Proiect Analiza Algoritmilor
Graph Coloring

Dobre Andrei-Teodor - andrei_teodor.dobre@stud.acs.upb.ro
Brotea Florin-Alexandru - florin.brotea@stud.acs.upb.ro
Istrate Alexandru-Daniel - alexandru.istrate04@stud.acs.upb.ro

Universitatea Nationala de Stiinta si Tehnologie Bucuresti, Facultatea de Automatica si Calculatoare, Romania - decanat@acs.pub.ro
https://acs.pub.ro/

1 Introducere

Colorarea grafurilor (*graph coloring*) reprezintă una dintre cele mai vechi și studiate probleme din teoria grafurilor, fiind strâns legată de numeroase aplicații practice și rezultate teoretice profunde [13] [2]. În esență, problema colorării unui graf se referă la atribuirea unei culori fiecărui nod astfel încât două noduri adiacente (legate printr-o muchie) să nu aibă aceeași culoare [3].

De-a lungul timpului, colorarea grafurilor a fost investigată atât pentru o mai bună înțelegere a structurii grafurilor, cât și pentru aplicații concrete, cum ar fi alocarea de resurse în sisteme de calcul [4], planificarea orarelor de examene și modelarea problemelor de tip scheduling [2]. Datorită faptului că determinarea numărului minim de culori pentru colorarea unui graf (**numărul cromatic**) este o problemă NP-hard, cercetările în acest domeniu au vizat atât algoritmi exacți, care funcționează în timp exponențial, cât și algoritmi euristici și metaeuristici, cu scopul de a obține soluții cât mai bune într-un timp rezonabil [13] [2].

Pentru a obține o imagine cât mai cuprinzătoare asupra comportamentului algoritmilor, vom realiza experimente pe un set variat de grafuri: grafuri generate aleator (dense și sparse), grafuri bipartite, grafuri planare, grafuri complete, precum și unele instanțe particulare.

În cadrul acestei lucrări, se va urmări structura recomandată de cerință: vom prezenta problema, demonstrarea faptului că este NP-Hard, descrierea algoritmilor, analiza complexității și, în final, vom compara rezultatele obținute pe setul de teste alese. Concluziile vor scoate în evidență punctele forte și limitările fiecărui algoritm, având ca reper timpii de execuție și calitatea soluției obținute.

1.1 Descrierea problemei rezolvate

Problema colorării unui graf poate fi definită formal astfel:

- Avem un graf neorientat $G = (V, E)$, unde V este mulțimea nodurilor, iar E este mulțimea muchiilor.

- O **colorare** a lui G reprezintă atribuirea unei culori fiecărui nod $v \in V$, în așa fel încât, pentru orice muchie $(u, w) \in E$, nodurile u și w să aibă culori diferite [3].
- **Numărul cromatic** $\chi(G)$ este numărul minim de culori necesar pentru a obține o colorare validă. Determinarea sa exactă este NP-hard [2], ceea ce explică de ce metoda exhaustivă, de regulă, nu este fezabilă pentru grafuri de mari dimensiuni.

În practică, problema colorării poate fi întâlnită sub două forme:

1. **Varianta de decizie**: „Poate fi colorat graful G folosind cel mult k culori?"
2. **Varianta de optimizare**: „Care este numărul minim de culori necesar pentru a colora valid graful G?"

Dificultatea problemei în varianta de optimizare a determinat apariția unei game largi de algoritmi euristici și metaeuristici. Aceștia oferă soluții rapide, deși nu garantează întotdeauna obținerea numărului minim de culori [13]. În continuare, vom evidenția câteva aplicații concrete care subliniază importanța practică a acestei probleme.

1.2 Exemple de aplicații practice

1. Analiza Big Data și partajarea memoriei în calcul distribuit:

În mediile de procesare Big Data (ex. Hadoop, Spark), se caută metode de optimizare a utilizării memoriei. Apare un conflict atunci când seturi de date folosite de job-uri diferite trebuie să fie rezidente în memorie simultan, iar colorarea grafului care reprezintă relațiile de conflict poate fi o abordare pentru a împărți corect memoria partajată. [6]

2. Securitatea în rețelele de vehicule autonome (VANETs) :

Vehiculele inteligente își coordonează comunicarea pentru a evita coliziuni (atât fizice, cât și de semnal). Colorarea grafului de vehicule care comunică între ele poate fi folosită pentru a reduce interferențele și pentru a aloca sloturi de transmisie. [7]

3. Optimizarea alocării frecvențelor în rețele 5G:

În rețelele 5G și IoT (Internet of Things), alocarea spectrului de frecvențe și minimizarea interferențelor devine din ce în ce mai complexă. Colorarea grafurilor poate fi folosită pentru a atribui canalele radio astfel încât nodurile (dispozitivele din rețea) care se pot „deranja" reciproc să nu folosească aceeași frecvență. [8]

4. Planificarea sarcinilor într-un sistem multi-procesor:

Sarcinile care se suprapun în timp și împart aceleași resurse nu pot fi executate simultan pe același procesor; acest tip de conflict se modelează sub forma unui graf, iar colorarea cu kkk culori indică separarea sarcinilor incompatibile . [13]

2 Demonstrație NP-Hard

Problema abordată este să demonstrăm că această problemă este NP-Complete (NPC), adică și NP-Hard și în NP. Prin extensie, varianta de optimizare (determinarea numărului minim de culori conform cerinței problemei) este NP-Hard.

Problema este în NP Pentru a arăta întâi că problema este în NP, ne limităm la problema de decizie și demonstrăm cum se poate verifica rapid, în timp polinomial, o soluție: Certificarea este reprezentată de o atribuire de culori la noduri: un vector de dimensiune $|V|$, în care, pentru fiecare nod v atribuit, avem o culoare (un întreg între 1 și k). Verificarea constă în parcurgerea fiecărei muchii $(u, v) \in E$ și verificarea că nodurile u și v au culori diferite. În cel mai rău caz, avem $|E|$ muchii, deci verificarea durează $O(|V| + |E|)$, care este polinomial în dimensiunea intrării. Așadar, pentru început, am arătat că problema se află în NP.

Problema este NP-Hard Pentru a demonstra că problema de k-colorare este NP-Hard, facem o reducere în timp polinomial de la o problemă bine-cunoscută ca fiind NP-Completă, deci și NP-Hard. Noi am ales să lucrăm cu 3-SAT (3-Satisfiability Problem):

Instanță: O formulă booleană φ în formă normală conjunctivă (CNF), unde fiecare clauză are exact 3 literali; avem variabile $x_1, x_2, \ldots, x_n$.

Întrebare: Există o atribuire (True/False) a variabilelor care face ca φ să fie adevărată?

Schema generală de proiectare constă în construirea grafului G, căruia îi atribuim 3 culori simbolice, logice: o culoare pentru True, o culoare pentru False și o culoare ce vine pe post de element auxiliar. Vrem ca φ să fie satisfiabilă $\iff$ graful G poate fi colorat cu aceste trei culori, respectând cerința.

Pentru fiecare variabilă x_i, introducem două noduri:

- un nod pentru x_i = True, notat T_i,
- un nod pentru x_i = False, notat F_i.

Pentru a ne asigura că nu putem colora ambele noduri T_i și F_i cu aceeași culoare, le punem în conflict: adică adăugăm o muchie (T_i, F_i). Astfel, cele două noduri trebuie să primească două culori diferite.

În plus, se poate adăuga un nod intermediar, care face imposibilă colorarea ambelor cu a treia culoare simultan, forțând efectiv o alegere (True sau False) pentru fiecare variabilă.

Interpretare:

- Dacă, în colorarea finală, nodul T_i a primit culoarea "True", atunci înseamnă că variabila x_i este setată la True.
- Dacă nodul F_i e colorat cu "True", atunci interpretăm ca fiind setare la False (și invers pentru culoarea "False").

Gadget pentru clauză

- Fie o clauză de forma $(l_1 \vee l_2 \vee l_3)$, unde fiecare l_j este fie x_i, fie $\neg x_i$, oricare ar fi x_i.

- Introducem un nod de clauză, notat C.
- Conectăm nodul C la nodurile corespunzătoare. În funcție de implementare, pot fi:
  - Dacă $l_1 = x_p$, atunci legăm C la T_p.
  - Dacă $l_1 = \neg x_p$, atunci legăm C la F_p.

Ideea este ca alăturarea să forțeze următoarea condiție: cel puțin unul dintre cei trei literali din clauză trebuie să fie colorat cu "True". Dacă toți cei trei literali ar fi colorați cu "False", nodul C n-ar mai putea să se coloreze corect (ar intra în contradicție cu toate cele trei noduri vecine, care i-ar interzice să folosească acele culori).

O metodă relativ simplă este să ne asigurăm că, dacă un literal e considerat True, atunci culoarea lui este "True" (una dintre cele 3 culori), iar nodul de clauză se poate colora cu altă culoare. Dacă însă un literal e considerat False, îl punem să folosească aceeași culoare cu clauza și astfel să rezulte o contradicție dacă toți trei literalii sunt falși.

Conectarea la întreaga formulă Se parcurg toate clauzele $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ și se creează o componentă de tip clauză pentru fiecare C_j. Se leagă în mod corespunzător la variabilele care apar. Rezultă un graf G cu subgrafe pentru variabile și subgrafe pentru clauze, plus muchiile care leagă literalii respectivi.

După ce am construit graful G, trebuie să demonstrăm:

1. Dacă φ este satisfiabilă, atunci graful G poate fi colorat cu 3 culori.
   - Luăm interpretarea booleeană care face φ adevărată.
   - Pentru fiecare variabilă x_i, colorăm T_i și F_i, astfel încât nodul care corespunde valorii alese (True sau False) să primească culoarea corespunzătoare.
   - Pentru fiecare clauză, cum clauza are cel puțin un literal adevărat (True), nodul de clauză își poate alege o culoare care nu a fost aleasă înainte.
   - Se arată pas cu pas că această colorare respectă toate regulile (nu avem două noduri adiacente cu aceeași culoare) și că totul este posibil în numai 3 culori.
2. Dacă graful G se poate colora cu 3 culori, atunci φ este satisfiabilă.
   - Ne uităm la modul în care a fost colorat gadgetul fiecărei variabile x_i. Dacă nodul T_i are culoarea "True", interpretăm x_i ca fiind True.
   - Verificăm că pentru fiecare clauză, există cel puțin un literal care nu intră în conflict cu nodul de clauză, ceea ce în interpretarea booleeană înseamnă că este satisfăcut.
   - Prin urmare, se satisface toată formula φ.

