Universitatea "Alexandru Ioan Cuza", Iaşi

Facultatea de Informatică

Student: Isac Lucian-Constantin

Coordonator: Lector dr. Olariu E. Florentin

Sesiunea iulie 2024

Bandwith coloring problem și aplicații

Contents

Introducere

Bandwidth coloring problem (BCP) poate fi considerată un puzzle de optimizare combinatorie de dimensiuni notabile; având amprente în mai multe domenii practice, cum ar fi planificarea frecvenței radio sau alocarea canalelor de comunicație și chiar ingineria rețelelor. Esența problemei este colorarea vârfurilor unui graf cu constrângeri specifice privind diferența de culori dintre vârfurile învecinate: ne propunem să minimizăm numărul total de culori utilizate, asigurând în același timp această proprietate de optimitate locală pentru aplicabilitatea globală pe întreaga structură a graficului.

Modalitățile de abordare tradiționale ale BCP, cum ar fi algoritmii exacți sau metaeuristica clasică, nu sunt capabile să gestioneze instanțe mari sau grafice structurate complex. Într-o astfel de situație, metodele bazate pe învățarea automată și tehnicile metaeuristice avansate sunt capabile să atragă mai mult interes datorită posibilității lor de a oferi soluții eficiente care pot fi scalate bine.

Pentru rezolvarea Bandwidth coloring problem propunem metodă prezentată în lucrarea ”Learning Based Path Relinking” - LPR. LPR îmbină învățarea automată cu operatorii de path relinking și căutarea Tabu pentru a optimiza soluțiile obținute în urmă aplocarii operatorilor.

Obiectivul principal al acestei lucrări este de a demonstra eficacitatea și performanța metodei LPR. Pentru a atinge acest obiectiv, vom prezenta o descriere detaliată a algoritmului propus, urmată de o serie de experimente pe instanțe de referință ale BCP. Rezultatele experimentale vor evidenția avantajele LPR în termeni de calitate a soluțiilor și eficiență computațională.

În plus în această lucrare vom prezența o aplicație a problemei Bandwidth coloring și anume atribuirea de examene slot-urilor dintr-un orar, având ca și constrângere necesitatea de pauze între examenele ce au studenți în comun.

Chapter 1

Descrierea problemei

1.1 Problema colorării

Problema colorării grafului (*Graph Coloring Problem*) este una dintre problemele fundamentale din teoria grafurilor și optimizarea combinatorică. Aceasta poate fi definită astfel:

Considerăm un graf neorientat $G = (V, E)$, unde V este mulțimea de vârfuri și E este mulțimea de muchii. O colorare a grafului este o funcție $c : V \to \mathbb{Z}^+$ astfel încât două vârfuri adiacente să nu primească aceeași culoare. Adica, pentru orice pereche de vârfuri adiacente $(u, v) \in E$, trebuie să satisfacem condiția:

$$c(u) \neq c(v), \quad \forall (u, v) \in E. \tag{1.1}$$

Obiectivul problemei de colorare a grafului este de a minimiza numărul de culori utilizate, notat de obicei ca k, astfel încât să obținem o colorare validă. Acest număr minim de culori necesar pentru a colora graful G este cunoscut sub numele de *numărul cromatic* al grafului, notat cu $\chi(G)$. Problema poate fi formulată:

$$
\begin{aligned}
\min \quad & k && (1.2)\\
\text{s.t.} \quad & c : V \to \{1, 2, \ldots, k\}, && (1.3)\\
& c(u) \neq c(v), \quad \forall (u, v) \in E. && (1.4)
\end{aligned}
$$

Problema colorării grafului este NP-dificilă, ceea ce implică faptul că nu există un algoritm cunoscut care să o rezolve în timp polinomial pentru toate instanțele posibile. Din acest motiv, în practică sunt utilizate adesea algoritmi euristici și metaeuristici pentru a obține soluții aproximative de bună calitate într-un timp rezonabil.

1.2 Bandwith Coloring Problem

Bandwidth Coloring Problem - BCP este o problemă de optimizare combinatorică ce poate fi formulată astfel: Considerăm un graf neorientat $G = (V, E)$, unde V este mulțimea de vârfuri și E este mulțimea de muchii. Fiecare muchie $(u, v) \in E$ are asociată o valoare $d(u, v)$ care reprezintă distanța între vârfurile u și v.

Scopul problemei BCP este de a găsi o colorare a vârfurilor, adică o funcție $c : V \to \mathbb{Z}^+$, astfel încât pentru orice pereche de vârfuri adiacente $(u, v) \in E$, diferența absolută între culorile alocate să fie cel puțin egală cu distanța dintre ele, adică:

$$|c(u) - c(v)| \geq d(u, v), \quad \forall (u, v) \in E. \tag{1.5}$$

Obiectivul este de a minimiza numărul maxim de culori utilizate, notat $\max_{v \in V} c(v)$. Problema poate fi formalizată prin următorul model:

$$
\begin{aligned}
\min \quad & \max_{v \in V} c(v) && (1.6)\\
\text{s.t.} \quad & |c(u) - c(v)| \geq d(u, v), \quad \forall (u, v) \in E, && (1.7)\\
& c(v) \in \mathbb{Z}^+, \quad \forall v \in V. && (1.8)
\end{aligned}
$$

Această problemă este NP-dificilă, ceea ce înseamnă că nu există un algoritm cunoscut care să o rezolve în timp polinomial pentru toate instanțele posibile. Datorită complexității sale, sunt utilizate frecvent metode euristice și metaeuristice pentru a obține soluții aproximative de calitate în timp rezonabil.

Chapter 2

Algoritmul de invatare

Algoritmul *Learning-Based Path Relinking* (LPR) pentru rezolvarea problemei BCP combină tehnici avansate de optimizare pentru a îmbunătăți calitatea soluțiilor și eficiența computațională. Algoritmul integrează două componente principale: optimizarea tabu și procedura de relinking.

