CONTRIBUŢII LA IMPLEMENTAREA, EVALUAREA ŞI ÎMBUNĂTĂŢIREA PERFORMANŢELOR TEHNICILOR DE CĂUTARE ASINCRONE ÎN CADRUL PROGRAMĂRII BAZATE PE CONSTRÂNGERI DISTRIBUITE

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea "Politehnica" din Timişoara
în domeniul ŞTIINŢA CALCULATOARELOR
de către

Ionel Muscalagiu

Conducător ştiinţific:	prof.univ.dr.ing. Vladimir Creţu
Referenţi ştiinţifici:	prof.univ.dr.ing. Mircea Petrescu
	prof.univ.dr.ing. Valentin Cristea
	prof.univ.dr.ing. Horia Ciocarlie

Ziua susţinerii tezei: 01.02.2008

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică şi Telecomunicaţii
8. Inginerie Industrială
9. Inginerie Mecanică
10. Ştiinţa Calculatoarelor
11. Ştiinţa şi Ingineria Materialelor

Universitatea „Politehnica” din Timişoara a iniţiat seriile de mai sus în scopul diseminării expertizei, cunoştinţelor şi rezultatelor cercetărilor întreprinse în cadrul şcolii doctorale a universităţii. Seriile conţin, potrivit H.B.Ex.S Nr. 14 / 14.07.2006, tezele de doctorat susţinute în universitate începând cu 1 octombrie 2006.

România, 300159 Timişoara, Bd. Republicii 9,
tel. 0256 403823, fax. 0256 403221
e-mail: editura@edipol.upt.ro

Cuvânt înainte

Teza de doctorat a fost elaborată pe parcursul activităţii mele în cadrul Departamentului de Calculatoare al Universităţii „Politehnica” din Timişoara.

Teza de faţă abordează problematica sistemelor multi-agent bazate pe tehnicile de căutare asincrone existente în cadrul programării cu constrângeri distribuite. Principalele idei pe care se bazează teza sunt utilizarea mediului NetLogo ca un simulator de bază în studiul tehnicilor de căutare asincrone, utilizarea completă şi eficientă a informaţiilor oferite de listele de tip nogood, limitarea fluxului de mesaje, introducerea sincronizării execuţiei agenţilor. Cercetările efectuate în cadrul tezei au urmărit conceperea şi dezvoltarea unui model pentru implementarea tehnicilor de căutare asincrone şi conceperea unei metodologii unitare de implementare şi evaluare a tehnicilor de căutare asincrone în NetLogo. Această metodologie a permis construirea unor sisteme multi-agent pentru studiul detaliat al comportamentului tehniciilor de căutare asincrone în diverse situaţii, identificând şi propunând mai multe îmbunătăţiri ale performanţelor acestora.

Mulţumesc domnului prof. dr. ing. Vladimir Creţu, conducătorul ştiinţific de doctorat, pentru ajutorul şi îndrumarea continuă şi plină de răbdare, acordate pe parcursul acestei perioade de doctorat. Activitatea mea de cercetare la doctorat a beneficiat de sprijinul mai multor colegi din cadrul Facultăţii de Inginerie Hunedoara. Pentru încurajările şi sfaturile primite pe parcursul redactării lucrării aduc cele mai sincere mulţumiri domnului conf. dr. ing Pănoiu Caius. Menţionez, de asemenea, colaborarea preţioasă şi sprijinul colegilor conf. dr. ing Pănoiu Manuela, sef lucrări dr. Osaci Mihaela, conf.dr. ing Rusu Nicolae, lector univ. Popa Horia Emil.

Programul de colaborare cu laboratorul de sisteme multi-agent existent în cadrul Departamentului de Ştiinţa şi Ingineria Calculatoarelor, Universitatea din Carolina de Sud (SUA), coordonat de prof. Jose VIDAL, a reprezentat un pilon important pe care s-a sprijinit activitatea mea de doctorat. Multumesc pe această cale profesorului Jose Vidal, pentru încrederea şi sprijinul acordat.

Îmi exprim întreaga consideraţie faţă membrii comisiei de doctorat, domnul preşedinte al comisiei prof. univ. dr. ing. Prostean decanul Facultăţii de Automatică şi Calculatoare din Timişoara şi domnii prof.univ.dr. Mircea PETRESCU de la Universitatea Politehnică din Bucureşti, prof. univ. dr. ing. Valentin CRISTEA de la Universitatea Politehnică din Bucureşti şi prof. univ. dr. ing. Horia CIOCÂRLIE de la Facultatea de Automatică şi Calculatoare din Timişoara, care au răspuns solicitării de a face parte din comisia de analiză a tezei, pentru observaţiile făcute şi pentru timpul acordat lucrării.

Contribuţia cea mai importantă la această teză o are soţia mea, Diana, care m-a înconjurat în toţi aceşti ani cu dragoste, răbdare şi devotament. Îi mulţumesc că m-a înţeles şi m-a ajutat. Mulţumesc familiei mele, în special fiicei mele, Andreea pentru ajutorul acordat în anumite momente ale redactării tezei. Le datorez adânca mea recunoştinţă părinţilor mei, Aurelia şi Dumitru, mai ales pentru că prin sacrificile lor am devenit ceea ce sunt astăzi.

Nu în ultimul rând, mulţumesc bunului Dumnezeu pentru că mi-a dat putere să lupt până la capăt şi norocul de a mă bucura de toate lucrurile minunate cu care El mă înconjoară.

Timişoara, decembrie 2007 Ionel Muscalagiu

Familiei mele

Muscalagiu, Ionel

Contribuţii la implementarea, evaluarea şi îmbunătăţirea performanţelor tehnicilor de căutare asincrone în cadrul programării bazate pe constrângeri distribuite

Teze de doctorat ale UPT, Seria10, Nr. 2, Editura Politehnica, 2008, 234 pagini, 104 figuri, 14 tabele.

ISSN: 1842-7707

ISBN: 978-973-625-592-2

Cuvinte cheie:
programare cu constrângeri, tehnici de căutare asincrone, mesaje, sisteme multi-agent, Netlogo, mesaje *nogood*.

Rezumat:
Programarea bazată pe constrângeri reprezintă un model al tehnologiilor software, folosit pentru descrierea şi rezolvarea unor clase largi de probleme cum ar fi problemele de căutare, problemele combinatoriale, problemele de planificare, etc. Reţelele cu constrângeri centralizate sau distribuite şi-au dovedit succesul în modelarea problemelor reale cum ar fi cele de planificare, configurare, diagnoză, teoria grafurilor, design-ul circuitelor, planificare genetică, design-ul maşinilor, înţelegerea limbajelor. Cele mai multe dintre problemele abordate de programarea cu constrângeri sunt probleme NP-complete. Programarea cu constrângeri distribuite (folosind tehnici de căutare asincrone) a permis reducerea costurilor ce apar la obţinerea soluţiei pentru astfel de probleme.
Cercetările efectuate în cadrul tezei au urmărit conceperea şi dezvoltarea unui model pentru implementarea tehnicilor de căutare asincrone şi conceperea unei metodologii unitare de implementare şi evaluare a tehnicilor de căutare asincrone pe baza modelului identificat anterior. Această metodologie a permis construirea unor sisteme multi-agent pentru studiul detaliat al comportamentului tehniciilor de căutare asincrone în diverse situaţii. Plecând de la acest studiu s-au identificat şi propus mai multe îmbunătăţiri ale performanţelor tehnicilor de căutare asincrone.

CUPRINS

Abrevieri

AAS - Asynchronous Search with Aggregations

ABT – Asynchronous Backtracking

ABT_DO - Asynchronous Backtracking with Dynamic Ordering

ABT_{temp} – Asynchronous Backtracking cu legături temporare.

ABTTPL - ABT cu legături temporare şi permanente.

ABTWFL - Asynchronous Backtracking cu flaguri

AC – Arc consistency

AI - Artificial Intelligence

AWCS - Asynchronous Weak –Commitment Search**.**

ConcBT - Concurrent Backtracking

CSP – Constraint Satisfaction Problems

DAC – Distributed arc-consistency

DB – Distributed Breakout

DCSP – Distributed Constraint Satisfaction Problem

DIBT - Distributed Backtracking

DisDBWFL - Distributed Dynamic Backtracking cu flaguri

DIFC - Distributed Forward Checking

DisDB - Distributed Dynamic Backtracking

SIES- Sistem de implementare şi evaluare cu sincronizarea execuţiei agenţilor.

SIEAS- Sistem de implementare şi evaluare cu operarea asincronă a agenţilor.

Lista figurilor

Lista tabelelor

SECŢIUNEA I.

INTRODUCERE

Secţiunea de faţă prezintă capitolele introductive ale lucrării, în care se prezintă tema şi principalele domenii abordate.

Capitolele 2 şi 3 prezintă o trecere în revistă a principalelor concepte actuale în domeniul programării cu constrângeri centralizate şi distribuite şi în domeniul dezvoltării sistemelor multi-agent.

În capitolul 4 sunt prezentate o serie de probleme actuale din domeniul programării bazate pe constrângeri distribuite. Preocupările din cadrul cercetărilor de doctorat, prezentate în acest material, se axează pe studierea şi soluţionarea acestor probleme.

1. INTRODUCERE

Teza de faţă prezintă contribuţii proprii, rezultate din activitatea de cercetare-dezvoltare cuprinsă în programul de doctorat cu tema „Contribuţii la implementarea, evaluarea şi îmbunătăţirea performanţelor tehnicilor de căutare asincrone în cadrul programării bazate pe constrângeri distribuite”, sub coordonarea ştiinţifică a prof. dr. ing. Vladimir CREŢU.

Societatea actuală a devenit astăzi mult mai dens interconectată prin intermediul reţelelor de calculatoare. Lucrul în colaborare, ce a început cu poşta electronică şi cu grupurile de ştiri s-a mutat în fluxul de date, în aplicaţii multimedia şi în construirea de medii distribuite. Surse de informaţii heterogene şi separate fizic au început să lucreze împreună. Sarcinile au fost împărţite în cadrul reţelelor între ceea ce se cheamă „agenţi”.

Un sistem distribuit este constituit dintr-o mulţime de procese autonome, care lucrează împreună într-o reţea, procese ce nu se execută în mod necesar pe acelaşi sistem de calcul. Aceste sisteme sunt fundamental diferite de sistemele cu un singur proces. În primul rând, aceste sisteme sunt inerent concurente şi nondeterministe. Nu există informaţii globale şi nici un timp global. Întârzierile de comunicaţii între procese sunt imprevizibile. În al doilea rând, sistemul este partajat, astfel că datele utilizatorilor trebuie să fie protejate de accesul altor utilizatori şi de agenţii lor de calcul. Scrierea aplicaţiilor distribuite este mult mai dificilă decât programarea clasică pentru că implică manipularea mai multor concepte diferite, incluzând funcţionalitatea aplicaţiilor, structuri distribuite, toleranţa la erori, securitate, sisteme deschise şi altele.

Programarea bazată pe constrângeri (CSP) reprezintă un model al tehnologiilor software, folosit pentru descrierea şi rezolvarea unor clase largi de probleme cum ar fi problemele de căutare, problemele combinatoriale, problemele de planificare, etc. O mare varietate de probleme din domeniul AI şi din alte domenii ale ştiinţei calculatoarelor, pot fi văzute ca un caz special de programare cu constrângeri. Este un model recent dezvoltat, aflat la confluenţa mai multor domenii ale ştiinţei calculatoarelor, incluzând AI, teoria algoritmilor, limbaje de programare distribuite, calcul simbolic, teoria grafurilor, programare logică [DF99], [Fro97], [JL94], [Nad89], [Tsa93], [YDIK98], [Yok01].

În ultimii ani comunitatea AI a arătat un interes crescut pentru problemele distribuite rezolvabile cu ajutorul modelării cu constrângeri distribuite. Ideea de a împărţi diferite părţi ale problemei între agenţi care acţionează independent şi care colaborează între ei folosind mesaje, în scopul obţinerii soluţiei, s-a dovedit viabilă, obţinându-se un nou tip de modelare numit Distributed Constraint Satisfaction Problem (DCSP) [YDIK92], [SGM96], [YDIK98], [Yok01]. De fapt, plecând de la modelarea bazată pe constrângeri, în cadrul modelării concurente/paralele, prin distribuirea variabilelor şi a constrângerilor între diverşi agenţi s-a obţinut acest nou tip de modelare numit DCSP, modelare identificată şi sub denumirea de reţea cu constrângeri distribuite.

Reţelele cu constrângeri centralizate sau distribuite şi-au dovedit succesul în modelarea problemelor reale. Multe probleme reale, cum ar fi cele de planificare, configurare, diagnoză, teoria grafurilor, design-ul circuitelor, planificare genetică,

design-ul maşinilor, înţelegerea limbajelor, etc. pot fi văzute ca şi categorii de probleme de satisfacere a constrângerilor. Cele mai multe dintre problemele abordate de programarea cu constrângeri sunt probleme NP-complete. Programarea cu constrângeri distribuite a permis reducerea costurilor ce apar la obţinerea soluţiei pentru astfel de probleme.

În cadrul modelării cu constrângeri distribuite au fost propuse mai multe metode de rezolvare a problemelor reale, metode numite tehnici de căutare asincronă. Ele se disting prin felul în care înregistrează combinaţiile de valori ce nu pot fi părţi ale soluţiei, combinaţii numite *nogood*, şi prin metodele prin care construiesc si stochează aceste valori nogood.

Mai multe lucrări au fost publicate în legătură cu formalizarea modelării DCSP şi a tehnicilor de rezolvare [AD97], [BMM01], [BM05], [BM03], [HBQ98], [SHF00], [Sil02], [SGM96], [YDIK98], [Yok01], [ZM04]. Toate aceste studii relative la modelarea cu agenţi şi constrângeri sunt motivate de existenţa problemelor distribuite în mod natural, mare parte dintre ele fiind probleme NP-complete. Este dificil sau imposibil de a reuni datele problemei într-un singur loc şi de a aplica un algoritm centralizat. Motivele cel mai des întâlnite sunt timpul de transmitere şi costul translatării datelor într-un format comun. De asemenea, furnizarea datelor unei probleme pentru un singur agent nu poate fi realizată şi din motive de securitate sau confidenţialitate.

Primele tehnici de căutare asincronă, a căror completitudine a fost demonstrată, se datorează lui Yokoo, dintre acestea remarcându-se tehnica Asynchronous Backtracking (ABT) şi tehnica Asynchrounous Weak-Commitment Search (AWCS) [YDIK98], [Yok01]**.** Alte studii se datorează lui Hamadi [HBQ98] care propune o altă tehnică numită Distributed Backtracking (DIBT), tehnică ce se bazează pe o ordine dinamică în graful constrângerilor. Trebuie menţionate rezultatele lui Silaghi din [SHF00]**,** care au dus la crearea unei alte tehnici de căutare de complexitate polinomială, tehnică numită Asynchronous Search with Aggregations (AAS). Algoritmul se bazează pe schimbul ansamblurilor (agregatelor) ce includ soluţii parţiale într-un model distribuit şi orientat spre variabile. Alte studii, datorate lui Bessiere, au dat naştere unei alte tehnici asincrone ce combină ideile din ABT cu DIBT, tehnică numită backtracking dinamic distribuit (DisDB **-** Distributed Dynamic Backtracking), publicată în [BMM01]**,** interesantă din mai multe puncte de vedere (completitudine, eficienţă, spaţiu polinomial). De asemenea, în [BM05]**,** se propune unificarea cadrului ABT, prin construirea unui nucleu general ABT, din care se pot deriva tehnici cunoscute ca ABT, DIBT, DisDB, obţinându-se familia ABT.

În prezent există o cantitate covârşitoare de informaţii ce tratează domeniul programării bazate cu constrângeri centralizate şi o cantitate mult mai mică de informaţii ce tratează domeniul distribuit (DCSP). Majoritatea documentaţiei necesare activităţii de cercetare a fost colecţionată începând cu anul 2000, având acces la o serie de resurse importante, recunoscute la nivel mondial în cadrul comunităţii academice: societăţi ştiinţifice IEEE, ACM, publicaţii cotate ISI, site-uri WEB ale unor instituţii academice. Trebuie menţionat faptul că în România domeniul sistemelor multi-agent bazate pe constrângeri nu a fost abordat, din cunoştiinţele noastre neexistând nici o referinţă. Informaţia colectată a fost procesată până în 2003, dar şi după aceea. Din materialele studiate a rezultat faptul că problematica sistemelor multi-agent bazate pe constrângeri distribuite nu este rezolvată complet, iar o parte a soluţiilor existente nu asigură o eficienţă satisfăcătoare. Astfel au fost identificate mai multe probleme, după cum urmează:

- Inexistenţa unor modele standard de implementare şi evaluare a tehnicilor de căutare asincrone. De asemenea, inexistenţa unei metodologii unitare de implementare şi evaluare a tehnicilor de căutare asincrone pe baza unor modele.
- Inexistenţa unor sisteme multi-agent publice bazate pe constrângeri, accesibile mediului academic, care să permită studiul tehnicilor de căutare asincrone.
- Existenţa unei varietăţi de tehnici de căutare asincrone, nici una nefiind acceptată integral ca tehnica cea mai potrivită în rezolvarea problemelor modelate DCSP. Principala problemă constă în inexistenţa unor studii complete relative la comportamentul agenţilor în diverse situaţii practice, studii care să identifice tehnica cea mai performantă şi eficientă unei clase de probleme.

Activitatea de cercetare, sintetizată în lucrarea de faţă, îşi propune să aducă o serie de contribuţii şi soluţii pentru rezolvarea problemelor de mai sus.

Principalele obiective propuse în cadrul acestui program de cercetare-dezvoltare sunt legate de ideile cheie enunţate anterior:

(1) Conceperea şi dezvoltarea unui model pentru implementarea tehnicilor de căutare asincrone.

(2) Conceperea unei metodologii unitare de implementare şi evaluare a tehnicilor de căutare asincrone pe baza modelului identificat anterior. Această metodologie va permite construirea unor sisteme multi-agent pentru studiul tehniciilor de căutare asincrone.

(3) Studiul detaliat al comportamentului anumitor tehnici de căutare asincrone în diverse situaţii cum ar fi apariţia întârzierilor în furnizarea mesajelor, existenţa unui graf al constrângerilor foarte dens, memorie de lucru disponibilă limitată.

(4) Identificarea şi propunerea unor posibile îmbunătăţiri ale performanţelor tehnicilor de căutare asincrone.

Abordarea din lucrare se axează pe următoarele idei principale:

- Utilizarea mediului NetLOGO ca un simulator de bază în studiul tehnicilor de căutare asincrone existente în cadrul programării bazate pe constrângeri distribuite.
- Utilizarea completă şi eficientă a informaţiilor oferite de listele de tip *nogood*, inclusiv renunţarea la stocarea acestora în anumite cazuri identificate în lucrarea de faţă.
- Limitarea fluxului de mesaje pentru legăturile temporare.
- Introducerea managementului mesajelor pe baza analizei cozilor de mesaje împreună cu sincronizarea execuţiei agenţilor.
- Problemele modelate cu constrângeri fiind în general probleme NP-complete, presupun analize într-o fază de preprocesare care să identifice tehnica de căutare potrivită, versiunea de implementare şi condiţiile de rulare.

Rezultatele obţinute în cadrul activităţii de doctorat sunt sintetizate în teza de faţă, cele mai multe dintre ele fiind publicate, după cum urmează:

- analiza mediului NetLogo a permis construirea unui model general de implementare şi evaluare pentru tehnicile de căutare asincrone astfel încât să se poată utiliza mediul NetLogo ca un simulator de bază în studiul tehnicilor asincrone. Acest model a fost publicat în [MusBP05] şi [MJP06].

- analiza comportamentului tehnicilor din familia ABT şi a celor din familia AWCS a permis introducerea managementului mesajelor în cazul tehnicilor din familia AWCS şi definirea unui protocol de tratare completă a mesajelor din canalele de comunicaţie pentru familia AWCS. Aceste rezultate au fost publicate în [MusCP06]. De asemenea, aceasta analiză a permis introducerea sincronizării în cazul familiei AWCS, rezultate publicate în [MusVCP07].
- analiza listelor nogood arată că nu este necesară stocarea acestor valori, o soluţie de eliminare a exploziei valorilor nogood fiind aceea de etichetare a valorilor din domeniul fiecărei variabile cu flaguri, soluţie propusă in această teză. Această tehnică a fost publicată în [Mus05] şi [MusB05].
- analiza comportamentului tehnicii ABT cu legături temporare a permis identificarea şi obţinerea mai multor soluţii originale de determinare a numărului de mesaje schimbate de agenţii cu legături temporare, număr. Apoi, o soluţie de combinarea a legăturilor temporare si permanente a fost propusă şi publicată. Aceste soluţii au fost publicate în [MusPP05a] şi [MusPP06].
- Analiza comportamentului tehnicii AWCS a permis aplicarea tehnicii nogood processor la această tehnică şi combinarea ei cu tehnicile de învăţare. Aceste soluţii ce au permis reducerea costurilor, au fost publicate în [MCPP06] şi [MusC06].

Datorită volumului mare de informaţii prezentate, teza este structurată în secţiuni.

Prima secţiune conţine capitolele introductive ale lucrării, în care se prezintă tema şi principalele domenii abordate. Astfel, capitolele 2 şi 3 prezintă o trecere în revistă a principalelor concepte actuale în domeniul programării cu constrângeri centralizate şi distribuite şi în domeniul dezvoltării sistemelor multi-agent.

În capitolul 2 se definesc formal problemele de satisfacere a constrângerilor, cazul centralizat şi sunt trecuţi în revistă algoritmii standard existenţi în literatura de specialitate, pentru rezolvarea problemelor CSP, începând cu algoritmul backtracking. Sunt prezentate versiuni de algoritmi care încearcă să îmbunătăţească backtrackingul prin modul de alegere a variabilelor, algoritmi bazaţi pe stategia backjumping, apoi algoritmii care combină căutarea de tip backtracking cu aplicarea unor consistenţe ale constrângerilor la variabilele neinstanţiate. O altă categorie importantă de algoritmi prezentaţi sunt algoritmii ce învaţă, algoritmi ce păstrează înregistrări adiţionale cu constrângeri în timpul căutării. Acest model va fi modelul de plecare ce va fi extins la modelul distribuit, iar tehnicile centralizate vor fi extinse la cadrul distribuit având ca ţintă rezolvarea unor probleme de căutare prin folosirea puterii mai multor agenţi.

În capitolul 3 este descris cadrul de lucru al problemelor de satisfacerea constrângerilor distribuite şi se prezintă cele mai importante tehnici de rezolvare a problemelor DCSP. Acest cadru va fi suportul de studiu al tezei de faţă. Pentru fiecare din aceste tehnici se prezintă principalele rutine de tratare a mesajelor împreună cu mecanismul de funcţionare. În acest capitol sunt prezentate cele mai importante tehnici de căutare asincrone, punându-se accentul pe acei algoritmi pentru care se vor propune îmbunătăţiri: tehnica „asynchronous backtracking”, tehnica „asynchrounous weak-commitment search”, familia ABT, tehnica „asynchronous search with aggregations” şi tehnica „distributed dynamic backtracking”.

Cu toate că domeniul programării bazate pe constrângeri a beneficiat de o atenţie deosebită din partea comunităţii academice şi industriale, domeniul programării bazate pe constrângeri distribuite la agenţi este mai puţin cunoscut şi dezvoltat, existând o serie de probleme nerezolvate, câteva fiind amintite în paragrafele anterioare. Astfel în capitolul 4 sunt prezentate o serie de probleme actuale din domeniul programării bazate pe constrângeri distribuite. În acest capitol sunt analizate condiţiile actuale în care se face implementarea şi evaluarea diverselor tehnici de căutare asincrone. Sunt identificare diverse defecte şi dezavantaje ale tehnicilor de căutare asincrone, plecând de la situaţiile reale ce pot apărea în practică. Sunt analizate redundanţa mesajelor, explozia valorilor de tip nogood, învechirea informaţiilor stocate de fiecare, respectiv modul de utilizare a informaţiilor stocate în mesajele de tip nogood în cazul tehnicilor de căutare asincrone. Preocupările din cadrul cercetărilor de doctorat, prezentate în acest material, se axează pe studierea şi soluţionarea acestor probleme.

Secţiunea a II-a prezintă contribuţii aduse în domeniul dezvoltării sistemelor multi-agent bazate pe tehnicile de căutare asincrone. Secţiunea prezintă un model de implementare şi evaluare pentru tehnicile de căutare asincrone. Folosirea acestui model permite construirea a două sisteme multi-agent pentru implementarea tehnicilor de căutare asincrone.Cele două sisteme multi-agent sunt utilizate ca suport în studiul comportamentului agenţilor şi identificarea unor îmbunătăţiri în secţiunile următoare.

În capitolul 5 este analizat mediul NetLogo cu scopul construirii unui model general de implementare şi evaluare pentru tehnicile asincrone astfel încât să se poată utiliza mediul NetLogo ca un simulator de bază în studiul tehnicilor de căutare asincrone. În acest capitol sunt propuse două metode de detecţie a terminării algoritmilor, împreună cu arhitecturile celor două sisteme multi-agent obţinute. În final, este prezentată o metodologie de implementare a tehnicilor de căutare asincrone, pe baza modelului propus.

În capitolul 6 sunt analizate cele mai importante tehnici de căutare asincrone, pe baza unui studiu experimental ce utilizează modelele de implementare şi evaluare prezentate anterior. Scopul principal al acestui studiu este de a vedea comportamentul agenţilor în cele două situaţii de implementare: cu sincronizarea execuţiei agenţilor şi execuţie asincronă. Sunt propuse mai multe soluţii de îmbunătăţire a performanţelor pe baza studiului comparativ, cum ar fi sincronizarea execuţiei agenţilor în cazul anumitor tehnici, introducerea managementului mesajelor, filtrarea mesajelor.

Secţiunea a III-a conţine contribuţii aduse în domeniul îmbunătăţirii performanţelor tehnicilor de căutare asincrone. Secţiunea prezintă mai multe soluţii legate de eliminarea efectului exploziei valorilor nogood, limitarea fluxului de mesaje pentru legăturile temporare şi utilizarea eficientă a listelor nogood.

În capitolul 7 este propusă o soluţie originală de aplicare a tehnicii cu flaguri în cazul familiei ABT. Este înlocuită stocarea valorilor nogood cu etichetarea valorilor din domeniu prin două tipuri de flaguri. Spaţiul de memorie necesar se reduce considerabil în cazul aplicării tehnicii cu flaguri. Această soluţie este aplicată pentru nucleul ABT obţinându-se o variantă originală numită *ABT kernel cu flaguri*. Plecând de la acest nucleu se poate ajunge la două tehnici cunoscute: Asynchronous Backtracking şi Distributed Dynamic Backtracking (Bessiere) în variantele cu flaguri. În felul acesta se obţine o generalizare a tehnicii cu flaguri pentru cazul familiei ABT, soluţie originală propusă în acest capitol. În acest capito, se investighează eficienţa noilor tehnici din familia ABT, în raport cu fluxul de mesaje schimbate de agenţi,

numărul de cicli consumaţi pentru găsirea soluţiei şi constrângerile verificate, comparativ cu tehnicile de bază derivate din nucleul ABT.

