UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Planificarea joburilor prin tehnici de multicolorare a grafurilor

propusă de

Denis-Gheorghe Hrițcu

Sesiunea: iulie, 2025

Coordonator științific

Lector dr. Olariu E. Florentin

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Planificarea joburilor prin tehnici de multicolorare a grafurilor

Denis-Gheorghe Hrițcu

Sesiunea: iulie, 2025

Coordonator științific

Lector dr. Olariu E. Florentin

Avizat,
Îndrumător lucrare de licență,
Lector dr. Olariu E. Florentin.
Data: ..... Semnătura: .....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Hrițcu Denis-Gheorghe** domiciliat în **România, jud. Suceava, com. Dumbrăveni, str. Nicolae Labiș, nr. 78**, născut la data de **23 aprilie 2003**, identificat prin CNP **5030423330265**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2025, 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 **Planificarea joburilor prin tehnici de multicolorare a grafurilor** elaborată sub îndrumarea domnului **Lector dr. Olariu E. Florentin**, 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. Declar că lucrarea de faţă are exact acelaşi conţinut cu lucrarea în format electronic pe care profesorul îndrumător a verificat-o prin intermediul software-ului de detectare a plagiatului.

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 **Planificarea joburilor prin tehnici de multicolorare a grafurilor**, 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 **Denis-Gheorghe Hrițcu**

Data: ..... Semnătura: .....

Cuprins

Introducere

Introdusă inițial ca o problemă teoretică legată de colorarea hărților geografice, colorarea grafurilor a evoluat într-un mecanism esențial de modelare și rezolvare a unei game diverse de probleme practice. Această problemă are o importanță deosebită în teoria grafurilor, ramură a matematicii discrete, și ilustrează perfect modul în care conceptele teoretice pot fi extinse și aplicate în informatică, optimizare și numeroase alte domenii științifice.

Colorarea grafurilor poate avea aparența unui probleme simple, asignarea de culori unor noduri ale unui graf astfel încât nodurile vecine să nu aibă aceeași culoare. Ne-am înșela însă daca am crede că este trivială, complexitatea acesteia este profundă și oferă o structură riguroasă pentru abordarea problemelor de alocare a resurselor, planificare și optimizare combinatorială.

În ultimele decenii, dezvoltarea tehnologiei și creșterea complexității sistemelor informatice au generat nevoia pentru extensii ale problemei clasice de colorare. Una dintre aceste extensii importante este multicolorarea grafurilor, care permite atribuirea mai multor culori unui singur vârf, reflectând astfel mai fidel realitatea problemelor din lumea reală unde o entitate poate necesita sau poate beneficia de multiple resurse simultan.

Contribuție

Lucrarea de față pornește de la teoria multicolorării și se axează pe rezolvarea problemei de planificare a joburilor. În capitolele ce urmează, voi combina conceptele teoretice de colorare în scopul implementării unor algoritmi de optimizare, algoritmi ce vor fi utilizați într-o aplicație web care permite chiar și utilizatorilor non-tehnici să programeze joburile dorite.

Contribuțiile principale ale lucrării sunt următoarele:

- **Studiul și adaptarea algoritmilor de multicoloring.** Am analizat problema matematică a multicolorării în contextul planificării joburilor și, pe baza acesteia, am dezvoltat și implementat trei algoritmi în Python. Algoritmii au fost proiectați astfel încât să optimizeze cerințele specifice problemei, fiind integrați într-o bibliotecă modulară, ușor de reutilizat, testat, extins și adaptat la noi cerințe. Performanța acestora a fost evaluată printr-o serie de teste comparative, evidențiind avantajele și limitările fiecărui algoritm.

- **Dezvoltarea unei aplicații web.** Având în vedere și utilizatorii non-tehnici, am realizat o aplicație web cu o interfață grafică intuitivă și ușor de utilizat. Aceasta permite testarea și aplicarea algoritmilor de multicolorare în mod interactiv, în funcție de constrângerile introduse de utlizator.

Motivație

Luând în considerare cerințele și scenariile tot mai complexe din lumea reală, se poate concluziona că planificarea tradițională, bazată pe colorarea clasică a grafurilor, nu mai este suficientă. Planificarea joburilor este considerată una dintre problemele centrale în domeniul sistemelor de calcul, al managementului de proiecte și al optimizării proceselor industriale. În toate aceste domenii, dificultatea principală constă în faptul că joburile nu mai pot fi tratate ca entități simple, care necesită o singură resursă la un anumit moment. Un job modern poate necesita simultan mai multe tipuri de resurse (procesor, memorie, rețea, stocare) și poate beneficia de paralelizare pe mai multe unități de procesare.

De ce este multicolorarea grafurilor un model mai potrivit? Răspunsul vine în mod natural: această metodă permite atribuirea mai multor „culori" (resurse) unui singur „vârf" (job). Astfel, se obține o modelare mai fidelă a cerințelor multiple ale fiecărui job și o gestionare mai flexibilă a constrângerilor de prioritate și dependență.

Capitolul 1

Descrierea problemei

1.1 Problema colorării și multicolorării grafurilor

Problema Colorării Grafurilor (GCP) Problema fundamentală în teoria grafurilor, constă în atribuirea de culori nodurilor unui graf astfel încât nodurile vecine să aibă culori diferite, folosind un număr minim de culori.

Definiția 1. *Fie $G = (V, E)$ un graf neorientat, unde:*

- $V = \{v_1, v_2, \ldots, v_n\}$ *este mulțimea vârfurilor*
- $E = \{e_{ij} \mid$ *există muchie între* v_i *și* $v_j\}$ *este mulțimea muchiilor.*

Formal problema constă în determinarea unei partiții a mulțimii V cu un număr minim de clase de culori $\{C_1, C_2, \ldots, C_k\}$, astfel încât pentru fiecare muchie $e_{ij} \in E$, nodurile v_i și v_j să nu aparțină aceleași clase de culori ([10]). O colorare validă a grafului G este o funcție $c : V \to \{1, 2 \ldots l\}$ ce atribuie fiecărui vârf o culoare (număr întreg pozitiv), respectând constrângerea:

$$\forall e_{ij} \in E, |c(v_i) - c(v_j)| > 0([3]) \tag{1.1}$$

Numărul cromati $\chi(G)$ al grafului G este numărul minim de culori necesare pentru o colorare validă a grafului.

$$\chi(G) = \min\{k \mid \exists \text{colorare validă } c : V \to \{1, 2, \ldots, k\}\} \tag{1.2}$$

GCP este NP-hard, așadar de aici existența unor algoritmi euristici ([1], [2]) și de aproximare.

Problema Multicolorarii Grafurilor (MGCP) Reprezintă o generalizare a problemei clasice de colorare a grafurilor, diferența constând în faptul că fiecărui vârf i se atribuie un număr prestabilit de culori diferite, menținând proprietatea că vârfurile adiacente nu pot împărtăși culori comune.

Definiția 2. *Fie $G = (V, E)$ un graf neorientat și $x : V \to \mathbb{N}^*$ o funcție care specifică numărul de culori necesare pentru fiecare vârf, unde:*

- $V = \{v_1, v_2, \ldots, v_n\}$ *este mulțimea vârfurilor*
- $E = \{e_{ij} \mid$ *există muchie între* v_i *și* $v_j\}$ *este mulțimea muchiilor.*
- $x(v_i)$ *este numărul de culori distincte care trebuie atribuite vârfului* v_i*.*

O multicolorare validă a grafului G este o funcție $c : V \to \mathcal{P}(\mathbb{N}^*)$ care atribuie fiecărui vârf un set de culori, respectând constrângerile:

$$\forall v_i \in V,\ |c(v_i)| = x(v_i) \tag{1.3}$$

$$\forall e_{ij} \in E,\ |c(v_i) \cap c(v_j)| = 0 \tag{1.4}$$

unde $\mathcal{P}(\mathbb{N}^*)$ reprezintă mulțimea părților mulțimii numerelor naturale pozitive.

Pentru un vârf v_i cu $x(v_i) = k$, multicolorarea sa poate să fie reprezintă ca: $c(v_i) = \{c_1^i, c_2^i, \ldots, c_k^i\}$, unde $c_1^i < c_2^i < \cdots < c_k^i$ sunt culorile atrubuite în ordine crescătoare.

Numărul cromatic al multicolorării $\chi(G, x)$ este numărul minim de culori necesare pentru o multicolorare validă:

$$\chi_m(G, x) = \min \left\{ k \;\middle|\; \exists \text{multicolorare validă } c \text{ cu } \bigcup_{v \in V} c(v) \subseteq \{1, 2, \ldots, k\} \right\} \tag{1.5}$$

Ca și *GCP* (1.1), problema multicolorării grafurilor este NP-hard, necesitând algoritmi euristici și de aproximare pentru instanțe mari.