Cautarea Tabu (*Tabu Search*) este o metodă de căutare locală care explorează un spațiu de soluții și evită ciclicitatea prin utilizarea unei liste tabu. În LPR, optimizarea tabu este folosită pentru a rafina soluțiile inițiale și pentru a îmbunătăți soluțiile intermediare generate de procedura de relinking.

Procedura de Relinking are scopul de a crea noi soluții prin generarea de căi între două soluții de înaltă calitate. Aceasta implică construirea unei serii de soluții intermediare care fac tranziția de la o soluție inițială la una de ghidare. Algoritmul LPR poate fi descris în următorii pași:

1. **Inițializarea Populației**: Se generează soluții aleatorii și se optimizează folosind metoda tabu pentru a ajunge la un optim local.

2. **Selectarea Perechilor de Soluții**: Se alege o pereche de soluții de calitate înaltă pentru a iniția procesul de relinking.

3. **Construirea Căii**: Se construiește o cale între soluția inițială și cea de ghidare prin modificări succesive, fiecare pas fiind optimizat folosind

metoda tabu.

4. **Îmbunătățirea Soluțiilor**: Soluțiile intermediare generate pe parcursul căii sunt optimizate suplimentar pentru a îmbunătăți calitatea lor.

5. **Actualizarea Populației**: Soluțiile de calitate superioară rezultate din procedura de relinking sunt adăugate în populația inițială.

6. **Criterii de Oprire**: Procesul se repetă până la îndeplinirea unui criteriu de oprire, cum ar fi un număr maxim de iterații sau identificarea unei solutii care rezolva problema.

Funcția *SumConstraintViolations* are scopul de a evalua o soluție generată, calculând câte dintre constrângerile sunt încălcate de soluția respectivă. Acest proces este esențial pentru a determina calitatea soluției și pentru a ghida procesele de optimizare și îmbunătățire.

$$\text{SumConstraintViolations} = \sum_{e(v,u)\in E} max(0, d(u,v) - |c(u) - c(v)|) \tag{2.1}$$

Algorithm 1 Learning-based path-relinking

```
procedure LPR
    MaxIterations ← 2
    Iterations ← 0
    BestSol ← {}
    WorstSol ← {}
    repeat
        Inițializează Populația
        if Iterations > 0 then
            Determină și înlocuiește cea mai proastă soluție din populatia
            curenta cu cea mai buna solutie din populatia anterioara
        end if
        Actualizează BestSol
        Creează setul de perechi de soluții
        while PairSet nu este gol do
            SelectedPair ← Pereche random din PairSet
            //Generază solutii folosind strategia de path-relinking
            FirstChild = PathRelinking(SelectedPair.first, SelectedPair.second)
            SecondChild = PathRelinking(SelectedPair.second, SelectedPair.first)
            // Aplic Tabu Search pe solutiile generate
            // Actualizeaza Populatia/Pairset cu solutiile noi daca este cazul
            Improvement_and_Updating(FirstChild, BestSol, PairSet, Population)
            Improvement_and_Updating(SecondChild, BestSol, PairSet, Population)
            if SumConstraintViolations(BestSol) == 0 then
                return BestSol
            end if
        end while
        Iterations ← Iterations + 1
    until Iterations < MaxIterations
    return {}                ▷ Returnează o soluție goală dacă nu s-a găsit o soluție optimă
end procedure
```

2.1 Initializarea populatiei

Funcția de inițializare a populației are rolul de a genera un set inițial de soluții pentru algoritmul de optimizare. Acest proces este esențial pentru a asigura diversitatea inițială a populației și pentru a crește șansele de a găsi soluții optime. Funcția descrisă mai jos detaliază pașii necesari pentru a crea această populație inițială. Se creează un set extins de soluții inițiale, denumit *LargerPopulation*. Acesta este de trei ori mai mare decât dimensiunea dorită a populației finale. Se genereaza solutii random care apoi sunt imbunatatite prin Cautarea Tabu. Acești pași asigură că populația inițială este diversificată și bine optimizată, oferind o bază solidă pentru etapele ulterioare ale algoritmului de optimizare.

Algorithm 2 Inițializare Populație

procedure InitializePopulation
  LargerPopulation lista goala
  for $Index = 0, Index < 3 * PopulationSize$ **do**
    RandSol $\leftarrow$ Genereaza o solutie random
    TabuSearch(RandSol, SumConstraintViolations)
    Adauga RandSol la LargerPopulation
  end for
  Sorteaza LargerPopulation in functie de SumConstraintViolations
  Adauga primele PopulationSize solutii in Populatia initiala
end procedure

2.2 Operatorii Path-Relinking

Path-Relinking este o metodă de explorare a spațiului de soluții utilizată frecvent în optimizarea combinatorică. Aceasta implică construirea de soluții intermediare între două soluții de referință: o soluție de pornire și o soluție țintă. Obiectivul este de a genera soluții noi, potențial mai bune, explorând traiectoria dintre aceste două soluții.

2.2.1 Greedy Path-Relinking

Operatorul Greedy de Path-Relinking construiește soluții intermediare prin alegerea la fiecare pas a mutării care conduce la cea mai bună îmbunătățire a funcției obiectiv. Aceasta implică parcurgerea iterativă a diferențelor dintre soluția de pornire și soluția țintă, aplicând mutările cele mai promițătoare.

2.2.2 Mixed Path-Relinking

Mixed Path-Relinking este o tehnică de optimizare combinatorică care construiește soluții intermediare prin alegerea alternativă a mutărilor din două soluții de referință la fiecare pas. Scopul este de a explora traiectoria dintre aceste soluții într-un mod echilibrat, combinând trăsături din ambele soluții pentru a genera soluții noi și potențial mai bune. Pentru a optimiza algoritmul, la fiecare in care se alege nodul de substitutie este evitata recalcularea costului functiei obiectiv in intregime deoarece este costisitor din punct de vedere al timpului de executie. In schimb este preluat costul precedent si se calculeaza incalcarea constrangerilor doar in nodul curent. Nodul diferit cu cel mai bun cost obtinut este propagat mai departe si este eliminat din multime. Acesta este algortimul folosit in implementare.