Complexitatea construcției

- Pentru fiecare variabilă x_i, introducem un număr constant de noduri (2 sau 3) și un număr constant de muchii.
- Pentru fiecare clauză, introducem un nod de clauză și legături către literalii (iarăși, un număr constant de noduri și muchii).

Așadar, dacă φ are n variabile și m clauze, mărimea grafului G construit este $O(n+m)$. Timpul de construire a acestui graf este, evident, tot $O(n+m)$.

Cum 3-SAT este NP-Completă, iar reducerea noastră arată că 3-colorarea este cel puțin la fel de greu de rezolvat, deducem că și 3-colorarea este NP-Completă. Așadar, problema de a determina numărul minim de culori pentru un graf generic este NP-Hard.

3 Prezentarea Algoritmilor

Plecăm de la premisa: Fie un graf G(V, E), unde V e mulțimea vârfurilor din graf și E este mulțimea muchiilor. De asemenea, notăm cu n numărul de vârfuri ale grafului, mai exact n = |V|. Pentru acest studiu, am ales să prezentăm câțiva algoritmi care ne-au atras atenția prin particularitățile fiecăruia în rezolvarea problematicii, precum și algoritmul brute-force, despre care știm că este întotdeauna corect.

3.1 Algoritmul Brute Force (Exhaustive Search)

Problema colorării nodurilor unui graf poate fi abordată printr-o strategie de tip *exhaustive search* [9]. Acest algoritm încearcă, în mod sistematic, toate mapările posibile

$$f : V \to \{1, 2, \ldots, q\},$$

unde q este un număr de culori predefinit, verificând pentru fiecare muchie $(v, w) \in E$ dacă $f(v) \neq f(w)$. În cazul în care o astfel de mapare satisface această condiţie pentru *toate* muchiile, algoritmul poate concluziona că a găsit o colorare validă.

1. Descrierea algoritmului
Algoritmul poate fi sumarizat astfel:

- **(Main loop)** Pentru fiecare funcţie (mapare) $f : V \to \{1, 2, \ldots, q\}$, faceţi pasul urmator :

- (**Check** f) Pentru fiecare muchie $(v, w) \in E$, verificaţi dacă $f(v) \neq f(w)$. Dacă această condiţie este satisfăcută pentru *toate* muchiile, algoritmul se poate opri, returnând colorarea f ca soluţie validă.

Ilustrarea schematică din Figura 1 (adaptată din [9]) arată un exemplu de graf şi modul în care fiecare nod stochează vecinii săi. Această formă de reprezentare ne permite să parcurgem vecinii unui nod v în timp proporţional cu `deg`(v).

De remarcat că această abordare garantează găsirea unei soluţii (dacă ea există) deoarece explorează toate posibilităţile, însă nu este practică decât pentru grafuri cu un număr relativ mic de noduri sau culori.

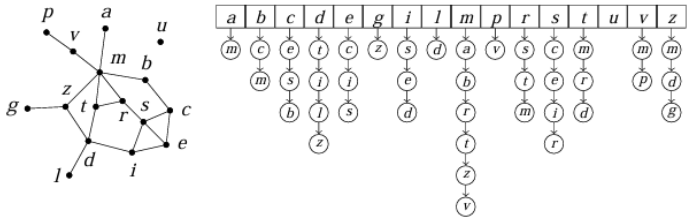


Fig. 1. Reprezentarea unui graf ca un tablou de liste de adiacenţă (adaptare din [9]).

2. Analiza complexitatii

Dacă graful G are n noduri şi m muchii, atunci numărul total de mapări posibile $f : V \to \{1, \ldots, q\}$ este q^n. Pentru fiecare mapare, trebuie să verificăm *toate* muchiile, ceea ce ia $O(n+m)$ timp într-o implementare cu liste de adiacenţă [9, 13]. În consecinţă, *timpul de execuţie* asimptotic poate fi estimat la

$$O\big(q^n \cdot (n+m)\big).$$

Această valoare reflectă caracterul *exponenţial* al algoritmului, ceea ce îl face dificil de aplicat pentru instanţe de mari dimensiuni. Totuşi, pentru grafuri mici sau cazuri speciale (de exemplu, dacă q e foarte mic), poate fi folosit ca metodă de referinţă pentru comparaţii sau pentru a obţine soluţii exacte.

3. Avantaje si dezavantaje

Avantaje.

- Garantarea unei soluţii *optime* (dacă iterăm crescător q până la obţinerea unei colorări valide).
- Foarte simplu de înţeles şi implementat, având puţine structuri de date implicate.

Dezavantaje.

- Timp de execuţie exponenţial, nepractic la scară mare.
- Nu oferă niciun fel de optimizări intermediare (soluţia apare doar la finalul parcurgerii spaţiului de căutare, de regulă).

Prin urmare, Brute Force reprezintă un **benchmark** pentru a evalua calitatea soluţiilor altor algoritmi (euristici sau metaeuristici), fiindcă oferă o *referinţă exactă* asupra validităţii şi a limitelor problemei de colorare.

3.2 Algoritmul Greedy (Sequential Coloring)

Problema colorării nodurilor unui graf poate fi abordată printr-o strategie de tip *greedy* [9]. Ideea de bază este că, parcurgând nodurile într-o anumită ordine,

atribuim fiecărui nod prima culoare disponibilă care nu este deja folosită de vecinii săi. Deşi nu garantează *întotdeauna* numărul minim de culori, în practică dă rezultate bune şi se execută rapid, fiind util în special când n (numărul de noduri) este mare.

1. Descrierea algoritmului

Algoritmul poate fi sumarizat astfel:

1. **Ordine:** Se alege o ordonare a nodurilor, de exemplu $v_1, v_2, \ldots, v_n$ (această ordine poate fi dată fie întâmplător, fie determinată de o anumită euristică precum cel mai mare grad, BFS/DFS etc.).
2. **Main loop:** Pentru fiecare nod v_i, se determină mulţimea culorilor deja folosită în vecinătatea lui v_i, notată C_i.
3. **Check color:** Se alege cea mai mică culoare c care nu aparţine lui C_i şi se atribuie $f(v_i) = c$.

2. Analiza complexităţii

Pentru a estima resursele de timp necesare:

- *Reprezentarea grafului:*
  - **Listă de adiacenţă:** Cel mai frecvent, graful este stocat într-o structură de tip `adjacency list`, unde pentru fiecare nod se întreţine o listă a vecinilor săi.
    - *Complexitate de acces la vecini:* Dacă notăm $\deg(v_i)$ gradul nodului v_i, verificăm culorile vecinilor săi în timp $O(\deg(v_i))$.
    - *Sumă gradelor:* Ţinând cont că suma gradelor pentru toate nodurile este $2m$, parcurgerea tuturor vecinilor se face în total în $O(m)$ timp.
  - **Matrice de adiacenţă:** Pentru un graf cu n noduri, stocarea se face într-o matrice $n \times n$.
    - *Complexitate de acces la vecini:* Pentru a afla vecinii lui v_i, algoritmul poate parcurge întreaga linie i, de lungime n, deci $O(n)$ per nod.
    - *Total:* Se obţine $O(n^2)$ pentru a examina toate nodurile, dacă revedem vecinii printr-o scanare completă a matricei.
- *Determinarea primei culori disponibile:*
  - *Gestionarea culorilor:* Dacă se foloseşte un tablou de dimensiune $\Delta + 1$ (unde Δ este gradul maxim) pentru a marca *culorile ocupate* de vecini, fiecare verificare în caz extrem costă $O(\Delta)$.
  - Cum, în general, $\Delta \leqslant n - 1$, în cel mai rău caz putem avea un cost de $O(n)$ per nod, deci $O(n \cdot n) = O(n^2)$ într-o implementare mai puţin optimizată.
  - *Optimizare:* Ţinând cont că în majoritatea implementărilor, căutăm doar *prima* culoare liberă, şi ne interesează doar culorile vecinilor, putem rămâne la $O(\deg(v_i))$ în cazul listelor de adiacenţă.

Combinând factorii de mai sus, *complexitatea finală* poate fi:

$$O(n + m) \quad \text{(pentru liste de adiacenţă, implementare eficientă)},$$

respectiv

$O(n^2)$ (pentru matrice de adiacenţă şi/sau gestionarea culorilor mai puţin optimizată).

În plus, **Algoritmul Greedy** consumă memorie minimă, stocând doar etichetele de culori şi o structură de date pentru vecini, în mod similar altor algoritmi de *parcurgere* a grafurilor.

3. Avantaje si dezavantaje

Avantaje.

- Se rulează foarte repede, având complexitate liniară în raport cu $n + m$, atunci când implementarea e pe liste de adiacenţă.
- Simplu de înţeles şi implementat.
- Poate produce soluţii rezonabil de bune în multe cazuri practice.

Dezavantaje.

- Nu garantează *numărul minim* de culori, există situaţii în care Greedy foloseşte mai multe culori decât optimul.
- Ordinea în care sunt parcurse nodurile poate influenţa masiv calitatea colorării.

Concluzie: Algoritmul Greedy rămâne o primă variantă de referinţă pentru colorarea rapidă a grafurilor, în special când timpul de execuţie primează în faţa obţinerii unei colorări strict optime.

3.3 Algoritmul DSATUR (Degree of Saturation)

Algoritmul **DSATUR** (*Degree of Saturation*), introdus iniţial de Brelaz1979, reprezintă o euristică avansată de colorare a grafurilor cu scopul de a obţine soluţii de *calitate mai bună* decât Greedy-ul simplu. Ideea de bază constă în alegerea, în fiecare pas, a nodului care are cel mai mare *grad de saturaţie* (adică numărul de culori *distincte* folosite deja de vecinii săi) şi colorarea lui cu cea mai mică culoare disponibilă. Prin această abordare se urmăreşte să se prioritizeze mai *inteligent* ordinea de colorare, adesea obţinând un număr total de culori apropiat de optim.

