăm mulțimea pentru un anumit interval de timp, S_i (mulțimea asociată intervalului de timp i). Funcția de cost folosită de algoritm este cardinalul mulțimii S' , $|S'|$. La fiecare iterație, este explorat întregul spațiu format din produsul cartezian dintre evenimentele neasignate S' și soluția curentă \mathcal{S} , fiecare mutare este făcută din S' în S_i astfel: Este mutat evenimentul e_j din S' în S_i și sunt scoase din S_i toate evenimentele în conflict cu acesta. După aceasta, crează un graf bipartit din noul S_i și mulțimea sălilor R , și se rezolvă problema cuplajului maxim. Dacă acest cuplaj nu este unul perfect (toate evenimentele au asociate săli), se anulează mutarea lui e_j în S_i . În cazul în care cuplajul este perfect, mutarea este reținută ca fiind validă. După explorarea spațiului $S' \times \mathcal{S}$, se alege mutarea validă cu cel mai mic cost (acest cost poate fi mai mare decât costul inițial, dar poate fi necesară explorarea soluției asociate lui pentru a găsi o soluție mai bună). De asemenea, la fiecare iterație verificăm și salvăm, dacă este cazul, soluțiile cu un cost mai mic decât cel curent. Toate mutările făcute sunt puse într-o tabelă 'tabu', ce impune ca aceste mutări

să nu se repete. [4]. În final, după un număr de iterații, cea mai bună soluție găsită este returnată și se aplică în continuare algoritmul de cuplaj maxim.

```
// explore neighborhoods
var currentSolution = new HashMap<>(bestIterationSolution);
var currentUnplacedEvents = new HashSet<>(bestIterationUnplacedEvents);
Map<Pair<Integer, Integer>, Integer> moveDeltas = new HashMap<>();
for (TimeslotDataNode event : currentUnplacedEvents) {
    for (Timeslot timeslot : currentSolution.keySet()) {
        int eventIndex = hardConstraintsModel.getEvents().indexOf(event.
            getEvent());
        int timeslotIndex = hardConstraintsModel.getTimeslots().indexOf(
            timeslot);

        if (tabuMatrix[eventIndex][timeslotIndex] > 0) {
            continue;
        }

        var pair = exploreNeighbor(event, timeslot, currentSolution,
            currentUnplacedEvents);
        if (pair == null) {
            continue;
        }
        var neighborSolution = new HashMap<>(pair.getFirst());
        var neighborUnplacedEvents = new HashSet<>(pair.getSecond());

        int neighborCost = neighborUnplacedEvents.size();
        int deltaCost = neighborCost - getCost(bestIterationUnplacedEvents)
            ;
        moveDeltas.put(new Pair<>(eventIndex, timeslotIndex), deltaCost);
        addMoveToTabuList(event, timeslot);
    }
}
```