Algorithm 3 Mixed Path-Relinking

Input: FirstParent, SecondParent

Output: Last

Initialize DiffPos lista goala

for $v_i \in V$ **do**

  if $FirstParent[v_i] \neq SecondParent[v_i]$ **then**

    Add $Index$ to DiffPos

  end if

end for

$DiffPosLen \leftarrow |DiffPos|$

$PrevLast \leftarrow FirstParent$

$Last \leftarrow SecondParent$

$SumConstraintsPrevLast \leftarrow \text{SumConstraintViolations}(PrevLast)$

$SumConstraintsLast \leftarrow \text{SumConstraintViolations}(Last)$

$CurrentLen \leftarrow 2$

while $|DiffPos| > 0$ **do**

  Initialize CurrentChoice

  if $CurrentLen\%2 == 0$ **then**

    $CurrentChoice \leftarrow SecondParent$

  else

    $CurrentChoice \leftarrow FirstParent$

  end if

  $BestSubstitutionCost \leftarrow \infty$

  $BestSubstitutionIndex \leftarrow \infty$

  for $Index \leftarrow 0$ to $|DiffPos| - 1$ **do**

    $CurrentDiffNode \leftarrow DiffPos[Index]$

    $NewSum \leftarrow SumConstraintsPrevLast$

Algorithm 4 Mixed Path-Relinking

for $e(CurrentDiffNode, Neighbour) \in E$ **do**
$NewSum \leftarrow NewSum - \max(0, d(CurrentDiffNode, Neighbour) - |PrevLast[CurrentDiffNode] - PrevLast[Neighbour]|)$
$NewSum \leftarrow NewSum + \max(0, d(CurrentDiffNode, Neighbour) - |CurrentChoice[CurrentDiffNode] - CurrentChoice[Neighbour]|)$
end for
if $NewSum < BestSubstitutionCost$ **then**
$BestSubstitutionCost \leftarrow NewSum$
$BestSubstitutionIndex \leftarrow Index$
end if
end for
$TempSol \leftarrow PrevLast$
$TempSol[DiffPos[BestSubstitutionIndex]] \leftarrow CurrentChoice[DiffPos[BestSubstitutionIndex]]$
$PrevLast \leftarrow Last$
$Last \leftarrow TempSol$

$TempSum \leftarrow BestSubstitutionCost$
$SumConstraintsPrevLast \leftarrow SumConstraintsLast$
$SumConstraintsLast \leftarrow TempSum$

Remove element at $BestSubstitutionIndex$ from $DiffPos$
$CurrentLen \leftarrow CurrentLen + 1$
end while
return $Last$

2.3 Cautarea Tabu

Căutarea Tabu este o metodă euristică de optimizare utilizată pentru a rezolva probleme complexe, cum ar fi problemele de colorare a grafurilor, problemele de rutare și multe altele. Această metodă este capabilă să scape din minimele locale și să exploreze mai eficient spațiul soluțiilor posibile. Algoritmul utilizează o listă tabu pentru a memora mișcările recente și pentru a preveni revenirea la soluțiile anterioare.

Pașii Algoritmului de Căutare Tabu

1. **Inițializarea**: - Algoritmul începe cu o soluție inițială și inițializează lista tabu, care este utilizată pentru a evita ciclurile și pentru a încuraja explorarea de noi regiuni ale spațiului de soluții.

2. **Generarea vecinilor**: - În fiecare iterație, se generează un set de soluții vecine. Aceste soluții vecine sunt obținute prin aplicarea unor mici modificări asupra soluției curente.

3. **Evaluarea soluțiilor vecine**: - Fiecare soluție vecină este evaluată pe baza unei funcții obiectiv, de exemplu, numărul de constrângeri încălcate (este folosita functia $SumConstraintsViolations$).

4. **Actualizarea soluției curente**: - Dintre soluțiile vecine, se selectează cea mai bună soluție care nu este interzisă de lista tabu. Dacă această soluție este mai bună decât soluția curentă, devine noua soluție curentă. Daca totusi exista o solutie care este mai buna decat solutia curenta si solutia noua descoperita dar care se regaseste in lista tabu, aceasta este luata in considerare ca fiind cea ma buna solutie (Criteriul de aspiratie).

5. **Actualizarea listei tabu**: - Mișcarea care a dus la noua soluție curentă este adăugată în lista tabu pentru un anumit număr de iterații. Lista tabu este utilizată pentru a preveni întoarcerea la soluțiile anterioare și pentru a forța explorarea de noi regiuni ale spațiului de soluții.

6. **Criteriul de oprire**: - Algoritmul se oprește după un număr fix de iterații sau când o soluție care nu incalca nicio constrangere este gasita.