1.2 Diferențe între colorare clasică și multicolorare.

Vom analiza în continuare comparația dintre colorarea clasică și multicolorarea grafurilor din mai multe perspective relevante pentru planificarea sarcinilor. Dincolo de observația deja binecunoscută conform căreia, în colorarea clasică, unui vârf i se atribuie o singură culoare, în timp ce în multicoloreare se atribuie un set de culori, există și alte diferențe semnificative:

- **Concepte de preempțiune și non-preempțiune:**
  - Colorarea clasică: Nu include noțiuni de preempțiune, deoarece fiecărui vârf i se alocă o singură culoare, deci execuția echivalentă a sarcinii este considerată continuă și indivizibilă.
  - Multicolorarea: Introduce noțiunile de:
    * Preempțiune: culorile (resursele) atribuite unei sarcini pot fi distribuite în timp, ceea ce înseamnă că sarcina poate fi întreruptă și reluată ulterior.
    * Non-preempțiune: culorile trebuie să formeze un interval contiguu, reflectând faptul că sarcina trebuie să fie executată fără întrerupere.
- **Obiectivele funcției de cost:**
  - Colorarea clasică: Obiectivul principal este minimizarea numărului de culori folosite, ceea ce echivalează cu utilizarea eficientă a resurselor fără conflicte.
  - Multicolorarea: Pe lângă minimizarea culorilor, apar și alte obiective multiple și ierarhizate, specifice problemelor reale de planificare:
    * Maximizarea câștigului (de exemplu prin prioritizarea sarcinilor mai importante),
    * Minimizarea preempțiunii (reducerea fragmentării execuției),
    * Minimizarea intervalului de culori (reducerea timpului total de execuție pentru o sarcină).
- **Restricții privind resursele și acceptarea sarcinilor:**
  - Colorarea clasică: De obicei, nu implică restricții de capacitate pentru fiecare culoare (adică, un număr nelimitat de vârfuri pot primi aceești culoare dacă nu sunt adiacente).
  - Multicolorarea: În aplicațiile de planificare, se regăsesc adesea un număr fix de "mașini" sau resurse partajate, limitând câte vârfuri pot aveea aceeași culoare la un moment dat. De asemenea, introduce problema acceptării/ respingerii sarcinilor, unde nu toate sarcinile pot fi programate.
- **Complexitate în clase specifice de grafuri:** Deși ambele probleme sunt NP-hard în general, complexitatea lor poate diferi semnificativ chiar și pentru clase de grafuri considerate "ușoare" în colorarea clasică ([9]).

- Drumuri (Paths): Colorarea unitară și problemele de makespan sunt triviale, dar problema de multicolorare sumă cu preempțiune este considerată dificilă ([9]).
- Arbori (Trees): Problema de maximizare a sumei în contextul multicolorării preemptive este puternic NP-hard, în timp ce colorarea sumă este ușor de rezolvat ([9]).
- Grafuri Interval: Multicolorarea makespan non-preemptivă este NP-hard și APX-hard, în timp ce colorarea clasică este ușor de rezolvat printr-o metodă greedy ([9]).

1.3 Aplicații relevante ale multicolorării

Problema multicolorării grafurilor este deosebit de importantă în contextul planificării joburilor (job scheduling) și alocării resurselor, extinzând aplicațiile clasice ale colorării grafurilor prin abordarea unor scenarii mai complexe. Următoarele aplicații exemplifică perfect această relavență:

- **Planificarea Joburilor (Job Scheduling).** Multicolorarea oferă un cadru natural pentru modelarea problemelor de planificare din producție, unde:
  - joburile cu durate variate necesită multiple unități de timp consecutive (culori)
  - resursele partajate limitează numărul de joburi care pot rula simultan.
  - resursele critice creează incompatibilități între anumite joburi
  - preempțiunea controlată permite întreruperea și reluarea joburilor.

  Limita așadar a colorării clasice constă în faptul că consideră doar scenariile simplificate unde toate taskurile au durata unitară, multicolorarea pe de altă partea poate surprinde durata variabilă a joburilor și necesitatea de alocare a unor intervale de timp consecutive. Mai multe detalii despre această aplicație vor fi prezentate în capitolul 2.

- **Planificarea orarului (Timetabling).** Dacă am presupune că toate cursurile au aceeași durată (o singură oră) colorarea clasică ar fi suficientă, însă realitate academică este mult mai complexă. Multicolorarea așadar poate permite:

- cursuri cu durate diferite − unele cursuri se rezumă la o oră, altele 2−3 ore.
- cursuri cu apariții multple − același curs poate avea mai multe sesiuni pe săptămână.
- alocarea de intervale consecutive pentru cursurile de lungă durată.

- **Alocarea frecvențelor în comunicațiile fără fir**. În rețelele celulare, problema alocării canalelor ilustrează perfect necesitatea multicolorării:
  - stațiile de bază (noduri) trebuie să gestioneze multiple apeluri simultan.
  - fiecare apel necesită un canal de frecvență dedicat.
  - interferența geografică între stații adiacente interzice folosirea acelorași frecvențe.
  - numărul de canale per stație variază în funcție de traficul de apeluri.

  Deci graful nostru ar reprezinta stațiile de bază prin noduri, iar muchille ar reprezinta adiacența geografică. Fiecare nod necesită atribuirea unui număr de culori (canale) egal cu numărul de apeluri care se conectează la acea stație de bază. Colorarea tradițională nu poate surprinde acest trafic care necesită multiple canale simultan, limitându-se la o singură frecvență per stație.

Capitolul 2

Multicolorarea în planificare joburilor

2.1 Modelarea problemei de job scheduling ca graf

În acest capitol vom începe discuția multicolorării într-un context precis, anume planificarea joburilor. Voi prezenta întâi un exemplu din viața reală pe baza căruia vom construi modelul problemei.

Problema noastră este întâlnită într-o companie internațională de produse cosmetice cu sediul în Geneva. Producția acestor loțiuni de înfrumusețare este caracterizată de cantități mari vândute și de o gamă largă de produse. Sistemul de producție utilizează un flux continuu și este foarte flexibil, așadar ușor de reconfigurat. Este clar că pentru fiecare tip de loțiune este necesară utilizarea simultană a mai multor resurse, cum ar fi: rezervoare cu diferite materii prime, mixere, angajați pentru operare și supervizare, chimiști pentru controlul calității și siguranței. Totuși datorită caracteristicilor lor tehnice, vom numi rezervoarele și chimiștii resurse critice, două produse ce necesită o resursă critică nu pot fi procesate concomitent. Pe lângă resursele critice, avem și resurse pe care le denumim partajate, acestea se referă la echipamente care pot fi utilizate de mai multe joburi simultan. Fiecare "job" se referă la o cantitate (în unități de rezervor) dintr-un anumit tip de produs (loțiune) care trebuie livrată la un centru de distribuție. Fiecare job are un timp de procesare proporțional cu cantitatea de produs și de asemenea are asociat un câștig. Este important de menționat că nu există câștig pentru un job finalizat parțial ([11]).

Acestea fiind spuse, definim formal:

- Graful neorientat de incompabilitate $G = (V, E)$,unde:
  - $V = \{1, 2, \ldots, n\}$ reprezintă mulțimea joburilor (produse de fabricat)

- $E \subseteq V \times V$ reprezintă mulțimea muchiilor de incompabilitate
- $(i, j) \in E$ dacă și numai dacă joburile i și j necesită aceeași resursă critică

Parametrii temporali:

- $D \in \mathbb{N}^* -$ numărul total de unități de timp disponibile (deadline global)
- $p_i \in \mathbb{N}^* -$ timpul de procesare al jobului *i* (numărul de unități de timp necesare)
- $C = \{1, 2, \ldots, D\} -$ mulțimea culorilor disponibile (unități temporale)

- Parametrii de resurse:
  - $l \in \mathbb{N}^* -$ numărul de unități de resurse partajate disponibile
  - $g_i \in \mathbb{R}^+ -$ profitul asociat jobului *i* (marja brută)

2.2 Constrângeri de alocare a resurselor și timpului

Problema P așadar constă în colorarea unui subset al grafului astfel încât să nu existe două vârfuri colorate identice optimizând simultan următoarele obiectivele în ordine lexicografică:

- f_1: maximizarea câștigului total al joburilor în favoarea vârfurilor colorare. Cel mai important obiectiv, fiind direct legat de venitul total.
- f_2: minimizarea numărului total de întreruperi ale joburilor. Ne dorim să nu avem întarziere și munca să fie productivă
- f_3: minimizarea timpului total de flux. Ne dorim reducerea costurilor de inventar, prin acest obiectiv se reduce volumul de "muncă în curs de desfășurare"

Spre deosebire de colorarea clasică, care nu include noțiuni de preempțiune, multicolorarea introduce explicit această noțiune, permițând ca sarcinile să fie întrerupte la puncte de timp întregi și reluate ulterior.Această flexibilitate este crucială în planificare, unde resursele pot fi realocate dinamic. Deși, în general, preempțiunea poate facilita rezolvarea problemelor de planificare, în medii de producție, aceasta poate duce la întreruperi costisitoare (ca întârzieri și muncă neproductivă). De aceea, în multe aplicații practice cum se întamplă în cazul nostru, se urmărește minimizarea numărului de întreruperi, chiar dacă preempțiunea este permisă.

