

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

**Algoritmi pentru testarea izomorfismului
grafurilor**

propusă de

Gabriel-Andrei Ignat

Sesiunea: iulie, 2024

Coordonator științific

Lect. Dr. Frăsinaru Cristian

UNIVERSITATEA "ALEXANDRU-IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

**Algoritmi pentru testarea
izomorfismului grafurilor**

Gabriel-Andrei Ignat

Sesiunea: iulie, 2024

Coordonator științific

Lect. Dr. Frăsinaru Cristian

Avizat,

Îndrumător lucrare de licență,

Lect. Dr. Frăsinaru Cristian.

Data:

Semnătura: 

Declarație privind autenticitatea conținutului lucrării de licență

Subsemnatul **Ignat Gabriel-Andrei** domiciliat în **România, jud. Vaslui, com. Stăniilești, sat. Chersăcosu, str. Teilor, nr. 1**, născut la data de **09 Februarie 2002**, identificat prin CNP **5020209375212**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2024, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Algoritmi pentru testarea izomorfismului grafurilor** elaborată sub îndrumarea domnului **Lect. Dr. Frăsinaru Cristian**, pe care urmează să o susțin în fața comisiei este autentică, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea autenticității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop. Declar că lucrarea de față are exact același conținut cu lucrarea în format electronic pe care profesorul îndrumător a verificat-o prin intermediul software-ului de detectare a plagiatului.

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

Data:

Semnătura:

Declarație de consimțământ

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

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

Absolvent **Gabriel-Andrei Ignat**

Data:

Semnătura:

Cuprins

Motivație	2
Introducere	5
1 Definiții și notații	6
2 Algoritmi existenți	9
2.1 Ullman	11
2.2 VF2	14
2.3 Alți algoritmi	17
2.3.1 Nauty	18
2.3.2 Babai	19
2.4 Algoritm liniar	20
3 Implementare	23
3.1 Platforma de programare Java	23
3.2 Biblioteca Graph4J	24
3.3 Arhitectura generală a algoritmilor	25
3.4 Implementare Ullman	30
3.5 Implementare VF2	32
3.6 Observații	36
3.7 Implementare algoritm pentru arbori	38
3.8 Testarea corectitudinii	41
4 Testarea performanțelor	43
4.1 VF2	43
4.2 VF2 și densități diferite ale grafurilor	49
4.3 Ullman vs VF2	49

4.4	Algoritm pentru arbori	51
5	Interfață testare algoritmi	53
5.1	Java Swing	53
5.2	Scopul aplicației	54
5.3	Arhitectura aplicației	55
5.3.1	Bara de meniu	55
5.3.2	Panoul de editare	56
5.3.3	Panoul de testare	58
5.4	Formatul standard GDF	58
	Concluzii	61
	Bibliografie	63

Motivație

În anul 1736, renumitul matematician Leonhard Euler a introdus pentru prima dată conceptul de grafuri în lucrarea sa intitulată "*Problema celor 7 poduri din Königsberg*". După multiple încercări eșuate de a rezolva problema folosind tehnici cunoscute acelei perioade, Euler a ajuns să abstractizeze problema printr-un graf (Fig. 1)¹.

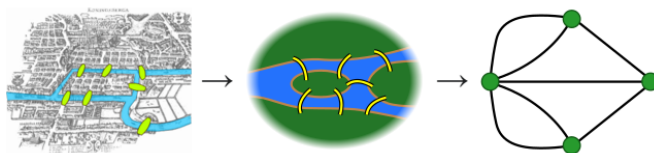


Figura 1: Abstractizare *Problema celor 7 poduri din Königsberg*

În această lucrare, Euler enunță prima sa teoremă, **Teorema Drumurilor Euleriene**, care prezintă o condiție **necesară** pentru existența unui drum eulerian într-un graf: toate nodurile grafului trebuie să aibă grad par. Astfel, el reușește să demonstreze că *Problema celor 7 poduri din Königsberg* nu are soluție.

Din acel moment teoria grafurilor devine un subiect de interes. În secolul XX are loc un progres semnificativ, grafurile devenind fundamentale în rezolvarea multor probleme practice și teoretice.

De la analiza rețelelor sociale, probleme de planificare, probleme de optimizare a costurilor până la modelarea bazelor de date de tip graf, a motoarelor de căutare online sau chiar probleme din domeniul bioinformaticii, grafurile au devenit un instrument puternic pentru a modela și a rezolva probleme complexe din diverse domenii.

În ultimele decenii, interesul pentru una dintre cele mai importante probleme din teoria grafurilor, mai exact **problema izomorfismului grafurilor**, a crescut semnificativ datorită aplicațiilor practice din domeniul recunoașterii de modele, recunoașterii de obiecte, procesarea imaginilor sau din domeniul chimiei organice și al bioinformaticii.

¹Sursa: https://en.wikipedia.org/wiki/Seven_Bridges_of_Knigsberg

Una dintre primele aplicații practice ale izomorfismului grafurilor a fost în domeniul chimiei organice. În această ramură a chimiei, molecula unei substanțe organice poate fi ușor reprezentată printr-un graf, unde atomii sunt noduri, iar legăturile dintre ei sunt muchii. Prin aplicarea izomorfismului pe aceste grafuri, putem determina dacă două substanțe organice au aceeași structură sau măcar substructuri identice, ceea ce este esențial pentru identificarea, clasificarea și înțelegerea proprietăților substanțelor chimice. Această abordare a fost crucială pentru dezvoltarea domeniului chimiei, având impact direct asupra diferitelor industrii: farmaceutică, industria materialelor, alimentară, etc.

Figura (Fig. 2) ilustrează una din cele mai timpurii aplicații practice ale problemei izomorfismului în domeniul chimiei organice: căutarea unui compus chimic într-o bază de date cu mulți alți compuși chimici. Datorită numărului imens de compuși chimici putem observa importanța găsirii unei soluții eficiente pentru această problemă.

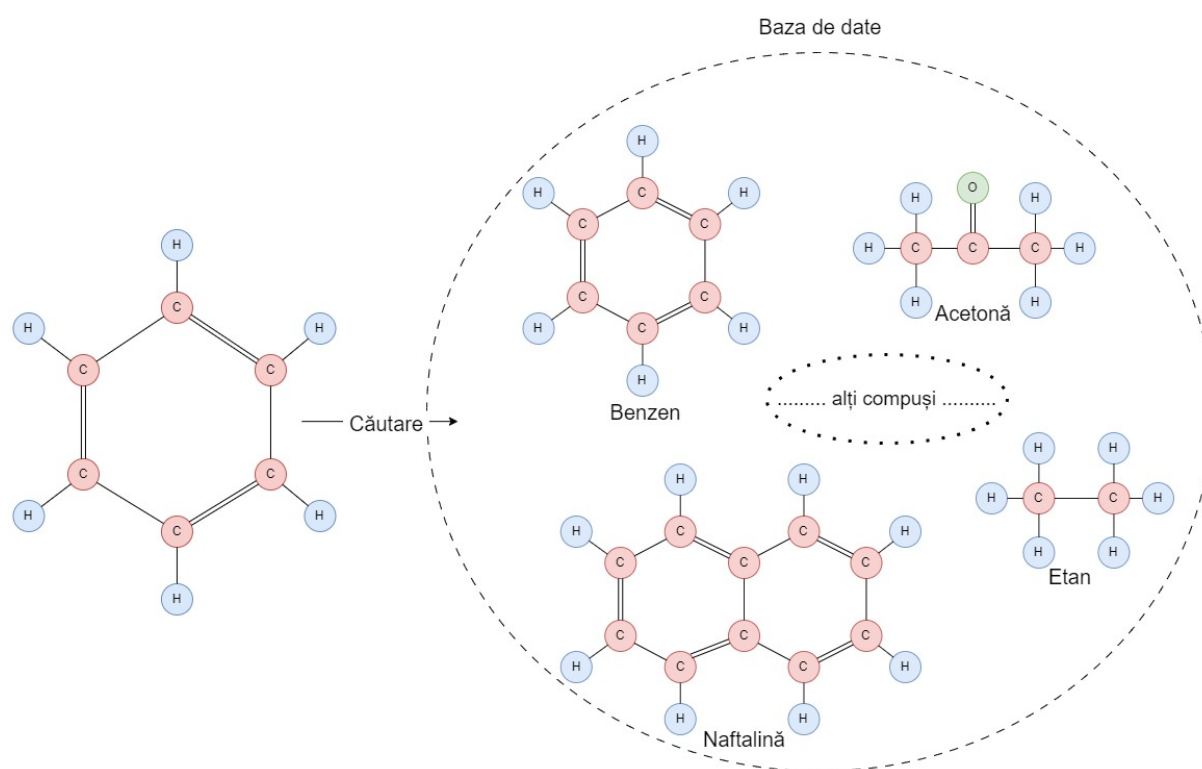


Figura 2: Problema izomorfismului aplicată în domeniul chimiei organice.

Am ales această temă din mai multe motive. Unul dintre ele este pasiunea pentru teoria grafurilor apărută încă de la orele de **Algoritmica Grafurilor** și a continuat să crească datorită materiei **Programare Avansată** la care a trebuit să punem în practică

acele concepte teoretice studiate anterior.

În plus, tema izomorfismului a venit din dorința și curiozitatea de a afla cât mai multe despre acest subiect care din cauza timpului prea scurt nu poate fi studiat suficient de detaliat. Ce a început ca un mic proiect pentru problema izomorfismului între **arbori** a ajuns la o analiză generală, complexă a acestei probleme.

De asemenea, faptul că **izomorfismul grafurilor** este una din puținele probleme *deschise*, pentru care nu a fost demonstrată apartenența la clasa de complexitate **NP-completă** sau **P**, a stârnit un interes puternic în a afla ce soluții teoretice și practice au fost realizate până acum. Totodată m-a determinat să examinez în detaliu câteva astfel de soluții și să realizez implementări pentru o bibliotecă de grafuri open-source pe care mai apoi să le compar, din punct de vedere al performanței, cu implementări similare din alte biblioteci foarte cunoscute.

Introducere

Prin această lucrare îmi propun să realizez o analiză teoretică și practică pentru problema izomorfismului grafurilor, cu accent pe aspectul algoritmic al problemei și performanța acestor algoritmi. Lucrarea își propune următoarele obiective:

Capitolul 1. Definiții și notații prezintă concepte fundamentale din teoria grafurilor și a noțiunilor legate de problema izomorfismului, pentru a înțelege în profunzime complexitatea ei.

Capitolul 2. Algoritmi existenți descrie conceptele teoretice și soluțiile propuse până în acest moment, cu accent pe cele mai folosite în practică.

Capitolul 3. Implementare introduce platforma de dezvoltare și biblioteca de grafuri utilizată, apoi prezintă în detaliu fiecare algoritm implementat.

Capitolul 4. Testarea performanțelor analizează performanțele algoritmilor pe diferite clase de grafuri și realizează o comparație cu implementări din alte biblioteci cunoscute.

Capitolul 5. Interfață testare algoritmi descrie dezvoltarea unui editor, vizualizor de grafuri și în final a unei interfețe care permite testarea izomorfismului grafurilor într-un mod cât se poate de intuitiv.

În **Concluzii** voi sublinia contribuțiile aduse, rezultatele obținute, îmbunătățiri pe care le-aș putea aduce implementărilor curente și direcții de cercetare viitoare.

Capitolul 1

Definiții și notații

În acest capitol vom menționa doar definiții și notații specifice problemei în cauză, considerând cunoscute noțiunile fundamentale din teoria grafurilor (ex.: graf orientat, graf neorientat, subgraf indus, arbore, pădure), precum și cele mai uzuale notații (ex.: **ordinul** grafului $G = (V, E)$ este notat $|G| = |V(G)|$).

Mai întâi voi începe cu definirea problemei pe care se bazează această lucrare.

Definiție 1. Două grafuri G_1 și G_2 de același ordin sunt **izomorfe** dacă există o funcție bijectivă $f : V(G_1) \rightarrow V(G_2)$, numită și **mapare**, astfel încât: $\{u, v\} \in E(G_1)$ dacă și numai dacă $\{f(u), f(v)\} \in E(G_2)$. (Fig. 2.1)

Definiție 2. Problema izomorfismului pe grafuri: sunt două grafuri G_1 și G_2 izomorfe?

Definiție 3. Problema izomorfismului pe subgraf indus: un graf G_1 este izomorf cu un subgraf indus din graful G_2 ?

Ambele probleme aparțin clasei de complexitate NP, deoarece putem verifica în timp polinomial dacă o mapare este într-adevăr izomorfă.

Problema izomorfismului pe **subgraf indus** este **NP-completă**: aparține NP și există o problemă NP-hard, anume CLIQUE(G, k), care se reduce la problema izomorfismului pe subgraf indus.

Definiție 4. CLIQUE(G, k) este o problemă decizională în care se determină existența unei cliki de dimensiune k în graful G .

În schimb, pentru problema izomorfismului pe grafuri încă nu se cunoaște clasa ei de complexitate. Se crede însă că ea ar fi într-o clasă NP-intermediară foarte apropiată de P [1], în ipoteza în care $P \neq NP$ (o altă problemă **deschisă**, una dintre cele mai faimoase din domeniul informaticii).

În încercarea de a afla cât mai multe despre această problemă, cercetătorii au propus definirea unei noi clase de complexitate, numită **GI** (Graph Isomorphism), ce include probleme care pot fi reduse în timp polinomial la problema izomorfismului pe grafuri. Similar clasei **NP**, s-au definit clasele de probleme **GI-hard** și **GI-complete**. Dintre aceste probleme vom menționa câteva **GI-complete**: problema numărării, respectiv determinării, tuturor mapărilor izomorfe între două grafuri, problema deciderii dacă un graf este autocomplementar, problema determinării izomorfismului între automate finite.

În ciuda eforturilor, progresul rămâne lent. Au fost descoperite diferite clase de grafuri pentru care există algoritmi de complexitate timp polinomial, precum arbori, grafuri planare sau grafuri de grad limitat. În cazul general, cea mai bună soluție teoretică are o complexitate timp cvasipolinomială: timpul de execuție crește mai rapid decât orice funcție polinomială, dar mai încet decât o funcție exponențială.

În capitolele ce urmează se vor descrie algoritmi care tratează atât problema decizională cât și problema determinării mapărilor în cazul existenței izomorfismului între cele două grafuri.

Următoarele noțiuni sunt importante pentru algoritmul liniar al izomorfismului între arbori.

Definiție 5. *Excentricitatea* unui nod v este distanța maximă de la v la oricare alt nod din graf, unde distanța este numărul de muchii de pe cel mai scurt drum între noduri.

$$excentricitate(v) = \max_{u \in V} d(v, u)$$

Definiție 6. *Raza* unui graf este excentricitatea minimă a unui nod.

Diametrul unui graf este excentricitatea maximă a unui nod.

Definiție 7. Un nod este numit **centroid** al unui graf dacă are excentricitatea egală cu raza grafului.

Într-un arbore, dar chiar și într-un graf neorientat conex, există o relație interesantă între rază și diametru.

$$raza = \left\lfloor \frac{diametru}{2} \right\rfloor + (diametru \bmod 2)$$

Prin urmare putem spune că centroidul unui arbore se află în mijlocul celui mai lung drum din arbore.

Însă datorită proprietăților sale, aciclicitate și conexitate, un arbore nu poate avea mai multe drumuri **disjuncte** de lungime maximă. Altfel spus, într-un arbore, orice două drumuri de lungime maximă se intersectează cel puțin în centroidul arborelui (unul -Fig. 1.1- sau 2 centroizi -Fig. 1.2- **unici** în funcție de paritatea diametrului).

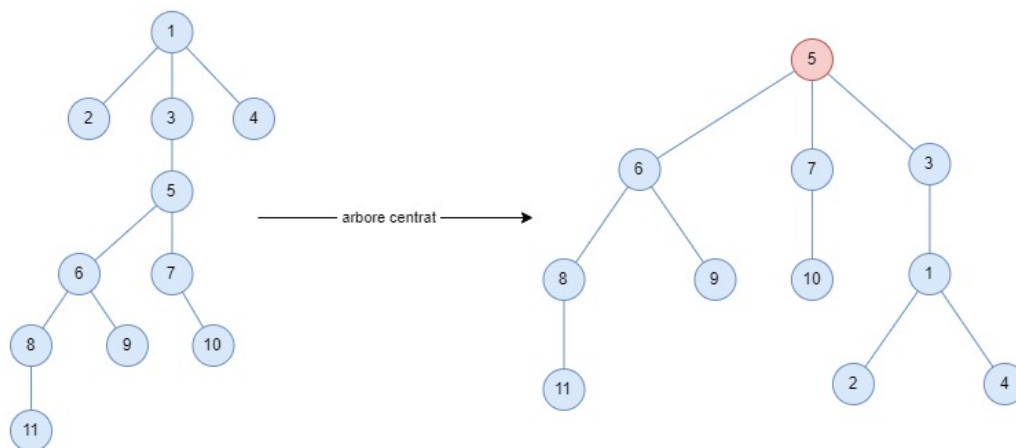


Figura 1.1: Arbore cu un singur centroid

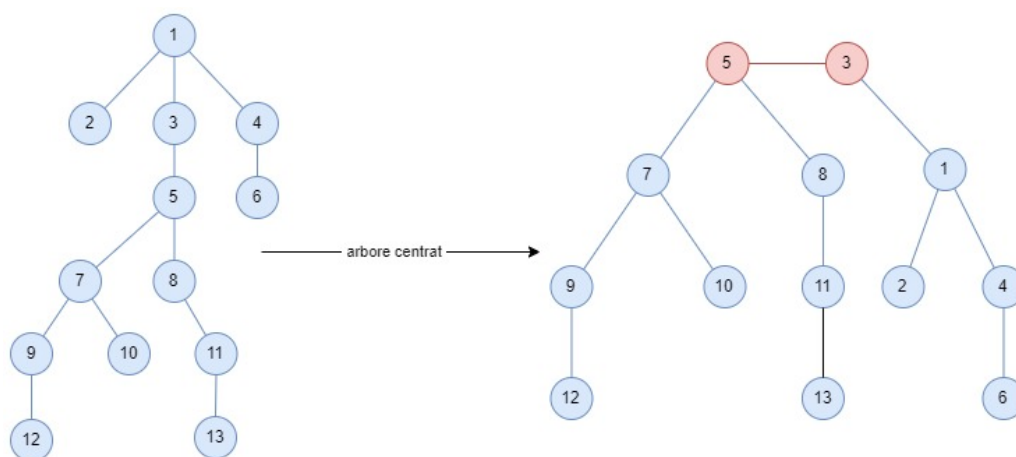


Figura 1.2: Arbore cu doi centroizi

Capitolul 2

Algoritmi existenți

În acest capitol voi descrie o serie de algoritmi propuși pentru rezolvarea izomorfismului grafurilor, în ordinea cronologică publicării acestora pentru a arăta evoluția lor.

Primul algoritm propus a fost cel al lui **Ullman**, inspirat din tehnici folosite în domeniul inteligenței artificiale. Acesta tratează problema izomorfismului prin căutarea soluției într-un spațiu de stări, folosind o strategie de căutare DFS (Depth First Search) cu un algoritm de tip backtracking (Fig. 2.1). Ulterior au fost propuși și alți algoritmi, precum **VF/VF2**, care abordează problema în același mod, însă care propun noi reguli și euristici pentru a reduce numărul de stări vizitate din arborele de căutare.

Vom vedea și alte abordări ale acestei probleme, precum reprezentarea canonică în cazul algoritmului **Nauty** sau modelul teoretic al lui **László Babai** bazat pe teoria grupurilor. Vom analiza mai târziu și o clasă specială de grafuri, anume arbori, pentru care s-a descoperit algoritm de complexitate timp **liniară** pentru problema izomorfismului.

Vom intra în detalii doar în cazul algoritmilor **Ullman**, **VF2** și cel pentru izomorfismul între arbori, aceștia fiind implementați personal într-o bibliotecă de grafuri open-source.

În cazul algoritmilor Ullman și VF2, vom face referire doar la grafuri orientate, generalizarea la grafuri neorientate fiind trivială. De altfel, vom folosi convenția: G_1 și G_2 sunt grafuri orientate de același ordin n și aceeași dimensiune m .

În cazul algoritmului pentru arbori, acesta tratează doar arbori neorientați fără muchii multiple și fără bucle.