În capitolul 8 este investigat comportamentul tehnicii ABT_{temp} pentru diferite valori ale numărului de mesaje transmise pentru legăturile temporare Pe baza acestei analize, în acest capitol sunt identificate şi prezentate mai multe soluţii originale de determinare a numărului de mesaje schimbate de agenţii cu legături temporare, număr optim de mesaje pentru care trebuie păstrată o legătură temporară. Este propusă o soluţie dinamică de determinare a numărului de mesaje necesare pentru păstrarea unei legături, experimentele arătând o reducere a costurilor relativ la varianta de bază ABT.

În capitolul 9 se propune un nou membru hibrid pentru familia ABT, membru derivat din nucleul ABT prin eliminarea informaţiilor învechite dintre agenţi, combinând două metode de eliminare a informaţiilor învechite dintre agenţi: adăugarea de legături permanente între agenţi, respectiv adăugarea de legături temporare. Noul membru se obţine prin identificarea şi transformarea anumitor legături temporare în legături permanente. Pentru aceasta se propune o metodă de identificare a legăturilor temporare bazată pe informaţiile relative la fluxul de mesaje învechite recepţionate de fiecare agent. Capitolul 9 continuă studiul legăturilor temporare cu scopul de a găsi soluţii de îmbunătăţire a performanţelor tehnicilor de căutare asincrone, studiul având ca punct de plecare soluţiile dinamice propuse în capitolul 8.

În capitolele 10 şi 11 sunt propuse soluţii de adaptare a tehnicii „nogood processor" pentru tehnica AWCS şi de combinare a acestei tehnici cu tehnicile de construire a unor nogooduri eficiente, obţinându-se o variantă hibridă cu costuri mai mici în obţinerea soluţiei.

În capitolul 10 se propune o soluţie de adaptare a tehnicii „nogood processor" introdusă de Armstrong, pentru tehnica AWCS [AD97]. Această tehnică constă în stocarea valorilor nogood şi folosirea ulterioară a informaţiilor din nogood-uri în procesul de selectare a unei noi valori pentru variabilele asociate agenţilor. Sunt analizate situaţiile în care se foloseşte un „nogood processor" centralizat şi situaţiile în care se distribuie nogood-urile la mai multe „nogood processor" ce sunt mânuite de anumiţi agenţi. Se propune o soluţie de distribuire a procesoarelor de nogood-uri către agenţi, în funcţie de ordinea agenţilor, cu scopul de a reduce costurile de căutare şi de stocare.

Plecând de la rezultatele din capitolul 10, în capitolul 11 se încearcă combinarea variantelor de nogood processor cu tehnicile de învăţare, cu scopul de a găsi o soluţie de îmbunătăţire a performanţelor tehnicii AWCS. Plecând de la aceste tehnici, sunt propuse în acest capitol anumite modificări pentru cele două tehnici cunoscute, obţinând două tehnici derivate. Scopul combinării celor două tehnici (tehnici ce se concentrează pe minimizarea şi selecţia valorilor nogood.) a fost de îmbunătăţire a eficienţei algoritmului AWCS.

Secţiunea a IV-a prezintă concluziile desprinse pe parcursul tratării subiectelor propuse în lucrare, rezumatul contribuţiilor, precum şi perspectivele de continuare a cercetării. În capitolul 12 se face o trecere în revistă a obiectivele propuse în această teză O prezentare detaliată a modului în care au fost tratate şi atinse aceste obiective se regăseşte în acest capitol. De asemenea, un rezumat al contribuţiilor este prezentat în capitolul 12, menţionându-se şi publicaţiile în care au fost prezentate rezultatele.

Întreaga activitate de cercetare-dezvoltare din cadrul programului de doctorat a fost realizată în laboratoarele de calculatoare ale Facultăţii de Inginerie Hunedoara şi în cele existente în cadrul Departamentului de Calculatoare al

Universităţii „Politehnica” din Timişoara, laboratoare coordonate de autor şi de către prof. dr. ing. Vladimir CREŢU. Diverse elemente şi rezultate au fost obţinute în colaborare cu laboratorul de sisteme multi-agent existent în cadrul Departamentului de Ştiinţa şi Ingineria Calculatoarelor, Universitatea din Carolina de Sud (SUA), coordonat de prof. Jose VIDAL ([MASNet-a]).

2. PROBLEME DE SATISFACEREA CONSTRÂNGERILOR. MODELUL CSP

Problemele de satisfacere a constrângerilor reprezintă un cadru nou, general, de modelare şi formalizare pentru foarte multe aplicaţii. Reţelele cu constrângeri şi-au dovedit succesul în modelarea problemelor reale. Multe probleme reale, cum ar fi cele de planificare, configurare, diagnoză, teoria grafurilor, proiectarea circuitelor, biologie moleculară, design-ul maşinilor, înţelegerea limbajelor, inginerie electrică (localizarea erorilor), etc. pot fi văzute ca şi categorii de probleme de satisfacere a constrângerilor.

În acest capitol este descris modelul CSP şi sunt trecuţi în revistă algoritmii standard existenţi în literatura de specialitate, pentru rezolvarea problemelor CSP.

În partea de început a capitolului se definesc formal problemele de satisfacere a constrângerilor, prezentându-se câteva exemple de probleme reale modelate cu constrângeri. Apoi, se prezintă primul algoritm de rezolvare CSP bazat pe căutare, algoritm cunoscut sub numele de backtracking. Cea mai mare parte a capitolului se concentrează pe prezentarea unor algoritmi care caută să elimine defectele backtrackingului. Sunt prezentate versiuni de algoritmi care îmbunătăţesc backtrackingul prin modul de alegere a variabilelor, algoritmi bazaţi pe stategia backjumping, apoi algoritmii care combină căutarea de tip backtracking cu aplicarea unor consistenţe ale constrângerilor la variabilele neinstanţiate. Sunt prezentate câteva euristici existente în literatura de specialitate, euristici ce încearcă îmbunătăţirea backtrackingului prin modificarea statică sau dinamică a ordinii variabilelor. O altă categorie de algoritmi bazaţi pe backtracking sunt algoritmii ce învaţă prin păstrarea unor înregistrări adiţionale cu constrângeri în timpul căutării.

Acest model va fi modelul de plecare ce va fi extins la modelul distribuit, iar tehnicile centralizate vor fi extinse la cadrul distribuit având ca ţintă rezolvarea unor probleme de căutare prin folosirea mai multor agenţi.

2.1. Definirea modelului Problemelor de Satisfacere a Constrângerilor.

În acest paragraf sunt definite noţiunile de bază necesare construirii modelului CSP. Acest model se bazează pe mai multe noţiuni de valoare, domeniu, asociere, constrângere şi consistenţă [DF99], [Tsa93].

2.1.1. Definirea problemelor de satisfacere a constrângerilor.

Definiţia 2.1.1 (Valoare). O *valoare* este un element ce poate fi atribuit unei variabile. Dacă variabila este notată cu x_i atunci notaţia $x_i=v_i$ semnifică atribuirea valorii v_i la variabila x_i.

În funcţie de tipul acestor valori ce pot fi atribuite variabilelor se obţin diverse tipuri de probleme. Valorile pot fi numerice (întregi, raţionale, reale, etc.) sau booleene (true sau false). În general, pentru problemele de satisfacere a

constrângerilor se consideră valoarea v_i ce poate fi atribuită variabilelor x_i de tip întreg.

Definiţia 2.1.2 (Domeniul unei variabile). *Domeniul unei variabile* este o mulţime de valori pe care le poate lua acea variabilă. Dacă variabila este x_i, se va nota cu D_i domeniul acestei variabile.

Definiţia 2.1.3 (Asociere). Se numeşte *asociere* pentru variabila x_i o pereche (x_i, v), unde v este o valoare din domeniul lui x_i. Dacă variabilei x_i i s-a atribuit o valoare v_i din domeniul său se spune că este *instanţiată*, în caz contrar *neinstanţiată*.

Variabilele cu care se operează sunt date de obicei într-o anumită ordine. Prin urmare, noţiunea de instanţiere poate fi extinsă la un grup ordonat de varibile.

Definiţia 2.1.4 (Instanţiere parţială). Se numeşte *instanţiere parţială* pentru un grup de variabile ordonate $(x_1, x_2, ..., x_k)$, notată $\overrightarrow{a_k}$ o mulţime de asocieri realizată în ordinea dată: $((x_1,a_1),(x_2,a_2),...,(x_k,a_k))$.

Definiţia 2.1.5 (Constrângere). O *constrângere* pentru o mulţime de variabile este un ansamblu de asocieri ce validează acele variabile. O constrângere va fi notată prin C_X, unde X este mulţimea de variabile validate.

Pentru domenii finite de valori, o constrângere poate fi reprezentată ca o submulţime a produsului cartezian al domeniilor variabilelor implicate.

În practică, o constrângere este o simplă relaţie logică între mai multe variabile ce pot lua valori dintr-un domeniu dat. O constrângere limitează valorile posibile pe care le pot lua variabilele. Constrângerile pot fi reprezentate în diverse moduri: matrici, funcţii, inegalităţi, mulţimi de valori interzise. De exemplu, dacă se consideră trei variabile x_1, x_2, x_3 fiecare luând valori din intervalul [1,10], o constrângere poate fi o relaţie algebrică $x_1+x_2+x_3 >17$, dar se poate reprezenta şi prin mulţimea de valori care sunt permise.

O constrângere ce implică k-variabile se mai numeşte de aritate k. Dacă constrângerea implică una sau maxim două varibile ea se numeşte constrângere binară. O constrângere binară va fi notată cu C_{ij}, unde cele două variabile implicate sunt x_i şi x_j. Orice constrângere de aritate k poate fi transformată într-o constrângere binară. În această lucrare se va considera cazul constrâgerilor binare.

O constrângere se precizează prin submulţimea de variabile şi combinaţiile de valori asociate care sunt permise. De fapt, o constrângere poate fi definită ca un predicat, notat $p_i(x_{i1},...,x_{ij})$, definit pe produsul cartezian $D_{i1} \times D_{i2} \times ... \times D_{ij}$. Predicatul este adevărat dacă toate asocierile de valori la aceste variabile satisfac constrângerea.

Definiţia 2.1.6 (Instanţiere parţială consistentă). O instanţă parţială a unui grup ordonat de variabile $\overrightarrow{a_k}$ este *consistentă* dacă toate constrângerile ce implică variabilele din instanţă sunt îndeplinite pentru asocierile din aceea instanţă.

Definiţia 2.1.7 (Variabilă eşec). O variabilă x_i se numeşte *variabilă eşec* dacă nu există nici o valoare în domeniul său care să fie consistentă cu instanţa

parţială consistentă $\overrightarrow{a_{i-1}}$. Cu alte cuvinte, variabila x_i nu are nici o valoare în domeniul său a.i. să fie consistentă cu variabilele anterioare x_k, i=1,i-1.

Definiţia 2.1.8 (Modelul CSP). Modelul bazat pe constrângeri Constraint Satisfaction Problems-CSP, existent pentru arhitecturi centralizate este un triplet ***P={X, D, C}*** , unde:

- ***X*** este un ansamblu finit de ***n*** variabile $\mathbf{x_1, x_2,..., x_n}$, ce pot lua valori finite, din mai multe domenii discrete finite $\mathbf{D_1, D_2,..., D_n}$ ce formează mulţimea ***D***.
- ***C*** este o mulţime de constrângeri între aceste variabile.

Definiţia 2.1.9 (Rezolvarea unei probleme CSP).
Rezolvarea unei probleme de satisfacerea constrângerilor ***P={X, D, C}*** constă în construirea unei asocieri de valori din domeniul ***D*** pentru toate variabilele din ***X*** astfel încât toate constrângerile din ***C*** să fie îndeplinite.

Remarca 2.1.1. Îndeplinirea constrângerilor este, în general, o problemă NP-completă.

O problemă care are o soluţie se mai numeşte *rezolvabilă* sau *consistentă*, în caz contrar i se spune *nerezolvabilă* sau *inconsistentă*. În anumite probleme se doreşte găsirea tuturor soluţiilor, în alte cazuri este suficientă găsirea unei singure soluţii sau un răspuns legat de existenţa acesteia.

Exemplu. Se consideră mulţimile ***P={X, D, C}***:

- ***X*** = $\{x_1, x_2, x_3\}$.
- ***D*** = $\{ D_1, D_2, D_3\}$ cu $D_1=D_2=D_3=\{1, 2, 3\}$.
- ***C*** = $\{ C_{123}=\{(v_1, v_2, v_3) \in D_1 \times D_2 \times D_3$, unde $v_1 + v_2 = v_3\}$, $C_{12}=\{(v_1, v_2) \in D_1 \times D_2$, unde $v_1 \neq v_2\}\}$.

Se observă că această problemă are două soluţii $\{x_1=1, x_2=2, x_3=3\}$ şi $\{x_1=2, x_2=1, x_3=3\}$.

Definiţia 2.1.10 (Constrângere de aritate k). O constrângere este de aritate k (constrângere k-ara) dacă ea implică cel mult k variabile. O constrângere se numeşte *unară* dacă se referă la o singură variabilă. În schimb, o constrângere *binară* corespunde la două variabile.

Definiţia 2.1.11 (Problemă de satisfacere a constrângerilor binare). O problemă CSP se numeşte binară, dacă toate constrângerile sunt unare sau binare.

Definiţia 2.1.12 (Graful constrângerilor). Fiecare problemă de satisfacere a constrângerilor poate fi reprezentată printr-un graf al constrângerilor, în care nodurile sunt reprezentate de variabile iar arcele conectează fiecare pereche de variabile conţinute în constrângerile binare.

Remarca 2.1.1 O problemă de satisfacerea constrângerilor se mai numeşte şi *reţea cu constrângeri*.

2.1.2. Exemple de probleme reale modelate CSP.

În acest paragraf sunt prezentate mai multe exemple clasice de probleme CSP, probleme ce vor putea fi extinse şi pentru modelarea cu constrângeri distribuite.

Exemplul 2.1.1-problema celor n regine.

Problema constă în plasarea a n regine pe tabla de şah astfel încât să nu atace reciproc. Problema poate fi formalizată ca o problemă CSP astfel:

- variabilele corespund celor n regine, variabila x_i va memora coloana de pe linia *i* în care se va aşeza regina *i* (se observă cu fiecare regină trebuie aşezată pe o linie separată, altfel problema nu se poate rezolva).
- domeniul variabilelor este {1,2,…,n}, adică cele n coloane ale tablei de şah.
- constrângerile se referă la faptul că reginele nu trebuie să se atace reciproc, acest lucru putând fi formalizat prin predicate. De exemplu, o constrângere între două variabile x_i şi x_j poate fi reprezentată astfel: $x_i \neq x_j$ (nu sunt pe aceiaşi coloană) $\wedge$ $|x_i - x_j| \neq |i-j|$ (nu sunt pe aceiaşi diagonală).

Găsirea unei soluţii constă în asocierea de valori la fiecare variabilă (regină) astfel încât constrângerile să fie îndeplinite.

În cazul acestei probleme constrângerile sunt binare (sunt definite între două variabile). Scopul oricărui algoritm CSP este de a asocia valori din domeniu (aici din cele n coloane) şi de a verifica constrângerile existente. Graful constrângerilor în cazul problemei celor n regine este un graf complet, oricare 2 regine apar într-o constrângere

Exemplul 2.1.2- problema satisfacerii formulelor propoziţionale (SAT)

Un alt exemplu de problemă clasică de satisfacerea constrângerilor este problema satisfacerii formulelor propoziţionale (SAT). Scopul problemei este de a determina când o formulă booleană este îndeplinită, adică dacă există asocieri de valori pentru care formula să aibă valoarea true. O formulă booleană este compusă din variabile booleene, care pot lua două valori : true sau false, variabile legate prin operatori logici cum ar fi $\wedge$ (şi) $\vee$ (sau) $\neg$ (negaţia) şi () parantezele de grupare. O clauză este o disjuncţie de literale, unde prin literal se înţelege valoarea ataşată unei variabile.

Dându-se o mulţime de clauze C_1, C_2, …,C_m şi variabilele $x_1,x_2,…,x_n$, problema SAT constă determinarea condiţiilor în care formula $C_1 \wedge C_2 \wedge … \wedge C_m$ este îndeplinită. În particular, dacă numărul de literale din fiecare clauză este limitat la 3, problema se mai numeşte 3-SAT . Se observă faptul că aici domeniile variabilelor sunt formate doar din două valori.

Exemplul 2.1.3- problema colorării unui graf.

Un alt exemplu de problemă clasică (nu numai) de satisfacerea constrângerilor este problema colorării unui graf (sau problema colorării hărţilor). Scopul acestei probleme este de a colora fiecare nod cu o culoare astfel încât două noduri vecine (legate printr-un arc) să nu aibă aceeaşi culoare. Fiecare nod poate fi colorat cu un număr finit de culori. Problema se poate descrie cu ajutorul modelului CSP:

- ***X*** = { x_1, x_2, ..., x_n} – cele n variabile asociate celor n noduri.
- ***D*** = { D_1, D_2, ..., D_n} cu $D_1=D_2$ =... =D_n ={1, 2,.., m} cu m numărul de culori utilizate.
- ***C*** = { Cij={(v_i, v_j) $\in$ D_i x D_j, unde dacă a[i,j]=1 atunci $v_i \neq v_j$ (adică dacă nodurile i şi j sunt vecine, culorile lor să fie diferite).

Practic culoarea fiecărui nod va fi stocată într-o variabilă, iar culorile disponibile unui nod vor forma domeniul acelei variabile. Constrângerile se referă la

faptul că două noduri vecine, să nu aibă aceeaşi culoare. De exemplu, dacă se cunoaşte matricea de adiacenţa a grafului, putem scrie aceste constrângeri astfel: dacă a[i,j]=1 atunci x[i]≠x[j] (dacă i şi j sunt vecine, culorile lor să fie diferite).

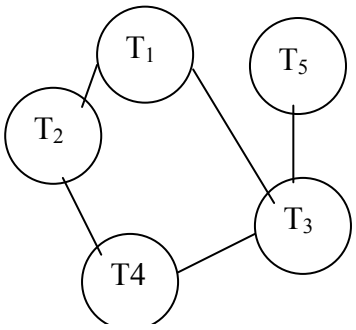
Exemplul 2.1.4- problema planificării lucrărilor.

O altă clasă de probleme, în care se poate folosi pentru formalizare şi rezolvare cadrul CSP, este cea a problemelor de planificare a lucrărilor. Problema poate fi văzută ca o problemă CSP astfel:

- ***X*** = { x_1, x_2, ..., x_n} – cele n variabile asociate celor n lucrări.
- ***D*** = { D_1, D_2, ..., D_n} cu $D_1=D_2$ =... =D_n – intervalele de timp la care pot începe lucrările.
- ***C*** = { Cij={(v_i, v_j) ∈ D_i x D_j}, unde există anumite restricţii între v_i şi v_j.

De exemplu, pentru o problemă de planificare a 5 lucrări, notate cu T_1, T_2, T_3, T_4 şi T_5, fiecare necesitând o oră pentru executarea lor. Lucrările pot începe la 1:00, 2:00 sau 3:00. În orice moment se pot executa mai multe lucrări simultan, planificarea lor fiind totuşi restricţionată de următoarele restricţii: T_1 trebuie să pornească după T_2 şi T_3, dar înainte de T_4 şi după T_5. De asemenea, T_2 nu se poate executa în acelaşi timp cu T_1 sau T_4 şi T_4 nu poate porni la ora 2:00 .

Cu cele cinci lucrări şi cele trei intervale de timp, se poate modela această problemă de planificare prin crearea a 5 variabile, câte una pentru fiecare lucrare. Variabilele pot lua valori din domeniul {1:00, 2:00, 3:00}. Constrângerile sunt cele prezentate mai sus (de asemenea, ele sunt prezentate sub o altă formă în figura 2.1). Pentru această modelare, în figura 2.1 este prezentat graful constrângerilor şi relaţiile dintre constrângeri. În acest exemplu constrângerile sunt binare.



Constrângeri unare: T_4≠2:00
Constrângeri binare:
T_1,T_2 :{(1:00,2:00), (1:00,3:00), (2:00,1:00), (2:00,3:00), (3:00,1:00), (3:00,2:00)}
T_1,T_3 :{(2:00,1:00), (3:00,1:00), (3:00,2:00)}
T_2,T_4 :{(1:00,2:00), (1:00,3:00), (2:00,1:00), (2:00,3:00), (3:00,1:00), (3:00,1:00),(3:00,2:00)}
T_3,T_4 :{(1:00,2:00), (1:00,3:00), (2:00,3:00)}
T_3,T_5 :{(2:00,1:00), (3:00,1:00), (3:00,2:00)}

Figura 2.1. Graful constrângerilor şi relaţiile dintre constrângeri pentru problema planificării lucrărilor

O altă formalizare de tip CSP constă în a crea trei variabile, fiecare corespunzând la cele trei momente de timp, domeniul fiind format din cele cinci lucrări {T_1, T_2, ..., T_5}.

Alte exemple de probleme CSP sunt cele de planificare şi programare ([KS92]), raţionamentul diagnosticelor, problema frecvenţei asocierilor (numită şi problema asocierii canalelor) [Box78] .

Trebuie remarcat faptul că, deşi formalizarea CSP este foarte simplă, această formalizare poate acoperi o varietate mare de probleme. De exemplu, problema etichetării scenelor dintr-o imagine de pe computer a fost prima aplicaţie modelată cu CSP. După ce o imagine este preluată şi stocată în computer folosind o cameră video, liniile pot fi recunoscute şi se pot reface scenele imaginii.

2.2 Metode de rezolvare pentru clasa de probleme CSP.

2.2.1. Introducere

Orice problemă modelată ca o problemă CSP poate fi rezolvată cu ajutorul metodelor generale oferite de algoritmii cu constrângeri, existenţi pentru arhitecturi centralizate. Modelarea CSP (numită şi reţea cu constrângeri) este specifică arhitecturilor centralizate.

Metodele de rezolvare a problemelor cu constrângeri pot fi împărţite în două mari grupe de algoritmi [DF99], [Ku92], [Fro97], [Tsa93]:

- **algoritmi pentru pentru consistenţa constrângerilor**. Aceştia sunt algoritmi de preprocesare ce se aplică înaintea algoritmilor de căutare, în scopul reducerii mai târziu a efortului de căutare.
- **algoritmi de căutare.** Algoritmii de căutare sunt algoritmii propriu-zişi de găsire a soluţiilor prin căutare.

Termenul de **căutare** este utilizat pentru a desemna o categorie largă de algoritmi care rezolvă problemele prin definirea operaţiilor ce vor fi făcute, posibil cu ajutorul unor euristici. O bună ghicire a ordinii de căutare poate duce la o apropriere de soluţie. În schimb, dacă operaţiile alese nu conduc spre soluţie, ele pot fi abandonate şi se pot încerca altele.

Pentru modelarea CSP, căutarea este exemplificată de algoritmii de tip *backtracking* [DF99], [Tsa93], [Ku92]. Backtrackingul are la bază asocierea de valori la variabile astfel încât o soluţie parţială este extinsă prin atribuirea unei noi valori la o nouă variabilă. Dacă nu se găseşte nici o valoare acceptabilă, se revine la asocierea precedentă şi se încearcă o altă asociere (de aici denumirea de backtracking). Acest tip de căutare, în cazul cel mai nefavorabil, necesită un timp exponenţial după numărul de variabile pentru găsirea unei soluţii.

Rezolvarea unei probleme prin deducere se aplică cu scopul de a transforma problema iniţială într-una echivalentă având o formă mai simplă (în sensul rezolvării mai uşoare). Cele mai cunoscute aplicări ale principiului deducerii sunt cunoscute sub numele de *propagarea constrângerilor* sau *consistenţa constrângerilor*. Aceste metode constau în transformarea grafului constrângerilor într-unul echivalent, de obicei prin *deducerea* unor noi constrângeri prin gruparea vechilor constrângeri şi anularea valorilor corespunzătoare din domeniul lor. Sunt întâlnite reţele de constrângeri numite 1-consistente în care valorile din domeniul fiecărei variabile satisfac graful constrângerilor unare (în care avem constrângeri legate de o singură variabilă). O foarte mare varietate de algoritmi au fost dezvoltaţi pentru introducerea consistenţelor locale [MF85], [Coo90], [VHDT92].

Algoritmii bazaţi pe consistenţe locale se aplică în faza de preprocesare a problemei înainte de aplicarea algoritmilor de căutare. În marea majoritate a cazurilor, metodele de căutare sistematică cum ar fi backtracking-ul, vor lucra mai eficient pe reprezentări având la bază constrângeri consistente. Nivele variate privind consistenţa constrângerilor intercalate în procesul de căutare dau naştere unor tehnici primare de consistenţă ce sunt încorporate în limbajele de programare cu constrângeri [JL94].

În plus faţă de căutarea bazată pe backtracking şi propagarea constrângerilor, alte metode de rezolvare includ *căutare locală stochastică* şi algoritmi *structure-driven*. Metodele stochastice caută soluţia executând salturi în spaţiul complet al instanţelor [MJPL90]. Interesul pentru soluţiile stochastice s-a mărit după succesul algoritmilor GSAT [SLM92].

2.2.2 Tehnica Backtracking.

2.2.2.1. Prezentarea metodei backtracking.

Un algoritm simplu pentru rezolvarea problemelor de satisfacere a constrângerilor este backtracking**,** algoritm bazat pe traversarea arborelui de căutare în maniera căutării în adâncime [DF99], [Tsa93], [Ku92].

Tehnica backtracking pleacă de la o mulţime iniţial vidă, de variabile ce vor trebui instanţiate. Tehnica încearcă extinderea mulţimii de variabile cu o nouă variabilă şi o nouă valoare asociată pentru această variabilă. Dacă valoarea asociată este bună , procesul se repetă până când toate variabilele sunt incluse. În caz de insucces, o altă valoare este considerată pentru aceea variabilă (o valoare neîncercată anterior) până se găseşte una acceptabilă. Dacă nu există nici o valoare se revine la variabila anterioară şi se repetă procesul. Revenirea la variabila anterioară poartă denumirea de backtrack. Se observă că algoritmul backtracking extinde o soluţie parţială, ce denotă starea din spaţiul de căutare, prin adăugarea unei noi variabile (şi a valorii respective). Un lucru foarte important este legat de ordinea în care se introduc variabilele în soluţia parţială. Ordinea poate fi fixată de la început sau determinată în timpul rulării.
Backtracking-ul parcurge trei etape [DF99], [Tsa93]:

- etapa *forward* (înainte) în care variabila următoare, conform ordinii existente, este selectată.
- etapa *extinderii* în care soluţia parţială este extinsă prin asocierea unei valori consistente (din cele nealese), dacă există, pentru variabila curentă.
- etapa *backward* (de revenire) în care se revine la variabila anterioară, dacă nici o valoare consistentă nu s-a găsit pentru variabila curentă.