Pentru a defini complet problema P, este necesar să specificăm constrângerile matematice care modelează alocarea reserselor și timpului în cadrul problemei noastre. Ne vom folosi de modelul de graf definit anterior și de următoarele definiții a variabilelor de decizie necesare pentru construirea modelului matematic:

- $z_i = 1$ dacă vârful *i* este colorat complet, 0 altfel.
- $Max_i =$ cea mai mare culoare atribuită vârfului *i*, 0 în cazul în care nu este colorat.
- $Min_i =$ cea mai mică culoare atribuită vârfului *i*, 0 în cazul în care nu este colorat.
- $s_{ic} = 1$ dacă secvența de culori atribuită lui *i* începe sau este continuată (după o întrerupere) cu culoarea *c*, 0 altfel.
- $x_{ic} = 1$ dacă vârful *i* are culoarea *c*, 0 altfel.
- $x_{ic}, z_i, x_{ic} \in \{0, 1\}$, $c \in C, i \in V$
- $Min_i, Max_i \geq 0$, $i \in V$

Constrângerile fundamentale:

- Constrângerea de completare a colorării:

$$\sum_{c=1}^{D} x_{ic} = p_i \cdot z_i, \forall i \in V \tag{2.1}$$

  Asigură că fiecare vârf complet colorat primește exact p_i culori distincte.

- Constrângerea de capacitate a resurselor partajate:

$$\sum_{i \in V} x_{ic} \leq l, \forall c \in C \tag{2.2}$$

  Limitează numărul de joburi care pot fi execute simultan la numărul disponibil de resurse partajate *l*.

- Constrângerea de incompabilitate:

$$x_{ic} + x_{jc} \leq 1, \forall c \in C, \forall (i, j) \in E \tag{2.3}$$

  Muchia (i, j) reprezintă cum am menționat anterior incompabilitatea dintre joburile *i* și *j*. Prin această constrângere ne asigurăm ca joburile incompatibile nu pot fi executate în aceeași unitate de timp.

Constrângerile obiectivelor:

1. **Obiectivul f2 (minimizarea numărului de întreruperi):**

   - Detectarea începutului de secvență:

   $$s_{ic} \geq x_{ic} - x_{i(c-1)}, \forall c \in C, \forall i \in V \tag{2.4}$$

   Această constrângere identifică fiecare reluare a execuției unui job și este folosită pentru a calcula numărul total de întreruperi.

2. **Obiectivul f3 (minimizarea timpului total de flux):**

   - Valoarea maximă:

   $$c \cdot x_{ic} \leq Max_i, \forall c \in C, \forall i \in V \tag{2.5}$$

   - Valoarea minimă:

   $$c \cdot x_{ic} + D \cdot (1 - x_{ic}) \geq Min_i, \forall c \in C, \forall i \in V \tag{2.6}$$

   - Validarea minimului pentru joburile colorate:

   $$D \cdot z_i \geq Min_i, \forall i \in V \tag{2.7}$$

   Prin aceste constrângeri se determină poziția primei și ultimei unități planificate pentru fiecare job

Ținând cont de toate definițiile și constrângerile enumerate până în acest punct, putem defeni modelul matematic al problemei astfel:

Definiția 3.

$$\max \alpha_1 \cdot \sum_{i \in V} (z_i \cdot g_i) - \alpha_2 \cdot \sum_{i \in V} \sum_{c=1}^{D} s_{ic} - \alpha_3 \cdot \sum_{i \in V} (Max_i - Min_i + z_i) \tag{2.8}$$

Unde fiecare termen al sumei corespunde obiectivelor f_1, f_2, f_3 si $\alpha_1, \alpha_2, \alpha_3$ sunt coeficienți pozitivi care definesc importanța fiecărui obiectiv în raport cu celelalte.

2.3 Dificultatea problemei

Problema multicolorării grafurilor, inclusiv varianta specifică pentru planficarea joburilor (problema P), este o problemă NP-hard. Această clasificare indică faptul că,

pentru instanțe de dimensiuni mari, găsirea unui soluții optime într-un timp rezonabil (polinomial) este imposibilă din punct de vedere computațional. Cum putem demonstra caracterul NP-hard al problemei? Prin reducere acesteia la problema k-colorării, care este recunoscută ca fiind NP-completă ([8], [5], [6], [7]), iar rezolvarea problemei P ar însemnă o rezolvare si a problemei k-colorării, lucru ce ar contrazice cunoștiințele din teoria complexității.

Este clar așadar că datorită dificultății acestei probleme, formulările de programare liniară cu numere întregi (ILP) ar putea să fie de folos doar pentru instanțele de dimensiuni mici. Pentru instanțele mai mari, devine necesară adoptarea unor abordări alternative. Aceste metode sunt reprezentate de metodele euristice și metaheuristice, ce oferă soluții rezonabile ca calitate și timp de calcul necesar. În următorul capitol vom vorbi mai amănunțit despre aceste metode, împreună cu o implementare a acestora.

Capitolul 3

Algoritmi implementați

Pentru rezolvarea problemei P, am dezvoltat 3 algoritmi: Greedy (constructiv), Descent (căutare locală), Tabu Search (metaeuristic avansat). Voi parcurge în ordine fiecare algoritm oferind pseudocodul corespunzător, urmat de o descriere intuitivă a logicii și principiului de funcționare. De asemenea, este de menționat că fiecare algoritm primește ca parametrii de bază graful $G = (V, E)$, vectorul de culori p, numărul total de unități de timp/culori D, limita numărului de unități de resurse partajate/culori l și vectorul de câștig g. Mai multe detalii despre metoda prin care acești parametri au fost aleși vor fi prezentate în capitolul următor.

3.1 Algoritmul Greedy

Această euristică constructivă pornește de la graful de incompatibilitate și se bazează pe alegerea, la fiecare pas, a nodului cu cel mai mare câștig. Din mulțimea de culori C sunt selectate doar acele culori care respectă constrângerile fundamentale specifice nodului respectiv, notate cu A_i. Un nod poate fi colorat complet sau deloc; dacă în mulțimea A_i nu există cel puțin $p[i]$ culori admisibile, nodul este exclus din colorare.

Algorithm 1 Greedy

$s \leftarrow$ soluție inițială goală
$V' \leftarrow V$
while $V' \neq \emptyset$ **do**
    $i \leftarrow \max_{v' \in V'} g[v']$
    $V' \leftarrow V' \setminus \{v'\}$
    $A_i \leftarrow$ culorile admisibile pentru i
    if $|A_i| < p[i]$ **then**
        continue
    $B_i \leftarrow$ cel mai mare subset consecutiv din A_i
    if $|B_i| \geq p[i]$ **then**
        Atribuie $p[i]$ culori consecutive aleatorii din B_i
    else
        Atribuie toate culorile din B_i
        Completează cu culori din $A_i \setminus B_i$ minimizând întreruperile și dacă este cazul diferența maximă dintre culori (range-ul)
    actualizează soluția s
return s

Pentru a minimiza întreruperile și diferențele dintre culorile alocate, se caută cel mai mare subset consecutiv de culori din A_i, denumit B_i. Dacă dimensiunea lui B_i este suficientă pentru a acoperi cerința $p[i]$, nodului i i se atribuie $p[i]$ culori consecutive, alese aleatoriu, din B_i. În caz contrar, se atribuie toate culorile din B_i, iar restul necesar este completat cu culori din $A_i \setminus B_i$. Completarea se face astfel încât să se minimizeze întreruperile — considerate binar: 0 dacă noile culori sunt consecutive cu cele deja atribuite, 1 în caz contrar. Atunci când întreruperile nu pot fi complet evitate, se preferă culorile aflate cât mai aproape de capătul superior al lui $p[i]$, pentru a menține diferențele dintre culori cât mai mici. Această alegere are ca scop secundar reducerea timpului total de flux, asociat obiectivului f_3.

