UNIVERSITATEA POLITEHNICA BUCUREŞTI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE DIPLOMĂ

Potrivirea în grafuri pentru recunoașterea contextului

Coordonatori ştiinţifici:
Prof. dr. ing. Adina Magda Florea
As. dr. ing. Andrei Olaru

Absolvent:
Marius Adrian Dobrescu

BUCUREŞTI

2012

Cuprins

Rezumat: *Inteligența Ambientală, din perspectiva calculatoarelor, este percepută ca fiind domeniul tuturor mijloacelor electronice care reacționează în preajma oamenilor și chiar anticipează situațiile viitoare. Dispozitivele sunt adaptive în timp, ajutându-ne în activitățile zilnice. Principiile întâlnite în Inteligența Ambientală evoluează în jurul unor cuvinte cheie precum: procesarea omniprezentă sau raționarea pe baza contextului. Această lucrare își propune să analizeze acel nivel dintr-un sistem inteligent în care o latură esențială din structura perceptivă și decizională a unei entități inteligente este recunoașterea unei situații. Se consideră contextul, fiind reprezentat printr-un graf orientat iar atenția se va concentra asupra algoritmilor de potrivire de grafuri, care identifică o situație din context pe baza unui graf șablon.*

Cuvinte cheie*: graf contextual, graf șablon, subgraf comun, clică maximală, Zest, Jung, satisfacerea restricțiilor, izomorfism*

1. Introducere

1.1 Motivație

Inteligența ambientală – sau **AmI** (Ambient Intelligence) este un termen tot mai des întâlnit în domeniul CTI, devenind un punct major de interes în cercetare. La ora actuală, putem spune că reprezintă noua frontieră în tehnologie, agregând capacitățile și beneficiile care pot fi introduse în activitățile zilnice ale oamenilor. Toate ramurile AmI reprezintă un ansamblu vast de dispozitive și concepte, de la senzori microscopici care monitorizează starea unui sistem până la serviciile web care au intrat în rutina zilnică a utilizatorilor. Efortul investit în cercetare și dezvoltare a fost unul foarte mare, de ani și ani, atât în posibilitățile tehnice cât și în amănunte mici dar critice cum ar fi estetica dispozitivelor și strategia lor de funcționare.

Oricâte impedimente ar bloca progresul de a implementa pe scară largă un univers inteligent, luând în calcul fezabilitatea, utilitatea și efortul investit, termenul generic de inteligență ambientală rămâne unul de nivel superior în tehnologie. Oamenii nu se vor opri niciodată să creeze dispozitive inteligente care să se integreze în rutina lor zilnică, motivația fiind una indiscutabilă – este spectaculos și în primă instanță **util**.

1.2 Obiective

Proiectele de inteligență ambientală folosesc foarte des noțiunea de “context”, deoarece o situație în care un sistem trebuie să ia o decizie pentru a face o acțiune este definită de un context. O bună reprezentare pentru descrierea contextului este un graf.[7] Într-adevăr, prin intermediul acestei structuri putem reprezenta orice concept ca fiind un nod și orice relație între două concepte ca fiind o muchie între două noduri. În această lucrare vom încerca să definim formal și matematic acest tip de graf pentru reprezentarea și manipularea contextului. De asemenea, ne propunem să studiem ce algoritmi putem folosi pentru recunoașterea diferitelor situații derivate din contextul inițial, atunci când el este reprezentat ca un graf.

În plus, pe lângă a avea o bază teoretică solidă care să fie un fundament în justificarea folosirii grafurilor contextuale, scopul acestui proiect și contribuția proprie însemnată se axează pe implementarea unei platforme de testare a algoritmilor discutați în lucrare și pe adaptarea acestora la problema grafurilor contextuale. Prin urmare este necesar să creăm un benchmark variat pe baza căruia algoritmii pot fi comparați în diferite aspecte ale particularităților lor (cazuri favorabile, defavorabile, limitări și cazuri pentru care unii algoritmi nu pot determina soluții valide).

1.3 Structura lucrării

În capitolele următoare vom aborda contribuțiile personale aduse la potrivirea grafurilor contextuale, analiza diverșilor algoritmi existenți pentru lucrul cu grafuri cât și modificările teoretice aduse acestor algoritmi pentru a fi utilizați în proiectul nostru. De asemenea, vom studia corectitudinea metodelor utilizate.

În capitolul 3 vor fi prezentate noțiuni generice despre grafurile contextuale, urmând ca în capitolul 4 să fie abordați algoritmii implementați pentru recunoașterea contextului. Ei vor fi studiați din punct de vedere teoretic. În capitolele 5 și 6 va fi prezentată platforma de testare respectiv rezultatele obținute. La final, în capitolul 7 vom trage concluziile desprinse din acest studiu cât și ce se poate aduce nou în această ramură a inteligenței ambientale.

1.4 Cadrul proiectului

Lucrarea curentă tratează potrivirea de grafuri contextuale, dintr-o perspectivă care este oarecum în corelație cu lucrarea lui **Mark Weiser** [2] : "The Computer for the 21th Century". Acesta introduce termenul de "Ubiquitous Computing". Ideea de bază ce trebuie avută în vedere este reprezentată de proprietatea mini-dispozitivelor de a nu ieși în evidență, de a se pierde în peisajul zilnic și de a se integra în viața noastră pentru a ne ajuta să facem alte activități. Imaginea este în analogie cu un om care știe să citească și extrage informația din simbolurile vizuale într-un mod inconștient. Așadar, tehnologia trebuie să fie mică, performantă, precisă și fiabilă. Aici intervin problemele, identificate și dezbătute în mulțimea de lucrări de specialitate.

Această lucrare face parte dintr-un proiect mai amplu, pornit în timpul tezei "**Un sistem multi-agent dependent de context pentru medii de Inteligență Ambientală**" [1], a lui Andrei Olaru. Aici se va extinde latura legată de nivelul superior al unui sistem și anume - percepția contextului. Se vor analiza diverși algoritmi pentru identificarea contextului, când acesta este reprezentat ca un "graf contextual", prezentat pe larg în lucrarea [1].

Nu ne putem gândi la Inteligența Ambientală, fără să nu facem referire la agenți. Ei aduc anumite caracteristici, de care este foarte mare nevoie, precum reactivitatea, autonomia, pro-activitatea, raționamentul și anticiparea. Ideea de a folosi agenți nu este nouă. Una dintre abordările care descriu organizarea lor constă în folosirea unui număr mic de agenți, care solicită informații contextuale și răspund rapid la schimbări. Acest model, în schimb, nu este foarte flexibil. Cealaltă abordare se bazează pe colaborarea și organizarea pe parcurs și de sine stătătoare a agenților. Dezavantajul este că aceștia nu mai au reprezentare flexibilă și puternică a contextului[1]. Noi ne vom concentra asupra felului în care contextul este perceput și recunoscut și asupra modului în care agentul reacționează la identificarea unei noi situații [1, 7].

Dacă e să privim sistemul inteligent ca pe o aplicație cu multe niveluri, putem considera că trei dintre ele sunt cu adevărat importante: aplicația, mijloacele de comunicare cu hardware-ul sau cu alți agenți din sistem, și hardware-ul [1]. Din punct de vedere al comunicațiilor și al hardware-ului efectiv, putem spune că lucrurile sunt clare de mulți ani și performanțele tehnologice sunt suficiente pentru a reprezenta o bază de dezvoltare a multor agenți interesanți. La nivelul aplicației, atât conceptual cât și matematic, există, de asemenea, o bază solidă. Algoritmii, deși au fost implementați și au avut rezultate bune, sunt limitați în a lua anumite decizii într-un anumit context, nereprezentând o rețetă generală, despre care să spunem cu adevărat că este performantă. Cealaltă problemă este adusă de complexitatea aplicațiilor și cantitatea mare de date care trebuie procesate pe un dispozitiv de dimensiuni mici, luând în considerare și faptul că procesele respective trebuie să fie transparente pentru om [2].

Proiectul de față țintește nivelul aplicației și își propune să determine ce algoritmi sunt cei mai potriviți și în ce situații, pentru un agent care identifică un șablon și trebuie să ia acțiunea cea mai potrivită pe baza informațiilor cunoscute. Universul de cunoștințe, cât și situația curentă în care este pus agentul, pot fi reprezentate printr-un context format din morfisme între 2 entități. În cazul de față contextul este reprezentat ca un graf orientat.

2. Potrivirea grafurilor

Acest concept a început să ia amploare pe la sfârșitul anilor ’70. Atunci au fost introduse grafurile, ca o unealtă puternică în rezolvarea problemelor de recunoaștere a modelelor sau **PR** - (Pattern Recognition) [3].

După această perioadă, interesul pentru folosirea grafurilor în probleme de PR a scăzut iar motivul principal este unul de înțeles: efortul computațional foarte mare, necesar pentru a aborda majoritatea algoritmilor folosiți în a determina asemănări între grafuri, deoarece problemele pe care aceștia le țintesc sunt NP-complete. În ultimii ani, în schimb, a apărut din nou un interes crescut pentru folosirea grafurilor în PR [3]. Deși efortul computațional a rămas semnificativ, optimizările descoperite și puterea de procesare mult mai mare din prezent, au făcut ca grafurile să fie folosite din nou pe scară largă în foarte multe domenii, inclusiv în inteligența ambientală.

Prin grafuri se pot modela rețele de agenți, ontologii și multe forme de reprezentare a informației. În principiu sunt 3 mari clasificări în care potrivirea de grafuri este esențială [1]:

- taxonomia algoritmilor de potrivire a grafurilor
- taxonomia tehnicilor care folosesc potrivirea de grafuri în PR.
- ontologiile reprezentate ca grafuri.

Dacă ar fi să definim ce înțelegem prin potrivirea de grafuri, am putea spune că este un proces de găsire a unei corespondențe între nodurile și muchiile unui graf, în care se satisfac anumite constrângeri (mai mult sau mai puțin stricte), asigurând că anumite substructuri dintr-un graf sunt mapate la substructuri similare din celălalt graf. Atenția noastră se va concentra pe taxonomia algoritmilor folosiți la potrivirea de grafuri.

Putem împărți metodele de potrivire a grafurilor în 2 categorii majore [3]:

- potrivire exacta, atunci când structurile trebuie să fie identice.
- inexactă, atunci când o potrivire poate să fie validă chiar dacă cele două entități sunt diferite cu o anumită marjă.

Particularitatea cea mai interesantă la grafurile contextuale constă în modul cum este construit un graf șablon, care se dorește a fi identificat. Problema nu se reduce doar la găsirea celui mai mare subgraf comun dar și la identificarea unor noduri necunoscute care sunt etichetate special. Vom discuta mai pe larg acest subiect în capitolul 3, unde vom introduce grafurile contextuale.

2.1 Potrivirea exactă

2.1.1 Algoritmi de căutare prin parcurgere arborescentă

Algoritmii de căutare trebuie să păstreze corespondența muchiilor, în sensul în care două noduri sunt conectate în primul graf dacă și numai dacă ele corespund în al doilea graf, conectate printr-o muchie care reprezintă aceeași relație definită peste mulțimea de relații [3].

Potrivirile exacte, în funcție de gradul de strictețe sunt:

- **izomorfisme** - relație bijectivă între nodurile din primul graf și cele din al doilea graf, în urma unei potriviri maximale.
- **monomorfisme** – relație surjectivă, unde poate fi o corespondență mai mulți la unu.
- **omomorfisme** – care restrâng condiția astfel încât nodurile din primul graf trebuie să fie mapate la noduri distincte din al doilea graf.

Mulți algoritmi se bazează pe faptul că anumite tipuri de grafuri prezintă proprietatea prin care o clică maximală în graful obținut din produsul modular al celor 2 grafuri, corespunde printr-o relație bijectivă celui mai mare subgraf comun căutat.

2.1.2 Alte tehnici care nu presupun explorarea grafurilor

O atenție specială aici, trebuie atribuită algoritmului **Nauty**,[4] creat de Brendan D. McKay. Acesta nu folosește parcurgeri arborescente clasice precum: limitarea spațiului de căutare sau eliminarea ramurilor care conduc la un rezultat nesatisfăcător. El folosește proprietățile de izomorfism și teoria grupurilor. Este recunoscut în prezent ca fiind unul dintre cei mai rapizi algoritmi în practică, în potrivirea de grafuri.

Algoritmul are la bază doi pași:

1) Se determină în timp exponențial automorfismul grupurilor.
2) Se verifică echivalența matricelor de adiacență ale celor 2 grafuri obținute, cu o complexitate $O(N^2)$.

Deoarece nu se poate adapta ușor la grafurile contextuale și a fost conceput astfel încât să lucreze pe grafuri numerice și cu preponderență neorientate (cu matrice de adiacență), nu va fi tratat în lucrarea curentă.

2.1.3 Tipuri speciale de grafuri

Această categorie se adresează mai mult algoritmilor care fac potrivire de imagini sau video.[3] Nu este interesantă din punctul nostru de vedere, deoarece proiectul se axează pe potrivirea grafurilor contextuale care au anumite proprietăți specifice.

2.2 Potrivire inexactă

În cazul grafurilor contextuale ne interesează să găsim o potrivire cât mai bună (maximală ca număr de muchii) între șablon și contextul cunoscut la un anumit moment de timp. Cu toate acestea, algoritmii de potrivire exactă cât și cei care acceptă potriviri parțiale pot fi adaptați problemei, cu anumite ajustări.

2.2.1 Căutare prin parcurgere

Se pot folosi algoritmi de tipul ‘branch and bound’ pentru a găsi o soluție optimă într-un timp convenabil, pentru a elimina ramurile care conduc la soluții mai proaste. Cealaltă metodă este de a folosi algoritmi de tipul A*. Dacă euristica folosită oferă o bună estimare a costului viitor al unei căi, atunci o soluție foarte bună este găsită rapid [3].

2.2.2 Optimizare continuă

Ideea interesantă din spatele acestei categorii este schimbarea modului de a privi problema potrivirii de grafuri, dintr-o reprezentare discretă într-una continuă. Algoritmii trebuie să evite maximele locale triviale, iar ei, în ansamblu, nu garantează soluția optimă. Soluția găsită trebuie convertită la loc din spațiul continuu în cel discret, ceea ce poate aduce un nivel suplimentar de aproximație [3].

Avantajul acestei abordări constă în reducerea efortului computațional, la o complexitate polinomială cu un exponent redus, ce depinde de mărimea grafului.

2.2.3 Metode spectrale

Se bazează pe următoarea observație: valorile proprii și vectorii proprii ai unei matrice de adiacență a unui graf sunt invariante cu permutarea nodurilor [3], care e o transformare liniară. Așadar, dacă două grafuri sunt izomorfe, matricele lor de adiacență vor avea aceiași vectori proprii și aceleași valori proprii. Din nefericire, reciproca nu este validă. Cu toate acestea calculul vectorilor proprii, se efectuează în timp polinomial ceea ce reprezintă un interes pentru a considera acești algoritmi în anumite tipuri de probleme.

Pentru grafurile contextuale nu prezintă o importanță mare deoarece metoda este pur structurală, și nu poate exploata atributele nodurilor/muchiilor.

3. Grafurile contextuale

Observăm în figurile de mai jos un exemplu de utilizare a grafurilor contextuale. Figurile sunt inspirate din [1] și [7].

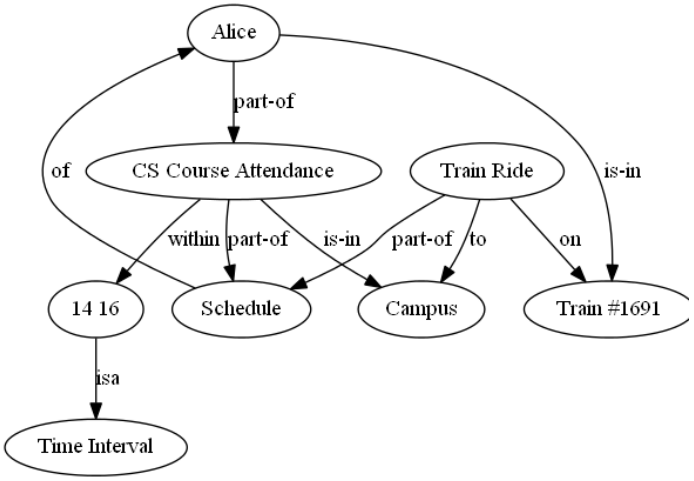


Fig 1.Graf contextual care descrie situația agentului

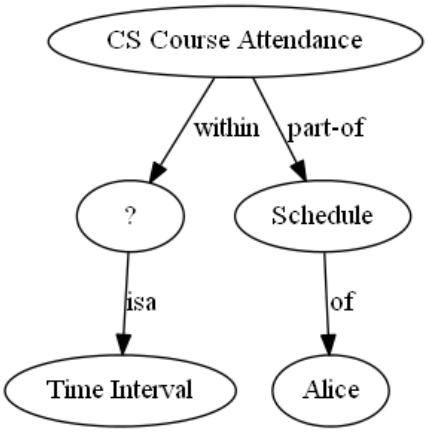


Fig 2. Graful șablon care va fi identificat

3.1 Graful contextual - Definiție

Acest tip particular de graf se definește în modul următor [1]:

CG = (V, E) care conține informație relevantă funcției sale, unde: V ∈ *Concepte* și E = {muchie | muchie = { (nod sursa, nod destinație, valoare) | nod sursa, nod destinație ∈ *Concepte* și valoare ∈ *Relații* } [2].

Este de remarcat faptul că mulțimea *Concepte* este constituită din șiruri de caractere, iar elementele mulțimii *Relații* – pot fi șiruri de caractere, pot fi vide în cazul când nu pot fi etichetate într-un mod anume.

Un agent are mai multe grafuri șablon, având anumite noduri necunoscute, etichetate cu „?”. Acestea pot corespunde unor noduri din graful contextual, într-o relație unu la unu. Acestea sunt folosite pentru a identifica o situație nouă, care este doar parțial cunoscută. Situația se dorește a fi cât mai clară și se atinge prin identificarea a cât mai multor relații între conceptele din graful șablon care au corespondent în graful contextual.

3.2 Graful șablon - Definiție

Graful șablon este definit în modul următor [1].

$G_s^P = \{V_s^P, E_s^P\}$, graful este perechea de noduri plus muchii.

$V_s^P = \{v_i^P(\text{etichetă}) \mid \text{etichetă} \in \textit{Concepte} \cup \{?\}\}$, cu condiția că dacă există i ≠ j, a.î. v_i^P.etichetă = v_j^P.etichetă, atunci v_i^P.etichetă = v_j^P.etichetă = “?”, ceea ce înseamnă că nu pot fi mai multe noduri cu aceeași etichetă, dacă ele nu sunt semne de întrebare.

E_s^P = {(nod sursă, nod destinație, etichetă)}, cu:

- nod sursă, nod destinație ∈ V_s^P

Menționăm că am folosit indicele superior *P*, ca notație pentru noțiuni legate de grafurile șablon, care pot conține noduri "?".

3.3 Potrivirea grafurilor contextuale

O **potrivire** [1] între un graf șablon G_s^P și un graf contextual al unui agent A, CG_A poate fi definită:

$M_{A\text{-}si}$ (G_A', G_m^P, G_x^P), unde G_A', G_m^P, G_x^P sunt grafuri cu $G_A' \subset CG_A'$, unde $G_A' = (V', E')$, $G_m^P = (V_m^P, E_m^P)$,

$G_x^P = (V_x^P, E_x^P)$, unde:

$V_m^P \cap V_x^P = \emptyset$, $V_m^P \cup V_x^P = V_s^P$

$E_m^P \cap E_x^P = \emptyset$, $E_m^P \cup E_x^P = E_s^P$.

G_A' reprezintă o potrivire completă pentru porțiunea rezolvată G_m^P (en. matched) a grafului șablon G_s^P. G_x^P, se definește ca fiind ‚problema' și reprezintă partea nerezolvată din G_s^P. Notațiile folosite pentru a defini aceste concepte au fost indroduse în [1].

Algoritmii de potrivire de grafuri au o complexitate mare atunci când pot exista mai multe noduri etichetate la fel. Aparent, în graful contextual, nu are sens să folosim un algoritm exponențial atunci când nodurile sunt unice și aveam garanția ipotezei că sunt unice. Agentul trebuie să valideze acest tip de graf și să nu execute un algoritm cu seturi de date invalide. Problema poate deveni ușor una cu multe noduri, deoarece nodurile etichetate cu "?", în cazul în care nu pot fi rezolvate printr-un algoritm de satisfacere a constrângerilor cu care se poate deduce direct ce etichete corespund acestor noduri, vor fi expandate la noduri multiple.