În figura 2.2 este descris algoritmul de bază al backtracking-ului [DF99]. Algoritmul din figura 2.2. returnează cel mult o soluţie, dar se poate adapta uşor încât să returneze toate soluţiile sau soluţiile care verifică anumite criterii (de exemplu soluţiile optime).

```
Procedure Backtracking
   Intrări. : Reţeaua de constrângeri cu variabilele {x1,x2,…,xn}şi domeniile
   {D1,D2,...,Dn}
   Ieşiri: O soluţie formată din valorile asociate, sau un mesaj că reţeaua de
   constrângeri este inconsistentă ( adică nu există soluţii).
   i←1                    {Se iniţializează variabila ce contorizează variabilele }
   D'i←Di                 {Se copiază domeniul}
   while i ≤n
        instanţiază xi ← SelectareValoare  {Etapa extinderii }
        if xi este nul then        {Nici o valoare nu s-a gasit pentru xi }
          i←i-1            {Etapa backtrack}
        else
          i←i+1            {Etapa forward }
          D'i←Di
   end while
   if i=0  then
```

```
        return "inconsistent"
    else
        return  valorile atribuite lui {x1,x2, …, xn}
end procedure

Procedure SelectareValoare
    while  D_i'  nu este vid do
        Selectează un element arbitrar a ∈ D_i'
        D_i' ← D_i' -{a}
        if (xi,a) este consistentă cu celelalte valori din soluţia parţială
          return a
    end while
    return  nul                    {Nu s-a găsit nici o valoare  consistentă}
end procedure
```

Figura. 2.2. Algoritmul Backtracking.

Algoritmul lucrează cu o serie de valori din domeniul D_i. De obicei, nu este necesar să se lucreze direct cu aceste valori ci se lucrează cu o mulţime de întregi mapate peste valorile din domeniu. Valorile sunt alese de o subrutină separată numită *SelectareValoare*. Subrutina *SelectareValoare* a fost separată de rutina principală pentru o mai bună claritate. Ea are ca scop selectarea unei valori pentru variabila x_i şi verificarea consistenţei constrângerilor existente în probleme. Această parte de algoritm se mai numeşte şi *verificarea consistenţelor* şi reprezintă un domeniu de studiu intens de îmbunătăţire a performanţelor algoritmilor CSP. De ea depinde foarte mult care este efortul de calcul necesar pentru rezolvarea problemei. Prin urmare, numărul de consistenţe verificate repezintă o unitate de măsură foarte des folosită pentru evaluarea efortului de calcul.

Costul (măsurat în timpii de lucru ai procesorului) consistenţelor căutate depinde şi de reprezentarea internă a constrângerilor. Dacă constrângerile sunt păstrate sub forma unei liste de tuple compatibile, va fi nevoie de căutarea unei anumite tuple. În astfel de cazuri tehnicile de sortare şi indexare reduc timpul mediu de acces. În schimb, când constrângerile sunt slabe, mai eficient este să fie păstrate cele incompatibile într-o listă. O altă tehnică care permite verificarea constrângerilor este aceea de a reprezenta constrângerile sub forma unui tablou cu valori booleene cu mai multe dimensiuni ce depind de numărul de variabile aflate în constrângere. O altă posibilitate este de a reprezenta o constrângere printr-o subrutină de test. Dacă constrângerile sunt de exemplu egalităţi între valorile variabilelor, metoda poate fi eficientă.

Un caz des întâlnit este cel în care se lucrează cu constrângeri binare. Pentru cazul constrângerilor binare (în care fiecare constrângere este între două variabile) complexitatea subrutinei *SelectareValoare* este de tipul *O(n)*. Dacă se foloseşte o listă pentru păstrarea constrângerilor consistente, căutarea necesită *O(log c)* unde c este numărul de constrângeri. În schimb, dacă se folosesc proceduri pentru păstrarea constrângerilor, complexitatea procedurii *Selectează*Valoare depinde de complexitatea procedurilor care verifică constrângerile.

Activitatea de căutare a algoritmului backtracking poate fi descrisă printr-un arbore de căutare, arbore parcurs în adâncime. În figura 2.3. este prezentat un

exemplu simplu de problemă a colorării unui graf. Pentru această problemă în figura 2.4 este prezentat arborele de căutare parcurs de tehnica backtracking. Exemplul din figura 2.4 prezintă un caz particular al problemei CSP – colorarea grafurilor, pentru 7 noduri.

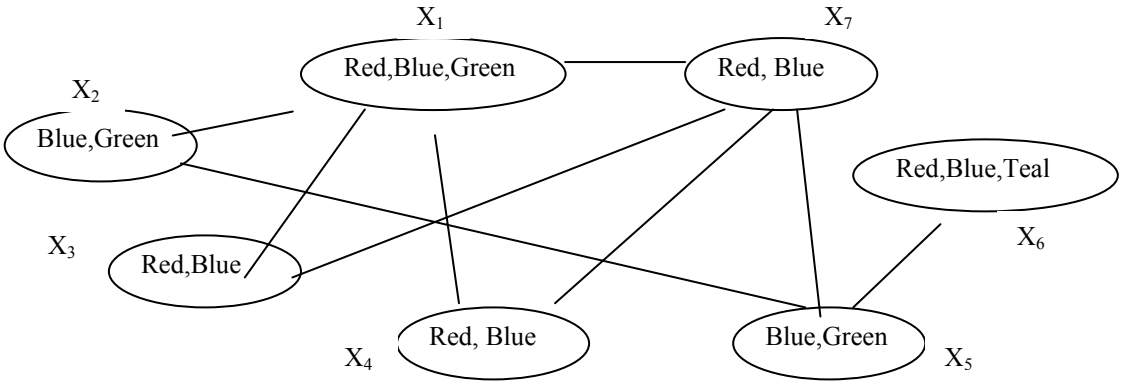


Fig. 2.3. Problema colorării unui graf cu 7 noduri.

Domeniul fiecărei variabile este scris în interiorul elipsei asociate nodului. Se observă că nu toate nodurile au domenii diferite. Pentru această problemă, considerând ordinea X_1, X_2,…, X_n, tehnica backtracking parcurge următorul arbore de căutare, prezentat în figura 2.4 (de fapt, în figura 2.4 apare o parte a acestui arbore) .

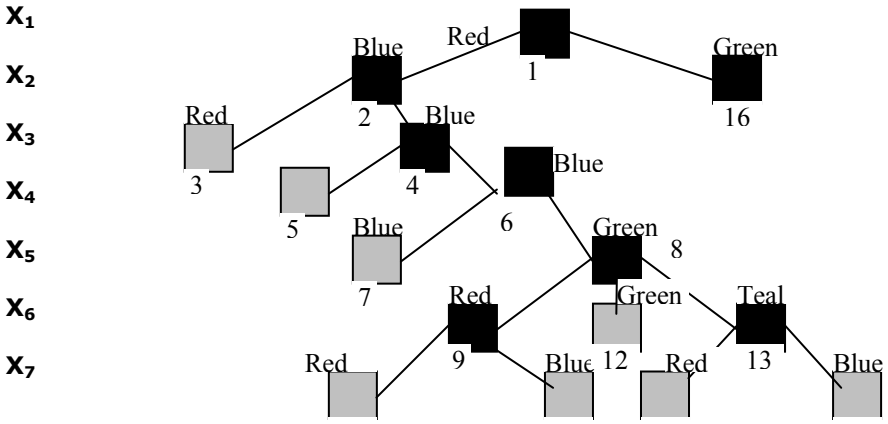


Figura 2.4. Arborele de căutare (parțial) obținut în urma aplicării tehnicii backtracking

Nodurile haşurate cu negru sunt noduri din care căutarea mai poate continua (pentru valorile asociate), în schimb nodurile haşurate cu gri sunt noduri ale căror valori sunt incompatibile cu valorile anterioare. Nodurile au fost numerotate în ordinea construirii lor de către backtracking.

2.2.2.2 Defectele algoritmului backtracking. Îmbunătăţirea performanţelor.

O intensă muncă de cercetare s-a depus în ultima decadă pentru îmbunătăţirea performanţelor căutării prin backtracking. Backtracking-ul obişnuit suferă de ceea ce se cheamă *trashing* adică algoritmul redescoperă în timpul căutării aceleaşi inconsistenţe şi soluţii parţiale, fără a ţine cont de acest lucru.

Performanţele backtracking-ului pot fi îmbunătăţite prin reducerea mărimii spaţiului de căutare expandat (care este determinat de spaţiul de căutare al problemei iniţiale şi de strategia tehnicii de căutare). Mărimea spaţiului de bază al problemei depinde de modul în care sunt reprezentate constrângerile, de ordinea de instanţiere a variabilelor şi, când o soluţie este suficientă, ordinea în care valorile sunt asociate la fiecare variabilă. Ţinându-se cont de aceste trei elemente, cercetătorii au dezvoltat două tipuri de strategii ce ar trebui urmate pentru reducerea spaţiului de căutare: preprocesarea spaţiului de căutare, înainte de a trece la căutarea soluţiei şi reducerea dinamică în timpul căutării a spaţiului de căutarea prin luarea de decizii de nevizitarea a anumitor porţiuni. Cele mai cunoscute sunt tehnicile de preprocesare numite arc şi path –consistency (din primul tip de strategii) [Mac77], [Nad89] şi tehnici euristice pentru determinarea ordinii variabilelor [BR75], [DM94], [DP89], [MJPL90], [FQ85], [Pur83].

Tehnicile de îmbunătăţire a backtrackingului prin micşorarea în mod dinamic (în timpul rulării) a spaţiului de căutare se pot grupa şi ele în două mari scheme: *tehnici de căutare înainte* [HE80], [Walt75] (numite scheme *look–ahead*) şi tehnici de căutare înapoi (numite scheme *look-back*) conform cu cele două etape principale ale backtracking-ului. Tehnica de căutare înainte se poate folosi atunci când algoritmul se află la pasul forward şi se doreşte asocierea unei valori la următoarea variabilă. În schimb, schema de căutare înapoi se aplică la pasul de back al backtrackingului [Gas79], [Dec90], [Pros93a], [Pros93b]. Aceste tehnici sunt prezentate mai departe în paragrafele care vor urma. Aceste tehnici sunt aplicate şi pentru cazul distribuit.

2.2.3. Tehnica Backjumping

2.2.3.1. Definirea strategiei backjumping.

Un alt algoritm primar (folosit şi la cadrul distribuit) ce încearcă să îmbunătăţească randamentul metodei backtracking, este backjumping-ul [Gas79], [Dec90], [Pros93b], [DF99]. Spre deosebire de tehnica backtracking care redescoperă aceleaşi eşecuri, algoritmul backjumping încearcă să reducă fenomenul de redescoperire a eşecurilor.

Un eşec apare atunci când pentru o variabilă x_i nu s-a găsit nici o valoare consistentă, algoritmul trebuind să revină la variabila x_{i-1}, conform definitiei 2.1.7, x_i este denumită variabilă eşec. Considerând că există o nouă valoare pentru x_{i-1}, dar că nu există nici o constrângere între x_i şi x_{i-1}, acelaşi eşec va fi reîntâlnit de x_i, încă o dată şi încă o dată, până când toate valorile lui x_i vor fi parcurse. De exemplu, pentru problema din figura 2.4 (exemplu particular al problemei colorării unui graf) se întâlneşte un eşec la x_7 pentru următoarea asociere: x_1=red, x_2=blue, x_3=blue, x_4=blue, x_5=green, x_6=red. În această situaţie, backtracking-ul va reveni la variabila x_6, va atribui o nouă valoare lui x_6 şi va reîntâlni acelaşi eşec pentru x_7. În schimb, algoritmul backjumping va identifica variabila *"vinovată"* de producerea

eşecului, va realiza un salt la aceasta şi va trece la reiniţializarea acesteia (în loc să continue cu instanţierea cronologică a variabilelor).

Identificarea variabilelor vinovate se bazează pe noţiunea de *mulţime conflictuală***.** Starea de eşec la nivelul i indică faptul că instanţierea curentă parţială $\vec{a}_i$ =(a_1,...,a_i) intră în conflict cu x_{i+1} (numită variabilă eşec). Dar şi subtuplul $\overrightarrow{a_{i-1}}$ =(a_1,...,a_{i-1}) poate să intre în conflict cu x_{i+1} şi, prin urmare, algoritmul backtracking revine la x_i pentru a–i schimba aceea valoare fără ca această valoare să rezolve întotdeauna eşecul lui x_{i+1}. În general, orice tuplu $\vec{a}_i$ ce a ajuns la eşec la nivelul *i* poate să conţină mai multe subtuple ce sunt în conflict cu x_{i+1}. Orice asemenea instanţă parţială nu va putea să fie parte a unei soluţii. Strategia backtracking este greşită pentru că revine la subtuplul $\vec{a}_i$ fără a rezolva aceste mulţimi de conflicte. Prin urmare, eşecul lui x_{i+1} va mai apărea.

Definiţia 2.2.1 (Mulţime conflictuală). Fie instanţa consistentă $\vec{a}_i = (a_1,...,a_i)$ şi y o variabilă neinstanţiată. Dacă nu există nicio valoare din domeniul variabilei y care să fie consistentă cu instanţa $\vec{a}_i$ atunci $\vec{a}_i$ se numeşte *mulţime conflictuală*. În plus, dacă $\vec{a}_i$ nu conţine o submulţime care să fie în conflict cu y, atunci $\vec{a}_i$ se numeşte *mulţime conflictuală minimală*.

Definiţia 2.2.2 (Nogood). Orice instanţă parţială ce nu apare într-o soluţie a problemei CSP se numeşte *nogood*.

Se remarcă faptul că orice mulţime conflict este nogood, dar există valori nogood care nu sunt mulţimi conflictuale cu orice variabilă.

Strategia algoritmului backjumping este diferită faţă de strategia backtracking. Stategia precizează că nu trebuie făcut un salt la variabila anterioară, ci la cea mai recentă variabilă x_k pentru care instanţierea $\overrightarrow{a_{k-1}} = (a_1,...,a_{k-1})$ nu conţine nici o mulţime conflictuală cu variabila eşec x_{i+1}.

În funcţie de modul de detectare a *variabilei vinovate*, există mai multe variante de backjumping. În continuare, sunt prezentate 3 variante de backjumping. Primul algoritm, numit *Gaschnig's backjumping* se bazează pe ideea de revenire la variabila vinovată de producerea eşecului. Al doilea algoritm, numit *graph-based backjumping* (backjumping bazat pe graful constrângerilor), extrage informaţii din graful constrângerilor, informaţii despre punctele de revenire nerelevante. Al treilea algoritm, numit *conflict-directed backjumping* combină cele două idei anterioare.

2.2.3.2 Tehnica Gaschnig's backjumping

Primul algoritm ce încearcă reducerea *thrashing-ului* a fost propus de Gaschnig în [Gas79]. Acest algoritm este capabil să revină de la variabila eşec la cea

mai apropiată variabilă care, instanţiată, este cauza directă a producerii eşecului. O astfel de variabilă este identificată atunci când variabila împreună cu zero sau mai multe variabile care o preced (conform ordinii fixate), este instanţiată în asemenea mod că constrângerile resping orice valoare a variabilei eşec.

Algoritmul *Gaschnig's backjumping* este prezentat în figura 2.5 [Gas79], [DF99].

```
Procedure GashnigBackjumping
   Intrări. : O reţea de constrângeri cu variabilele {x1,x2,…,xn} şi domeniile
   {D1,D2,...,Dn}
   Ieşiri: O soluţie formată din valorile asociate, sau un mesaj că reţeaua de
   constrângeri este inconsistentă ( adică nu există soluţii).
   i←1                    {Se iniţializează variabila ce contorizează variabilele }
   Highi←0                {Se iniţializează pointerul la variabila vinovată de eşec }
   D'i ←Di                {Se copiază domeniul}
   while i ≤n
        instanţiază xi ← SelectareValoareGBJ  {Etapa extinderii }
        if xi este nul then        {Nici o valoare nu s-a gasit pentru xi }
           i←Highi                 {Se face backjump}
        else
           i←i+1                   {Etapa forward }
           D'i ←Di
           Highi←0
   end while
   if i=0  then
        return "inconsistent"
   else
        return  valorile atribuite lui {x1,x2, …, xn}
end procedure

Procedure SelectareValoareGBJ
   while  D'i  nu este gol do
        Selectează un element arbitrar a ∈ D'i
        D'i ← D'i  -{a}
        consistent ←true
        k ← 1
        while  k<i  and  consistent
          if  k > Highi
             Highi← k
             if  a'k  este în conflict cu (xi, a)
                   consistent ← false
          else
             k ← k+1
        end while
        if consistent
             return a
```

```
    end while
    return  nul                    {Nu s-a găsit nici o valoare  consistentă}
end procedure
```

Fig. 2.5. Algoritmul Gashnig's Backjumping

Remarca 2.2.1. În cazul tehnicii backjumping etapa de backtrack existentă la tehnicile backtracking este înlocuită cu etapa de backjumping.

Pentru localizarea variabilei care a produs eşecul, algoritmul foloseşte un tablou, numit $High_i$, i=1,n. În acest tablou se memorează, în timpul generării lui $\vec{a_i}$ (adică în etapa extinderii), anumite informaţii folosite apoi la identificarea variabilei vinovate de eşec. De fapt, $High_i$ va reţine ultima variabilă conform ordinii, care a fost testată pentru consistenţa cu x_i. De obicei, dacă x_i nu este variabilă eşec, atunci $High_i$ va conţine i-1.

De exemplu, pentru problema din figura. 2.3 (problema colorării unui graf) se întâlneşte un eşec la x_7 pentru următoarea asociere: x_1=red, x_2=blue, x_3=blue, x_4=blue, x_5=green, x_6=red. În această situaţie, $High_7$ este 3 deoarece x_7=red a fost exclus de x_1=red, valoarea blue pentru x_7 este exclusă de x_3=blue şi nici o altă variabilă nu a mai fost examinată. În schimb, alte valori pentru x_6 nu sunt necesare a mai fi explorate. În acestă situaţie algoritmul revine la variabila x_3, unde nu mai are nici o valoare de încercat ($D_3^{'}=\varnothing$). Cum $High_3$=2, următoarea variabilă examinată va fi x_2.

Acest comportament permite algoritmul Gashnig's backjumping să nu mai exploreze o parte a arborelui de căutare (parcurs de tehnica backtracking). În felul acesta se reduc costurile necesare obţinerii soluţiei.

2.2.3.3. Tehnica Graph–based backjumping

În algoritmul Gashnig's backjumping un salt era făcut în momentul întâlnirii unui eşec. Acest algoritm este capabil să revină de la variabila eşec la cea mai apropiată variabilă care, instanţiată, este cauza directă a producerii eşecului. O astfel de variabilă era o frunză în arborele de căutare. Totuşi, un eşec poate să apară şi în interiorul unui nod al arborelui de căutare. Dacă toţi copiii unui nod interior în arborele de căutare conduc la un eşec, atunci se spune că nodul respectiv este un eşec interior. Când apare un eşec interior, cel puţin o valoare a variabilelor este compatibilă cu variabilele anterioare, dar aceste valori compatibile nu conduc la o soluţie.

Graph-based backjumping este la bază un backtracking publicată în [Dec90], ce poate sări peste variabilele responsabile de eşec. Spre deosebire de algoritmul Gaschnig's backjumping, backjumping-ul bazat pe graful constrângerilor poate sări peste eşecurile interioare. De asemenea, acest algoritm extrage informaţii despre posibilele mulţimi de conflicte exclusiv din graful constrângerilor.

Când apare un eşec şi o soluţie nu poate fi extinsă la următoarea variabilă x, algoritmul graph-based backjumping sare la cea mai recentă variabilă y conectată la x în graful constrângerilor. Dacă y nu mai are alte valori, algoritmul face un salt înapoi încă odată la următoarea variabilă conectată la x, ş.a.m.d. Pentru a realiza aceste salturi, algoritmul consultă o listă de informaţii preprocesate anterior, listă numită *mulţimea părinţilor* sau *strămoşilor* unei variabile eşec x_i.

Definitia 2.2.3 (strămoşi, părinte). Dându-se graful constrângerilor şi ordinea variabilelor d, *mulţimea strămoşilor* variabilei x_i, notată cu $anc(x_i)$, este mulţimea variabilelor care preced pe x_i conform ordinii d şi sunt conectate cu aceasta. Părintele lui x_i, notat cu $p(x_i)$ este cea mai recentă variabilă din $anc(x_i)$.

Această mulţime va fi notată în algoritmul din figura 2.7 cu $anc(x_i)$ este formată din variabilele care sunt conectate la x_i în graful constrângerilor şi preced variabila x_i (conform ordinii existente).

Definitia 2.2.4 (strămoşi induşi, părinte indus). Dându-se variabila x_i şi submulţimea de varibile Y ce succed pe x_i în ordinea dată, mulţimea de strămoşi induşi a lui x_i relativ la mulţimea Y, notată cu $Ind_i(Y)$, este formată din toate nodurile x_k, cu k<i a.i. există un drum de lungime cel puţin 1 de la x_i la x_k ce trece prin nodurile din Y. În mod similar, se numeşte părinte indus, notat cu $P_i(Y)$, cea mai recentă variabilă din $Ind_i(Y)$.

În figura 2.6 este prezentat graful constrângerilor (a) împreună cu graful indus de mulţimea părinţilor (b). De exemplu, pentru variabila x_6, $anc(x_6) = \{x_2\}$ iar $p(x_6) = x_2$. În schimb, pentru x_7, $anc(x_7) = \{x_1, x_3, x_4, x_5\}$, iar $p(x_7) = x_5$.

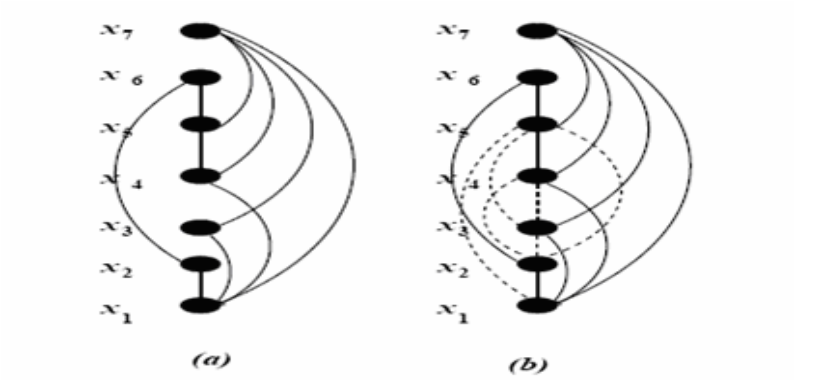


Figura 2.6. Graful constrângerilor (a) şi graful indus de mulţimea părinţilor (b)

Remarca 2.2.2. Graful indus se obţine din graful constrângerilor prin adăugarea unor legături între fiecare nod şi părinţii induşi.

După ce apare un eşec la variabila x_i, algoritmul revine la o anumită variabilă bazându-se pe informaţiile din mulţimea strămoşilor. Pot apărea două situaţii:

- dacă variabila eşec x_i este o frunză, atunci se revine la părintele lui x_i.
- dacă x_i este un eşec interior se construieşte o nouă mulţime prin reunirea mulţimii strămoşilor lui x_i cu toate acele variabile eşec care au fost găsite în arborele de căutare mai jos de x_i. Apoi, se revine la cea mai recentă variabilă din această mulţime, care în algoritm apare sub numele de I_i.

În comparaţie cu algoritmul anterior, această variantă are avantajul de nu avea nevoie să se mai actualizeze acel tablou $High_i$ după fiecare inconsistenţă găsită. În schimb, la acest algoritm este nevoie de o operaţie de preprocesare , necesară pentru a determina mulţimile strămoş $anc(x_i)$.

Algoritmul graph-based backjumping este prezentat în figura 2.7 [Dec90], [DF99].

```
Procedure Graph-Based-Backjumping
   Intrări: O reţea de constrângeri cu variabilele {x1,x2,…,xn} şi domeniile
   {D1,D2,...,Dn}
   Ieşiri: O soluţie formată din valorile asociate, sau un mesaj că reţeaua de
   constrângeri este inconsistentă (adică nu există soluţii).
   Construieşte anc(xi) pentru fiecare xi  {se construieşte lista strămoşilor pt.
   fiecare xi }
   i←1                    {Se iniţializează variabila ce contorizează variabilele }
   D'i←Di                 {Se copiază domeniul}
   Ii←anc(xi)             {Se copiază lista strămoşilor pentru a o putea modifica }
   while i ≤n
        instanţiază xi ← SelectareValoare   {Etapa extinderii }
        if xi este nul then        {Nici o valoare nu s-a gasit pentru xi }
           i-prev←i
           i ← cea mai recentă din Ii          {Se face backjump}
           Ii ← Ii ∪ Ii-prev – {xi}
        else
           i←i+1                                {Etapa forward }
           D'i←Di
           Ii←anc(xi)
   end while
   if i=0  then
        return "inconsistent"
   else
        return  valorile atribuite lui {x1,x2, …, xn}
end procedure

Procedure SelectareValoare          { identică cu cea de la  backtracking  }
…
end procedure
```

Figura 2.7. Algoritmul graph-based Backjumping

Un dezavantaj al folosirii mulţimilor de strămoşi bazate pe graful constrângerilor, comparativ cu strategia de descoperire a conflictelor în momentul căutării consistenţelor, este că se folosesc prea puţine informaţii "rafinate" despre cauza eşecului. Astfel, o variabilă poate fi conectată la o variabilă eşec, dar nu poate fi cauza directă a eşecului. În astfel de cazuri, algoritmul de backjumping bazat pe graful constrângerilor, nu va sări prea departe, şi astfel, puţine eşecuri se vor evita.

2.2.3.4 Tehnica Conflict–directed backjumping.

O altă variantă de backjumping este cea numită *conflict–directed backjumping*.. Tehnica este descrisă în [Pros93b]. Această tehnică combină două idei legate de locul în care se face saltul: revenire la variabila care, instanţiată, este în conflict cu variabila eşec curentă şi revenire la variabilele care produc un eşec interioare. Când o submulţime de variabile sunt instanţiate, ele şi alte constrângeri

care nu sunt relevante pot fi temporar şterse din graful constrângerilor, creându-se un nou graf numit graf condiţional.