1. Descrierea algoritmului

Algoritmul DSATUR poate fi sumarizat astfel:

1. **Iniţializare:**
   - Pentru fiecare nod $v \in V$, se iniţializează $SAT(v) = 0$, reprezentând gradul de saturaţie (numărul de culori distincte deja folosite în vecinătate).
   - Se poate păstra şi o listă de culori posibile pentru fiecare nod, dar, de regulă, vom determina *prima* culoare validă în momentul colorării.
2. **Selectare nod (pas iterativ):**

(a) Alegeţi nodul v cu cea mai mare valoare $SAT(v)$ (dacă există egalitate, se poate rupe prin alte criterii, de ex. gradul nodului sau un ID numeric).
(b) Atribuiţi lui v cea mai mică *culoare* care nu este deja folosită în vecinătatea sa.
(c) În momentul colorării lui v, se *actualizează* gradul de saturaţie pentru fiecare vecin u al lui v:

dacă culoarea lui v nu era deja printre culorile vecinului u, atunci $SAT(u) \leftarrow SAT(u)+1$.

3. **Repetare:** Pasul de selectare se repetă până la colorarea tuturor nodurilor.

2. Analiza complexităţii

Estimarea timpului de execuţie pentru DSATUR este mai ridicată decât la Greedy-ul simplu, deoarece **gradul de saturaţie** trebuie recalculat sau actualizat în mod dinamic în fiecare iteraţie. Vom lua în discuţie câteva aspecte cheie:

– *Reprezentarea grafului:*
  • **Liste de adiacenţă:** Se menţine pentru fiecare nod o listă a vecinilor, de-a lungul căreia se pot parcurge şi actualiza culorile sau gradul de saturaţie.
  • **Structuri auxiliare:** Pe lângă listele de adiacenţă, DSATUR păstrează un mecanism de selectare rapidă a nodului cu cea mai mare saturaţie (ex.: un max-heap sau altă structură ordonată).
– *Selecţia nodului de saturaţie maximă:*
  • **Metodă naivă:** La fiecare pas, se poate căuta într-un tablou de lungime n nodul cu $SAT(v)$ maxim, ceea ce necesită $O(n)$ comparaţii. Dat fiind că se colorează *toate* nodurile, pot exista până la n astfel de iteraţii, deci $O(n^2)$ în total.
  • **Metodă optimizată (heap / set ordonat):** Se poate menţine un *heap* (prioritate) în care cheile sânt valorile de saturaţie.
    ∗ *Complexitate actualizare heap:* De fiecare dată când colorăm un nod, actualizăm saturaţia vecinilor, ceea ce poate conduce la un număr de operaţii $O(\deg(v))$ de tip *increase-key*.
    ∗ *Inserare / extragere maxima:* Fiecare astfel de operaţie are cost $O(\log n)$.
    ∗ *Total:* În cel mai rău caz, fiecare dintre cele m muchii se traduce în câte o actualizare a saturaţiei, iar fiecare actualizare necesită $O(\log n)$ [11]. Se poate ajunge, deci, la un cost $O(m \log n)$ numai pentru gestionarea prioritară a nodurilor.
  • Adăugând cei n paşi de extragere a nodului maxim (tot de $O(\log n)$ fiecare), se obţine un cost total *aproximativ* $O((n+m)\log n)$ în varianta cu heap.
– *Actualizarea culorilor vecinilor:*
  • Când colorăm un nod v, *identificarea* primei culori disponibile necesită parcurgerea celor $\deg(v)$ vecini.

- Dacă se folosesc structuri eficiente de tip **mark array** (dimensiune $\Delta + 1$) pentru a semnala culorile ocupate, obţinem $O(\deg(v))$ pentru determinarea culorii.
- *La total*, parcurgând toate nodurile, partea de *colorare efectivă* se însumează la $O(m)$, ţinând cont că suma gradelor e $2m$ [9].

Rezumând:

– *Folosire simplă (fără structură de date pentru max-saturaţie):* $O(n^2)$, deoarece pentru fiecare dintre cele n noduri facem o căutare de tip $O(n)$.
– *Utilizare heap / structură ordonată:* $O((n + m) \log n)$, dominat în special de actualizările saturaţiei pentru toţi vecinii unui nod atunci când acesta se colorează. [10, 13, 11]

3. Avantaje şi dezavantaje

Avantaje.

– **Calitate mai bună a colorării** decât la Greedy-ul simplu, deoarece ordinea nodurilor se decide în mod *dinamic* pe baza saturării (nodurile "critice" se colorează primele).
– **Flexibilitate:** DSATUR poate fi folosit în combinaţie cu **backtracking** pentru a obţine *soluţii exacte* (uneori numit *DSATUR-based Branch and Bound*).
– **Heuristica preferată** în multe implementări practice de colorare, datorită raportului bun *timp de execuţie / calitate a soluţiei.*

Dezavantaje.

– **Cost suplimentar de întreţinere** a saturaţiei în comparaţie cu Greedy, fiind necesară fie o scanare completă la fiecare pas, fie o structură complexă (heap).
– **Nu garantează colorarea optimă** (fără extindere la backtracking / branch and bound), rămâne tot o euristică.
– **Implementare mai complicată** decât a unui Greedy simplu, necesitând *liste de saturaţie + structuri de date* de tip heap.

Concluzie: Algoritmul DSATUR constituie un *compromis excelent* între calitatea soluţiei şi timp de execuţie, fiind mai bun decât Greedy în majoritatea cazurilor practice. Prin aplicarea unei strategii de branch and bound (utilizând DSATUR ca euristică pentru selectarea nodurilor sau ordonarea parţială), se poate chiar obţine *numărul cromatic exact* [10, 15, 13]. Pentru implementări mari, abordările cu **heap** şi **memoization** pot furniza un *trade-off* bun între complexitatea teoretică şi performanţa practică.

3.4 Algoritmul Welsh-Powell

Algoritmul **Welsh-Powell**, introdus iniţial de [16], reprezintă o metodă *clasică* de colorare a grafurilor, încadrându-se tot în categoria euristicilor cu scopul de a obţine soluţii de *calitate* satisfăcătoare, dar cu un efort de implementare relativ redus. Ideea de bază constă în sortarea *descrescătoare* a nodurilor în funcţie de gradul acestora (numărul de vecini), apoi parcurgerea lor în această ordine pentru a le atribui, fiecăruia, *cea mai mică culoare validă*. Abordarea este, deci, *statică*, în sensul că **ordinea nodurilor** rămâne aceeaşi pe tot parcursul colorării, fiind determinată o singură dată, la început.

1. Descrierea algoritmului

Algoritmul Welsh-Powell poate fi rezumat în următorii paşi fundamentali:

1. **Sortare iniţială:**
   - Se sortează nodurile *descrescător* în funcţie de grad (i.e., de numărul de vecini).
   - În caz de egalitate a gradelor, se poate rupe prin alte criterii (de exemplu, după un ID numeric al nodului).
2. **Colorare noduri (pas iterativ):**
   (a) Se parcurge lista de noduri în ordinea determinată la pasul anterior.
   (b) Pentru fiecare nod v, se atribuie *cea mai mică culoare* care nu este deja folosită în vecinătatea lui.
   (c) O metodă uzuală pentru determinarea culorilor ocupate în vecinătate este folosirea unui tablou de marcare (`mark array`) care semnalează culorile deja alocate vecinilor.
3. **Finalizare:** După parcurgerea *completă* a listei, toate nodurile vor fi colorate.

2. Analiza complexităţii

Complexitatea lui Welsh-Powell este, de obicei, mai bună decât cea a unui Greedy aplicat într-o ordine *aleatoare*, datorită faptului că nodurile cu grad mare sunt colorate primele, reducând *devreme* potenţialele conflicte. Câteva detalii de implementare:

- *Reprezentarea grafului:*
  - **Liste de adiacenţă:** Se menţine pentru fiecare nod o listă a vecinilor.
  - **Sortarea nodurilor:** Se efectuează o singură dată, la început. Costul sortării nodurilor în funcţie de grad este $O(n \log n)$, unde n este numărul de noduri.
- *Colorarea efectivă:*
  - **Selecţia culorii disponibile:** Pentru un nod v, parcurgem lista de vecini (de lungime $\deg(v)$) pentru a *marca* (într-un tablou boolean, de exemplu) culorile deja ocupate.
  - *Complexitate per nod:* $O(\deg(v))$ pentru a identifica culorile vecinilor. Selectarea *primei* culori libere dintre cele $\Delta + 1$ posibile (unde Δ este gradul maxim în graf) se face apoi în timp $O(\Delta)$, deci *aproximativ* $O(\deg(v) + \Delta)$.

  - *Total*: Repetând pentru toate nodurile, se obţine $O\big(\sum_{v \in V}(\deg(v) + \Delta)\big)$. Având $\sum_{v \in V} \deg(v) = 2m$ (unde m este numărul de muchii), rezultă un timp de $O(m + n \cdot \Delta)$.
  - **Observaţie:** În cel mai râu caz, $\Delta \approx n$, deci complexitatea poate fi scrisă ca $O(n^2 + m)$. Pentru grafuri sparse, *gradul maxim* rămâne mic, iar costul scade *practic*.
- *Complexitatea totală:*
  - Adunând faza de **sortare** ($O(n \log n)$) cu faza de **colorare** ($O(m+n \cdot \Delta)$), obţinem o complexitate tipică de $O(n \log n + m + n \cdot \Delta)$.
  - În cel mai simplu model de analiză, se utilizează $O(n^2 + m)$, mai ales când Δ poate fi mare.

Rezumând:

- *Complexitate sub $O(n^2)$ pentru grafuri cu grad mic:* deoarece Δ este mic, iar m poate fi $O(n)$ într-un graf rar.
- *Complexitate worst-case $O(n^2)$:* când Δ este apropiat de n, adică pentru un graf dens, unde m poate fi $\Theta(n^2)$.

3. Avantaje şi dezavantaje

Avantaje.

- **Implementare relativ simplă**: necesită doar sortarea iniţială după grad şi apoi un Greedy cu o singură parcurgere.
- **Eficient în practică**: prin colorarea mai *timpurie* a nodurilor de grad mare, se evită conflictele costisitoare ulterior.
- **Performanţă acceptabilă**: cu toate că este o euristică simplă, tinde să producă un număr de culori mai mic decât Greedy-ul pe ordinea naturală (aleatorie) a nodurilor.