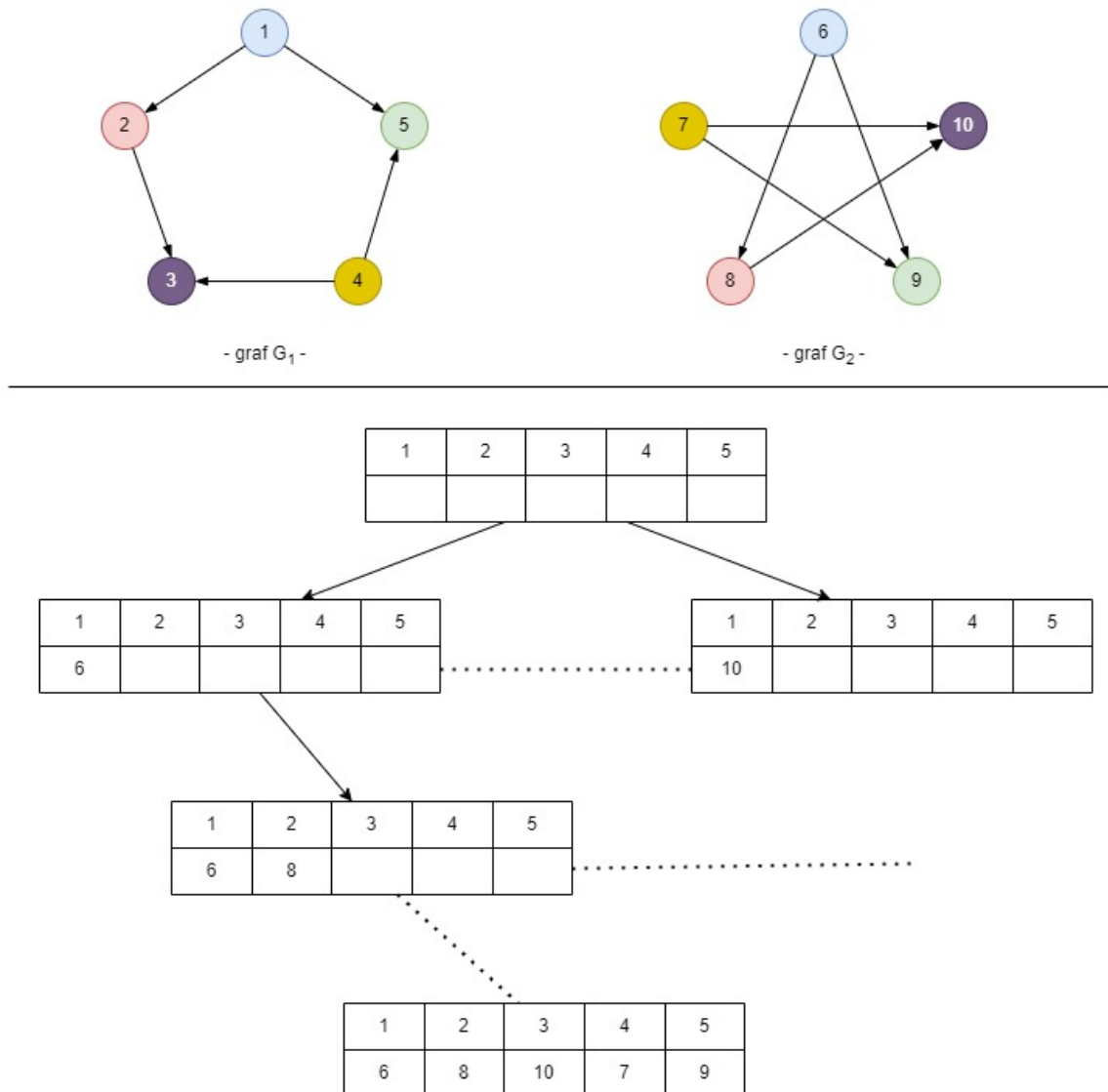


Figura 2.1: Arborele de căutare al soluției problemei izomorfismului pentru grafurile exemplu G_1 și G_2

2.1 Ullman

Publicat în anul 1976 [2], algoritmul lui **Ullman** introduce o primă îmbunătățire a algoritmului de enumerare exhaustivă, care reduce substanțial spațiul de căutare.

Notăție 1. Notăm cu $M = \{(i, j) | i, j \in \{0, \dots, n-1\}\}$ maparea, eventual parțială, a nodurilor grafului G_1 la nodurile grafului G_2 , unde i este indicele unui nod din G_1 , iar j este indicele unui nod din G_2 .

Din acest moment ne vom referi la nodurile grafurilor doar prin indici.

Notăție 2. Notăm cu $M_1 = \{i | i \in \{0, \dots, n-1\} \text{ și } \exists (i, j) \in M\}$, adică nodurile mapate ale grafului G_1 . Similar, notăm cu M_2 nodurile mapate ale grafului G_2 .

Atât algoritmul lui Ullman cât și VF2 abordează modelul bazat pe stări.

Notăție 3. $G_1(s)$ subgraful indus de nodurile din $M_1(s)$. Similar, notăm cu $G_2(s)$ subgraful indus de nodurile din $M_2(s)$, unde s este starea curentă.

O stare s este reprezentată de o mapare, eventual parțială, **consistentă** cu problema izomorfismului: subgrafurile induse $G_1(s)$ și $G_2(s)$ sunt izomorfe.

Consistența mapărilor este asigurată de o matrice binară de *compatibilitate*, notată m , între nodurile celor două grafuri, G_1 și G_2 , de dimensiune $n \times n$. Această matrice este inițializată în felul următor:

$$m_{(i,j)} = \begin{cases} 1 & \text{dacă nodul } i \in G_1 \text{ și nodul } j \in G_2 \text{ au gradul intern/extern egal} \\ 0 & \text{altfel} \end{cases}$$

Explorarea spațiului de căutare se face în manieră DFS prin adăugarea unei perechi de noduri candidat (i, j) , cu $i \in G_1$ și $j \in G_2$, la maparea stării curente.

După adăugarea unei noi perechi candidat (i, j) , se modifică matricea de compatibilitate astfel:

1. nodul i al lui G_1 va fi compatibil doar cu nodul j al lui G_2 , deci linia i a matricii m va avea doar valori 0, în afară de coloana j .
2. procesul de **rafinare** a candidaților: pentru un nod $i \in V(G_1)$ numim **candidat** orice nod $j \in V(G_2)$ compatibil, adică $m_{(i,j)} = 1$.

$$m_{(i,j)} \leftarrow 0 \quad \text{dacă} \quad \exists x \in N^{+/-}(i) \text{ a.i. } \forall y \in N^{+/-}(j) : m_{(x,y)} = 0$$

Tradus în cuvinte, două noduri (i, j) , cu $i \in V(G_1)$ și $j \in V(G_2)$, devin incompatibile dacă există cel puțin un succesori/predecesor al lui i care nu are niciun candidat în graful G_2 succesori/predecesor al lui j .

În continuare voi prezenta pe scurt algoritmul de tip backtracking pentru explorarea stărilor, pe care se bazează atât Ullman cât și VF2.

Algorithm 1: Algoritmul de căutare Ullman

Input : o stare parțială s (starea inițială s_0 are $M(s_0) = \emptyset$)

Output: maparea izomorfă dintre grafuri dacă există

```

1 Procedure match( $s$ ):
2   if  $M(s)$  este mapare completă then
3     return  $M(s)$ ;
4   Calculează mulțimea  $P(s)$  a perechilor candidat pentru a fi incluse în maparea  $M(s)$ ;
5   foreach  $p$  in  $P(s)$  do
6     Creează o nouă stare  $s'$  identică cu  $s$ ;
7     Adaugă perechea  $p$  la maparea  $M(s')$ ;
8     match( $s'$ );

```

Perechile candidat (i, j) sunt calculate în felul următor:

- i este următorul nod nemapat din G_1 , adică cel aflat la indicele $|M(s)|$;

Exemplu: dacă $M_1(s) = \emptyset$, atunci $i \leftarrow 0$. Dacă $M_1(s) = \{0\}$, atunci $i \leftarrow 1$.

- j este un nod nemapat din G_2 compatibil cu i , adică are loc $m_{(i,j)} = 1$;

Procedura de rafinare a candidaților se aplică stării s' imediat după ce a fost adăugată perechea candidat p la maparea parțială $M(s')$.

Algorithm 2: Algoritmul de rafinare

Input: Matricea de compatibilitate m , starea curentă s

```

1 Procedure refine( $s$ ):
2   for  $i$  in  $G_1$  do
3     for  $j$  in  $G_2$  do
4       if  $i$  not in  $M(s)$  AND  $j$  not in  $M(s)$  AND  $m_{(i,j)} = 1$  then
5         Pentru fiecare succesori(predecesor) al lui  $i$ ;
6         for  $x$  in  $N^{+/-}(i)$  do
7           if nu există  $y$  in  $N^{+/-}(j)$  astfel încât  $m_{(x,y)} = 1$  then
8              $m_{(i,j)} \leftarrow 0$ ;
9           goto 3;

```

Complexitate timp

Operațiile realizate asupra unei stări au complexitate $\Theta(n^2)$:

- calculul unei perechi candidat: complexitate timp în cazul nefavorabil $\Theta(n)$ atunci când parcurgem toate nodurile $j \in G_2$ pentru a găsi unul compatibil cu $i \in G_1$.
- adăugarea unei perechi candidat la o stare curentă: complexitate timp $\Theta(n)$.
- procedura de rafinare a candidaților: complexitate timp $\Theta(n^2)$.

Complexitatea timp a întregului algoritm de căutare:

- cazul favorabil: pentru fiecare stare există exact o pereche candidat care va duce într-un final către soluție. În acest caz, numărul stărilor vizitate este n , iar complexitatea timp a algoritmului este: $\Theta(n \cdot n^2) = \Theta(n^3)$.
- cazul nefavorabil: procedeul de rafinare nu împiedică vizitarea tuturor stărilor parțiale înainte de a găsi soluția. Acest fenomen poate apărea în cadrul grafurilor aproape complete.

În acest caz, numărul de stări vizitate poate fi calculat astfel:

- pe nivelul 0 al arborelui, avem o singură stare, starea inițială cu maparea vidă.
- pe nivelul 1 al arborelui, avem n stări descendente din starea inițială.
- pe nivelul 2 al arborelui, pentru fiecare stare de pe nivelul 1, avem $n - 1$ stări descendente, deci în total $n \cdot (n - 1)$ stări.
- ...

În total, avem n nivele:

$$1 + n + n \cdot (n - 1) + \dots + n \cdot (n - 1) \cdot \dots \cdot 2 \cdot 1 = 1 + \frac{n!}{(n - 1)!} + \frac{n!}{(n - 2)!} + \dots + \frac{n!}{1!} = 1 + n! \cdot \sum_{i=1}^{n-1} \frac{1}{i!}$$

Suma $\sum_{i=1}^{n-1} \frac{1}{i!}$ reprezintă un faimos șir Maclaurin [3], anume

$$\sum_{i=0}^{\infty} \frac{1}{i!} \cdot x^i = e^x, \text{ în care } x = 1.$$

Prin urmare, numărul de stări vizitate este proporțional cu $n!$, ducând algoritmul spre o complexitate: $\Theta(n! \cdot n^2)$.

Complexitate spațiu

O stare în algoritmul lui Ullman ocupă un spațiu $\Theta(n^2)$: matricea de compatibilitate ocupă $\Theta(n^2)$, maparea nodurilor ocupă $\Theta(n)$, plus un factor constant. Datorită strategiei DFS în memorie vor fi salvate cel mult n stări la un moment dat. Deci complexitatea spațiu a algoritmului este $O(n \cdot n^2) = O(n^3)$.

2.2 VF2

VF2 [4] este un alt algoritm de testare al izomorfismului bazat pe modelul arborelui de căutare. Acesta este o versiune îmbunătățită a algoritmului VF(Vento - Foggia)[5] în ceea ce privește memoria folosită.

Pentru a reduce numărul stărilor vizitate, VF2 propune o serie de reguli pentru a "tăia" (eng. **prune**) în avans ramurile care nu duc spre izomorfism.

Notăție 4. $T_1in(s)$ = mulțimea nodurilor nemapate din G_1 care 'intră'(predecesori) în subgraful indus $G_1(s)$. Similar, $T_2in(s)$ pentru graful G_2 .

Notăție 5. $T_1out(s)$ = mulțimea nodurilor nemapate din G_1 care 'ies'(succesori) din subgraful indus $G_1(s)$. Similar, $T_2out(s)$ pentru graful G_2 .

Notăție 6. $T_1(s) = T_1in(s) \cup T_1out(s)$. Similar, $T_2(s) = T_2in(s) \cup T_2out(s)$.

Ne vom referi la aceste mulțimi prin termenul de mulțimi **terminale**.

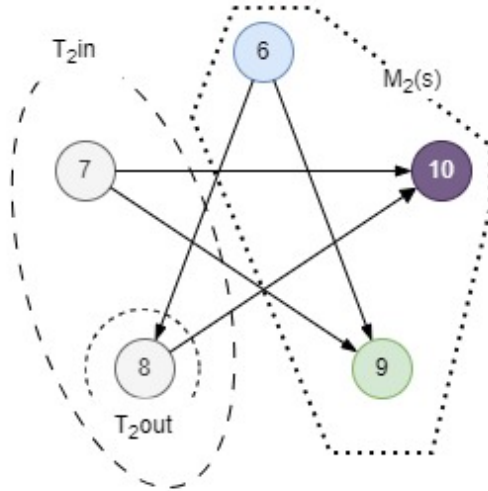


Figura 2.2: Maparea parțială grafului G_2 din (Fig. 2.1) împreună cu mulțimile T_2in și T_2out

Structura algoritmului este asemănătoare cu cea a lui Ullman, însă cu modificări în ceea ce privește calculul perechilor candidat și introducerea regulilor de fezabilitate.

Perechile candidat (i, j) sunt calculate astfel:

- dacă $T_1in(s) \cap T_1out(s) \neq \emptyset$ și $T_2in(s) \cap T_2out(s) \neq \emptyset$, atunci alegem o pereche (i, j) cu $i \in T_1in(s) \cap T_1out(s)$ și $j \in T_2in(s) \cap T_2out(s)$;

- altfel, dacă $T_{1out}(s) \neq \emptyset$ și $T_{2out}(s) \neq \emptyset$, se alege ca pereche candidat (i, j) cu $i \in T_{1out}(s)$ și $j \in T_{2out}(s)$
- altfel, dacă $T_{1in}(s) \neq \emptyset$ și $T_{2in}(s) \neq \emptyset$, se alege ca pereche candidat (i, j) cu $i \in T_{1in}(s)$ și $j \in T_{2in}(s)$
- altfel, se alege perechea (i, j) cu $i \in V(G_1) \setminus M_1(s)$ și $j \in V(G_2) \setminus M_2(s)$, adică noduri nemapate.

Regulile de filtrare a unei perechi candidat (i, j) , sunt:

- R_PRED: pentru orice predecesor x al lui i care este mapat la un nod y , trebuie ca y să fie predecesorul lui j ;
- R_SUCC: pentru orice succesor x al lui i care este mapat la un nod y , trebuie ca y să fie succesorul lui j ;
- R_IN: numărul de succesori/predecesori ai lui i care se află în mulțimea $T_{1in}(s)$ trebuie să fie egal cu numărul de succesori/predecesori ai lui j care se află în mulțimea $T_{2in}(s)$;
- R_OUT: numărul de succesori/predecesori ai lui i care se află în mulțimea $T_{1out}(s)$ trebuie să fie egal cu numărul de succesori/predecesori ai lui j care se află în mulțimea $T_{2out}(s)$;
- R_NEW: numărul de succesori/predecesori ai lui i care nu sunt nici în $T_1(s)$ și nici în $M_1(s)$ (noduri **noi**) trebuie să fie egal cu numărul de succesori/predecesori ai lui j care nu sunt nici în $T_2(s)$ și nici în $M_2(s)$.

Primele două reguli, R_PRED și R_SUCC, asigură **consistența** mapărilor parțiale în sens izomorf. Celelalte trei reguli, numite și reguli de **fezabilitate**, verifică în avans dacă adăugarea perechii candidat nu va duce către izomorfism, reușind astfel să reducem spațiul de căutare.

Pe lângă regulile de consistență și fezabilitate, trebuie să se respecte pentru orice stare următoarele condiții:

$$\begin{aligned}
 |T_{1in}(s)| &= |T_{2in}(s)|, \quad |T_{1out}(s)| = |T_{2out}(s)| \\
 \text{și } |T_{1in}(s) \cap T_{1out}(s)| &= |T_{2in}(s) \cap T_{2out}(s)|
 \end{aligned}
 \tag{2.1}$$

Mulțimile terminale, $T_{1in/out}(s)$, $T_{2in/out}(s)$, nu sunt calculate de fiecare dată la crearea unei stări noi, ci se folosesc structuri de date pentru a se marca incremental

nodurile. La adăugarea unei noi perechi candidat (i, j) se vor marca vecinii lor ca fiind drept succesori/out sau predecesori/in.

Algoritmul predecesor lui VF2, algoritmul VF, face această marcă prin vectori de flag-uri: **flags_1[]**, **flags_2[]**, de dimensiune n , corespunzătoare nodurilor celor două grafuri. Această reprezentare aduce un dezavantaj în ceea ce privește memoria: fiecare stare are neapărat propria zonă de memorie pentru vectorii de flag-uri.

VF2 aduce o îmbunătățire în acest sens. El propune un nou mod de a marca apartenența nodurilor la mulțimile terminale, prin patru vectori de întregi: **in_1[]**, **out_1[]**, **in_2[]**, **out_2[]**, tot de dimensiune n .

La adăugarea unei noi perechi de noduri candidat, vecinii acestora vor fi marcați în vectorii corespunzători cu o valoare egală cu adâncimea stării curente în arborele de căutare. Inclusiv nodurile din perechea candidat vor fi marcate ca fiind în același timp nod **in** și **out**. Pentru memorarea mapării parțiale se vor folosi însă variabile separate.

Important este de reținut că odată marcat un nod ca aparținând unei mulțimi terminale, valoarea lui în vectorul de marcă nu va fi schimbată în stări descendente. Acest lucru ne permite partajarea între stări a aceleași zone de memorie pentru vectorii de marcă. Revenirea la starea anterioară adăugării ultimei perechi candidat se face prin identificarea vecinilor care au fost marcați în pasul anterior.

Așadar, pseudocodul algoritmului lui Ullman (algoritmul 1) poate fi adaptat pentru algoritmul VF2 prin următoarele: la linia 5 fiecare stare candidat trebuie verificată prin regulile de consistență și fezabilitate, iar la linia 8 după fiecare apel recursiv trebuie să restaurăm variabilele partajate, adică vectorii de marcă.

Complexitate timp

Operațiile realizate asupra unei stări au complexitate $\Theta(n)$:

- calculul unei perechi candidat: complexitate timp în cazul nefavorabil $\Theta(n)$.
- procedura de verificare a fezabilității unei perechi candidat: complexitate timp în cazul nefavorabil $\Theta(n)$, atunci când numărul de succesori/predecesori ai unui nod din pereche este foarte aproape de n (grafuri dense).
- adăugarea unei perechi candidat la o stare curentă: complexitate timp $\Theta(n)$, deoarece presupune revizitarea succesorilor/predecesorilor pentru a le fi marcată apartenența la mulțimile terminale.

- procedura de **backtrack** manual: restaurarea variabilelor partajate între stări pentru a reveni la starea anterior adăugării ultimei perechi candidat. Are complexitate $\Theta(n)$, din același motiv al vizitării vecinilor.

Complexitatea timp a întregului algoritm de căutare:

- cazul favorabil: pentru fiecare stare există o singură pereche candidat fezabilă care va duce spre soluție. În acest caz numărul stărilor vizitate este n , iar complexitatea timp a algoritmului este: $\Theta(n \cdot n) = \Theta(n^2)$.
- cazul nefavorabil: fiecare pereche candidat este fezabilă, astfel nu putem evita vizitarea tuturor stărilor parțiale până la găsirea soluției. Din nou, aceste cazuri pot apărea în cadrul grafurilor aproape complete. La fel ca la algoritmul lui Ullman, numărul de stări vizitate este proporțional cu $n!$, ducând algoritmul spre o complexitate timp: $\Theta(n! \cdot n)$.

Complexitate spațiu

O stare în algoritmul VF2 ocupă un spațiu $\Theta(n)$: vectorii de marcare ocupă $\Theta(n)$, maparea nodurilor ocupă $\Theta(n)$, plus un factor constant. Datorită strategiei DFS în memorie vor fi salvate cel mult n stări la un moment dat. Deoarece atât vectorii de marcare cât și maparea nodurilor sunt partajate între stări, complexitatea spațiu a algoritmului este $\Theta(n + n \cdot c) = \Theta(n)$, unde c este acel factor constant de memorie.

2.3 Alți algoritmi

Deși VF2 rămâne unul dintre cei mai folosiți algoritmi pentru rezolvarea izomorfismului grafurilor, acesta prezintă o serie de dezavantaje și limitări, care au fost adresate ulterior în cadrul algoritmilor: VF2Plus [6], VF2++ [7], VF3 [8] și VF3-light [9]. La fel ca VF2, toți acești algoritmi abordează problema izomorfismului prin modelul bazat pe stări.

Dezavantajele majore ale algoritmului VF2 sunt:

- lipsa unei ordonări a nodurilor grafului G_1 : alegerea nodului următor se face neținând cont de anumite criterii care să reducă spațiul de căutare.
- structura mulțimilor terminale: în cazul grafurilor mari, mulțimile terminale pot deveni și ele foarte mari chiar dacă numărul de perechi candidat fezabile poate

fi considerabil mai mic. În acest caz, algoritmul poate explora multe stări inutile care vor aduce o penalizare a timpului de execuție.