Algorithm 5 Tabu Search

```
procedure TabuSearch(Solution, ObjFunction)
    LowestConstraintViolation ← ObjFunction(Solution)
    MaxDepth ← α
    while CurrentDepth < MaxDepth do
        if LowestConstraintViolation == 0 then
            return BestSol
        end if
        for each Node do
            if Node nu incalca constrangeri then
                continue
            end if
            for each NewColor do
                if Solution[Node] == NewColor then
                    continue
                end if
                CurrChoice ← (Node, NewColor)
                IsTabu ← verific daca CurrChoice este in TabuTable
                Delta ← modificarea costului dupa ce Solution[Node] = Newcolor
                if !IsTabu then
                    if SolutionCost − Delta < BestCandidateValue then
                        BestCandidateValue ← SolutionCost − Delta
                        BestCandidateList ← CurrChoice
                    else if SolutionCost − Delta == BestCandidateValue then
                        push(BestCandidateList, CurrChoice)
                    end if
                else
                    if SolutionCost − Delta < BestCandidateValueTabu then
                        BestCandidateValueTabu ← SolutionCost − Delta
                        BestCandidateListTabu ← CurrChoice
```

```
                else if SolutionCost − Delta == BestCandidateValueTabu then
                    push(BestCandidateListTabu, CurrChoice)
                end if
            end if
        end for
    end for
    if nu exista solutii in BestCandidateList si BestCandidateListTabu then
        return BestSol
    end if
    if BestCandidateValueTabu are cea mai buna valoare then
        BestCandidate ← aleg random din BestCandidateListTabu
        BestCandidateValue ← BestCandidateValueTabu
    else
        BestCandidate ← aleg random din BestCandidateList
    end if
    TabuTable[BestCandidate] ← CurrentIteration + T[I_i] + r
    SolutionCost ← BestCandidateValue
    Solution[BestCandidate.first] ← BestCandidate.second
    if BestCandidateValue < LowestConstraintViolation then
        LowestConstraintViolation ← BestCandidateValue
        BestSol ← Solution
        CurrentDepth ← 0
    else
        CurrentDepth ← CurrentDepth + 1
    end if
    CurrentIteration ← CurrentIteration + 1
  end while
  return BestSol
end procedure
```

2.3.1 Cautarea Tabu in doua faze

Algoritmul de căutare tabu în două faze este creat pentru a îmbunătăți procesul de optimizare prin alternarea între faze de explorare și exploatare. Acesta permite algoritmului să evite blocajele în minime locale și să exploreze spațiul soluțiilor mai bine.

Faza de Explorare

În timpul fazei de explorare, obiectivul este de a vizita cât mai multe regiuni ale spațiului de soluții. Pentru a realiza acest lucru, vecinătatea soluției curente este generată astfel încât să includă soluții mai diverse. Funcția de cost în această fază adaugă penalizări suplimentare (pe langa functia initiala de constrangeri) pentru a încuraja tranzițiile către soluții mai îndepărtate. Aceste penalizari pot fi definite ca gradul de dificultate pe care o muchie il are pentru a indeplini o constrangere in spatiul curent de cautare.

$$\text{AugmentedSum} = \text{SumConstraintViolations} + \sum_{e(v,u)\in E, |c(u)-c(v)|<d(u,v)} w(u,v) \tag{2.2}$$

Greutățile sunt ajustate dupa fiecare iteratie a algoritmului tabu. Pentru fiecare muchie ce incalca o constrangere costul muchiei este incrementat. Daca o penalizare depaseste un threshold maxim (in cazul acesta este folosit $MaxPenaltyWeight = 30$) atunci toate weight-urile sunt scalate pentru a putea evita informatiile prea vechi (ramase de multe iterații in w):

Algorithm 6 UpdatePenaltyMatrix

procedure UPDATEPENALTYMATRIX(Solution)
  $MaxPenalty \leftarrow 0$
  for $e(v1, v2) \in E$ **do**
    if $|c(v1) - c(v2)| < d(v1, v2)$ **then**
      $w(v1, v2) \leftarrow w(v1, v2) + 1$
      $w(v2, v1) \leftarrow w(v2, v1) + 1$
    end if
    $MaxPenalty \leftarrow \max(MaxPenalty, w(v1, v2))$
  end for
  if $MaxPenalty > MaxPenaltyWeight$ **then**
    for $e(v1, v2) \in E$ **do**
      $w(v1, v2) \leftarrow \lfloor ScalingFactor * w(v1, v2) \rfloor$
      $w(v2, v1) \leftarrow \lfloor ScalingFactor * w(v2, v1) \rfloor$
    end for
  end if
end procedure

Faza de Exploatare

După identificarea unei regiuni promițătoare, algoritmul trece în faza de exploatare. În această fază, vecinătatea soluției curente este generată pentru a include soluții similare, permițând o căutare mai fină în jurul celei mai bune soluții găsite.

Această abordare combinată ajută algoritmul să evite blocajele în minime locale și să găsească soluții de calitate superioară.

Algorithm 7 TwoPhaseTabuSearch

procedure TWOPHASETABUSEARCH(Solution)
  TABUSEARCHIMPR(Solution, **AugmentedSum**)
  TABUSEARCHIMPR(Solution, **SumConstraintViolations**)
end procedure

2.3.2 Durata Tabu Dinamica

În algoritmul de căutare tabu, utilizarea unei **durate tabu dinamice** este esențială pentru a echilibra între explorarea noului spațiu de soluții și exploatarea celor mai bune soluții descoperite până în prezent. Metoda prezentată se bazează pe o abordare periodică, unde durata tabu-ului variază în mod ciclic pe parcursul algoritmului.

Avantajele Abordării Dinamice

- **Evitarea Blocajelor**: Prin varierea periodică a duratei tabu-ului, algoritmul evită blocajele în minime locale, explorând mai eficient spațiul soluțiilor.
- **Echilibru între Explorare și Exploatare**: Perioadele scurte de tabu permit explorarea rapidă a vecinătăților soluțiilor curente, în timp ce perioadele mai lungi permit diversificarea căutării și evitarea revenirii premature la soluțiile recente.
- **Adaptabilitate**: Componentele aleatorii introduse în durata tabu-ului asigură că algoritmul nu urmează un tipar rigid, permițându-i să se adapteze dinamic la caracteristicile problemei pe măsură ce căutarea progresează.

Parametrii de Bază

- $T_{\mathbf{max}}$: Valoarea maximă pentru durata tabu-ului.
- $P_{\mathbf{max}}$: Numărul total de intervale sau perioade.