Algoritmul de conflict-directed backjumping foloseşte schema schiţată de algoritmul de backjumping bazat pe graful constrângerilor, folosind informaţiile din graf, exploatând informaţiile acumulate în timpul căutării. Pentru fiecare variabilă, algoritmul păstrează o mulţime (notată în algoritm cu J_i) numită mulţime de revenire. Această mulţime înlocuieşte mulţimea strămoşilor de la algoritmul precedent. Spre deosebire de algoritmul anterior, unde mulţimea anc(x) se calcula înainte, la acest nou algoritm, mulţimea de revenire se recalculează de fiecare dată când x_i este instanţiată şi nu mai conţine variabile precedente care instanţiate intrau în conflict cu câteva valori ale lui x_i. În felul acesta se poate reduce substanţial spaţiul de căutare comparativ cu algoritmul anterior, pentru că aceste mulţimi J_i sunt mai mici decât anc(x).

Algoritmul este prezentat în figura 2.8 [Pros93b].

Procedure **Conflict-Directed –Backjumping**
Intrări: O reţea de constrângeri cu variabilele $\{x_1,x_2,\ldots,x_n\}$ şi domeniile $\{D_1,D_2,\ldots,D_n\}$
Ieşiri: O soluţie formată din valorile asociate, sau un mesaj că reţeaua de constrângeri este inconsistentă (adică nu există soluţii).

```
i←1                          {Se iniţializează variabila ce contorizează variabilele }
D'_i ←D_i                    {Se copiază domeniul}
J_i← ∅                       {Se iniţializează mulţimile conflict }
while i ≤n
    instanţiază x_i ← SelectareValoare   {Etapa extinderii }
    if x_i este nul then     {Nici o valoare nu s-a gasit pentru x_i }
        i-prev←i
        i ← cea mai recentă din J_i      {Se face backjump}
        J_i ← J_i ∪ J_i-prev − {x_i}
    else
        i←i+1                            {Etapa forward}
        D'_i ←D_i
        J_i←∅
end while
if i=0 then
    return "inconsistent"
else
    return valorile atribuite lui {x_1,x_2, …, x_n}
end procedure

Procedure SelectareValoare
while D'_i nu este gol do
    Selectează un element arbitrar a ∈ D'_i
    D'_i ← D'_i -{a}
    consistent ←true
    k ← 1
    while k<i and consistent
```

```
            if  $\overrightarrow{a_k}$  este în conflict cu ($x_i$, a)
                adaugă $x_k$ la $J_i$
                consistent ← false
          else
              k ← k+1
        end while
        if consistent
            return a
   end while
   return  nul                    {Nu s-a găsit nici o valoare  consistentă}
end procedure
```

Fig. 2.8. Algoritmul conflict –directed Backjumping

Trebuie amintit faptul că mai există o variantă de backjumping numită *dependency–directed backtracking*, algoritm propus de Stallman şi Sussman în contextul analizei circuitelor [SS77]. Ideea de bază a algoritmului este de a înregistra cauzele eşecului într-o bază de date a constrângerilor, astfel permiţând algoritmului să facă revenirea direct la variabila care era parte a eşecului.

În încheiere, mai trebuie spus, că termenul de backtracking inteligent este de asemenea folosit pentru algoritmii de tip backjumping în programarea logică. De asemenea, aceşti algoritmi de backjumping au avut un rol foarte important în construirea tehnicilor de căutare asincrone cum ar fi *DIBT*.

2.2.4 Strategii bazate pe învăţare.

2.2.4.1. Definirea strategiilor de învăţare bazate pe nogood-uri.

Conform definiţiei 2.2.2 un nogood este o instanţă parţială ce nu poate fi parte a unei soluţii. Totuşi, un nogood oferă foarte multe informaţii ce pot fi folosite în procesul de căutare a soluţiei. Algoritmii bazaţi pe învăţare folosesc informaţiile din nogoodurile ce apar în timpul căutării. Cu alte cuvinte, la un anume pas al algoritmului se poate învăţa din greşeli astfel încât situaţia conflictuală identificată de un nogood să nu se mai repete.

În cazul algoritmului *conflict-directed–Backjumping* se întâlneşte noţiunea de *mulţime minimă de conflict*. Această mulţime este de fapt un nogood ce poate să apară de mai multe ori în timpul căutării (în mod inutil). Strategiile bazate pe învaţare folosesc acest lucru pentru a curăţi spaţiul de căutare. O altă idee este de a introduce noul nogood ca o constrângere suplimentară astfel încât să nu se mai permită selectarea acelor valori. În literatura CSP şi DCSP se foloseşte foarte des termenul de *learning* şi cu sensul de a înregistra potenţiale informaţii ce pot deveni utile în timpul procesului de căutare. Informaţiile pot fi deduse din informaţiile primite la intrare, dar în timpul căutării. De exemplu, apariţia unui eşec în cazul backtrackingului înseamnă că instanţa $\overrightarrow{a_i}$ este în conflict cu variabila x_{i+1}. O astfel de combinaţie poate fi înregistrată pentru mai târziu ca o constrângere astfel încât un astfel de eşec să nu se mai repete. Trebuie remarcat că înregistrarea sub această formă nu are nici o valoare pentru că strategia backtrackingului nu va atinge

această situaţie. Totuşi, când o instanţă $\overrightarrow{a_i}$ conţine una sau mai multe mulţimi în conflict cu x_{i+1}, se înregistrează cea mai mică dintre mulţimile aflate în conflict ca o constrângere. Astfel, în timpul căutării, se va putea exclude o combinaţie mai mare de valori ce conţine şi acel nogood.

Plecând de la modul în care se identifică mulţimea conflict minimală, strategiile de învăţare pot fi în *adâncime* („deep") sau la suprafaţă („shallow"). Prima strategie de tip *deep* înregistrează numai mulţimile conflict minimale. În schimb, a doua strategie *shallow* înregistrează mulţimile conflict neminimale. Aceşti algoritmi pot înregistra un singur nogood pentru un eşec sau mai multe nogooduri şi în felul acesta permit învăţarea la nivel de eşec de tip frunză sau la nivel interior în arborele de căutare. În acest paragraf sunt trecuţi în revistă trei algoritmi de învăţare: învăţarea bazată pe graful asociat problemei (graph-based learning), învăţarea de tip deep (deep learning) şi învăţarea de tip jumpback (jumpback learning).

2.2.4.2. Învăţarea bazată pe graful constrângerilor

Acest algoritm se obţine din algoritmul graph-based backjumping. El utilizează aceeaşi metodă ca şi algoritmul graph-based backjumping în identificarea nogoodurilor. Practic, informaţiile despre conflicte sunt deduse din graful constrângerilor. Pentru o instanţă eşec $\overrightarrow{a_i}$, acele submulţimi de valori asociate cu strămoşii lui x_{i+1} sunt identificate şi incluse în mulţimea conflict.

Algoritmul graph-based backjumping-learning este prezentat în figura 2.9 [Dec90], [DF99].

Procedure **Graph-Based-Backjumping-Learning**
Intrări. : O reţea de constrângeri cu variabilele $\{x_1,x_2,\ldots,x_n\}$ şi domeniile $\{D_1,D_2,\ldots,D_n\}$
Ieşiri: O soluţie formată din valorile asociate, sau un mesaj că reţeaua de constrângeri este inconsistentă (adică nu există soluţii).
Construieşte **anc(x_i)** pentru fiecare x_i {se construieşte lista strămoşilor pt. fiecare x_i }
i←1 {Se iniţializează variabila ce contorizează variabilele }
$D_i^{'}$←D_i {Se copiază domeniul}
I_i←anc(x_i) {Se copiază lista strămoşilor pentru a o putea modifica }
while i ≤n
instanţiază x_i ← SelectareValoare *{Etapa extinderii }*
if x_i este nul then {Nici o valoare nu s-a gasit pentru x_i }
record $\overrightarrow{a_i}$ *ca un nogood* {se înregistrează eşecul }
i-prev←i
i ← cea mai recentă din I_i *{Se face backjump}*
I_i ← $I_i \cup I_{i\text{-prev}} - \{x_i\}$
else
i←i+1 *{Etapa forward }*
$D_i^{'}$←D_i

```
            Ii←anc(xi)
    end while
    if i=0  then
          return "inconsistent"
    else
          return  valorile atribuite lui {x1,x2, …, xn}
end procedure

Procedure SelectareValoare            { identică cu cea de la  backtracking  }
…
end procedure
```

Figura 2.9. Algoritmul graph-based Backjumping-learning

2.2.4.3 Strategii de învăţare în adâncime.

Identificarea şi înregistrarea numai a mulţimilor minimale conflictuale poartă denumirea de strategie de învăţare în adâncime. Spre deosebire de strategia de învăţare anterioară, care înregistrează mulţimile conflictuale neminimale, ea descoperă toate mulţimile conflictuale minimale şi în felul acesta stochează toate informaţiile despre eşec.

Descoperirea tuturor mulţimilor conflictuale minimale poate fi făcută astfel:

- se recunosc toate mulţimile conflict pentru un element;
- apoi se recunosc mulţimile de conflict pentru două elemente, s.a.m.d.;
- în general, dacă s-au recunoscut toate mulţimile conflictuale de dimensiune 1,...i-1 se construiesc mulţimile conflictuale de dimeniune *i* care nu conţin nici o mulţime conflictuală de dimeniune mai mică.

Pentru a elimina explozia de timp şi spaţiu ce apare în cazul învăţării de tip deep, se poate identifica doar mulţimea conflictuală minimală relativă la prefixul mulţimii conflict. Ceea ce se obţine este o *mulţime de salt* la pasul de backtracking ce apare în cazul algoritmului conflict-directed backjumping.

Algoritmul este prezentat în figura 2.10 [FD94].

```
Procedure Conflict-Directed –Backjumping-Learning
  Intrări: O reţea de constrângeri cu variabilele {x1,x2,…,xn} şi domeniile
  {D1,D2,...,Dn}
  Ieşiri: O soluţie formată din valorile asociate, sau un mesaj că reţeaua de
  constrângeri este inconsistentă (adică nu există soluţii).
  i←1                    {Se iniţializează variabila ce contorizează variabilele }
  D'i←Di                 {Se copiază domeniul}
  Ji← ∅                  {Se  iniţializează mulţimile conflict }
  while i ≤n
       instanţiază xi ← SelectareValoare   {Etapa extinderii }
       if xi este nul then      {Nici o valoare nu s-a gasit pentru xi }
          record Ji şi valorile corespunzătoare ca un nogood
          i-prev←i
          i ← cea mai recentă din Ji          {Se face backjump}
          Ji ← Ji ∪ Ji-prev – {xi}
       else
          i←i+1                                {Etapa forward}
```

```
            D'i ←Di
            Ji←∅
    end while
    if i=0  then
        return "inconsistent"
    else
        return  valorile atribuite lui {x1,x2, …, xn}
end procedure

Procedure SelectareValoare {identică cu  conflict–directed Backjumping}
…
end procedure
```

Figura 2.10. Algoritmul conflict–directed Backjumping-Learning

Ca o concluzie, prima metodă de învăţare bazată pe graful constrângerilor înregistrează constrângeri de dimensiune mare (ea este mai performantă), în schimb cele de tip deep pe cele mai scurte. Explozia valorilor nogood va putea fi redusă sau eliminată prin diverse metode ce vor fi prezentate într-un capitol următor.

2.2.5 Strategii look-ahead.

Algoritmii de căutare specifici clasei CSP pot combina backtraking-ul şi propagarea locală a constrângerilor, prin aplicarea unor proceduri de consistenţă a constrângerilor la variabilele neinstanţiate. Această combinaţie este cunoscută sub numele de *"look-ahead"*. Extinderea unei instanţe parţiale poate influenţa constrângerile variabilelor rămase şi aplicând aceste constrângeri consistente se poate reduce dimensiunea spaţiului de căutare.

Pentru a putea exemplifica strategiile "look-ahead", se reconsideră exemplul iniţial prezentat în figura 2.3, exemplu relativ la problema colorării grafului, unde iniţial x_1 are asociată valoarea red. Conform acestei strategii *look-ahead*, valoarea red în domeniile variabilelor x_3, x_4 şi x_7 (care urmează a fi instanţiate) este incompatibilă cu instanţierea parţială (aici formată doar din x_1) şi ar trebui ştearsă provizoriu. Mai mult, analizând graful constrângerilor se poate observa că x_3 şi x_7 sunt conectate în graful constrângerilor şi valorile rămase în domeniu sunt incompatibile, fiecare variabilă având în domeniu doar valoarea {blue}şi, astfel, problema cu x_1=red nu are nici un arc consistent. O astfel de instanţiere a lui x_1 cu red, va conduce în mod inevitabil la un eşec şi, prin urmare, această asociere trebuie eliminată.

În mod sigur, strategiile look–ahead implică un cost suplimentar după fiecare instanţiere, totuşi ele oferă câteva beneficii. Ştergând din domeniul viitoarelor variabile toate valorile inconsistente cu soluţia parţială, se elimină necesitatea de a testa valorile variabile curente (în sensul consistenţei) cu variabilele anterioare. Ca un corolar există situaţia în care toate valorile unei variabile neinstanţiate (vezi situaţia variabilei x_7) sunt inconsistente şi strategia *look-ahead* le şterge din domeniul acelei variabile, atunci instanţierea parţială nu poate fi parte a unei soluţii şi, prin urmare, algoritmul poate reveni la o variabilă anterioară. În felul acesta se va reduce spaţiul de căutare.

În general, folosirea unor nivele puternice de propagare a constrângerilor (adică constrângeri puternice) combinată cu strategia look–ahead, are ca efecte

explorarea unui spaţiu de căutare mult mai mic dar şi costuri suplimentare legate de strategia look-ahead.

2.2.5.1 Algoritmii bazaţi pe strategiile look-ahead

Strategiile look-ahead ce folosesc proceduri de consistenţă pentru constrângeri necesită, în cazul cel mai nefavorabil, acelaşi timp exponenţial ca şi backtracking-ul. Dacă procedurile sunt bazate pe consistenţa arcelor sau pe o formă de slăbire a consistenţelor, atunci spaţiul necesar nu este mai mare ca şi mulţimea principală D'. Se vor prezenta mai departe cele mai importante strategii *look-ahead.* Aceste strategii folosesc o subrutină pentru obţinerea unor constrângeri consistente între două variabile, subrutină prezentată în figura 2.11 şi publicată pentru prima dată în [Mac77].

Procedure **Revise(m,n)**
Intrări.: O reţea de constrângeri cu variabilele $\{x_1,x_2,\ldots,x_n\}$ şi domeniile $\{D_1,D_2,\ldots,D_n\}$ şi o instanţă parţială $\overrightarrow{a_i}$ şi indicii celor două variabile m şi n
Ieşiri: Domeniul D'_m modificat pentru fiecare valoare *a* din D'_m
if nu există nici o valoare $b \in D'_n$ a.î instanţierea ($\overrightarrow{a_i}$,x_m=a, x_n=b) este consistentă
Şterge *a* din D'_m
endif
end procedure

Figura 2.11. Procedura pentru consistenţa constrângerilor

Cele trei strategii au dat naştere la trei algoritmi, după cum urmează:

- *Forward checking* [HE80]- este o formă limitată de propagarea constrângerilor. Forward checking ţine cont de efectul valorii selectate la variabilele viitoare.
- *Arc-consistency look ahead* – include backtracking-ul bazat pe algoritmii care aplică consistenţa arcelor la variabilele neinstanţiate după fiecare asociere a unei valori la variabila curentă. Este de fapt o combinaţie de strategii *look ahead* şi consistenţa arcelor .
- *Directional arc-consistency look-ahead; Partial looking ahead* [HE80]– reprezintă soluţii bazate pe forward-checking şi arc consistency.

Lista de mai sus nu este completă. Există alte tehnici performante, bazate pe aplicarea consistenţei arcelor şi a strategiilor look ahead, cum ar fi variantele lui Waltz [Walt75]. Cercetările recente au început să pună în balanţă costurile algoritmilor de tip look ahead şi cei care se ocupă de curăţarea spaţiului de căutare, mai ales pentru probleme dificile şi cu dimensiuni mari.

2.2.5.2. Tehnica Forward checking.

Algoritmul *forward checking*, descris în [HE80], este utilizat ca o variantă de backtracking. În contrast însă cu tehnicile backtracking şi backjumping, ce se bazau pe ştergerea valorilor din domeniul variabilei curente prin căutarea unor valori bune

în variabilele anterioare, algoritmul *forward checking* atribuie o valoare variabilei curente şi şterge temporar valorile aflate în conflict din domeniul tuturor variabilelor viitoare (care sunt neinstanţiate). Valorile sunt şterse temporar din domeniul respectiv.

Ca şi backtracking-ul şi backjumping-ul, în algoritm se folosesc mulţimile D' pentru a păstra domeniul efectiv de căutare. Anularea temporară a unor valori din mulţimile D', mai este cunoscută şi sub denumirea de *"filtrare"*. În literatura de specialitate, algoritmii care filtrează domeniul variabilelor neinstanţiate sunt adesea numiţi algoritmi *"look-ahead"* .

Algoritmul forward checking din [HE80] este descris în figura 2.12.

Procedure **Forward-Checking**
Intrări: O reţea de constrângeri cu variabilele $\{x_1,x_2,\ldots,x_n\}$şi domeniile $\{D_1,D_2,\ldots,D_n\}$
Ieşiri: O soluţie formată din valorile asociate, sau un mesaj că reţeaua de constrângeri este inconsistentă (adică nu există soluţii).

$D_i' \leftarrow D_i$ pentru i=1,n {Se copiază domeniul}
i←1 {Se iniţializează variabila ce contorizează variabilele }
while i ≤n
instanţiază $x_i \leftarrow$ SelectareValoareFC *{Etapa extinderii }*
if x_i este nul then {Nici o valoare nu s-a gasit pentru x_i }
resetează fiecare mulţime D' la valorile avute înainte de ultima instanţiere a lui x_i
i←i-1 *{Etapa backtrack}*
else
i←i+1 *{Etapa forward }*
end while
if i=0 then
return "inconsistent"
else
return valorile atribuite lui $\{x_1,x_2,\ldots,x_n\}$
end procedure

Procedure **SelectareValoareFC**
while D_i' nu este gol do
Selectează un element arbitrar a $\in D_i'$
$D_i' \leftarrow D_i'$ -{a}
Empty-Domain ← false
Pentru toţi k, i<k≤n
Revise(k,i)
if D_k' este gol {Adică atribuirea x_i=a conduce la un eşec}
Empty-Domain ← true
if Empty-Domain {Nu se selectează valoarea a}
resetează fiecare mulţime D_k' la valorile avute înainte de a selecta valoarea *a*

```
        else   return a
    end while
    return  nul                          {Nu s-a găsit nici o valoare  consistentă}
end procedure
```

Figura 2.12. Algoritmul forward checking

Algoritmul forward checking, descris în figura 2.12, diferă faţă de backtracking în mai multe moduri. În primul rând, selectarea unei valori pentru extinderea instanţei parţiale presupune căutarea unei valori care să fie compatibilă cu cel puţin o valoare pentru fiecare variabilă din cele viitoare. O altă diferenţă este legată de faptul că nu este nevoie să se compare valoarea curentă cu valorile precedente. Algoritmul forward checking tinde în a face să apară eşecul cât mai repede în timpul căutării, ceea ce este opus ca şi comportare faţă de algoritmii de backjumping prezentaţi anterior.

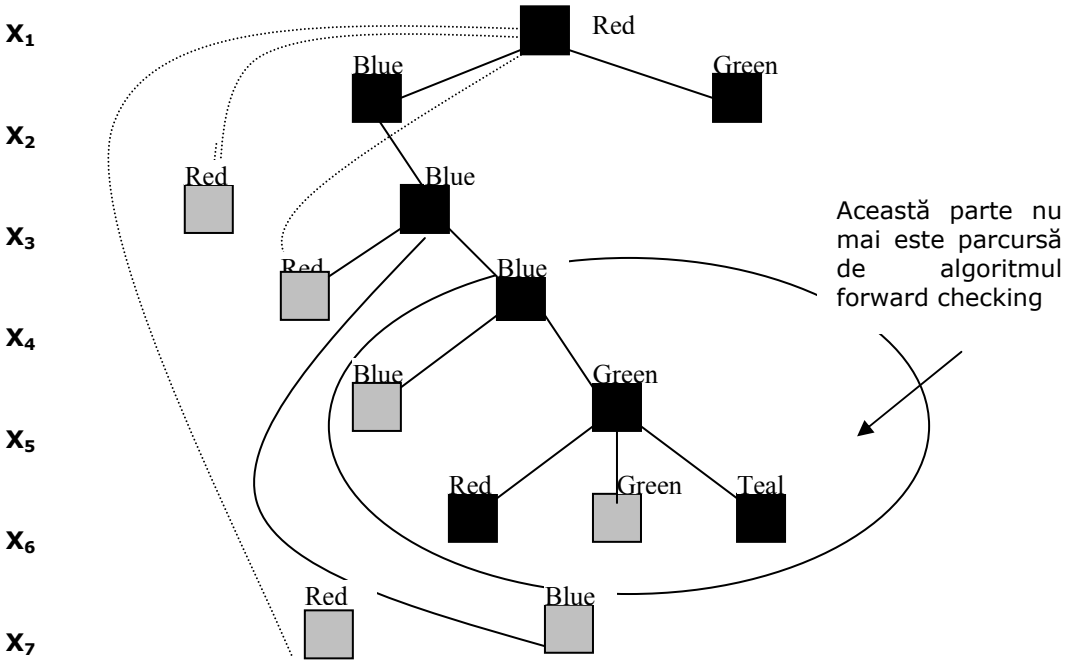


Figura 2.13. Arborele de căutare (parţial) parcurs de algoritmul forward checking

Comportamentul tehnicii forward checking se observă în figura 2.13, pentru exemplul referitor la problema colorării grafului. Se observă că instanţierea lui x_1 cu *red* va reduce domeniile lui x_3, x_4 şi x_7. În schimb, instanţierea lui x_2 cu *blue* nu va afecta nici un domeniu viitor. Variabila x_3 are numai valoarea *blue* în domeniu şi selectarea acelei valori va conduce la golirea domeniului lui x_7, aşa încât x_3=blue este respinsă.

În figura 2.13 se observă că o parte din valori sunt conectate cu linie întreruptă. Acele valorile vor fi filtrate de algoritm (de exemplu x_7=red).

2.2.5.3 Arc–consistency. Combinarea algoritmilor de consistenţa arcelor cu strategiile look ahead.

O altă clasă de algoritmi pentru procesarea problemelor CSP sunt algoritmii care aplică consistenţa arcelor sau propagarea constrângerilor .

Aşa cum s-a prezentat în paragraful §2.1.2, fiecare problemă de satisfacere a constrângerilor poate fi reprezentată printr-un graf al constrângerilor, în care nodurile sunt reprezentate de variabile iar arcele conectează fiecare pereche de variabile conţinute în constrângeri (o astfel de problemă se mai numeşte şi reţea de constrângeri). O astfel de reţea de constrângeri poate avea mai multe grade de consistenţă.

Definiţia 2.2.5 (reţea 1-consistentă). O reţea de constrângeri este *1-consistentă* sau *nod-consistentă* dacă valorile din domeniul fiecărei variabile satisfac reţeaua unară de constrângeri.

Definiţia 2.2.6 (reţea arc-consistentă). O reţea de constrângeri este *2-consistentă* sau *arc-consistentă,* dacă fiecare valoare din domeniul fiecărei variabile este consistentă cu cel puţin o valoare din domeniul oricărei alte variabile.

Definiţia 2.2.5 (reţea k-consistentă). O reţea de constrângeri este *k-consistentă,* $k \geq 2$, dacă dându-se orice instanţă parţială consistentă a oricăror k-1 variabile distincte, ea poate fi extinsă la o altă instanţă consistentă de k variabile, prin adăugarea de noi variabile.

Remarca 2.2.4. O reţea 3-consistentă se mai numeşte şi *path-consistency.* Dacă graful constrângerilor este 2–consistent se spune că problema CSP este *arc-consistentă*. În schimb, dacă reţeaua de constrângeri este k-consistentă pentru toţii k, se spune că problema este consistentă *global.*

Algoritmii *arc-consistency* sunt algoritmi care se execută înainte de începerea căutării. Ideea de bază este aceea de a şterge anumite valori din domeniul anumitor variabile astfel încât fiecare valoare din domeniul fiecărei variabile să fie consistentă cu cel puţin o valoare din domeniul fiecărei alte variabile.

În figura 2.11 a fost prezentată o rutină (numită *Revise*) având ca scop examinarea valorilor a două variabile x_m şi x_n pentru a determina când fiecare valoare y din D'_m are cel puţin o valoare compatibilă în domeniul lui x_n. Dacă nici o valoare nu exista, atunci x_m=y nu putea fi parte a unei soluţii şi, astfel, y era şters din D'_m. Acesta rutină este folosită în construirea algoritmilor de consistenţa arcelor şi în algoritmul final ce combina strategiile de *arc consistent* şi *look ahead*.

Există o lungă istorie a algoritmilor de consistenţa arcelor. Primul algoritm, foarte simplu, a fost prezentat de Mackworth în [Mac77] şi se numeşte *AC-1***.** Pentru a face ca fiecare arc să fie consistent, nu este suficient să se execute procedura *Revise* doar o dată. Aplicând rutina Revise numai o dată, se reduce domeniul unei anumite variabile x_m , iar fiecare din arcele revizuite anterior nu este revizuit încă odată. Acest lucru este necesar deoarece reducând domeniul variabilei x_m este posibil ca un arc, care iniţial era consistent, să nu mai fie consistent. Aceasta este ideea de bază a algoritmului. Algoritmul *AC-1* încearcă obţinerea de arce consistente pentru toate constrângerile grafului. În figura 2.14. este prezentat acest prim algoritm de consistenţa arcelor- *AC-1*.

```
Procedure AC-1
Q ←{(xi,xj) / unde (xi,xj) ∈ arcelor grafului de constrângeri, i≠j}
   repeat
     Change ← false
      pentru fiecare arc (xi,xj) din Q execută
         Revise(i, j)
         if Revise(i, j) a şters valori din domeniul variabilei xi atunci
            Change ← True
   until Not Change
end procedure
```

Figura 2.14. Algoritmul AC-1 pentru consistenţa arcelor

Se observă faptul că dacă procedura Revise a şters o valoare, acest lucru are efect asupra domeniului altor variabile. Prin urmare, procedura Revise este apelată până când nici o valoare nu mai este ştearsă. În felul acesta problema CSP va fi arc-consistent.

Acest algoritm suferă de un anume defect, faptul că dacă o singură valoare din domeniul variabilelor este ştearsă, bucla din repeat este apelată încă odată, fără a fi nevoie pentru anumite variabile. În cazul cel mai nefavorabil, complexitatea algoritmului AC-1 este de tipul $O(d^3\ ne)$ unde d este mărimea domeniului, n este numărul de variabile şi e este numărul de constrângeri.