Primul algoritm care propune soluții pentru aceste probleme este **VF2Plus**. Acesta propune o procedură de sortare a nodurilor grafului G_1 bazată pe un model probabilistic. Așadar, nodurile cu prioritate mai mare sunt cele cu probabilitatea cea mai mică de a găsi un candidat în graful G_2 și cu cel mai mare număr de vecini deja sortați.

Odată stabilită această ordine, vom ști la fiecare pas care nod din G_1 să alegem pentru a-l mapa cu un nod din G_2 . Totodată, se pot calcula mulțimile terminale ale subgrafului indus $G_1(s)$ înainte de a începe procesul de căutare. În acest fel, explorarea spațiului se face într-o manieră sistematică, mai eficientă, reducând numărul de stări vizitate.

Problema mulțimilor terminale este tratată în felul următor: nodul candidat se caută mai întâi în vecinătatea nodurilor deja mapate. În plus, înainte de a începe explorarea se realizează o clasificare a nodurilor în funcție de anumite criterii (ex.: eticheta nodului), astfel încât mulțimile terminale să fie divizate în submulțimi mai mici și astfel regulile de fezabilitate devin mai puternice.

Propus de alți autori, **VF2++** tratează aceleași probleme ale lui **VF2**. De această dată, nodurile grafului G_1 sunt sortate în funcție de grad și raritatea etichetei. Astfel, nodurile prioritare au cel mai mare număr de vecini deja sortați și au cea mai rară etichetă. În plus, autorii propun o serie de reguli de fezabilitate mai puternice, care iau în considerare atât nodurile din mulțimile terminale definite în cadrul algoritmului VF2, cât și nodurile din mulțimile complementare (ex.: $\tilde{T}_1(s) = (V_1 \setminus M_1(s)) \setminus T_1(s)$, adică noduri nemapate din afara mulțimii terminale $T_1(s)$).

Ulterior a apărut succesorul algoritmului VF2Plus, anume **VF3**, o versiune optimizată pentru grafuri mari și dense. Acesta propune o serie de preprocesări complexe și un set de reguli de fezabilitate mai avansate pentru a reduce spațiul de căutare.

În final, **VF3-light** este o versiune simplificată a algoritmului VF3 care funcționează mai bine pe grafuri de dimensiuni mici și medii, prin eliminarea unor reguli de fezabilitate complexe.

2.3.1 Nauty

Algoritmul Nauty [10] este considerat unul dintre cei mai rapizi algoritmi practici pentru rezolvarea izomorfismului grafurilor, deși există exemple de grafuri pentru care

timpul de execuție este exponențial.

Algoritmul este folosit ca o subprocedură pentru a calcula o etichetare canonică a unui graf, numită forma canonică a grafului. În același timp, algoritmul calculează și grupul de automorfisme al grafului.

Odată calculată această formă canonică, testarea izomorfismului este trivială, deoarece două grafuri sunt izomorfe dacă și numai dacă formele canonice sunt identice.

Procedura de calcul a formei canonice se realizează într-o manieră backtracking în care o stare reprezintă o partiționare ordonată a nodurilor grafului. În rădăcina arborelui de căutare se află partiția inițială a nodurilor grafului, în care o clasă conține noduri cu același grad. Pornind de la această partiție, începem un proces de rafinare echitabilă a partițiilor, adică nodurile din aceeași clasă au același număr de vecini în fiecare altă clasă. Această rafinare se face până când nu mai putem împărți nodurile în clase mai mici.

Frunzele arborelui de căutare reprezintă potențiale forme canonice ale grafului. Automorfismele grafului sunt determinate de 2 astfel de forme care definesc același graf, graf izomorf cu graful inițial. Aceste automorfisme sunt folosite pentru a face pruning în arborele de căutare.

Dintre toate aceste forme se va alege una singură, conform unor proceduri avansate, care va reprezenta forma canonică a grafului.

2.3.2 Babai

Modelul teoretic propus de **László Babai** [11] pentru rezolvarea izomorfismului grafurilor introduce cea mai bună complexitate timp cunoscută pentru această problemă, anume o complexitate quasipolinomială: $\exp((\log n)^{O(1)})$.

Acesta folosește idei din algoritmul lui **Luks**, care propune o soluție polinomială a problemei izomorfismului pentru grafuri de grad mărginit [12].

Algoritmul lui Babai implică o serie de tehnici și rezultate teoretice avansate din domeniul algebrei grupurilor și combinatoricii. De asemenea, el folosește tehnici avansate de partiționare a grafurilor pentru a trata subgrafuri mai mici.

În ciuda eficienței teoretice a algoritmului, implementarea practică a acestuia este dificilă, în mare parte din cauza nivelului ridicat de cunoștințe din teoria grupurilor și combinatorică, necesare pentru a înțelege cum funcționează acesta. În plus, complexitatea teoretică a algoritmului poate ascunde constante mari care pot face algoritmul

neutilizabil în practică.

Din aceste motive, în practică se preferă algoritmi mai simpli, mai ușor de înțeles și de implementat, precum cei descriși anterior, care deși au complexitate exponențială în cazul nefavorabil, se dovedesc a fi suficient de eficienți în practică.

2.4 Algoritm liniar

În anul 1974 Aho, Hopcroft și Ullman publică o carte [13] ce surprinde diverse aspecte ale dezvoltării algoritmilor, precum sortarea, căutarea, explorarea grafurilor și multe altele. În cadrul acestei cărți, autorii prezintă și un algoritm liniar pentru testarea izomorfismului arborilor cu rădăcina cunoscută.

Algoritmul se bazează pe etichetarea nodurilor arborilor. Pașii algoritmului sunt:

- se împart nodurile arborilor pe niveluri, începând de la rădăcină cu nivelul 0; Dacă numărul de niveluri este diferit pentru cei doi arbori, atunci aceștia nu sunt izomorfi;
- se iau pe rând nodurile de pe același nivel, de la cel mai jos nivel până la rădăcină, pentru ambii arbori și se începe etichetarea;
- etichetarea nodurilor se face astfel (Fig. 2.3): se formează o listă cu etichetele descendenților direcți ai nodului curent și se sortează această listă.

Pentru fiecare astfel de listă vom asocia o valoare numerică unică, care va reprezenta eticheta nodului. Dacă lista nu are asociată o valoare numerică, se va asocia o nouă valoare prin incrementarea unui contor global.

Pentru nodurile frunză lista va fi vidă, iar eticheta nodului va fi numărul 0.

- după etichetarea tuturor nodurilor de pe același nivel în ambii arbori, se formează câte o listă cu etichetele acestor noduri abia etichetate. Aceste 2 liste se vor ordona și se vor compara. În caz de inegalitate, putem spune că arborii nu sunt izomorfi.
- se continuă procesul de etichetare pentru restul nivelurilor superioare.
- dacă în final nodurile rădăcină au aceeași etichetă, atunci arborii sunt izomorfi.

Intuitiv, eticheta unui nod reprezintă o formă de codificare a structurii subarborului cu rădăcina în acel nod.

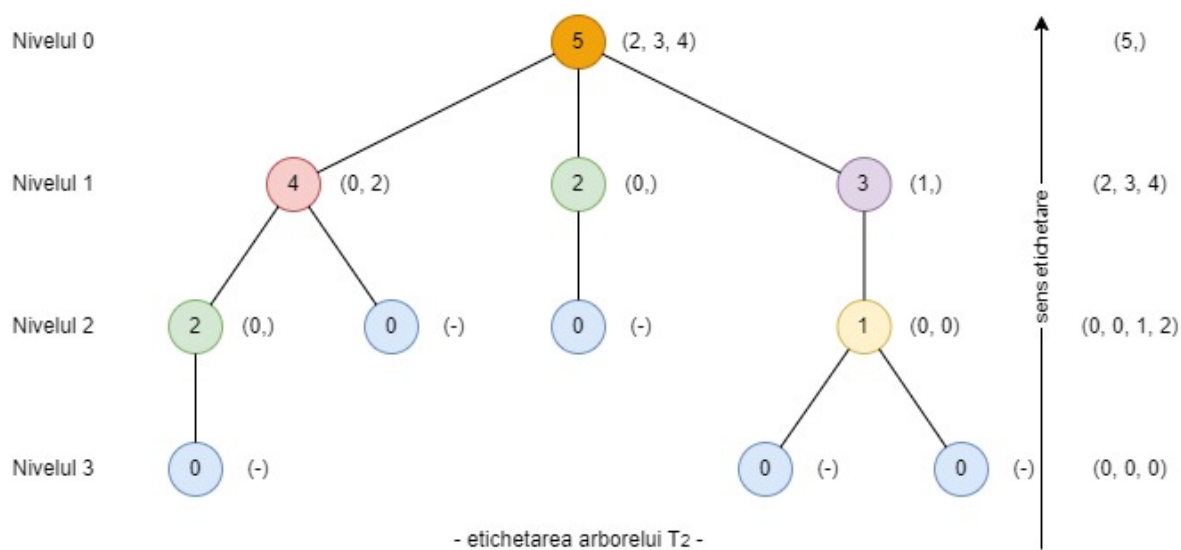
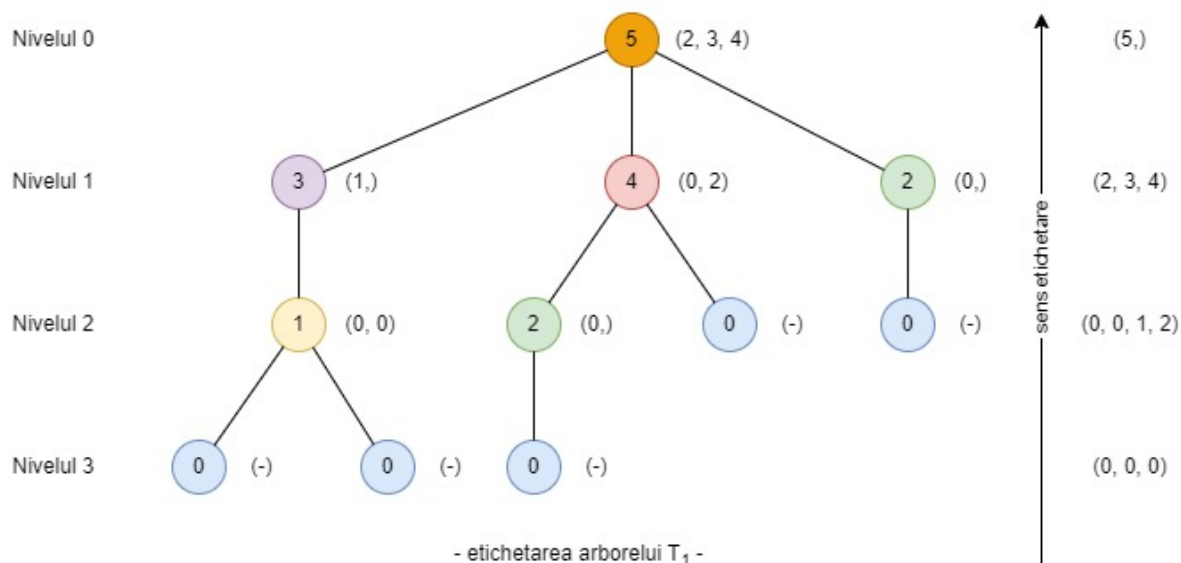


Figura 2.3: Etichetarea nodurilor arborilor T_1 și T_2 ; în interiorul nodului se află eticheta calculată, iar în dreapta nodului se află lista de etichete a descendenților direcți pe baza căreia s-a calculat eticheta; în dreapta arborilor pentru fiecare nivel se află lista sortată de etichete a nodurilor de pe acel nivel. Se poate observa că arborii T_1 și T_2 sunt izomorfi.

Pentru sortarea listelor de etichete se folosește algoritmul de sortare **RadixSort**, care are complexitate liniară $O(m \cdot k)$, unde m este numărul de elemente din listă, iar k este numărul maxim de cifre al unui element. Valoarea lui k poate fi considerată constantă: numărul de cifre al celui mai mare întreg reprezentabil pe o anumită platformă de programare, deci complexitatea sortării este liniară. Deoarece un nod al arborelui poate fi sortat de cel mult 2 ori, o dată ca descendent direct și altă dată ca nod abia etichetat, putem considera că sortarea consumă în total un timp liniar $O(n)$, unde n este numărul de noduri al arborelui.

Explorarea arborelui are o complexitate timp $O(n)$, deci complexitatea timp totală a întregului algoritm este liniară $O(n + n) = O(n)$.

Capitolul 3

Implementare

În această secțiune vom descrie detaliile tehnice ale implementării algoritmilor pentru testarea izomorfismului, dar vom argumenta și alegerile în ceea ce privește platforma de programare cât și biblioteca de grafuri folosită.

3.1 Platforma de programare Java

Lansat în anul 1996, limbajul de programare Java rezolvă una din cele mai mari probleme ale acelei perioade, anume **dependența** de platformă ori sistemul de operare. Pe lângă independența de platformă, el aduce o serie de alte avantaje importante care au contribuit la creșterea în popularitate, fiind chiar și în momentul de față (iunie 2024) unul dintre cele mai populare limbaje de programare (Fig. 3.1).






Jun 2024	Jun 2023	Change	Programming Language	Ratings	Change
1	1		 Python	15.39%	+2.93%
2	3	▲	 C++	10.03%	-1.33%
3	2	▼	 C	9.23%	-3.14%
4	4		 Java	8.40%	-2.88%
5	5		 C#	6.65%	-0.06%

Figura 3.1: Top 5 limbaje de programare cele mai populare conform [14] în luna iunie a anului 2024

Limbajul de programare Java are următoarele avantaje [15]:

1. **independența** de platformă: compilatorul transformă codul sursă Java într-un cod intermediar numit **bytecode** care este apoi interpretat de către **JVM** (Java Virtual Machine) și translatat în cod mașină;
2. **portabilitate**: datorită compilării în bytecode și a existenței mașinilor virtuale Java pentru diferite sisteme de operare (Windows, Linux, macOS, Android, etc.), un dezvoltator software poate rula aplicația sa pe orice sistem de operare, aceasta fiind și ideologia de bază a platformei de programare Java: *"Write Once Run Anywhere"*;
3. **robustețe**: Java nu permite manipularea directă a memoriei. În schimb, **JVM** gestionează automat memoria folosită, folosind un așa-numit **Garbage Collector** care dealocă automat obiectele nefolosite. Prin acestea Java devine mult mai rezistentă la vulnerabilități legate de gestionarea memoriei (ex. în limbajul C/C++: problema scurgerii de informații prin alocarea de memorie dar care nu este niciodată dealocată)
4. **performanțe** ridicate: deși nu se poate compara cu performanțele limbajelor care compilează direct în cod nativ(ex.: C/C++), **Java** reușește datorită compilatorului **JIT** (Just In Time) să obțină performanțe considerabile. Cu ajutorul **JIT** compilarea în cod nativ se va face la runtime, iar din moment ce o metodă a fost compilată ea va fi apelată direct în cod nativ.

Pe lângă acestea, Java aduce și avantajul unei mari comunități de dezvoltatori software care la rândul lor contribuie la o multitudine de biblioteci open-source. Acesta este și cazul bibliotecii de grafuri **Graph4J** pe care am ales-o pentru implementarea algoritmilor de testare a izomorfismului, deoarece la rândul ei aduce niște beneficii cruciale.

3.2 Biblioteca Graph4J

Graph4J [16] este o bibliotecă de grafuri creată pentru platforma de programare Java. Există și alte astfel biblioteci de grafuri: JGraphT [17], JUNG, Google Guava Graph. Toate acestea abordează un model orientat pe obiect care impune penalități de performanță atât timp cât și spațiu.

Graph4J însă abordează grafurile într-un mod matematic, iar majoritatea tipurilor de date sunt primitive (ex.: `int` în schimbul `Integer`), având drept scop final minimizarea costurilor de memorie și timp. În urma experimentelor realizate, această bibliotecă s-a dovedit a fi foarte eficientă, în anumite cazuri mult mai performantă decât celelalte biblioteci menționate.

De exemplu în Graph4J un nod este reprezentat printr-un număr pozitiv unic, denumit de către autori drept "numărul/identificatorul nodului" (eng. **vertex number**). Acest lucru face ușoară memorarea nodurilor prin variabile de tip primitiv: vectori de întregi. Prin urmare, ne putem referi la un nod în două moduri: fie prin identificatorul său, fie prin indicele său în graf. În plus, dacă dorim să atribuim unui nod un obiect oarecare, se folosesc așa numitele etichete. Aceste etichete se pot atribui și muchiilor.

Având în vedere cele spuse, am ales biblioteca **Graph4J** din dorința de a implementa algoritmi cât mai performanți. Așa cum vom vedea mai târziu, într-adevăr modelul matematic și eficiența operațiilor de bază pe grafuri aduc un câștig considerabil de performanță în cazul algoritmilor implementați personal.

3.3 Arhitectura generală a algoritmilor

Așa cum am văzut în capitolul anterior, atât algoritmul lui Ullman cât și VF2 sunt algoritmi de tip backtracking, care explorează arborele de căutare folosind strategia DFS.

Cei doi algoritmi folosesc tehnici diferite de a reduce spațiul de căutare. În esență ei urmează aceeași pași cu diferențe în ceea ce privește implementările stărilor.

Înainte de a discuta despre stările algoritmilor voi introduce două clase importante:

1. clasa **OrderedDigraph**(Fig. 3.2): permite ordonarea nodurilor descrescător după grad pentru a îmbunătăți algoritmi de bază care nu impun o ordine a nodurilor. Permite și memorarea **opțională** a succesorilor, predecesorilor și muchiilor pentru a avea acces la ele cât mai rapid, în schimbul unei creșteri considerabile a memoriei.

Pe parcursul algoritmilor nodurile grafurilor vor fi referite doar prin indicii lor din lista ordonată: la indicele 0 se va afla nodul cu cel mai mare grad, iar la indicele $n - 1$ se va afla nodul cu cel mai mic grad. Singurul moment în care este

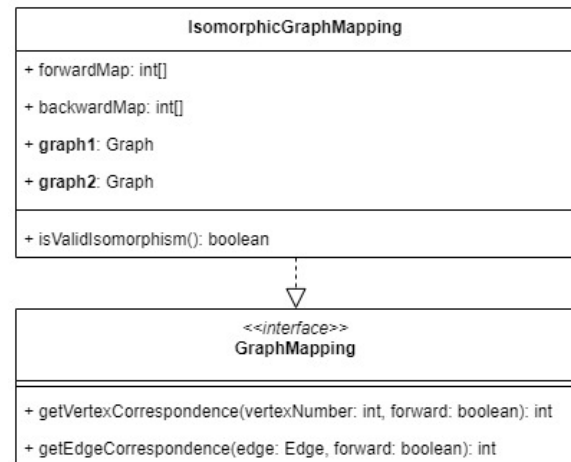
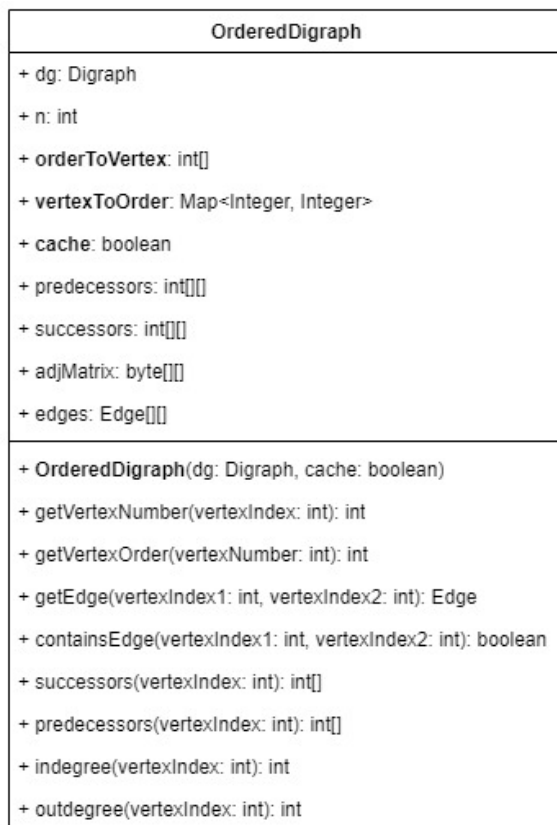


Figura 3.3: Clasa IsomorphicGraphMapping

Figura 3.2: Clasa OrderedDigraph

nevoie de identificatorul nodurilor este la afișarea soluției, caz în care se folosește metoda **getVertexNumber()**.

Metodele **getEdge()**, **successors()**, **predecessors()** vor apela metodele din clasa **Digraph** dacă variabila **cache** este **false**, altfel vor întoarce informația memorată la primul apel. Această memorare este utilă în cazul în care aceste metode sunt apelate într-un număr foarte mare, de exemplu în cazul grafurilor dense.

2. clasa **IsomorphicGraphMapping**(Fig. 3.3): modelează o soluție.

Variabilele **forwardMap** și **backwardMap** reprezintă maparea nodurilor grafului G_1 la nodurile grafului G_2 și invers. Am definit și o metodă pentru a verifica dacă o astfel de mapare este într-adevăr izomorfă, metodă folosită în partea de testare a corectitudinii algoritmilor.