3.2 Algoritmul Descent

Algorithm 2 Descent

while numărul de restarturi > 0 **do**
  $s \leftarrow$ soluție inițială generată cu Greedy (vezi Secțiunea 1)
  while nu este îndeplinită o condiție de oprire **do**
    for all $i \in V$ **do**
      if $|A_i| \geq p[i]$ **then**
        $s_i \leftarrow Recolorare^{SD}$ (i)
      else
        $s_i \leftarrow Recolorare^{Enf}$ (i)
      păstrează cea mai bună soluție din vecinătatea $N(s)$
    if s-a găsit o soluție mai bună **then**
      $s \leftarrow$ soluția îmbunătățită
return s

Algorimtul Descent este o metodă de căutare locală cu restarturi multiple. Acesta pornește de la o soluție inițială construită cu o euristică greedy și o îmbunătățește iterativ prin aplicarea unor mutări asupra vârfurilor, în funcție de culorile disponibile. Deoarece problema nu permite colorarea parțială (fiecare vârf trebuie colorat complet cu exact p_i culori), mutările aplicate recolorează/colorează complet un vârf. Procesul continuă până când nu se mai pot obține îmbunătățiri (optimum local) sau până la atingerea unui număr maxim de iterații. La finalul fiecărei runde, dacă soluția curentă este mai bună decât cea optimă cunoscută, aceasta este salvată. Alegerea mutării depinde de numărul de culori admsibile $|A_i|$ pentru un vârf i. Daca $|A_i| \geq p[i]$ se aplică $Recolorare^{SD}$, în caz contrar se aplică $Recolorare^{Enf}$.

Algorithm 3 $Recolorare^{SD}$ (i)

$s' \leftarrow s$
while nodul i nu este colorat complet **do**
  $Z_i \leftarrow \{j \in Vecini(i) \mid j$ necolorat$\}$
  $c \leftarrow \max_{c \in A_i} \sum_{j \in Z_i} u_c(j)$
  Atribuie culoarea c nodului i în s'
  $A_i \leftarrow A_i \setminus \{c\}$
return s'

$Recolorare^{SD}$ are ca scop minimizarea gradului de saturație al vecinilor necolorați ai lui i (numărul de culori diferite utilizate de vecinii săi). La fiecare pas alegem o culoare $c \in A_i$ care maximizează suma $\sum_{j \in Z_i} u_c(j)$, unde Z_i este mulțimea vecinilor necolorații ai nodului i și $u_c(j) = 1$ dacă culoarea este prezentă în vecinii lui j, 0 altfel. Pe scurt, sunt preferate culorile deja „vizibile" în vecinătatea vecinilor lui *i*, pentru a evita creșterea gradului de saturație și, implicit, restrângerea opțiunilor de colorare în pașii următori.

Algorithm 4 $Recolorare^{Enf}$ (i)

$s' \leftarrow s$
$s'[i] \leftarrow A_i$
$C_{nesaturate} \leftarrow \{c \in C \setminus A_i \mid \text{count}(c) < l\}$
for all $c' \in C_{nesaturate}$ **do**
    $pierdere_{c'} \leftarrow \sum_{j \in Vecini(i), c' \in p[j]} g[j]$
Sortează $C_{nesaturate}$ descrescător după $pierdere_{c'}$
while $p_i - |A_i| > 0$ **do**
    for all $c' \in C_{nesaturate}$ **do**
        Decolorează toți vecinii nodului i ce conțin culoare c'
        Atribuie culoarea nodului i în s'
return s'

$Recolorare^{Enf}$ începe prin a atribui toate culorile admisibile vârfului ($|A_i| < p[i]$). Pentru culorile lipsă, identifică culori "ne-saturate", adică culori ce respectă constrângerea de capacitate (apar în mai puțin de l vârfuri). Apoi, se estimează pierderea de câștig (în funcție de ponderea vecinilor afectați) pentru eliminarea acestor culori din vecini. Se selectează culorile cu pierdere minimă și se decolorează vecinii care le folosesc, pentru a permite „furtul" acestor culori și completarea colorației vârfului i.

3.3 Algoritmul Tabu Search

Algorithm 5 Tabu Search

$s \leftarrow$ soluție inițială generată cu Greedy (vezi Secțiunea 1)
$s^* \leftarrow s$
$I \leftarrow 0$
while nu este îndeplinită condiția de oprire **do**
    Generează vecinătatea $N(s)$ conform strategiei alese
    Selectează $(v, s') \in N(s)$, cea mai bună soluție non-tabu
    Actualizează lista tabu pentru nodul v pentru tab iterații
    $s \leftarrow s'$
    if $f(s) < f(s^*)$ **then** ▷ funcția f returnează valorile obiectivelor f1,f2,f3 pentru soluția s
        $s^* \leftarrow s$, $I \leftarrow 0$
    else
        $I \leftarrow I + 1$
    if $I \geq I_{\max}$ **then**
        Decolorează $b\%$ din vârfuri
        $I \leftarrow 0$
return s^*

Tabu Search este un algortim de căutare locală, similar cu algoritmul Descent, care pornește de la o soluție inițială generată prin metodă euristică Greedy (1). La fiecare iterație, se generează vecinătatea soluției curente $N(s)$, adică mulțimea soluțiilor obținute prin modificarea culorii atribuite unui singur vârf din s, respectând constrângerile problemei. Din această vecinătate, se selectează cea mai bună soluție non-tabu, adică o soluție care nu implică mutări interzise temporare. Mutările interzise sunt gestionate printr-o listă tabu, care stochează nodurile modificate (recolorate/colorate) recent, împiedicând revenirea pentru un interval de timp la soluții anterioare și astfel evitând ciclurile sau blocarea în minime locale. Pentru fiecare nod v afectat, lista tabu este actualizată și îi interzice modificarea pentru un număr fix de iterații tab.

Algoritmul include și un mecanism de diversificare. Dacă în $I_{\max}$ iterații consecutive nu se obține nicio îmbunătățire privind cele trei obiective, se decolorează un procentaj $b\%$ din vârfurile soluției curente. Scopul esențial este a de a permite algoritmului explorarea altor regiuni din spațiul soluțiilor.

Algorithm 6 Generarea vecinătății $N(s)$

$N(s) \leftarrow \emptyset$, $Q \leftarrow \emptyset$
if strategia = 2 **then**
    Setează $Recolorare \in \{Recolorare^{Exact}, Recolorare^{SD}\}$ cu probabilități γ și $1 - \gamma$
for all $i \in V'$ **do** ▷ V' reprezintă o submulțime de vârfuri alese aleatoriu din V
    if i este complet colorat și prob < 0.25 **then**
        Aplică $Recolorare^{Drop}(i)$
    if prob < 0.25 **then**
        if $|A_i| < p[i]$ **then**
            Aplică $Recolorare^{Enf}(i)$
        else
            if strategia = 1 **then**
                Aplică $Recolorare^{SD}(i)$ și adaugă i în Q
            else
                Aplică $Recolorare(i)$ ▷ selectat anterior între $Recolorare^{Exact}$ și $Recolorare^{SD}$
if strategia = 1 și $Q \neq \emptyset$ **then**
    Sortează Q descrescător după câștig
    Aplică $Recolorare^{Exact}$ la 25% dintre primii q din Q (aleator)
return $N(s)$

Vecinătatea unei soluții curente s este generată prin aplicarea unor mișcări specifice asupra unui subset aleatoriu de vârfuri dintr-un procentaj prestabilit al mulțimii V, notat cu V' (strategie utilizată pentru a reduce timpul de execuție).

Pentru fiecare $i \in V'$, se aplică una dintre următoarele mișcări, în funcție de strategia selectată și de condițiile locale:

- $Recolorare^{Drop}(i)$, cu probabilitate de 25%, dacă nodul i este complet colorat. Această acțiune elimină toate culorile asociate nodului i.
- Tot cu o probabilitate de 25%, se aplică una dintre următoarele:
  - $Recolorare^{Enf}(i)$, dacă $|A_i| < p[i]$, adică numărul de culori admisibile este insuficient pentru colorarea completă a nodului.
  - **Strategia 1:** Dacă $|A_i| \geq p[i]$, se aplică $Recolorare^{SD}(i)$, iar nodul i este adăugat în lista Q. După parcurgerea tuturor vârfurilor, lista Q este sortată

descrescător după câștig, iar $Recolorare^{Exact}$ este aplicată pentru 25% dintre primele q noduri (selectate aleatoriu).

- **Strategia 2:** Dacă $|A_i| \geq p[i]$, se aplică fie $Recolorare^{SD}(i)$, fie $Recolorare^{Exact}(i)$, cu probabilități γ și $1-\gamma$, respectiv.

În urma aplicării acestor mișcări, mulțimea soluțiilor obținute formează vecinătatea $N(s)$ a soluției curente s, care este apoi returnată și utilizată în cadrul algoritmului Tabu Search (vezi Secțiunea 5).