Vectorul de Coeficienți

Se definește un vector de coeficienți A cu $P_{\max}$ elemente:

$$A = \{a_0, a_1, a_2, \ldots, a_{P_{\max}-1}\}$$

Calculul Duratelor Tabu și al Intervalelor

Pentru fiecare iterație considerăm în mod ciclic că se încadrează în câte un interval pentru care știm lungimea I_i. Pentru fiecare interval definim o durata Tabu diferită T_i. Acestea sunt calculate astfel:

1. **Durata Tabu** pentru fiecare interval i:

$$T_i = \frac{T_{\max} \cdot a_i}{8}$$

unde T_i este durata tabu-ului pentru intervalul i.

2. **Lungimea Intervalului** pentru fiecare interval i:

$$I_i = \frac{T_{\max} \cdot a_i}{2}$$

unde I_i este lungimea intervalului pentru i-lea interval.

Alegerea Duratei Tabu-ului în Căutare

În fiecare iterație t a algoritmului:

1. **Marcarea Soluției**: Soluția curentă considerată cea mai bună (denumită x_{best}) este marcată ca tabu pentru o perioadă determinată de:

$$\text{TabuTenure}(x_{\text{best}}) = t + T_i + r$$

unde:

- t este numărul curent al iterației.
- T_i este durata tabu-ului pentru intervalul curent i.
- r este un număr aleatoriu mic (de exemplu, $r \in \{0, 1, 2\}$) pentru a introduce variabilitate.

2. **Actualizarea Intervalului**:

- Se utilizează un contor k pentru a urmări numărul de iterații în intervalul curent i.

- Dacă k depășește lungimea intervalului I_i, se trece la următorul interval:

$$i = (i + 1) \mod P_{\max}$$

- Contorul k este resetat dupa parcurgerea tuturor intervalelor generate:

$$k = 0$$

In solutia curenta utilizam $T_{\max} = 50$, $P_{\max} = 15$ si $A = \{1, 2, 1, 4, 1, 2, 1, 8, 1, 2, 1, 4, 1, 2, 1\}$.

2.3.3 Optimizarea recalcularii costului

Se observă că dacă pentru fiecare modificare de culoare ce se realizează în cadrul unei iterații din algoritmul prezentat se calculează costul funcției obiectiv (fie că este funcția inițială sau cea augmentat) atunci algoritmul per total va avea o execuție înceată. Pentru fiecare apel al funcției vom avea o complexitate $\mathcal{O}(|E|)$. Deoarece căutarea soluțiilor vecine se face schimbând câte un nod, ne putem folosi de costul curentă al soluției realizând modificări doar la nivelul nodului pe care îl schimbăm (exact la fel că la operatorul de path-relinking). Mai mult pentru a reduce complexitatea la $\mathcal{O}(1)$ se observă că pentru fiecare nod putem precalculă într-o matrice costul. Același lucru poate fi făcut și pentru calcularea greutăților.

$ccs[Node][NewColor]$ = costul functiei **SumConstraintViolations** pentru solutia curenta daca nodul **Node** este colorat cu **NewColor**

$ccws[Node][NewColor]$ = costul greutatilor muchiilor ce incalca constrangeri daca nodul **Node** este colorat cu **NewColor**

Prin urmare schimbarea costului (Delta 18) se poate calcula astfel:

$Delta \leftarrow ccs[Node][Solution[Node]] - ccs[Node][NewColor] + ccws[Node][Solution[Node] - ccws[Node][NewColor]$ Pentru prima faza de cautare.

Pentru a 2-a faza de cautare se elimina adaugarea costurilor greutatilor de pe muchii.

Algorithm 8 InitializePrecalcMatrixes

```
procedure InitializePrecalcMatrixes
    for each Node do
        for each NewColor do
            ccs[Node][NewColor] ← 0
            for each Neighbour s.t. e(Node, Neighbour) exists do
                ccs[Node][NewColor] += max(0, d(Node, Neighbour) − |c(Neighbour) − NewColor|)
            end for
        end for
    end for
    for each Node do
        for each NewColor do
            ccws[Node][NewColor] ← 0
            for each Neighbour s.t. e(Node, Neighbour) exists do
                if |Solution[Neighbour] − NewColor| < Edges[Node][Neighbour] then
                    ccws[Node][NewColor] += w[Node][Neighbour]
                end if
            end for
        end for
    end for
end procedure
```

După descoperirea celei mai bune soluții pentru a continuă folosirea acestor matrici ele trebuie actualizate conform modificării făcute. Se observă că modificarea afectează vecinii nodului curent iar pentru a nu parcurge toate culorile posibile se observă că trebuie modificate doar culorile care ar încalcă constrângerile:

$$|c(Neighbour) - c(Node)| < d(Node, Neighbour) \implies$$

$$c(Node) - d(Node, Neighbour) < c(Neighbour) < c(Node) + d(Node, Neighbour)$$

Algorithm 9 UpdatePrecalcMatrixes

procedure UPDATEPRECALCMATRIXES
    OldColor $\leftarrow$ Solution[Node]
    for each Neighbour s.t. $e(Node, Neighbour)$ exists **do**
        Start $\leftarrow \max(1, OldColor - d(Node, Neighbour) + 1)$
        End $\leftarrow \min(NoColors, OldColor + d(Node, Neighbour) - 1)$
        for It = Start to End **do**
            $ccs[Neighbour][It] \leftarrow ccs[Neighbour][NewColor] - (d(Neighbour, Node) - |OldColor - It|)$
        end for
        Start $\leftarrow \max(1, NewColor - d(Node, Neighbour) + 1)$
        End $\leftarrow \min(NoColors, NewColor + d(Node, Neighbour) - 1)$
        for It = Start to End **do**
            $ccs[Neighbour][It] \leftarrow ccs[Neighbour][NewColor] + (d(Node, Neighbour) - |NewColor - It|)$
        end for
    end for
    for each Neighbour s.t. $e(Node, Neighbour)$ exists **do**
        Start $\leftarrow \max(1, OldColor - d(Node, Neighbour) + 1)$
        End $\leftarrow \min(NoColors, OldColor + d(Node, Neighbour) - 1)$
        for It = Start to End **do**
            $ccws[Neighbour][It] \leftarrow ccws[Neighbour][It] - w[Neighbour][Node]$
        end for
        Start $\leftarrow \max(1, NewColor - d(Node, Neighbour) + 1)$
        End $\leftarrow \min(NoColors, NewColor + d(Node, Neighbour) - 1)$
        for It = Start to End **do**
            $ccws[Neighbour][It] \leftarrow ccws[Neighbour][It] + w[Neighbour][Node]$
        end for
    end for
end procedure