Prin urmare, pentru stările algoritmilor am abordat următoarea arhitectură, ilustrată în figura (Fig. 3.4). Am inclus și clasele pentru problema izomorfismului pe subgrafuri induse, adaptările fiind triviale.

Metodele din interfața **State**:

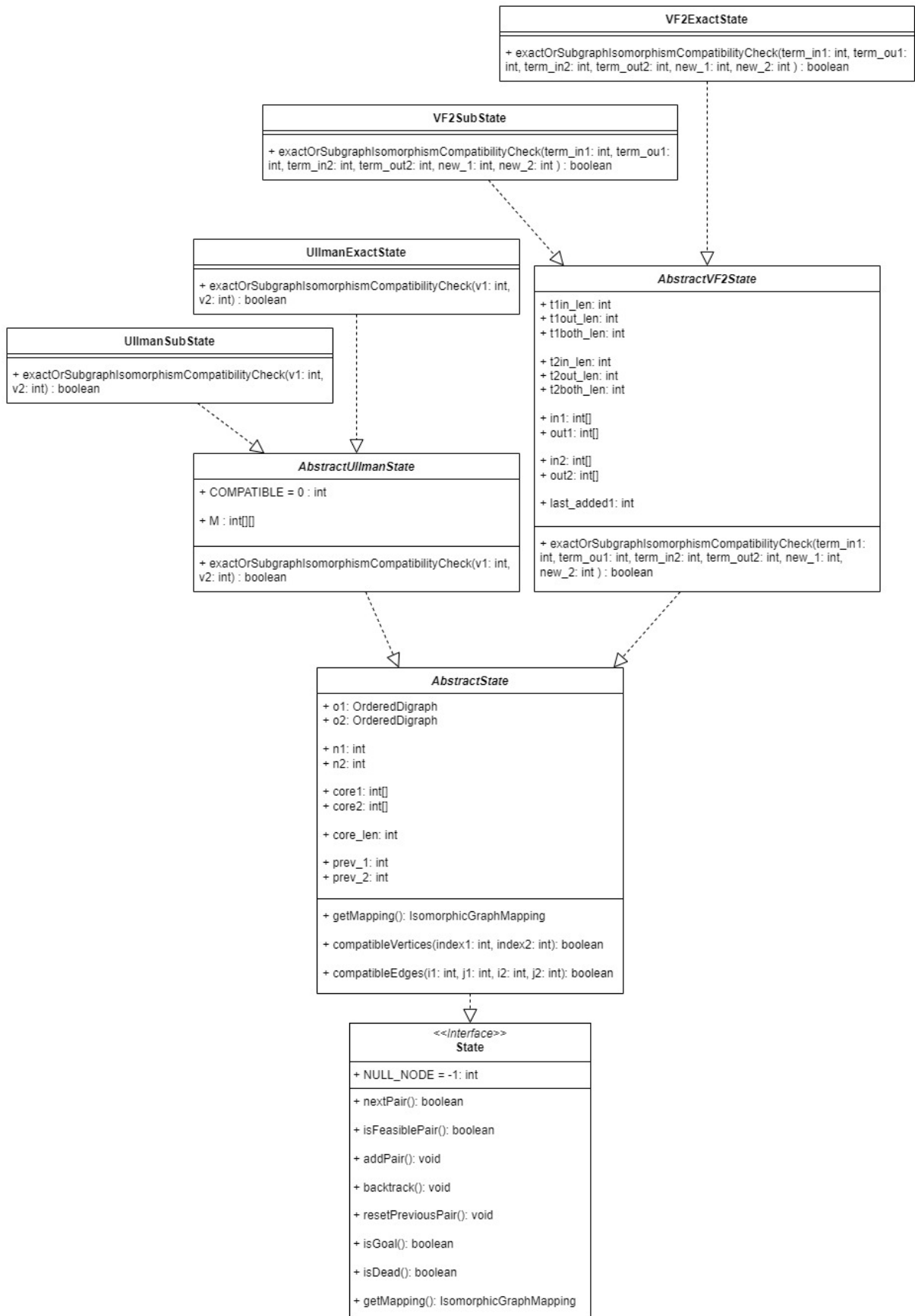


Figura 3.4: Diagrama de clasă a stărilor algoritmilor Ullman și VF2

1. metoda **nextPair()**: calculează următoarea pereche candidat (i, j) , cu $i \in G_1$ și $j \in G_2$ noduri nemapate;
2. metoda **isFeasiblePair()**: verifică dacă perechea candidat ar putea duce către o soluție;
3. metoda **addPair()**: adaugă la maparea curentă perechea candidat găsită.
4. metoda **backtrack()**: restaurează manual variabilele partajate între stări pentru a reveni la starea anterioară adăugării ultimei perechi candidat.
5. metoda **resetPreviousPair()**: folosit pentru a semnala că o stare este nouă, adică pentru care nu a fost calculată nicio pereche candidat.
6. metoda **isGoal()**: verifică dacă am ajuns la o mapare completă;
7. metoda **isDead()**: verifică dacă această stare este un blocaj, adică în mod cert nu putem ajunge la o soluție. Este diferită de regulile de fezabilitate care se referă la o pereche candidat;
8. metoda **getMapping()**: la găsirea unei mapări complete vom putea extrage soluția pentru a o procesa mai departe.

Algoritmul de explorare al spațiului de căutare este comun pentru cei doi algoritmi, Ullman și VF2 (Fig. 3.5). Metoda **match()** care face căutarea de tip backtracking este realizată într-o abordare iterativă în schimbul celei recursive, din mai multe motive [18]:

- eliminăm **overhead**-ul adăugării și eliminării pe stiva programului a stării sale. În schimb, vom simula modul în care programul își gestionează starea.
- eliminăm posibilitatea de a ajunge la limita stivei programului (eng. **stack overflow**). Acest fenomen este foarte întâlnit în cazul metodelor recursive în care factorul de ramificare este ridicat, ceea ce se poate întâmpla și în cazul nostru.

Metoda **match()** primește parametrul boolean **onlyFirst** care indică dacă se dorește găsirea unei singure soluții sau a tuturor. Această metodă folosește o stivă pentru a simula modul în care metoda recursivă echivalentă gestionează stiva programului.

Variabila s reprezintă starea curentă, inițial având maparea vidă întoarsă de metoda **getInitialStateInstance(dg1, dg2, cache)**.

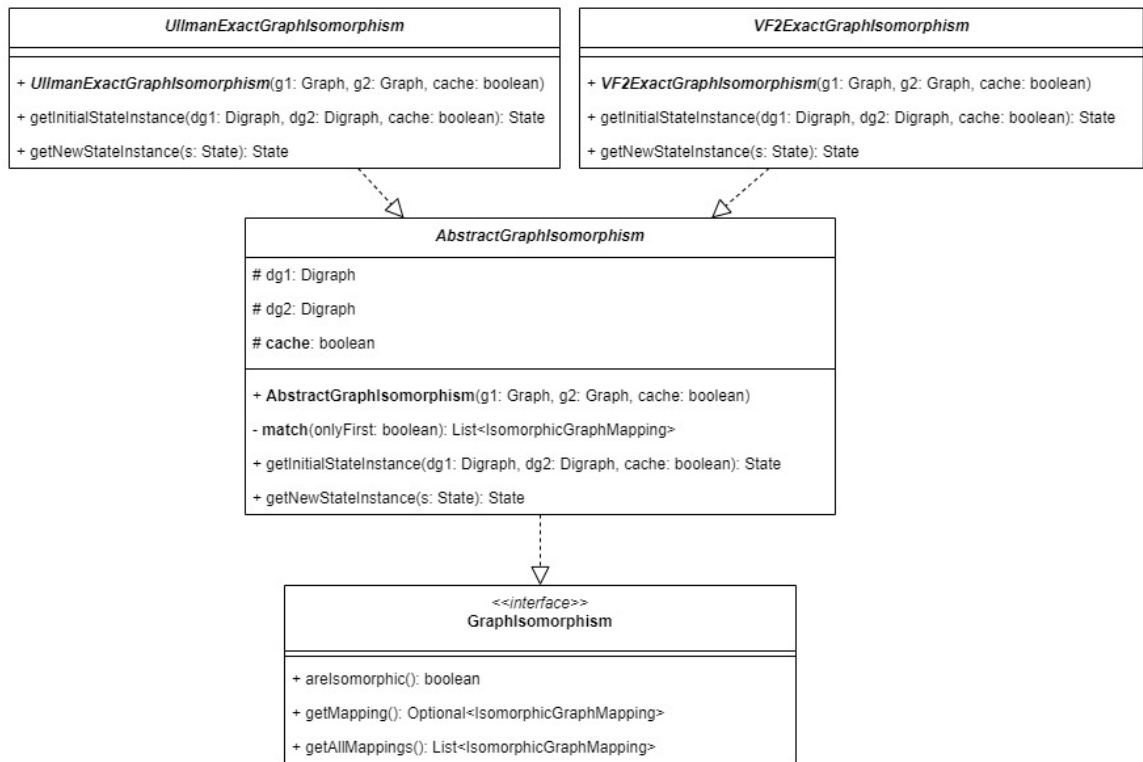


Figura 3.5: Arhitectura claselor de căutare al soluțiilor

Pentru fiecare pereche candidat a stării curente (linia 7) verificăm fezabilitatea ei (linia 8), iar în caz pozitiv vom adăuga în stivă starea curentă s pentru a putea reveni la ea, exact cum acționează o metodă recursivă asupra stivei programului.

```

1 List<IsomorphicGraphMapping> match(boolean onlyFirst){
2     List<IsomorphicGraphMapping> mappings = new ArrayList<>(); // solutions
3     Deque<State> stack = new ArrayDeque<>(); // state stack
4
5     State s = getInitialStateInstance(this.dg1, this.dg2, this.cache); // current state
6     while(true){
7         while(s.nextPair()){ // candidate pairs
8             if(s.isFeasiblePair()){ // feasibility check
9                 stack.push(s);
10
11                 s = getNewStateInstance(s);
12                 s.addPair(); // add the candidate pair to the state
13                 s.resetPreviousVertices(); // set as new state
14
15                 if(s.isDead()) // dead end
16                     break;
17                 if(s.isGoal()) { // solution found
18                     mappings.add(s.getMapping());
  
```

```

19         if(onlyFirst)
20             return mappings;
21     }
22 }
23 }
24 if(stack.isEmpty()) // explored all the searching space
25     break;
26 s.backtrack(); // restore shared memory
27 s = stack.pop(); // restore not-shared memory
28 }
29 return mappings;
30 }

```

Metoda **getNewStateInstance(s)** întoarce o nouă stare identică cu s , în care variabilele partajate sunt doar referențiate. Această stare va deveni noua stare curentă la care adăugăm perechea candidat. Dacă suntem în situația unui blocaj (linia 15), ieșim din bucla while interioară deoarece chiar dacă există perechi candidat fezabile pentru această stare știm sigur că ea nu poate duce spre izomorfism (ex.: la algoritmul lui Ullman unul din nodurile grafului G_1 nu are niciun nod candidat din G_2). Altfel, dacă starea curentă este o soluție (maparea este completă), o adăugăm în lista de soluții și eventual ne oprim dacă se dorește doar prima soluție găsită.

În cazul în care am parcurs toate perechile candidat ale stării curente ori starea curentă este un blocaj, atunci va trebui să revenim la starea anterioară adăugării ultimei perechi candidat, dacă nu am explorat întreg spațiul de căutare (linia 24). Aceasta se realizează prin extragerea din stivă (linia 27), dar înainte de aceasta trebuie să restaurăm variabilele partaje între stări (linia 26).

3.4 Implementare Ullman

Implementarea algoritmului lui Ullman urmărește descrierea din capitolul 2, având însă o modificare majoră. Matricea de compatibilitate a fost modificată astfel: 0 înseamnă **COMPATIBIL**, iar o valoare strict pozitivă înseamnă incompatibilitate între noduri. Inițial, în maparea vidă, nodurile incompatibile sunt marcate prin valoarea -1 .

Prin modificarea adusă matricii de compatibilitate, am reușit să aduc o îmbunătățire algoritmului original în ceea ce privește memoria. Mai exact, această modificare mi-a permis să partajez matricea între stări și astfel să ajung la o complexitate spațiu $\Theta(n^2)$.

Pe lângă matricea de compatibilitate, avem nevoie și de structurile:

- vectorii **core1[]** și **core2[]** de lungime n pentru a memora maparea curentă.
- *core_len* reține lungimea mapării curente.
- *prev_1*, *prev_2* rețin perechea candidat. Nu vom calcula toate perechile candidat deodată, ci o să calculăm iterativ o nouă pereche candidat, pentru a face economie de timp. La crearea unei noi stări, aceste variabile sunt inițializate cu valoarea -1, pentru a semnala că starea este nouă și astfel că nu a fost calculată nicio pereche candidat.

În cele ce urmează, vom detalia metodele din cadrul unei stări Ullman:

- metoda **nextPair()**: în *prev_1* vom salva valoarea *core_len* care reprezintă următorul nod din G_1 ce trebuie mapat, iar în *prev_2* vom salva indicele următorului nod din G_2 care este compatibil cu *prev_1*.

Dacă starea curentă este nouă, adică că $prev_1 = prev_2 = -1$, atunci $prev_2 \leftarrow 0$. Altfel, *prev_2* este incrementat până când găsim un nod compatibil sau depășim numărul de noduri.

- Procedura **isFeasiblePair()**: nu face nimic, deoarece știm deja că nodurile sunt compatibile;
- Procedura **addPair()**: adaugă la starea s perechea candidat găsită.

Mai întâi incrementăm *core_len*, după care modificăm matricea de compatibilitate astfel: nodul *prev_1* va fi compatibil doar cu nodul *prev_2*, restul nodurilor care erau compatibile (valoarea 0) cu *prev_1* fiind marcate drept incompatibile prin valoarea *core_len*.

- Procedura **refine()**: se procedează similar pseudocodului (algoritm 2), cu simpla diferență că nodurile care devin incompatibile vor fi marcate cu valoarea *core_len*.
- Procedura **backtrack()**: restaurăm matricea de compatibilitate la starea anterioară adăugării ultimei perechi adăugate. Din modul în care am ales modificat matricea la pașii anteriori, vom putea căuta în matrice valorile *core_len* și să le rescriem în valoarea 0 (COMPATIBIL).

- Procedura **isDead()**: verifică dacă există un nod din G_1 pentru care nu există nod compatibil în G_2 . Aceasta se face prin parcurgerea liniilor matricei de compatibilitate: dacă o linie are doar elemente diferite de 0, atunci nodul corespunzător acelei linii nu are candidați.

3.5 Implementare VF2

Și în cazul algoritmului VF2 implementarea sa urmărește descrierea din capitolul anterior. Structurile de date dintr-o stare VF2 sunt:

- vectorii **core1[]** și **core2[]** pentru maparea parțială.
- *core_len* reține lungimea mapării.
- *prev_1*, *prev_2* rețin perechea candidat.
- *last_added_1* ultimul nod al grafului G_1 adăugat în maparea stării curente.
- vectorii de marcare **in1[]**, **out1[]**, **in2[]** și **out2[]**.
- contorii $t1in_len = |T_1in(s)|$, $t1out_len = |T_1out(s)|$ și $t1both_len = |T_1in(s) \cap T_1out(s)|$ pentru a nu calcula cardinalele acestor mulțimi la fiecare verificare a condiției 2.1. Similar, avem și pentru graful G_2 variabilele: *t2in_len*, *t2out_len* și *t2both_len*.

În cele ce urmează, vom detalia metodele din cadrul unei stări VF2:

- metoda **nextPair()**: calculează pe rând *prev_1*, apoi *prev_2*.

```

1 public boolean nextPair() {
2     if (prev_1 == NULL_NODE && prev_2 == NULL_NODE)
3         prev_1 = prev_2 = 0;
4     else
5         prev_2++;
6     if (t1both_len - core_len > 0 && t2both_len - core_len > 0) { // both 'in' and 'out' vertices
7         while (prev_1 < n1 &&
8             (core_1[prev_1] != NULL_NODE || out1[prev_1] == 0 || in1[prev_1] == 0))
9             { prev_1++; prev_2 = 0; }
10    }
11    else if (t1out_len - core_len > 0 && t2out_len - core_len > 0) { // 'out' vertices

```

```

12     while (prev_1 < n1 && (core_1[prev_1] != NULL_NODE || out1[prev_1] == 0))
13     { ... }
14 }
15 else if (t1in.len - core.len > 0 && t2in.len - core.len > 0) { // 'in' vertices
16     while (prev_1 < n1 && (core_1[prev_1] != NULL_NODE || in1[prev_1] == 0))
17     { ... }
18 }
19 else { // unmapped vertices
20     while (prev_1 < n1 && core_1[prev_1] != NULL_NODE)
21     { ... }
22 }
23
24 // compute prev_2 in the same way
25 ...
26 }

```

Dacă starea curentă este nouă, adică $prev_1 = prev_2 = -1$, atunci se vor inițializa $prev_1 \leftarrow 0$ și $prev_2 \leftarrow 0$. Altfel, $prev_2$ este incrementat pentru a găsi următorul nod din G_2 posibil candidat pentru $prev_1$.

Se alege apoi $prev_1$: căutarea începe în mulțimea $T_1in(s) \cap T_1out(s)$, apoi continuă cu T_1out , urmat de T_1in și în final de mulțimea nodurilor nemapate. Nodul $prev_2$ se va alege din *aceeași categorie* de noduri ca și $prev_1$. Un aspect important este că la fiecare modificare a lui $prev_1$ trebuie să reinițializăm $prev_2 \leftarrow 0$.

Comparativ cu **Ullman**, observăm că nodurile grafului G_1 nu sunt mapate într-o anumită ordine, ci în funcție de mulțimile terminale.

Pentru a verifica dacă o mulțime terminală are elemente, trebuie să scădem din contorul corespunzător cardinalului ei lungimea mapării curente, deoarece și nodurile mapate sunt incluse în mulțimile terminale.

- Procedura **isFeasiblePair()**: verifică atât regulile de consistență cât și regulile de fezabilitate.

```

1 public boolean isFeasiblePair() {
2     int term_in1 = 0, term_out1 = 0, new_1 = 0,
3     term_in2 = 0, term_out2 = 0, new_2 = 0;
4     for (int other1 : o1.successors(prev_1)) { // successors
5         if (core_1[other1] != NULL_NODE) { // mapped vertex
6             int other2 = core_1[other1];

```

```

7         if (!o2.containsEdge(prev_2, other2)) // consistency rule
8             return false;
9     } else { // not mapped
10         if (in1[other1] != 0) { term_in1++; }
11         if (out1[other1] != 0) { term_out1++; }
12         if (in1[other1] == 0 && out1[other1] == 0) { new_1++; } // new vertex
13     }
14 }
15 for (int other1 : o1.predecessors(prev_1)) { ... } // predecessors
16
17 // do same for prev_2
18 ...
19 return term_in1 == term_in2 && term_out1 == term_out2 && new_1 == new_2;
20 }

```

Pentru fiecare succesori/predecesor al lui *prev_1*, respectiv *prev_2*: dacă acesta este mapat trebuie să verificăm consistența adăugării perechii candidat, adică dacă există muchie corespunzătoare în celălalt graf (linia 7); dacă nodul nu este mapat, trebuie să contorizăm apartenența lui la mulțimile terminale.

Pentru a stabili dacă adăugarea perechii candidat va putea duce spre izomorfism, proprietățile numerice calculate în cele două grafuri trebuie să fie egale (linia 19).

- Procedura **addPair()**: adaugă la starea *s* perechea candidat găsită.

```

1 public void addPair() {
2     last_added1 = prev_1;
3     core_1[prev_1] = prev_2;
4     core_2[prev_2] = prev_1;
5
6     core_len++;
7     if (in1[prev_1] == 0) { // not marked, then mark it as 'in'
8         in1[prev_1] = core_len;
9         tlin_len++; // increment the 'in' counter
10        if (out1[prev_1] != 0) { t1both_len++; } // if also 'out', then increment the 'both' counter
11    }
12    if (out1[prev_1] == 0) { ... }
13    for (int other1 : o1.successors(prev_1)) { // mark successors as 'out' and update the out/both counters
14        if (out1[other1] == 0) {
15            out1[other1] = core_len;
16            t1out_len++;

```

```

17         if (in1[other1] != 0) { t1both_len++; }
18     }
19 }
20 for (int other1 : o1.predecessors(prev_1)) // mark predecessors as 'in' and update the in/both counters
21     { ... }
22
23 // do same for prev_2
24 ...
25 }

```

Mai întâi salvăm ultimul nod mapat al grafului G_1 în variabila *last_added1*, pentru a putea face **backtrack** manual. Apoi se va completa maparea cu nodurile adăugate.

Incrementăm variabila *core_len* pe care o folosim pentru a marca succesorii/predecesorii lui *prev_1*, respectiv *prev_2*, nemarcați până acum. Vor fi marcate inclusiv nodurile *prev_1* și *prev_2*.