Soluția constă ori într-o rezolvare cu backtracking, dar nu toți algoritmii permit acest tip de atribuire pe parcurs a etichetelor și este o soluție nefericită care străbate toate domeniile de valori. Cealaltă variantă este expandarea fiecărui nod vi, prin înlocuirea lui cu un set de noduri $v_j...v_k$ și {etichetă(v_j)... etichetă(v_k)}, unde eticheta (v_x) ∈ Concepte(CG_A) j,i ≠ k, cu proprietatea că nu există v_j'… v_k', unde v_j' … v_k' ∈ Concepte(G_s^P) noduri cu aceeași valoare ca $v_j...v_k$.

Pentru rezolvarea acestui tip de graf, ca parcurgere cu căutare arborescentă, se pot folosi algoritmi care compară etichetele direct, precum McGregor [5]. Pot fi folosiți cei care se bazează pe corespondența dintre clică maximală în graful asocierilor și cel mai mare subgraf comun [5, 8, 11].

Altă abordare a problemei constă în îngustarea spațiului de căutare al soluției între limitele asimptotice date de cea mai mare clică în graful *cordal* derivat din graful inițial (sau triangulat, unde orice ciclu de lungime mai mare de 4 are cel puțin o coarda)[5]. Dimensiunea acestei clici reprezintă asimptota inferioară. Asimptota superioară este dată de numărul cromatic al grafului[5]. Numărul cromatic reprezintă numărul minim de culori necesare pentru a colora nodurile unui graf astfel încât oricare două noduri adiacente nu au aceeași culoare.

În [1], Andrei Olaru consideră particularitatea grafurilor șablon de a avea expresii regulate pe muchii. În lucrarea curentă, deoarece expresiile regulate introduc probleme în folosirea și adaptarea algoritmilor clasici, acestea nu vor fi tratate.

3.3.1 Tipuri de potriviri

După gradul de completitudine care ne interesează în identificarea unei situații, potrivirea grafurilor contextuale poate fi de 3 tipuri:

- Potrivire exactă, când toate nodurile și muchiile grafului șablon se regăsesc în graful contextual.

- Potrivire inexactă, atunci când ne interesează cea mai bună soluție posibilă, chiar dacă graful șablon nu este un subgraf inclus în totalitate în graful contextual.
- Găsirea de rezultate parțiale, când nu ne interesează în mod special cea mai bună soluție, dacă aceasta s-ar determină cu un efort computațional foarte mare, sau atunci când ea nu se poate determina.

În lucrarea curentă, vom încerca pe cât posibil să obținem și să formalizăm algoritmi care pot face potriviri inexacte deoarece graful șablon poate fi de orice natură. Nu dorim să existe restricții asupra acestuia dat fiind faptul că într-o aplicație practică, un agent poate fi expus la orice situație care poate avea sau nu legătură cu contextul din universul de cunoștințe al agentului.

4. Algoritmi implementați

Partea interesantă în rezolvarea problemelor potrivirii grafurilor contextuale constă în adaptarea algoritmilor generali folosiți în potrivirea unor grafuri uzuale (cu numere în etichete, ontologii, imagini, etc.). Deși particularitățile grafurilor contextuale par potrivite folosirii algoritmilor obișnuiți întocmai, aceștia trebuie să suporte modificări însemnate pentru a păstra corectitudinea.

Transformările care trebuie efectuate pentru a obține un algoritm corect derivă din anumite diferențe structurale:

- Adaptarea algoritmilor folosiți pentru grafuri neorientate la grafurile contextuale (orientate).
- Folosirea algoritmilor care obțin soluții exacte sau fac potriviri complete pentru a rezolva o problemă în care încearcă să găsească o potrivire parțială.
- Folosirea de algoritmi care merg doar dacă există muchii unice, cu același nume, între două noduri.

Fiecare dintre aceste transformări poate exclude o serie întreagă de algoritmi care devin nepotriviți pentru a fi adaptați pentru problema noastră. În general, dacă specificațiile din ipoteza unui algoritm nu se respectă, el fie nu mai este corect, fie necesită modificări majore pentru a-l modifica astfel încât să se respecte corectitudinea lui. Cealaltă soluție este modificarea problemei pentru a satisface ipotezele în care un algoritm își păstrează corectitudinea. Și această abordare are dezavantajele ei deoarece trebuie să existe un algoritm care poate transforma soluțiile găsite în soluții ale problemei inițiale. De asemenea, trebuie să existe o incluziune a mulțimilor soluțiilor celor 2 probleme în felul următor:

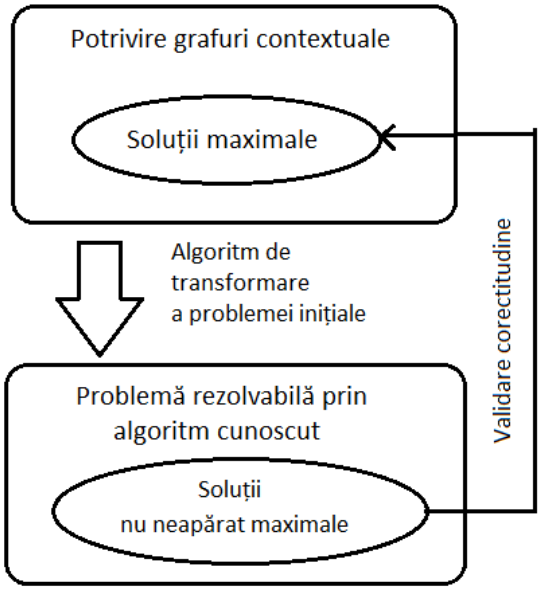


Fig. 3 Tranformarea problemei potrivirii garfurilor contextuale

Notăm cu:

S_M – mulțimea soluțiilor inițiale, maximale ale problemei potrivirii celor 2 grafuri contextuale.

S – mulțimea soluțiilor obținute pentru o problemă rezolvabilă printr-un algoritm aplicabil.

A_t – Algoritmul de transformare a problemei inițiale într-o problemă pe care știm să o rezolvăm.

Algoritmul At este corect dacă și numai dacă $S \supseteq S_M$. Este de notat că proprietatea trebuie să fie adevărată pentru orice soluție S_i, i = 0,card(S), $S_i \in S_M$.

Dacă există cel puțin o soluție $S_k \in S_M$, pentru care nu există nicio soluție $S_k' \in S$ care să o includă, atunci prin invalidarea oricărei soluții $S_i \in S_m$, $i \neq k$, S_k nu va mai fi găsită ca singura soluție maximală. Tehnica aceasta este valabilă doar dacă există o funcție de validarea a incluziunii soluțiilor prin care se poate valida corectitudinea lor. De asemenea trebuie să existe o funcție de transformare dintr-o soluție găsită într-o soluție corectă.

O altă problema de performanță poate apărea este în momentul când încercăm să validăm soluțiile obținute. Această verificare se face în timp polinomial, maxim $O(N_1*N_2) + O(E_1*E_2)$, unde N_1, N_2 sunt numărul de noduri ale CG, respectiv numărul de muchii.

Pentru a rezolva problema printr-un algoritm specific grafurilor neorientate se poate folosi următoarea tehnică:

- Se transformă cele 2 grafuri în grafuri neorientate. Prin urmare, pentru orice muchie e ∈ E, unde E este mulțimea de muchii a unui graf, e = (u, v), cu proprietatea că există muchie de la u la v sau de la v la u, iar cele două sunt mutual exclusive, vom obține e' $\mathfrak{R}$ e, e = (u', v') astfel încât:
  - Există muchie bidirecțională între u' și v'.
  - $\mathfrak{R}$ – este o relație între e și e', caracterizată prin:

$$\begin{cases} \text{etichetă(e)} = \text{etichetă(e')} \\ \text{etichetă(u')} = \text{etichetă(u')} \\ \text{etichetă(v)} = \text{etichetă(v')} \end{cases}$$

Nodurile u și u', respectiv v și v' pot fi atât noduri etichetate cu valori concrete cât și noduri necunoscute (etichetate cu ‚?').

- Se rezolvă problema printr-un algoritm de grafuri neorientate și se găsesc soluții.
  Orice soluție care are potențial de a fi mai bună este evaluată astfel:

- Pentru orice muchie e_1' din graful neorientat, rezultat din graful șablon și e_1' aparține soluției curente, se identifică e_1 – muchie unidirecțională din care a derivat e_1'.
- Se identifică e_2' în graful neorientat derivat din graful contextual, corespunzătoare lui e_1'. Știm sigur că există e_2', deoarece a fost inclusă în soluția găsită. e2' corespunde unei muchii e2 din graful contextual.
- Dacă e_1 și e_2 au aceeași orientare, atunci e_1 face parte dintr-o soluție a problemei noastre și se va stoca în soluție cu orientarea inițială.
- Din mulțimea de muchii obținute în soluția inițială, care după eliminare, poate forma o pădure de subgrafuri, se alege doar cel mai mare subgraf conex.
- Dacă cardinalul mulțimii muchiilor este mai mare decât al altei soluții precedente, aceasta se va stoca și va deveni soluția provizorie cea mai bună.

Particularitatea care face ca algoritmii de potrivire a grafurilor contextuale să fie interesanți este dată de nodurile necunoscute, deoarece ele se pot potrivi cu orice alt nod, iar prin această caracteristică, algoritmii se diferențiază foarte mult. Soluția pentru a transforma graful cu noduri necunoscute într-un graf care se poate rezolva printr-un algoritm obișnuit de potrivire de grafuri este determinată de expandarea nodurilor prin următorul algoritm:

```
procedură  expandează_noduri(Șablon, G)
      pentru fiecare nod N in Șablon
           dacă nodul e necunoscut
               pentru fiecare nod înlocuitor în G
           dacă nodul înlocuitor are cel puțin o muchie
           compatibilă cu N
                creează_nod(nod_nou) cu aceeași etichetă
                duplicareMuchii(nod_nou, N)
```

Problemele care apar cu introducerea nodurilor suplimentare sunt datorate posibilității de a genera o potrivire a muchiilor potențiale, introduse, care se regăsesc în primul graf dar sunt incorecte pentru potrivire. De aceea, în momentul în care se alege un nod care a apărut în urma unei expandări a unui nod necunoscut, trebuie eliminate toate nodurile care au degenerat din acel nod în potrivirea curentă.

În problema noastră ne interesează ca potrivirea găsită să fie maximală, chiar dacă ea este una parțială. Aici se pot considera 2 situații:

- Efortul computațional pentru a găsi cea mai bună soluție este prea mare și se preferă o soluție corectă dar care nu este maximală.
- Găsim soluția maximală, indiferent de efortul de calcul, însă potrivirea este parțială. Graful șablon nu este în totalitate inclus în al doilea graf.

Pentru cea de-a doua situație, dacă am alege o problemă în care se dorește identificarea unei potriviri totale, precum un algoritm de backtracking care validează domeniile de definiție ale variabilelor, atunci această abordare este posibilă. O soluție parțială nu poate fi tratată de un astfel de algoritm deoarece domeniile de definiție ale unor noduri vor deveni nule la un moment dat, ceea ce va invalida soluția parțială găsită.

Având în vedere că potrivirea grafurilor este o problemă NP-completă nu ne putem aștepta să găsim algoritmi cu o limită asimptotică de timp foarte bună, în cel mai defavorabil caz. Cu toate acestea se pot găsi unele mijloace de a reduce complexitatea care ar fi generată de un backtracking exhaustiv. Reducerea complexității prin vizitarea unui număr mai mic de noduri necesită, în general, un timp de execuție crescut pentru menținerea consistenței stării unui nod și refacerea mulțimii de valori ale domeniului din care nodul necunoscut poate fi instanțiat.

Deoarece acești algoritmi au un timp de execuție ridicat, sunt greu de evaluat performanțele individuale ale diverselor abordări sau euristici folosite. Sunt multe situații în care un algoritm poate avea un timp foarte bun de execuție pentru un graf contextual în care ordonarea nodurilor a fost una fericită și un timp foarte prost în rest, pentru grafuri care generează un timp mediu de execuție. Din acest motiv, evaluarea performanțelor trebuie făcută printr-un benchmark. Dacă un benchmark este format dintr-un set larg de instanțe particulare ale problemei, pentru o anumită dimensiune a numărului de noduri, respectiv muchii și un algoritm este mai bun decât ceilalți în majoritatea situațiilor atunci probabilitatea să se comporte în practică mai bine decât ceilalți, pentru probleme generice de potrivire a grafurilor contextuale, este mare.

Algoritmii care se bazează pe metode numerice de rezolvare și transformări ale matricelor de adiacență sunt nepotriviți pentru acest tip particular de grafuri. Se poate alege o sortare lexicografică a nodurilor astfel încât să poată fi trecute printr-un procedeu de numerotare de pe urmă căruia se poate construi o matrice de adiacență. În afară de faptul că grafurile sunt orientate, anumite relații între nodurile necunoscute nu sunt știute de la început, ceea ce ar genera mai multe matrice de adiacență, fiecare corespunzând unei noi probleme. Cealaltă problemă majoră este

prezența muchiilor etichetate. Nu este suficient doar să ținem cont doar de noduri și de prezența unor muchii anonime între ele. Asta ar duce la rezolvarea de probleme care nu au soluție sau au soluții nesatisfăcătoare, pe care un algoritm simplu de backtracking le-ar elimina de la început.

Deoarece nu există niciun benchmark standard pentru evaluarea algoritmilor, este necesar să creăm o plajă variată de exemple prin care se decide cât de potriviți sunt algoritmii. În mod cert, nu există un algoritm 'campion' care să fie cel mai bun iar restul mult mai slabi, deoarece în această situație nu ar mai avea sens folosirea unor alți algoritmi. Este o alegere conștientă a unor diferite metode de rezolvare, prin care diverse grafuri pot translata un algoritm de la o performanță foarte slabă la una foarte bună pentru seturi de date diferite ale unei probleme de dimensiune asemănătoare.

Unele soluții interesante au fost propuse prin tehnici de învățare automată [17]. Acestea pot fi bune în timp pentru o situație concretă în care graful contextual rămâne același și suferă doar modificări ușoare. De aceea necesită revalidarea unui număr mic de noduri care s-au schimbat, pentru situații de potrivire uzuale. Aceste abordări, deși reprezintă o direcție care nu poate lipsi dintr-o aplicație din domeniul inteligenței ambientale, sunt diferite de cea pe care vrem să ne fixăm atenția în această lucrare și anume evaluarea performanțelor diverșilor algoritmi pe grafurile contextuale.

O să vedem cum se schimbă mecanica potrivirii grafurilor și cum au fost transformați unii algoritmi pentru a putea rezolva într-un mod corect potrivirea de grafuri.

4.1 Algoritmul MgGregor

Acest algoritm poate fi descris printr-o reprezentare în spațiul stărilor. O stare poate fi definită prin S = (V, E, dim), unde V e numărul de noduri, E – numărul de muchii și dim = card(V) . V și E formează un graf conex, care este un subgraf, atât pentru CG cât și pentru G_s^P. Acesta este o parte dintr-o posibilă soluție a problemei, dată de cel mai mare subgraf comun. [18]

Pas 1:

Starea inițială a algoritmului este vidă. Două noduri nule sunt analizate și este stocata o soluție cu card(V) = 0 și card(E) = 0.

Pas 2:

Se alege o pereche de noduri. Fie acestea V1 și V2. Se creează nodul V' cu etichetă(V') = etichetă(V1) = etichetă(V2). Daca se poate extinde soluția cu V', adică există cel puțin o muchie E'(V', V_a), pentru care $V_a \in$ V (mulțimii de noduri a soluției anterioare) atunci se va extinde.

Pas 3:

Dacă soluția este mai mare decât un maxim precedent, atunci aceasta se reține ca fiind cea mai bună soluție găsită până la pasul curent.

Pas 4:

Se generează o nouă stare corespunzătoare celei actuale și se continuă cu o căutare în adâncime prin backtracking. Această căutare se desfășoară până când se întâlnește o stare care este frunză. În acest moment se reface starea anterioară și se generează starea următoare, expandând un nou nod valid și adăugându-l la mulțimea de noduri a stării precedente. Este necesar de menționat că dacă o stare nu mai este necesară, va fi eliminată din memorie.

Complexitatea acestui algoritm în cazul cel mai defavorabil aste aceeași ca a produsului cartezian între nodurile primului graf și nodurile celui de-al doilea.

Algoritmul McGregor [5] este prezentat în modul următor:

```
procedură McGregor(s)
    cât timp perecheUrmătoare(s,n1,n2))
      dacă perecheCompatibilă(s,n1,n2)) atunci
         s' = adaugă pereche(s,n1,n2);
         dacă card(s') > subgrafMaximal) atunci
            salveazăSubgrafCurent(s');
            subgrafMaximal = card(s');
         dacă(!frunzăÎnArboreleDeCăutare(s')și
         !condițieEliminare(s')) atunci
            McGregor(s');
         end if
         BackTrack(s');
```

Pentru a adapta acest algoritm la problema grafurilor contextuale, trebuie făcute următoarele transformări:

- Se expandează nodurile necunoscute prin aplicarea procedurii expand_nodes(G), prezentată mai sus. Numărul de noduri al problemei rezultate crește cu maxim produsul dintre
  *nr. noduri necunoscute * nr. noduri care nu sunt prezente în graful șablon.*
- Se elimină muchiile care sunt în plus și care ori nu au corespondent în graful contextual, ori au rămas deconectate în urma eliminării altor muchii.
- Se verifică din nou graful rezultat din expandarea grafului șablon și se păstrează doar cea mai mare componentă conexă, rezultată după eliminarea muchiilor.
- Se aplică algoritmul prezentat mai sus.

Avantajul acestui algoritm constă în claritatea și ușurința implementării. Acesta e potrivit pentru o gamă întreagă de aplicații practice în care agenții recunosc un context relativ mic (până la 15 noduri independente, dacă sunt puține noduri necunoscute). Pentru majoritatea situațiilor este o dimensiune suficientă. Algoritmul, în cazul cel mai defavorabil, explorează un număr de stări egal cu produsul cartezian între noduri. Dacă grafurile cresc în dimensiune sau există foarte multe noduri necunoscute, timpul de execuție al acestui algoritm datorat construirii de stări noi și mulțimi noi la fiecare pas este foarte mare și nu este fezabil într-o aplicație reală.

4.2 Algoritmul Bron-Kerbosch

Implementarea acestui algoritm se reduce la identificarea de clică maximală în graful asocierilor, rezultat prin efectuarea produsului modular între graful contextual și graful șablon. Prin aplicarea produsului modular, se obține un graf G(V, E) cu următoarele proprietăți:

- V = (u, v) , unde u este un nod ce aparține grafului contextual, iar v aparține grafului șablon.
- E = (V1, V2) - generare etichetă unică pe baza etichetelor lui V1 și V2
- Muchiile sunt intre perechi de noduri 'compatibile' (au aceeași etichetă).
- Orice 2 noduri (u, v) si (u', v') sunt adiacente doar daca:
  u adiacent cu u' si v cu v' și muchiile dintre u - u' și v – v' au aceeași etichetă sau u nu e adiacent cu u' si v nu e adiacent cu v'

Mai multe detalii despre produsul modular al două grafuri se pot găsi la [19].

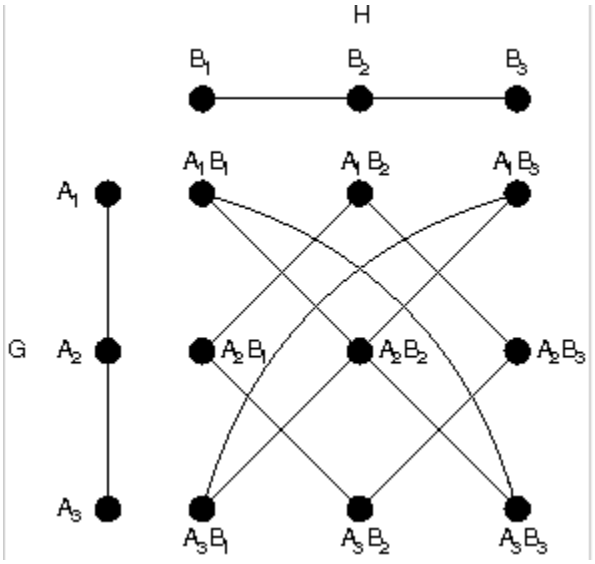


Fig. 4 – Exemplu de graf obtinut prin aplicarea produsului modular. [9]

Este de notat că algoritmul funcționează cu grafuri neorientate și nu este conceput pentru grafuri orientate. Acest neajuns poate fi validat la sfârșit, verificând soluția găsită, eliminând din ea muchiile care nu au aceeași direcție sau etichetă în cele 2 grafuri dar apar în soluția găsită.

```
procedură BronKerbosch(R,P,X)
      dacă P și X sunt ambele vide:
            R este o clică maximală
      sfârșit
      pentru fiacare nod v în P:
            BronKerbosch(R ∪ {v}, P ∩ N(v), X ∩ N(v))
            P := P \ {v}
            X := X ∪ {v}
      sfârșit
sfârșit procedură
```

Prin notațiile folosite se înțelege:

BronKerbosch (R, P, X) – identifică toate clicile maximale care includ toate nodurile din R, unele dintre nodurile din P și niciun nod din X. [5]

Acest algoritm poate fi îmbunătățit prin introducerea unui pivot u din mulțimea P. Orice clică maximală trebuie să includă fie pe u, fie vecinii săi. Așadar, pentru un nod v care este adăugat la mulțimea v este necesar să testăm doar nodul u și nodurile care nu sunt vecine cu u. Pentru a putea adapta algoritmul Bron-Kerbosch la grafurile contextuale trebuie să treacă printr-o serie de pași.