O prima variantă îmbunătăţită a acestui algoritm a fost numită *AC-3* şi a fost propusă în [Mac77]. Această variantă încearcă să elimine defectul algoritmului AC-1, prin apelarea procedurii *Revise* doar pentru acele arce care ar putea să fi afectate de apelul anterior al lui Revise. Este prezentat în figura 2.15 , algoritmul *AC-3*.

```
Procedure AC-3
Q ←{(xi,xj) / unde (xi,xj) ∈ arcelor grafului de constrângeri, i≠j}
   while Q not empty
        Selectează şi şterge orice arc (xk,xm) din Q
        Revise(k, m)
        if Revise(k, m) a şters valori din domeniul variabilei xk atunci
           Q ← ∪ {(xi,xk) / unde (xi,xk) ∈ arcelor grafului de constrângeri, i≠k i≠m }
        endif
    endwhile
end procedure
```

Figura 2.15. Algoritmul AC-3 pentru consistenţa arcelor

Şi acest algoritm are defectele sale. Când algoritmul AC-3 revizuieşte a doua oară un arc, el retestează multe perechi de valori care erau cunoscute (de la iteraţiile anterioare) a fi consistente şi care nu afectează reducerea domeniului. Acest lucru este o sursă potenţială de ineficienţă a algoritmului *AC-3*, prin urmare un alt lgoritm *AC-4* a introdus rafinarea arcelor.

Algoritmii *AC-3* şi *AC-4* sunt algoritmii cei mai utilizaţi pentru obţinerea consistenţei arcelor. Trebuie precizat că mai există şi alţi algoritmi notaţi cu *AC-5, AC-6, AC-7*, dar ei sunt mai puţin folosiţi în literatura de specialitate.

Combinarea algoritmilor de consistenţă a arcelor cu strategiile *look-ahead* conduc la un nou algoritm de căutare a unei soluţii, algoritm numit *arc-consistency look-ahead*. Acest algoritm este asemănător cu forward checking, dar foloseşte algoritmi de consistenţă a arcelor pentru procesarea spaţiului variabilelor

neinstanţiate. Acest lucru permite o mai bună curăţire a spaţiului variabilelor următoare.

În figura 2.16. este prezentat algoritmul *arc-consistency look- ahead***.**

```
Procedure Arc-Consistency-Look-Ahead
  Intrări : O reţea de constrângeri cu variabilele {x1,x2,…,xn}şi domeniile
  {D1,D2,...,Dn}
  Ieşiri: O soluţie formată din valorile asociate, sau un mesaj că reţeaua de
  constrângeri este inconsistentă ( adică nu există soluţii).
  i←1                    {Se iniţializează variabila ce contorizează variabilele }
  D'i←Di      pentru i=1,n    {Se copiază domeniul}
  while i ≤n
      instanţiază xi ← SelectareValoareAC  {Etapa extinderii }
      if xi este nul then        {Nici o valoare nu s-a gasit pentru xi }
        resetează fiecare mulţime D' la valorile avute înainte de ultima instanţiere a lui xi
        i←i-1          {Etapa backtrack}
      else
        i←i+1          {Etapa forward}
  end while
  if i=0 then
      return "inconsistent"
  else
      return valorile atribuite lui {x1,x2, …, xn}
end procedure

Procedure SelectareValoareAC
  while D'i nu este gol do
      Selectează un element arbitrar a ∈ D'i
      D'i ← D'i -{a}
      Aplică un algoritm de consistenţa arcelor  {Se aplică AC-3 sau AC-4 }
      la toate variabilele neinstanţiate
      if toate domeniile viitoare sunt goale      {Nu se selectează valoarea a }
        resetează fiecare mulţime D'k la valorile avute înainte de a selecta valoarea a
      else
        return a
  end while
  return nul                          {Nu s-a găsit nici o valoare consistentă}
end procedure
```

Figura 2.16. Algoritmul arc-consistency look ahead

Se observă aplicarea unui algoritm de consistenţa arcelor la variabilele viitoare şi identificarea situaţiei în care spaţiul viitor nu are nici un arc consistent.

În [Nad89] sunt propuşi o serie de algoritmi parţiali de consistenţa arcelor, numiţi $AC\frac{1}{2}$, $AC\frac{1}{3}$, $AC\frac{1}{4}$ şi $AC\frac{1}{5}$, algoritmi destinaţi a fi folosiţi în combinaţie cu backtracking-ul. Sufixul fracţional indică aproximativ proporţia în

care lucrează algoritmii faţă de algoritmii anteriori de consistenţa arcelor. De exemplu, forward checking corespunde combinării metodei backtracking cu algoritmul de consistenţă $AC ¼$.

2.2.6. Euristici pentru stabilirea ordinii variabilelor

În algoritmii prezentaţi anterior, s-a considerat că ordinea variabilelor este statică, astfel ea nu putea fi schimbată de rutinele acestor algoritmii. Cercetările experimentale au arătat că ordinea în care sunt instanţiate variabilele poate avea un impact substanţial asupra căutării, în cazul problemelor CSP rezolvate cu metodele de tip backtracking. Câteva euristici au fost propuse şi analizate pentru alegerea ordinii variabilelor.

În [Walt75] se propune o euristică pentru ordinea variabilelor, având ca scop eliminarea a cât mai multor apariţii timpurii ale backtrack-ului.

O euristică puternică, dezvoltată de Bitner şi Reingold în [BR75], este adesea folosită împreună cu algoritmul de forward checking. Conform acestei metode, variabilele cu cele mai puţine alternative rămase sunt selectate pentru instanţiere. O astfel de ordine de instanţiere a variabilelor este determinată dinamic, ea fiind numită *metodă de căutare prin rearanjare*. Această euristică a mai fost studiată extensiv în [Pur83]. Purdom şi Brown arată că pentru o clasă importantă de probleme, euristica căutării prin rearanjare furnizează o îmbunătăţire consistentă pentru căutarea bazată pe backtracking.

O altă euristică dezvoltată este aceea de a instanţia prima dată acele variabile care participă la cât mai multe constrângeri. Aceasta încearcă să elimine cât mai timpuriu ramurile din arbore care produc insucces [FQ85]. Pentru aceasta sunt introduse mulţimile stabile. O mulţime formată din variabile care nu au constrângeri directe între orice pereche de variabile se numeşte *mulţime stabilă de variabile*. În această euristică, algoritmul backtracking instanţiază la final toţi membrii mulţimii stabile. Instanţierea acestor variabile are ca efect principal faptul că spaţiul de căutare nu se mai multiplică ca de obicei ci creşte mult mai încet .

O altă euristică de selectare a ordinii variabilelor, numită *cardinalitatea maximă*, a fost dezvoltată de Dechter în [DM94]. Această euristică selectează prima variabilă arbitrar, apoi selectează fiecare variabilă prin alegerea acelei variabile ce este conectată cu cel mai mare număr de variabile alese anterior. O variaţie a acestei euristici este de a alege ca prima variabilă să fie una din cele care participă în majoritatea constrângerilor.

Trebuie precizat faptul că utilizarea euristicilor pentru ordinea variabilelor nu schimbă complexitatea algoritmilor de tip backtracking, în cazul cel mai nefavorabil. În [DM94] sunt realizate mai multe experimente ce compară performanţele mai multor euristici. Aceste experimente arată că anumite euristici aduc îmbunătăţiri semnificative, dar esenţa este aceeaşi.

Odată ce o decizie a fost luată în legătură cu instanţierea unei variabile, vor fi disponibile doar anumite valori de instanţiere pentru celelelalte. Ordinea în care aceste valori sunt considerate poate avea un impact substanţial asupra timpului de găsire a primei soluţii. O altă euristică s-a dezvoltat pe ideea de a prefera acele valori care maximizează numărul de opţiuni pentru atribuirile viitoare [HE80]. Acolo se arată, folosind metode de analiză probabilistică, că alegerea unei variabile cu cel mai mic număr de valori rămase micşorează şansele de a avea variabile instanţiate consistent.

Minton, Johnston, Philips şi Laird construiesc în [MJPL90] o altă euristică numită *min-conflict heuristic*. Ideea lor este de a pleca iniţial cu anumite valori asociate variabilelor şi să se încerce apoi repararea valorilor variabilelor până când o soluţie corectă este obţinută. Conform acestei scheme, backtracking-ul pleacă cu o bună asociere de valori la variabile (adică valori ce nu îndeplinesc numai câteva constrângeri) ce se poate obţine printr-o metodă Greedy. Apoi, valorile variabilelor ce sunt în conflict cu alte variabile, sunt schimbate sistematic în algoritmul lor (principiul lor mai era numit *hill climbing)*. Această euristică este folosită în cazul tehnicii AWCS.

O altă euristică este să se aleagă acea valoare (din cele disponibile) care conduce cel mai uşor la rezolvarea CSP. În [DP88], discută un mod de estimare a dificultăţii rezolvării CSP. Conform acestei metode, problema CSP este convertită la un arbore structurat CSP, prin ştergerea unui număr minim de arce şi obţinerea unei CSP ce păstrează toate soluţiile. Numărul de soluţii găsite în arborele structurat CSP este luat ca unitate de măsură a dificultăţii de rezolvare a CSP.

2.3 Concluzii

O mare varietate de probleme din domeniul AI şi din alte domenii ale ştiinţei calculatoarelor, pot fi văzute ca şi cazuri speciale de CSP. Modelul CSP este un model recent dezvoltat, aflat la confluenţa mai multor domenii ale ştiinţei calculatoarelor, incluzând inteligenţa artificială, teoria algoritmilor, limbaje de programare, calcul simbolic, teoria grafurilor, programare logică, programare distribuită. De aici, interesul crescut din ultimul deceniu pentru studiul metodelor de rezolvare CSP.

În consecinţă, au fost dezvoltate un număr mare de soluţii pentru rezolvarea acestor probleme, cele mai importante fiind descrise în acest capitol. Cele mai cunoscute sunt cele bazate pe utilizarea backtrackingului, ca şi metodă de căutare pentru posibile soluţii, şi propagarea constrângerilor.

S-au făcut multe cercetări în scopul îmbunătăţirii tehnicilor de căutare de tip backtracking şi reducerii costurilor necesare găsirii soluţiilor. Drept urmare, s-au descoperit multe tehnici derivate, cele mai importante fiind prezentate în cadrul capitolului: tehnicile backjumping, strategii look-ahead, algoritmii de tip forward-checking, etc. O altă categorie importantă de algoritmi, derivaţi din cei de tip backtracking, sunt algoritmii ce învaţă, algoritmi ce păstrează înregistrări adiţionale cu constrângeri în timpul căutării. Ideile acestor tehnici au fost aplicate, apoi, în cadrul distribuit pentru a obţine metode similare de rezolvare pentru modelul distribuit DCSP.

Un alt domeniu în care s-au făcut cercetări şi s-au obţinut rezultate a fost cel al propagării constrîngerilor, unde s-au descoperit foarte mulţi algoritmi de preprocesare a problemelor, înainte de a aplica metodele de căutare.

S-au realizat multe studii comparative privind eficienţa acestor metode şi s-au propus tehnici standard de evaluare a eficienţei acestor algoritmi. Foarte multe metode CSP au fost adaptate la clase particulare de probleme şi dezvoltate mai departe doar pentru acele clase de probleme. Aminitim de categoria problemelor combinatoriale, de problemele de planificare şi programare, de alocarea resurselor. Modelarea CSP se aplică astăzi, în industrie, ea oferind soluţii foarte bune pentru multe probleme.

S-au pus la punct medii de programare ce implementează modelul CSP şi folosesc tehnicile de rezolvare CSP, dintre ele aminitm categoria de limbaje CLP (Constraint Logic Programing).

Multe din aceste tehnici au fost extinse la cadrul distribuit, ideiile prezentate aici fiind folosite la construirea unor tehnici eficiente în cadrul distribuit. De exemplu, strategia tehnicii graph-based backjumping a fost introdusă în cadrul distribuit obţinând o tehnică eficientă.

Scopul acestui capitol a fost acela de a descrie cadrul de lucru pentru tehnicile centralizate. Acest cadru de lucru va fi extins la modelul distribuit, iar tehnicile centralizate vor fi extinse la cadrul distribuit având ca ţintă rezolvarea unor probleme de căutare prin folosirea mai multor agenţi. În capitolul următor va fi prezentat modelul DCSP şi tehnicile de căutare asincrone, multe dintre tehnici folosind ideiile tehnicilor centralizate.

3. PROBLEME DE SATISFACEREA CONSTRÂNGERILOR DISTRIBUITE. MODELUL DCSP

În aceast capitol este descris cadrul de lucru al problemelor de satisfacerea constrângerilor distribuite şi se prezintă cele mai importante tehnici de rezolvare a acestor tipuri de probleme. Acest cadru va fi suportul de studiu al tezei de faţă. Pentru fiecare din aceste tehnici se prezintă principalele rutine de tratare a mesajelor împreună cu mecanismul de funcţionare.

Acest cadru de modelare a problemele CSP se bazează pe împărţirea diferitelor părţi ale problemei între *agenţi* care acţionează independent şi care colaborează în scopul obţinerii soluţiei. Acest nou tip de modelare numit **Distributed Constraint Satisfaction Problem**, pe scurt DCSP, s-a dovedit viabil, oferind posibilitatea reducerii costurilor obţinerii soluţiei, comparativ cu tehnicile CSP.

Modelul DCSP se obţine prin extensia modelării bazate pe constrângeri la cadrul de programare concurent şi distribuit, prin distribuirea variabilelor şi a constrângerilor între diverşi agenţi. Acest tip de modelare distribuită a apărut în mod natural deoarece la anumite probleme era imposibil de a aduna toate informaţiile la un singur agent astfel încât să se aplice algoritmi centralizati, neparaleli, cum sunt cei prezentaţi în capitolul 2. Mai mult, în cazul anumitor probleme, costul colectării informaţiilor la un singur agent poate fi foarte mare din cauza costurilor de comunicare, dar şi a costurilor necesare translatării informaţiilor într-un format comun.

3.1. Definirea modelului Problemelor de Satisfacere a Constrângerilor distribuite.

În acest paragraf sunt definite noţiunile de bază necesare construirii modelului DCSP. Acesta este modelul care stă la baza cercetării abordate în teza de faţă [SGM96], [YDIK98], [BMM01], [Yok00], [Yok01].

3.1.1 Definirea problemelor de satisfacere a constrângerilor distribuite.

Definiţia 3.1.1 (Modelul DCSP). Modelul bazat pe constrângeri distribuite Distributed Constraint Satisfaction Problems-DCSP (Distributed CSP) este un CSP, în care variabilele şi constrângerile sunt distribuite între agenţi autonomi ce comunică şi colaborează în scopul obţinerii soluţiei pe baza unui model de comunicaţie. Formal, modelul DCSP se defineşte ca fiind un 5-tuplu ***P={X, D, C, A, ϕ}***, unde:

- ***X*** este un ansamblu finit de ***n*** variabile $\mathbf{x_1, x_2,..., x_n}$, ce pot lua valori finite, din mai multe domenii discrete finite $\mathbf{D_1, D_2,..., D_n}$ ce formează mulţimea ***D***.
- ***C*** este o mulţime de constrângeri între aceste variabile.
- ***A*** este o mulţime de *p* agenţi autonomi.
- $\phi: \boldsymbol{X \rightarrow A}$ este o funcţie ce asociază fiecare variabilă la un anume agent (fiecare variabilă este legată doar de un singur agent).

Distribuirea variabilelor la agenţi duce la împărţirea mulţimii constrângerilor în două submulţimi: $\boldsymbol{C}_{intra} = \{c_{ij}$, unde $\phi(x_i) = \phi(x_j)\}$- mulţimea constrângerilor asociate unui singur agent şi $\boldsymbol{C}_{inter} = \{c_{ij}$, unde $\phi(x_i) \neq \phi(x_j)\}$- mulţimea constrângerilor dintre agenţi.

Definiţia 3.1.2 (Model de comunicaţie pentru agenţi). Agenţii comunică pe baza următorului model de comunicaţie :

- agenţii comunică diversele informaţii prin transmiterea de mesaje;
- un agent poate transmite mesaje la alţi agenţi dacă agentul cunoaşte adresa sau identificatorul acelor agenţi;
- există o întârziere în furnizarea (transmiterea) mesajelor, întârziere care se consideră finită, dar aleatoare;
- pentru transmiterea de mesaje între orice pereche de agenţi, mesajele sunt recepţionate în ordinea în care au fost transmise.

Fiecare agent are "în grijă" câteva variabile şi încearcă să determine valorile corecte pentru acestea. Formal, se vor nota cei *p* agenţi prin $A_1, A_2,..., A_p$, iar fiecare variabilă x_j va fi în „grija" unui agent A_i. Constrângerile sunt, de asemenea, distribuite între agenţi.

Fără a pierde din generalitate, în ceea mai mare parte a tezei se va considera că fiecare agent are *în grijă* doar o singură variabilă (agentul va fi identificat prin indicele variabilei) şi toate constrângerile sunt binare (adică implică două variabile). De asemenea, fiecare agent cunoaşte care sunt constrângerile legate de variabila sa.

Definiţia 3.1.3 (Rezolvarea unei probleme DCSP).

Ca şi în cazul centralizat, rezolvarea unei probleme modelate DCSP, ***P={X, D, C, A, ϕ},*** presupune găsirea unei asocieri de valori la toate variabilele astfel încât fiecare constrângere să fie îndeplinită.

În mod obişnuit agenţii asociază valori la variabilele asociate, valori ce sunt consistente local, comunicând aceste valori la agenţii cu care sunt conectaţi datorită existenţei constrângerilor. Prin urmare, agenţii trebuie să coopereze în scopul găsirii acelei asocieri pentru fiecare variabilă astfel încât constrângerile să fie îndeplinite.Fie exemplul 2.1.1, modelat ca şi problemă CSP-problema celor n regine. Transformarea problemei celor n regine în varianta distribuită presupune asocierea unui agent la fiecare regină, agent ce va încearca să găsească o poziţie corectă pentru aceea regină astfel încât să nu se atace cu celelalte regine. Pentru această problemă, constrângerile existente între doi agenţi (pentru două variabile x_i şi x_j, $x_i \neq x_j \wedge |x_i - x_j| \neq |i-j|$) se distribuie doar la unul din agenţi, care devine agent evaluator. În acest mod se obţine o varianta distribuită ce va fi identificată ca problema distribuită a celor n- regine.

3.1.2 Căutare paralelă şi distribuită.

Trebuie făcută distincţie între modelele de programare paralele şi cele distribuite. În algoritmii de backtracking paraleli, mai multe procesoare execută fiecare câte un algoritm de backtracking local astfel încât arborele de căutare este împărţit în mai multe părţi disjuncte. Implementările şi abordările distribuite de căutare, ce sunt descrise în această teză sunt diferite de cele paralele. Un agent (care poate fi un procesor) nu implementează un backtracking local. Agentul participă interdependent la căutare, agoritmul de căutare fiind obţinut ca rezultatul cooperării tuturor agenţilor. Cu alte cuvinte, modelarea prezentată mai departe se referă la cooperarea agenţilor în obţinerea soluţiei finale prin aplicarea unui singur algoritm de căutare.

Se va putea observă faptul că algoritmii pentru rezolvarea problemelor distribuite pot fi similari cu cei pentru procesele paralele sau distribuite, dar motivaţia este fundamental diferită. Preocuparea principală a algoritmilor care procesează paralel este eficienţa şi se caută folosirea de arhitecturi paralele sau distribuite care să ofere o soluţie eficientă. În schimb, în varianta distribuită CSP, există situaţii în care cunoştinţele despre problemă (adică variabile şi constrângeri) sunt distribuite automat între agenţi. De exemplu, pot exista situaţii în care mai mulţi agenţi au cunoştinţe diferite şi parţiale despre problema globală pentru a o rezolva. Prin urmare, într-o astfel de situaţie scopul principal este de afla cum se ajunge la o soluţie. Şi în cazul tehnicilor de rezolvare DCSP se va pune problema eficienţei, dar aceste tehnici şi această modelare sunt de preferat mai ales pentru aceste situaţii.

O altă problemă este legată de existenţa unor situaţii în care nu se poate aplica modelul CSP. Dacă toate cunoştinţele despre problemă pot fi stocate la un agent, acest agent poate rezolva problema singur utilizând algoritmii centralizaţi existenţi pentru CSP. Totuşi, colectarea tuturor informaţiilor despre problemă la acel agent necesită mai multe costuri legate de comunicarea acestor informaţii şi de translatarea lor într-un format comun. Acest cost al centralizării tuturor cunoştinţelor la un singur agent poate fi prohibitiv. De asemenea, există probleme în care păstrarea tuturor informaţiilor la un agent este indezirabilă sau imposibilă din motive de securitate sau personale. În aceste cazuri, mai mulţi agenţi pot rezolva problema fără a centraliza toate informaţiile. Pe lângă aceste probleme, principala motivaţie a modelării DCSP, rămâne rezolvarea eficientă a acestor probleme, prin reducerea costurilor de căutare şi folosirea în comun a mediilor distribuite existente astăzi.

3.2 Metode de rezolvare a problemelor DCSP. Tehnici de căutare asincrone

În ultimii ani, s-au descoperit şi formalizat o serie de algoritmi pentru rezolvarea problemelor DCSP. În general, algoritmii cunoscuţi din cadrul modelării CSP, au fost extinşi în cadrul modelării distribuite, aşa cum se va vedea mai departe în acest paragraf, dar s-a exploatat şi cadrul nou al modelării paralele şi distribuite. Aceste metode de rezolvare se mai numesc şi *tehnici de căutare asincrone*.

Aceşti algoritmii sunt clasificaţi de Yokoo în [Yok00],[Yok01] în două mari clase:

- algoritmi de bază (variantele de backtracking, variante de backtracking îmbunătăţit, algoritmii de consistenţă, etc.);
- algoritmi particulari pentru anumite clase de probleme (algoritmi DCSP cu o singură variabilă locală, cu variabile locale multiple şi CSP parţial distribuite).

Tehnicile de căutare asincrone se caracterizează prin diverse proprietăţi.Ele sunt o mixtură de tehnici de căutare, tehnici de consistenţă, tehnici de reordonare pentru agenţi, tehnici de backjumping sau tehnici de învăţare. O altă clasificare a acestor tehnici este realizată de Silaghi în [Sil02].Această clasificare, actualizată, este prezentată mai jos:

a. Variante derivate direct din tehnica *ABT-Asynchronous Backtracking***:**
- *ABT* cu euristica min-conflict pentru ordinea valorilor variabilelor-Yokoo [Yok00], [Yok01]**.**
- ABT_p**,** obţinut prin aplicarea tehnicii etichetării asocierilor-Silaghi [Sil02].
- ABT_r**,** obţinut prin aplicarea tehnicii retransmisiei mesajelor-Silaghi [Sil02].
- ABT_{kernel}, obţinut prin unificarea tehnicilor asincrone-Bessiere [BMM02]
- ABT_0 obţinut plecând de la nucleul ABT_{kernel} şi adăugând legături înainte de începerea căutării-Bessiere [BM05]. Este asemănătoare cu tehnica DIBT.
- ABT_1 obţinut plecând de la nucleul ABT_{kernel} şi adăugând legături în timpul căutării -Bessiere [BM05].
- *ABTR* obţinut prin aplicarea reordonării-Silaghi [Sil02].
- *ABT cu ordine dinamică*- ABT_DO se obţine din ABT prin aplicarea unor ordine dinamice determinate cu ajutorul unor euristici- Zivan [ZM05].

b. Tehnica *AWCS - Asynchronous Weak –Commitment Search***:**
- tehnica *AWCS* împreună cu strategia selectivă de learning ce limitează numărul de nogood-uri, algoritmul este incomplet-Yokoo [YDIK98].
- tehnica *AWCS cu rezolventul bazat pe învăţare***.** Se obţine plecând de la AWCS la care se aplică un algoritm de învăţare numit *rezolvent bazat pe învăţare*-Hirayama [HY00].
- tehnica *AWCS cu minim conflict*. Se obţine plecând de la AWCS la care se aplică o metodă de învăţare bazată pe mulţimea de conflict minim [ML98], [HY00].
- tehnica *AWCS cu nogood caching***.** Se obţine plecând de la AWCS la care se aplică o metodă de învăţare de tipul nogood caching publicată de Havens în [Hav97]**.**

c. Tehnica *DB - Distributed Breakout***:**
- Tehnica *HCC- hill-climbing coalition* este o altă mixtură ce foloseşte ideile din tehnica DB, dar are la bază strategia hill-climbing folosită în DB combinată cu strategia „coaliţiilor” (dacă o coaliţie se măreşte incluzând peste cinci agenţi ea este adusă la starea de start cu noi valori iniţiale) –Hirayama şi Toyoda [HT95].
- Tehnica *DB-BC***,** numită *distributed breakout with broadcasting*, ce diferă de varianta DB prin aceea că mesajele care nu sunt de la

vecini sunt utilizate pentru starea de local–minim –Yokoo [Yok00], [Yok01].

d. Tehnica *AAS - Asynchronous Search with Aggregations***:**
- *AAS-2***:** se bazează pe înregistrarea completă a listelor *nogood*, în mod similar cu algoritmul de backtracking asincron al lui Yokoo (ABT).
- *AAS-1***:** procedează similar cu varianta de backtracking dinamic din [Gin93]. Sunt şterse nogood-urile depinzând de instanţierile variabilelor modificate [SHF00].
- *AAS-0*: este o variantă modificată a lui AAS-1 cu cele mai puţine înregistrări nogood. Fiecare agent păstrează o singură listă nogood explicită care integrează fiecare noua listă nogood care soseşte la agent, folosind regulile de relaxare [SHF00].
- *AASe***:** se obţine din AAS prin extinderea înregistrării valorilor nogood [SHF00].

e. Tehnica *DisDB - Distributed Dynamic Backtracking***:**
- Tehnica *DisDB* - Distributed Dynamic Backtracking varianta de bază - [BMM01] .
- ABT_2 obţinut plecând de la nucleul ABT_{kernel} prin eliminarea legăturilor-Bessiere [BM05]. Este asemănătoare cu tehnica DisDB.

f. Tehnica *DIBT - Distributed Backtracking***:**
- Tehnica *DIBT* [HBQ98] la care se aplică patch-ul din [BMM01] pentru eliminarea situaţiei de incompletitudine găsită de Yokoo.

g. Tehnica *DIFC - Distributed Forward Checking***:**
- Tehnica *DIFC- Distributed Forward Checking*, obţinută plecând de la varianta DIBT prin aplicarea tehnicilor de căutare înainte- [MJ00].
- Tehnica *DIFC- Distributed Forward Checking*, obţinută plecând de la varianta ABT_0 prin aplicarea tehnicilor de căutare înainte- [BM03].

h. Tehnica *ConcBT-Concurrent Backtracking* [ZM04]: este o tehnică nouă în care un număr dinamic de procese independente explorează concurent şi asincron părţi disjuncte ale spaţiului de căutare DCSP.