În această metodă vom actualiza și contorii cardinalelor mulțimilor terminale: la fiecare marcare a unui nod, se va incrementa contorul corespunzător (linia 16), iar în cazul în care acesta este marcat ca aparținând ambelor mulțimi terminale ('in' și 'out'), atunci se va incrementa și contorul intersecției (linia 17).

- Procedura **backtrack()**: restaurăm vectorii de marcare la starea anterioară adăugării ultimei perechi adăugate.

```

1 public void backTrack() {
2     int prev_1 = last_added1;
3     if (out1[prev_1] == core_len) { out1[prev_1] = 0; }
4     for (int other1 : o1.successors(prev_1))
5         if (out1[other1] == core_len) { out1[other1] = 0; } // marked as 'out' at last add(), then unmark
6     if (in1[prev_1] == core_len) { in1[prev_1] = 0; }
7     for (int other1 : o1.predecessors(prev_1))
8         if (in1[other1] == core_len) { in1[other1] = 0; } // marked as 'in' at last add(), then unmark
9
10    int prev_2 = core_1[last_added1];
11    // do same for prev_2
12    ...
13    core_1[prev_1] = core_2[prev_2] = NULL_NODE;
14    core_len--;
15 }

```


Având ultimul nod mapat din G_1 salvat în variabila *last_added1*, vom putea afla ușor și nodul corespondent din G_2 (linia 10). Apoi vom putea parcurge succesorii/predecesorii lor pentru a identifica nodurile marcate la pasul anterior, adică acelea care au valoarea *core_len* în vectorii de marcarea (liniile 3, 5, 6, 8).

În final, nodurile *prev_1* și *prev_2* vor fi eliminate din mapare (liniile 13, 14).

- Procedura **isDead()**: este verificată condiția 2.1, prin utilizarea contorilor.

3.6 Observații

Algoritmii Ullman și VF2 au fost adaptați pentru toate tipurile de graf:

- pseudodigrafuri: două noduri sunt compatibile dacă au același număr de bucle;
- multidigrafuri: două arce sunt compatibile dacă au aceeași multiplicitate;
- grafuri neorientate: acestea sunt convertite în grafuri orientate prin transformarea unei muchii în două arce.

Acești algoritmi analizează doar izomorfismul sintactic: se analizează doar structura grafurilor. Însă, în aplicațiile practice dorim să luăm în considerare și informația salvată într-un nod sau muchie. Folosind suportul din Graph4J pentru grafuri etichetate, stabilim o compatibilitate și în sens **semantic** între noduri/muchii.

Astfel:

- două noduri (*node1*, *node2*), cu $node1 \in G_1$ și $node2 \in G_2$, sunt compatibile dacă (Fig. 3.6):
 - etichetele nodurilor *node1* și *node2* sunt egale.
 - *node1* și *node2* au același număr de bucle.
 - listele $L1 = [\text{etichetele buclelor lui } node1]$ și $L2 = [\text{etichetele buclelor lui } node2]$ trebuie să fie egale.
- două muchii (*edge1*, *edge2*), cu $edge1 \in E(G_1)$ și $edge2 \in E(G_2)$, sunt compatibile dacă (Fig. 3.7):
 - *edge1* și *edge2* au aceeași multiplicitate.

- listele $L1 = [\text{etichetele tuturor muchiilor } edge1]$ și $L2 = [\text{etichetele tuturor muchiilor } edge2]$ trebuie să fie egale.

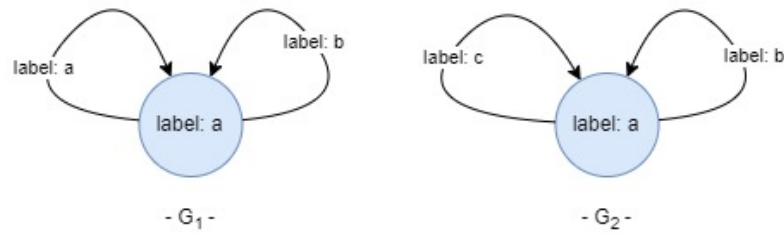


Figura 3.6: Două noduri cu aceeași etichetă, cu același număr de bucle, dar listele etichetelor $L1 = [a, b]$ și $L2 = [b, c]$ sunt diferite, deci nodurile sunt incompatibile.

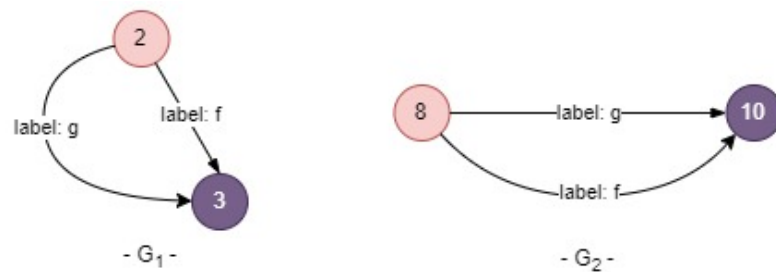


Figura 3.7: Două muchii cu aceeași multiplicitate, iar listele etichetelor $L1 = [g, f]$ și $L2 = [f, g]$ sunt egale, deci muchiile sunt compatibile.

Algoritmii sunt modificați astfel:

- **Ullman**: la inițializarea matricii de compatibilitate pe lângă egalitatea între gradele nodurilor se va verifica și compatibilitatea semantică descrisă anterior.

În plus, în procesul de rafinare a candidaților: pentru orice pereche de noduri compatibile (i, j) , cu $i \in G_1$ și $j \in G_2$, și pentru orice muchie $(i, x) \in E(G_1)$ trebuie să existe o muchie $(j, y) \in E(G_2)$ **compatibilă** astfel încât y să fie candidat pentru x (la fel și pentru orice muchie (x, i)).

- **VF2**: în cadrul procedurii de verificare a fezabilității unei perechi candidat trebuie să verificăm următoarele:
 1. compatibilitatea nodurilor din perechea candidat;
 2. în cadrul verificării regulilor de consistență (linia 7 din metoda **isFeasiblePair()**) pe lângă existența unei muchii corespunzătoare în celălalt graf trebuie să verificăm și compatibilitatea lor.

De asemenea, algoritmi propuși pot fi ușor extinși pentru a rezolva o problemă derivată: problema izomorfismului pe subgraf indus. În această problemă, vom căuta în graful G_2 un subgraf indus izomorfic cu graful G_1 .

În cazul algoritmului lui Ullman diferența constă în faza de inițializare a matricei de compatibilitate: un nod $i \in G_1$ este compatibil cu un nod $j \in G_2$ dacă gradul intern/extern al lui i este mai mic sau egal (\leq) decât gradul intern/extern al lui j .

În cazul algoritmului VF2 diferența constă în faza de verificare a fezabilității: regulile R_IN, R_OUT și R_NEW vor fi modificate prin înlocuirea operatorului $=$ cu \leq . Aceeași modificare se face și în cadrul condiției 2.1 folosită de către metoda **isDead()**.

3.7 Implementare algoritm pentru arbori

Implementarea algoritmului pentru determinarea izomorfismului între arbori respectă îndeaproape descrierea din capitolul anterior.

Vom descrie mai întâi procedura de etichetare a unui nod dintr-un arbore.

```
1 private int findLabel(Graph tree, int node, int[] labels, Map<List<Integer>, Integer> labelListToInt){
2     List<Integer> list = new ArrayList<>();
3     for (int child : tree.neighbors(node)){
4         int label = labels[tree.indexOf(child)];
5         if(label != -1) { list.add(label); } // not the parent
6     }
7     RadixSort.sort(list);
8     Integer intLabel = labelListToInt.get(list);
9     if (intLabel == null) { // not in the map
10         labelListToInt.put(list, newLabel);
11         intLabel = newLabel;
12         this.newLabel++;
13     }
14     labels[tree.indexOf(node)] = intLabel; // update
15     return intLabel;
16 }
```

Metoda primește ca parametri: arborele, nodul din arbore ce trebuie etichetat, vectorul de etichete care va fi completat și o mapare de la listele de etichete la un număr întreg (mapare comună pentru ambii arbori).

Pentru a afla descendenții direcți ai unui nod, vom parcurge vecinii lui și vom verifica dacă aceștia au fost deja etichetați (linia 5). După ce am calculat lista de etichete

a descendenților direcți, o vom sorta și o vom căuta în maparea trimisă ca parametru (linia 8). Dacă nu există o valoare asociată acestei liste, vom adăuga o nouă etichetă prin incrementarea variabilei *newLabel*. În final, eticheta găsită va fi asociată nodului curent.

După finalizarea etichetării nodurilor de pe un același nivel, vom construi 2 liste cu etichetele nodurilor de pe nivelul respectiv pentru cei 2 arbori, pe care le vom compara, iar în caz de inegalitate algoritmul se oprește, însemnând că arborii nu sunt izomorfi.

Metoda principală care verifică izomorfismul între cei 2 arbori, trebuie să realizeze următorii pași:

1. gruparea nodurilor pe nivele;
2. etichetarea nodurilor de pe fiecare nivel, pornind de la cel mai de jos nivel;
3. compararea etichetelor nodurilor de pe același nivel;
4. stabilirea mapării între nodurile arborilor.

În cazul în care nu avem același număr de nivele sau același număr de noduri pe un anumit nivel, atunci arborii în mod cert nu sunt izomorfi.

Dacă rădăcinile celor 2 arbori au aceeași etichetă, atunci ei sunt izomorfi caz în care va trebui să calculăm o mapare între nodurile acestora, mapare calculată pe baza etichetării nodurilor.

```
1 private void matchNodes(int[] labelList1, int[] labelList2) {
2     forwardMapping = new int[tree1.numVertices()];
3     backwardMapping = new int[tree2.numVertices()];
4     Arrays.fill(forwardMapping, -1);
5     Arrays.fill(backwardMapping, -1);
6
7     Queue<Pair<Integer, Integer>> q = new ArrayDeque<>(); // queue for BFS
8     q.add(new Pair<>(root1, root2));
9     while(!bfsQueue.isEmpty()){
10         Pair<Integer, Integer> p = q.poll();
11         int node1 = p.first();
12         int node2 = p.second();
13         forwardMapping[tree1.indexOf(node1)] = node2;
14         backwardMapping[tree2.indexOf(node2)] = node1;
15         // match every child of node1 with a child of node2 with the same label
```

```

16      // and add them to the queue
17      ...
18  }
19 }

```

Această metodă primește ca parametri etichetarea nodurilor celor doi arbori. Pentru a calcula maparea între nodurile arborilor, vom parcurge în lățime nodurile celor doi arbori simultan, cu diferența că în coadă vom avea perechi de noduri $(node1, node2)$, însemnând că nodul $node1$ din arborele 1 este mapat cu nodul $node2$ din arborele 2.

Pentru fiecare pereche de noduri $(node1, node2)$ eliminată din coadă, vom marca mai întâi maparea lor în vectorii de mapare (liniile 13, 14), după care vom grupa fiecare descendent direct al nodului $node1$ cu un descendent direct al nodului $node2$ cu aceeași etichetă, pe care îi vom adăuga în coadă pentru a fi procesați la rândul lor.

Algoritmul de mai sus funcționează doar pentru arbori în care este specificată rădăcina. În cazul general al arborilor, unde nu avem specificată rădăcina, vom folosi drept rădăcină centroizii arborilor, datorită unicității și a centralității lor în arbore. De exemplu, dacă avem câte un singur centroid în fiecare arbore, atunci în cazul existenței izomorfismului acești centroizi cu siguranță vor fi mapați între ei.

Important de menționat este că trebuie să avem același număr de centroizi în ambii arbori. Deoarece sunt cel mult doi centroizi pentru fiecare arbore, vom avea cel mult 2 combinații de rădăcini și respectiv cel mult 2 apeluri ale algoritmului de mai sus. Dacă un apel confirmă că arborii sunt izomorfi, atunci aceștia sunt izomorfi cu rădăcinile respective. Deci, complexitatea algoritmului pentru arbori fără rădăcină rămâne liniar.

De asemenea, algoritmul a putut fi extins și pentru păduri de arbori. Dacă rădăcinile arborilor sunt specificate, atunci putem transforma pădurea într-un arbore cu rădăcină: creăm un nou nod și adăugăm muchii de la el la rădăcinile arborilor. Se va apela apoi algoritmul pentru arbori cu rădăcină, în care rădăcină este noul nod creat. Complexitatea timp este liniară și în acest caz.

În cazul în care nu sunt specificate rădăcinile arborilor, vom proceda astfel:

- pentru arborii cu un singur centroid, alegem rădăcina în acel centroid;
- pentru arborii cu doi centroizi u și v , caz în care sunt adiacenți, vom insera un nou nod pe muchia (u, v) , nod care va deveni rădăcina arborelui.

Vom grupa apoi arborii cu un singur centroid, unind rădăcinile lor cu o nouă rădăcină. La fel vom grupa și arborii cu doi centroizi. Facem această grupare, pentru a mapa noduri aparținând unor arbori cu același număr de centroizi, o proprietate esențială în determinarea izomorfismului. În final, cei doi arbori obținuți vor fi uniți printr-o rădăcină, după care va fi apelat algoritmul pentru arbori cu rădăcină. În continuare, complexitatea timp a rămas liniară.

3.8 Testarea corectitudinii

O parte importantă a dezvoltării algoritmilor a fost testarea corectitudinii acestora pentru a asigura funcționarea lor corectă în orice situație. Pentru acest lucru am folosit cadrul de testare **JUnit** care ne ajută în definirea și analiza testelor unitare.

Pentru cazul izomorfismului între grafuri, am definit teste pentru mai multe categorii de grafuri:

- grafuri mici, eventual etichetate, izomorfe și neizomorfe unde pot fi verificate și mapările găsite;
- grafuri de diferite forme: linie, triunghi, stea, arbore, graf complet, graf bipartit;
- grafuri de diferite tipuri: orientate, neorientate, pseudografuri, multigrafuri.
- cazuri marginale de grafuri: grafuri fără noduri/muchii, grafuri cu număr diferit de noduri/muchii, grafuri de tipuri diferite (orientate, neorientate);
- grafuri izomorfe generate aleatoriu.

Similar, pentru cazul izomorfismului între arbori, am definit teste pentru categoriile:

- arbori mici izomorfi și neizomorfi;
- arbori de diferite forme: arbori degenerați, arbori de adâncime 1;
- cazuri marginale de arbori: arbori fără noduri, arbori cu număr diferit de noduri, cu rădăcina specificată sau nu;
- arbori izomorfi generați aleatoriu.

De altfel, am testat și cazul pădurilor de arbori cu rădăcina specificată sau nu.

În cazul grafurilor/arborilor izomorfi generați aleatoriu, am verificat că mapările găsite de algoritmi sunt într-adevăr izomorfe folosind metoda **isValidIsomorphism()** din cadrul clasei **IsomorphicGraphMapping**.

Testarea riguroasă a confirmat corectitudinea implementărilor în diverse situații. După testarea corectitudinii algoritmilor, am măsurat și performanțele acestora, rezultate prezentate în capitolul următor.

Capitolul 4

Testarea performanțelor

În acest capitol vom prezenta performanțele algoritmilor implementați, comparând atât timpul de execuție cât și memoria cu algoritmi similari din alte biblioteci cunoscute, precum **JGraphT** sau **NetworkX**.

Experimentele au fost rulate pe un laptop cu următoarele specificații: Intel i5-1035G1 CPU @ 0.950GHz (8 CPUs), 1.2 GHz, cu o memorie RAM 24 GB, pe care rulează Windows 11 Pro 64-bit.

Pentru **Graph4J** și **JGraphT**, testele au fost realizate cu versiunea 20 a limbajului de programare Java, iar pentru **NetworkX** cu versiunea 3.12 a limbajului de programare Python.

Înainte de a măsura efectiv metricile de timp și memorie, am 'încălzit' Mașina Virtuală Java (JVM) prin rularea repetată a unui test de dimensiuni mici. Astfel, ne asigurăm că au fost încărcate în memorie acele clase de care avem nevoie și că declanșează compilatorul Just-In-Time(JIT) care va optimiza performanțele aplicației ([19]).

La fel ca pentru platforma de programare **Java**, pentru testele bibliotecii **NetworkX** am realizat o etapă de 'încălzire' pentru a mă asigura că toate pachetele folosite sunt deja încărcate în memorie în momentul începerii testelor, iar pentru măsurarea memoriei și a timpului am folosit module specializate, **memory_profiler** și **timeit**.

4.1 VF2

Pentru biblioteca **NetworkX** am luat în considerare atât algoritmul VF2, cât și versiunea îmbunătățită **VF2++**. Deoarece algoritmi au fost scriși în totalitate în limbajul Python, aceștia tind să nu performeze foarte bine.

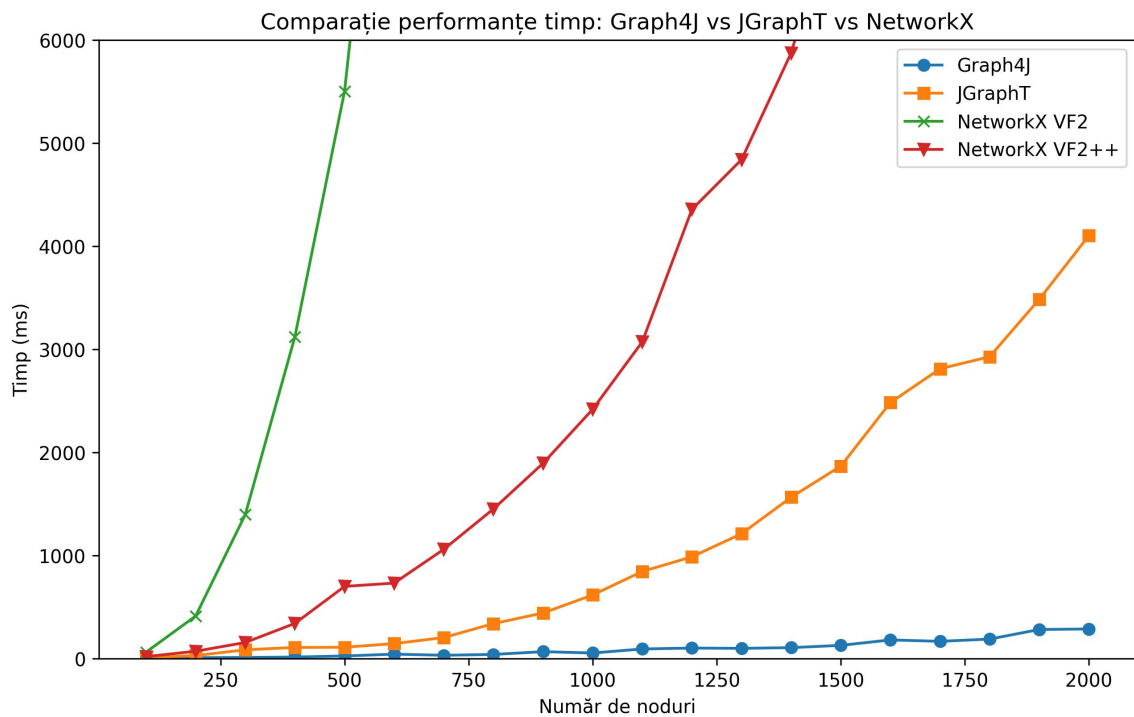


Figura 4.1: Timpul de execuție al algoritmilor pe grafuri aleatorii gnp, unde $p = 0.3$

Pentru următoarele 3 teste am generat o serie de grafuri orientate aleatorii cu dimensiuni între 50 și 2000 de noduri. Am folosit modelul **Erdős–Rényi** [20], cunoscut și sub abrevierea **Gnp**, unde **n** este numărul de noduri, iar **p** este probabilitatea ca între două noduri diferite să existe o muchie.

Primul test este realizat pe un graf **rar**. Figura (Fig. 4.1) validează performanțele în ceea ce privește timpul de execuție al algoritmului implementat în biblioteca **Graph4J**. Deși algoritmului din biblioteca **JGraphT** are o creștere a timpului de execuție mult mai rapidă decât cel din **Graph4J**, acesta performează mult mai bine decât ambii algoritmi din **NetworkX**, însă putem observa că versiunea VF2++ este într-adevăr mai rapidă.

În figura (Fig. 4.2) putem observa că algoritmi din Graph4J și JGraphT performează similar din punct de vedere al memoriei, acest lucru datorându-se abordării similare. Aceste diferențe devin nesemnificative atunci când considerăm performanțele în ceea ce privește timpul de execuție. În schimb, NetworkX consumă considerabil mai multe resurse de memorie, însă din nou versiunea VF2++ pare a fi mai eficientă.