Pas 1:

Aplicarea unui algoritm care transformă graful contextual și graful șablon într-un graf neorientat astfel încât, orice muchie unidirecțională, va deveni o muchie neorientată.

Pas 2:

Are loc expandarea nodurilor necunoscute în G_s^P, la fel cum am procedat la algoritmul McGregor. Această etapă se poate face ori pe graful orientat, ori pe cel neorientat obținut ulterior. Este preferabil ca acest pas să se întâmple înainte de pasul 1 deoarece există probabilitatea de a elimina de la început muchiile care au aceeași etichetă dar au sensul opus. Astfel se va reduce mai mult complexitatea problemei.

Pas 3:

Se efectuează produsul modular între graful contextual și graful șablon. Îl vom numi graful asocierilor. (en: associations graph) [5].

Pas 4:

Se aplică algoritmul Bron-Kerbosch pe graful asocierilor și se identifică toate clicile maximale. Această soluție de căutare exhaustivă este necesară atunci când vrem să determinăm cea mai bună soluție a problemei inițiale, adică cel mai mare subgraf comun. Avem nevoie de toate clicile maximale deoarece prin transformarea soluției găsite, reprezentată de un graf neorientat, într-un graf orientat, se pierd noduri și muchii.

Teoremă: Având o mulțime cu toate clicile maximale din graful asocierilor, atunci cel puțin o clică din mulțime poate fi asociată cu o soluție maximală a problemei potrivirii grafurilor contextuale.

Demonstrație:

Cunoscând lema: Orice clică maximală în graful asocierilor corespunde celui mai mare subgraf comun al celor 2 grafuri asupra cărora se efectuează produsul modular. (1)

Avem din ipoteză că deținem toate clicile maximale din graful asocierilor (2)

Din (1) și (2) rezultă că putem obține pe baza clicilor identificate, toate subgrafurile comune de dimensiune maximală, bineînțeles neorientate.

Dacă există o clică maximală în graful orientat, atunci aceasta se va regăsi și în graful neorientat obținut în urma pasului 1. (3)

Această clică din graful neorientat va fi inclusă într-o clică maximală.(4)

Argumentare: Presupunem că poate să existe o clică de dimensiune non-maximală, dar care să fie soluție a problemei. Atunci această clică se poate îmbogăți cu noduri suplimentare, astfel încât să se formeze o clică maximală.

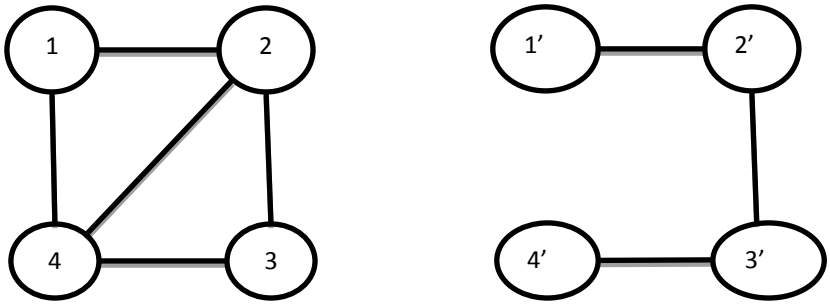
Din (3) și (4) rezultă că orice soluție a problemei potrivirii grafurilor contextuale se va regăsi în mulțimea de clici maximale ale grafului asocierilor.

Complexitatea în cazul cel mai defavorabil pentru algoritmul prezentat este $O(3^{n/3})$ [5].

4.2.1 Condiții pentru corespondența sugraf comun – clică maximală

În foarte multe lucrări care analizează potrivirea de grafuri, conceptele de cel mai mare subgraf comun și clică maximală sunt folosite interschimbabil. Rezultatele algoritmilor pe anumite seturi de date sunt bune și sunt prezentate comparativ cu alți algoritmi. În foarte puține locuri, însă, se precizează că această echivalență nu este validă în orice condiții și necesită un tip special de graf pentru ca transformarea să fie corectă.

O să arăt un exemplu contradictoriu pentru care problema echivalența între a calcula cel mai mare subgraf comun și clica maximală din graful asocierilor, obținut după efectuarea produsului modular, nu funcționează.



a) b)

Fig . 5 – a) Graf contextual, b) Graf șablon

Presupunem că muchiile prezente în graful șablon au aceeași etichetă ca cele din graful contextual. În urma efectuării produsului modular vom obține următorul graf:

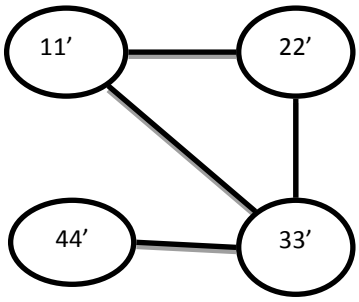


Fig 6 - Graful asocierilor

Dacă aplicăm procedura de determinare a clicii maximale, în graful asocierilor, vom găsi mulțimea de noduri: { 11’, 22’, 33’} corespunzătoare unui subgraf comun format din nodurile { 1, 2 , 3} respectiv {1’, 2’, 3’} în graful șablon. Acest rezultat este eronat deoarece subgraful comun maximal cuprinde și nodul ‘4’.

O lucrare în care s-a cercetat și analizat această situație este “A Necessary and Sufficient Condition for Graph Matching to be equivalent to Clique Search” [19] de Brijnesh Jain și Klaus Obermayer, foarte recentă, din 2009. Aceleași probleme de echivalență a celor două concepte se regăsesc la următorii autori:

- Pelillo, în lucrarea “Feasible and Infeasible Maxima in a Quadratic Program for Maximal Cliques” [20]
- Bunke, în numeroasele lucrări despre corecția erorilor în determinarea potrivirii grafurilor. [15]

Deși problema potrivirii grafurilor, rezolvată prin intermediul grafului asocierilor este foarte interesantă și optimă, profitând de toate studiile și optimizările aduse de algoritmii de clică, nu este universal valabilă. S-a ajuns la concluzia că este o metodă validă doar pentru anumite clase de grafuri. În studiul lui Jain și Obermayer [19], folosirea grafului asocierilor este considerată o tehnică foarte eficientă deoarece:

- Problema clicii maximale este o problemă foarte bine fondată matematic.
- Oferă o plajă întreagă de algoritmi care să rezolve problema inițială.
- Abstractizează particularitățile grafurilor pentru care se încearcă potrivirea.

Într-adevăr, la grafurile contextuale, algoritmul de identificare a clicii maximale nu mai ține cont de etichetările nodurilor necunoscute și nici de orientarea lor, deoarece au fost transformate în muchii neorientate. Aceste constrângeri apar la un nivel de abstractizare mai înalt și vor conta abia la final, în momentul validării soluțiilor găsite.

- Abstractizează constrângerile legate de fezabilitatea potrivirilor.

Din nou, aceeași etichetare a muchiilor pentru grafurile contextuale rămâne să se efectueze într-o etapă ulterioară, tot în momentul validării soluțiilor, deoarece în algoritmul de clică nu contează etichetarea muchiilor ci doar prezența conexiunilor între diverse noduri.

Trebuie ținut cont că atât găsirea celui mai mare subgraf comun cât și a clicii maximale, reprezintă două probleme combinatorice. Tehnicile pentru acest tip de probleme pot fi clasificate în două grupuri:

- Metode discrete - care generează o secvență de probleme parțiale ale problemei inițiale. Acest procedeu se face cu tehnici de căutare locală, plus procedee de a ieși din situația de maxime locale triviale, eliminând efectul problemei cunoscute ca “Hill Climbing”.
- Metode continue - care încapsulează spațiul discret, într-unul mai mare, continuu. Bazându-se pe proprietățile topologice și structurale ale spațiului continuu, algoritmul creează o serie de puncte intermediare care converg către soluția problemei originale.

Abordările discrete au marele dezavantaj că prezintă un număr foarte mare de maxime locale. Folosind graful asocierilor, avem acces la soluții continue eficiente, care scalează foarte bine pentru probleme de origine combinatorică. [19] Această abordare cu folosirea grafului asocierilor are dezavantajele ei.

În lucrările prezentate de Schädler & Wysotzki (citate în [19]), Chen & Yu [22], Pelillo [20], Bunke [15] se prezintă într-un mod trivial transformarea problemelor de subgraf comun în probleme de identificare a clicii, fără a prezenta însă, motivele și ipotezele pentru care această abordare are sens.

În mod particular, pentru grafurile contextuale, ne interesează cu precădere să găsim o condiție necesară și suficientă (C), pentru care transformarea prezentată mai sus este validă.

Fie X și Y două grafuri și M_{XY} un set al tuturor morfismelor parțiale de la X la Y. Notăm cu Z graful asocierilor. Z = X * Y, unde “*” este operator de produs modular. Z va satisface următoarele proprietăți:

1) Z clasifică unic fiecare morfism parțial Ø din spațiul de căutare M, ca fiind o clică $C_{\emptyset}$ în Z, astfel încât f(Ø, X, Y) = w(c), unde w – reprezintă dimensiunea clicii C.
2) Setul $C_Z(M) = \{ C_{\emptyset} \in C_Z : \emptyset \in M\}$ al tuturor clicilor care clasifică morfisme parțiale din M este egal cu C_Z – al tuturor clicilor din Z

Am păstrat aceleași simboluri pentru notații ca cele prezentate în [19].

Așa cum am arătat în exemplul din figura de mai sus, proprietatea 2 nu este general valabilă. Scopul este acela de a găsi o condiție (C) astfel încât, CZ = CZ(M), iar această egalitate să depindă doar de o proprietate a modului cum este construit spațiul de căutare M [19].

Jain și Obermayer au definit această proprietate în modul următor.

Definiție: Dându-se o problemă de potrivire a grafurilor, scopul unei proprietăți este acela de a descrie caracteristicile lui M astfel încât putem deriva relația de echivalență dorită. C exemplu, fie M mulțimea tuturor morfismelor parțiale din M_{XY} care satisfac o proprietate P, de a fi injectivă. Deși proprietatea P descrie complet caracteristicile lui M, acțiunea de a verifica dacă uniunea morfismelor care satisfac proprietatea P, este foarte complex [19].

Definiție: O proprietate P pe o mulțime S este o funcție binară P : S -> {0, 1}. Un element oarecare x ∈ S satisface proprietatea P dacă P(x) = 1 pentru fiecare x ∈ S' ⊂ S [19]. Putem extinde noțiunea de proprietate la submulțimile lui S considerând comportamentul lor local, astfel:

O submulțime S' ⊂ S satisface proprietatea P dacă P(x) = 1 pentru fiecare element x ∈ S', mulțimea S' satisface local pe P.

Considerăm graful asocierilor Z_{XY}, definit pe baza mulțimii nodurilor sale:

$V_{XY} = V(X) * V(Y)$.

Fie P o proprietate a mulțimii formată din anumite elemente, astfel încât nodurile și muchiile lui Z sunt elemente care satisfac P. Notăm această mulțime cu $I_{XY} = I(Z_{XY})$. Submulțimile mulțimii I_{XY}, care satisfac proprietatea P sunt chiar subgrafurile formate din clicile grafului Z.

Putem formula mulțimea tuturor clicilor din Z ca fiind $C_Z = \{ U \subseteq I_{XY} \mid U$ satisface proprietatea P$\}$. Iar cum clicile din $C_Z(M)$ clasifică morfismele posibile din M, puteam să asociem această condiție pe tot spațiul M [19].

Astfel, având aceste concepte în lucrarea lui Jain și Obermayer, putem enunța condiția (C), despre care am stabilit că trebuie să fie necesară și suficientă pentru a putea rezolva problema potrivirii de grafuri prin graful asocierilor.

(C) – Există o proprietate P astfel încât $M = \{ \emptyset \in M_{XY} \mid \emptyset$ satisface pe P$)$

Pentru a ajunge să descoperim forma lui (C), o să ne folosim de două noțiuni și anume închiderea unei mulțimi în raport cu proprietatea P descrisă mai sus și noțiunea de P-morfism.

Definim un P-morfism Ø ca fiind o submulțime a mulțimii $\{ (i, r) \in IXY \mid i \sim_P r \}$

Spunem că două elemente i și r sunt P-similare dacă (i, r) satisface pe P și notăm cu $i \sim_P r$

În termini mai puțin formali, spunem că un morfism parțial Ø satisface pe P, dacă satisface local pe P. [8]

Fie $M_{XY}^P \subseteq M_{XY}$, mulțimea tuturor P-morfismelor. Spunem că o submulțime M a lui M_{XY} este închisă în raport cu proprietatea P dacă $M = M_{XY}^P$.

Lemă: Următoarele submulțimi ale lui M_{XY} sunt închise în raport cu P:

a) M = mulțimea tuturor morfismelor parțiale.
b) M = mulțimea tuturor monomorfismelor parțiale
c) M = mulțimea tuturor omomorfismelor parțiale
d) M = mulțimea tuturor izomorfismelor parțiale
e) M = mulțimea tuturor morfismelor parțiale de subgrafuri.

Pentru a urmări demonstrația acestei proprietăți se poate analiza lucrarea [19].

Teoremă: Fie X și Y două grafuri. Atunci problema de potrivire a celor două grafuri este echivalentă cu cea mai mare clică în graful asocierilor Z = X * Y, dacă și numai dacă există o proprietate P peste mulțimea I_{XY} astfel încât spațiul de căutare M este o mulțime a tuturor morfismelor din mulțimea M_{XY}.

Demonstrația se găsește în [19].

Așadar, pentru orice problemă de potrivire a grafurilor, rămâne să determinăm că spațiul M al problemei este o mulțime închisă în raport cu P, pentru o proprietate P.

O problemă exactă de potrivire a grafurilor are următoarele valori de compatibilitate:

$$K_{ij} = \begin{cases} av \mid xi = yj, i \in V(x), j \in V(y), \\ ae \mid xi = yj, i \in E(X), j \in E(y), \\ \exists ae \mid i \in \exists E(X), j \in \exists E(Y), \\ 0, altfel \end{cases}$$

Este necesar ca av, ae și lae să fie non-negative și $av + ae + lae > 0$. Astfel, dacă vrem să maximizăm numărul de noduri care sunt potrivite, vom seta $av = 1$, iar restul coeficienților de compatibilitate pe 0.

Dacă dorim să maximizăm potrivirea de muchii, vom seta pe $ae = 1$, iar restul pe 0.

În problema grafurilor contextuale avem o situație mai specială. Și nu trebuie să ne rezumăm la grafurilor contextuale, în particular, ci mai degrabă la grafurile care au degenerat în urma pasului de transformare din graful orientat în graf neorientat. Pentru a evita să decidem dacă există o închidere a mulțimii morfismelor în raport cu o proprietate P, putem elimina trivial, la început, toate muchiile din graful contextual care nu se regăsesc cu aceeași etichetă în graful șablon și invers. Această abordare este posibilă și avem garanția că rezultatele sunt corecte conform următoarei teoreme, pe care o să o enunțăm:

Teoremă: Fie X și Y două grafuri neorientate, cu noduri cu etichete **unice** și toate muchiile au aceeași caracteristică, pentru care se poate ignora ae, din valorile de compatibilitate ale potrivirii. Atunci problema identificării celui mai mare subgraf comun este echivalentă cu identificarea de clică maximală.

Demonstrație: Fie mulțimea $M = \{V1, V2, V3, \dots Vn\}$, reprezentând nodurile unui subgraf maximal. În cele două grafuri pot exista următoarele situații:

1) În CG există muchie de la V_i la V_j. De asemenea, în G_s^P există muchie de la V_i' la V_j'.
2) În CG nu există muchie de la V_i la V_j. Nici în G_s^P nu există muchie de la V_i' la V_j'.
3) Într-unul din grafuri există muchie între cele 2 noduri. În celălalt nu există.

Situația 3) nu poate apărea, deoarece cele 2 grafuri au fost trecute printr-o procedură de eliminare a muchiilor și nodurilor care nu mai există. Așadar, putem avea doar cazurile 1) și 2). În cazul 1), pentru orice pereche de noduri (V_1, V_1') și (V_2, V_2'), V1 ∈ grafului contextual și V2 ∈ grafului șablon, între care există o muchie, vom avea o muchie și în graful asocierilor între nodul V_1V_2 și V_2V_2'.

În cazul 2), dacă nu există nicio muchie care să conecteze 2 noduri cu aceeași etichetă, nici în graful șablon, nici în graful contextual, atunci va exista o muchie care să le conecteze în graful asocierilor, între perechile de noduri corespunzătoare.

Pe baza celor două situații de mai sus, deducem că nu există o pereche de noduri (V_1, V_2) din mulțimea M, între care să nu existe muchie. Prin urmare, dacă toate nodurile din mulțime sunt conectate în graful asocierilor, acestea formează o clică. Clica este de asemenea maximală.

Presupunem prin absurd că există o altă mulțime M2 de noduri care formează o clică maximală. Atunci această clică a fost formată din perechi de noduri care au aceeași etichetă în cele 2 grafuri pe care dorim să le potrivim. Înseamnă că ar exista o mulțime de noduri cu aceeași etichetă ca cele din M2, care ar forma cel mai mare subgraf comun. Dar din ipoteză, am presupus că mulțimea M este cel mai mare subgraf comun, ceea ce duce la contradicție.

Este de notat că la demonstrațiile de mai sus, ne-am referit la graful contextual și graful șablon(CG și G_s^P) prin prisma grafurile neorientate, asociate lor, care au rezultat din transformare.

Observăm că dacă graful contextual deține noduri unice, problema este foarte ușor rezolvabilă printr-un algoritm de identificare a clicii maximale. Noi introducem o etapă suplimentară de transformare a problemei inițiale, prin expandarea nodurilor necunoscute. În urma acestei expandări, vor rezulta, cel mai probabil, noduri care au aceeași etichetă.

Prin introducerea nodurilor multiple cu aceeași etichetă putem ajunge la următoarea situație, în care două noduri etichetate cu "?" sunt expandate la "Student":

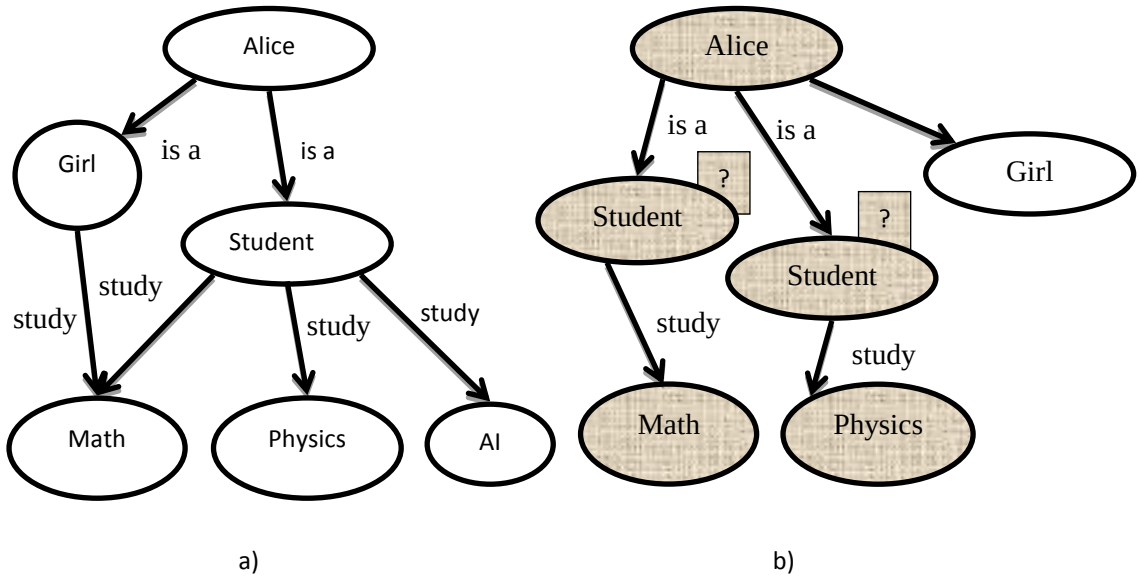


Fig. 7 – a) Graf contextual, b) Graf șablon

Deoarece de la un nod la el însuși se consideră că nu există muchie, în urma construirii grafului asocierilor vom obține o clică maximală corespunzătoare părții reliefate în figura de mai sus, ceea ce nu este corect deoarece există un singur nod cu eticheta "Student" în graful contextual.