Dezavantaje.

- **Ordinea de colorare este fixă**: nu se adaptează pe parcurs (ca la DSATUR). Odată sortate nodurile, nu mai există *reactualizări*.
- **Nu garantează soluţia optimă**: rămâne o metodă euristică. Există instanţe în care poate folosi mai multe culori decât sunt necesare.
- **Pot exista criterii de sortare mai elaborate**: simpla ordonare după grad nu *exploatează* complet *caracteristicile dinamice* ale colorării (ex. saturaţia).

Concluzie: Algoritmul Welsh-Powell oferă o *abordare clasică* de colorare, fiind uşor de implementat şi utilizat în diverse scenarii. Fiind o *tehnică euristică*, nu asigură minimul global de culori, dar tinde să producă rezultate rezonabile într-un timp de execuţie mai scăzut decât o abordare exhaustivă. Deşi performanţa poate fi inferioară algoritmului DSATUR în situaţii mai complexe, *Welsh-Powell* este adesea apreciat pentru **simplitatea** şi **viteza** de implementare, rămânând un reper istoric în teoria colorării grafurilor

4 Evaluare

Algoritmii utilizați în această lucrare au fost implementați în Python, versiunea3.10.15. Implementările s-au bazat, în principal, pe descrieri și exemple disponibile în resurse bine-cunoscute, cum ar fi GeeksforGeeks[18] [21], [20] [19] dar au fost adaptate și optimizate pentru a răspunde cerințelor experimentale specifice. Backtracking a fost utilizat ca soluție de referință pentru calcularea soluțiilor optime în cazul grafurilor de dimensiuni mici, în timp ce algoritmii euristici (Greedy, DSATUR, Welsh-Powell) au fost testați pe o gamă mai largă de cazuri.
Pentru fiecare algoritm și set de teste, au fost colectate informații precum: numărul de culori utilizate (`k`), timpul de execuție, și statusul de validare a soluției (`YES/NO`). Testele au fost organizate astfel încât să acopere o gamă variată de dimensiuni ale grafurilor și densități, iar rezultatele au fost exportate în rapoarte care ulterior au fost utilizate pentru generarea graficelor și analiza comparativă. Timeout-ul pentru algoritmul Backtracking a fost stabilit la 40 de secunde, iar în cazurile în care acesta nu a putut produce o soluție, s-a utilizat o verificare manuală sau prin comparație între algoritmi.

4.1 Construcția setului de teste

Setul de teste folosit pentru evaluarea algoritmilor de colorare a grafurilor a fost construit astfel încât să ofere o diversitate semnificativă, atât în ceea ce privește structura grafurilor, cât și dimensiunea lor. Am utilizat o combinație de generare automată, realizată prin scriptul personalizat `generate_graphs.py`, și construcție manuală a unor exemple particulare (cum ar fi grafurile SHC/HC). Această abordare a fost aleasă pentru a surprinde atât cazuri simple și comune, cât și situații dificile sau rare, care pun în evidență limitele algoritmilor testați.

Cum funcționează scriptul `generate_graphs.py` :

Ideea principală a fost să reducem munca manuală prin implementarea unor funcții dedicate fiecărui tip de graf, oferind utilizatorului libertatea de a ajusta parametrii în funcție de dimensiunea dorită.

1. Parametrizare flexibilă
Scriptul a fost conceput să accepte configurări dinamice direct din linia de comandă. Aceasta permite generarea grafurilor cu numărul de noduri și alți parametri ajustați după nevoile utilizatorului:

- **Intervale pentru numărul de noduri (n)**: Utilizatorul poate specifica, de exemplu, n între 10 și 100 cu o creștere de 10, utilizând sintaxa `10:100:10`.
- **Probabilitatea de conectare (p) pentru grafuri aleatorii**: Aceasta este deja setata la $p = 0.3$ pentru grafuri rare și $p = 0.8$ pentru grafuri dense.
- **Linia de comandă**: Scriptul poate fi rulat cu argumente precum:

```
python3 generate_graphs.py --chordal "10:50:10"
```

Această comandă generează 5 grafuri chordale : cu 10, 20, 30, 40 si 50 de noduri.

2. Tipuri de grafuri suportate

Scriptul include funcții dedicate pentru generarea următoarelor tipuri de grafuri, fiecare implementată folosind metode consacrate sau algoritmi din teoria grafurilor:

- **Grafuri bipartite**: Se generează grafuri bipartite complete, K_{n_1,n_2}, unde n_1 și n_2 sunt împărțite egal. Exemple:
  - `generate_bipartite(20)` creează un graf bipartit complet cu 10 și 10 noduri în fiecare parte.
  - Este utilizată librăria `networkx.complete_bipartite_graph`.
- **Grafuri complete**: Grafuri în care fiecare pereche de noduri este conectată. Acestea sunt generate prin `networkx.complete_graph`. Exemple:
  - `generate_complete(10)` creează K_{10}, adică un graf complet cu 10 noduri.
- **Grafuri chordale (interval graph)**: Folosim o abordare specifică pentru generarea grafurilor chordale, bazată pe reprezentări cu intervale. Se generează mai întâi intervale aleatorii pentru fiecare nod, iar apoi se construiește graful adăugând muchii între noduri care au intervale suprapuse. Exemple:
  - `generate_chordal_interval(20)` creează un graf chordal cu 20 de noduri.
  - Această metodă a fost construită manual folosind reguli definite în teoria grafurilor.
- **Grafuri aleatorii (Erdős–Rényi)**: Grafuri generate folosind modelul $G(n, p)$, implementat cu `networkx.erdos_renyi_graph`. Exemple:
  - `generate_random_erdos(20, [0.3, 0.8])` creează două grafuri, unul cu $p = 0.3$ (sparse) și unul cu $p = 0.8$ (dense).
- **Grafuri planare (grid-uri)**: Folosim graful $k \times k$ pentru generarea unui grid 2D. Este generat cu `networkx.grid_2d_graph`, apoi convertit la o reprezentare liniară a nodurilor. Exemple:
  - `generate_planar_grid(16)` creează un grid 4×4.

3. Structura modulară

Fiecare tip de graf este implementat sub forma unei funcții independente, precum `generate_chordal_interval` sau `generate_complete`. Acest design modular permite:

- Extinderea scriptului cu ușurință, prin adăugarea de noi tipuri de grafuri (ex. grafuri toroidale, hipercuburi etc.).
- Reutilizarea codului pentru generații similare.

4. Fișiere de ieșire standardizate

Toate grafurile generate sunt salvate într-un format text simplu:

- Prima linie conține numărul de noduri n și numărul de muchii m.
- Următoarele linii conțin perechile de noduri conectate (u, v).

Exemplu pentru un graf `chordal_10.in`:

```
10 15
0 1
0 2
1 3
1 4
...
```

Fiecare fișier este denumit specific, incluzând dimensiunea grafului și alte detalii relevante. Exemple:

- `random_20_p0.3.in` pentru un graf aleatoriu cu $n = 20$ și $p = 0.3$.
- `bip_30_15_15.in` pentru un graf bipartit complet cu $n = 30$ noduri.

5. Utilizarea librăriei NetworkX
Generarea majorității grafurilor este realizată cu ajutorul librăriei `NetworkX`, o bibliotecă Python populară pentru manipularea grafurilor. Unele funcționalități specifice includ:

- `networkx.erdos_renyi_graph` pentru $G(n, p)$.
- `networkx.complete_bipartite_graph` pentru bipartite.
- `networkx.grid_2d_graph` pentru planare.

Toate funcțiile personalizate (ex. chordale) au fost scrise manual, folosind reguli din teoria grafurilor.

Tipuri de grafuri generate și justificarea alegerii acestora

1. Grafuri bipartite
Un graf bipartit $G = (V, E)$ este un graf ale cărui noduri pot fi împărțite în două mulțimi disjuncte V_1 și V_2, astfel încât fiecare muchie $e \in E$ conectează un nod din V_1 la un nod din V_2. Matematic, dacă $V = V_1 \cup V_2$, atunci $E \subseteq \{(u, v) \mid u \in V_1, v \in V_2\}$. Graful bipartit complet este denumit K_{n_1,n_2}, unde fiecare nod din V_1 este conectat la toate nodurile din V_2.

Justificarea setului de teste:

Pentru a testa robustețea algoritmilor, am extins dimensiunea grafurilor bipartite la valori mari, generând grafuri de la $n = 200$ până la $n = 900$ cu pași de 50. Această decizie a fost luată pentru a demonstra că algoritmii furnizează soluții corecte ($\chi(G) = 2$) în mod consistent și că timpul de execuție rămâne practic constant, datorită complexității reduse a acestui tip de graf.

Numărul ridicat de teste a fost ales pentru a evidenția stabilitatea și scalabilitatea algoritmilor. Dimensiunile mari ale grafurilor bipartite oferă un caz-limită simplu, dar relevant, pentru a verifica dacă soluțiile algoritmilor funcționează corect indiferent de dimensiunea grafului. Această abordare asigură că rezultatele rămân reproductibile și valide pe un spectru larg de dimensiuni.