În acest paragraf vor fi prezentate cele mai importante tehnici de căutare asincrone, punându-se accentul pe acei algoritmi pentru care se vor propune îmbunătăţiri. Primul algoritm prezentat va fi algoritmul *backtracking distribuit sincron* pentru DCSP. Acest prim algoritm distribuit, foarte simplu, se obţine prin extensia algoritmului clasic de căutare sistematică – backtracking, la modelarea distribuită.

Al doilea algoritm ce va fi prezentat este algoritmul de bază *backtracking asincron*. Algoritmul backtracking distribuit asincron este o extensie directă a algoritmului distribuit sincron prin eliminarea defectelor algoritmului sincron. Principalul defect al variantei sincrone consta în nerularea sa în mod concurent şi asincron. În varianta asincronă, agenţii pot rula concurent şi asincron (nu se aşteaptă, ca în cazul sincron, ca un agent să termine instanţierea sa). El este considerat algoritmul clasic de backtracking pentru DCSP, tehnica de bază la care se raportează celelalte tehnici.

Următorul algoritm, numit algoritm *backtracking distribuit asincron îmbunătăţit* (termenul original din [YDIK98]**,** este de asynchronous weak-

commitment search). Acesta este obţinut prin extinderea algoritmului backtraking distribuit asincron, în ideea de a obţine un algoritm mai eficient.

Următorii 2 algoritmi reprezintă algoritmi de căutare asincronă, similari cu algoritmul asincron al lui Yokoo, fiecare bazându-se pe idei diferite. Este vorba de *backtracking distribuit*, varianta Hamadi [HBQ98] şi de *căutare asincronă cu agregate*, având ca autor pe Silaghi [SHF00]**.** Algoritmul backtracking distribuit (DIBT -Distributed Backtracking) este diferit de variantele de backtracking asincron distribuit şi asincron distribuit îmbunătăţit. El are la bază execuţia în graful de bază a unor salturi peste mulţimea agenţilor cu probleme. Algoritmul se bazează pe existenţa unei ordini totale între agenţi. În graful cu constrângeri, constrângerile sunt orientate (dând naştere unui graf aciclic) de la care ierarhia agentului este construită folosind diverse criterii euristice. Celălalt algoritm de căutare asincronă cu agregate se bazează pe noţiunile de constrângeri private şi agregate.

Un alt algoritm descoperit, numit *backtracking distribuit dinamic*, publicat în [BMM00], se bazează pe salturi dinamice peste setul de agenţi aflaţi în conflict, fiind o adaptare a algoritmilor de backjumping (numiţi şi algoritmi de backtracking dinamic) în cadrul DCSP. Acest algoritm, spre deosebire de alţi algoritmi distribuiţi de căutare, nu se bazează pe existenţa de legături suplimentare între agenţi.

În finalul acetui capitol este prezentat un cadru unificator pentru câteva din tehnicile de căutare asincrone, cadru ce formează familia ABT. Acest cadru are ca punct de plecare un nucleu de bază pentru tehnicile ABT, nucleu numit ABT kernel. Plecând de la acest cadru unificator, se poate ajunge la tehnici cunoscute sau la variante apropiate de aceastea: asynchronous backtracking (ABT), Distributed Dynamic Backtracking (DisDB) sau tehnica Distributed Backtracking (DIBT).

3.2.1 Backtracking sincron pentru DCSP.

Algoritmul backtracking distribuit sincron este o extensie directă a algoritmilor de tip backtracking existenţi la varianta centralizată din cadrul CSP, la cadrul DCSP [YDIK98], [Yok00], [Yok01]. Aceştia sunt la bază algoritmi de căutare sistematică pentru rezolvarea CSP. Aşa cum a fost prezentat în capitolul 2, ei asociază pe rând valori la variabile, din domeniile finite, construindu-se soluţia parţială. Iniţial, soluţia parţială este vidă. Apoi, ea este expandată prin adăugarea de variabile (fiecare variabilă primind valori) până când se obţine soluţia completă. Asocierea unei valori la o variabilă şi adăugarea ei în soluţia parţială presupune şi verificarea subsetului parţial de constrângeri (nu se adaugă variabila respectivă în soluţie până când nu se verifică aceste constrângeri). În situaţia în care pentru o variabilă nu se poate asocia o valoare astfel încât să fie satisfăcute constrângerile parţiale, se revine la variabila anterioară şi se încearcă modificarea valorii anterioare.

Cel mai trivial algoritm pentru rezolvarea DCSP constă în alegerea unui *agent lider* dintre ceilalţi agenţi şi, foarte important, stocarea tuturor informaţiilor despre variabile, despre domeniile lor şi despre constrângerile lor la acest agent lider. Agentul lider va rezolva problema ca şi o problemă CSP (ca oricare algoritm centralizat cunoscut în cadrul modelării CSP). Din păcate, costul colectării tuturor informaţiilor despre problemă poate fi prohibitiv de mare. Mai mult, în anumite tipuri de aplicaţii, stocarea informaţiilor la un singur agent este imposibilă (este posibil să nu fie permisă din motive de securitate). Din acest motiv, această soluţie este total ineficientă şi neinteresantă.

Algoritmul backtracking sincron pentru DCSP se obţine din algoritmul standard backtracking, existent pentru CSP. Se consideră că agenţii agreează o

ordine de iniţializare pentru variabile (cum ar fi ordinea lexicografică, x_1 se va iniţializa primul, apoi x_2, s.a.m.d.). Fiecare agent recepţionează soluţia parţială (adică valorile cu care au fost instanţiate variabilele anterioare) de la agentul anterior şi încearcă instanţierea variabilei sale cu o valoare astfel încât să se satisfacă constrângerile legate de aceste variabile. Dacă se găseşte o astfel de valoare, ea este adăugată la soluţia parţială şi transmise la următorul agent (conform ordinii stabilite). Dacă nu există nici o valoare consistentă pentru variabila curentă, atunci se va transmite mesajul *backtracking* la agentul anterior.

Trebuie observat faptul că algoritmul de backtracking sincron nu necesită costuri mari pentru comunicarea de informaţii ca în cazul algoritmului centralizat. Totuşi, determinarea unei ordini de instanţiere a variabilelor necesită costuri. O altă problemă a acestui algoritm simplu, este aceea că el nu beneficiază deloc de pe urma avantajelor paralelismului, din cauză că, la un moment dat, doar un singur agent primeşte soluţia parţială, ceilalţi agenţi fiind în aşteptare. Cu alte cuvinte, algoritmul rezolvă secvenţial problema, chiar dacă modelarea ar permite şi o abordare paralelă.

Algoritmul a fost evaluat în [YDIK98], [Yok00], [Yok01]. El necesită costuri mici doar pentru probleme de dimensiune mică, pentru probleme de dimensiune mare costurile devenind prohibitive. Lipsa paralelismului în varianta sincronă duce la creşterea costurilor spre deosebire de variantele asincrone ce vor fi prezentate mai departe, variante ce exploatează avantajele paralelismului. Există şi o altă variantă îmbunătăţită a acestui algoritm, prezentată în [CDK91], variantă numită *protocol pentru consistenţa reţelei*. În acest algoritm, agenţii construiesc un arbore de căutare în adâncime. Agenţii acţionează sincron pe baza unor privilegii, dar agenţii care au acelaşi părinte în arborele de căutare pot acţiona concurent, îmbunătăţind astfel performanţele algoritmului.

3.2.2. Principiile căutării asincrone

În acest paragraf sunt prezentate mecanismele de bază după care funcţionează tehnicile de căutare asincrone pentru care se vor propune îmbunătăţiri în această teză. Principiile căutării asincrone au fost introduse în [YDIK98], [Yok00], [Yok01], [Sil02]**.**

Agenţii sunt ordonaţi în ordinea priorităţilor. În această teză se va considera că agenţii sunt ordonaţi conform ordinii lexicografice, excepţie făcând tehnicile din familia AWCS care folosesc o ordine dinamică. Astfel, dacă i < j atunci agentul A_i are prioritatea mai mare decât A_j. Fiecare constrângere este aplicată de agentul care are prioritatea cea mai mică dintre cei care implicaţi în constrângere.

Definiţia 3.2.1. - agenţi *conectaţi* (vecini). Între doi agenţi se consideră ca există o conexiune dacă există o constrângere ce implică variabilele lor. Această conexiune va fi folosită pentru transmiterea mesajelor.

Definiţia 3.2.2.– lista agent_view (*agent_view*). Lista agent_view a unui agent A_i este mulţimea celor mai noi asocieri recepţionate de agentul A_i pentru variabilele agenţilor cu care este conectat.

Definiţia 3.2.3.– lista nogood (*nogood*). Lista nogood este o mulţime de asocieri pentru variabile distincte pentru care s-a găsit o inconsistenţă (o constrângere neîndeplinită).

Lista *agent_view* împreună cu valorile nogood stocate formează contextul de lucrul al fiecărui agent, în funcţie de care agentul ia decizii. Lista agent_view este considerată ca o vedere a agentului asupra unor părţi din problemă.

Definiţia 3.2.4.– lista agenţilor părinte. Lista agenţilor părinte pentru un agent A_i este formată din agenţii conectaţi cu el, aflaţi înainte în ordinea priorităţilor (având prioritate superioară).

Definiţia 3.2.5.– lista nogood *consistentă*. O listă nogood recepţionată de agentul A_i este consistentă pentru acel agent, dacă ea conţine aceleaşi asocieri pentru variabilele sale ca şi vederea agentului A_k (agent_view), pentru toate variabilele agenţilor părinte A_k conectaţi cu A_i şi conţine o asociere (x_i, d_i) unde d_k este aceeaşi cu valoarea curentă a agentului A_i.

Definiţia 3.2.6.– mesaj nogood *învechit*. Un mesaj nogood este învechit dacă conţine o listă nogood care nu este consistentă cu contextul agentului receptor.

Cu alte cuvinte, un agent acceptă un mesaj nogood ca fiind consistent dacă mesajul nogood conţine aceleaşi asocieri pentru variabilele agenţilor conectaţi cu el şi valoarea sa curentă coincide cu cea din lista nogood. In caz contrar, un astfel de mesaj nogood este învechit şi nu va fi acceptat şi prelucrat.

Pentru a pune în evidenţă principiile căutării asincrone (căutare asincronă ce stă la baza celor mai importanţi algoritmi distribuiţi existenţi în cadrul DCSP), se consideră exemplul 3.2.1. Fără a pierde din caracteristicile fundamentale ale căutării asincrone, se considera că fiecare agent are în grijă o variabilă şi, prin urmare, este responsabil pentru asocierea valorilor acelei variabile.

Exemplul 3.2.1. Fie o problemă DCSP cu 4 agenţi A_1, A_2, A_3 şi A_4, ce controlează patru variabile x_1, x_2, x_3 şi x_4, cu domenii identice $D_1=D_2=D_3=D_4=\{0, 1, 2, 3\}$. Între aceste variabile există constrângerile din figura 3.1. Există mai multe constrângeri binare între agenţi. Agentul A_3 trebuie să verifice dacă $3x_1+1 > x_3$ şi $x_1>x_3-2$, agentul A_2 verifică dacă $x_1>x_2-2$, iar A_4 verifică constrângerea $x_2+x_3-x_4 >3$. Se remarcă faptul că fiecare constrângere este verificată doar de agentul cu prioritatea mai mică dintre cei implicaţi. Fiecare agent porneşte prin asocierea unei valori aleatoare pentru variabila sa, valoarea aleasă din domeniul său (pentru exemplul de faţă se asociază valoarea 0) .

Spaţiul de căutare local pentru fiecare agent se determină prin constrângerile locale ale sale şi prin restricţiile impuse de alţi agenţi prin trimiterea a două tipuri de mesaje: mesaje *ok?* şi mesaje *nogood*. Aceste două mesaje sunt folosite de agenţi pentru a comunica informaţii despre soluţia parţială ce se construieşte şi despre situaţiile conflictuale identificate. Când un agent asociază o valoare la variabila sa, el trimite un mesaj de tipul *ok?(var=valoare)* la toţi agenţii de prioritate mai mică cu care este conectat. Scopul acestui mesaj este de a informa agenţii de asocierea acestei valori şi de a întreba dacă ea poate fi acceptată. Aceşti agenţi evaluează constrângerile lor în contextul valorii recepţionate. Dacă aceste constrângeri sunt îndeplinite de noua asociere, agenţii trec în aşteptare. În caz contrar, fiecare agent încearcă o nouă asociere de valori pentru variabila sa. Dacă nici o valoare nu mai este disponibilă pentru aceea variabilă agentul în cauză

generează un mesaj nogood ce este transmis la primul agent cu prioritatea mai mică decât cel curent. Agentul ce recepţionează mesajul nogood reţine aceste informaţii din mesajul nogood în spaţiul său local de căutare. Apoi el încearcă să schimbe asocierea sa greşită. Dacă reuşeste informează agenţii cu care este conectat, în caz contrar generând un nou mesaj nogood. Se remarcă faptul că fiecare constrângere este evaluată întotdeauna de agentul cu prioritatea cea mai mică, iar mesajele ok sunt trimise de agenţii cu prioritatea mai mare. Acestea sunt principiile de căutare asincronă ce sunt aplicate de tehnicile din familia ABT sau AWCS.

În figura 3.1 se observă mesajele schimbate pentru exemplul cu 4 agenţi, pentru cazul majoritatea tehnicilor de căutare asincrone, în special cele derivate din tehnica ABT.

Fiecare agent porneşte prin asocierea valorii 0 la variabila sa. Agentul A_1, trimite apoi mesajul *ok?* la A_2 şi A_3. Agenţii A_2 şi A_3 trimit fiecare mesajul *ok?* la agentul A_4. Agenţii A_2 şi A_3 găsesc că valoarea recepţionată de la A_1 este compatibilă cu constrângerile lor. Prin urmare, ei nu mai reacţionează. Totuşi, constrângerea lui A_4 este neîndeplinită şi acest mesaj returnează mesajul *nogood* (în figură este etichetat cu 4) la agentul A_3.

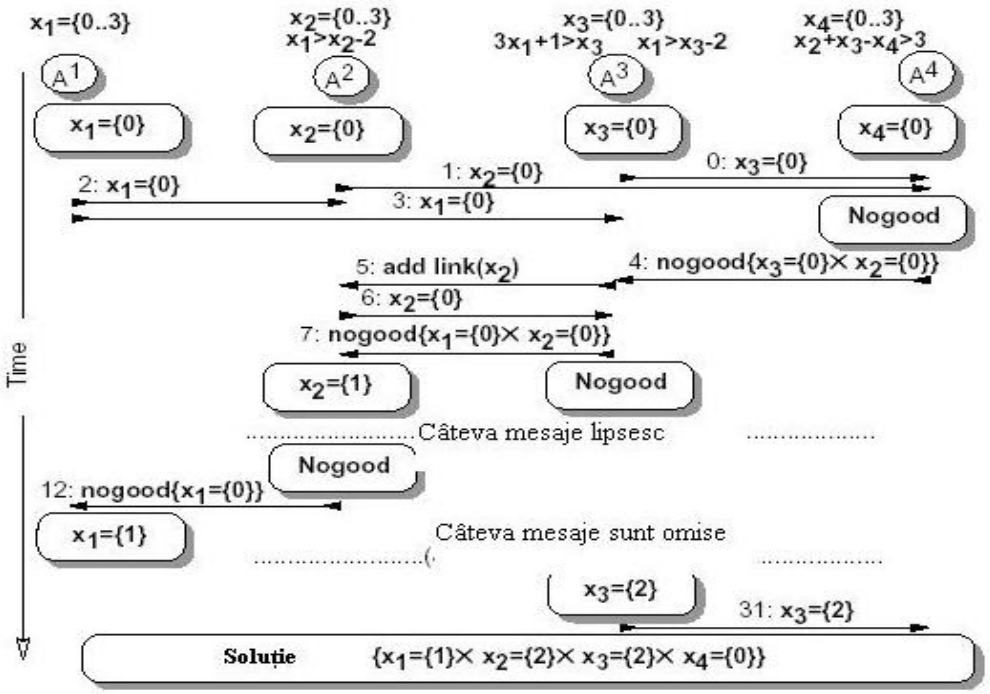


Figura 3.1. Un exemplu de comportament al tehnicilor de căutare asincrone

Aceste principii se vor regăsi în funcţionarea tehnicii ABT (prezentată în paragraful care urmează), dar şi în cazul tehnicilor din familia ABT sau AWCS.

3.2.3 Tehnica „Asynchronous Backtracking"- ABT

Această tehnică reprezintă prima tehnică de căutare asincronă completă, introdusă pentru prima dată în [YDIK92]. În această teză se vor prezenta şi trata variantele îmbunătăţite din [YDIK98], [Yok00], [Yok01].

Conform principiilor căutării asincrone, fiecare agent iniţializează variabila sa în mod concurent şi trimite valoarea la agenţii cu care este conectat direct, aşteaptând apoi să i se răspundă la mesaje. Sunt utilizate canale de comunicaţie directă care funcţionează după principiile FIFO. Există o ordine globală statică între agenţi, în care A_j are prioritate mai mică decât A_i, dacă i<j. Existenţa acestei ordini asigură completitudinea tehnicii ABT . De asemenea, algoritmul ABT necesită direcţionarea constrângerilor (ce apar între doi agenţi). Dintre cei doi agenţi implicaţi într-o constrângere, agentul de prioritate mai mică este cel care va face evaluarea constrângerii.

Algoritmul ABT foloseşte trei tipuri de mesaje :

- mesajul *ok?*, ce conţine o asociere *variabilă-valoare,* este trimis de un agent la agentul evaluator pentru a vedea dacă valoarea este bună. Totdeauna agenţii de prioritate mai mare trimit aceste mesaje la agenţii de prioritate mică.
- mesajul *nogood* ce conţine o listă (notată *nogood*) cu asocierile pentru care s-a găsit o inconsistenţă, el fiind trimis în cazul în care agentul evaluator a găsit o constrângere neîndeplinită şi nu a putut schimba propria valoare.
- mesajul add-link, trimis pentru a anunţa necesitatea creării unei noi legături directe, datorită apariţiei unui *nogood* ce conţine agenţi neconectaţi cu agentul curent.

În figura 3.2. sunt prezentate rutinele de tratare a mesajelor pentru tehnica ABT pentru varianta ABT din [YDIK98], [Yok00], [Yok01].

Fiecare agent primeşte o mulţime de valori de la agenţii cu care este conectat prin legături, aceste valori formând *agent_view* (linia 1 din figura 3.2.) Dacă se primeşte mesajul *ok?,* agentul evaluator adaugă variabila şi valoarea în lista de valori şi verifică dacă noua pereche este consistentă cu celelalte. Verificarea este făcută în rutina check-agent-view din figura 3.2. Dacă se găseşte o asociere care nu este consistentă, agentul încearcă să schimbe valoarea variabilei sale astfel încât să fie consistentă cu valorile din lista *agent_view*. Este posibil ca acel agent să nu găsească nici o combinaţie bună pentru anumite perechi din listă (o astfel de submulţime este nogood), atunci valorile asociate pentru alţi agenţi trebuie schimbate. În această situaţie, se spune că agentul a cauzat un backtrack (este nevoie să se revină şi să se schimbe anumite valori), agentul trebuind să transmită un mesaj *nogood* la unul din agenţi cu care a intrat în conflict (linia 3 din fig. 3.2.).

when received (ok?, (x_j,d_j)) **do** **(1)**
  adaugă(x_j,d_j) la lista *agent_view;*
  check_agent_view;
end do;

when received (nogood, x_j, nogood) **do** **(2)**
  adaugă nogood to *nogood_list;*
  when (x_k,d_k), unde x_k nu este conectat este conţinut în lista *nogood_list* **do**
    agentul x_k cere să se adauge o legătură de la x_k la x_i ;
    adaugă(x_k,d_k) la lista *agent_view;*

```
   end do
   old_value ← current_value
   check_agent_view;
   when old_value = current_value do
     send (ok?, xj, current_value)) to xj
   end do
end do;

procedure check_agent_view
  when agent_view şi curent_value nu sunt consistente do
    if nici o valoare din Di nu este consistentă cu lista agent_view then
        backtrack
    else
        Selectează d∈Di unde agent_view şi d sunt consistente
        current_value ← d
        send (ok?, (xi, d)) to legăturile care pleacă.
    end if
  end do
end procedure

Procedure Backtrack
  nogoods ← {V/ V= submulţime inconsistentă din agent_view}
  when mulţimea vidă este element al mulţimii nogoods do
    transmite la alţi agenţi că nu există soluţie
    opreşte algoritmul
  end do
  for each V ∈ nogoods do
    selectează (xj,dj) unde xj are cea mai mică prioritate din V
    send (nogood, xi, V ) la xj                                    (3)
    şterge (xj, dj) din lista agent_view
  end do
  check_agent_view
end procedure
```

Figura 3.2. Tehnica "Asynchronous Backtracking" şi subrutinele folosite la recepţionarea mesajelor

În cazul recepţionării unui mesaj de tip *nogood* (linia 2 din figura 3.2.) acesta este mai întâi verificat dacă este consistent cu vederea agentului, în caz afirmativ fiind acceptat de agent. Un mesaj nogood acceptat este stocat şi utilizat ulterior ca o constrângere, pentru ca acel agent să nu mai asocieze această valoare. După recepţionarea mesajului agentul verifică dacă acel nogood nu conţine valori pentru agenţi cu care nu este conectat. În caz afirmativ, se transmite mesajul *add-link* ce implică adăugarea unei legături între cei doi agenţi.

Este posibil ca un agent să nu găsească nici o combinaţie bună pentru anumite perechi din listă. Dacă un agent găseşte o astfel de submulţime (listă nogood), atunci valorile asociate pentru alţi agenţi pot fi schimbate. În această situaţie, se spune că agentul a cauzat un backtrack (este nevoie să se revină şi să se schimbe anumite valori). Din figura 3.2 se remarcă faptul că agentul care a cauzat backtrack transmite un mesaj *nogood* la unul din agenţii conectaţi cu el.

Exemplul 3.2.2. Fie o problemă DCSP cu mai mulţi agenţi din care se consideră doar o parte ce implică 3 agenţi A_1, A_2, A_3 ce controlează trei variabile x_1, x_2, x_3 cu domeniile şi constrângerile prezentate în figura 3.3

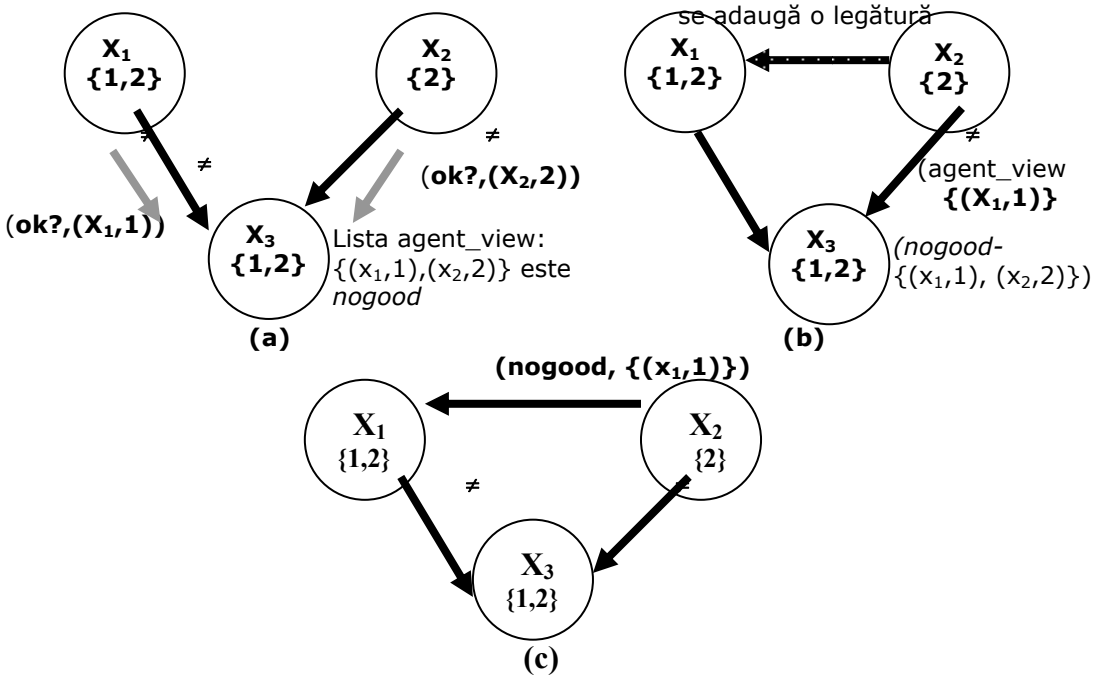


Figura 3.3 Exemplu din execuţia algoritmului ABT

În exemplul 3.2.2 în prima fază agenţii x_1 şi x_2 instanţiază variabilele lor cu 1 sau 2, lista *agent_view* a agentului x_3 va fi formată din $\{(x_1,1), (x_2,2)\}$ (figura 3.3(a),). Din faptul că nu există nici o valoare posibilă pentru x_3 care să fie consistentă cu perechile din lista *agent_view*, această listă agent_view este *nogood*. Agentul x_3 alege din lista agent_view agentul cu cea mai mică prioritate, adică x_2 şi trimite mesajul *nogood.* Prin recepţionarea acestui mesaj *nogood*, agentul x_2 înregistrează acest nogood . Această listă $\{(x_1,1), (x_2,2)\}$ (care este nogood) conţine agentul x_1, care nu este conectat la x_2 printr-o legătură. Prin urmare, o nouă legătură va fi adăugată între x_1 şi x_2 . Agentul x_2 necesită ca x_1 să trimită valoarea sa lui x_2 şi adaugă $(x_1, 1)$ la lista agent_view (figura 3.3.(b)). Agentul x_2 caută dacă valoarea sa este consistentă cu lista agent_view. Din faptul că mesajul nogood recepţionat de la agentul x_3 este compatibil cu asocierea $(x_2, 2)$ şi cu lista sa *agent_view*, asocierea $(x_2, 2)$ este inconsistentă cu agent_view. Lista agent_view $\{(x_1,1)\}$ este nogood pentru că x_2 nu are nici o valoare posibilă. Prin urmare, există numai un agent în această listă nogood (agentul x_1), aşa că x_2 trimite mesajul nogood la agentul x_1 (fig. 3.3.(c)).