Pentru această situație particulară, trebuie o verificare separată, în care nu putem permite în clica parțială, generată prin algoritmul implementat, noduri cu aceeași etichetă. În principiu, am găsit o soluție pentru a rezolva într-un mod corect, indiferent de natura exemplelor, problema potrivirii grafurilor contextuale. Mai rămâne însă, un ultim impediment, care nu a fost tratat în mod corespunzător în literatura de specialitate, trecându-se peste acest aspect fără a fi analizat într-un mod corespunzător. Să considerăm exemplul din figura 8:

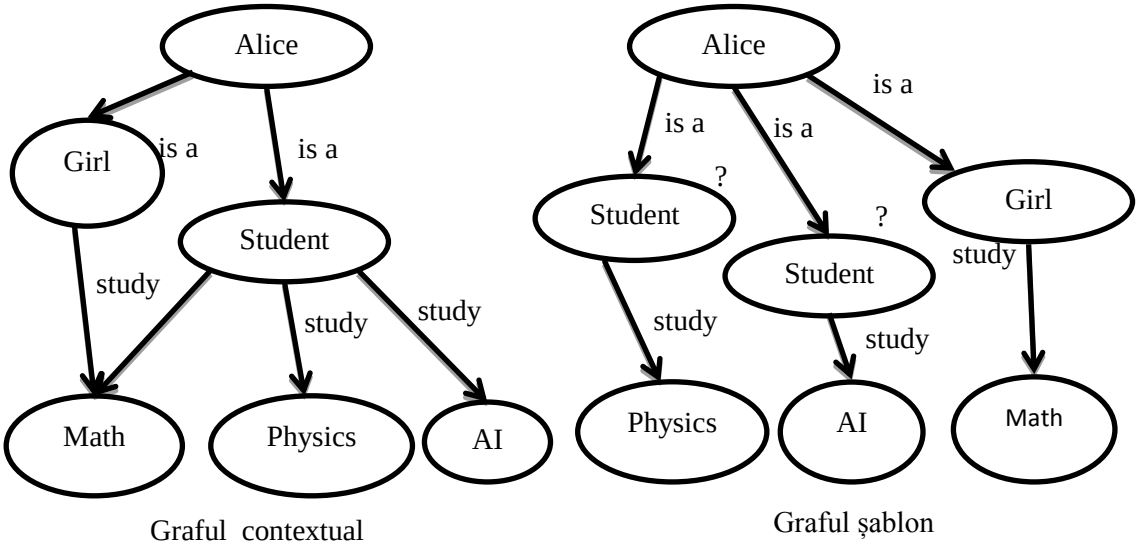


Fig. 8

În urma transformării în grafuri neorientate, graful asocierilor va arăta astfel:

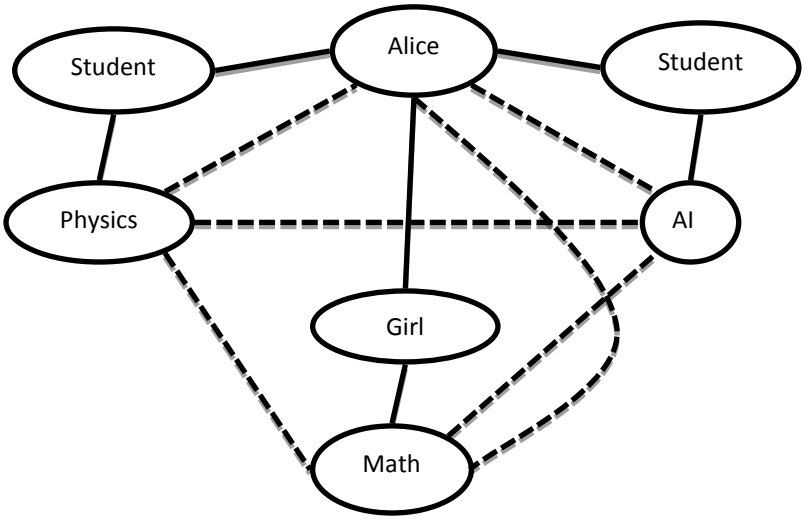


Fig. 9 Exemplu de echivalență invalidă

Observăm că cel mai mare subgraf comun al celor două grafuri are lungimea 3 și poate fi una dintre cele 3 ramuri ale grafului șablon. În schimb, clica maximală în graful asocierilor conține 4 noduri și este evidențiată cu linie punctată în figura 9. Mulțimea {Alice, AI, Physics și Math } nu formează cel mai mare subgraf comun deoarece între aceste noduri nu există nicio muchie. Explicația este următoarea: dacă dorim să găsim o soluție care reprezintă un subgraf conex, teorema de echivalență enunțată mai sus nu mai este valabilă.

Valorile de compatibilitate Kij nu mai sunt suficiente. Trebuie să existe restricția suplimentară asupra clicii găsite, ca nodurile corespunzătoare din graful contextual și graful șablon să formeze un graf conex. Această etapă introduce în plus o complexitate temporală O(n) pentru fiecare clică găsită. (În cazul cel mai defavorabil se vor evalua k*n noduri, unde k este un coeficient de conexiune a nodurilor, care este mai mare atunci când graful conține mai puține muchii).

Concluzia care se poate desprinde în urma acestor considerente teoretice este una avantajoasă, deoarece nu avem nevoie de o condiție foarte strictă pentru a demonstra morfismul spațiului de căutare în raport cu o proprietate consistentă. Cele două probleme sunt mereu echivalente, doar efectuând transformări și considerând doar soluțiile valide, pentru care graful obținut este conex.

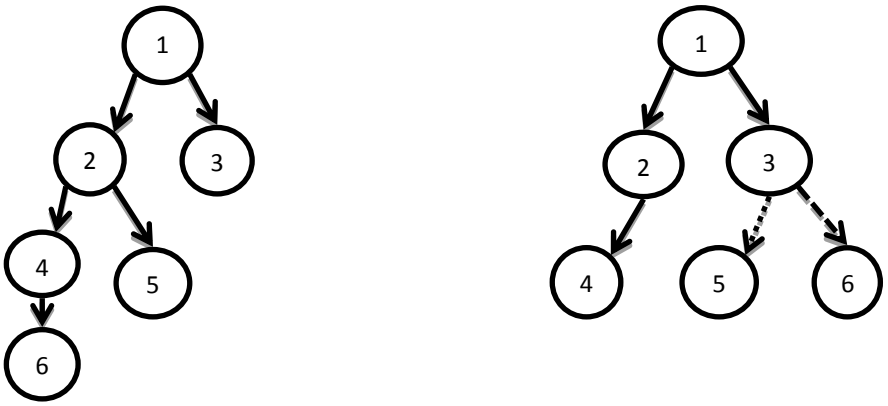
4.3 Algoritm Akkoyunlu

Deși acest algoritm este prezentat diferit, el constă din aceleași tehnici de optimizare ca și Bron Kerbosch și generează același arbore în spațiul de căutare. Prin urmare poate fi considerat același algoritm însă studiile și argumentările sunt mult mai recente în lucrarea lui Akkoyunlu [8]. Optimizările care se pot efectua asupra algoritmului Bron-Kerbosch pe care le-am adăugat si le-am testat in vederea optimizării complexității sunt următoarele:

La fiecare pas, nodurile sunt ordonate după gradul de ramificație. Această operație poate fi făcută în O(N) dacă nodurile sunt ținute într-un vector. Complexitatea este log(N), dacă nodurile sunt ținute într-o structură heap, dar cu dezavantajul că heap-ul trebuie modificat și reconstruit. În platforma care însoțește această lucrare, am ales să stocăm nodurile într-o lista înlănțuită, cu complexitate O(N). Această ordonare după gradul de ramificație are următoarele fundamente teoretice:

Definim degenerarea unui graf G [8], ca fiind cel mai mic număr d (degree), astfel încât fiecare subgraf al lui G are cel puțin un nod cu gradul d sau mai mic. Se definește ordonarea degenerativă, ca fiind o sortare a nodurilor astfel încât fiecare nod are cel mult d vecini, care urmează mai târziu în ordonare. Exemplul din figura de mai jos ilustrează ce înseamnă ordonare degenerativă cu gradul 2.

Fie secvența ordonată: 1, 2, 4, 6, 5, 3.



a) b)

Fig 10. a) Graf care respectă ordonarea degenerativă
b) Graf care nu respectă ordonarea

Deoarece graful neorientat derivat din graful contextual sau din graful șablon, poate conține cicluri, sau chiar clici, pentru a optimiza complexitatea problemei este necesară o alegere a nodurilor cu o degenerare cât mai mica. În algoritmul de căutare care folosește parcurgeri arborescente, în urma traversării grafului, va rezulta un arbore, ce corespunde nodurilor, în ordinea în care au fost vizitate. O degenerare mica a ordonării nodurilor din graf poate fi foarte utilă, deoarece ne interesează să reducem gradul de ramificație al nodurilor apropiate de rădăcină. Dacă majoritatea nodurilor sunt frunze, este foarte ușor dacă ele fac sau nu parte din clica deja descoperită parțial. În schimb, dacă ramificația este foarte mare la rădăcină, se va efectua un efort considerabil pentru a străbate diferite căi pentru a valida abia la sfârșit că toată clica parțial generată nu va fi maximală.

```
procedură Akkoyunlu(R,P,X)
      dacă P și X sunt vide:
            R este o clică maximală
      sfârșit
      alege un nod pivot în P ∪ X
      pentru fiecare nod v în P \ N(u) într-o ordine
      degenerativă a lui G:
            Akkoyunlu(R ∪ {v}, P ∩ N(v), X ∩ N(v))
            P = P \ {v}
            X = X ∪ {v}
      sfârșit
sfârșit procedură [8]
```

Pentru a obține o problemă rezolvabilă în această situație, la fel ca la Bron-Kerbosch, trebuie urmați aceeași pași de transformare a problemei inițiale.

4.4 Algoritmul Durand-Pasari

Algoritmul se bazează tot pe echivalența subgraf maximal – clică maximală în graful asocierilor [5]. Prin urmare, transformările care trebuie făcute sunt aceleași care au fost analizate la algoritmii de mai sus.

Acest algoritm are ușor altă abordare față de tehnica propusă în lucrarea [3], respectiv algoritmul Bron-Kerbosch. Se generează o listă de noduri care reprezintă o clică în graful asocierilor folosind o strategie arborescentă de căutare în adâncime. Se va selecta sistematic câte un nod, de pe nivele succesive, până când nu mai este posibil să se adauge un alt nod în listă. Când un nod este considerat, se verifică dacă acest nod este fezabil pentru clica în construcție, adică este conectat cu restul nodurilor plus verificările de fezabilitate introduse de acest tip particular de graf, care au fost analizate în capitolul 4.2.

La fiecare nivel k, trebuie să ne asigurăm că spațiul de căutare este un arbore și că odată ce a fost vizitat, nu va mai fi parcurs a doua oară. După ce sunt considerate toate nodurile posibile la nivelul k, se va adăuga în listă un nod special, numit „**null_node**” [5]. Acest nod este întotdeauna valid și poate fi adăugat mai mult de o singură dată în listă. Nodul null e folosit pentru a deține informația că nici un izomorfism nu este asociat unui nod particular din graful contextual. Când toate nodurile posibile (incluzând nodurile nule) au fost considerate, algoritmul va continua cu procesul de backtracking și va explora o ramură diferită în spațiul de căutare. Cea mai lungă listă de noduri, după ce a fost verificată fezabilitatea ultimului nod adăugat, va fi stocată.

Algoritmul în pseudocod arată astfel [5]:

```
procedura DurandPasari(S)
  cât timp există un nou nod n care poate fi atins din starea S
      dacă este_fezabil(S, n)
          dacă nu exista o condiție de eliminare a ramurii
          alese din considerente de performanță atunci
              S' = adaugă_nod(S, n)
              dacă card(S') > MaxClică atunci
                  Salvează(S')
                  MaxClică = card(S')
              dacă nu s-a ajuns la o frunză în arborele de
              căutare atunci
                      DurandPasari(S')
              Backtracking(s')
```

Dacă N_1 și N_2 sunt numărul de noduri ale celor două grafuri, cu $N_1 < N_2$, ordine aleasă arbitrar, poate fi demonstrate că înălțimea maxima a arborelui de căutare este maxim N_1. Deaoarece soluția se poate reține într-o mulțime cu comportament de stivă (se adaugă și se scot noduri doar dintr-un capăt), aceasta poate fi partajată între stări diferite, corespunzând unor nivele diferite în apelurile

recursive. Prin urmare, complexitatea spațială a algoritmului este $O(N_1)$ [10]. La această complexitate poate fi adăugat și spațial necesar pentru stocarea grafului asocierilor.

Acesta poate avea maxim $N_1 * N_2$ noduri, în cazul cel mai defavorabil și va rămâne constant pe toată durata execuției algoritmului.

4.5 Algoritmul Koch

Ina Koch în articolul "Enumerating all connected maximal common subgraphs in two graphs" [11], prezintă problema celui mai mare subgraf comun prin echivalență cu clica maximală, introducând o nouă abordare. Prin metoda expusă mai jos, se vor evita oarecum problemele care au apărut la algoritmul Bron-Kerbosch, la adaptarea la grafurile contextuale.

Când am analizat algoritmul Bron-Kerbosch, atenția s-a concentrat pe graful asocierilor, făcând produsul modular al nodurilor celor două grafuri. Poate fi creat, însă, un graf al asocierilor, efectuând produsul muchiilor. Îl vom numi "graful produsului muchiilor" sau H_e [11].

$He = G_1 x_e G_2$. He este format din mulțimea de noduri $V_H = E_1 x E_2$, în care perechea de muchii (ei, ej) cu ($0 < n < n + 1$ și $0 < j < m + 1$) trebuie să aibă aceeași etichetă și în plus să coincidă etichetele nodurilor care formează cele două muchii. În graful He, există o muchie între două noduri e_H și $f_H \in V_H$ cu $e_H = (e_1, e_2)$ și $f_H = (f_1, f_2)$, doar dacă:

- $e_1 \neq f_1$
- $e_2 \neq f_2$
- e_1 și f_1 sunt conectate prin intermediul unui nod, în graful G_1, cu aceeași etichetă a nodului prin care sunt conectate e_2 și f_2 în graful G_2. **Sau** e_1, f_1 și e_2, f_2 nu sunt adiacente nici în G_1 și nici în G_2. [12]

Definiție: Subgraful indus Gi al unui graf G este o submulțime a nodurilor lui G, în care $\forall (v_1, v_2) \in E_{Gi}$ dacă și numai dacă (v_1, v_2) [13]. Mai precis, toate muchiile din subgraful indus se găsesc în graful inițial.

Am evitat introducerea conceptului de subgraf indus la algoritmul Bron-Kerbosch pentru a înțelege mai bine etapele necesare transformării acelui algoritm. Trebuie remarcat că prin identificarea unei clici în graful asocierilor, se obține cel mai mare subgraf comun indus. Potrivirea grafurilor contextuale nu necesită un izomorfism în care să fie incluse toate muchiile ci ne interesează doar cel mai mare izomorfism.

Pentru a obține cel mai mare subgraf comun și nu cel mai mare subgraf indus comun, muchiile care nu au corespondență în potrivire, sunt eliminate la sfârșit, înainte să se afișeze soluția.

Graful He, în schimb, nu mai suferă de ambiguitatea gestionării tuturor muchiilor între 2 noduri și găsirea subgrafului comun este mai facilă. În figura 11 (adaptată după scenariul din [11]), se poate observa o comparație între graful asocierilor, graful produsului muchiilor, subgraful comun și subgraful indus comun.

Definiție: Fie (e_1, e_2) și (f_1, f_2) două perechi de muchii ale celor două grafuri G_1 și G_2. O muchie în graful He între nodurile (e_1, e_2) și (f_1, f_2) se numește c-muchie dacă muchiile e_1, f_1 din G_1 sau e_2, f_2 din G_2 sunt conectate printr-un nod cu aceeași etichetă. Altfel, dacă cele două muchii nu partajează un nod comun și nu sunt adiacente se numesc d-muchii.

Deoarece pot exista foarte multe d-muchii în H_e, vom încerca să identificăm doar așa numitele c-clici.

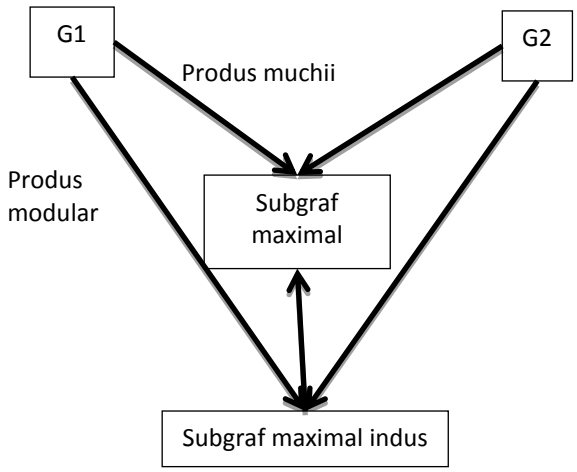


Fig. 11 Comparație concepte subgrafuri

Se observă că prin efectuarea produsului muchiilor se obține direct subgraful maximal.

Definiție: O clică într-un graf G care constă din c-muchii și d-muchii se numește c-clică dacă este formată din c-muchii astfel încât este conexă și aciclică.

Definiție: O c-clică în graful produsului muchiilor H_e este o clică care reprezintă cel mai mare subgraf comun în grafurile G_1 și G_2 din care a rezultat He în urma produsului muchiilor

Așa cum am observat la algoritmul Bron-Kerbosch, nu este suficient să găsim cea mai mare clică, ci trebuie să enumerăm toate soluțiile maximale. Spre deosebire de problema determinării celei mai mari clici care este NP-completă, se știe că enumerarea tuturor clicilor este o problemă NP-dură. Numărul total de clici poate crește exponențial, odată cu creșterea numărului de noduri.

Algoritm de detecție a tuturor c-clicilor:

Enumerare_c-clici (C, P, D, S)

Notații:

C: mulțimea nodurilor din clica curentă

P: mulțimea nodurilor care pot fi adăugate unei clici C, deoarece ele sunt vecini ai nodului u doar prin intermediul c-muchiilor.

D: mulțimea nodurilor care nu pot fi adăugate direct la C, deoarece ele sunt vecini ai lui u prin intermediul d-muchiilor.

S: mulțimea nodurilor care nu sunt permise în C.

N – mulțimea vecinilor

Fie P mulțimea $\{u_1, \ldots, u_k\}$;

```
Dacă P = Ø și S = Ø
    s-a găsit clică
altfel
       pentru i ← 1 … k
           P ← P \ {ui};
              P' ← P;
              D' ← D;
              S' ← S;
             N ←{v ∈ V | {ui, v} ∈ E };
             Pentru toate v ∈ D'
            Dacă v și ui sunt adiacente printr-o c-muchie
                 P' ← P' ∪ {v};
                 D' ← D' | {v};
             Enumerare_c-clici(C ∪ {ui},P' ∩ N, D' ∩ N, S' ∩ N)
             S ← S ∪ {ui};
```

Lemă: O c-clică poate fi raportată dacă mulțimea D nu este vidă. (1)

Teoremă: Algoritmul de mai sus dispune de următoarele proprietăți: (2)

1) Toate perechile de noduri din C sunt adiacente și C este generată de c-muchii. Fiecare nod al lui G care este adiacent cu toate nodurile din C este P, D sau S.
2) Fiecare nod u ∈ P este adiacent cu toate nodurile din C și cu cel puțin un nod este adiacent printr-o c-muchie. Nodurile din P sunt utilizate pentru extinderea mulțimii C. Dacă un nod din P a fost folosit vreodată pentru a extinde pe C sau pentru a inițializa nodurile funcției enumerare_c-clici(), atunci va fi mutat în S.
3) Fiecare nod u ∈ D este adiacent prin c-muchii și d-muchii tuturor nodurilor din C. Mulțimile de noduri ale tuturor clicilor care conțin mulțimea C ∪ {u} au fost deja enumerate o dată.
4) Fiecare nod u ∈ D este adiacent tuturor nodurilor din C prin d-muchii. Un nod din mulțimea D poate fi folosit pentru a extinde pe P, doar dacă un nod din P a fost mutat în C și care e adiacent cu acel nod în D printr-o c-muchie.
5) Mulțimea T conține doar nodurile care deja au fost folosite pentru inițializarea funcției enumerare_c-clici.
6) Dacă un apel al funcției enumerare_c-clici s-a terminat, mulțimea de noduri a c-clicilor este enumerată exact o singură dată.