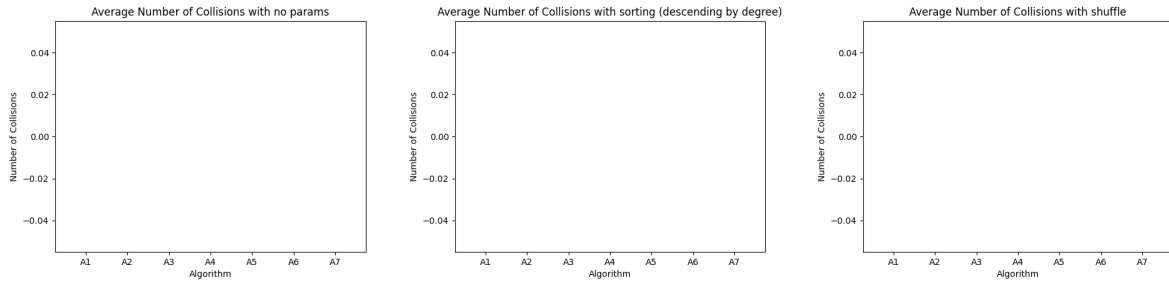
4.3 Testare

Algoritmii au fost testați pe o instanță reprezentată de orarul Facultății de Informatică din cadrul Universității "Alexandru Ioan-Cuza" Iași. Această instanță conține 408 evenimente de tip curs/seminar/laborator (din care 80 evenimente "curs", 328 evenimente "seminar/laborator") și 16 sali (din care 3 săli "curs" și 13 "seminar/laborator"). Pentru generarea intervalelor de timp, am considerat numărul de zile ca fiind 5 (luni-vineri), ora de start a zilei 08:00, ora de final a zilei 20:00, și durata generală a unui eveniment 2 ore. Astfel obținem 30 de intervale de timp disponibile.

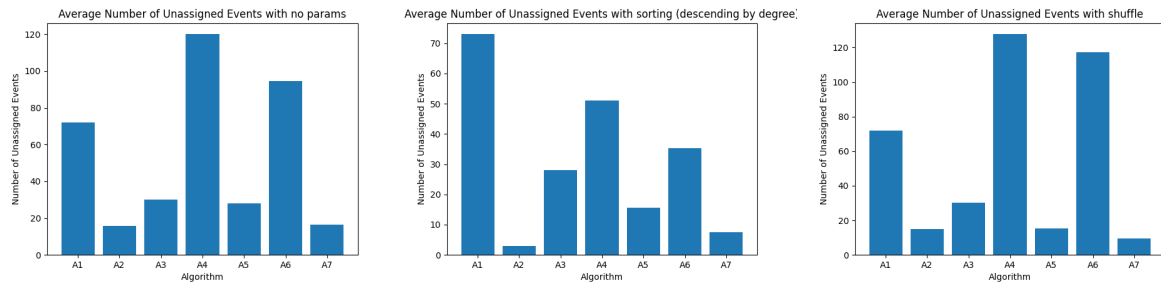
Am făcut câte 10 rulări testând fiecare dintre următoarele:

- Cazul 1: când nu avem nici parametrul de sortare, nici cel de shuffle setate
- Cazul 2: când parametrul de sortare este setat
- Cazul 3: când parametrul de shuffle este setat

Datorită respectării constrângerilor, evenimentele nu se vor suprapune. În următoarele grafice putem observa că numărul de coliziuni între evenimente este 0, pentru fiecare algoritm.



Pentru fiecare rulare, am salvat numărul de evenimente neasignate (respectiv numărul de cursuri și numărul de seminarii/laboratoare neasignate), și timpul de execuție.

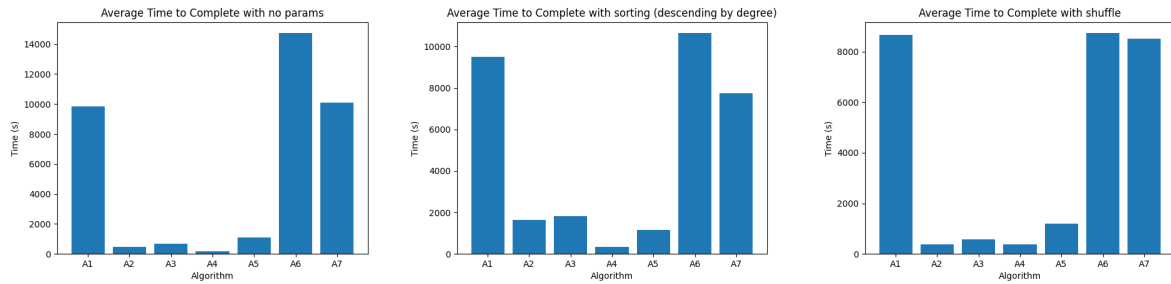


Aceste grafice reprezintă numărul de evenimente neasignate, în medie, pe algoritm. Maparea pentru algoritmi este următoarea:

- A1: Consecutive Room Coloring
- A2: Day-Time-Room Coloring Greedy
- A3: Day-Time-Room Coloring DSatur
- A4: Two-Step Day-Time Room Coloring Greedy
- A5: Two-Step Day-Time Room Coloring DSatur
- A6: Two-Step Day-Time Room Coloring Greedy cu PartialCol Optimisation
- A7: Two-Step Day-Time Room Coloring DSatur cu PartialCol Optimisation