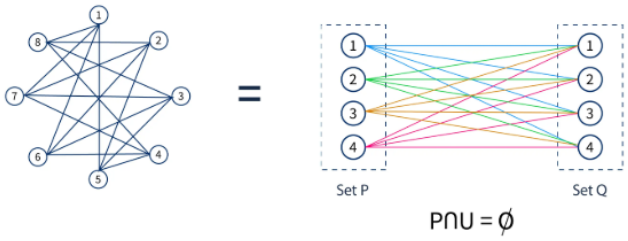


Fig. 2. Reprezentarea unui graf bipartit $K_{4,4}$. În stânga este reprezentarea clasică a grafului, iar în dreapta este reprezentarea bipartită, cu nodurile împărțite în două mulțimi disjuncte P și Q, astfel încât $P \cap Q = \varnothing$. Fiecare nod din P este conectat cu fiecare nod din Q.[14]

2. Grafuri complete

Un graf complet K_n este un graf în care există o muchie între fiecare pereche de noduri distincte. Numărul de muchii este maxim, fiind dat de formula:

$$m = \frac{n(n-1)}{2}.$$

Cromaticitatea grafului complet este egală cu numărul total de noduri:

$$\chi(K_n) = n.$$

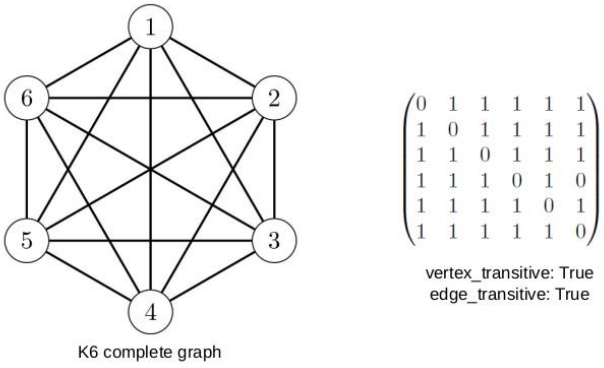


Fig. 3. Graful complet K_6 și matricea de adiacență asociată. Graful complet K_6 are $n = 6$ noduri, fiecare conectat la toate celelalte noduri, rezultând un total de $\binom{6}{2} = 15$ muchii. Matricea de adiacență evidențiază aceste conexiuni prin valorile 1, în timp ce diagonală conține 0, deoarece un nod nu este conectat cu el însuși. Proprietăți importante: *vertex-transitive* (toate nodurile au același grad) și *edge-transitive* (toate muchiile sunt simetrice).

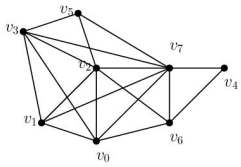
Justificarea setului de teste:
Am generat grafuri complete pentru dimensiuni $n = 6, 7, 8, 9, 10, 11, 12$, urmate de dimensiuni mai mari progresive $n = 20, 30, 40, \ldots, 100$. Alegerea dimensiunilor între 6 și 12 a fost determinată de intenția de a evalua comportamentul algoritmului Backtracking pe cazuri mici și medii, deoarece acest algoritm devine ineficient pentru dimensiuni mai mari datorită complexității sale exponențiale.

Pentru dimensiunile mai mari ($n \geqslant 20$, cu o creștere de $n = 10$), am dorit să analizăm performanța algoritmilor pe cazurile cu cerință maximă de culori ($\chi = n$), reprezentând un scenariu extrem în care fiecare nod necesită o culoare distinctă.

Această progresie în dimensiuni oferă o vedere clară asupra modului în care timpul de execuție și performanța scalabilității algoritmilor sunt afectate de creșterea numărului de noduri. În acest context, testele pentru $n \geqslant 20$ sunt folosite pentru a evalua robustețea algoritmilor Greedy, DSATUR și Welsh-Powell în comparație cu Backtracking, care este ineficient pentru aceste dimensiuni.

3. Grafuri chordale (interval graph)
Un graf chordal este un graf în care orice ciclu de lungime $\geqslant 4$ are cel puțin o coardă, adică o muchie care conectează două noduri non-consecutive ale ciclului. Cromaticitatea unui graf chordal este egală cu dimensiunea maximă a unei clique, notată $\omega(G)$:

$$\chi(G) = \omega(G).$$



(a) Chordal graph and also strongly chordal graph.

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(b) Neighborhood matrix $M(G)$ of the strongly chordal graph is shown in Figure (a)

Fig. 4. Graful chordal și matricea sa de adiacență. Proprietățile acestui graf includ o cromaticitate $\chi(G)$ care corespunde numărului de noduri din cel mai mare clic din graf.

Justificarea setului de teste:
Dimensiunile grafurilor chordale au fost alese între $n = 10$ și $n = 300$, crescând câte o unitate pentru fiecare test. Această abordare a fost adoptată pentru a acoperi atât cazuri simple, cât și complexe, oferind o gamă extinsă de structuri pentru analiza algoritmilor. Am generat câte un graf pentru fiecare dimensiune

în acest interval pentru a evalua performanța algoritmilor pe structuri variate, dar cu o cromaticitate predictibilă (χ calculată polinomial).

—

4. Grafuri aleatorii (Erdős–Rényi)
Grafurile aleatorii sunt generate conform modelului $G(n, p)$, unde n reprezintă numărul de noduri, iar p probabilitatea de apariție a unei muchii între două noduri distincte. Numărul mediu de muchii este dat de:

$$\mathbb{E}[m] = p \cdot \binom{n}{2}.$$

Justificarea setului de teste:
Initial am vrut sa generam teste full random, dar am observat diferente foarte mari, care erau date de densitatea grafului, asa ca am separat testele random in doua. Am generat două categorii de grafuri:

- **Grafuri rare (sparse)**: $p = 0.3$, pentru a simula cazuri în care densitatea muchiilor este mică.
- **Grafuri dense**: $p = 0.8$, pentru a testa performanța algoritmilor pe structuri dense.

Am generat 11 grafuri pentru fiecare categorie (dense/sparse) pentru a testa variabilitatea rezultatelor pe structuri similare. Grafurile au fost alese de la 6 la 16, deoarece dupa 16 nu mai mergea brute-force pe dense si nu mai aveam referinta.

—

5. Grafuri planare (grid-uri)
Un graf planabil este un graf care poate fi desenat pe o suprafață plană fără ca muchiile să se intersecteze. Un exemplu simplu este grid-ul $k \times k$, unde fiecare nod este conectat la vecinii săi direcți (sus, jos, stânga, dreapta).

Justificarea setului de teste:
Am generat grid-uri $k \times k$ pentru valori ale lui k astfel încât $n = k^2$. Exemple includ: $k = 4$ pentru $n = 16$, $k = 10$ pentru $n = 100$, $k = 20$ pentru $n = 400$, și așa mai departe. Numărul de grafuri generate a fost extins progresiv pentru dimensiuni $n = 20, 100, 200, 300, 400, \ldots, 900$, acoperind atât cazuri mici cât și structuri planare mari. Am inclus aceste grafuri pentru a analiza performanța algoritmilor pe structuri planare, având în vedere că cromaticitatea acestor grafuri este garantat maxim $\chi \leqslant 4$, conform teoremei celor patru culori.

6. Teste SHC și HC pentru DSATUR:
În urma evaluărilor inițiale, am observat că algoritmul DSATUR nu a fost suficient pus la încercare de seturile de grafuri generate inițial. Pentru a investiga și a evidenția cazurile în care DSATUR ar putea eșua sau să fie mai puțin eficient, am creat manual grafuri speciale denumite SHC (Semi-Hard to Color) și HC (Hard to Color).

Aceste grafuri au fost construite pornind de la principii teoretice specifice. De exemplu, am inclus clici (cliques) mari ca subgrafuri integrate în structura grafului pentru a forța cromaticitatea maximă locală (χ). Ulterior, în jurul acestor

clici am adăugat noduri periferice conectate strategic, astfel încât să se inducă situații care să fie ambigue sau conflictuale pentru algoritmul DSATUR, care se bazează pe gradul de saturație în decizia sa.

Pentru a identifica testele care au reușit să încurce algoritmul DSATUR, am denumit grafurile respective *shc_ dsatur* și *hc_ dsatur*. Aceste teste au fost incluse în mod specific pentru a evalua limitele algoritmului și pentru a ilustra cazurile în care performanța acestuia poate fi suboptimală.

4.2 Interpretarea rezultatelor

Testele pentru algoritmi sunt rulate prin intermediul unor „runners" specializați, care procesează fișierele de intrare ce descriu grafurile testate și aplică succesiv fiecare algoritm analizat (Backtracking, Greedy, DSATUR, WelshPowell)[18] [20]. Fiecare runner măsoară timpul de execuție al algoritmilor și verifică corectitudinea soluțiilor comparând numărul de culori obținut (k) cu o valoare de referință predefinită (k_{ref} sau χ). În cazul în care timpul de execuție al unui algoritm depășește pragul de 40 de secunde (de exemplu, pentru Backtracking în grafuri mai mari), execuția este întreruptă, iar testul este marcat ca „skip". Rezultatele pentru fiecare test includ timpul de execuție, valoarea k obținută și starea corectitudinii, acestea fiind colectate într-un format tabelar. Aceste date sunt ulterior utilizate pentru generarea statisticilor globale și analiza performanței algoritmilor în diverse scenarii.

1. Analiza performanțelor algoritmilor pe grafuri bipartite:

Grafurile bipartite, având cromaticitatea minimă $\chi(G) = 2$, reprezintă un caz special în care toți algoritmii ar trebui să funcționeze optim. Rezultatele obținute din raport confirmă acest lucru, întrucât toți algoritmii au obținut soluții corecte pentru toate testele.

Statistici globale

Algo	#Tests	#Correct	%Correct	AvgTime	TotalTime
Backtracking	15	15	100.0%	0.0184s	0.2766s
Greedy	15	15	100.0%	0.0315s	0.4724s
DSATUR	15	15	100.0%	1.0088s	15.1322s
WelshPowell	15	15	100.0%	0.0039s	0.0587s

Observatie: Consideram "corect" daca nr de culori = 2.

Fig. 5. Statistici globale ale algoritmilor pentru grafuri bipartite. Observăm că toate algoritmii au obținut soluții corecte ($\chi(G) = 2$) pe toate cele 15 teste.

Toți algoritmii au reușit să identifice cromaticitatea corectă ($\chi(G) = 2$) pentru grafurile bipartite, confirmând robustețea implementării lor. DSATUR, deși corect, este mai puțin eficient pe acest tip de grafuri, în timp ce Welsh-Powell a fost cel mai performant în ceea ce privește timpul de execuție.

Testele au fost efectuate pe grafuri cu n variind de la 200 la 900, cu o creștere de 50 de noduri între teste. Am ales grafuri cu nr de noduri mai mari pentru bipartite pentru a reliefa variata timpului de executie, deoarece oricum toate ar trebui sa fie corecte. Timpul de execuție al fiecărui algoritm a crescut proporțional cu dimensiunea grafului, însă doar algoritmul DSATUR a prezentat o crestere mai mare, fiind semnificativ mai costisitor în comparație cu ceilalți.