Al doilea test a fost realizat pe grafuri de densitate medie, având probabilitatea p egală cu 0,6. Analizând figura (Fig. 4.3), putem observa diferențe enorme între

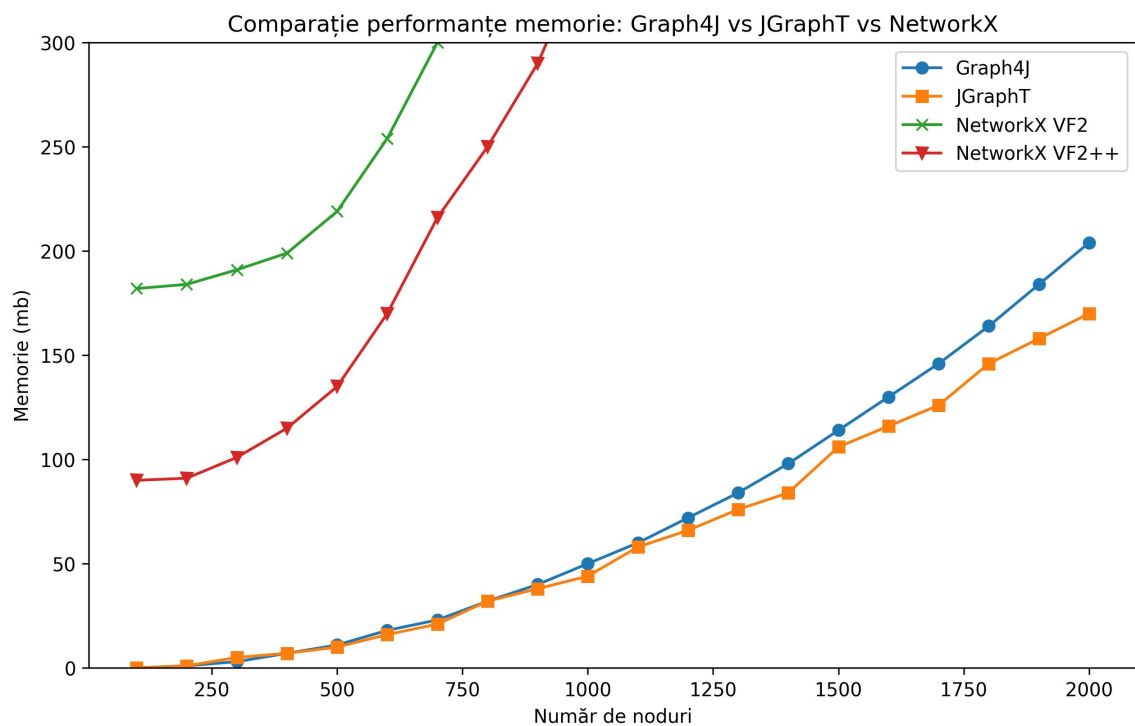


Figura 4.2: Memoria consumată de algoritmi pe grafuri aleatorii gnp, unde $p = 0.3$

cei patru algoritmi din privința timpului de execuție. În continuare, algoritmul din Graph4J rămâne net superior celorlalți algoritmi.

În ceea ce privește memoria (Fig. 4.4) lucrurile rămân la fel: diferențele dintre Graph4J și JGraphT rămân mici, iar NetworkX consumă, în ambele versiuni, cea mai multă memorie.

Al treilea test a fost realizat pe grafuri **dense**, având probabilitatea p egală cu valoarea 0,9 . Din nou putem observa în (Fig 4.5) diferențele enorme în timpul de execuție pentru cei patru algoritmi, cel din GraphJ fiind iarăși lider.

Putem observa și o creștere a timpului de execuție odată cu creșterea densității grafurilor în cazul algoritmilor din JGraphT și NetworkX. În schimb algoritmul din Graph4J pare să rămână aproape neschimbat.

Memoria consumată urmărește același tipar (Fig. 4.6): diferențe minore de memorie dintre Graph4J și JGraphT. Însă putem observa în cadrul tuturor algoritmilor o creștere a memoriei proporțională cu densitatea grafurilor.

În urma experimentelor realizate, putem trage o concluzie: algoritmul implementat în biblioteca **Graph4J** este mult mai rapid decât cei din **JGraphT** și **NetworkX**

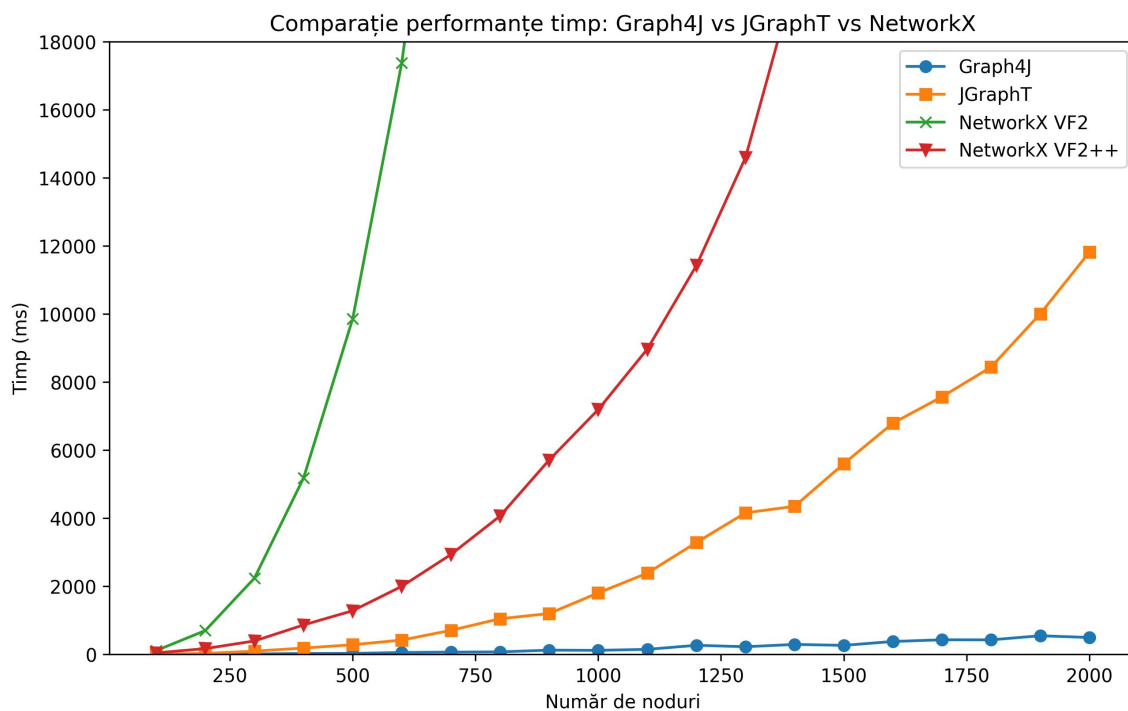


Figura 4.3: Timpul de execuție al algoritmilor pe grafuri aleatorii gnp, unde $p = 0.6$

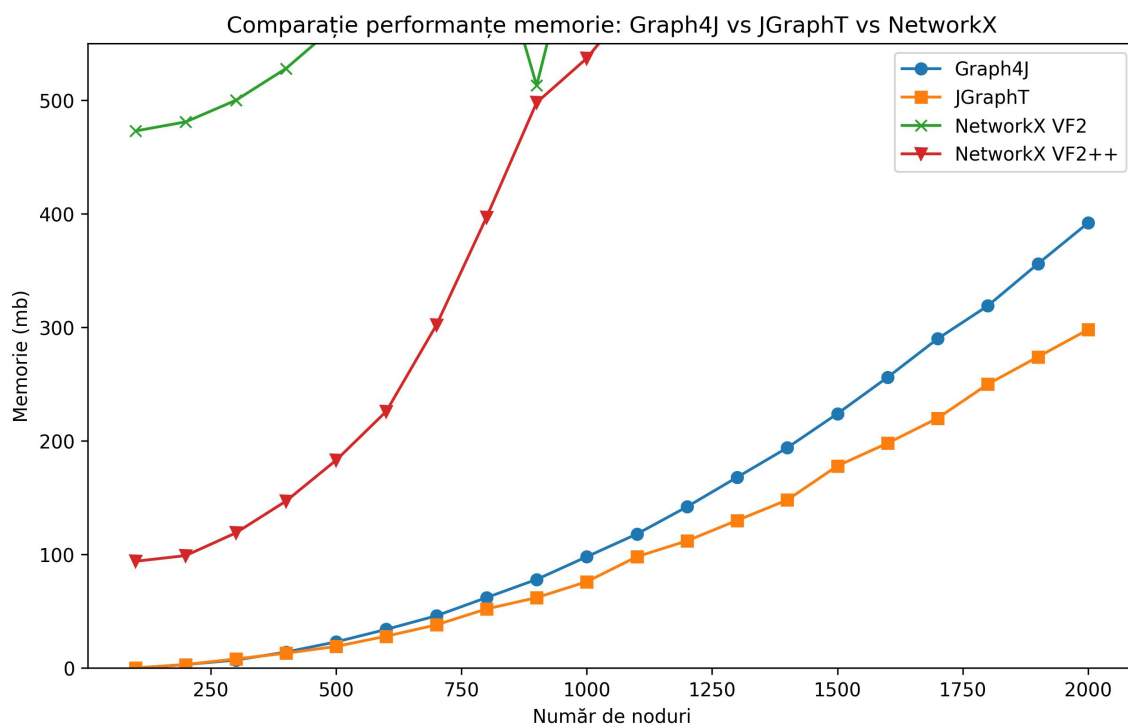


Figura 4.4: Memoria consumată de algoritmi pe grafuri aleatorii gnp, unde $p = 0.6$

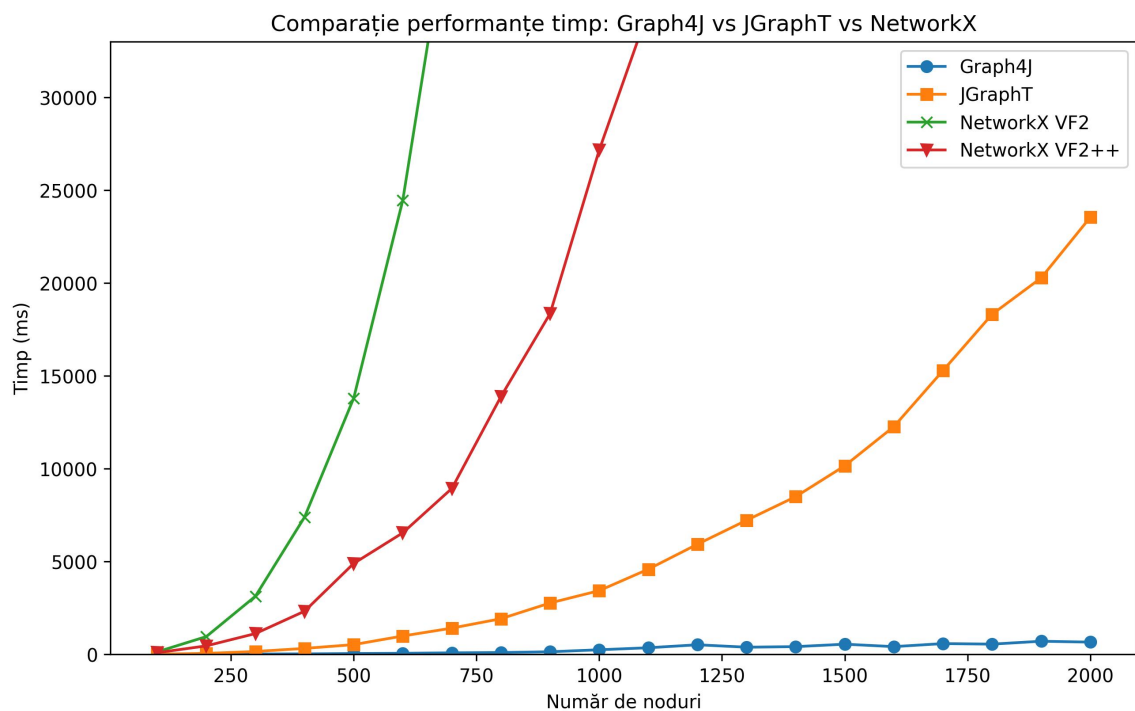


Figura 4.5: Timpul de execuție al algoritmilor pe grafuri aleatorii gnp, unde $p = 0.9$

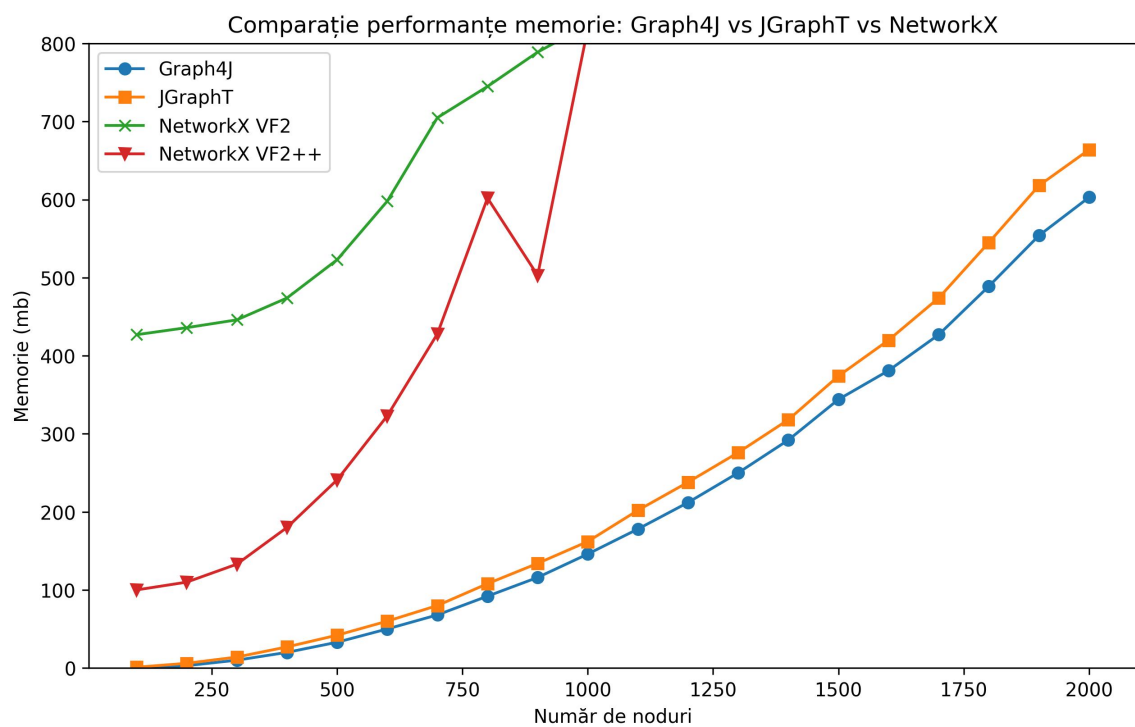


Figura 4.6: Memoria consumată de algoritmi pe grafuri aleatorii gnp, unde $p = 0.9$

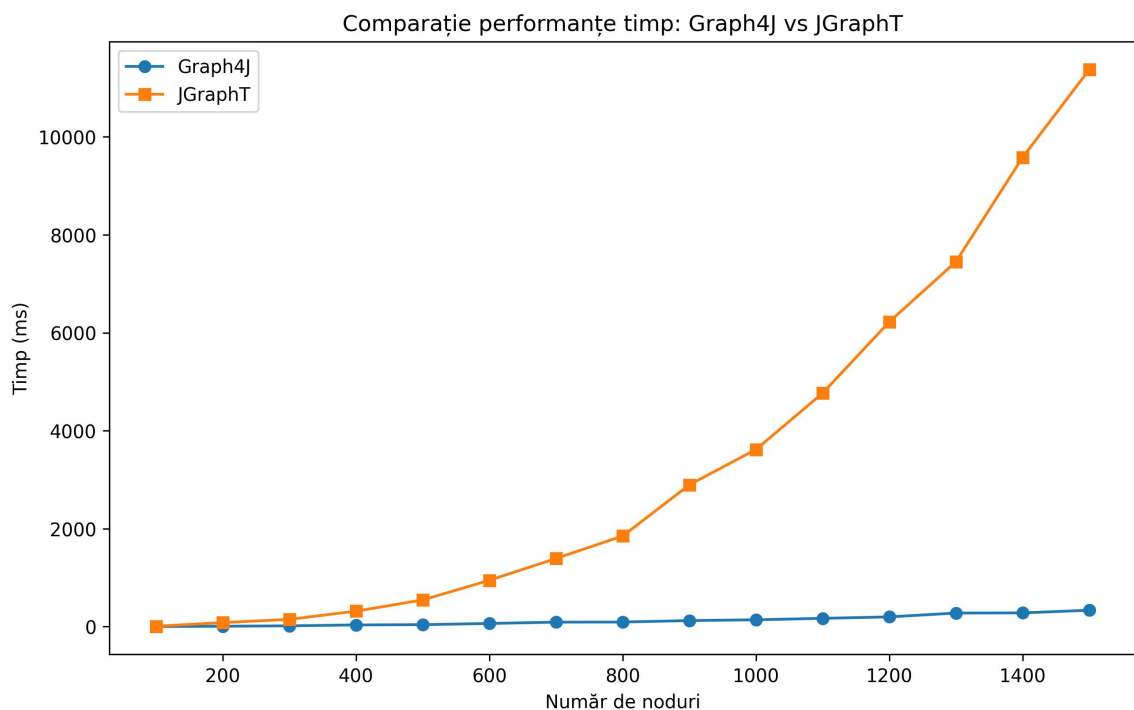


Figura 4.7: Timpul de execuție al algoritmilor pentru grafuri complete

(inclusiv versiunea VF2++). În ceea ce privește memoria, algoritmul implementat în **Graph4J** consumă un volum de memorie similar cu cel din **JGraphT**.

În cele ce urmează, vom analiza doar timpul de execuție dintre bibliotecile Graph4J și JGraphT pentru anumite clase speciale de grafuri.

Figura (Fig. 4.7) prezintă timpul de execuție al algoritmilor pentru grafuri complete. Algoritmul din Graph4J rămâne aproape constant în timp ce algoritmul din JGraphT are o creștere exponențială a timpului de execuție.

Figura (Fig. 4.8) prezintă performanțele pentru un tip grafuri puțin mai special: grafuri generate după modelul **Barabási–Albert** [21]. Acest model generează grafuri asemănătoare rețelelor sociale, rețeaua Internet sau **World Wide Web**. Pentru a genera astfel de grafuri avem nevoie de 3 parametri: număr inițial de noduri, gradul mediu al nodurilor, numărul total de noduri.

Chiar și în cazul acestor tipuri de grafuri, algoritmul din Graph4J are un timp de execuție mult mai mic decât cel din JGraphT.

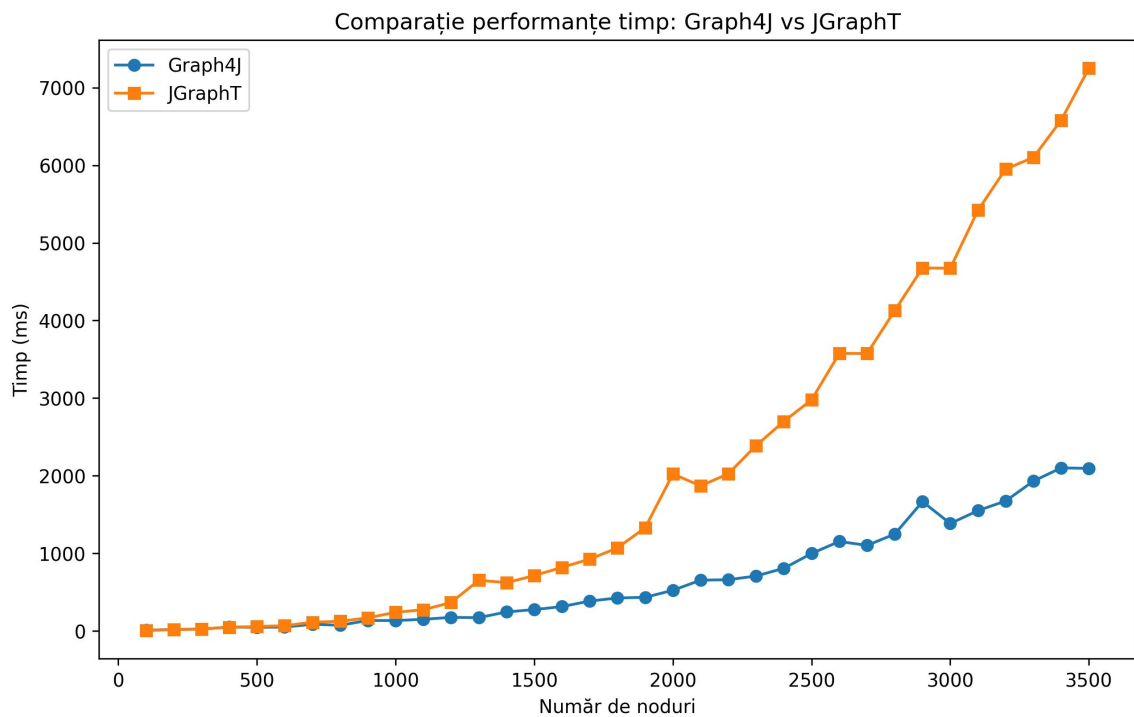


Figura 4.8: Timpul de execuție al algoritmilor pe grafuri Barabassi: număr inițial de noduri = 40% din numărul total de noduri; grad mediu = 40.