Algorithm 7 $Recolorare^{exact}$ (i)

for $q = 1$ **to** p_i **do**
  for all $c \in A_i$ **do**
    if $q = 1$ **then**
      $BestS(q,c) \leftarrow \{c\}, \quad Lint(q,c) \leftarrow 0, \quad Lrg(q,c) \leftarrow 1$
    else
      $Best_{int} \leftarrow \infty$, $Best_{rg} \leftarrow \infty$, $B \leftarrow \emptyset$
      for fiecare $c' \in A_i$ cu $c' < c$ **do**
        if $c - c' = 1$ **then**
          $Int \leftarrow Lint(q-1, c')$
        else
          $Int \leftarrow Lint(q-1, c') + 1$
        if $Int = Best_{int}$ **then**
          $B \leftarrow B \cup \{c'\}$
        else if $Int < Best_{int}$ **then**
          $Best_{int} \leftarrow Int$, $B \leftarrow \{c'\}$
      for all $c' \in B$ **do**
        $Rg \leftarrow Lrg(q-1, c') + (c - c')$
        if $Rg = Best_{rg}$ **then**
          $c^* \leftarrow c'$ (cu o anumită probabilitate)
        else if $Rg < Best_{rg}$ **then**
          $c^* \leftarrow c'$, $Best_{rg} \leftarrow Rg$
      $BestS(q,c) \leftarrow BestS(q-1, c^*) \cup \{c\}$
      $Lint(q,c) \leftarrow Best_{int}$, $Lrg(q,c) \leftarrow Best_{rg}$
return $BestS$

Algoritmul $Recolorare_{exact}$ are ca scop generarea unei secvențe optime de culori pentru un vârf i, minimizând două obiective: **numărul de întreruperi (f_2)** și **diferența de interval (range) (f_3)**. O *întrerupere* este definită ca situația în care două culori consecutive dintr-o secvență au o diferență strict mai mare decât 1. *Range-ul* reprezintă diferența dintre cea mai mică și cea mai mare culoare din secvență.

Algoritmul parcurge secvențe de lungime q de la 1 la p_i (lungimea maximă dorită) și, pentru fiecare lungime q, evaluează fiecare culoare $c \in A_i$ (mulțimea culorilor admisibile pentru vârful i) ca potențială ultimă culoare a unei secvențe optime. Se utilizează trei structuri de date: $BestS(q, c)$ pentru a reține secvența optimă, $Lint(q, c)$ pentru numărul de întreruperi și $Lrg(q, c)$ pentru range-ul corespunzător.

Pentru $q = 1$, fiecare culoare $c \in A_i$ inițializează o secvență formată doar din $\{c\}$. În acest caz, nu există întreruperi, deci $Lint(1, c) = 0$. Range-ul unei singure culori este 1, deci $Lrg(1, c) = 1$. $BestS(1, c)$ devine $\{c\}$.

Pentru $q > 1$, algoritmul încearcă să extindă secvențele valide de lungime $q - 1$ prin adăugarea culorii curente c. Procesul se face în doi pași efectuați în ordine:

1. **Minimizarea întreruperilor (f_2):** Se evaluează fiecare $c' \in A_i$ cu $c' < c$. Numărul de întreruperi (Int) este calculat ca $Lint(q - 1, c')$ dacă $c - c' = 1$ (fără o nouă întrerupere) sau $Lint(q - 1, c') + 1$ dacă $c - c' \neq 1$ (o nouă întrerupere). Se construiește un set B care conține toate culorile c' ce minimizează acest Int.

2. **Minimizarea range-ului (f_3):** Dintre culorile c' aflate în setul B, se selectează culoarea c^* care minimizează $Rg = Lrg(q - 1, c') + (c - c')$. Acesta reprezintă range-ul noii secvențe de lungime q.

3. **Gestionarea Egalității:** În cazul în care mai multe culori c' din setul B conduc la același range minim, selecția lui c^* se face aleatoriu, fiecare candidat având o probabilitate egală de $1/|B|$ de a fi ales.

Capitolul 4

Experimente și rezultate

4.1 Măsurători și metrici

Inițial, vom discuta despre cum arată instanțele de test in ceea ce privește parametri de bază ai algoritmilor: graful $G = (V, E)$, vectorul de culori p, numărul total de unități de timp/culori D, limita numărului de unități de resurse partajate/-culori l și vectorul de câștig g. Pentru generarea grafului se folosesc doi parametri: numărul n de vârfuri și o densitate a grafului d (probabilitate de avea o muchie între două noduri), conform modelului Erdős-Rényi ([4]). Am considerat $n \in \{30, 150, 300\}$ și $d \in \{0.2, 0.5, 0.8\}$. Pentru fiecare combinație (n,d) sunt generate câte 10 instanțe. Numărul de culori p_i a unui vârf i este un număr întreg pozitiv ales aleatoriu în intervalul $[1, 10]$. Pentru a simula scenariile reale unde câștigul este în strânsă legătură cu timpul de procesare (p_i), cu cât e un timp de procesare mai mare cu atât și câștigul este mai mare. Prin urmare $g_i = \beta \cdot p_i$, unde β este ales aleatoriu în intervalul $[1, 20]$. Limita de resurse l este setată la 25.

În privința lui D (numărul total de culori) am dori să aibă o valoarea ce nu permite colorarea completă pentru a vedea comportamentul algoritmilor cât mai bine. Pentru a determina o valoare potrivită a lui D, a fost creat un script care, pentru fiecare combinație (n, d), generează 100 de instanțe folosind algoritmul Greedy (1). Pentru fiecare D, se calculează procentul mediu de colorare, precum și intervalul [min, max] al procentelor de colorare obținute. Valoarea aleasă pentru testare este cel mai mare D care nu a permis atingerea unui grad de colorare de 100% în niciuna dintre instanțe.

Tabela 4.1: Rezultatele generării valori D

n	d	D	Procent colorat mediu	Interval
30	0.2	24	85.5%	[70.0% − 96.7%]
30	0.5	42	83.7%	[70.0% − 96.7%]
30	0.8	68	82.0%	[60.0% − 96.7%]
150	0.2	63	85.1%	[79.3% − 91.3%]
150	0.5	132	84.6%	[74.7% − 91.3%]
150	0.8	239	82.8%	[72.0% − 91.3%]
300	0.2	101	85.8%	[81.7% − 90.0%]
300	0.5	228	85.8%	[78.3% − 90.0%]
300	0.8	423	84.7%	[79.0% − 90.0%]

Import de menționat că deși valorile prezentate în tabel sunt rezultatul a 100 de instanțe, generarea grafului dar și algoritmul Greedy sunt realizate prin mecanisme predominat aleatorii, deci aceste date pot varia. Aceste aspecte aleatorii datorează și variața valorilor din intervale.

Algoritmii supuși instanțelor de test implică, pe lângă parametrii de bază, și o serie de parametri suplimentari specifici fiecărui algoritm, după cum urmează:

1. **Greedy** 1: utilizează exclusiv parametrii de bază ai instanței, ceea ce explică simplitatea implementării și eficiența în ceea ce privește timpul de execuție.

2. **Descent** 2: necesită definirea unui număr maxim de iterații și a unui număr de restarturi. Aceste valori au fost stabilite prin rulari consecutive pentru fiecare combinație (n, d), pana cand s-a gasit pragul in care creșterea celor două variabile nu produce nicio îmbunătățire (cu o limită de timp de $60 \cdot n$ secunde).

3. **Tabu Search** 5: similar algoritmului Descent, acest algoritm primește un număr maxim de iterații, stabilit conform aceleiași logici de limitare temporală. În plus, include următorii parametri:

   - dimensiunea numărului de mutări interzise (tab): setată la 3 pentru $n = 30$ și la 10 pentru $n \in \{150, 300\}$;
   - Parametrul de diversificare / procentul de decoloare (b): 20%;
   - numărul maxim de iterații fără îmbunătățire ($I_{\max}$): 150;

- numărul de noduri selectate pentru $Recolorare^{Exact}$ (q): 20;
- procentajul de vârfuri utilizat în generarea vecinătății (vezi 6): 25%;
- Probabilitate aplicări $Recolorare^{SD}$ în cadrul celei de a doua strategii (γ): 0,2.

Valorile acestor parametri au fost determinate în urma unei etape de ajustare, bazată pe rulări preliminare ale algoritmului pe un set reprezentativ de instanțe.

Pentru rularea scripturilor, a fost utilizată o mașină virtuală prin intermediul serviciului *Google Cloud*, de tip *e2-highcpu-16* (16 vCPUs, 16 GB RAM). Alegerea acestui tip de instanță a fost determinată de necesitatea unei puteri de procesare ridicate și a unui număr mare de nuclee, având în vedere că scriptul lansează în paralel cele 10 instanțe de test, fiecare rulând un algoritm complex, ceea ce presupune o utilizare intensivă a resurselor CPU.

Instanțele de test vor fi încadrate in 3 tabele distincte, grupate în funcție de numărul de vârfuri n astfel încât să se evidențieze comportamentul algoritmilor în cadrul instanțelor de dimensiuni reduse, medii și mari. Pentru fiecare combinație (n, d) și pentru fiecare algoritm analizat: Greedy, Descent, TS1(Tabu Search cu strategia 1), TS2(Tabu Search cu strategia 2) sunt acordate două coloane: o coloană cu cel mai bun rezultat și o coloană cu rezultatul mediu din cele 10 instanțe. Rezultatele o să fie de forma unui triplet (scor obiectiv f_1,scor obiectiv f_2,scor obiectiv f_3) însoțit de raportul dintre numărul de vârfuri colorate și numărul total de vârfuri.