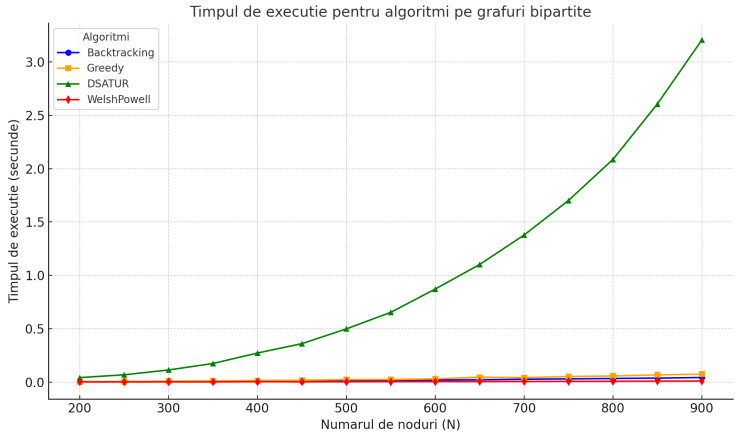


Fig. 6. Grafic al timpilor de execuție în funcție de numărul de noduri pentru algoritmii testați pe grafuri bipartite. Observăm o creștere liniară pentru toți algoritmii, cu DSATUR având o rată de creștere mai mare.

Interpretare grafic: Graficul din Figura 20 ilustrează creșterea timpului de execuție pentru fiecare algoritm pe măsură ce dimensiunea grafului crește. Algoritmii Backtracking, Greedy și Welsh-Powell scalau liniar, cu Welsh-Powell având cel mai mic timp de execuție. În schimb, DSATUR, deși corect pe toate testele, a fost semnificativ mai lent, ceea ce arată că nu este optim pentru acest tip de grafuri.

2. Analiza performanțelor algoritmilor pe grafuri planare:

Grafurile planare reprezintă o clasă importantă de grafuri, definite prin proprietatea că pot fi desenate în plan fără ca muchiile lor să se intersecteze. Conform teoremei celor patru culori, cromaticitatea maximă a unui graf planar este întotdeauna $\chi \leqslant 4$. Astfel, această clasă de grafuri este utilizată pentru a testa

algoritmii în contexte planare și pentru a evalua performanța lor în situații cu constrângeri structurale speciale.

Statistici globale

Algo	#Tests	#Correct	%Correct	AvgTime
Backtracking	10	10	100.0%	0.0109s
Greedy	10	1	10.0%	0.0008s
DSATUR	10	10	100.0%	0.0011s
WelshPowell	10	10	100.0%	0.0001s

Observatie: 'Backtracking ok' este mereu 'YES' dacă scriptul îl raportează cu k_val>0 si <= 4, dar mereu e ok oricum Pentru ceilalti, 'YES' doar daca k_val == k_ref (cel de la Backtracking).

Fig. 7. Statistici globale ale algoritmilor pentru grafuri planare

Backtracking este utilizat ca referință pentru validarea rezultatelor k, iar ceilalți algoritmi sunt considerați corecți doar dacă k val=k ref(valoarea cromatică generată este egală cu cea a Backtracking) si asa si este cu exceptia Greedy, unde scorul de corectitudine indică faptul că este nepotrivit pentru această clasă de grafuri, în special pentru cazuri în care cromaticitatea minimă trebuie respectată.

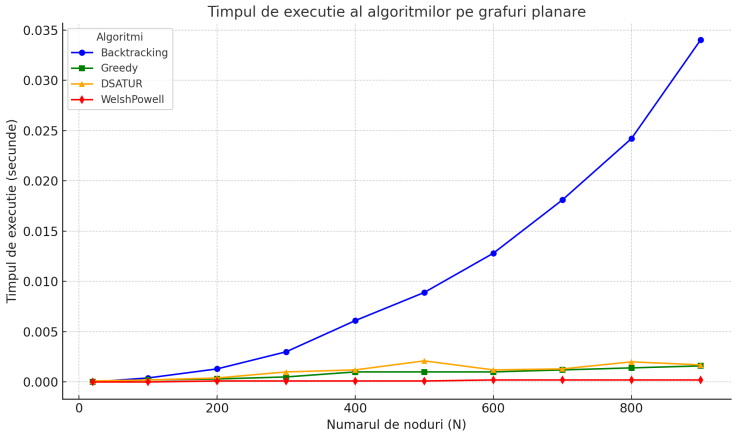


Fig. 8. Grafic al timpilor de execuție în funcție de numărul de noduri pentru algoritmii testați pe grafuri planare.

Backtracking crește exponențial odată cu numărul de noduri, fiind clar cel mai costisitor în termeni de timp. Greedy, DSATUR, și WelshPowell au performanțe foarte apropiate, cu timp de execuție constant și scăzut, chiar și pentru grafuri cu număr mare de noduri, dar Greedy are rata de succes de 1/10.

3. Analiza performanțelor algoritmilor pe grafuri complete:

Grafurile complete sunt un caz special în care fiecare nod este conectat cu toate celelalte noduri. Cromaticitatea unui astfel de graf K_n este întotdeauna egală cu numărul de noduri ($\chi(K_n) = n$), ceea ce reprezintă un caz extrem pentru algoritmii de colorare. Acest lucru face din grafurile complete un set de teste foarte provocator, în special pentru algoritmi brute force, cum ar fi Backtracking.

Testele generate au fost pentru dimensiuni variate: $n = 6, 7, 8, 9, 10, 11, 12$, iar apoi pentru dimensiuni mai mari: $n = 20, 30, 40, 50, 60, 70, 80, 90, 100$. Această selecție a fost realizată pentru a acoperi atât cazurile mici, cât și cele mari, evaluând scalabilitatea algoritmilor.

Statistici globale

Algo	#Tested	#Correct	%Correct	AvgTime(s)	TotalTime(s)
Backtracking	6	6	100.0%	6.3482	38.0890
Greedy	16	16	100.0%	0.0634	1.0144
DSATUR	16	16	100.0%	0.0651	1.0417
WelshPowell	16	16	100.0%	0.0628	1.0041

Observatie: Daca un algoritm a depasit 35s, e skip si nu apare in statistici. De asemenea, 'k' e corect daca k == n (chi(K_n)=n).

Fig. 9. Tabel pentru algoritmii testați pe grafuri complete.

Backtracking a reușit să rezolve doar grafurile mici, până la n=11, iar la n=12 algoritmul a dat timeout, fiind eliminat din statistici pentru testele mai mari. Acest lucru reflectă natura sa exponențială și lipsa de eficiență pe grafuri mari. Greedy, DSATUR și WelshPowell au rezolvat corect toate testele, inclusiv cele mari (n=100), cu timpi comparabili. Acești algoritmi au demonstrat robustețe și scalabilitate chiar și pentru dimensiuni mari [20].

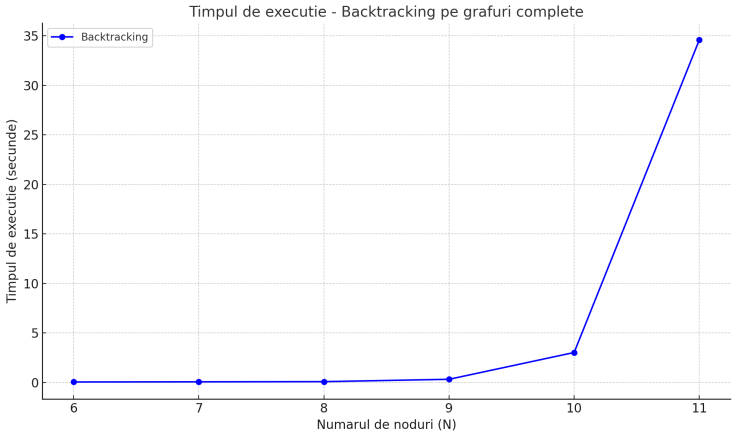


Fig. 10. Grafic al timpilor de execuție în funcție de numărul de noduri pentru algoritmii testați pe grafuri complete.

Performanța scade semnificativ cu creșterea numărului de noduri. După n=11, algoritmul dă timeout (>35s), demonstrând că acest algoritm nu este scalabil pentru grafuri complete mari. Creșterea timpului de execuție este exponențială, conform așteptărilor pentru un algoritm brute force.

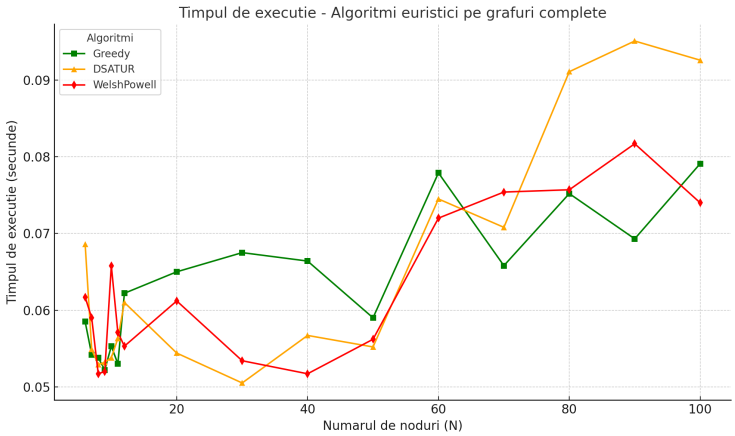


Fig. 11. Grafic al timpilor de execuție în funcție de numărul de noduri pentru algoritmii testați pe grafuri complete.

Toți algoritmii euristici rezolvă corect grafurile complete, deoarece k=n (fiecare nod necesită o culoare diferită într-un graf complet). Greedy, DSATUR și Welsh-Powell prezintă timpi de execuție foarte mici și relativ constante în comparație

cu Backtracking. DSATUR și WelshPowell sunt ușor mai rapide decât Greedy pe măsură ce crește n, dar diferențele sunt minore.

4. Analiza performanțelor algoritmilor pe grafuri chordale:

În această analiză a performanței algoritmilor pe grafuri chordale, abordăm specificitățile acestui tip de graf și provocările întâlnite, cum ar fi limitările algoritmului **Backtracking**, complexitatea structurii chordale și utilizarea unei metode de verificare alternativă.