În ceea ce priveşte completitudinea, în [YDIK92] se propune o soluţie de eliminare a ciclurilor într-o reţea cu constrângeri, pe baza unei tehnici bazate pe unic identificator, tehnică folosită la evitarea impasului în sistemele cu baze de date distribuite. Această tehnică constă în a folosi o relaţie de ordine totală între noduri. Dacă un nod are un unic identificator, se poate defini o ordine de prioritate între agenţi prin utilizarea unei ordini alfabetice a acelor identificatori (agentul care

precede în ordine alfabetică un agent are prioritate mai mare). Dacă o legătură este direcţionată prin utilizarea acestei ordini de priorităţi (de la agentul mai prioritar la cel de prioritate inferioară), atunci nici un ciclu nu se va produce în reţea. Aceasta înseamnă că pentru fiecare constrângere agentul de prioritate mai mică va fi evaluator şi agentul de prioritate mai mare va transmite mesajul de ok spre evaluator.

Autorii algoritmului demonstrează în [YDIK92] că algoritmul este complet. Ei arată că, dacă există o soluţie, algoritmul ajunge într-o stare stabilă unde toate valorile variabilelor satisfac toate constrângerile şi toţi agenţii sunt în aşteptarea unor mesaje. De asemenea, dacă nu există soluţie, ei arată că algoritmul descoperă această situaţie şi se încheie .

În cazul cel mai nefavorabil, complexitatea algoritmului backtracking distribuit asincron tinde să devină exponenţială, după numărul de variabile n. Complexitatea exponenţială a algoritmului, în cazul nefavorabil, este determinată de numărul înregistrărilor de mesaje nogood.

3.2.4. Tehnica Asynchrounous Weak-Commitment Search-AWCS

Tehnica AWCS este o tehnică hibridă obţinută prin extinderea tehnicii ABT în ideea de îmbunătăţire a perfomanţelor. Tehnica se bazează pe ideile introduse în algoritmul „weak-commitment search" existent în cadrul modelării CSP [YDIK98].

Principala caracteristică a algoritmului weak-commitment search, folosit pentru arhitecturi centralizate, consta în faptul că agenţii revizuiau o decizie proastă fără a face o căutare exhaustivă, lucru posibil prin schimbarea dinamică a ordinii agenţilor. În algoritmul asynchrounous backtracking, ordinea de prioritate a agenţilor era determinată, fiecare agent încercând să îndeplinească constrângerile acelor variabile avute în grijă de agenţii de prioritate mai mare. Când un agent asociază o anumită valoare variabilei avute în grijă, valoarea este fixată pentru mult timp, ea putând fi schimbată doar printr-o căutare exhaustivă a agenţilor de prioritate inferioară. Din această cauză, pentru probleme a căror dimensiune este mare, selectarea unei valori proaste are un efect dezastruos (acesta este un defect comun al algoritmilor de tip backtracking). Acest defect se încearcă a fi eliminat în cazul tehnicii AWCS. Astfel, când un agent nu poate găsi o valoare consistentă pentru agenţii de prioritate mare, este schimbată ordinea de prioritate astfel încât agentul să aibă o prioritate superioară. Efectul acestei decizii este acela că dacă un agent face o greşeală în selectarea unei valori, prioritatea altui agent devine mai mare. Astfel, agentul care a produs greşeala nu va mai lua o decizie proastă şi valoarea selectată va fi schimbată (de aici şi denumirea acestei tehnici).

Algoritmul WCS, ce se aplică la modelul CSP, are la bază două idei principale:

- algoritmul utilizează o euristică parţială, numită min-conflict, euristică folosită la stabilirea ordinii valorilor;
- abandonarea soluţiei parţiale şi repornirea procesului de căutare, dacă nu exista nici o valoare consistentă pentru soluţia parţială.

Aceste două idei sunt aplicate la algoritmul de ABT, obţinându-se această tehnică bazată şi pe o ordine dinamică a agenţilor [YDIK98], [Yok00], [Yok01]. Aplicarea primei caracteristici este destul de simplă. Ori de câte ori se selectează o valoare pentru variabile, dacă există valori multiple consistente cu cele din lista *agent_view* (adică acelea care satisfac constrângerile cu variabilele agenţilor de prioritate maximă), agentul preferă valori care minimizează numărul de constrângeri neîndeplinite cu variabilele agenţilor având prioritatea cea mai mică. În [YDIK98] şi

[Yok00], [Yok01] se arată că acest algoritm poate fi construit prin schimbarea dinamică a ordinii de prioritate. Pentru aceasta, se defineşte o metodă de stabilire a ordinii de priorităţi prin introducerea unor *valori de prioritate*, care se pot schimba folosind următoarele reguli:

- pentru fiecare variabilă/agent, o valoare întreagă ne-negativă reprezintă ordinea de prioritate a acelei variabile/agent . Acea valoare este numită *valoare de prioritate*.
- ordinea se defineşte astfel încât orice variabilă/agent având o valoare de prioritate foarte mare să aibă o prioritate cât mai mare.
- dacă valorile de prioritate pentru mai mulţi agenţi sunt aceleaşi, ordinea se stabileşte alfabetic după identificatorii lor.
- pentru fiecare variabilă/agent, valoarea de prioritate iniţială este 0.
- dacă nu există nici o valoare consistentă pentru x_i,valoarea de prioritate a lui x_i este schimbată la k+1, unde k este cea mai mare valoare prioritară a agenţilor asociaţi.

Cu ajutorul regulilor de mai sus, când apare un backtracking, ordinea de prioritate se va schimba astfel încât agentul ce avea prioritatea cea mai mare înainte de apariţia backtrackingului, să satisfacă constrângerile cu agentul care cauzează backtrackingul şi are acum prioritatea cea mai mare. Pe deasupra, în algoritmul de backtracking asincron, agenţii încearcă să evite situaţiile găsite ca "nogood". Totuşi, datorită întârzierilor ce apar la transmiterea mesajelor, mulţimea agent_view a unui agent poate fi ocazional un superset al valorilor anterioare găsite ca fiind *nogood*. Pentru a evita efectele câtorva din situaţiile instabile şi cele în care s-au făcut schimbări inutile în valorile de prioritate, fiecare agent înregistrează situaţiile de *nogood* care se transmit. Când lista *agent_view* este identică cu lista *nogood* ce a mai fost transmisă, agentul nu va schimba valoarea de prioritate şi va aştepta pentru mesajul următor.

În figura 3.4 este descris algoritmul de bază pentru AWCS [YDIK98], [Yok00], [Yok01] .

Se observă mai multe deosebiri între procedurile tehnicii ABT şi cele din AWCS. În primul rând, la algoritmul ABT fiecare agent transmite valoarea variabilei sale numai la agenţii având prioritatea cea mai mică dintre cei cu care este conectat. În schimb, la algoritmul AWCS fiecare agent transmite valoarea variabilei sale la ambele categorii de agenţi, la cei de prioritatea mai mică şi la cei cu prioritatea mai mare, agenţi de care este legat prin constrângeri (aceşti agenţi sunt notaţi în figura 3.4 cu *neighbors).*

O altă deosebire este legată de faptul că valoarea de prioritate este comunicată prin mesajul *ok?* (în algoritmul din fig. 3.4. linia etichetată cu 1). A treia diferenţă este dată de faptul că ordinea de prioritate este determinată utilizând valoarea de prioritate comunicată. Dacă valoarea curentă nu este consistentă cu lista *agent_view*, adică câteva constrângeri cu variabilele agenţilor cu prioritatea cea mai mare nu sunt satisfăcute, agentul schimbă valoarea sa astfel încât să fie consistentă cu lista *agent_view* şi să minimizeze numărul de constrângeri în conflict cu variabilele agenţilor cu prioritatea cea mai mică (în algoritm este notat acest punct cu (2)). Ultima diferenţă apare atunci când x_i nu poate găsi o valoare consistentă cu lista *agent_view.* În acest caz, x_i transmite mesajul *nogood* la alţi agenţi şi incrementează valoarea de prioritate a sa. Dacă x_i a mai trimis un mesaj identic de tipul *nogood,* x_i nu va mai schimba valoarea de prioritate dar va aştepta următorul mesaj (punctul (3) din algoritm).

```
 when received (ok?, (xj,dj, prioritate))  do                    -(1)
   adaugă(xj,dj, prioritate) la lista agent_view;
   check_agent_view;
end do;

when received (nogood, xj, nogood)  do
   adaugă nogood  to  nogood_list;
   when (xk,dk,prioritate), unde xk nu este în lista neighbors şi este conţinut în nogood   do
     adaugă xk la lista neighbors
     adaugă (xk,dk,prioritate) la lista agent_view;
   end do
   check_agent_view;
end do;

procedure check_agent_view
  when  agent_view şi curent_value  nu sunt consistente  do
    if   nici o valoare din Di nu este consistentă cu lista agent_view then
      backtrack
    else
      Selectează d∈Di unde agent_view şi d sunt consistente şi d minimizează număr.
      de constrângeri neîndeplinite de agenţii de prioritatea cea mai mică        -(2)
        current_value ← d
        send (ok?, (xi, d,prioritate_curentă)) la  neighbors
    end if
  end do
end procedure

Procedure Backtrack
  nogoods ← {V/ V= submulţime inconsistentă din  agent_view}   -(3)
  when  mulţimea vidă este element al mulţimii  nogoods  do
    transmite la alţi agenţi că nu există soluţie
    opreşte algoritmul
  end do
  when  nici un element din mulţimea  nogoods nu este inclus lista noogod_sent   do
   for each  V ∈ nogoods  do
     adaugă V la nogood_sent
     for  each (xj,dj,pj) din V
        send  (nogood, xi, V ) la xj
     end do
   end do
   Pmax←max(xj,dj,pj) ∈agent:view (pj)
   current_priority ← 1+Pmax
  Selectează d∈Di unde d minimizează numărul de constrângeri neîndeplinite de
  agenţii de prioritatea cea mai mică
   current_value ← d
    send (ok?, (xi, d,prioritate_curentă)) la  neighbors
  end do
end procedure
```

Figura 3.4. Algoritmul AWCS şi subrutinele folosite pentru recepţionarea mesajelor

Pentru o mai bună înţelegere a modului de aplicare a tehnicilor derivate prezentate în capitolele următoare, se prezintă mai multe informaţii legate de comportamentul unui agent A_i, pentru tehnica AWCS [YDIK98]. Când un agent A_i recepţionează un mesaj ok?, el îşi actualizează lista agent_view şi verifică dacă anumite valori nogood sunt neîndeplinite. Agentul A_i testează numai acele valori nogood ce au prioritatea mai mare decât a lui x_i (autorii folosesc termenul de nogood de prioritate maximă pentru aceste valori). De fapt, prioritatea unei valori nogood este definită ca cea mai mică prioritate a variabilelor din nogood, cu excepţia lui x_i. Ca o concluzie, un agent generic A_i poate avea următorul comportament:

- dacă nici o valoare de prioritate superioară nu este îndeplinită, el nu face nimic.
- dacă există câteva valori nogood de prioritate superioară ce au valori inconsistente şi aceste valori pot fi eliminate prin schimbarea valori lui x_i, agentul va schimba această valoare şi va trimite mesajul ok?. Dacă are de ales din mai multe valori, el va selecta aceea valoare ce minimizează inconsistenţele în valorile nogood de prioritate inferioară (o valoare nogood de prioritate inferioară este aceea în care prioritatea sa este inferioară priorităţii lui x_i).
- dacă câteva valori de prioritate superioară sunt inconsistente şi nu se poate elimina această inconsistenţă, agentul creează un nou mesaj nogood în afara listei sale agent_view şi trimite un mesaj nogood la fiecare agent ce are variabile în nogood. Apoi, agentul măreşte prioritatea lui x_i, schimbând valoarea lui x_i cu o altă valoare ce minimizează numărul de inconsistenţe la toate valorile nogood şi trimite mesajul ok?. Dacă noul nogood este identic cu valoarea nogood anterioară, atunci agentul nu face nimic. Acest pas este necesar pentru asigurarea completitudinii algoritmului [YDIK98].

Trebuie subliniat un anume comportament al agentului A_i, specific algoritmului AWCS: când se recepţionează un mesaj nogood, agentul adaugă valoarea nogood la mulţimea sa nogood şi execută un test de verificare a inconsistenţelor pentru nogood. Dacă noul nogood include o variabilă necunoscută, agentul are nevoie să primească de la agentul corespunzător valoarea variabilei avute în grijă. Din păcate, informaţiile din nogood nu sunt folosite complet. Este vorba despre cazul atribuirii unei noi valori pentru variabila asociată agentului. Este posibil ca valorile nogood să conţină o referire la această valoare, în sensul că aceea atribuire să mai fi apărut ca şi inconsistentă. Utilizarea acestor informaţii va fi baza construirii tehnicii nogood processor.

Problema completitudinii algoritmului AWCS este rezolvată plecând de la completitudinea algoritmul de bază ABT din care este derivat. În plus faţă de tehnica ABT, autorii identifică situaţiile în care pot apărea situaţii de blocaj. Prioritatea valorilor se schimbă dacă şi numai dacă o combinaţie ce nu este soluţie este găsită. Dar, numărul de combinaţii nogood este finit (chiar dacă este mare), prioritatea valorilor nu poate fi schimbată la infinit. Prin urmare, după un anumit timp, cu siguranţă finit, prioritatea valorilor va fi stabilă. Autorii, în [Yok00], [Yok01] şi [YDIK98], arată că situaţia de mai sus nu poate apărea şi în cazul în care prioritatea valorilor este stabilă.

În ceea ce priveşte performanţele acestui algoritm, satisfacerea constrângerilor este, în general, NP-completă. Prin urmare, în cazul cel mai nefavorabil, şi complexitatea algoritmului AWCS creşte exponenţial după numărul de

variabile *n*. Acest caz poate apărea în momentul în care algoritmul schimbă ordinea de căutare. Autorii sugerează faptul că se poate restricţiona numărul de înregistrări care sunt *nogood*, adică fiecare agent înregistează numai un număr fix al celor mai recente *nogood* găsite. Acest lucru are un preţ şi anume nu se mai poate garanta completitudinea algoritmului. Autorii arată că, în practică, o limitare la 10 este suficientă (în cele mai multe cazuri) pentru găsirea soluţiei.

Rezultatele experimentale din [Yok00], [Yok01] şi [YDIK98] arată că acest algoritm, în care agenţii pot modifica o decizie proastă fără a face o căutare exhaustivă, este mai performant decât algoritmul ABT. Ideea de a permite agenţilor să poată modifica o decizie proastă, prin schimbarea dinamică a ordinii de prioritate a agenţilor s-a dovedit benefică. Conform rezultatelor experimentale, algoritmul AWCS, mai ales pentru probleme de dimensiune mare, s-a dovedit foarte eficient. Mai mult, algoritmul a putut da un răspuns la o anumită instanţă într-un timp rezonabil, ceea ce algoritmul de bază (pentru valori mai mari ale problemelor DCSP) nu a reuşit. Se poate concluziona că organizarea flexibilă a agenţilor (adică posibilitatea ca ei să-şi schimbe ordinea de prioritate) este mult mai bună ca şi varianta statică.

3.2.5 Tehnici incomplete de căutare. Tehnica "Distributed BreakOut".

Un alt algoritm distribuit, inspirat şi el dintr-un algoritm existent pentru CSP, în cadrul arhitecturilor centralizate, este algoritmul *distributed breakout* [YDIK98], [Yok00], [Yok01]. Algoritmul din care este inspirat acest nou algoritm este algoritmul breakout al lui Morris [Mor93], acesta fiind la bază un algoritm de îmbunătăţire iterativă.

Pentru a descrie noul algoritm distribuit, mai întâi, se vor descrie elementele de bază după care funcţionează algoritmul centralizat al lui Morris [Mor93]. Acesta foloseşte noţiunea de soluţie cu defecte (flawed solution), care este o soluţie ce conţine câteva constrângeri neîndeplinite. O astfel de soluţie este revizuită prin schimbări locale până când toate constrângerile sunt satisfăcute.

Pentru fiecare pereche de variabile şi valori care nu îndeplinesc constrângerile se defineşte ponderea acelei perechi (care iniţial este 1) iar suma ponderilor perechilor care nu îndeplinesc constrângerile este folosită pentru evaluarea unei soluţii cu defecte. În prima etapă, suma anterioară este egală cu numărul de constrângeri neîndeplinite. În algoritmul breakout, valorile variabilelor sunt schimbate astfel încât să scadă valoarea acelei sume de evaluare (adică a numărului de constrângeri violate). Dacă valoarea nu poate fi micşorată prin schimbări ale valorilor variabilelor, starea curentă este numită *local-minimum*. Când se ajunge într-o astfel de situaţie, algoritmul breakout creşte ponderea perechilor de constrângeri neîndeplinite în starea curentă, cu o unitate, astfel încât valoarea de evaluare la acestă stare să devină mai mare decât starea vecinilor; astfel algoritmul poate evada din starea de *local-minimum*.

În continuare sunt prezentate ideile de bază ale algoritmului distribuit, varianta din [YDIK98], [Yok00], [Yok01]. Pentru fiecare agent x_i, autorii definesc o mulţime de agenţi ce sunt direct conectaţi la x_i printr-o legătură, mulţime notată în algoritm prin *neighbors* (mulţimea vecinilor lui x_i identici cu cei din algoritmul AWCS). În algoritm este definită distanţa între doi agenţi ca fiind numărul de legături ale celui mai scurt drum ce conectează cei doi agenţi.

În [YDIK98] se arată că aplicarea algoritmului breakout la cadrul distribuit al DCSP a necesitat mai multe modificări. Astfel, dacă se permite numai unui agent să schimbe valoarea sa în acelaşi timp, nu se va profita de avantajele paralelismului. Pe de altă parte, dacă la două mulţimi de agenţi vecini li se permite să schimbe valorile lor în acelaşi timp, valoarea de evaluare nu mai poate fi îmbunătăţită, apărând oscilaţii în care agenţii tot repetă aceleaşi operaţii. O altă problemă este legată de cum se va detecta faptul că agenţii sunt prinşi într-o minimă locală, faptul că agenţii trebuie să facă schimb global de informaţii despre ei. Autorii rezolvă aceste probleme, prin aplicarea următoarelor idei :

- agenţii vecini schimbă valorile astfel încât să obţină posibile îmbunătăţiri. Pentru aceasta, doar agentului care maximizează îmbunătăţirea valorii de evaluare i se dă dreptul de a schimba valoarea sa. De asemenea, doi agenţi care nu sunt vecini pot să-şi schimbe concurent valorile lor.
- în loc de a detecta faptul că agenţii sunt în ansamblu într-un minim-local, fiecare agent detectează faptul că el este într-un semi minim-local, care este o formă mai slabă de minim-local ce poate fi detectată local.

În acest algoritm, două tipuri de mesaje *ok?*şi *improve* sunt transmise între vecini [YDIK98], [Yok00], [Yok01]. Mesajul *ok?* este utilizat pentru a schimba valoarea curentă asociată la un agent, iar mesajul *improve* este utilizat pentru a comunica o posibilă îmbunătăţire a valorii de evaluare prin schimbarea valorii agentului. Prin schimbarea mesajului *improve* între vecini şi oferirea dreptului de schimbare a valorii numai la agentul care poate maximiza îmbunătăţirea valorii evaluate, vecinii nu pot schimba valorile lor în mod concurent, în timp ce agenţii care nu sunt vecini pot face acest lucru. Autorii definesc faptul că agentul x_i este într-un semi local-minim dacă x_i violează câteva constrângeri şi îmbunătăţirile posibile ale lui x_i şi ale tuturor vecinilor lui sunt 0.

În acest algoritm fiecare agent îşi determină în mod aleator valoarea iniţială şi transmite mesajul *ok?* la toţi vecinii lui. După recepţionarea mesajelor *ok?* de către toţi vecinii, se calculează valoarea curentă de evaluare (suma ponderilor perechilor de constrângeri neîndeplinite relative la variabilele lor) şi îmbunătăţirea posibilă a valorii de evaluare. Apoi se transmite mesajul *improve* la toţi vecinii săi.

Procedurile executate de agentul x_i când recepţionează mesajele *ok?* şi *improve?* sunt descrise în figurile 3.5. şi 3.6, ele alcătuind de fapt algoritmul distribuit breakout [YDIK98],[Yok00], [Yok01]. Agenţii alternează între modul wait_ok? (figura 3.5) şi modul wait_improve (figura 3.6).

În modul wait_ok?, x_i înregistrează valoarea asociată la vecini în agent_view. După recepţionarea mesajului *ok?* de la vecinii săi, se calculează valoarea curentă de evaluare şi îmbunătăţirea posibilă, apoi se transmite mesajul *improve* la vecinii săi şi se intră în modul wait_improve.

Algoritmul DB utilizează următoarele structuri de date:

- *can_move* reţine când îşi poate schimba valoarea variabila x_i. Dacă o posibilă îmbunătăţire a vecinilor x_j este mai mare ca şi îmbunătăţirea lui x_i sau îmbunătăţirile sunt egale şi x_j precede pe x_i în ordine alfabetică, variabila can_move va deveni *false.*
- *quasi_local_minimum* stochează momentul în care x_i este într-un semi minim-local. Dacă îmbunătăţirile posibile ale vecinilor lui x_j sunt pozitive, atunci variabila primeşte valoarea *false.*

- *my_termination_counter*: se foloseşte pentru a detecta situaţia în care algoritmul se încheie. Dacă valoarea variabilei este d, fiecare agent a cărui distanţă de la x_i este aproape de d satisface toate constrângerile. Se consideră în algoritm că x_i cunoaşte distanţa maximă la alţi agenţi (*max_distance)*. Dacă valoarea variabilei *my_termination_counter* devine egală cu *max_distance*, x_i poate confirma că toţi agenţii satisfac constrângerile lor.

După recepţionarea mesajului *improve* de la toţi agenţii, x_i schimbă ponderile constrângerilor neîndeplinite dacă starea variabilei quasi_local_minimum este true. De îndată ce fiecare agent independent înregistrează ponderile, agenţii vecini nu mai au nevoie să negocieze despre dreptul de a mări ponderile. Dacă variabila *can_move* este true, valoarea variabilei x_i este schimbată, iar în celelelate cazuri ea rămâne neschimbată. Agentul trimite mesajul *ok?* şi intră în modul wait_ok?.

```
wait_ok? mode
when received (ok?, (xj,dj))  do
   counter ← counter +1
   adaugă(xj,dj) la lista agent_view;
   when   counter=numărul de neighbors  do
      send_improve
      counter ← 0
      goto wait_improve mode
   end do
   goto wait_ok mode
end do;

procedure  send_improve
   current_eval ← valoarea evaluată a lui current_value
   my_improve ← îmbunătăţirea maximă posibilă
   if  current_eval =0 then
       consistent ← true
    else
       consistent ← false
       my_termination_counter ← 0
   end if
   if   my_improve  >0  then
       can_move ← true
       quasi_local_minimum ← false
    else
       can_move ← false
       quasi_local_minimum ← true
   end if
  send(improve,xi,my_improve,current_eval,my_termination_counter)la neighbors
end procedure
```

Figura 3.5 Algoritmul distribuit breakout – modul wait_ok?

```
wait_improve? mode
when received (improve?, (xj,improve, eval, termation_counter))  do
   counter ← counter +1
   my_termination_counter ← min(termantion_counter, my_termantion_counter
   when   improve >my_improve  do
       can_move ← false
       quasi_local_minimum ← false
   end do
   when   improve =my_improve şi xj precede xi do
       can_move ← false
   end do
   when   eval > 0  do
       consistent ← false
   end do
   when   counter=numărul de neighbors  do
      send_ok
      counter ← 0
      clear  agent_view
      goto wait_ok mode
   end do
   goto wait_improve mode
end do;

procedure  send_ok
   when consistent=true    do
      my_termination_counter ← my_termination_counter +1
      when my_termination_counter =max_distance do
           Transmite vecinilor că o soluţie  s-a găsit
           Stop algoritm
      end do
   end do
   when  quasi_local_minimum = true  do
       măreşte  ponderile constrângerilor neîndeplinite
   end do
   when   can_move =true  do
       curent_value ← new_value
   end do
   send (ok?,  xi, curent_value) la neighbors
end procedure
```

Fig. 3.6 Algoritmul distribuit breakout - modul wait_improve

În [YDIK98], [Yok00], [Yok01] este testată eficienţa acestui algoritm, comparativ cu algoritmul AWCS. Experimentele arată că algoritmul este mai performant decât AWCS pentru anumite clase de probleme („critical problems"), dar este depăşit de AWCS în cazul altor probleme („sparse problems" şi "dense problems").

În ceea ce priveşte completitudinea algoritmului, această tehnică eficientă nu o poate garanta. Din păcate, completitudinea acestui algoritm nu poate fi

garantată odată ce el poate intra într-un ciclu infinit, spre deosebire de AWCS a cărui completitudine este garantată.

În concluzie, acest algoritm incomplet în care agenţii încearcă să minimizeze numărul de constrângeri neîndeplinite prin schimbarea valorilor curente asociate şi a valorii de îmbunătăţire între agenţii vecini, s-a dovedit mult mai performant decât algoritmul anterior doar pentru o clasă particulară de instanţe de probleme.

3.2.6 Tehnica "Distributed Backtracking" - DIBT

O altă soluţie de backtracking distribuit a fost publicată în [HBQ98]. Varianta de backtracking distribuit (DIBT-Distributed Backtracking) are ca bază de plecare algoritmul clasic de backtracking, cazul centralizat. Dacă algoritmul asincron foloseşte scheme de învăţare, acest algoritm elimină schemele de învăţare, cum ar fi înregistrarea listei "nogood". De asemenea, acest algoritm foloseşte câteva tehnici de îmbunătăţire a strategiei backtracking în contextul distribuit. De exemplu, informaţiile de instanţiere sunt transmise între agenţii conectaţi astfel încât să se beneficieze de topologia reţelei de constrângeri. Plecând de la studiile CSP care arată că mărimea spaţiului de căutare depinde în mare măsură de euristica ce stabileşte ordinea variabilelor, autorii introduc o metodă generică de determinare a ordinii pentru variabilele statice.

Această tehnică foloseşte (ca şi celelalte variante de tehnici de căutare asincrone) o ordine între agenţii, pentru a se asigura completitudinea algoritmului. Practic, se caută o ordine parţială între agenţi, care va fi folosită la instanţierea variabilelor şi care va fi extinsă la o ordine totală cu scopul de a ghida paşii backtrackingului. Din faptul că nu există restricţii în ce priveşte ordinea ce va fi utilizată, se va putea decide asupra folosirii unei ordini ce se potriveşte cel mai bine cu topologia grafului de constrângeri. Acest lucru are ca efect reducerea a spaţiului de căutare şi a numărului de mesaje pasate.