Putem vedea că cele mai bune rezultate sunt date de algoritmul "Two-Step Day-Time Room Coloring DSatur" (și mai bune în cazul folosirii optimizării PartialCol), pentru cazul în care sortăm evenimentele descrescător după numărul de conflicte și pentru cel în care facem shuffle la evenimente, și respectiv de algoritmul "Day-Time-Room Coloring Greedy" când facem sortarea nodurilor.

Următoarele grafice reprezintă timpul de execuție, în medie, pentru fiecare din algoritmi (măsurat în milisecunde - ms).



După cum ne așteptam, algoritmi A6 și A7, care folosesc optimizarea PartialCol, rulează mai mult decât ceilalți. Algoritmul A1 are și el un timp de execuție mare datorită faptului că în acesta, majoritatea verificărilor conflictelor se fac local.

5 Manual de utilizare

5.1 Instalare

Cerințe preliminare:

- Java versiunea 17
- O baza de date PostresQL
- Angular CLI (**npm install -g @angular/cli**)

Pentru a rula aplicația backend, este necesară întâi arhivarea aplicației algoritmice într-un JAR file:

```
jar cf 'name-of-jar'.jar 'path-to-project'/src
```

În continuare, fișierul .jar obținut trebuie copiat în modulul backend (de exemplu, la path-ul 'path-to-backend'/lib/'name-of-jar'.jar), și instalat folosind comanda:

```
mvn install:install-file -Dfile=lib/'name-of-jar'.jar -DgroupId=org.timetable -DartifactId=Timetable
```

```
Project -Dversion=1.0-SNAPSHOT -Dpackaging=jar
```

După, trebuie construit modulul backend și toate dependențele sale cu:

```
mvn clean install
```

Rularea se poate face direct dintr-un IDE, sau la linia de comandă:

```
mvn package
```

```
java -jar 'path-to-backend-jar'.jar
```

Pentru a rula aplicația frontend, este necesară construirea ei (cu dependențele) și rularea aplicației (din root-ul proiectului):

```
npm install (intălează dependențele aplicației)
```

```
ng serve
```

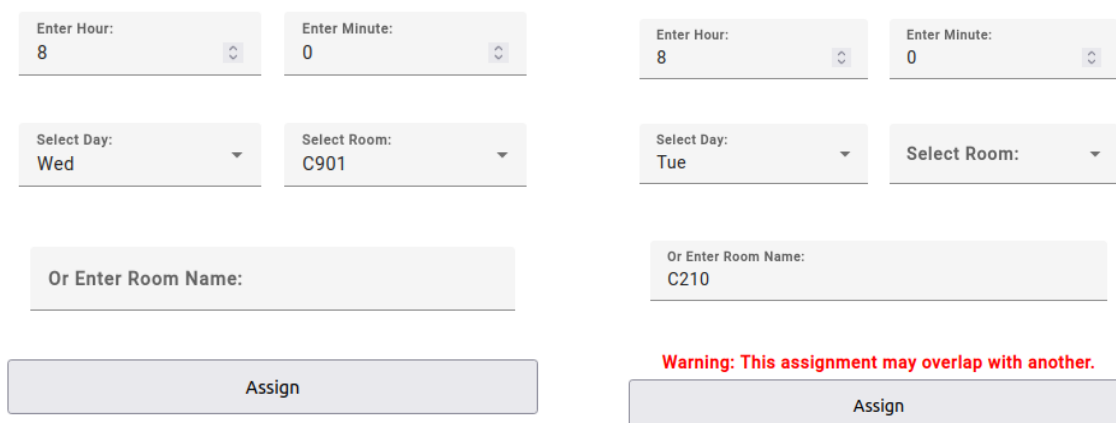
Prin interfața oferită de modulul frontend, utilizatorii pot interacționa cu modulul backend, astfel pot genera orarul folosind unul din algoritmi, pot să-l vizualizeze pentru o anumită grupă de studenți / un anumit profesor / o anumită sală.