Pentru demonstrația acestei lemei de mai sus și a teoremei (2) se poate consulta [11]

Folosind (1) și (2), un algoritm optimizat propus de Ina Kock în [11] arată în felul următor:

```
Enumerare_c-clici (C, P, D, S)
Fie P mulțimea {u1, …, uk};
   dacă P = ∅
       dacă S = ∅
           s-a găsit clică
    altfel fie ut un nod al mulțimii P
    pentru i ← 1 … k
       dacă ui nu este adiacent cu ut sau ut nu este conectat
       printr-o c-muchie cu un nod din D care nu e adiacent cu
       ut atunci:
            P ← P \ {ui}; P' ← P;  D' ← D;  S' ← S;
            N ←{v ∈ V | {ui, v} ∈ E };
            Pentru toate v ∈ D'
                 Dacă v ∈ P atunci
                      P' = P' ∪ {v}
                 Altfel dacă v ∈ D \\ poate fi adaugat v la P?
                     Dacă v și ui sunt adiacente printr-o
                     c-muchie
                         Dacă v ∈ T
                             S' = S' ∪ {v};
                         altfel
                             P' ← P' ∪ {v};
                             D' ← D' \ {v};
                 altfel dacă v ∈ S atunci
                       S' = S' ∪ {v}
            sfârșit pentru
            Enumerare_c-clici(C ∪ {ui},P'∩ N,D'∩ N,S' ∩ N)
            S ← S ∪ {ui};
```

Această metodă a fost folosită de Koch pentru analiza structurală a proteinelor [3,11]. Acest algoritm poate fi folosit cu rezultate mulțumitoare și la grafurile contextuale. Faptul că permitem generarea de clică doar pentru subgrafuri conexe reduce foarte mult complexitatea. Problema rămâne însă una NP – completă și nu poate găsi o soluție pentru anumite tipuri de grafuri foarte mari, cu dimensiuni și cu proprietăți arbitrare.

4.6 Algoritmul Balas-Yu

În esență, algoritmul dezvoltat de E. Balas și C. S. Yu în aricolul "Finding a Maximum Clique in an arbitrary graph", se axează tot pe echivalența subgraf comun – clică în graful asocierilor. [21]

Spre deosebire de ceilalți algoritmi prezentați mai sus, acesta folosește o euristică mult mai sofisticată pentru a tăia ramuri din arborele de căutare. Această euristică este fondată pe două

concepte și anume colorarea unui graf și triangularea acestuia. Unele definiții de bază sunt necesare pentru a descrie cum funcționează acest algoritm.

Definiție: Fie G un graf neorientat și fie V, respectiv e numărul de noduri și de muchii ale lui G. Fie w(G) dimensiunea clicii maximale. O colorare a lui G atribuie culori nodurilor lui G astfel încât nu vor exista două noduri adiacente cu aceeași culoare. Numărul minimul de culori necesare pentru a colora nodurile lui G în acest fel este χ(G). S-a demonstrat că χ(G) este o limită superioară pentru w(G). Problema k-colorării, așa cum este cunoscută în literatura de specialitate are complexitatea O(V + E).

Definiție: Un graf G_T se numește cordal sau triangulat dacă fiecare ciclu al lui GT, a cărui lungime este cel puțin 4, are o coardă [5, 11].

În [14] se definește LTS(GT) ca fiind cel mai mare graf triangulat al lui G. S-a demonstrat că identificarea lui LTS(GT) are complexitate O(V + E). O clică maximală a lui LTS(GT), ϕ(G) poate fi găsită ca fiind un rezultat adițional obținut în timpul parcurgerii lui LTS(GT) și este o limită inferioară pentru w(G).

Avem relația: $|\varphi(G)| \leq |w(G)| \leq |\chi(G)|$

```
procedură BalasYu(s)
      cât timp se poate selecta o subproblemă (s, n)
            Rezolvă problema (s, n);
            dacă card(subproblemă selectată(s))> MaxClică atunci
                  Salvează clica maximală curentă(s)
                  MaxClică = card(subproblemă selectată (s))
            cât timp există noi subprobleme (s)
                  s' = update(s);
                  BalasYu(s')
            Backtracking(s')
```

Algoritmul Balas-Yu poate fi descris printr-o reprezentare în spațiul stărilor. Fiecare stare s este asociată cu subproblema găsirii unei clici maximale în graful asocierilor sau graful produsului muchiilor [5].

Fiecare problemă de dimensiune k poate fi considerată partiționând graful inițial în 3 submulțimi [5, 21]:

1) I – conține nodurile care sunt forțat incluse în subproblemă
2) E_x – nodurile care sunt excluse forțat din subproblemă
3) S – noduri neclasificate, adică restul

Plecând dintr-un nod n ales din mulțimea S, soluția subproblemei constă în a identifica câte noduri pot fi colorate în graful G', câte noduri sunt noduri din S, exceptând pe n. De asemenea trebuie determinat care muchii aparțin grafului inițial care conectează nodurile din S. Acest procedeu se aplică fixând dinainte numărul c = |ϕ(G)| culori. Nodurile necolorate sunt stocate într-o mulțime W. Un rezultat adițional al acestui proces de colorare este o clică a cărei dimensiune este în cel mai bun caz |I| + c. În cazul în care clica este mai mare decât cea mai bună soluție găsită este stocată ca fiind noua soluție, cea mai bună.

În urma experimentelor efectuate, s-a observat că algoritmul Balas-Yu este ineficient deoarece necesită foarte multă supraîncărcare în a construi și a menține o euristică destul de complicată iar ramurile care sunt eliminate din arborele de căutare nu reduc semnificativ complexitate. Acest algoritm este bun însă pentru grafuri mari și foarte dense. Atunci când există o densitate mare a conexiunilor între noduri, ceilalți algoritmi petrec foarte mult timp generând toate clicile până la determinarea soluției optime și nu încadrează această soluție între niște limite.

O soluție și mai bună care limitează spațiul în care căutăm clicile maximale a fost propusă de Yuji Shinano în lucrarea "Solving the maximum clique problem"[12]. Algorimtul respectiv folosește tehnica de "branch and bound", și introduce conceptual de PUBB (Parallelization Utility for Branch-and-Bound algorithms). Algoritmii paraleli pentru potrivirea de grafuri nu se încadrează în scopul studiului nostru însă optimizările prezentate de Shinano sunt interesant de analizat.

4.7 Algoritmul Larrosa

Înainte de toate, să introducem algoritmul de backtracking cu verificare predictivă, de la care a pornit studiul condus de Javier Larrosa și Gabriel Valiente în lucrarea "Graph pattern matching using Constraint Satisfaction" [16].

procedură rezolvă (X, D)
    dacă X este vidă **atunci**
    atunci s-a găsit o soluție pentru problemă
    altfel
        D' ← propagă constrângerile la vecini (X, D)
        dacă nici un domeniu din D' nu este vid **atunci**
            se selectează o variabilă i din X
            pentru toate v_a **în** D_i **execută**
                D'' ← verifică înainte (i, v_a, D')
                dacă niciun domeniu în D'' nu este vid **atunci**
                    rezolvă (X – {i} , D'')

procedura propagă constrângerile la vecini (X, D)
  schimbat = adevărat
  cât timp schimbat
      schimbat = false
      pentru fiecare i în X **execută**
          pentru fiecare va în D_i **execută**
              fie p, …, q vecinii lui i în X
              dacă (v_{a1}, …, v_{ar}) ∉ $R_{i,p, \ldots q}$ pentru toate v_{a1}, …, v_{ar} ∈ D_p x … x D_q **atunci**
                  D_i ← D_i – {v_{a1}}
                  schimbat ←adevărat
  întoarce D

Pentru a înțelege mai bine notațiile și motivele acestei abordări, vom defini câteva noțiuni:

O problemă de satisfacere a restricțiilor (CSP) este definită de o mulțime ordonată de n variabile X = (1, 2, …, n), o mulțime Di ce reprezintă valorile posibile pentru fiecare variabilă i și o mulțime C de constrângeri între variabile.

O constrângere R_{j1}, …, j_r pe mulțimea ordonată a variabilelor (j_1, …, j_r) este o submulțime a mulțimii D_{j1} * D_{j2} * D_{jr} care conține doar combinațiile permise de valori pentru variabilele j_1, …, j_r.e

O atribuire de valori pentru variabile este completă dacă include fiecare variabilă din X. O atribuire de valori este consistentă dacă satisface orice constrângere. O soluție pentru problema satisfacerii restricțiilor este formată dintr-o atribuire de valori completă și consistentă. [16]

Prin satisfacerea restricțiilor se poate urmări o generare exhaustivă a tuturor soluțiilor problemei, o singură soluție sau găsirea celei mai bune soluții în anumite circumstanțe. La problema grafurilor contextuale ne interesează să găsim o singură soluție iar aceasta poate fi desemnată atunci când tuturor variabilelor, (în cazul nostru sunt nodurile etichetate cu "?" din graful șablon), li s-a atribuit o valoare. Dacă știm ce valori au nodurile din graful șablon, misiunea de a găsi cel mai mare subgraf comun este foarte ușoară deoarece nodurile sunt unice și se poate determina trivial în timp polinomial.

Așadar problema de a găsi un subgraf care să fie un izomorfism al grafului contextual (V_1, E_1) cu graful șablon (V_2, E_2) este echivalent cu a găsi o instanțiere corespunzătoare a nodurilor etichetate cu "?" la etichete concrete care se regăsesc în graful contextual și apar o singură dată în graful șablon. Pentru a atribui valori variabilelor, într-un mod corect trebuie respectată următoarea constrângere structurală:

$R_{i,j} = \{ (v_a, v_b) \in V_2 \times V_2 \mid v_a \neq v_b \wedge \text{muchie}(G_1, i, j) \rightarrow \text{muchie}(G_2, v_a, v_b)\}$, pentru $\forall$ i,j = 1, ..., n, cu $i \neq j$. [16]

Există trei tipuri principale de abordări ale acestei probleme, în satisfacerea restricțiilor, în domeniul izomorfismelor de grafuri:

1) backtracking cu verificare predictivă, cunoscut ca FC (forward checking), se verifică doar constrângerile cu vecinii și se propagă arborescent în graful constrângerilor.
2) backtracking cu predicție totală, denumit și RFLA (really full look-ahead) – se efectuează algoritmul cu verificare predictivă, până când nu se mai pot elimina valori din domeniul variabilelor.
3) backtracking cu salt înapoi (back – jumping) – în caz de insucces se sare înapoi la atribuirea variabilei care a produs un rezultat nesatisfăcător și se încearcă altă cale.

Este de menționat faptul că acești algoritmi sunt concepuți pentru un graf de constrângeri binare. O soluție interesantă a fost propusă de J.-C. Régin în "A filtering algorithm for constraints of difference in constraint satisfaction" [23], unde introduce conceptul de constrângeri de ordin "n" (sau n-are). În algoritmul propus, se dorește ca fiecare variabilă să respecte restricțiile în raport cu toate celelalte variabile. Autorul a demonstrate că acest algoritm este mai optim decât cei cu constrângeri binare. Pentru problema grafurilor contextuale, această soluție impune o supraîncărcare prea mare, deoarece reprezentarea în program a grafului, a nodurilor și muchiilor nu este tocmai favorabilă. De asemenea nu dorim să consumăm timpul de procesor și memoria consumată cu preprocesarea constrângerilor inițiale și menținerea arc-consistenței a relațiilor n-are în timpul rulării algoritmului.

Nu vom experimenta varianta cu salt înapoi la grafurile contextuale. Este cunoscut faptul că algoritmul de verificare predictivă nu va expanda niciodată mai multe noduri în arborele de căutare decât cel cu salt înapoi. Studii în această direcție au fost conduse de G. Condrak și P. V. Beck în lucrarea "A theoretical evaluation of selected backtracking algorithms"[24].

În schimb FC reduce mai puțin complexitatea decât algoritmul RFLA. Ca o formulare intuitivă, algoritmul cu predicție totală va expanda întotdeauna mai puține noduri decât cel cu verificare predictivă deoarece la fiecare iterație de backtracking se elimină cel puțin valorile din domenii, pe care le-ar fi eliminat și algoritmul FC.

Prima formulare a izomorfismului între două grafuri ca o problemă CSP, este atribuită lui McGregor, în lucrarea "Relational consistency algorithms and their application in finding subgraph and graph isomorphisms" [25].

Studii ulterioare în acest domeniu au fost întreprinse de J. Larrosa, care a propus un nou algoritm, expus la începutul acestui subcapitol. Algoritmul constă într-un concept nou și anume constrângerile față de vecini, care arată faptul că un nod $v_i \in V_1$ poate fi atribuit prin izomorfism unui nod $v_j \in V_2$, dacă toți vecinii nodului v_i pot fi atribuiți unor noduri din mulțimea vecinilor lui v_j.

Constrângerile legate de vecini au fost definite de Larrosa în [16] astfel:

$R_{i,\ p,\ \ldots,\ q} = \{ (v_{a1}, \ldots, v_{ar}) \in V_2 \times V_2 \times \ldots \times V_2 \mid$

$v_{ak} \mathrel{!}= v_{al},\ \forall\ k, l = 1, \ldots, r\ (k \neq l) \wedge$

$muchie(G_2, v_{a1}, \ldots v_{ak}), \forall\ k = 2, \ldots, r\}$

{p, …, q} reprezintă mulțimea de vecini a nodului i în graful G1.

Teoremă: Algoritmul lui Larrosa pentru izomorfismul grafurilor nu vizitează niciodată mai multe noduri decât algoritmul cu verificare predictivă sau decât cel cu predicție totală.

Pentru demonstrația acestei teoreme se poate consulta [16].

Folosirea acestui algoritm în problema noastră impune o serie de considerente pentru care abordarea problemei prin acest algoritm necesită unele ajustări.

Modificarea 1:

Principalul neajuns al algoritmului Larrosa la grafurile contextuale este acela că el se poate adapta doar dacă problema constă în găsirea unui izomorfism complet, între mulțimile de noduri din primul graf și cel de-al doilea.

Pentru a transforma izomorfismul de grafuri la un izomorfism parțial, între subgrafuri, se verifică la fiecare adăugare a unui nou nod în soluția generată parțial în procesul de backtracking, dacă soluția are dimensiunea maximă până în momentul respectiv. În caz afirmativ această soluție este reținută.

Modificarea 2:

Un alt impediment este reprezentat de posibilitatea potrivirii grafurilor contextuale, de a avea un subgraf comun care nu atribuie etichete concrete tuturor nodurilor. Mai precis, putem avea noduri etichetate cu “?” în graful șablon, care nu au corespondent în graful contextual. Acest detaliu are un impact foarte mare asupra mecanismului de backtracking deoarece nu mai este validă verificarea vecinătăților și eliminarea atribuirilor care fac ca unele domenii ale variabilelor să devină vide. Prin urmare, toate optimizările de “prunning” pe care arc-consistența le aduce unei probleme NP-complete, nu mai pot fi aplicate, deoarece nu putem defini o soluție validă a problemei, ci doar o căutăm pe cea mai optimă.

Algoritmul Larrosa nu va putea rezolva problema grafurilor contextuale, deoarece nu putem profita de avantajele pe care le introduce. În final, pentru a obține o soluție corectă este necesară expandarea întregului spațiu de căutare prin backtracking. Algoritmul poate fi însă aplicat cu succes în două tipuri de probleme particulare:

- atunci când nodurile etichetate cu “?” fac mereu parte din cel mai mare subgraf comun.
- atunci când pentru agentul care trebuie să identifice o situație din context, este suficientă o soluție ne-optimă dar pe care o poate identifica foarte rapid.

Rezultatele algoritmului pentru acest tip de problemă sunt analizate în capitolul 6.

5. Prezentare platformă

5.1 Utilitatea dezvoltării platformei de testare a algoritmilor

Pentru a avea o metodă de evaluare și comparare a algoritmilor este necesară o platformă de testare generică a performanțelor algoritmilor pentru graful contextual.

Utilizatorul trebuie să aibă acces la o fereastră din care să aleagă tipurile de grafuri, dacă grafurile sunt generate aleator sau sunt extrase din fișiere care conțin exemple particulare sau didactice. De asemenea trebuie să se poată vizualiza graful inițial, graful șablon și graful rezultat în urma potrivirii. Pentru a alege ce fel de algoritm este folosit, există posibilitatea de a alege între cei implementați sau a rula toți algoritmii pe cele 2 grafuri alese și a compara rezultatele obținute, optimalitatea soluției găsite pentru algoritmii inexacți.

Platforma trebuie să ofere posibilitatea de a genera noduri și etichete de muchii dintr-un alfabet din concepte existente în limbajul natural și atribute relaționale care stabilesc legăturile între concepte. Prin intermediul interfeței, putem să alegem ce tip de graf particular aleator vrem să generăm:

1) **Small-world** – un tip particular de graf în care majoritatea nodurilor nu sunt vecine cu celelalte car se poate ajunge din orice nod în oricare din celelalte noduri într-un număr cât mai mic de pași. Are avantajul de a prezenta clici mici și numeroase la periferie, fiind util în testarea algoritmilor cu grafuri mari.

2) **Scale free** – este o rețea în care gradul de distribuție urmează legea puterii. Fie $f(x) = ax^k$. Scalând pe x cu un factor constant c, determină doar o scalare proporțională a funcției însăși.

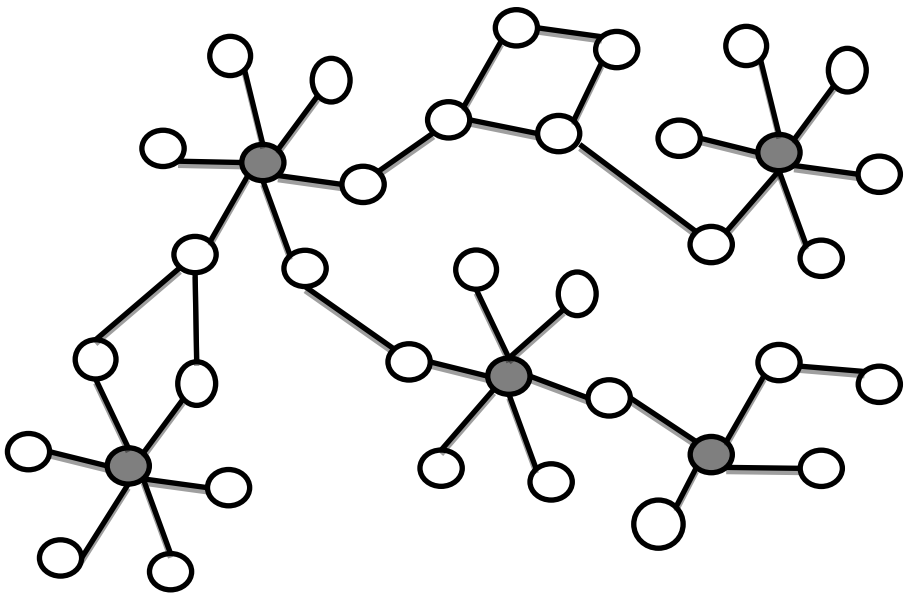


Fig. 12 – Exemplu graf scale free

3) **Erdős–Rényi** – O muchie între oricare pereche de două noduri se stabilește cu o anumită probabilitate egală pentru toate perechile de noduri, independentă de celelalte muchii alese. Este o metodă utilă pentru a testa algoritmii pe un graf complet aleator, dar controlat ca densitate de muchii conectate în graf.

Platforma oferă mai multe modalități de a vizualiza grafurile, puse la dispoziție de diverse framework-uri care se ocupă cu desenarea de structuri sub formă de grafuri.