Tabela 4.2: Rezultate pentru $n = 30$

d	Algoritm	Cea mai bună rulare		Medie	
		(f_1, f_2, f_3)	Nr. noduri colorate	$(\overline{f_1}, \overline{f_2}, \overline{f_3})$	Nr. noduri colorate
0.2	Greedy	(1747, 5, 185)	22/30	(1623.10, 2.70, 148.40)	20.8/30
	Descent	(1808, 30, 313)	25/30	(1762.10, 30.90, 296.90)	24/30
	TS1	(1808, 22, 243)	25/30	(1771.40, 18.80, 242.00)	23.6/30
	TS2	(1808, 58, 332)	25/30	(1782.70, 40.40, 297.00)	23.6/30
0.5	Greedy	(1614, 7, 170)	26/30	(1555.30, 5.60, 169.30)	24.89/30
	Descent	(1704, 40, 464)	29/30	(1671.50, 26.60, 365.70)	28/30
	TS1	(1704, 31, 384)	29/30	(1684.60, 30.30, 378.00)	27.7/30
	TS2	(1704, 32, 409)	29/30	(1669.70, 53.30, 492.70)	28.7/30
0.8	Greedy	(1617, 7, 169)	27/30	(1588.70, 6.10, 198.80)	26.4/30
	Descent	(1644, 43, 749)	28/30	(1631.40, 42.00, 699.20)	28/30
	TS1	(1688, 35, 620)	29/30	(1646.30, 26.20, 473.10)	28/30
	TS2	(1644, 38, 556)	28/30	(1644.00, 43.80, 608.40)	28.1/30

Tabela 4.3: Rezultate pentru $n = 150$

d	Algoritm	Cea mai bună rulare		Medie	
		(f_1, f_2, f_3)	Nr. noduri colorate	$(\overline{f_1}, \overline{f_2}, \overline{f_3})$	Nr. noduri colorate
0.2	Greedy	(8075, 17, 736)	138/150	(7994.20, 25.40, 882.40)	134.7/150
	Descent	(8358, 193, 3259)	148/150	(8335.20, 240.10, 3330.10)	148.5/150
	TS1	(8364, 77, 1421)	149/150	(8326.80, 64.40, 1298.60)	147/150
	TS2	(8364, 68, 1537)	149/150	(8341.70, 68.50, 1393.00)	148/150
0.5	Greedy	(8517, 53, 1903)	129/150	(8253.50, 43.70, 1608.70)	124.6/150
	Descent	(8862, 296, 7593)	147/150	(8807.70, 284.40, 7148.60)	145.9/150
	TS1	(8862, 130, 3335)	147/150	(8802.50, 133.00, 3350.20)	144.9/150
	TS2	(8891, 172, 4249)	148/150	(8829.20, 155.50, 3720.80)	145.4/150
0.8	Greedy	(7655, 42, 2247)	131/150	(7579.80, 41.80, 1904.30)	126.8/150
	Descent	(7992, 280, 11956)	149/150	(7964.10, 250.90, 11422.90)	147.8/150
	TS1	(7985, 150, 5995)	148/150	(7953.90, 144.50, 5258.40)	146.7/150
	TS2	(8006, 212, 7657)	150/150	(7991.30, 188.10, 6812.80)	148.5/150

Tabela 4.4: Rezultate pentru $n = 300$

d	Algoritm	Cel mai bun rezultat		Rezultat mediu	
		(f_1, f_2, f_3)	Nr. noduri colorate	$(\overline{f_1}, \overline{f_2}, \overline{f_3})$	Nr. noduri colorate
0.2	Greedy	(16746, 61, 2431)	255/300	(16609.60, 69.50, 2420.80)	254.2/300
	Descent	(17772, 635, 12001)	297/300	(17693.80, 612.70, 11860.80)	294.7/300
	TS1	(17602, 200, 4231)	288/300	(17520.90, 182.80, 4178.20)	286.9/300
	TS2	(17638, 186, 4599)	289/300	(17534.10, 187.40, 4546.00)	287.1/300
0.5	Greedy	(16705, 80, 3758)	251/300	(16633.50, 89.40, 4227.80)	251/300
	Descent	(17533, 640, 25642)	291/300	(17480.90, 626.50, 25436.50)	290.4/300
	TS1	(17466, 307, 10714)	287/300	(17336.40, 275.90, 10164.40)	282.7/300
	TS2	(17510, 270, 10697)	288/300	(17380.40, 252.00, 9754.40)	284/300
0.8	Greedy	(15227, 105, 5457)	258/300	(15132.40, 98.50, 5706.40)	259.5/300
	Descent	(15646, 330, 22361)	291/300	(15610.20, 319.10, 23408.60)	288.5/300
	TS1	(15488, 125, 9583)	282/300	(15436.10, 149.80, 10756.60)	279.0/300
	TS2	(15556, 174, 14805)	285/300	(15444.40, 131.40, 9697.10)	277.8/300

4.2 Interpretarea rezultatelor și concluzii parțiale

Împărțirea tabelelor în funcție de dimensiunea are ca scop principal vizualizare eficienței algoritmilor în diferite contexte. Timpul de rulare a fost omis deoarece accentul este pus pe calitatea soluțiilor obținute. Așa cum am menționat și în argumentarea numărului de iterații, parametrii de configurare au fost aleși astfel încât timpul de rulare să fie comparabil între algoritmi (cu excepția algoritmului Greedy, care finalizează mult mai rapid).

Acum vom primi rezultatele din mai multe puncte de vedere:

- **Numărul nodurilor colorate.** Algoritmii Descent, TS1, TS2 sunt semnificativ mai performanți în colorarea unui număr mai mare de noduri comparativ cu algoritmul Greedy, indiferent de dimensiunea grafului sau de densitatea acestuia. Densitatea grafului (d) pare să influențeze pozitiv numărul de noduri colorate, sugerând că, pentru grafuri mai dense, acești algoritmi reușesc să maximizeze utilizarea culorilor.

- **Obiectivul f_1 (maximizarea câștigului)** Algoritmii TS1 și TS2 domină pentru $n = 30$ și $n = 150$, dar pentru $n = 300$, Descent obține cel mai bun scor mediu. Această performanță superioară a lui Descent pentru grafuri mari se datorează probabil strategiei sale de multi-start. Chiar dacă mecanismul său iterativ de îmbunătățire a soluției poate nu este la fel de sofisticat ca al algoritmilor Tabu Search, pornind din multiple soluții inițiale generate de Greedy, Descent are șansa de a găsi o soluție inițială excepțional de bună. Pentru grafuri foarte mari, cu un spațiu de căutare imens, această capacitate de a exploata diverse puncte de pornire poate depăși avantajul mecanismelor iterative de rafinare ale TS1 și TS2.

- **Obiectivul f_2 (minimizarea numărului de întreruperi și Obiectivul f_3 (minimizarea timpului de flux.))** Valorile f_2 și f_3 sunt semnificativ mai mici pentru Greedy, dar nu indică o performanță superioară ci mai mult consecința colorării unui număr redus de noduri. Prin încercările algoritmilor Descent,TS1 și TS2 de a obține un câștig mai mare (corelat cu numărul de noduri colorate) generează inevitabil mai multe întreruperi și diferențe mai mari între culori. Aceste obiective cresc o dată cu numărul de noduri, dar și mai clar a densității pentru că apar constrângeri mai complexe ce necesită optimizarea multi-obiectiv.

Analizând în detaliu rezultatele, putem concluziona că metodele TS1 și TS2 (cu un ușor avantaj pentru TS2) oferă performanțe superioare în ceea ce privește calitatea soluțiilor, atât din perspectiva numărului de noduri colorate, cât și în raport cu cele trei obiective de optimizare. Algoritmul Descent rămâne competitiv în special în maximizarea câștigului, în timp ce Greedy se remarcă prin viteza de execuție, oferind o soluție rapidă, dar cu performanțe mai reduse în optimizarea calității.

Capitolul 5

Dezvoltarea aplicației web

În acest capitol, conceptele teoretice prezentate anterior și algoritmii de multicolorare implementați sunt integrați într-o aplicație practică: o aplicație web care permite utilizatorilor să își planifice joburile, utilizând tehnici de multicolorare a grafurilor pentru optimizarea alocării resurselor și a timpului.

5.1 Arhitectura generală a aplicației

Aplicația creată pornește de la arhitectura clasică MERN (acronim provenit de la cele patru componente principale: MongoDB, Express.js, React.js, Node.js), extinsă prin integrarea algoritmilor descriși în capitolul 3, implementați în Python.