În această imagine, observăm orarul generat pentru o anumită grupă după selectarea tipului de algoritm și apăsarea butonului 'Generate'. Selecția a fost făcută după grupa de studenți, și a fost selectată grupa 'Informatică, anul 1A, grupa 1'. Orezle apar ca niște carduri în orar, durata acestora fiind specificată în orar în funcție de lungimea cardului (de exemplu ora 'Logică pentru informatică', asignata sălii 'C403' este joi, de la ora 8 la ora 10).

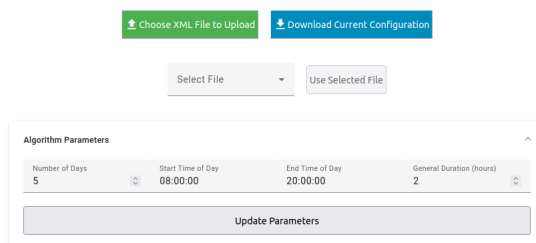
În lista din stânga (deasupra secțiunii de selecție) conține evenimente neasignate, care ori nu au putut fi asignate de algoritm, ori nu a fost nevoie ca algoritmul să le asigneze (ele pot fi alte tipuri de evenimente decât cursuri/seminarii/laboratoare).

O dată cu selecția (click) unuia dintre evenimentele de pe orar (asignate) sau din lista din stânga (neasignate), în secțiunea dreapta orarului apare un card cu detaliile pentru evenimentul respectiv (și sala, ziua și ora, în cazul în care evenimentul este asignat). Pe acest card putem selecta checkbox-ul

'Availability', lucru ce va rezulta în colorarea orarului pe secțiuni, în funcție de disponibilitatea timeslot-urilor. Astfel, putem muta un eveniment asignat sau asigna interval de timp și sală pentru unul neasignat, știind unde avem locuri libere în orar. Vizualizarea se face în funcție de disponibilitatea studenților (ei pot avea alte evenimente în perioadele ocupate) sau a profesorilor (ei pot predă alte materii în restul secțiunilor ocupate). În secțiunea de asignare a evenimentului putem vedea și sălile disponibile, în funcție de tipul acestuia.



Secțiunea de mai sus este secțiunea de asignare/mutare a unui eveniment. După selectarea 'Availability', această secțiune apare sub orar, și are diferite input-uri pentru timpul la care începe, ziua și camerele disponibile în acel interval. În loc să selectăm una dintre camerele disponibile, putem scrie noi numele uneia dintre camere dacă suntem siguri pe asignare. În cazul de conflict cu alte evenimente, va apărea o atenționare, dar asignarea va putea fi făcută în continuare.



Ultima secțiune este cea de deasupra orarului. În această secțiune putem adăuga un nou fișier XML ca input și a selecta unul dintre cele salvate pentru a-l folosi în generarea orarului. Butonul 'Download Current Configuration' va salva, tot în format XML, evenimentele asignate și le va descărca pentru utilizator. În secțiunea de sub aceste butoane există inputuri prin care utilizatorii pot schimba datele de generare pentru intervalele de timp (de exemplu, dacă majoritatea evenimentelor durează o oră, pot schimba 'General Duration' pentru evenimente).

6 Concluzii și direcții viitoare

6.1 Concluzii

Complexitatea programării cursurilor/seminariilor/laboratoarelor pentru universitate este direct proporțională cu numărul de constrângeri, și nu există un algoritm exact pentru a rezolva acest tip de problemă. [3]. În această lucrare, am observat cum, folosindu-ne de constrângerile acestei probleme, putem modela problema ca un graf cu nodurile drept evenimente, și muchii drept constrângeri, și putem să o reducem la o problema de colorare a unui graf. Am făcut comparații între mai multe tipuri de implementări și euristici pentru algoritmii de colorare.