Chapter 3

Experimente

Pentru a testa performanta CPU am folosit seturile de date de referință DIMACS. Testele au fost realizate pe un calculator echipat cu un procesor 11th Gen Intel(R) Core(TM) i7-11800H, cu o frecvență de 2.30GHz. Timpurile de execuție pentru grafurile r300.5.b, r400.5.b și r500.5.b au fost înregistrate așa cum este prezentat în Tabelul 3.1.

Graf	**Timp de Execuție (secunde)**
r300.5.b	0.0058686
r400.5.b	0.0084827
r500.5.b	0.0134917

Table 3.1: Timpurile de execuție pentru grafurile de referință DIMACS pe un procesor 11th Gen Intel(R) Core(TM) i7-11800H la 2.30GHz.

Grafurile utilizate în experimente (pentru testarea LPR) sunt preluate din lucrarea lui Trick, M.A., 2002, intitulată ”Computational Symposium: Graph Coloring and Its Generalization”. Aceste grafuri sunt folosite pentru a testa performanța algoritmilor de colorare a grafurilor. Tabelul de mai jos sumarizează soluțiile și rezultatele experimentale pentru algoritmul LPR (Learning-Based Path Relinking).

Algoritmul LPR a avut un succes complet (20/20) pentru aproape toate instanțele de grafuri, cu excepția câtorva cazuri unde ratele de succes au fost mai mici. Aceasta indică faptul că algoritmul este foarte eficient în a găsi soluții viabile pentru problemele de colorare a grafurilor.

Timpul mediu de succes variază semnificativ, de la 0.02 secunde pentru grafurile mai mici (GEOM20) până la 254.89 secunde pentru grafurile mai mari (GEOM110a). Aceasta arată o creștere semnificativă a timpului de succes odată cu creșterea complexității grafurilor.

Timpul mediu de execuție crește semnificativ odată cu mărimea și complexitatea grafurilor. De la 0.36 secunde pentru GEOM20 la 7616.63 secunde pentru GEOM110a, ceea ce sugerează o creștere exponențială a timpului de calcul necesar pentru grafuri mai complexe.

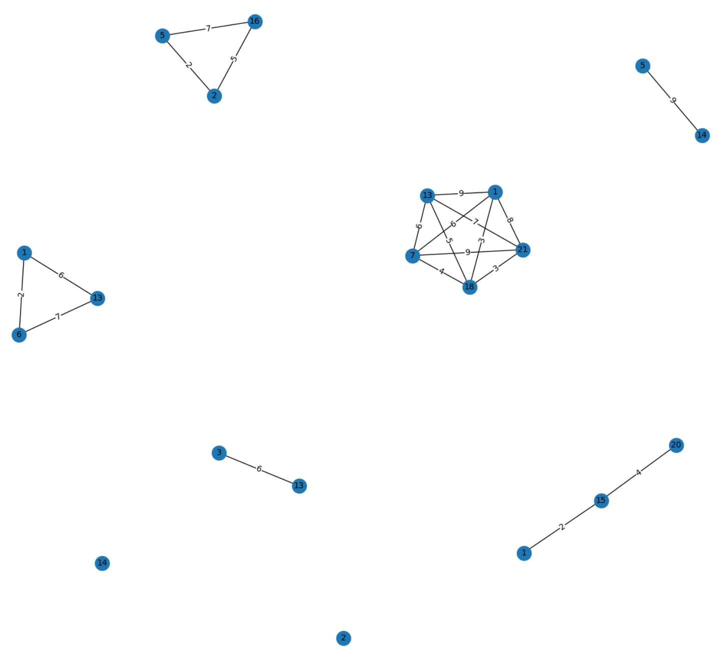


Figure 3.1: Reprezentare vizuala a unei solutii pentru graful GEOM20

Filename	**SR (Success Rate)**	**Average Success Time (s)**	**Average Execution Time (s)**
GEOM20	20/20	0.02	0.36
GEOM40	20/20	0.03	0.65
GEOM20b	20/20	0.04	0.75
GEOM30	20/20	0.03	0.52
GEOM60	20/20	0.31	6.24
GEOM20a	20/20	0.21	4.29
GEOM30a	20/20	0.87	17.47
GEOM30b	20/20	0.03	0.61
GEOM40b	20/20	1.87	37.36
GEOM40a	20/20	1.65	32.98
GEOM50	20/20	0.05	1.06
GEOM50a	20/20	4.09	81.75
GEOM80	20/20	4.16	83.18
GEOM60a	20/20	6.79	135.82
GEOM100	20/20	7.77	155.48
GEOM110	20/20	13.56	271.26
GEOM50b	20/20	9.44	188.79
GEOM80a	20/20	18.7	373.93
GEOM70a	20/20	33.09	661.74
GEOM80b	20/20	47.2	944.04
GEOM90	20/20	1.46	29.29
GEOM90a	20/20	55.37	1107.43
GEOM60b	18/20	52.78	1261.52
GEOM70	20/20	0.56	11.23
GEOM70b	11/20	83.69	2342.11
GEOM90b	2/20	127.17	4209.58
GEOM100a	5/20	178.05	5205.34
GEOM100b	0/20	-	-
GEOM120a	1/20	120.35	6685.57
GEOM110b	0/20	-	-
GEOM120	20/20	6.84	136.78
GEOM110a	2/20	254.89	7616.63
GEOM120b	1/20	92.15	5666.5