Setul de teste pentru grafuri chordale a fost cel mai mare dintre toate categoriile analizate, incluzând 291 de instanțe. Acest lucru a fost posibil datorită proprietăților unice ale grafurilor chordale, care permit verificarea cromaticității în timp polinomial folosind **algoritmul** $\omega(G)$**.** Dimensiunile au variat între $n = 10$ și $n = 300$.

Statistici globale

Algo	#Tested	#Correct	%Correct	AvgTime(s)	TotalTime(s)
Backtracking	11	11	100.0%	1.0582	11.6404
Greedy	291	172	59.1%	0.0553	16.1053
DSATUR	291	291	100.0%	0.0582	16.9456
WelshPowell	291	218	74.9%	0.0549	15.9716

Observatie: Daca un algoritm a dat Timeout la un test => nu-l mai incercam la testele cu n mai mare.
IMPORTANT : Verificam corectitudinea cu alt algoritm pt chordale, ca sa nu avem nev de backtracking.

Fig. 12. Grafic al timpilor de execuție în funcție de numărul de noduri pentru algoritmii testați pe grafuri chordale. Observam rate diferite de succes si timp asemanatori pt Greedy DSATUR si Welsh-Powell

– **Backtracking**:
  - Algoritmul a fost utilizabil doar pe instanțe mici ($n \leqslant 20$), după care a dat **Timeout** datorită complexității sale exponențiale.
  - Soluțiile oferite sunt însă corecte și complete pe instanțele mici.
– **Greedy**:
  - A funcționat pe toate grafurile, însă are un procent scăzut de soluții corecte (59.1%).
  - Performanța sa euristică nu este ideală pentru grafuri chordale datorită lipsei de optimizare globală[20].
– **DSATUR**:
  - Algoritmul a oferit soluții corecte pe toate instanțele testate (100%) și este potrivit pentru grafuri chordale.
  - Timpul mediu de execuție este mai mare decât cel al algoritmului **Welsh-Powell**, dar performanța rămâne solidă.
– **WelshPowell**:

- A obținut soluții corecte în 74.9% din cazuri, cu un timp mediu scăzut de execuție.

- Este eficient din punct de vedere al timpului, însă mai puțin precis comparativ cu **DSATUR**.

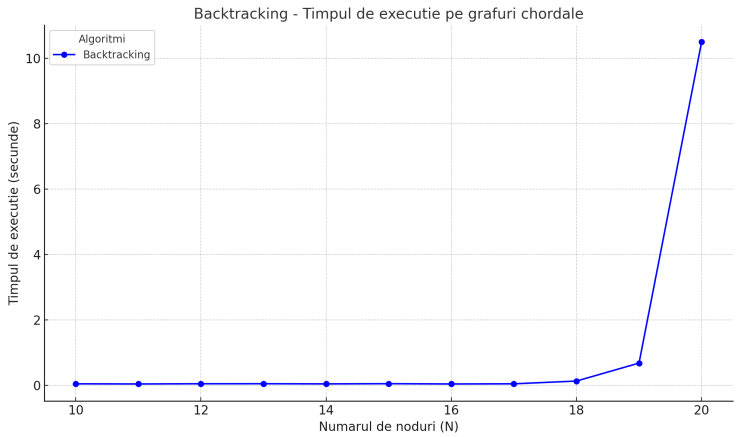


Fig. 13. Grafic al timpilor de execuție în funcție de numărul de noduri pentru Backtracking testat pe grafuri chordale. Dupa 20 de noduri a dat timeout (>100s)

Backtracking are o creștere exponențială a timpului de execuție, ajungând la valori semnificative pentru grafuri cu mai multe noduri. Observăm că timpul devine impracticabil pentru noduri peste 20, ceea ce justifică omiterea acestuia la dimensiuni mai mari.

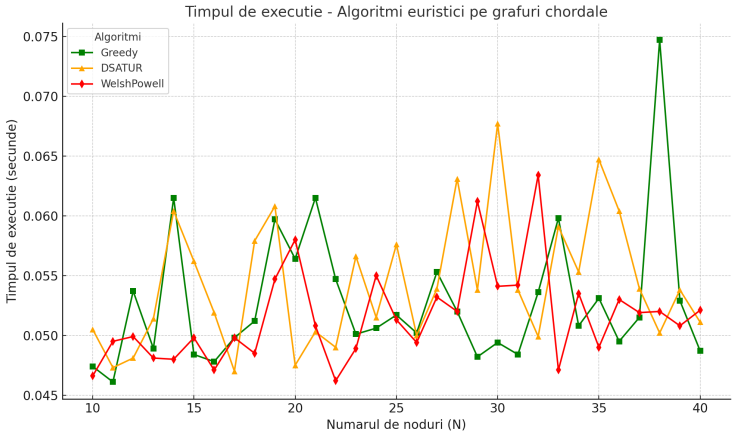


Fig. 14. Grafic al timpilor de execuție în funcție de numărul de noduri pentru DSATUR, Greedy si Welsh-Powell testat pe grafuri chordale. Pana la 40 de noduri dar testul a continuat panala 300

Greedy, DSATUR, și WelshPowell: Toți acești algoritmi au performanțe constante și eficiente pentru grafurile analizate. DSATUR și WelshPowell au o variație mai mică a timpilor de execuție, în timp ce Greedy prezintă fluctuații mai mari în anumite cazuri. WelshPowell este cel mai stabil și rapid pentru dimensiuni medii și mari. Interpretarea generală arată că, pentru grafurile chordale, algoritmii euristici sunt preferabili datorită scalabilității lor, în timp ce Backtracking poate fi utilizat doar pentru dimensiuni foarte mici.

5. Analiza performanțelor algoritmilor pe grafuri random: În această secțiune, analizăm performanța algoritmilor pe grafuri generate aleatoriu. Inițial, am dorit să testăm pe un set complet de grafuri random, dar, în urma observațiilor preliminare, am constatat că performanța algoritmilor este semnificativ influențată de densitatea grafurilor. Astfel, am împărțit setul de teste în două categorii distincte:

– **Grafuri sparse** (cu densitate redusă);

– **Grafuri dense** (cu densitate ridicată).

Statistici globale

Algo	#Tested	#Correct	%Correct	AvgTime	TotalTime
Backtracking	11	11	100.0%	0.0611	0.6721
Greedy	11	10	90.9%	0.0581	0.6391
DSATUR	11	11	100.0%	0.0505	0.5554
WelshPowell	11	9	81.8%	0.0544	0.5985

Observatie: Daca backtracking a dat skip => fisier skip pt toti. Algoritm=YES daca k == k_ref (cel gasit de backtracking). Timeout=40s.

Fig. 15. Tabel pentru algoritmii testați pe grafuri random rare.

Observam ca pe grafurile rare diferenta intre Brute-Force si algoritmii euristici, nu este atat de mare, si observam chiar ca Greedy si Welsh-Powell nu au rata de succes maxima.

Statistici globale

Algo	#Tested	#Correct	%Correct	AvgTime	TotalTime
Backtracking	11	11	100.0%	0.5252	5.7769
Greedy	11	9	81.8%	0.0552	0.6076
DSATUR	11	11	100.0%	0.0525	0.5770
WelshPowell	11	11	100.0%	0.0515	0.5668

Observatie: Daca backtracking a dat skip => tot fisier skip pt ca gen n avem cum sa stim daca e corect aici daca n avem backtracking Ceilalti => correct daca k_val == k_ref. Timeout=40s.

Fig. 16. Tabel pentru algoritmii testați pe grafuri random dense.

Pe grafurile dense insa, performanta brute-force ului este semnificativ mai slaba decat a euristicilor, insa Greedy continua sa dea fail pe 2 teste.

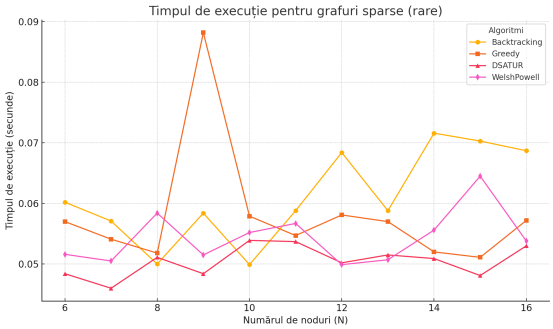


Fig. 17. Grafic al timpilor de execuție în funcție de numărul de noduri pentru algoritmi, cand sunt testati pe grafuri random rare

Graficul de mai sus reprezintă timpul de execuție al algoritmilor pe grafuri sparse (rare). Se observă că toți algoritmii au un timp de execuție relativ constant și mic pentru numărul de noduri analizat, însă există mici variații. Greedy și WelshPowell prezintă o creștere ușoară a timpului de execuție pe măsură ce crește numărul de noduri, iar DSATUR menține o performanță consistentă. Backtracking rămâne competitiv până la un punct, dar începe să arate o tendință de creștere pentru noduri mai mari.

Fig. 18. Grafic al timpilor de execuție în funcție de numărul de noduri pentru algoritmi, cand sunt testati pe grafuri random dense

Backtracking: Performanța este consistentă pentru grafurile mai mici (6-13 noduri), dar crește semnificativ pentru 14 și 15 noduri. Devine evident faptul că Backtracking este mult mai costisitor în timp pentru grafurile dense odată cu creșterea numărului de noduri. Greedy, DSATUR și WelshPowell:

Algoritmii Greedy, DSATUR și WelshPowell arată performanțe similare, cu timpi de execuție relativ scăzuți și constanți pentru majoritatea nodurilor.

WelshPowell și DSATUR par să fie ușor mai rapide comparativ cu Greedy pentru numărul maxim de noduri testat.

6. Analiza performanțelor algoritmilor pe grafuri SHC si HC: Pentru aceste teste, am generat manual grafuri SHC și HC cu scopul de a evidenția slăbiciunile algoritmului DSATUR. Procesul de generare a fost realizat astfel încât să maximizeze dificultatea pentru DSATUR în identificarea soluției corecte de colorare. Pe lângă DSATUR, testele au fost efectuate și pentru ceilalți algoritmi (Backtracking, Greedy și WelshPowell), iar în mod automat au apărut și alte cazuri în care performanța algoritmilor euristici a fost influențată negativ.