5.2 Tehnologii folosite

Implementarea algoritmilor pentru potrivirea grafurilor a fost făcută în Java. De asemenea platforma care permite vizualizarea și efectuarea de statistici folosește același limbaj de programare. Nu a existat o constrângere pentru care am ales să dezvolt aplicația în Java. Având în vedere că proiectul are ca scop extinderea unui studiu mai mare început de Andrei Olaru în teza de doctorat [1] și implementările au fost făcute în Java, pentru simplitate și ușurința integrării am ales același limbaj de programare.

Mediul de dezvoltare este Eclipse, versiunea “Indigo Service Release 2”.

Pentru interfața grafică s-au folosit SWT, cu JFace, în cea mai mare parte. Ferestrele au fost construite prin intermediul unui plugin de Eclipse, dezvoltate de Google și anume GWT. Am pus la dispoziție 3 moduri de a vizualiza grafurile:

- **Zest**. Prin intermediul pluginului Zest care se integrează în Eclipse cu SWT/JFace.
- **Jung.** o aplicație de cercetare dezvoltată de Universitatea Regents din California. Se integrează cu containere Swing.
- **Dot de la Graphviz**. Este o aplicație specifică pentru .NET și întâlnită în aplicațiile care folosesc grafuri în C#. Se folosește acest utilitar pentru a obține o imagine cu graful desenat.

Pentru a putea genera grafuri contextuale care să fie etichetate corespunzător, cu cuvinte existente am folosit aplicația **RitaWN**. Aceasta este utilizată cu preponderență în procesarea de limbaj natural și se folosește de ontologia **Wordnet** pentru generarea de cuvinte. Astfel putem genera grafuri care să conțină substantive în noduri iar muchiile să fie etichetate cu verbe sau cuvinte de legătură .

Grafurile pot fi generate aleatoriu dar și citite din fișier. Pentru a utiliza un graf existent stocat într-un fișier, acesta trebuie să fie în format .dot. Am ales această reprezentare deoarece fișierul poate fi utilizat direct de către pluginul Zest pentru a obține un graf pe care putem să-l manipulăm prin program. De asemenea, Zest permite vizualizarea grafurilor în format dot.

Reprezentarea structurilor de date are la bază aplicația Jung deoarece folosim algoritmii puși la dispoziție de acest API pentru generarea de grafuri aleatorii. Astfel, am introdus structuri de date noi pentru noduri, muchii, CG și G_s^P care extind clasele abstracte și tipurile generice de concepte cu care lucrează aplicația Jung.

Pentru trasarea de grafice care să evidențieze rezultatele comparative ale algoritmilor am folosit aplicația **OpenChart2**. Codul ei este open source. Am introdus câteva modificări asupra funcționalității pentru a reuși crearea de grafice care să se potrivească necesităților noastre. Rezultatele rulării algoritmilor sunt stocate într-un XML pentru ca la sfârșitul execuției acestora pe un anumit set de date să poată fi construite grafice cu rezultatele obținute.

5.3 Funcționalitate

Deoarece este un proiect Java platforma poate fi rulată direct ca .jar sau dacă se dorește depanarea codului, aplicația poate fi lansată direct din Eclipse.

În partea din stânga interfeței grafice se pot încărca din fișier sau genera automat atât graful contextual cât și graful șablon. Se poate consulta figura [10]. Dacă se alege generarea automată a grafului șablon, nodurile și muchiile sale vor fi în strânsă legătură cu structura grafului contextual din

care sunt generate . Va exista o rată probabilistică de generare a numărului de noduri comune, necomune și etichetate cu “?”. Aceleași considerente probabilistice se întâlnesc și la generarea muchiilor.

În partea dreaptă, se alege tipul algoritmului cu care se dorește să se facă potrivirea. În partea de jos a panoului sunt afișate informații despre numărul de noduri care au fost expandate în căutare, despre numărul de muchii care au fost comparate. Tot în zona respectivă se poate vedea și timpul de execuție pe care l-a necesitat algoritmul, exprimat în milisecunde. Butonul “Run all”, în schimb, rulează toți algoritmii și stochează informațiile într-un fișier. În urma acestei etape se vor afișa rezultatele comparative sub formă de grafice, cu ajutorul OpenChart2. Butonul “Stop” are ca efect întreruperea algoritmului curent în cazul în care acesta durează prea mult și întoarce cel mai bun rezultat identificat până în acel moment. Pentru un agent real, această funcționalitate poate fi foarte utilă atunci când luarea unei decizii este limitată în timp și ne interesează o decizie corectă, chiar dacă aceasta nu este completă sau cea mai bună.

Din meniul ferestrei se pot alege două tipuri de vizualizări. Prima opțiune afișează grafurile prin facilitățile puse la dispoziție de framework-ul Jung. Un exemplu se poate vedea în fig. 12. Această vizualizare va deschide o nouă fereastră de tip Swing care va conține cele două grafuri și rezultatul potrivirii lor în trei containere separate în care grafurile se pot edita și rearanja după preferințe. Cealaltă opțiune oferă aceeași vizualizare care se poate obține folosind framework-ul Graphviz specific pentru limbajul C#. Un exemplu al acestei vizualizări sunt grafurile din capitolul 3, fig 1 și 2.

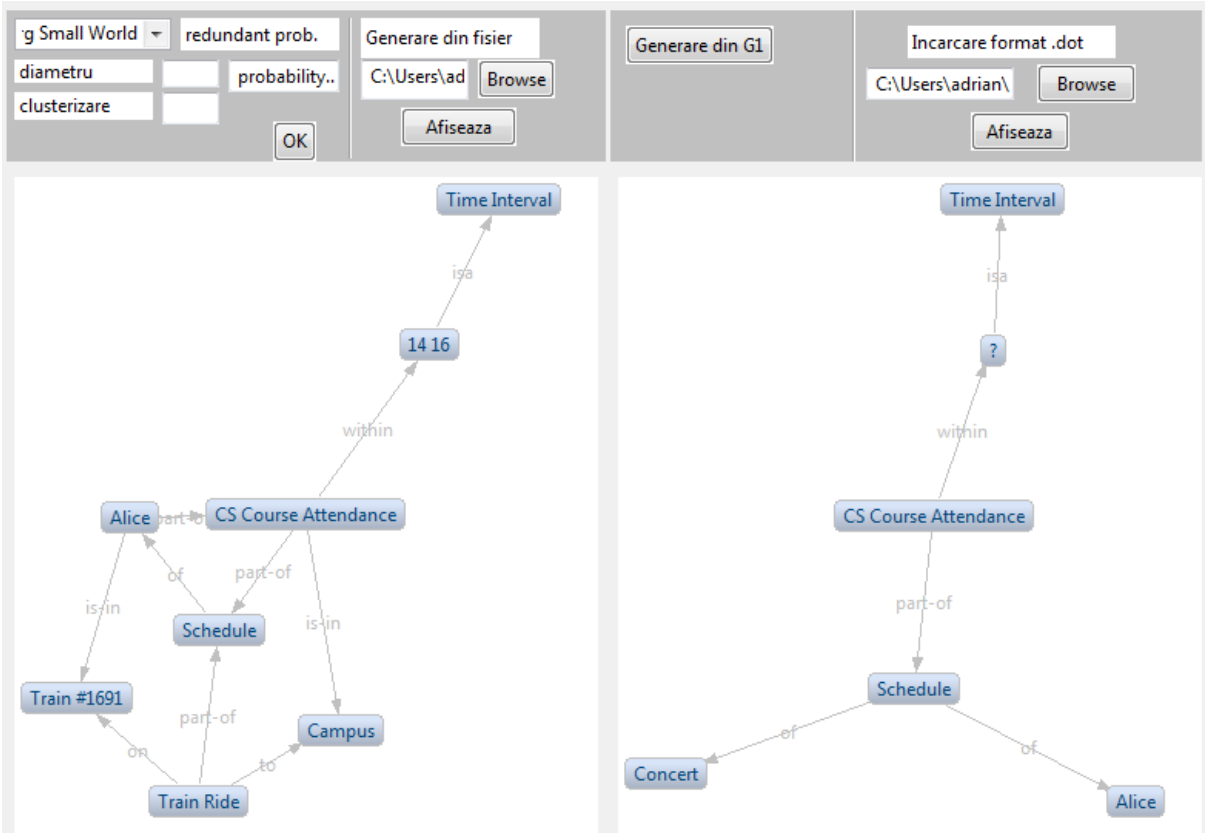


Fig 13. Vizualizarea grafurilor contextual și șablon

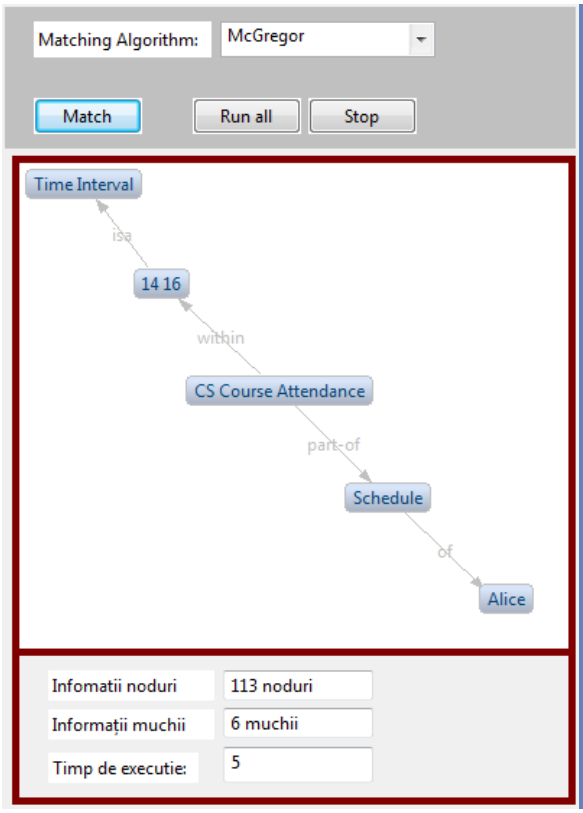


Fig 14 – Vizualizare cu Zest

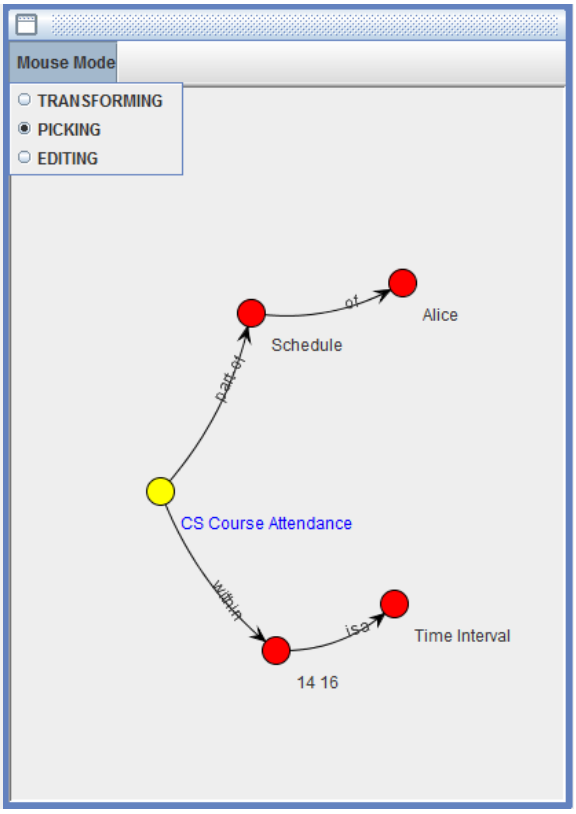


Fig 15 – Vizualizare cu Jung

5.4 Format .dot pentru reprezentare grafuri

Acest format de reprezentare a grafurilor este folosit în principiu de aplicațiile care folosesc Graphviz și au nevoie de desenarea de grafuri în C#. Acest format oferă foarte multe posibilități și metode de a particulariza modul cum va fi afișat graful. Mai multe detalii pot fi citite în [15], care este un manual de folosire a sintaxei dot. Pentru proiectul nostru ne interesează funcționalitatea de bază și anume posibilitatea de a avea muchii multiple cu aceeași etichetă într-un graf orientat. Un exemplu de graf în format “dot” este prezentat mai jos. Acesta reprezintă graful din cap. 3, fig. 1.

```
digraph G {
        Alice -> CS_Course_Attendance [label="part-of"];
        Alice -> "Train_#1691" [label="is-in"];
        CS_Course_Attendance -> "14_16" [label="within"];
        "14_16" -> Time_Interval [label="isa"];
        Schedule -> Alice [label="of"];
        Train_Ride -> "Train_#1691" [label="on"];
        CS_Course_Attendance -> Schedule [label="part-of"];
        Train_Ride -> Schedule [label="part-of"];
        Train_Ride -> Campus [label="to"];
        CS_Course_Attendance -> Campus [label="is-in"];
        "Train_#1691" [label="Train #1691"];
        CS_Course_Attendance [label="CS Course Attendance"];
        Train_Ride [label="Train Ride"];
        Time_Interval [label="Time Interval"];
        "14_16" [label="14 16"];
        Alice [label="Alice"] }
```

5.5 Extensibilitate

Platforma actuală oferă funcțiile de bază pentru a rula algoritmii implementați pe anumite exemple cât și pe grafuri generate aleatoriu. De asemenea sunt suportate 3 moduri de vizualizare a grafurilor. Datorită faptului că este o aplicație creată din nevoia de a avea un mod automat de comparare a algoritmilor, este util ca aplicația să nu se rezume doar la folosirea grafurilor contextuale ci să poată fi adaptată cu ușurință altor domenii care folosesc grafuri, în general.

De aceea arhitectura trebuie să fie cât mai extensibilă și scalabilă. Se pot aduce îmbunătățiri și optimizări de performanță pentru algoritmii implementați sau se pot implementa algoritmi noi în clase care oferă aceleași particularități structurale, adică:

- Una sau mai multe funcții exportate prin care clasa respectivă primește două grafuri contextuale și întoarce potrivirea obținută.
- Mai multe funcții prin care se pot extrage informații statistice despre numărul de comparații de noduri, muchii, timpul de execuție din secțiunea algoritmului care ne interesează.

Se încurajează crearea de noi statistici și noi informații care se pot extrage dintr-o problemă de grafuri precum memoria folosită, efortul maxim de procesor, numărul de soluții parțiale care au fost identificate, etc.

Alte îmbunătățiri care se pot aduce platformei pot fi considerate:

- Întreruperea execuției atunci când se rulează toți algoritmii pe un exemplu.
- Configurarea din exterior care să permită doar rularea unor algoritmi.
- Crearea unui benchmark diversificat care să testeze și alte aspecte ale algoritmilor.
- Adaptarea vizualizărilor și pentru grafuri neorientate sau alte tipuri de grafuri, astfel încât să poată fi configurabil din exterior.
- O aranjare mai bună a elementelor în cadrul interfeței grafice

Astfel se pot face multe alte mici ajustări pentru platformă. Studiul mai mare, însă, care poate deriva de la proiectul meu ar fi posibilitatea de a testa algoritmi care pot avea muchii cu expresii regulate iar potrivirea să se facă în consecință. Astfel, o singură muchie în graful șablon poate corespunde unui întreg subarbore din graful contextual.

În urma unor analize personale, dificultatea de a adapta algoritmii implementați la o potrivire care folosește expresii regulate este foarte mare deoarece grafurile se modifică complet și își modifică total structura. Deși potrivirea grafurilor cu expresii regulate pe muchii poate fi rezolvată prin backtracking deoarece este tot o problemă NP-completă, este interesant de văzut ce alți algoritmi se pot utiliza și ce particularități vor avea aceștia.

6. Rezultate obținute

6.1 Prezentarea comparativă a rezultatelor

În acest subcapitol vom prezenta din punct de vedere statistic, performanțele algoritmilor implementați. Este de precizat că algoritmii au fost rulați pe anumite exemple potrivite astfel încât să se poată arăta în grafic comportamentul lor și modul cum evoluează odată cu creșterea dimensiunii grafurilor. Numărul de noduri luat ca etalon de creștere a grafurilor este oarecum relativ deoarece:

- Există diferențe mari pentru un algoritm între cazul cel mai favorabil și nefavorabil.
- Pentru un număr fixat care reprezintă totalitatea nodurilor din graful șablon, există multe alte criterii care pot varia. Acestea sunt dimensiunea grafului contextual cu care se face potrivirea dar și numărul de noduri necunoscute. Astfel, luând ca exemplu algoritmul McGregor, pentru un graf șablon cu 12 noduri avem următoarea situație:
    - Pentru 2 noduri etichetate cu "?", timpul de execuție este trivial.
    - Pentru 10 noduri etichetate cu "?" timpul crește exponențial odată cu expandarea nodurilor necunoscute, rulând secunde întregi, în unele cazuri chiar minute. În acest caz algoritmul nu este fezabil pentru o aplicație reală.

Pentru graficele din subcapitolele de mai jos s-a ales o suită de teste care să poată arăta exact avantajele și dezavantajele algoritmilor. În tabelele de mai jos, rezultatele identificate sunt obținuta prin rularea algoritmilor pe grafuri cât mai apropiate de o situație reală. În majoritatea testelor, graful șablon are cam jumătate din numărul total de noduri al grafului contextual. Raportul dintre numărul de noduri necunoscute și numărul de noduri al grafului șablon este cam de 30%.

Trebuie menționat că valorile din tabelele expuse care sunt rezultatele exacte pe un graf sau în alte cazuri, media rulării algoritmilor pe 2-3 grafuri cât mai reprezentative. Valorile din grafice sunt puțin manipulate în unele locuri pentru a evidenția diferența dintre algoritmi. De exemplu, la algoritmul McGregor, deoarece creșterea exponențială a valorilor începe de la grafuri mai mici și este mult mai abruptă decât a celorlalți algoritmi, pentru a nu strica scala la care este desenat graficul, am ales valori reprezentative.