3.1 Aplicațiile Algoritmului BCP

Algoritmul Bandwidth Coloring Problem (BCP) este folosit în diverse domenii unde este necesară minimizarea diferenței maxime dintre valori asociate entităților adiacente. Aplicațiile includ:

- **Inginerie de rețele**: Alocarea frecvențelor în rețelele de telecomunicații pentru a minimiza interferențele.
- **Planificare urbană**: Distribuirea infrastructurilor critice, cum ar fi spitalele sau stațiile de pompieri, pentru a reduce timpul de răspuns.
- **Gestiunea traficului**: Programarea semafoarelor pentru a optimiza fluxul traficului și a reduce congestia.
- **Sisteme de transport**: Crearea orarelor pentru trenuri sau autobuze astfel încât să se minimizeze timpul de așteptare între conexiuni.

3.2 Problema Asignării Examenelor într-un Orar folosind BCP

Problema asignării examenelor într-un orar (ETP - Examination Timetabling Problem) poate fi transformată într-o problemă de Bandwidth Coloring (BCP) pentru a găsi o soluție validă care să respecte constrângerile impuse de diferențele minime necesare între zilele de programare ale examenelor cu studenți comuni.

Definirea Problemei

Obiectiv: Programarea examenelor astfel încât zilele examenelor cu studenți comuni să respecte diferenta minima setata intre ele.

1. **Reprezentare Grafului**:
   - **Vârfuri (Examene)**: Fiecare examen este reprezentat printr-un vârf. In plus fiecare examen are o anumita dificultate.

- **Muchii (Conflicte de Studenți)**: O muchie între două vârfuri indică faptul că există cel puțin un student care trebuie să susțină ambele examene. Consideram ca exista 90 de sloturi orare, mai exact 3 saptamani (de luni pana vineri) de examene cate 6 sloturi pe zi (de la 8:00 pana la 18:00 cate doua ore) de luni pana vineri. Aceste slot-uri pot fi modificate dar pentru exemplificare vom folosi aceasta reprezentare. In plus aplicam o conditie suplimentara si anume ca in functie de dificultatea examanelor cu studenti comuni acestea vor fi separate de mai multe zile pauza (costuri pe muchii). Vom considera costul ca fiind minimul dintre dificultatile examenelor (1 - easy, 2 - mediu, 3 -hard). De ex. pentru 2 examene cu studenti comuni dintre care unul este greu si celalalt medium vom considera 2 zile pauza intre ele.

2. **Colorare cu Bandwidth**:

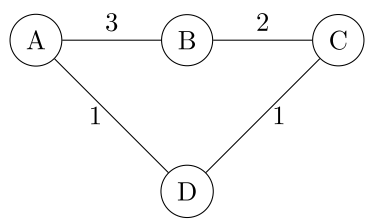
   - **Culori (zile)**: Atribuirea de culori (zile) vârfurilor (examenelor).
   - **Respectarea Constrângerilor de Bandwidth**: Diferența dintre zilele atribuite examenelor adiacente trebuie să fie cel puțin valoarea costului muchiei care le leagă.

Exemplu Detaliat

Considerăm examenele A, B, C și D cu următoarele conflicte de studenți:

- A (greu) și B (greu) au studenți comuni.
- B (greu) și C (mediu) au studenți comuni.
- C (mediu) și D (easy) au studenți comuni.
- A (greu) și D (easy) au studenți comuni.

Reprezentarea Grafică cu Costuri:



Atribuirea Intervalelor Orare folosind BCP

Scopul este de a găsi o soluție validă în care diferențele dintre zilele atribuite examenelor adiacente să respecte constrângerile de bandwidth impuse de costurile muchiilor.

Soluție Validă:

Atribuim intervale orare astfel:

- A: Ziua 1
- B: Ziua 4
- C: Ziua 2
- D: Ziua 3

Verificarea Condițiilor:

Verificăm dacă soluția respectă condițiile de bandwidth:

$$(\text{A și B}) : |1 - 4| = 3 \geq 3$$
$$(\text{B și C}) : |4 - 2| = 2 \geq 2$$
$$(\text{C și D}) : |2 - 3| = 1 \geq 1$$
$$(\text{A și D}) : |1 - 3| = 2 \geq 1$$

Toate condițiile sunt respectate, deci aceasta este o soluție validă pentru problema BCP.

Examen	Ziua
A	saptamana 1, luni
B	saptamana 1, joi
C	saptamana 1, marti
D	saptamana 1, miercuri

Table 3.2: Orarul examenelor conform soluției BCP, Sloturile orare pot fi alese random

Generarea datelor

Pentru a putea testa algoritmul am generat fișiere de tip json cu ajutorul unui script de python (**generate_exams.py**) în care am reprezentat examenele obligatorii, pachetele de opționale (pentru fiecare pachet fiecare elev trebuie să aibă cel puțin un examen) și cum sunt distribuite examenele studenților. Pentru fiecare set de date s-au considerat un număr random de studenți (300 - 400), examene (6 - 10) și numărul de pachete de opționale (1 - 5). Diferența dintre numărul de examene și numărul de pachete de opționale reprezintă examene obligatorii. Fiecărui student i-au fost atribuite examenele obligatorii iar pentru fiecare pachet de opționale a fost ales un examen random cu condiția că numărul de studenți atribuiți examenului să nu depaseaca o treime din numărul de studenți. Pentru testare au fost generate 10 fișiere. O exemplificare a unui fișier 3.1:

```json
{
    "mandatory_exams": {
        "0": [
            "e0",
            "easy"
        ],
        "1": [
            "e1",
            "hard"
        ],
        ...
    },
    "optional_packs": {
        "0": [
            [
                "e2",
                "e3",
                "e4",
                "e5"
            ],
            "medium"
        ],
        "1": ...
    },

    "students": {
        "0": [
            "e0",
            "e1",
            "e2",
            ...
        ],
        "1": ...
    }
}
```