4.2 VF2 și densități diferite ale grafurilor

În acest experiment vom analiza tipul de graf pentru care algoritmul VF2 rulează cel mai rapid.

Așa cum am bănuir din descrierea algoritmului, acesta rulează mai rapid pentru grafuri rare (Fig. 4.9), deoarece va analiza în medie mai puțini vecini în cadrul metodelor de verificare a fezabilității unei perechi candidat, de adăugare a unei perechi la o mapare parțială și de **backtrack** manual.

4.3 Ullman vs VF2

În acest experiment vom arăta eficiența algoritmului **VF2** în comparație cu algoritmul lui **Ullman**. Experimentul a fost realizat pe grafuri rare și confirmă performanțele algoritmului VF2, net superioare algoritmului Ullman (Fig. 4.10).

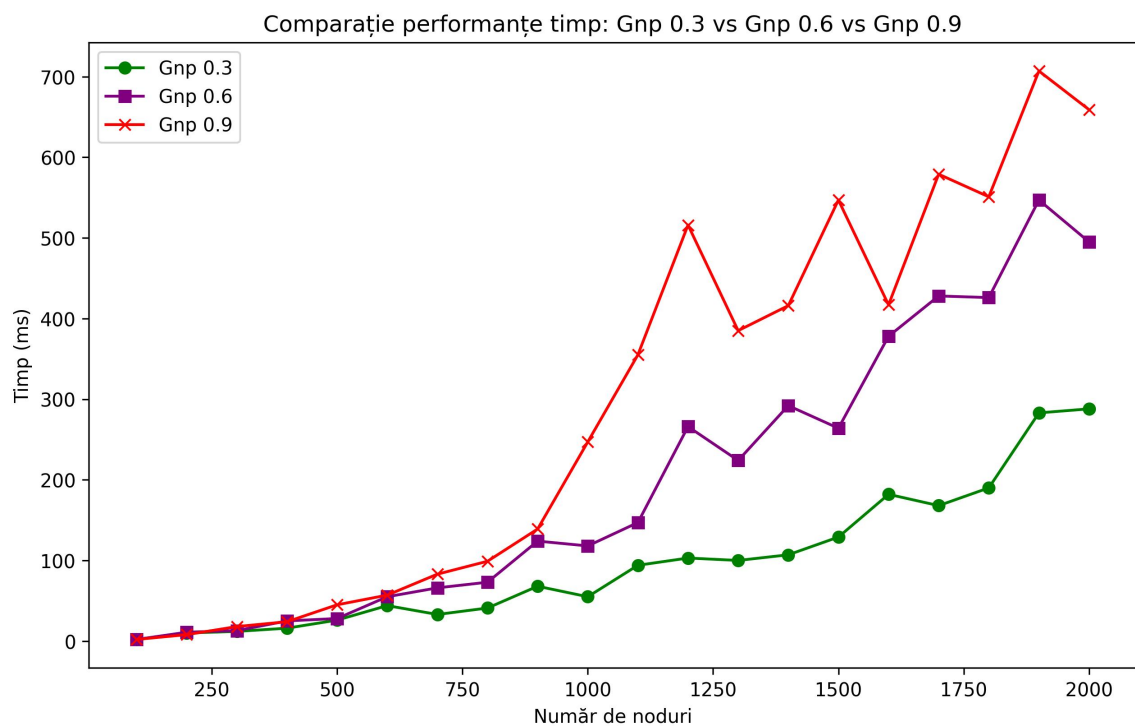


Figura 4.9: Timpul de execuție al algoritmului VF2 pentru grafuri aleatorii de anumită densitate

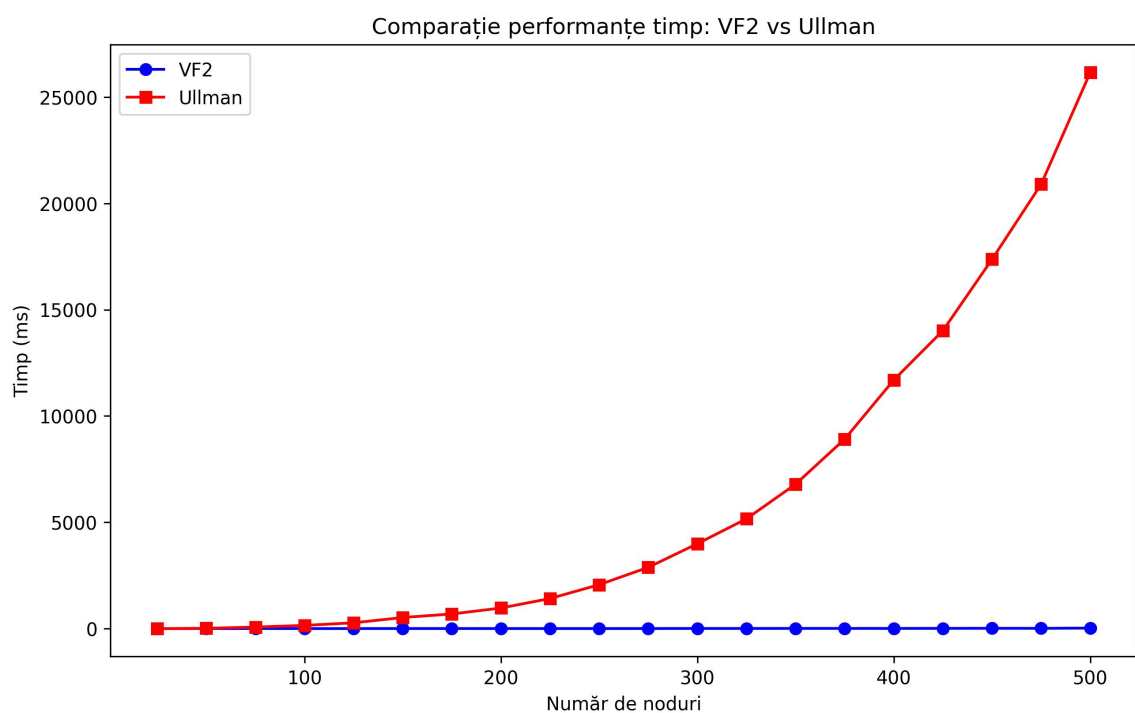


Figura 4.10: Diferența timpului de execuție dintre algoritmi Ullman și VF2 pentru grafuri aleatorii gnp, unde $p = 0.3$

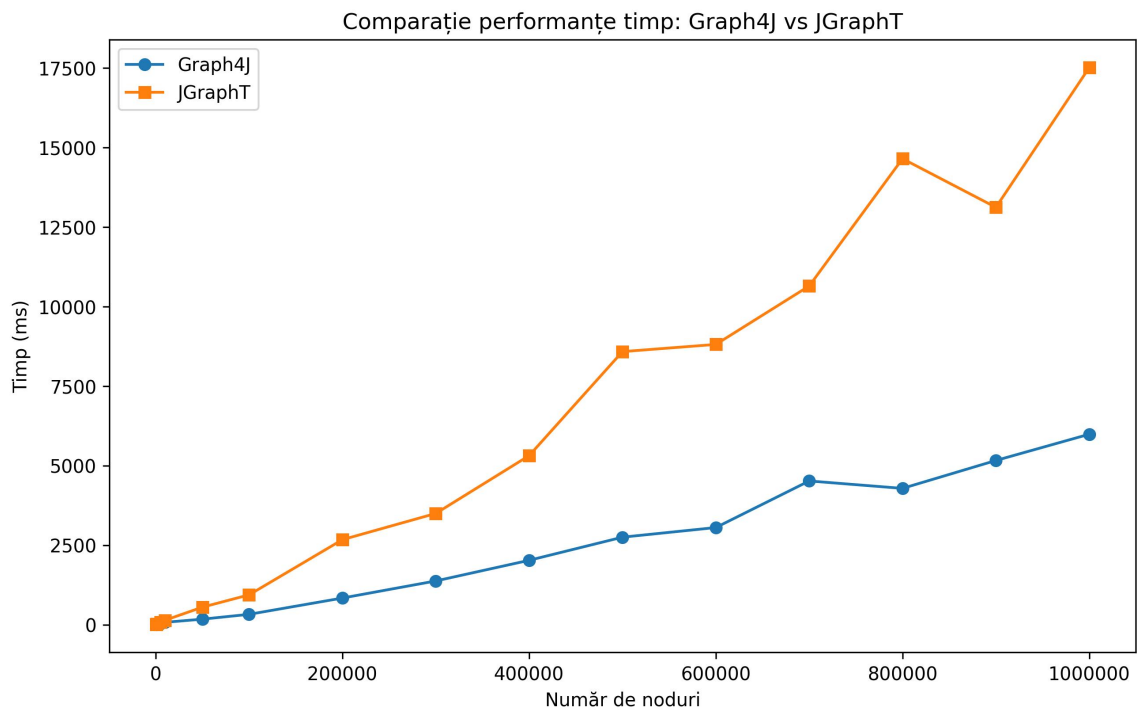


Figura 4.11: Timpul de execuție al algoritmului de izomorfism între arbori generați aleator

4.4 Algoritm pentru arbori

Performanța algoritmului pentru izomorfismul arborilor a fost testată pe o serie de arbori generați aleator cu număr de noduri între 10 și 1.000.000 , folosind modulul **generate** din biblioteca **Graph4J**.

După cum se poate vedea în (Fig. 4.11), algoritmul implementat în biblioteca Graph4J este mult mai rapid decât cel din JGraphT, acest lucru datorându-se în mare parte eficienței operațiilor de bază pe grafuri.

Algoritmul implementat în Graph4J este mai eficient și din punct de vedere al memoriei consumate (Fig. 4.12), datorită folosirii predominante a tipurilor de date primitive.

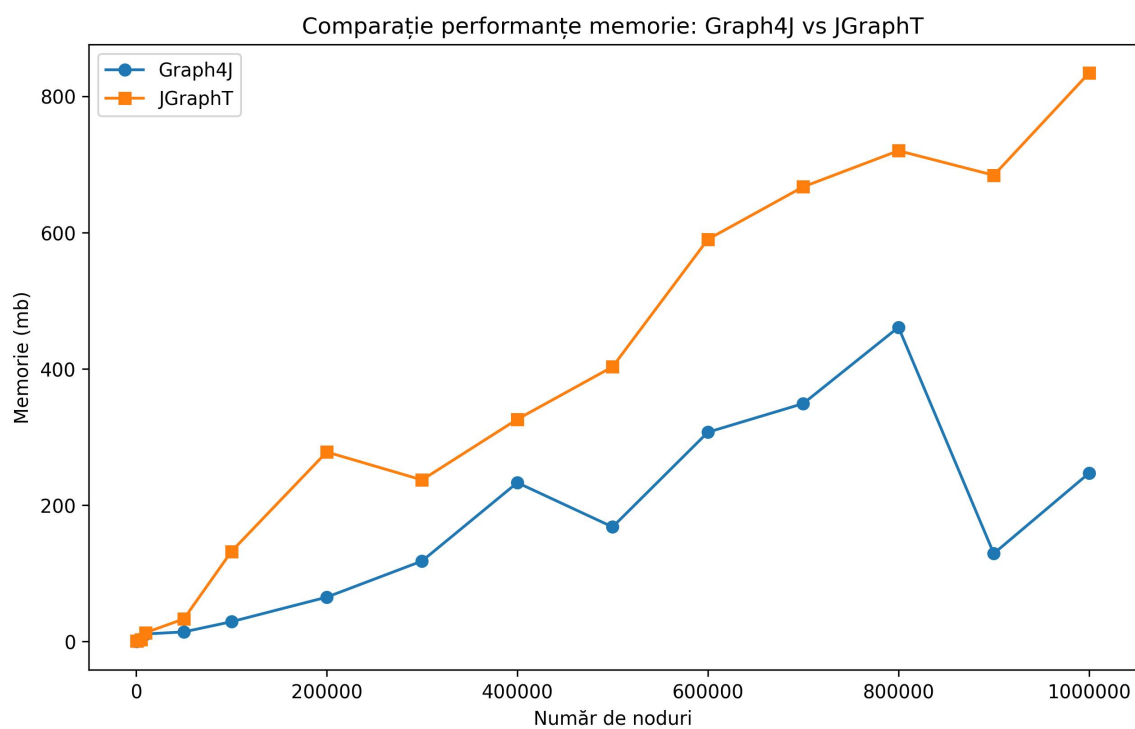


Figura 4.12: Memoria consumată de algoritmul de izomorfism între arbori generați aleator

Capitolul 5

Interfață testare algoritmi

5.1 Java Swing

Java Swing este un cadru pentru construirea interfețelor grafice(GUI) apărut pentru a combate anumite dezavantaje ale subsistemului GUI original al platformei de programare Java, anume **AWT**.

AWT definește un număr de componente de bază: butoane, ferestre, care suportă o interfață grafică utilizabilă, însă foarte limitată. Una din limitările sale vine din **dependența** față de platformă, fiecare componentă a sa fiind tradusă într-o componentă a sistemului de operare pe care rulează. [22]

Din cauza utilizării de cod nativ, componentele AWT sunt considerate ca fiind greoaie (eng. **heavyweight**). Totodată, au apărut probleme din cauza diferențelor dintre sistemele de operare, care constau în aspectul unei componente și modul cum se comportă (eng. **look and feel**). Această instabilitate a amenințat strategia urmărită de Java: *"Write Once Run Anywhere"*.

Nu mult după lansarea Java, au devenit evidente limitările pe care AWT le impunea. Astfel, s-a propus o abordare mai bună, anume **Java Swing**, construit peste conceptele de bază din AWT, dar care înlocuiește complet componentele sale.

Swing aduce avantajul portabilității, fiind independent de platformă. Toate componentele sale sunt scrise în totalitate în Java și în consecință sunt mult mai eficiente și flexibile.

În plus, deoarece componentele sale nu sunt translate în cod nativ, aspectul și modul în care se comportă sunt determinate de către Swing, ci nu de sistemul de operare.

De asemenea, Swing permite separarea logicii componentelor sale de modul în care arată. Astfel, a devenit posibilă conectarea, chiar la runtime, a unui look-and-feel personalizat care va fi consistent pe toate platformele.

Swing a fost construit pe baza modelului **Separable Model-and-View** care, așa cum sugerează și numele, permite separarea datelor aplicației față de componentele interfeței, facilitează dezvoltarea separată a celor două părți, contribuind la eficientizarea procesului de dezvoltare, dar și a costurilor de mentenanță.

Iar în ceea ce privește aplicații populare dezvoltate în Swing, majoritatea în domeniul afacerilor, IT și cercetare, putem menționa următoarele: suita de IDE-uri oferite de către **JetBrains**, **NetBeans** IDE, **Gephi** - vizualizator de grafuri și multe altele.

În aplicația prezentată, am ales **Java Swing** pentru dezvoltarea interfeței grafice datorită flexibilității, maturității sale și a comunităților create de-a lungul timpului. De altfel, oferă absolut tot suportul pentru dezvoltarea unei aplicații cu un aspect plăcut, dar mai ales care va funcționa ireproșabil pe orice sistem de operare.

Adițional, faptul că biblioteca **Graph4J** este scrisă în limbajul Java a făcut facilă integrarea cu algoritmul de testare a izomorfismului, deoarece pentru modelul aplicației am folosit tipul de date grafuri din această bibliotecă.

5.2 Scopul aplicației

Am dezvoltat această interfață desktop pentru a face mai ușoară și mai intuitivă testarea algoritmilor propuși. În acest sens, am construit mai întâi un editor și vizualizator de grafuri, capabil de următoarele operații: adăugare de noduri/muchii, ștergere nodurilor/muchiilor, editarea proprietăților nodurilor/muchiilor, ștergerea completă a grafului și posibilitatea de a importa/exporta grafuri într-un format standard.

Apoi am integrat acest editor într-o interfață care pe lângă operațiile menționate anterior, permite testarea izomorfismului dintre cele 2 grafuri desenate și vizualizarea intuitivă a mapării în cazul existenței izomorfismului prin colorarea nodurilor.

5.3 Arhitectura aplicației

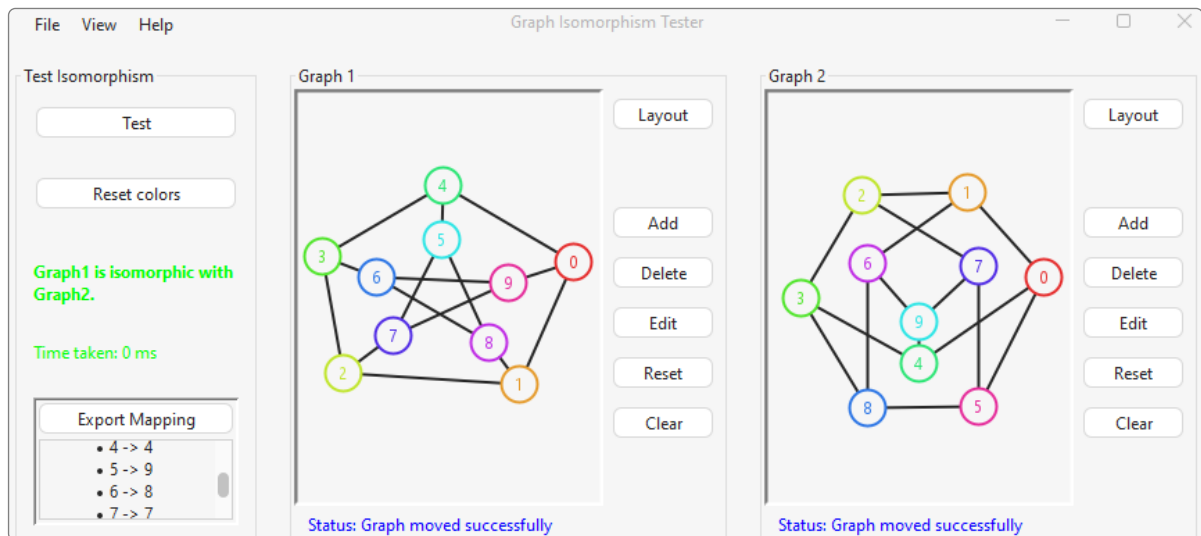


Figura 5.1: Imagine de ansamblu a aplicației GUI

După cum se poate vedea din figura (Fig. 5.1), interfața este constituită din următoarele mari componente:

- o bară de meniu;
- două panouri pentru editarea celor 2 grafuri;
- un panou pentru testare.

5.3.1 Bara de meniu

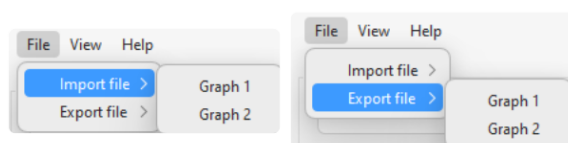


Figura 5.2: Import și export grafuri

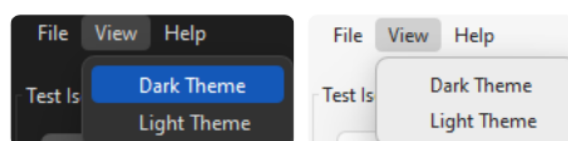


Figura 5.3: Look-and-feel

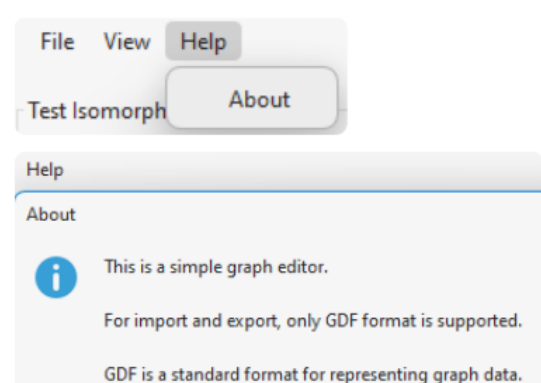


Figura 5.4: Informații despre formatul de import, folositoare utilizatorului.

Bara de meniu conține următoarele opțiuni:

- File → Import → Graph1 / Graph2: permite importarea unui graf în interfață;
- File → Export → Graph1 / Graph2: permite exportarea unui graf din interfață;
- View → Light / Dark: permite schimbarea **look and feel**-ului;
- Help → About: descrie pe scurt formatul de import/export al grafurilor. Formatul de import/export va fi descris în secțiunea următoare.