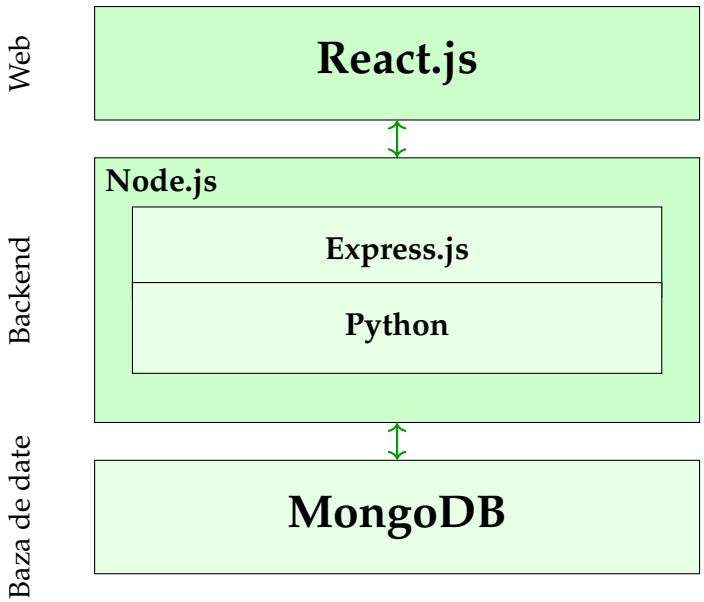

Figura 5.1: Arhitectura aplicației web

O să analizăm figura 5.1 pornind de jos în sus.

Baza de date MongoDB

MongoDB este o bază de date NoSQL (neralațională), flexibilitatea ei permite o stocare eficientă a datelor necesare planificării joburilor. Astfel prin intermediul ei se stochează:

- informațiile despre utilizatori (cu date de autentificare și profil),
- datele despre planificarea joburilor (parametrii necesari precum numărul de resurse partajate, detalii despre joburi, incompabilitățile)
- rezultatele algoritmilor de planificare

Biblioteca folosită pentru interacțiunea cu baza de date, simplificând operațiile CRUD este *Mongoose*.

Backend Node.js

Node.js a fost ales ca mediu de execuție pentru backend datorită simplității și a ecosistemului bogat de pachete NPM, care susține dezvoltarea rapidă de funcționalități pentru planificarea și gestionarea joburilor Utilizarea JavaScript pe ambele părți (frontend și backend) reduce complexitatea dezvoltării și facilitează întreținerea codului. Express.js completează această alegere ca framework web minimalist, oferind structura necesară pentru API-urile RESTful care gestionează operațiile de planificare, monitorizare și control al job-urilor. Dependențele cheie utilizate în backend sunt:

- *bcrypt.js* pentru hash-uirea și securizarea parolelor utilizatorilor.
- *child_process* vital pentru aplicația noastră deorece permite rularea proceselor externe cum ar fi scripturile Python care implementează algoritmii de multicolorare.
- *jsonwebtoken și cookie-parser*: Pentru implementarea autentificării bazate pe tokenuri JWT și gestionarea sesiunilor utilizatorilor.
- *cors*: Middleware care permite comunicarea între frontend și backend când sunt pe domenii diferite.

Web React.js

React.js a fost ales pentru dezvoltarea interfeței utilizator (UI) deoarece arhitectura sa bazată pe componente permite construirea unor interfețe complexe și interactive într-un mod organizat și scalabil. În contextul unei aplicații de planificare a joburilor, unde utilizatorii interacționează cu grafuri și trebuie să vizualizeze și să modifice datele, utilizarea React.js facilitează implementarea unor elemente UI dinamice și oferă o experiență de utilizare fluidă și intuitivă. Dependențele ce merită menționate sunt:

- *axios* pentru cereri HTTP asincrone către backend.
- *html-to-image* permite generare unui imagini (în cazul nostru, PNG) dintr-un element HTML. Este folosită pentru a permite utilizatorilor să exporte rezultate sub formă de imagine.
- *papaparse* permite parsarea unui fisier CSV în JavaScript. Este utilizată pentru a permite utilizatorilor să importe date despre planificarea dorită.
- *reactflow* – biblioteca de bază pentru aplicația construită. Prin intermediul ei se realizează și se vizualizează grafurile ce reprezintă planficarea utilizatorului.

5.2 Prelucrarea și introducerea datelor în aplicație

5.2.1 Modelul datelor în backend și legătura cu problema planificării

Pentru a reprezenta modelul teoretic al grafului de incompabilitate și al parametrilor asociați (prezentați în capitolul 2), datele sunt stocate în baza de date MongoDB folosind schema următoare:

- *Joburile V* sunt modelate prin obiecte în lista jobs, fiecare având câmpurile: processing_time (p_i), gain (g_i).
- Muchiile de incompabilitate *E* sunt reprezentate în lista conflicts, fiecare conflict indicând două joburi incompatibile.
- Parametrii globali precum deadline-ul D și numărul de resurse l sunt stocați ca atribute directe ale planificării.

Listing 5.1: Schema modelului de date pentru planificare

```
  const embeddedJobSchema = new mongoose.Schema({
  _id: { type: String, required: true },
  name: { type: String, required: true },
  processing_time: { type: Number, required: true },
  gain: { type: Number, required: true },
  position: {
    x: { type: Number, required: true },
    y: { type: Number, required: true },
  },
}, { _id: false });

const embeddedConflictSchema = new mongoose.Schema({
  _id: { type: String, required: true },
  job1: { type: String, required: true },
  job2: { type: String, required: true },
}, { _id: false });

const scheduleSchema = new mongoose.Schema({
  userId: { type: mongoose.Schema.Types.ObjectId,
  ref: "User", required: true },
  name: { type: String, required: true },
  jobs: [embeddedJobSchema],
  conflicts: [embeddedConflictSchema],
  l: { type: Number, required: true },
  D: { type: Number, required: true },
  created_at: { type: Date, default: Date.now },
  updated_at: { type: Date, default: Date.now },
});

const Schedule = mongoose.model("Schedule", scheduleSchema);
export default Schedule;
```

5.2.2 Modelul pentru stocarea rezultatelor algoritmilor de planificare

Pentru a salva rezultatele algorimilor prezentați în capitolul 3 împreună cu scorurile funcțiilor obiectiv (f_1,f_2,f_3) astfel încat utilizatorul să poate vizualiza rezultatele, am definit următorul modul Mongoose:

Listing 5.2: Schema modelului de date pentru rezultatul planificării

```
const scheduleSchemaResultSchema = new mongoose.Schema({
  userId: { type: mongoose.Schema.Types.ObjectId,
  ref: "User", required: true },
  schemaId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "Schedule",
    required: true,
  },
  fully_colored_jobs: [{ type: String }],
  color_map: { type: mongoose.Schema.Types.Mixed,
  default: {} },
  f1: { type: Number },
  f2: { type: Number },
  f3: { type: Number },
  algorithm_used: { type: String },
  timestamp: { type: Date, default: Date.now },
});

const ScheduleSchemaResult = mongoose.model(
  "ScheduleSchemaResult",
  scheduleSchemaResultSchema
);
export default ScheduleSchemaResult;
```

5.2.3 Introducerea datelor

Acum vom urmări fluxul datelor de la interfața grafică până la backend, iar apoi la algoritmii de planificare. Utilizatorul are două metode prin care poate introduce

datele despre planificare:

1. Manual, prin adaugăre joburilor și trasarea incompabilităților.

2. Import, prin intermediul unui fișier CSV.

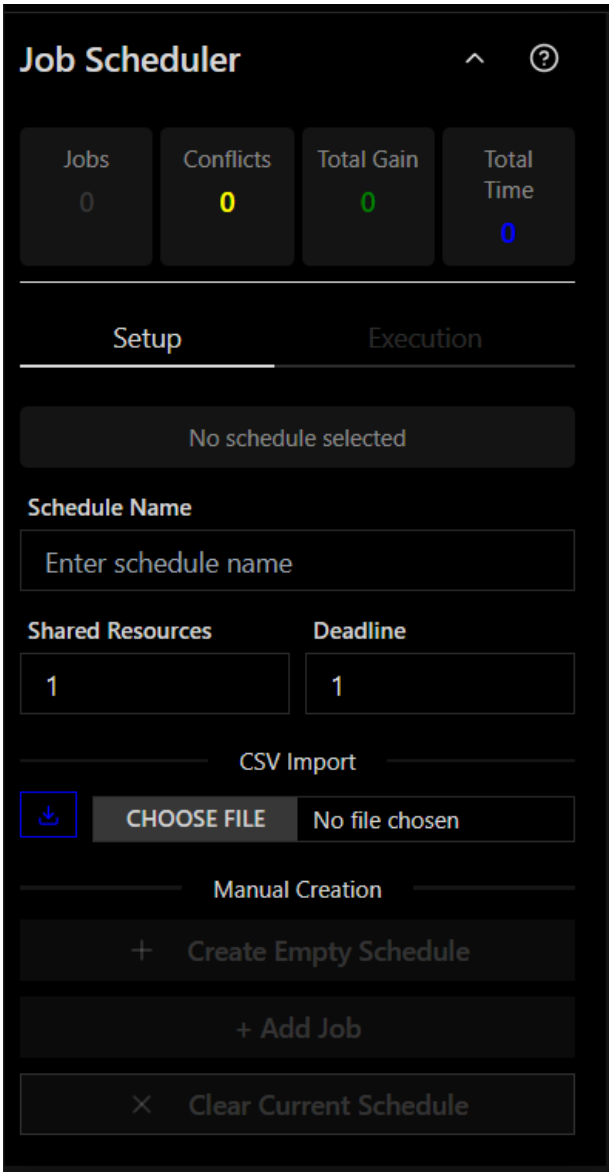

Figura 5.2: Introducerea datelor

Procesul de introducere a datelor respectă un flux logic pentru a asigura consistența și validitatea acestora. În primul rând, utilizatorul trebuie să creeze un nou obiect de planificare (`Schedule`), fără de care nu se pot adăuga joburi. Odată creat, utilizatorul are la dispoziție două metode pentru a introduce joburi și conflicte: manual (prin interfața grafică) sau prin importul unui fișier CSV. Pentru a facilita această operațiune, aplicația permite descărcarea unui template de CSV, iar fișierul importat este supus