6.1.1 Comparație după numărul de noduri expandate

Tabel 1

Noduri	McGregor	Larrosa	Akkoyunlu	Bron-Kerbosch	Balas-Yu	Ina-Koch	Durand-Pasari
5	124	10	51	31	72	30	32
6	130	15	47	45	84	38	41
7	223	43	105	91	122	71	84
8	992	56	134	123	183	101	115
9	7872	82	282	273	320	220	235
10	42100	105	1222	1255	1420	830	1100
11	524100	122	6723	6620	8522	5400	6113

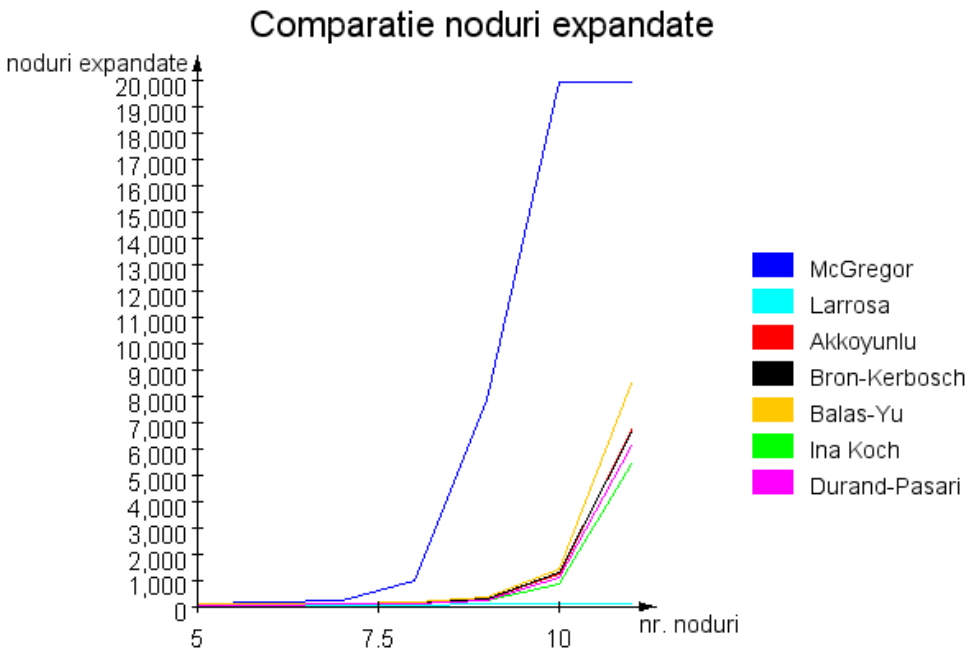


Fig. 16

6.1.2 Comparație după numărul de muchii evaluate

Tabel 2

Muchii	McGregor	Larrosa	Akkoyunlu	Bron-Kerbosch	Balas-Yu	Ina-Koch	Durand-Pasari
5	42	46	124	120	135	140	119
7	115	391	330	321	372	420	310
9	1250	1699	1995	1464	2542	2115	1330
11	42343	2108	5431	5440	6423	7345	5219
13	1976312	4324	20470	19989	22170	23575	18322

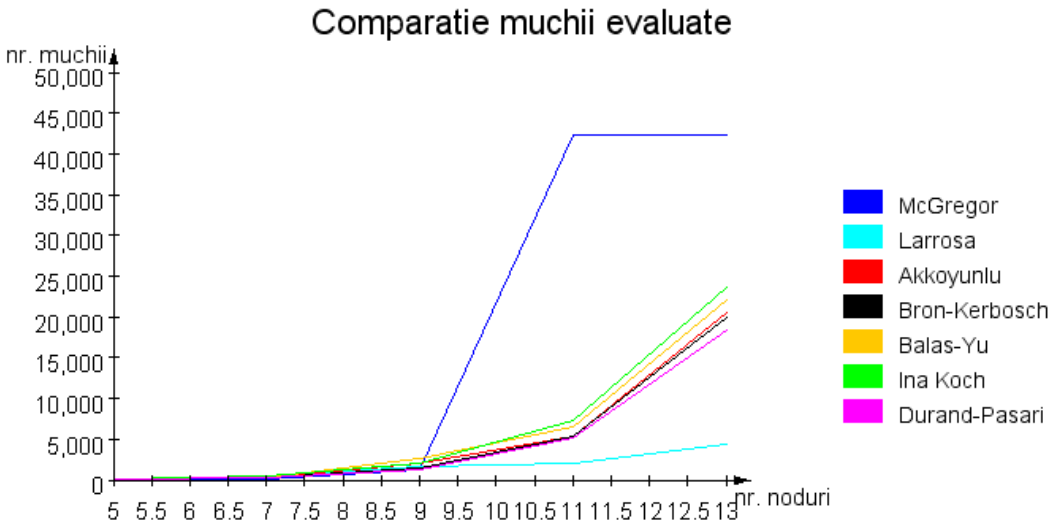


Fig. 17

6.1.3 Comparație timpi de execuție

Timp(ms)	McGregor	Larrosa	Akkoyunlu	Bron-Kerbosch	Balas-Yu	Ina-Koch	Durand-Pasari
5	14	1	9	7	23	10	9
10	82	11	22	34	42	36	30
15	5021	25	50	56	89	82	51
20	143023	36	56971	56392	54112	88523	52332
25	X	53	179434	181302	178342	221390	171000

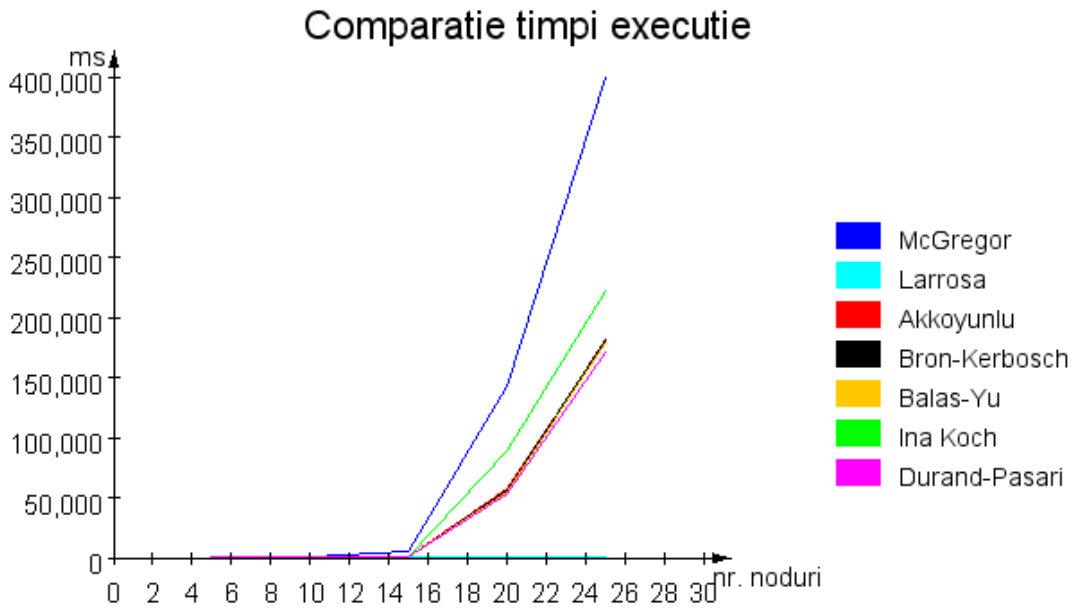


Fig. 18

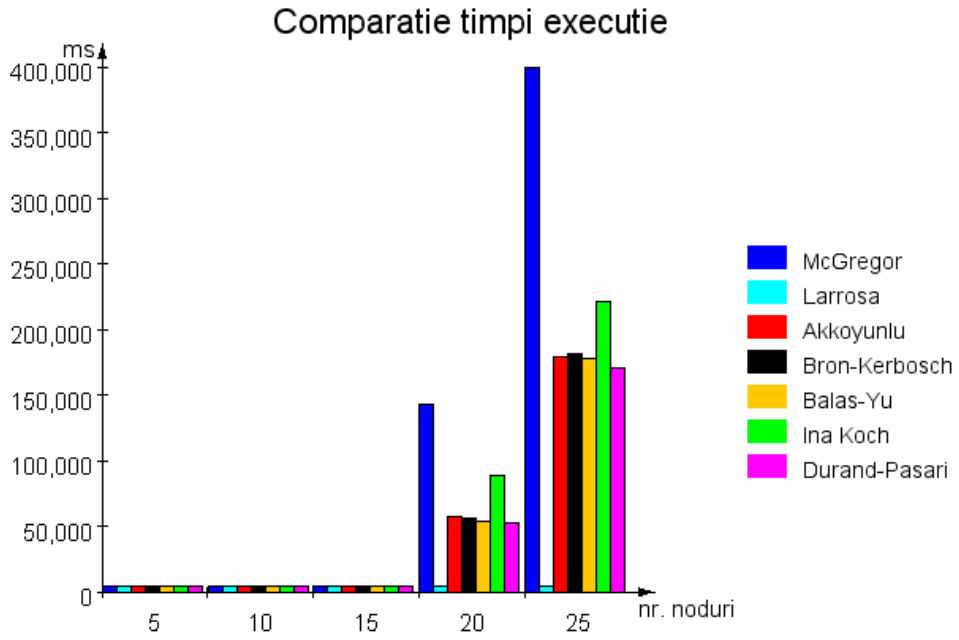


Fig. 19

6.2 Analiza rezultatelor obținute

Analizând rezultatele extrase după rularea algoritmilor, pute observa anumite particularități. Pentru comparația după numărul de noduri evaluate:

- Algoritmul Larrosa are cele mai bune performanțe de departe. La restul comparațiilor, tot el este cel mai bun, având o creștere polinomială, pentru dimensiunile considerate ale grafurilor. Pentru un număr de noduri mai mare de 40, se va comporta ca un algoritm de backtracking uzual și va începe o scădere abruptă a performanțelor.
  Singurul dezavantaj al algoritmului Larrosa constă în inabilitatea lui de a găsi soluții parțiale de potrivire între cele două grafuri și el nu rezolvă propriu-zis problema noastră, cu specificațiile dorite.
- Algoritmul McGregor, în forma implementată în proiectul curent, cu construcție de stări noi la fiecare tentativă de expansiune în spațiul de căutare, are performanțe foarte slabe pentru un număr relativ mic de noduri.
  Avantajul lui însă, constă în ușurința implementării și poate fi o soluție de compromis și ușor de dezvoltat atunci când contextul are dimensiuni foarte mici.
- Observăm că la 7 noduri, are loc o creștere cu un prag mare a numărului de noduri care au fost expandate. Acel test a fost făcut pe un graf șablon cu 6 noduri necunoscute dintr-un total de 7 pe care le are graful șablon.

La algoritmii care transformă problema curentă în identificare de clici maximale, observăm că algoritmul Balas-Yu evaluează cele mai multe noduri și cele mai multe muchii deoarece are o euristică foarte costisitoare. La timpul de execuție, în schimb, datorită reducerii complexității, acesta este foarte apropiat de restul algoritmilor dar nu putem spune că aduce îmbunătățiri de performanță considerabile. Datorită probabilității mici ca grafurile să fie de așa natură astfel încât euristica de la Balas-Yu să fie eficientă și în lipsa unui câștig semnificativ de timp, algoritmul poate fi considerat ca unul nu foarte potrivit și dificil de implementat.

Se observă că Algoritmul Akkoyunlu, deși sortează nodurile și reduce complexitatea alegând o anumită ordine a lor, efortul de căutare a minimului compensează parcurgerea mai ineficientă a algoritmului Bron-Kerbosch. De aceea, în multe situații Bron-Kerbosch are timpi mai buni decât versiunea sa îmbunătățită, studiată mai târziu și de Akkoyunlu.

Algoritmul Ina-Koch, are un număr mic de noduri expandate dar are cel mai mare număr de muchii care sunt comparate, deoarece abordează o strategie de identificare a clicilor după muchii. Se observă că dacă graful inspectat are un număr foarte mic de muchii așa cum a fost cazul grafului de test de 25 de noduri pentru care s-au obținut timpii de execuție, algoritmul are performanțe mai bune decât cele obținute de Bron-Kerbosch. În majoritatea cazurilor și cu preponderență în problema grafurilor contextuale, algoritmul Ina-Koch nu este unul eficient deoarece numărul de muchii este cu mult mai mare decât numărul de noduri, ba chiar pot exista muchii multiple între două noduri.

Algoritmul Durand-Pasari, în final, are aproximativ aceeași strategie ca ceilalți algoritmi care lucrează cu graful asocierilor. El beneficiază de o implementare mai optimizată dar mai neclară decât ceilalți algoritmi, deși teoretic nu aduce îmbunătățiri de performanță. Dat fiind faptul că diferențele de performanță ale acestui algoritm nu sunt semnificative, rămâne ca algoritmul de clică implementat în practică să rămână să fie decis de particularitățile problemei și a contextului.

În urma graficelor prezentate mai sus și ținând seama de comportamentului algoritmilor în diferite situații putem trage următoarele concluzii despre performanța algoritmilor:

- Nu există o abordare foarte proastă și una foarte bună pentru a rezolva această problemă prin intermediul unui algoritm și toate metodele au avantaje particulare pentru care merită să fie folosite.
- Observăm că algoritmul Larossa este cu mult mai eficient decât restul chiar dacă nu rezolvă problema în întregime însă poate fi extins prin tentative “brute force” să

găsească și soluții parțiale, deoarece în esență este un algoritm de backtracking, de satisfacere a restricțiilor. Poate că în unele situații, o abordare cu backtracking optimizat este mai eficientă și mai clară decât abordări euristice sau ocolitoare care încearcă să reducă din complexitatea exponențială prin efort de calcul suplimentar.

7. Concluzii

Soluția dată de reprezentarea contextului prin grafuri este una ce poate fi folosită cu succes în multe laturi ale inteligenței ambientale și nu numai. Bineînțeles, trebuie avut în vedere dezavantajul efortului computațional mare pe care îl prezintă utilizarea grafurilor. Unele aplicații sunt potrivite pentru folosirea acestor algoritmi iar în alte situații cerințele sunt mai specifice sau pur și simplu, contextul este unul specific pentru care se preferă reprezentări optime și particulare.

Partea foarte bună a acestui proiect a fost dezvoltarea unei platforme extensibile, pentru testarea de algoritmi. Aceasta poate fi un mijloc foarte util pentru dezvoltatorii de aplicații inteligente în care noțiunea de context este sintetizată sub forma unui graf, deoarece se poate decide ușor ce algoritm are performanțele optime pentru problema particulară. Faptul că avem algoritmi clasici care au putut fi adaptați problemei grafurilor contextuale cât și garanția teoretică că această adaptare este corectă, reprezintă un pas important în cercetarea din acest domeniu. Pornind de la această lucrare, se vor avea în vedere mult mai ușor ce avantaje și dezavantaje oferă fiecare metodă de rezolvare a problemelor de potrivire de grafuri. De asemenea am sumarizat studii diverse conduse în acest domeniu și am prezentat probleme și omisiuni pentru care unele noțiuni teoretice nu se aplică unui agent de inteligență ambientală.

În urma analizei rezultatelor obținute pentru algoritmi se pot reține ca fiind cele mai semnificative, comportările algoritmilor Larrosa și McGregor pentru diferite grafuri de test. În concluzie, algoritmul Larossa este foarte rapid chiar și atunci când graful crește în dimensiune dar nu poate identifica potriviri parțiale între CG și G_s^P. McGregor se remarcă detașat ca fiind cel mai 'nepractic' algoritm, ajungând la performanțe slabe pentru dimensiuni ale grafurilor nu foarte mari. Pentru acest algoritm, rămâne ca avantaj ușurința implementării pentru probleme tipice, reprezentate de grafuri care chiar au foarte puține noduri.

În cele din urmă, algoritmii pe grafuri reprezintă o unealtă puternică care aduce stabilitate structurală și generalitate pentru agent. Pentru un context definit peste mulțimea de structuri ale limbajului natural, algoritmii ce stau la baza raționamentului unei entități din sistem, pot folosi cu succes aceste metode de identificare a contextului.

În ultimii ani a apărut un interes tot mai mare pentru potrivirea de grafuri și cred că este o abordare de viitor în inteligența ambientală.

7.1 Dezvoltări ulterioare

Pe viitor, în această direcție, se pot aduce îmbunătățiri suplimentare algoritmilor existenți, se pot alege noi implementări auxiliare ale altor algoritmi sau ale acelorași. Având în vedere analiza din lucrarea curentă, ar fi foarte interesantă activitatea la o aplicație practică în care să fie folosiți unii dintre algoritmii prezentați. În capitolul 5.5 am prezentat alte modalități de a extinde studiul început de mine prin extinderea funcționalităților și a gradului de generalitate pe care trebuie să îl prezinte platforma de testare.

Primul proiect de cercetare care poate fi abordat constă în includerea algoritmilor analizați, potriviți pentru această problemă, într-un sistem multi-agent de inteligență ambientală, prezentat în [1] de A. Olaru, cu scopul de a identifica contextul agenților din respectivul sistem.

B I B L I O G R A F I E

[1]. Andrei Olaru - A Context-Aware Multi-Agent System for AmI Environments, București, 15 dec. 2011.

[2]. Mark Weiser – The Computer for the 21st century, 1999, ACM SIGMOBILE, Mobile Computing and Communications Review, Volume 3 Issue 3, July 1999, ACM New York, USA.

[3]. D. Conte, P. Foggia, C. Sansone, M. Vento - Thirty years of graph matching in pattern recognition, International Journal of Pattern Recognition and Artificial Intelligence, Vol. 18, No. 3 (2004) 265-298.

[4]. B. D. McKay, Practical graph isomorphism, Congressus Numerantium 30 (1981), 45-87

[5]. D. Conte, C. Guidobaldi, C. Sansone - A Comparison of Three Maximum Common Subgraph Algorithms on a Large Database of Labeled Graphs, I-84084 Fisciano (SA), Italy

[6]. C. Bron and J. Kerbosch, Finding all cliques of an undirected graph, Commun. ACM, 16 (1973) 575-577.

[7]. Andrei Olaru - Olaru, A., Florea, A. M., and El Fallah Seghrouchni, A. (2011). Graphs and patterns for context-awareness. In Novais, P., Preuveneers, D., and Corchado, J. M., editors, Proceedings of International Symposium on Ambient Intelligence, University of Salamanca (Spain), 6-8th April, volume 92 of Advances in Intelligent and Soft Computing, pages 165{172. Springer. ISBN 978-3-642-19936-3, ISSN 1867-5662.

[8]. *Akkoyunlu, E. A. (1973), "The enumeration of maximal cliques of large graphs", SIAM Journal on Computing* 2**.**

[9]. Barrow, H.; Burstall, R. (1976), "Subgraph isomorphism, matching relational structures and maximal cliques", Information Processing Letters 4 (4): 83–84, DOI:10.1016/0020-0190(76)90049-1.

[10]. Boros Endre, Khaled Elbassioni, Vladimir Gurvich, Leonid Khachiyan, „Extending the Balas-Yu Bounds on the Number of Maximal Independent Sets in Graphs to Hypergraphs and Lattices, 2002.

[11]. I. Koch, Enumerating all connected maximal common subgraphs in two graphs. Theoret. Comput. Sci. 250 (2001).

[12]. Y. Shinano, T. Fujie, Y. Ikebe and R. Hirabayashi, Solving the maximum clique problem using PUBB, in Proc. First Merged Int. Parallel Processing Symposium, 1998, pp. 326-332.

[13]. Skiena, S. "Induced Subgraphs." §3.2.2 in *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Reading, MA: Addison-Wesley, pp. 90-92, 1990.

[14]. L. P. Cordella, P. Foggia, C. Sansone, F. Tortorella and M. Vento, Graph matching: a fast algorithm and its evaluation, in Proc. 14th Int. Conf. Pattern Recognition, 1998, pp. 1582-1584.

[15]. H. Bunke, P. Foggia, M. Vento, C. Sansone and C. Guidobaldi, A comparison of algorithms for maximum common subgraph on randomly connected graphs, in Proc.Joint IAPR Int. Workshops SSPR and SPR, 2002, pp. 123-132.

[16]. J. Larrosa and G. Valiente, Constraint satisfaction algorithms for graph pattern matching, Math. Struct. Comput. Sci. 12 (2002) 403-422.

[17]. M. Lazarescu, H. Bunke and S. Venkatesh, Graph matching: fast candidate elimination using machine learning techniques, in Proc. Joint IAPR Int. Workshops SSPR and SPR, 2000, pp. 236-245.

[18]. J. J. McGregor, Backtrack search algorithm and the maximal common subgraph problem, Software | Practice and Experience 12 (1982) 23-34.

[19]. Brijnesh J. Jain, Klaus Obermayer, A Necessary and Sufficient Condition for Graph Matching to be equivalent to Clique Search, Berlin Univeristy of Technology, Dec. 2009.
[20]. M. Pelillo, Matching free trees, maximal cliques and monotone game dynamics, IEEE Trans. Patt. Anal. Mach. Intell. 24 (2002) 1535-1541.
[21]. E. Balas and C. S. Yu, Finding a maximum clique in an arbitrary graph, SIAM J. Comput. 15 (1986) 1054-1068.
[22]. H. T. Chen, H. Lin and T. L. Liu, Multi-object tracking using dynamical graph matching, in Proc. 2001 IEEE Computer Soc. Conf. Computer Vision and Pattern Recognition, 2001, pp. 210-217.
[23]. J.C. Régin, „A filtering algorithm for constraints of difference in CSPs", 1994
[24]. G. Kondrak, P. van Beek, „A theoretical evaluation of selected backtracking algorithms", University of Alberta, Canada.
[25]. J.J. McGregor, „Relational consistency algorithms and their applications in finding subgraph and graph isomorphism", University Of Sheffield, 2003.