Listing 3.1: Exemplu de fisier de intrare

Vizualizarea solutiilor

După generarea unei soluții folosind algoritmul LPR pe datele de intrare generate anterior, pentru vizualizare s-au generat imaginea grafului pe care se găsește soluția (imaginea a fost generată cu ajutorul unui script de python **generate_graph.py**, cu ajutorul librăriilor **networkx** și **matplotlib**) și imaginea orarului cu examenele distribuite (imaginea a fost generată cu ajutorul unui script de python **generate_timetable.py**, cu ajutorul librăriilor **pandas** și **matplotlib**). Datele de intrare alese că exemplu construiesc un graf dens așa cum se observă în figura 3.2. Există 3 examene obligatorii (2 easy, 1 hard) și 5 pachete de opționale (2 easy, 2 medium, 1 hard).

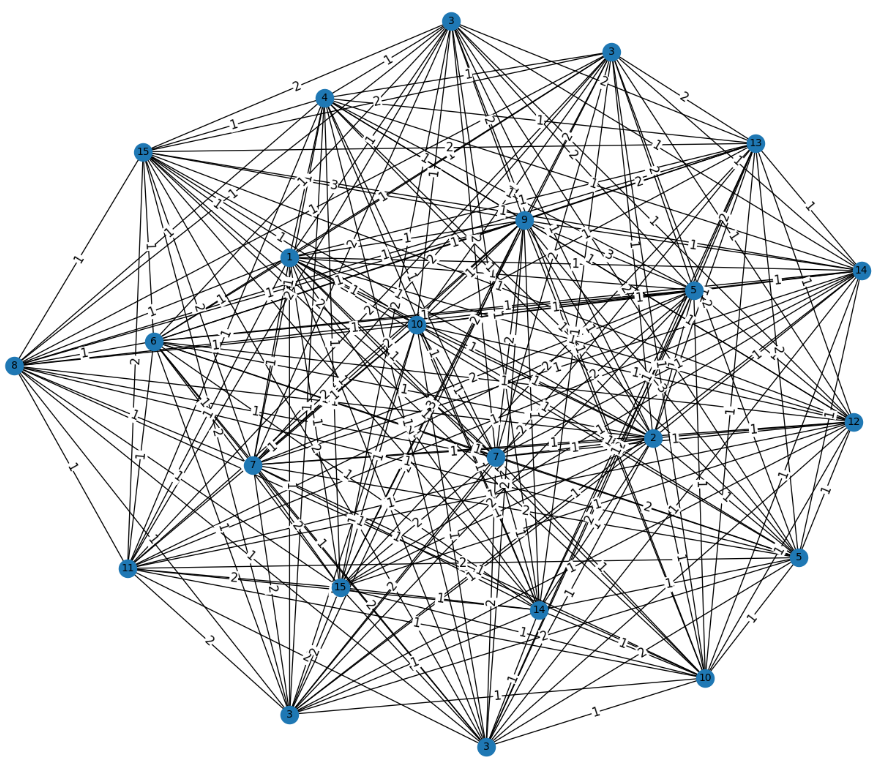


Figure 3.2: Reprezentare vizuala a unei solutii

	Week 1					Week 2					Week 3				
Time	Mon	Tue	Wed	Thu	Fri	Mon	Tue	Wed	Thu	Fri	Mon	Tue	Wed	Thu	Fri
08:00			e18			e3									e20
10:00	e0		e16 e17		e22				e2						
12:00		e1					e13					e10			
14:00													e12		
16:00			e15	e5				e9		e4	e14			e7 e8	
18:00					e21		e11			e6					e19

Figure 3.3: Reprezentare orarului in urma gasirii unei soltii valide

Concluzii

Algoritmul LPR demonstrează o performanță ridicată, obținând o rată de succes de 100% pentru aproape toate instanțele de grafuri testate.

Deși algoritmul are o rată de succes constantă în majoritatea cazurilor, timpul de execuție crește semnificativ pentru grafuri mai mari și mai complexe. Aceasta sugerează că, în timp ce algoritmul este eficient, optimizările suplimentare ar putea fi necesare pentru a îmbunătăți timpul de execuție pentru instanțe foarte mari.

Timpul mediu de succes crește semnificativ odată cu complexitatea grafurilor, ceea ce indică necesitatea unor strategii eficiente de gestionare a timpului de calcul.

Datorită creșterii exponențiale a timpului de execuție pentru grafuri mai complexe, este important să se exploreze tehnici de optimizare suplimentare pentru a gestiona mai eficient resursele de calcul și a reduce timpul de execuție.

Aceste concluzii sugerează că algoritmul LPR este foarte eficient pentru Bandwidth coloring problem, dar necesită optimizări suplimentare pentru a gestiona instanțele de grafuri foarte mari și complexe într-un timp mai bun.

Bibliography

[1] X. Lai et al. *A learning-based path relinking algorithm for the bandwidth coloring problem*, Eng. Appl. Artif. Intell. (2016).

[2] Dias, B., & Freitas, R.D. (2016). *Constraint and integer programming models for bandwidth coloring and multicoloring in graphs.*

[3] Matić, Dragan & Kratica, Jozef & Filipović, Vladimir. (2017). *Variable Neighborhood Search for solving Bandwidth Coloring Problem. Computer Science and Information Systems.*

[4] Carter, M., Laporte, G. & Lee, S. *Examination Timetabling: Algorithmic Strategies and Applications.* J Oper Res Soc 47, 373–383 (1996)