5.3.2 Panoul de editare

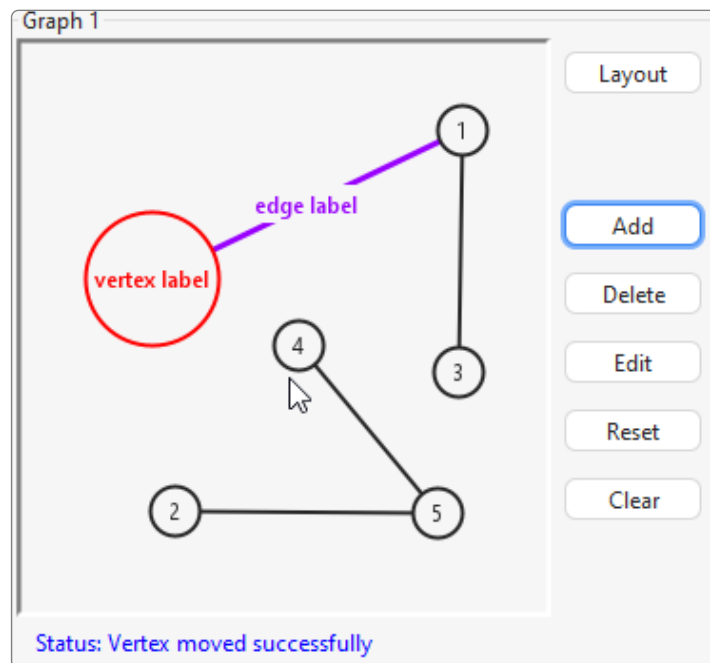


Figura 5.5: Panoul de editare al grafurilor

Panoul de editare (Fig. 5.5) al grafurilor este format din:

- un panou de desenare a grafurilor;
- un panou de selectare a operațiunii care se dorește a se realiza;
- un panou pentru afișarea statusului operațiunii curente (ex.: nod creat cu succes).

Operațiunile sunt:

- **ADD**: atunci când această operațiune este selectată, fiecare **click** stânga al mouse-ului în panoul de desenare va avea ca efect crearea unui nou nod. Dacă se face

click pe un nod existent, atunci se înțelege că vrem să adăugăm o muchie și va trebui să selectăm și pe cel de-al doilea nod.

- **DELETE:** prin click stângă selectăm nodul/muchia pentru a fi eliminat/ă.
- **EDIT:** prin click stângă selectăm un nod/muchia, după care se va deschide o fereastră pentru a modifica proprietățile lui/ei.

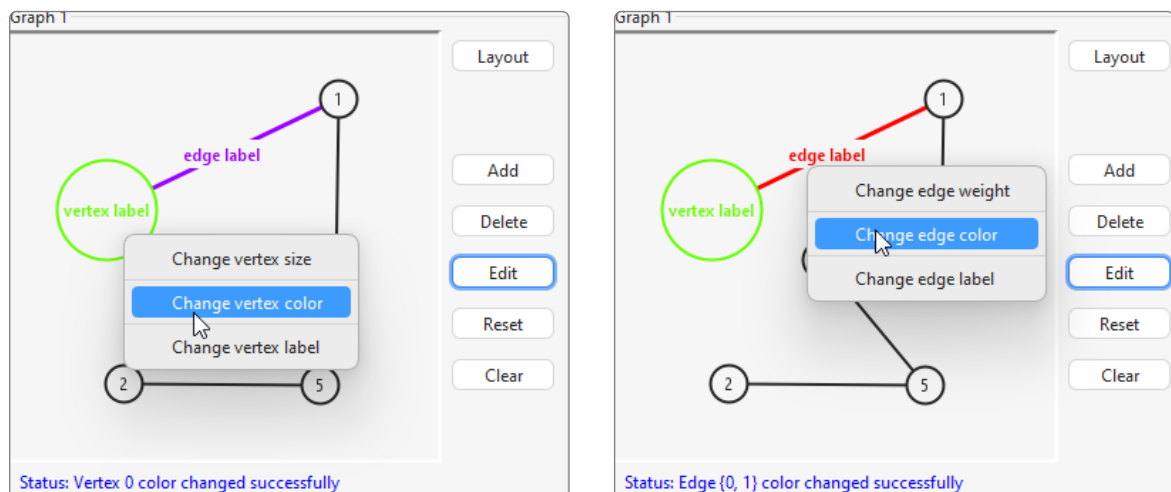


Figura 5.6: Editarea proprietăților nodurilor și muchiilor

- **RESET:** va reseta la valorile default proprietățile nodurilor/muchiilor.
- **CLEAR:** va elimina întregul graf.
- **LAYOUT:** va lista o serie de layout-uri care vor rearanja pozițiile nodurilor.

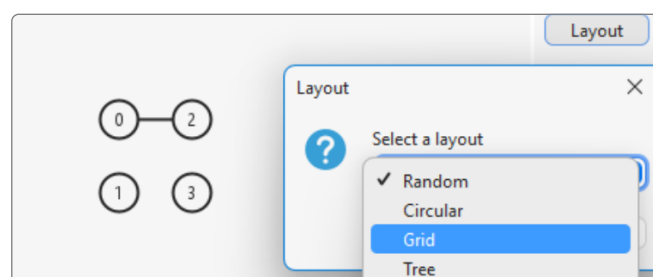


Figura 5.7: Aranjarea nodurilor într-o anumită formă

În plus, cu ajutorul mouse-ului putem realiza următoarele:

- **ZOOM IN/OUT:** rotind de rotația mouse-ului.
- **MOVE:** prin click dreapta putem muta poziția unui nod, a unei muchii sau a întregului graf.

5.3.3 Panoul de testare

Panoul de testare este format dintr-un buton **Test**, care va porni rularea algoritmului de testare al izomorfismului dintre **Graph1** și **Graph2**. În funcție de rezultatul testului, acesta va afișa următoarele:

- test negativ: mesaj că cele 2 grafuri nu sunt izomorfe și timpul de execuție.
- test pozitiv: mesaj că cele 2 grafuri sunt izomorfe, timpul de execuție și maparea efectivă plus posibilitatea de a exporta maparea într-un fișier text.

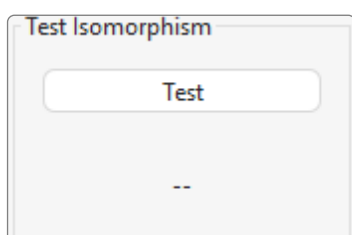


Figura 5.8: Panoul de testare

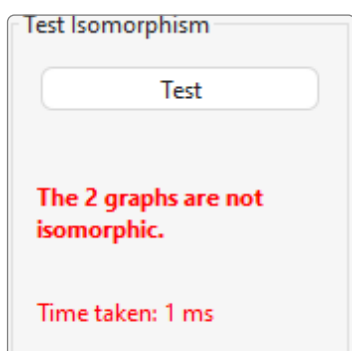


Figura 5.9: Test negativ

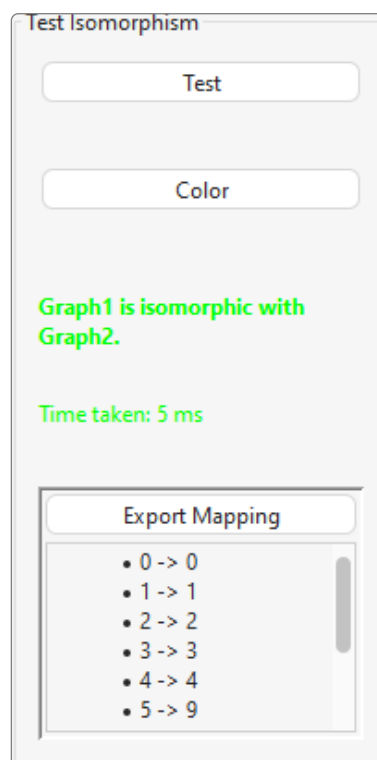


Figura 5.10: Test pozitiv

5.4 Formatul standard GDF

Am menționat anterior posibilitatea de a importa/exporta grafuri în/din interfața grafică.

Deoarece intenția aplicației a fost simplitate, intuitivitate, am decis să folosesc un format cât mai ușor de înțeles și de modificat de către un utilizator uman, dar care să fie destul de expresiv, adică să permită specificarea unor atribute precum culoare, rază, pondere și altele.

Prin urmare, am considerat că cel mai potrivit pentru acest scop este formatul standard **GDF**(Graph Data Format), un format text, simplu, propus și utilizat de **GU-ESS** [23], o aplicație de vizualizare și editare a grafurilor. O altă aplicație foarte populară care folosește acest format, printre multe altele, este **Gephi** [24].

Formatul GDF este construit similar unui fișier CSV. Un fișier GDF este împărțit în două secțiuni, o secțiune pentru definirea nodurilor și o secțiune pentru definirea muchiilor. Fiecare secțiune are un antet în care sunt specificate numele atributelor și tipul acestora. [25]

```
1 # comentariu: graf cu 4 noduri si o muchie
2 nodedef>name INTEGER,label VARCHAR,x DOUBLE,y DOUBLE,color VARCHAR,width DOUBLE
3 0,'vertex label',101.0,139.0,'255,0,0',35.0
4 1,'',123.0,237.0,,13.0
5 2,'',312.0,118.0,,13.0
6 3,'',267.0,264.0,,13.0
7 edgedef>node1 INTEGER,node2 INTEGER,weight DOUBLE,label VARCHAR,color VARCHAR
8 0,2,1.0,'edge label','204,51,255'
```

GUESS propune în manualul său o serie de attribute default([26]), însă pentru aplicația noastră am păstrat doar următoarele:

Pentru noduri, avem definite următoarele attribute:

1. **name** → **INTEGER**: reprezintă identificatorul unui nod dintr-un graf al bibliotecii **Graph4J**.
2. **label** → **VARCHAR**: eticheta nodului.
3. **x** → **DOUBLE**: coordonata x în sistemul de coordonate din **Java Swing**.
4. **y** → **DOUBLE**: coordonata y în sistemul de coordonate din **Java Swing**.
5. **color** → **VARCHAR**: culoarea specificată în formatul ' r, g, b ', unde $r(g,b)$ sunt numere din intervalul $[0, 255]$.
6. **width** → **DOUBLE**: raza cercului corespunzător unui nod în vizualizatorul de grafuri.

Pentru muchii, avem definite următoarele attribute:

1. **node1** → **INTEGER**: identificatorul primului nod.

2. **node2** → **INTEGER**: identificatorul celui de al doilea nod.
3. **label** → **VARCHAR**: eticheta muchiei.
4. **color** → **VARCHAR**: culoarea specificata în formatul '*r, g, b*'.
5. **weight** → **DOUBLE**: ponderea muchiei.

În cele din urmă am construit un parser pentru acest format, care va returna un obiect de tipul **GDFGraph**, detaliat în figura (Fig. 5.9):

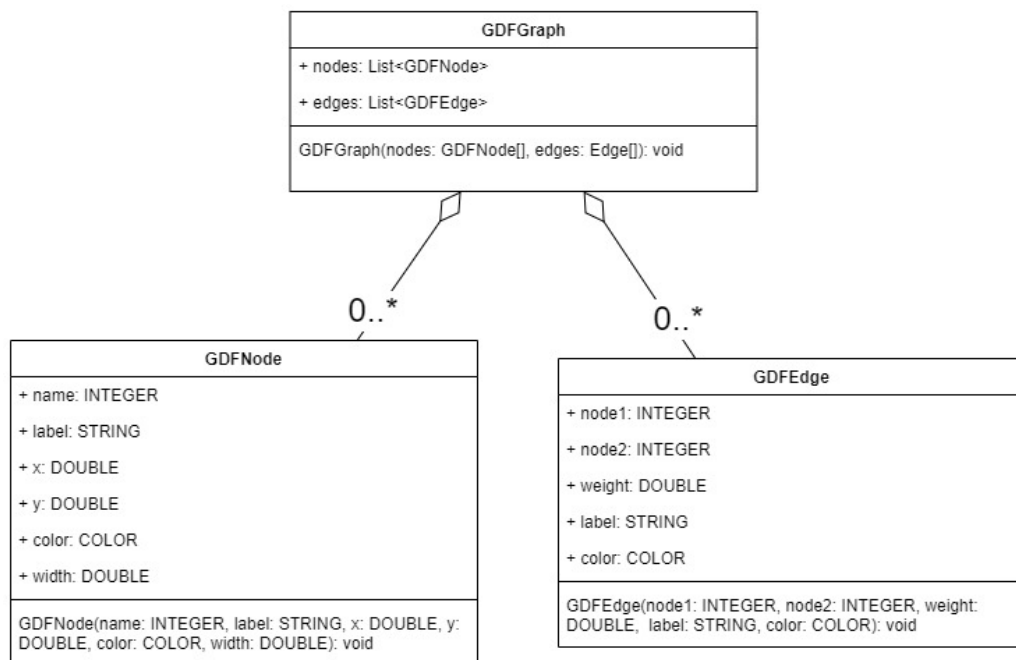


Figura 5.11: Diagrama de clase a modelului utilizat pentru parserul formatului GDF

Folosind un astfel de obiect, putem apoi foarte ușor să-l importăm în modelul din interfața grafică. Tot la fel de ușor se poate implementa și procesul invers, de exportare, mai întâi creând din model un obiect de tip **GDFGraph**, urmată de o scriere simplă după formatul descris anterior.

Concluzii

Această lucrare surprinde aspectele teoretice și practice ale problemei izomorfismului grafurilor. În cadrul ei am descris o serie de algoritmi pentru cazul general al problemei, cu atenție specială asupra implementării algoritmilor Ullman și VF2. De asemenea, a fost studiat și izomorfismul între arbori, pentru care am văzut că există un algoritm de complexitate timp liniară.

Toți algoritmi implementați au fost supuși unor teste de performanță, care au confirmat eficiența lor net superioară algoritmilor similari din alte biblioteci de grafuri.

Contribuții personale:

- am implementat algoritmi Ullman, VF2 și cel pentru arbori în biblioteca de grafuri **Graph4J**;
- spre deosebire de alte biblioteci, algoritmi implementați tratează toate tipurile de grafuri, inclusiv pseudografuri și multigrafuri;
- am ajutat la remedierea unor greșeli, probleme din cadrul bibliotecii Graph4J;
- am realizat o interfață grafică Swing pentru vizualizarea, editarea grafurilor și testarea facilă a algoritmilor implementați.

Îmbunătățiri ce pot fi aduse:

În faza incipientă testării izomorfismului pot fi verificate anumite proprietăți, precum secvența gradelor nodurilor, numărul de triunghiuri, etc. Toate aceste proprietăți trebuie să fie egale pentru a exista izomorfismul între cele două grafuri.

Pe lângă aceste proprietăți uzuale, am putea folosi un algoritm de reprezentare canonică a grafurilor, numit **Weisfeiler-Leman** [27]. Dacă reprezentările canonice ale grafurilor sunt egale, atunci cele două grafuri pot fi izomorfe, altfel cu siguranță grafurile nu sunt izomorfe.

Direcții de viitor:

- studierea detaliată și implementarea unor algoritmi mai eficienți, precum VF3 și VF3-light.
- cercetarea izomorfismului **inexact** al grafurilor. În această lucrare s-a studiat izomorfismul **exact** al grafurilor în care testăm existența unei mapări unu-la-unu a nodurilor.

Însă în majoritatea aplicațiilor practice precum recunoașterea de obiecte sau validarea amprentelor, poate apărea fenomenul denumit **zgomot**. În acest caz găsirea unei mapări unu-la-unu este aproape imposibilă. Prin urmare, se dorește găsirea celei mai bune mapări prin diferite mijloace: învățare automată, tehnici de comprimare a grafurilor.

În concluzie, prin prezenta lucrare am urmărit definirea unei baze pentru înțelegerea dimensiunii problemei izomorfismul grafurilor, a importanței sale teoretice și practice, precum și a soluțiilor propuse de-a lungul timpului. Aceasta poate reprezenta punctul de plecare al unei cercetări viitoare, mult mai practice, anume problema izomorfismului inexact.

Bibliografie

- [1] Uwe Schöning, “Graph isomorphism is in the low hierarchy”, în *Journal of Computer and System Sciences* 37.3 (1988), pp. 312–323, ISSN: 0022-0000, DOI: [https://doi.org/10.1016/0022-0000\(88\)90010-4](https://doi.org/10.1016/0022-0000(88)90010-4), URL: <https://www.sciencedirect.com/science/article/pii/0022000088900104>.
- [2] J. R. Ullmann, “An Algorithm for Subgraph Isomorphism”, în *Journal of the ACM* 23.1 (1976), pp. 31–42, DOI: <https://doi.org/10.1145/321921.321925>.
- [3] *Maclaurin Series*, URL: <https://mathworld.wolfram.com/MaclaurinSeries.html>.
- [4] Carlo Sansone Luigi P. Cordella Pasquale Foggia și Mario Vento, “A (sub)graph isomorphism algorithm for matching large graphs”, în *IEEE Trans. on Pattern Analysis and Machine Intelligence* 26.10 (2004), pp. 1367–1372, DOI: <https://doi.org/10.1109/TPAMI.2004.75>.
- [5] Carlo Sansone Luigi P. Cordella Pasquale Foggia și Mario Vento, “Performance evaluation of the VF graph matching algorithm”, în *Proceedings 10th Int’l Conf. on Image Analysis and Processing* (1999), pp. 1172–1177, DOI: <https://doi.org/10.1109/ICIAP.1999.797762>.
- [6] Vincenzo Carletti, Pasquale Foggia și Mario Vento, “VF2 Plus: An Improved version of VF2 for Biological Graphs”, în *Graph-Based Representations in Pattern Recognition*, ed. de Cheng-Lin Liu et al., Cham: Springer International Publishing, 2015, pp. 168–177, DOI: https://doi.org/10.1007/978-3-319-18224-7_17.
- [7] Alpár Jüttner și Péter Madarasi, “VF2++ — An improved subgraph isomorphism algorithm”, în *Discrete Applied Mathematics* 242.10 (2018), pp. 69–81, DOI: <https://doi.org/10.1016/j.dam.2018.02.018>.

- [8] Vincenzo Carletti et al., “Introducing VF3: A New Algorithm for Subgraph Isomorphism”, în *Graph-Based Representations in Pattern Recognition*, ed. de Pasquale Foggia, Cheng-Lin Liu și Mario Vento, Cham: Springer International Publishing, 2017, pp. 128–139, DOI: https://doi.org/10.1007/978-3-319-58961-9_12.
- [9] Vincenzo Carletti et al., “VF3-Light: A lightweight subgraph isomorphism algorithm and its experimental evaluation”, în *Pattern Recognition Letters* 125 (2019), pp. 591–596, ISSN: 0167-8655, DOI: <https://doi.org/10.1016/j.patrec.2019.07.001>.
- [10] Brendan D. McKay și Adolfo Piperno, “Practical graph isomorphism, II”, în *Journal of Symbolic Computation* 60 (2014), pp. 94–112, ISSN: 0747-7171, DOI: <https://doi.org/10.1016/j.jsc.2013.09.003>.
- [11] László Babai, “Graph isomorphism in quasipolynomial time”, în *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, 2016, pp. 684–697.
- [12] Eugene M. Luks, “Isomorphism of graphs of bounded valence can be tested in polynomial time”, în *Journal of Computer and System Sciences* 25.1 (1982), pp. 42–65, ISSN: 0022-0000, DOI: [https://doi.org/10.1016/0022-0000\(82\)90009-5](https://doi.org/10.1016/0022-0000(82)90009-5), URL: <https://www.sciencedirect.com/science/article/pii/0022000082900095>.
- [13] Alfred V. Aho, John E. Hopcroft și Jeffrey D. Ullman, “The Design and Analysis of Computer Algorithms”, în 1974, URL: <https://api.semanticscholar.org/CorpusID:29599075>.
- [14] *TIOBE Index for May 2024*, URL: <https://www.tiobe.com/tiobe-index/>.
- [15] *Why Java is So Popular?*, URL: <https://www.javatpoint.com/why-java-is-so-popular>.
- [16] Emanuel Florentin Olariu Cristian Frăsinaru, “Graph4J – A computationally efficient Java library for graph algorithms”, în (2023), DOI: <https://doi.org/10.48550/arXiv.2308.09920>.
- [17] Barak Naveh Dimitrios Michail Joris Kinable și John V Sichi, “JGraphT – A Java library for graph data structures and algorithms”, în 46.2 (2020), DOI: <https://doi.org/10.1145/3381449>.

- [18] Yanhong A. Liu și Scott D. Stoller, "From recursion to iteration: what are the optimizations?", în *SIGPLAN Not.* 34.11 (Nov. 1999), pp. 73–82, ISSN: 0362-1340, DOI: <https://doi.org/10.1145/328691.328700>.
- [19] Poonam Parhar Charlie Hunt Monica Beckwith și Bengt Rutisson, *Java Performance Companion*, Addison-Wesley Professional, 2016, ISBN: 978-0133796827.
- [20] *Erdős–Rényi model*, URL: https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model.
- [21] *Barabási–Albert model*, URL: https://en.wikipedia.org/wiki/Barab%C3%A1si%E2%80%93Albert_model#cite_note-Barabasi1999-17.
- [22] *Introducing Swing*, URL: https://dducollegedu.ac.in/Datafiles/cms/ecourse%20content/chapter31_swing.pdf.
- [23] *GUESS: The Graph Exploration System*, URL: <http://graphexploration.cond.org>.
- [24] *Gephi - The Open Graph Viz Platform*, URL: <https://gephi.org/>.
- [25] *GDF Format*, URL: <https://gephi.org/users/supported-graph-formats/gdf-format/>.
- [26] *GUESS manual*, URL: <http://graphexploration.cond.org/manual.pdf>.
- [27] Adrien Leman, "THE REDUCTION OF A GRAPH TO CANONICAL FORM AND THE ALGEBRA WHICH APPEARS THEREIN", în 2018, URL: <https://api.semanticscholar.org/CorpusID:49579538>.