ANEXE

Exemple demonstrative de grafuri pentru potrivire

Ex 1:

Graf contextual:

```
digraph G {
  Alice -> CS_Course_Attendance [label="part-of"];
  Alice -> "Train_#1691" [label="is-in"];
  CS_Course_Attendance -> "14_16" [label="within"];
  "14_16" -> Time_Interval [label="isa"];
  Schedule -> Alice [label="of"];
  Train_Ride -> "Train_#1691" [label="on"];
  CS_Course_Attendance -> Schedule [label="part-of"];
  Train_Ride -> Schedule [label="part-of"];
  Train_Ride -> Campus [label="to"];
  CS_Course_Attendance -> Campus [label="is-in"];
  "Train_#1691" [label="Train #1691"];
  CS_Course_Attendance [label="CS Course Attendance"];
  Train_Ride [label="Train Ride"];
  Time_Interval [label="Time Interval"];
  "14_16" [label="14 16"];
  Alice [label="Alice"]
}
```

Graf șablon:

```
digraph G {
  "?#1" -> Time_Interval [label="isa"];
  CS_Course_Attendance -> "?#1" [label="within"];
  Schedule -> Alice [label="of"];
  Schedule -> "my concert" [label = "of"]
  CS_Course_Attendance -> Schedule [label="part-of"];
  CS_Course_Attendance [label="CS Course Attendance"];
  Time_Interval [label="Time Interval"];
  "?#1" [label="?"];
  "my concert" [label = "Concert"]
}
```

Ex 2:

Graf contextual:

```
digraph G {
"Reading" -> "Activities" [label="part-of"];
"The Book" -> "Home" [label="is-in"];
"Bob" -> "User" [label="isa"];
"Meeting" -> "Activities" [label="part-of"];
"Reading" -> "Activity" [label="isa"];
"Home" -> "Place" [label="isa"];
"Alice" -> "User" [label="isa"];
"The Book" -> "Alice" [label="of"];
"Alice" -> "Meeting" [label="part-of"];
"Bob" -> "Home" [label="is-in"];
"Activities" -> "Bob" [label="of"];
"Reading" -> "The Book" [label="what"];
"Reading" -> "Done" [label="status"];
"Meeting" -> "Activity" [label="isa"];
}
```

Graf șablon:

```
digraph G {
"Take" -> "Activities" [label="part-of"];
"Meeting" -> "Activities" [label="part-of"];
"?#3" -> "Place" [label="isa"];
"?#2" -> "User" [label="isa"];
"Reading" -> "Done" [label="status"];
"Meeting" -> "Activity" [label="isa"];
"?#1" -> "User" [label="isa"];
"Reading" -> "Activity" [label="isa"];
"Reading" -> "?#4" [label="what"];
"?#1" -> "?#3" [label="is-in"];
"Take" -> "?#4" [label="what"];
"?#4" -> "?#3" [label="is-in"];
"Reading" -> "Activities" [label="part-of"];
"?#2" -> "Meeting" [label="part-of"];
"Activities" -> "?#1" [label="of"];
"Take" -> "Activity" [label="isa"];
"?#4" -> "?#2" [label="of"];
"?#1" [label="?"];
"?#3" [label="?"];
```

```
    "?#4" [label="?"];
    "?#2" [label="?"];
}
```

Ex 3:

Graf contextual:

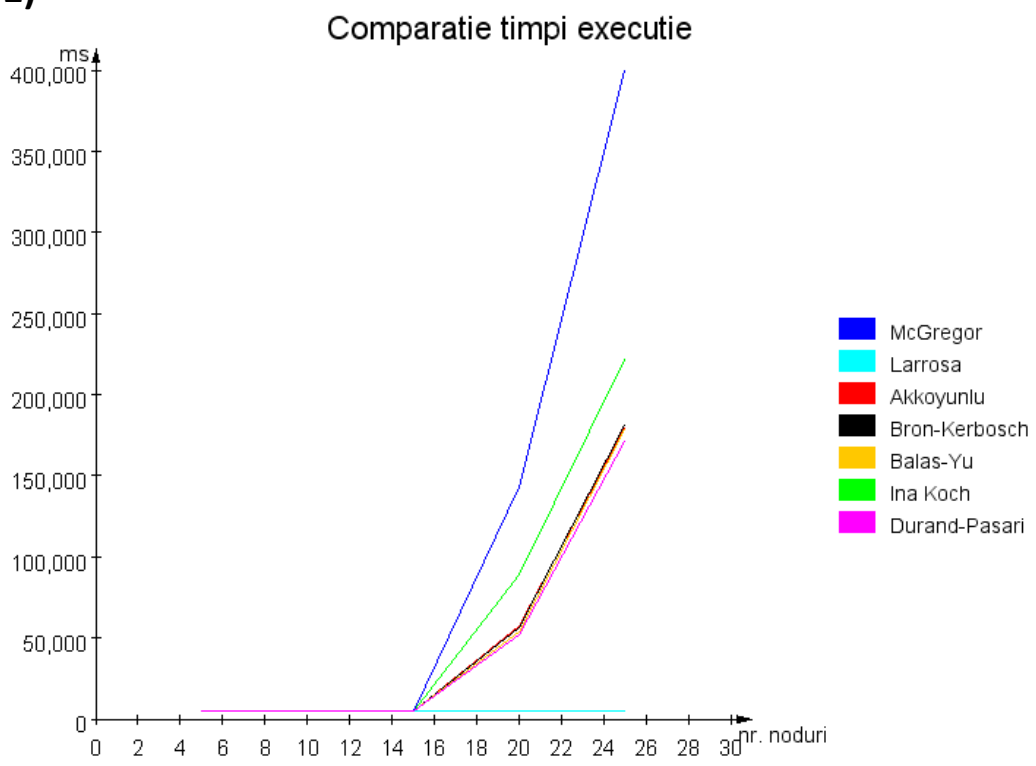
```
digraph G {
  "CFP" -> "document" [label="isa"];
  "AIConf" -> "30032011";
  "CFP" -> "30032011" [label="contains"];
  "CFP" -> "05012011" [label="contains"];
  "CFP" -> "conftime" [label="contains"];
  "05012011" -> "date" [label="isa"];
  "conftime" -> "interval" [label="isa"];
  "30032011" -> "date" [label="isa"];
  "AIConf" -> "CFP";
  "AIConf" -> "conftime";
  "CFP" -> "AIConf";
}
```

Graf șablon:

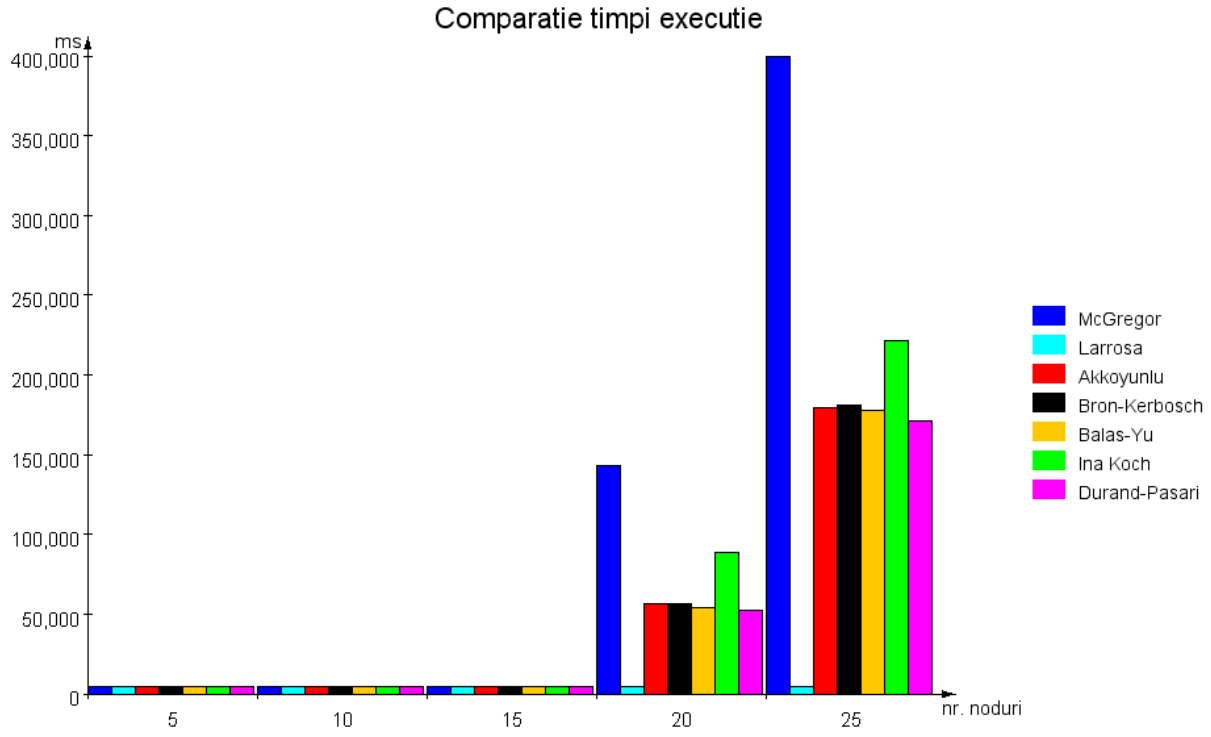
```
digraph G {
  "?#4" -> "document" [label="isa"];
  "?#1" -> "?#4" [label="article"];
  "?#2" -> "date" [label="isa"];
  "?#1" -> "conference" [label="isa"];
  "?#3" -> "?#2" [label="contains"];
  "?#1" -> "?#2" [label="deadline"];
  "?#3" -> "document" [label="isa"];
  "?#1" -> "?#3" [label="CFP"];
  "?#2" [label="?"];
  "?#3" [label="?"];
  "?#1" [label="?"];
  "?#4" [label="?"];
}
```

Tipurile de vizualizare a rezultatelor

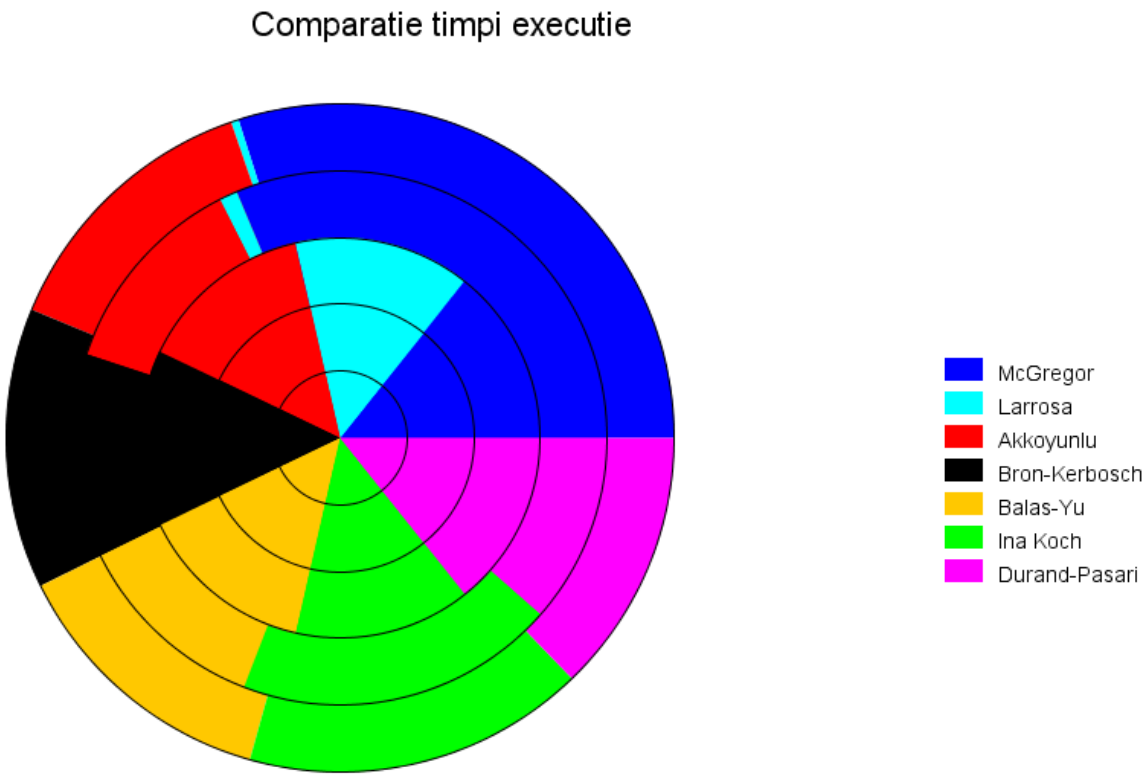
1)



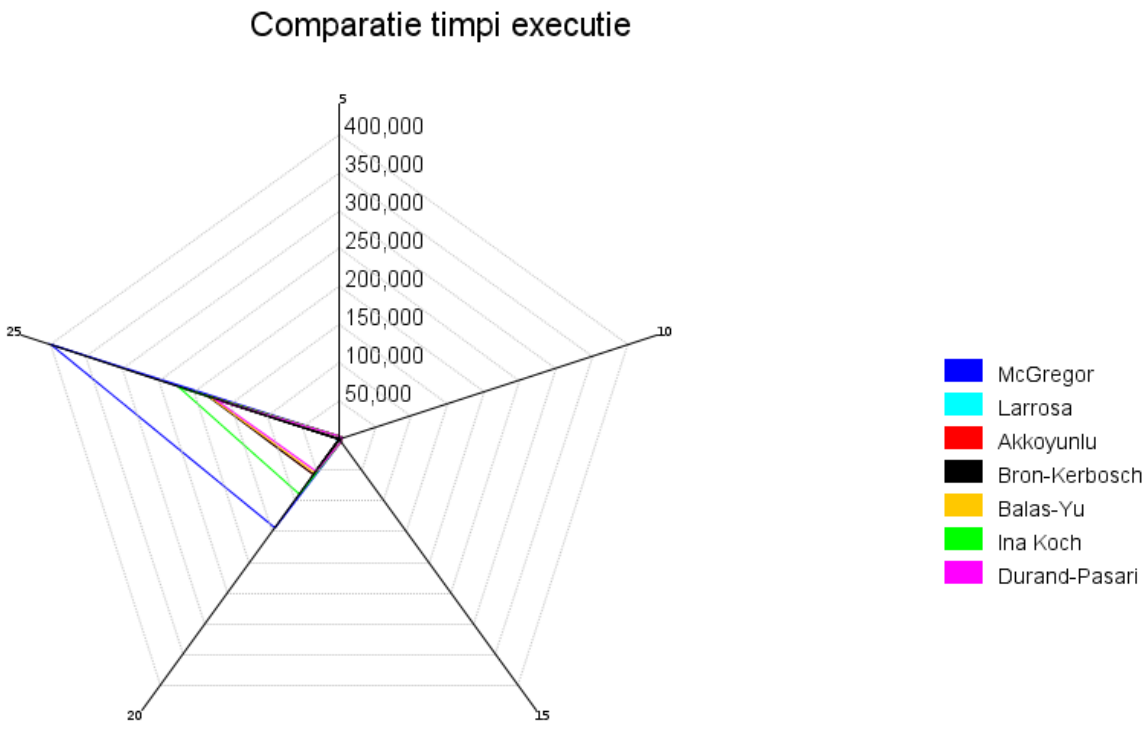
2)



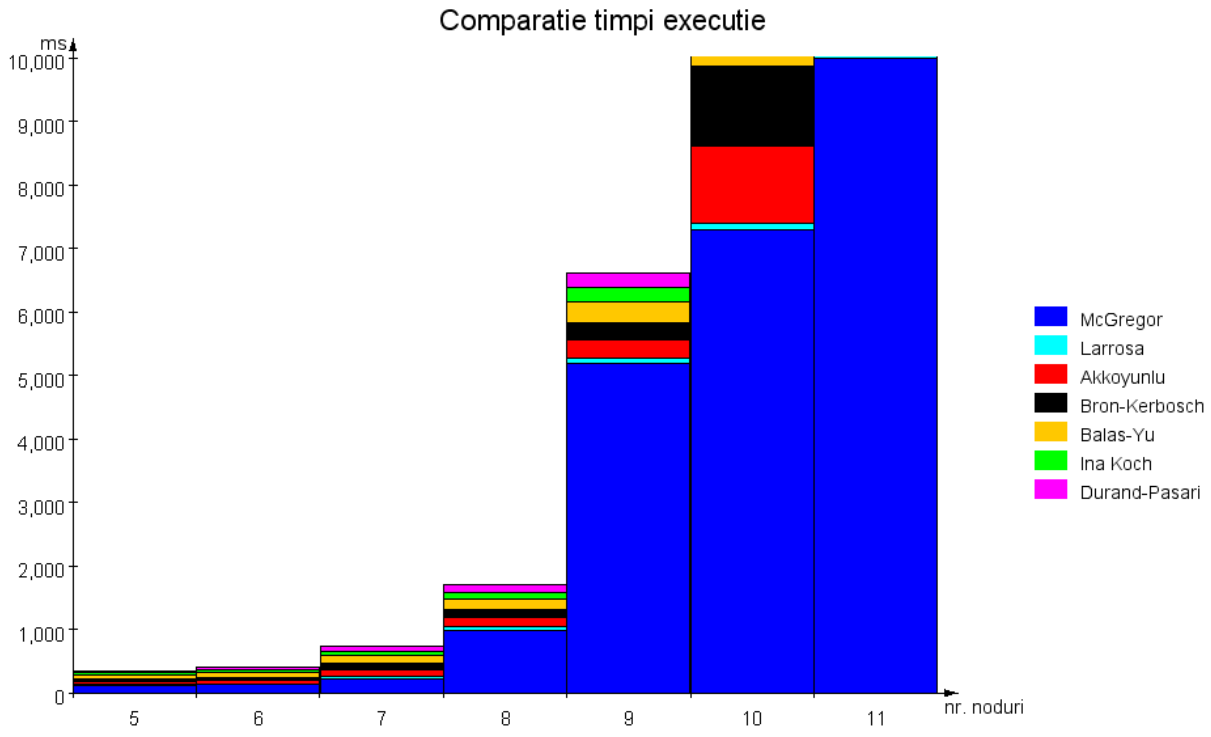
3)


Comparatie timpi executie
McGregor
Larrosa
Akkoyunlu
Bron-Kerbosch
Balas-Yu
Ina Koch
Durand-Pasari

4)


Comparatie timpi executie
5
400,000
350,000
300,000
250,000
200,000
150,000
100,000
50,000
10
15
20
25
McGregor
Larrosa
Akkoyunlu
Bron-Kerbosch
Balas-Yu
Ina Koch
Durand-Pasari

6)


Comparatie timpi executie
ms
10,000
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
5
6
7
8
9
10
11
nr. noduri
McGregor
Larrosa
Akkoyunlu
Bron-Kerbosch
Balas-Yu
Ina Koch
Durand-Pasari

Tipuri și structuri de date folosite

Reprezentarea nodurilor

```
public class Vertex implements Comparable<Vertex> {
    String name;
    private boolean bVisited;
    private boolean bUnknown;
    private int iExpandIndex;
    private boolean bDisabled;

    @Override
    public int compareTo(Vertex o) {
        return hashCode() - o.hashCode();
    }

    public Vertex(String name)
    {
        this.name = name;
        bVisited = false;
        bDisabled = false;

        if(name.equals("?"))
        {
             bUnknown = true;
        }
    }

    public boolean sameLabel(Vertex v2)
    {
        return name.equals(v2.name);
    }

    public boolean wasVisited()
    {
       return bVisited;
    }

    public void setVisited(boolean bVisited)
    {
       this.bVisited = bVisited;
    }

    public boolean isUnknown()
    {
       return bUnknown;
    }

    public void setUnknown(boolean bUnknown)
    {
       this.bUnknown = bUnknown;
    }
```

```
    public void setExpandIndex(int iIndex)
    {
       iExpandIndex = iIndex;
    }

    public boolean getDisabled()
    {
       return bDisabled;
    }

    public void setDisabled(boolean bDisabled)
    {
       this.bDisabled = bDisabled;
    }

    public int getExpandIndex()
    {
       return iExpandIndex;
    }

    @Override
    public String toString()
    {
        return name;
    }
}
```

Reprezentarea muchiilor

```
public class Edge implements Comparable<Edge>{
    private Vertex vSource;
    private Vertex vDest;

    String name;

    public Edge(String name)
    {
        this.name = name;
    }

    public Edge(Vertex s, Vertex d, String name)
    {
        vSource = s;
        vDest = d;
        this.name = name;
    }

    public Edge(Vertex s, Vertex d)
    {
        vSource = s;
        vDest = d;
        this.name = "E" + s.name + "-" + d.name;
    }

    public boolean sameName(Edge e)
    {
       return this.name.equals(e.name);
```

```
    }

    public boolean isCompatibleWith(Edge e)
    {
       return e.vSource.sameLabel(vSource) &&
              e.vDest.sameLabel(vDest) && sameName(e);
    }

    @Override
    public int compareTo(Edge o) {
        return hashCode() - o.hashCode();
    }

    public boolean containsUnknownVertex()
    {
       return vSource.isUnknown() || vDest.isUnknown();
    }

    @Override
    public String toString()
    {
        return name;
    }
}
```

Graful contextual

```
public class ContextGraph extends AbstractTypedGraph<Vertex, Edge>
             implements DirectedGraph<Vertex, Edge>, MultiGraph<Vertex, Edge>
{
    private static final long serialVersionUID = 549277843158182628L;
    HashMap<Vertex, ArrayList<Vertex>> outVertex;
    HashMap<Vertex, ArrayList<Edge>> outEdges;

    TreeSet<Edge> allEdges;
    TreeSet<Vertex> allVertex;

    public ContextGraph()
    {
       super(EdgeType.DIRECTED);
         outVertex = new HashMap<Vertex, ArrayList<Vertex>>();
         outEdges = new HashMap<Vertex, ArrayList<Edge>>();
         allEdges = new TreeSet<Edge>();
         allVertex = new TreeSet<Vertex>();
     }
}
```

Graful Șablon

```
public class GraphPattern extends ContextGraph{

    ContextGraph contextGraph;

    public GraphPattern()
    {
        super();
     }}
```