Pe lângă partea algoritmică, am creat o aplicație prin care utilizatorii pot interacționa cu generarea orarului, pot edita anumite asignări făcute, și își pot modela datele în funcție de nevoile lor. Astfel am oferit mai mult decât un studiu asupra relației dintre problema generării orarului și algoritmii de colorare a grafurilor, am oferit un mod interactiv prin care utilizatorii pot folosi aplicația pentru a genera propriile orare.

6.2 Direcții viitoare

După cum am menționat, algoritmii dezvoltati nu își propun să rezolve constrângerile slabe precizate anterior. Astfel, o direcție viitoare ar fi folosirea unor algoritmi evolutivi pentru a rezolva aceste constrângeri. O schemă de algoritmi este Simulated Annealing (SA), prin care putem explora spațiul soluțiilor corecte, și putem încerca să minimizăm numărul de constrângeri slabe nerespectate. [4]

De asemenea, pentru cazurile în care instanțele sunt mult mai largi, putem încerca implementarea unui Algoritm Genetic pentru rezolvarea problemei generării orarului. În algoritmi genetici fiecare soluție posibilă este reprezentată ca un 'indiviz', și după generarea unui set de soluții valide (populația), indivizii sunt combinați și mutați aleatoriu pentru a genera noi soluții, pentru care se calculează un fitness. [13] Acest tip de abordare ar putea fi aplicat pentru încercarea soluțiilor pentru orar, dar este necesară asocierea problemei ca o instanță a unui algoritm genetic.

Pe lângă constrângerile tari abordate, am putea să mai avem constrângeri de tipul "acest eveniment nu se poate petrece în aceste intervale de timp" sau "acest eveniment trebuie să se petreacă înaintea celui alt eveniment". O abordare a acestui tip de constrângeri (constrângeri oferite de datele problemei) ar putea fi folositor pentru a crea orare mai eficiente.

În final, o îmbunătățire a interfeței web, prin care să putem folosi drag-and-drop pentru a asigna evenimentele ar fi un plus pentru interacțiunea utilizatorului cu aplicația.

7 Bibliografie

Referințe

- [1] Tommy R Jensen și Bjarne Toft, *Graph coloring problems*, John Wiley & Sons, 2011.
- [2] Norman Biggs, E Keith Lloyd și Robin J Wilson, *Graph Theory, 1736-1936*, Oxford University Press, 1986.

- [3] Runa Ganguli și Siddhartha Roy, “A study on course timetable scheduling using graph coloring approach”, în *international journal of computational and applied mathematics* 12.2 (2017), pp. 469–485.
- [4] Rhyd Lewis, *A guide to graph colouring*, vol. 7, Springer, 2015.
- [5] Ken Arnold, James Gosling și David Holmes, *The Java programming language*, Addison Wesley Professional, 2005.
- [6] Sam Newman, *Building microservices*, ” O’Reilly Media, Inc.”, 2021.
- [7] Brad Green și Shyam Seshadri, *AngularJS*, ” O’Reilly Media, Inc.”, 2013.
- [8] GA Neufeld și John Tartar, “Graph coloring conditions for the existence of solutions to the timetable problem”, în *Communications of the ACM* 17.8 (1974), pp. 450–453.
- [9] Christopher Ireland et al., “Understanding object-relational mapping: A framework based approach”, în *International Journal On Advances in Software* 2.2 (2009).
- [10] Joseph Culberson, “Iterated greedy graph coloring and the difficulty landscape”, în (1992).
- [11] Isabel Méndez-Díaz și Paula Zabala, “A branch-and-cut algorithm for graph coloring”, în *Discrete Applied Mathematics* 154.5 (2006), pp. 826–847.
- [12] Norbert Blum, *A simplified realization of the Hopcroft Karp approach to maximum matching in general graphs*, vol. 19, Citeseer, 1999.
- [13] Alberto Colorni, Marco Dorigo și Vittorio Maniezzo, “A genetic algorithm to solve the timetable problem”, în *Politecnico di Milano, Milan, Italy TR* (1992), pp. 90–060.