unei funcții de parsare și validare. În cadrul acestei etape, datele din CSV sunt verificate pentru consistență (de exemplu, nu sunt permise joburi cu nume duplicate), iar utilizatorul poate vizualiza un *preview* al conținutului înainte de salvare. De asemenea, backend-ul implementează reguli stricte de validare a datelor introduse. Această abordare asigură că datele transmise algoritmilor de planificare respectă structura necesară pentru generarea unor rezultate valide.

5.2.4 Prelucrarea datelor

După ce utilizatorul a introdus datele despre planificare (joburi, conflicte și parametri globali), acestea sunt salvate în baza de date prin modelul *Schedule* 5.1 iar când utilizatorul dorește generarea unei planificării, backend-ul extrage aceste date și le pregătește într-un format compatibil cu algorimtii implementați în Python. Pe scurt, datele sunt transformate într-un obiect JSON care este trimis prin stdin către un script Python. Acesta preia informațiile și le convertește într-un mod compatibil cu algoritmii. Rezultatul returnat este de asemenea un JSON ce este primit,validat și stocat în *ScheduleSchemaResult* 5.2.Ulterior, rezultatul poate fi vizualizat de către utilizator în interfața grafică și de asemenea descărcat sub formă unei imagini.

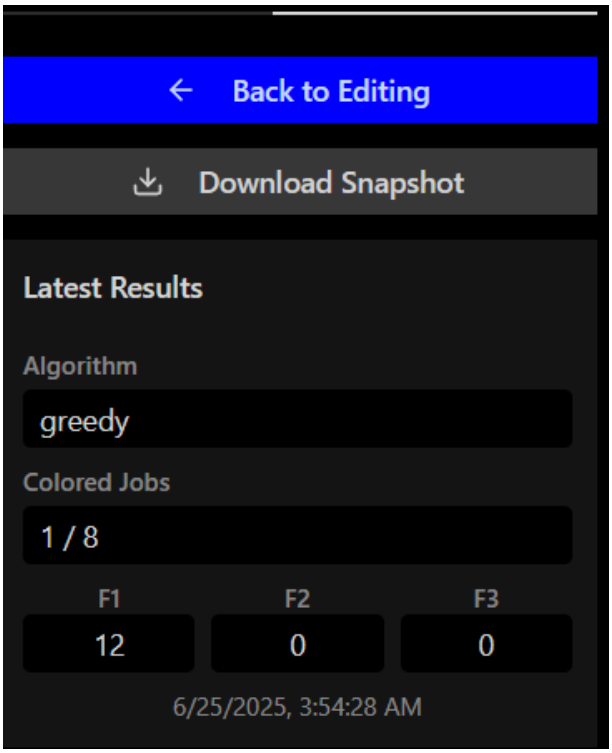

Figura 5.3: Rezultatul unei rulări

5.3 Modulul de multicolorare și interfața utilizator

În secțiunea anterioară am prezentat fluxul de introducere și prelucrare a datelor, însă nu am detaliat modul de afișare grafică a joburilor — componentă esențială a interfeței aplicației. Aceasta cuprinde două stări distincte: modul de editare, în care utilizatorul configurează planificarea, și modul de afișare a rezultatului, în care vizualizează soluția generată de algoritmi.

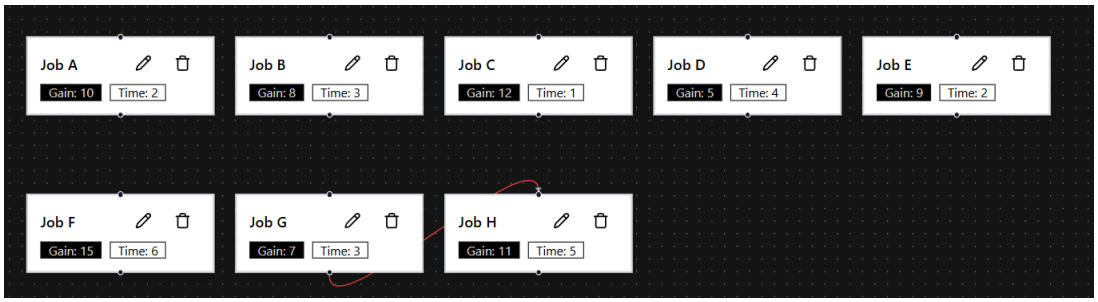

Figura 5.4: Modul de editare al joburilor

În figura 5.4 este ilustrat modul de editare. În această etapă, utilizatorul poate adăuga și poziționa noduri (joburi), poate trasa muchii de incompatibilitate prin conectarea a două noduri, precum și edita parametrii unui job (timp de procesare, câștig, denumire) sau șterge complet un job existent.

Figura 5.5: Afișarea rezultatului algoritmului de multicolorare

În figura 5.5 este redat modul de afișare a rezultatului. După rularea unui algoritm de planificare, nodurile colorate complet sunt evidențiate în verde, iar fiecare job afișează culoarea alocată de algoritm, reprezentând resursa la care a fost atribuit.

Concluzii

Lucrarea de față a avut ca scop proiectarea, implementarea și testarea unei soluții pentru problema planificării joburilor folosind tehnici de multicolorare a grafurilor. În prima parte, am prezentat fundamenetele teoretice ale problemei, precum și algoritmii spefici de multicolorare ce permit optimazarea dorită în cadrul planificării joburilor și anume de alocare de resurse întru-un mod eficient și respectând constrângerile de incompabilitate între joburi.

Partea practică a constat în dezvoltarea unui aplicații web, integrând algoritmii implementați în Python cu un backend Node.js și o interfață utilizator construită în React.js. Mediu în care manipularea datelor, interacțiunea utilizatorilor cu aceste date și vizualizarea lor se desfășoară într-un mod simplu și eficient.

În concluzie, lucrarea evidențiază importanța colaborării dintre teorie algoritmică și dezvoltarea software pentru crearea unor soluții eficiente și practice.

Bibliografie

[1] D. Costa, A. Hertz, and O. Dubuis. Embedding of a sequential procedure within an evolutionary algorithm for coloring problems in graphs. *Journal of Heuristics*, 1:105–128, 1995.

[2] R. Dorne and J. K. Hao. A new genetic local search algorithm for graph coloring. In *Proceedings of PPSN'98*, Lecture Notes in Computer Science, 1998. To appear.

[3] Raphael Dorne and Jin-Kao Hao. Tabu search for graph coloring, t-colorings and set t-colorings. In *Proceedings of a Conference on Graph Theory or Combinatorics*, 2000.

[4] Paul Erdős and Alfréd Rényi. On random graphs. i. *Publicationes Mathematicae*, 6(3–4):290–297, 1959. Archived from the original on 2020-08-07. Retrieved 2011-02-23.

[5] H. Gabow and O. Kariv. Algorithms for edge coloring bipartite graphs and multigraphs. *SIAM Journal on Computing*, 11(1):117–129, February 1992.

[6] R. Gandhi, M. M. Halldórsson, G. Kortsarz, and H. Shachnai. Approximating non-preemptive open-shop scheduling and related problems. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP)*, 2004.

[7] R. Gandhi, M. M. Halldórsson, G. Kortsarz, and H. Shachnai. Improved bounds for sum multicoloring and weighted completion time of dependent jobs. Unpublished manuscript, 2004.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[9] Magnus M. Halldórsson and Guy Kortsarz. Multicoloring: Problems and techniques. In *Lecture Notes in Computer Science*. Springer, 2004. Conference paper, August 2004.

[10] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[11] Simon Thevenin, Nicolas Zufferey, and Jean-Yves Potvin. Graph multi-coloring for a job scheduling application. *European Journal of Operational Research*, 255(1):128–141, 2016.