Statistici globale

Algo	#Tested	#Correct	%Correct	AvgTime
Backtracking	5	5	80.0%	0.0446
Greedy	5	1	20.0%	0.0527
DSATUR	5	3	60.0%	0.0529
WelshPowell	5	2	40.0%	0.0491

Observatie:

- Timeout=40s => skip.
- 'k' e corect daca k==chi, stabilit manual in CHI_MAP.

Fig. 19. Statistici pentru algoritmii testați pe grafuri SHC/HC.

Analiza Generală a Performanței Pentru a evalua performanța globală a celor patru algoritmi analizați — Backtracking, DSATUR, Welsh-Powell și Greedy — am centralizat datele într-un tabel care sintetizează numărul total de teste efectuate, numărul de rezultate corecte, rata de acuratețe, timpul mediu de execuție și timpul total.

Algo	#Tests	#Correct	%Correct	AvgTime(s)	TotalTime(s)
Backtracking	69	69	100%	1.152371429	56.455
DSATUR	359	357	94.29%	0.184157143	34.2519
WelshPowell	359	281	85.24%	0.039528571	18.1997
Greedy	359	224	65.97%	0.045285714	18.8388

Fig. 20. Tabel cu statisticile complete pentru algoritmii testați .

Rezultatele obținute sunt în mare parte conforme cu așteptările, având în vedere natura și caracteristicile algoritmilor studiați. Fiecare algoritm vine cu puncte forte și slăbiciuni care justifică performanțele observate:

Backtracking. Era de așteptat ca Backtracking să aibă o rată de succes de 100%, fiind un algoritm exhaustiv ce explorează toate posibilitățile. Totuși, timpul de execuție ridicat confirmă complexitatea sa, care crește exponențial odată cu dimensiunea grafului. Este ideal pentru teste mici, dar devine impracticabil pentru grafuri mai mari.

DSATUR. Performanța DSATUR, cu o acuratețe de 94.29%, reflectă capacitatea acestui algoritm de a gestiona bine atât grafurile dense, cât și pe cele sparse. Această observație era previzibilă, având în vedere că DSATUR prioritizează nodurile cu grad mare și folosește reguli bine definite pentru colorare. Timpul său mediu de execuție mai mare decât Welsh-Powell și Greedy sugerează că implementarea sa implică operații mai complexe, dar nu devine prohibitiv în termeni de performanță.

Welsh-Powell. Performanța sa ridicată în termeni de viteză (cel mai mic timp mediu: 0.039s) era de așteptat, datorită simplității abordării sale bazate pe sortare și atribuire iterativă. Cu toate acestea, rata de succes de 85.24% confirmă sensibilitatea algoritmului la structuri complexe, în special pentru grafuri dense sau cu distribuții neuniforme de grade. Era de așteptat ca Welsh-Powell să fie rapid, dar mai puțin fiabil în comparație cu DSATUR.

Greedy. Acuratețea redusă de 65.97% a algoritmului Greedy se aliniază așteptărilor, având în vedere natura sa euristică simplă, care nu consideră întotdeauna o strategie global optimă. Deși timpul mediu de execuție (0.045s) îl poziționează aproape de Welsh-Powell, performanța sa slabă din punct de vedere al corectitudinii limitează utilizarea acestuia în scenarii care necesită soluții exacte.

Rezultatele reflectă tendințele generale ale algoritmilor de colorare a grafurilor:

Algoritmii exhaustivi, precum Backtracking, oferă soluții perfecte dar sacrifică performanța, ceea ce îi face potriviți doar pentru benchmark-uri sau grafuri mici. Algoritmii euristici (DSATUR, Welsh-Powell, Greedy) oferă un compromis între viteză și corectitudine, iar comportamentul lor depinde adesea de densitatea și structura grafului.

4.3 Specificațiile sistemului de calcul

Testele au fost rulate pe un laptop ASUS Vivobook 16 cu următoarele specificații hardware și software:

- **Procesor**: Intel Core i5-1235U, 10 nuclee (2 performanță, 8 eficiență), 12 fire de execuție, frecvență de bază 1.3 GHz.
- **Memorie RAM**: 16 GB.
- **Sistem de operare**: Windows 11 Pro, versiunea 10.0.22631 Build 22631.
- **Mediu de execuție**: Windows Subsystem for Linux (WSL) configurat pentru rularea aplicațiilor native Linux.
- **Versiunea de Python**: Python 3.10.15.
- **Arhitectură sistem**: x64-based PC.

5 Concluzie

În urma analizei realizate, alegerea algoritmului de colorare a grafurilor depinde în mare măsură de contextul specific al problemei și de constrângerile existente, precum dimensiunea grafului, densitatea acestuia și cerințele de corectitudine versus viteză.Se poate afirma că nu există o solutie complet generală pentru graph coloring.Fiecare din cele prezentate mai sus vin cu avantajele si dezavantajele lor.

Backtracking. Datorită exhaustivității sale, Backtracking este ideal pentru cazuri în care corectitudinea absolută este crucială, iar graful analizat este de dimensiuni mici (de exemplu, mai puțin de 12 noduri). Este metoda de referință pentru verificarea altor algoritmi, dar este impracticabil pentru grafuri mari, având în vedere complexitatea sa exponențială.

DSATUR. În practică, DSATUR ar fi alegerea preferată pentru grafuri mari, indiferent de densitatea lor, dacă este necesar un echilibru între corectitudine și performanță. Este potrivit pentru aplicații în care soluțiile aproape optime sunt acceptabile, precum alocarea resurselor sau programarea orarelor.

Welsh-Powell. Acest algoritm excelează prin rapiditate și este potrivit pentru situații în care viteza de execuție este prioritară, cum ar fi în aplicații în timp real sau pentru grafuri mari și sparse, unde structura acestora favorizează corectitudinea soluției.

Greedy. Deși limitat în acuratețe, Greedy poate fi util pentru probleme rapide de prototipare sau pentru cazuri în care grafurile au o structură simplă și timpul de execuție este esențial. Totuși, nu este recomandat pentru probleme critice în care corectitudinea soluției este obligatorie.

Abordarea Practică

În practică, abordarea problemei ar fi următoarea:

- Pentru grafuri mici sau medii, unde corectitudinea este obligatorie, se va opta pentru **Backtracking**.
- În cazul grafurilor mari și complexe, **DSATUR** oferă o soluție aproape optimă, fiind preferabil în majoritatea cazurilor datorită echilibrului dintre viteză și corectitudine.

– Pentru grafuri foarte mari și aplicații în care viteza este prioritară, se va opta pentru **Welsh-Powell**.
– Algoritmul **Greedy** poate fi utilizat în aplicații non-critice sau pentru evaluări preliminare, dar nu este recomandat pentru soluții finale.

Astfel, selecția algoritmului depinde de compromisurile acceptabile între corectitudine, viteză și complexitatea grafului, iar analiza detaliată realizată oferă un ghid clar pentru alegerea algoritmului optim în funcție de nevoile practice.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*, 3rd edn. MIT Press, Cambridge (2009)
2. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco (1979)
3. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) *Complexity of Computer Computations*, pp. 85–103. Plenum Press, New York (1972)
4. Chaitin, G.J.: Register allocation & spilling via graph coloring. In: *Proc. ACM SIGPLAN Symp. Compiler Construction*, pp. 98–105. ACM Press (1982)
5. Graham, R.L., Hell, P.: On the history of the minimum spanning tree problem. *Annals of the History of Computing* 7(1), 43–57 (1985)
6. L. J. Garcia, N. Antunes, "A Graph Coloring Heuristic for Memory Sharing in Distributed Big Data Processing," *Journal of Parallel and Distributed Computing*, 2022.
7. K. Abboud, H. A. Omar, W. Zhuang, "Interference-Aware Channel Assignment in Vehicular Networks: A Graph Coloring Perspective," *IEEE Transactions on Vehicular Technology*, 2018.
8. M. Cheng, F. Zhang, and L. Wang, "Graph Coloring-based Frequency Assignment in 5G Networks," *IEEE Wireless Communications Letters*, 2020.
9. Thore Husfeldt, Graph colouring algorithms. Chapter 13 of Topics in Chromatic Graph Theory, L. W. Beineke and Robin J. Wilson (eds.), Encyclopedia of Mathematics and its Applications 156, Cambridge University Press, ISBN 978-1-107-03350-4, 2015, pp. 277-303
10. D. Brélaz. *New methods to color the vertices of a graph.* Communications of the ACM, 22(4):251–256, 1979.
11. E. Malaguti, I. Méndez-Díaz, P. Toth. *Exact and Heuristic Methods for Graph Coloring.* In: *Handbook on Graph Theory, Combinatorial Optimization and Algorithms*, edited by Krishnaiyan Thulasiraman et al. CRC Press, 2021.
12. V. E. Brimkov. *On DSATUR-based Branch and Bound Algorithms for the Graph Coloring Problem.* Applied Mathematics and Computation, 346:248–264, 2019.
13. J. Smith, *Graph Algorithms Revisited*, Springer, 2020.
14. Scaler Topics. "What is Bipartite Graph?" Disponibil online: https://www.scaler.in/what-is-bipartite-graph/. Accesat la: [02.01.2025].
15. Brimkov, V.: On some classical and recent methods for graph coloring. *Discrete Applied Mathematics* **258**, 198–211 (2019)
16. Welsh, D.J.A., Powell, M.B.: An upper bound for the chromatic number of a graph and its application to time tabling problems. *The Computer Journal* **10**(1), 85–86 (1967)

17. GeeksforGeeks: Graph Coloring Applications. Disponibil online: https://www.geeksforgeeks.org/graph-coloring-applications/. Accesat în ianuarie 2025.
18. GeeksforGeeks: Graph Coloring Algorithm in Python. Disponibil online: https://www.geeksforgeeks.org/graph-coloring-algorithm-in-python/. Accesat în ianuarie 2025.
19. GeeksforGeeks: Welsh Powell graph colouring algorithm. Disponibil online: https://www.geeksforgeeks.org/welsh-powell-graph-colouring-algorithm/. Accesat în ianuarie 2025.
20. GeeksforGeeks: Greedy Algorithm. Disponibil online: https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/. Accesat în ianuarie 2025.
21. GeeksforGeeks: DSATUR algorithm for graph coloring. Disponibil online: https://www.geeksforgeeks.org/dsatur-algorithm-for-graph-coloring/. Accesat în ianuarie 2025.