Trebuie amintit, varianta backtracking asincron ABT foloseşte ordinea lexicografică pentru agenţi pentru a elimina posibilitatea de intrare într-un ciclu infinit. Această ordine lexicografică are dezavantajul de a nu folosi caracteristicile iniţiale ale problemei. Prin urmare, în [HBQ98] este definită o metodă de determinare a ordinii variabilelor, metoda ce este folosită împreună cu noul algoritm.

Fiecare agent determină local poziţia sa conform cu euristica aleasă. Concret, fiecare agent determină mulţimile Γ^+ şi Γ^-, numite mulţimea agenţilor copii, respectiv părinţi, cunoscuţi de agentul respectiv, folosind funcţia de evaluare *f* şi operatorul de comparare *op* care definesc total euristica aleasă.. Acest lucru este realizat în liniile 1 şi 2 ale algoritmului din figura 3.7, algoritm din [HBQ98], [Ham99]. Se observă faptul că evaluarea funcţiei f poate implica anumite comunicaţii între agenţi, de aceea se recomandă folosirea unor euristici pentru care funcţia asociată f necesită numai comunicaţii locale între agenţii vecini, cum ar fi euristica *max-degree*.

	% se execută pentru fiecare agent (self) **D.I**: f definit pe Γ şi *op* operatorul de comparaţie **D.O**: cele două mulţimi Γ^+ şi Γ^-, obţinuta din mulţimea Γ, Γ^- ordonată. **begin**
1	$\Gamma^+ \leftarrow \varnothing$; $\Gamma^- \leftarrow \varnothing$; for fiecare $Agent_j \in \Gamma$ do

```
     if f(Agent_j) op f(self) then  Γ+ ←Γ+ ∪{Agent_j}
2    else  Γ- ←Γ- ∪{Agent_j}
3  % se ordonează mulţimea Γ-
   max ←0
   for (i=0; i< |Γ+| ; i++ ) do
     m←getMsg()
     if (m=value:v;from:j) then
       if (max<v) then max ← v;
   max++;
   sendMsg(Γ-, "value:max;from:self");
   sendMsg(Γ+, "position:max;from:self");
4  for(i=0;i<|Γ-|;i++) do
     m←getMsg();
     if(m=position:p;from:j) then Level[j] ←p;
   Rearrange Γ- după Level[];
end
```

Figura 3.7 Algoritmul de determinare a ordinii variabilelor

După executarea acestor etape, agenţii vor cunoaşte mulţimile Γ^- şi Γ^+. Aceste informaţii sunt utilizate în timpul căutării, agenţii vor trimite valorile instanţiate la copii şi în cazul apariţiei unui eşec, ei vor reveni la primul agent din Γ^-. Din acest motiv, partea a doua a algoritmului (liniile 3–4) se ocupă cu ordonarea mulţimii Γ^-.

În figura 3.8 este ilustrat procesul distribuit pentru un exemplu, aplicând euristica max-degree pentru ordinea variabilelor. În partea stângă a figurii apare graful constrângerilor. Pentru obţinerea euristicii max-degree, algoritmul din figura 3.8 trebuie să fie apelat pentru fiecare agent cu funcţia $f(Agent_i) = |\Gamma_i|$, unde Γ_i este mulţimea de cunoştinţe a lui $Agent_i$, şi comparatorul < . Ordinea statică a variabilelor este obţinută în partea dreaptă a figurii 3.8. Săgeţile arată ordinea relaţiilor, care reprezintă ordinea de transmitere a instanţelor în procesul de căutare. Ordinea mulţimilor Γ^-, care va fi utilizată după fiecare eşec, este următoarea:

- $Agent_1$, $\Gamma^- = \varnothing$.
- $Agent_2$, $Agent_5$, $\Gamma^- = \{Agent_1\}$.
- $Agent_3$, $Agent_4$, $\Gamma^- = \{Agent_2, Agent_1\}$.
- $Agent_2$, $\Gamma^- = \{ Agent_5, Agent_1\}$.

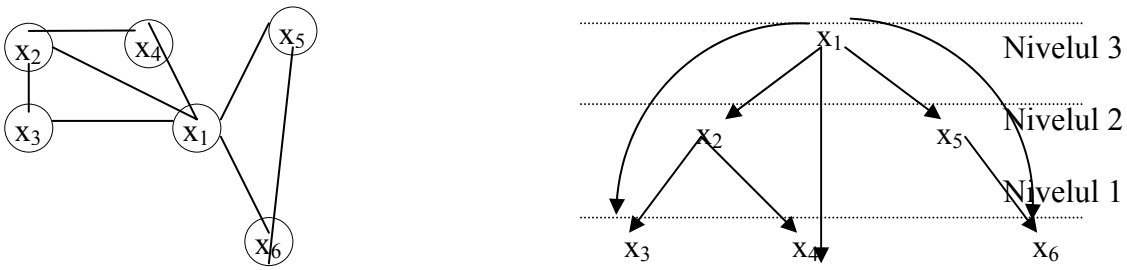


Figura 3.8. Ordinea variabilelor distribuite

Câteva observaţii se pot face referitor la această nouă tehnică relativă la ordonarea agenţilor. Dacă funcţia de evaluare este bine aleasă, fiecare agent poate

furniza o ordine locală algoritmului, cu un număr constant de mesaje cu fiecare din aceste cunoştinţe. Ca un rezultat al ordinii, agenţii de pe acelaşi nivel sunt independenţi. Ei nu împart nici o constrângere astfel încât ei pot să execute calcule în paralel. Acest lucru este foarte important pentru că conduce la un bun paralelism.

Algoritmul din [HBQ98] este prezentat în figura 3.9. Fiecare agent instanţiază variabilele sale cu respectarea constrângerilor părinte. După instanţiere, agenţii informează copiii lor de valorile alese (conţinutul mesajului startând prin *"infoVal"*). Dacă nici o valoare nu satisface constrângerile cu agenţii din Γ^-, un mesaj backtrack se trimite la părinţii apropiaţi (mesajul startând prin *„btSet"*). Din faptul că apare backtrack între variabilele conectate, acest algoritm este un backtracking bazat pe graful constrângerilor. La recepţionarea unui mesaj de backtrack, în primul rând se caută validarea mesajului prin compararea valorii curente cu cea raportată pentru acest mesaj (linia 6). În cazul în care există o valoare diferită, aceasta înseamnă că transmiţătorul încă nu a recepţionat informaţii despre ultimele valori recepţionate, aşa că backtrack-ul decide că poate fi învechit. Dacă comparaţia se potriveşte şi dacă agentul nu poate lua o valoare diferită, apare un backtrack. Este trimis un mesaj de backtrack la agentul cel mai apropiat în ordinea mulţimii lui Γ^- şi se transmite mulţimea transmiţătorului (linia 9).

Încheierea algoritmului pentru un agent apare când agentul este instanţiat şi intră în starea de aşteptare sau când agentul sursă găseşte un mesaj nogood gol (linia 8). În ultimul caz, este rezolvată situaţia în care problema nu are soluţie. Mesajul *noSolution* este trimis la agentul sistem. Acest agent opreşte calculul distribuit prin transmiterea unui mesaj de oprire la toţii agenţii.

Algoritmul din [HBQ98] este prezentat în figura 3.9.

```
Calculează Γ+, Γ-, ordonează Γ- folosind algoritmul din fig 3.8
nearest ←first(Γ- );
myValue ←getvalue(info);
sendMsg(Γ+, "infoVal: myValue; from self");
end ←false;
while (not end) do
    m←getMsg();
    if (m=stop) then
       end←true;
    if (m=infoVal:a; from j) then
        Value[j] ← a;
        myValue ←getValue(info) ;
        if (myValue ) then
              sendMsg(Γ+, "infoVal: myValue; from: self");
        else
              sendMsg(nearest, "btSet: Γ-, values: value[Γ-]");
    if (m=btSet:set; Values:values) then
         if (values[j]=myValue)
              myValue ←getValue(bt) ;
              if (myValue) then
                    sendMsg(Γ+, "infoVal: myValue; from self");
              else
                    if (Γ- =∅ and set=∅)  then
                          sendMsg(system, „NoSolution");
                          end ← true;
```

```
                    else
                            followSet ← merge (Γ-, set);
                            follow ← first (followSet) ;
                            sendMsg(follow,"btSet:followSet values: value[Γ-]∪
                                    ∪values");
                    if (follow ∉ Γ- )
                            myValue ← getValue(info);
end
```

Figura 3.9. Algoritmul backtracking distribuit - DIBT

În algoritmul prezentat în figura 3.9 sunt folosite următoarele structuri şi rutine:

- *self* este un agent generic ce execută algoritmul, D_{self} este domeniul său şi *myvalue* este valoarea sa curentă;
- *value[]* stochează parinţii cunoscând valorile;
- *first(S)* returnează primul element al mulţimii ordonate S;
- *getValue(type)*:
  - dacă type="info", metoda returnează prima valoare din D_{self} compatibilă cu agenţii din Γ^-, începând cu *myValue;*
  - dacă type="bt", metoda returnează prima valoare după myValue din D_{self} compatibilă cu agenţii din Γ^- ;

În ce priveşte completitudinea algoritmului, autorii propun o schiţă de demonstraţie în [HBQ98]. Mai târziu, în [Ham99], pentru a asigura completitudinea algoritmului, autorul extinde mulţimile Γ^+ şi Γ^-, numite mulţimea copiilor şi părinţilor. Din păcate, în [Yok00p], Yokoo arată că acest algoritm este incomplet (cu toate aceste extensii). Totuşi, de curând, s-a propus un patch la algoritm care să asigure completitudinea algoritmului, patch prezentat împreună cu familia ABT în paragraful 3.4.

3.2.8 Tehnica "Distributed Dynamic Backtracking" - DisDB

Un alt algoritm distribuit, ce funcţionează pentru modelul DCSP, în aceleaşi condiţii ca şi algoritmii prezentaţi anterior, a fost publicat în [BMM01] şi [BM05]. Acest nou algoritm asincron foloseşte ce este mai bun de la algoritmii de căutare anteriori ABT şi DIBT, încercând să funcţioneze cât mai asincron posibil şi cu minimul de mesaje pasate între agenţi. El se bazează pe algoritmul de backtracking distribuit al lui Ginsberg [Gin93], existent pentru arhitecturi centralizate (algoritmul de backtracking dinamic al lui Ginsberg este un algoritm de tip backjumping).

Tehnica DisDB are la bază salturi dinamice peste mulţimile de agenţi aflaţi în conflict. Constrângerile sunt orientate formând un graf aciclic orientat, care este un punct de plecare în stabilirea ordinii agenţilor (ierarhia agenţilor poate fi construită utilizând euristici cunoscute şi aplicând schema de ierarhizare propusă de Hamadi în [HBQ98]**,** schemă prezentată şi aici, în paragraful 3.2.6.1. Dacă nu se foloseşte nici o euristică, ierarhia se stabileşte implicit folosind ordinea lexicografică dintre agenţi.

Algoritmul se bazează pe folosirea a două mulţimi, notate cu Γ^+ şi Γ^-, mulţimi folosite şi în cazul algoritmului de backtracking distribuit, prezentat în paragraful anterior (dar cu o altă semnificaţie). Considerând un agent generic, notat

cu *self* în algoritm, Γ^-(self) este mulţimea agenţilor care au constrângeri cu *self* şi apar la nivele superioare în ierarhie. În schimb, Γ^+(self) este mulţimea agenţilor care au constrângeri cu *self* ce apar la nivele inferioare în ierarhie. Această ierarhie induce o ordine parţială între agenţi, care trebuie completată la o formă de ordine totală, pentru a asigura completitudinea algoritmului.

În algoritmul DisDB fiecare agent păstrează un context de lucru format din vederea agentului şi lista de valori *nogood.* Vederea agentului self este o mulţime de valori ce se crede a fi asociate la agenţii care sunt înaintea lui self conform ordinii totale definite.Vederea agentului este întotdeauna compatibilă cu listele nogood aflate în stoc. Agenţii schimbă atribuirile şi listele nogood. DisDB acceptă întotdeauna noi atribuiri, actualizând vederea agentului în cauză. Când se recepţionează un nogood, el este acceptat dacă este consistent cu vederea agentului din Γ^-(self) $\cup$ {self}, în caz contrar renunţându-se la el ca fiind învechit. Un nogood acceptat este utilizat la actualizarea vederii agent dintre agenţii care nu sunt în Γ^-(self) şi la actualizarea mulţimii de liste nogood stocate. Când toate valorile agenţilor sunt înlăturate datorită câtorva liste nogood, mulţimea de liste nogood stocate este rezolvată ca în cazul centralizat, generând un nou nogood ce este trimis la variabila din partea dreaptă. Aceste variabile, plus toate variabilele aflate în partea stângă a noilor nogood ce nu sunt în Γ^-(self), sunt neasociate în vederea agent, iar listele nogood sunt actualizate în mod corespunzător. Acest proces se termină când se obţine o perioadă de „linişte" în trasmiterea de mesaje (fie o soluţie a fost găsită sau o listă nogood goală este generată, înţelegând prin aceasta că problema este nerezolvabilă).

În ceea ce priveşte mesajele, în algoritm sunt utilizate trei tipuri de mesaje:

- *Stop(system)*. Nu există soluţie şi agentul recepţionează stop. Este folosit un extra agent, numit agent system, care se ocupă cu transmiterea mesajului stop la întreaga reţea.
- *Info(child, valoare)*. Este informat child că agentul *self* a luat acea valoare. El este trimis la agenţii din mulţimea Γ^+(self).
- *Back(nogood)***.** Este un mesaj de backtracking, conţinând nogood şi adresat la agenţii din partea dreaptă.

Algoritmul DisDB este descris în figura 3.10 [BMM01]. Mesajele sunt schimbate folosind două primitive getMsg şi sendMsg. Procedura principală DisDB recepţionează repetat mesaje ce sunt tratate în funcţie de tipul lor. În momentul recepţionării mesajului Stop (de la agentul sistem) algoritmul se încheie.

În cazul în care se primeşte un mesaj info, procedura *GoAhead* este executată. Aceasta actualizează vederea agentului curent (în algoritm este notată prin myContext). De asemenea, dacă noile informaţii dezactivează valoarea curentă, o nouă valoare este încercată. Dacă astfel valori există, Γ^+(self) este informat de noile valori ale lui *self* (liniile 5,6 din procedură). Altfel, se apelează o altă procedură numită *ResolveNogoods.*

În cazul în care se primeşte un mesaj Back, se apelează procedura *ResolveConflict.* Mesajul Back (linia 1 din procedură) este acceptat, dacă el este consistent cu valoarea lui *self* şi dacă vederea agentului pentru agenţii pentru care *self* are o legătură directă, să cunoască valorile lor printr-un mesaj Info. Altfel, mesajul Back este eliminat ca fiind învechit. După acceptarea unui mesaj Back, se actualizează vederea agent pentru acei agenţi care nu sunt în Γ^-(self) (linia 2), se stochează în *stocul de nogood* (linia 3) şi o nouă valoare pentru self este încercată (linia 4). Dacă astfel de valori există, Γ^+(self) este informat de noile valori ale lui self

(liniile 6,7). În caz contrar, (adică toate valorile lui self sunt eliminate) se apelează procedura *ResolveNogoods* (linia 8).

Procedura *ResolveNogoods* rezolvă toate listele nogood stocate în *myNogoods*, generând un nou nogood numit în algoritm *newNogood* (linia 1 din procedură). Variabilele ce apar în partea dreaptă a lui *newNogood* sunt variabile din *myNogoods* apropiate de *self* în ordinea totală dintre agenţi. Dacă un nogood gol este generat, problema nu are soluţie şi mesajul Stop este trimis (vezi liniile 3 şi 4). În caz contrar, *newNogood* este trimis. Apoi, variabilele din partea dreapta a lui *newNogood* sunt neasociate şi lista nogood suportată de ele este uitată (liniile 8, 9).

Procedura *ChoseValue,* selectează valorile consistente cu vederea agentului, actualizând stocul de nogood, iar procedura *Update* actualizează vederea agentului şi păstrează coerenţa dintre vederea agentului şi mulţimea de liste nogood stocate (procedurile lhs şi rhs sunt proceduri auxiliare ce au ca sarcină determinarea părţii din stânga, respectiv dreapta, a listei nogood).

```
  procedure  DisDB()
1 compute Γ+(self),Γ−(self);
2 GoAhead(null);
3 end ←false;
4 while(¬end) do
5   msg<-getMsg ();
6   switch(msg.type)
7      Stop    : end ←true;
8      Info    : GoAhead(msg);
9      Back    :ResolveConflict(msg);

procedure GoAhead(msg)
1 if (msg) then Update(myContext,msg.Context));
2 if (¬msg or ¬ consistent (myValue,myContext) then
3   myValue ←ChooseValue();
4   if(myValue) then
5     for each child ∈ Γ+(self ) do
6       sendMsg:Info(child,myValue);
7   else ResolveNogoods();

procedure ResolveConflict(msg)
1 if consistent (myContext, msg.Context in  Γ−(self) ∪ {self }) then
2   Update(myContext,msg.Context);
3   add(msg.Context => ¬myValue,myNogoods);
4   myValue <-ChooseValue();
5   if(myValue) then
6     for each child ∈ Γ+(self ) do
7       sendMsg:Info(child,myValue);
8   else ResolveNogoods();

procedure ResolveNogoods()
1 newNogood ←solve(myNogoods);
2 if(newNogood=empty)
3   end ←true;
4   sendMsg:Stop(system);
```

```
5 else
6     sendMsg:Back(newNogood);
7     Update(myContext,rhs(newNogood) ←unknown);
8     for each var ∈ lhs(newNogood) \ Γ⁻(self)
9       Update(myContext, var ←unknown);
10    Go Ahead(null);

function ChooseValue()
1 for  each v ∈ D0(self) not eliminated by myNogoods do
2     if consistent(v, myContext) then
3       return(v);
4     else                    /*X:var inconsistent with v*/
5       add(X=valX=> ¬v, myNogoods);
6     return(empty);

procedure Update(myContext, newContext)
1 include(newContext,myContext);
2 for each  ng  ∈ myNogoods
3   if(¬consistent(lhs(ng),newContext)) then
4           myNogoods ← myNogoods\{ng};
```

Figura 3.10. Algoritmul backtracking dinamic distribuit -DisDB

Pentru o mai bună înţelegere a funcţionării acestui algoritm, în figura 3.11 este prezentat un exemplu din [BMM01]. În acest exemplu se consideră patru variabile şi patru agenţi diferiţi, ordonaţi lexicografic, cu domeniile din paranteze şi trei constrângeri diferite conectând variabilele x_1, x_2 şi x_3 la x_4.

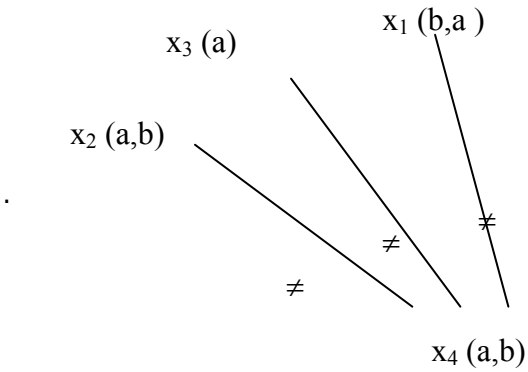


Figura 3.11. O problemă DCSP cu patru agenţi

Iniţial, cele patru variabile iau valori din domeniul lor şi primele trei informează variabila x_4 de valorile lor. Variabila x_4 recepţionează mesaje de la x_1 şi x_2, determină două liste nogoods $x_1=b \Rightarrow x_4 \neq b$, $x_2=a \Rightarrow x_4 \neq a$, sunt rezolvate şi trimise mesaje bactracking la x_2. Variabila x_4 uită listele nogoods incluzând x_2 şi ia valoarea a.

Mai departe, se recepţionează un mesaj de la x_3 care cauzează un nod nogood $x_3=b \Rightarrow x_4 \neq b$, care este rezolvat cu $x_1=b \Rightarrow x_4 \neq b$, producându-se un mesaj

backtracking la x_3. variabila x_4 uită lista de valori nogood incluzând x_3 şi ia valoarea a.

Variabila x_2 recepţionează nogood-ul său, dar ea nu mai are nici o valoare disponibilă. Se calculează un nou nogood $\Rightarrow$ $x_1 \neq b$, care este trimis lui x_1. Variabila x_2 uită lista de nogood-uri incluzând x_1 şi ia valoarea a, informând variabila x_4. Variabila x_3 recepţionează nogood-ul său şi schimbă valoarea sa cu b, informând x_4. Variabila x_1 recepţionează nogood-ul său şi schimbă valoarea sa cu a, informând x_4.

Variabila x_4, recepţionează mesaje de la x_1, x_2,x_3 (în această ordine) cu valorile noi ale lor. După recepţionarea noii valori a pentru x_1, x_4 uită lista nogood anterioară ($x_1=b \Rightarrow x_4 \neq b$) şi se generează o nouă listă nogood $x_1=a \Rightarrow x_4 \neq a$. Prin urmare x_4 ia valoarea b. Nici un nogood nu este generat de la mesajul lui x_2 . Mesajul lui x_2 generează un nogood $x_3=b \Rightarrow x_4 \neq b$, care este rezolvat ca într-unul din cazurile de mai sus, cauzând un mesaj backtracking la x_3. Variabila x_4 uită lista nogood, incluzând x_3 şi primeşte încă o dată valoarea b.

Variabila x_3 recepţionează mesajul său şi elimină lista nogood anterioară $x_1=b \Rightarrow x_3 \neq a$ deoarece este învechită: x_1 nu este în $\Gamma^-(x_3)$ şi nogood-ul recepţionat de la x_1 are o valoare diferită. Variabila x_3 ia valoarea a şi informează x_4, care nu face nimic din cauză că noua valoare este compatibilă cu contextul său şi stocată în lista nogood. Execuţia se termină cu găsirea unei soluţii.

În [BMM01] se demonstrează că algoritmul DisDB este complet şi se termină. Ideea de construire a unui algoritm asincron de căutare cu un spaţiu polinomial de învăţare dar fără legături adiţionale, reprezintă o soluţie naturală ce este aplicată la acest algoritm, care prin aceste elemente se deosebeşte de ceilalţi algoritmi distribuiţi prezentaţi până acum.

3.3. Tehnicile din familia ABT.

Având ca bază de plecare algoritmul asynchronous backtracking (ABT), în [BM05] a fost propus un cadru unificator, un nucleu de plecare pentru câteva dintre cele mai importante şi cunoscute tehnici asincrone. Din acest nucleu au fost derivate mai multe tehnici, tehnici cunoscute sub numele de familia ABT. Ele diferă prin modul în care stochează valorile nogood, dar şi prin modul în care adaugă legături între agenţii neconectaţi cu scopul de a detecta şi elimina informaţiile învechite dintre agenţi. În [BM05] se pleacă de la o procedură simplă (numită ABT kernel) care cuprinde principalele caracteristici ale tehnicilor de căutare asincrone. Plecând de la această procedură, care formează cadrul unificator, se poate ajunge la tehnici cunoscute sau la variante apropiate de aceastea: asynchronous backtracking (ABT), Distributed Dynamic Backtracking (DisDB) sau tehnica Distributed Backtracking (DIBT).

Algoritmul ABT kernel necesită, ca şi tehnica ABT, direcţionarea constrângerilor de la agentul care transmite o valoare la agentul care evaluează constrângerile, formând un graf orientat aciclic. În această teză se consideră că agenţii sunt ordonaţi static, după anumite criterii (în mod implicit, dacă nu se precizează, se consideră ordinea lexicografică). Prin urmare, un agent având o prioritate mai mare decât altul, va fi considerat la un nivel superior într-un arbore ce reprezintă ierarhia cu agenţi. Se vor considera mai multe notaţii consacrate: dacă *self* este un agent generic, atunci Γ^-(self) este mulţimea agenţilor care au constrângeri cu *self* şi apar la nivele superioare în ierarhie (numită mulţimea părinţilor) iar cu Γ^+(self) este mulţimea agenţilor care au constrângeri cu *self* ce apar la nivele inferioare în ierarhie (numită mulţimea copiilor) [BM05].

În figura 3.12 sunt prezentate rutinele de tratate a mesajelor pentru algoritmul ABT kernel [BM05].

Fiecare agent are nevoie să înregistreze anumite informaţii legate de starea globală a căutării (toate tehnicile asincrone au nevoie de aceste informaţii). Această stare presupune cunoaşterea valorilor asociate agenţilor de pe nivelele superioare (numite de obicei *agent_view*), ele formând *contextul* de lucru local al agenţilor şi a listelor de valori inconsistente (numite *nogood*). Agenţii fac schimb de aceste valori alocate de agenţi şi de nogood-uri. Căutarea se efectuează printr-o buclă simplă până la găsirea unei soluţii sau detectarea unui mesaj de oprire. Acest mecanism se întâlneşte la majoritatea tehnicilor asincrone.

Bucla principală, ce se aplică la fiecare agent, presupune parcurgerea a patru etape. Prima constă în recepţionarea unui mesaj şi încheierea execuţiei dacă s-a primit mesajul *Stop*. A doua etapă se referă la situaţia în care mesajul este pertinent, caz în care se actualizează contextul de lucru. În a treia etapă se selectează o soluţie locală. În ultima etapă, se genereză un nou mesaj care informează „copiii" lui *self* în legătură cu noua sa valoare sau se opreşte procesul de căutare globală (în caz de eşec).

Nucleul de bază (prezentat în figura 3.12) are la bază procedura ABT_{kernel}, procedură executată de fiecare agent şi conţinând cele patru etape prezentate anterior [BM05].

În figura 3.13 este prezentat şi fluxul de mesaje schimbat de agenţi, punându-se în evidenţă cele două mesaje principale utilizate *„info"* şi *„back"*. Cele două mesaje sunt folosite pentru informarea agenţilor „copiii" de schimbarea unei valori, respectiv de apariţia unei situaţii de eşec. Mesajele de tip *info* corespund mesajelor de tip ok prezentate în paragraful 3.3.3, iar cele de tip back mesajelor de tip nogood întâlnite la tehnica ABT sau AWCS..

```
Procedure ABTkernel()
1  myValue←empty; end ← false;
2     CheckAgentView();
3     while (not end) do
4       msg ← getMsg();
5       switch(msg:type)
6             Info : ProcessInfo(msg);
7             Back : ResolveConflict(msg);
8             Stop : end← true;
End

procedure CheckAgentView(msg)
1     if not consistent(myValue;myAgentView) then
2             myValue ← ChooseValue();
3             if (myValue) then for each child ∈ Γ+ (self) do
                 sendMsg:Info(child;myValue);
4             else Backtrack();
End

procedure ProcessInfo(msg)
1     Update(myAgentView; msg.Assig);
2     CheckAgentView();
End
```

```
procedure ResolveConflict(msg)
1     if Coherent(msg.Nogood; Γ⁻ (self) ∪ {self}) then
2             for each assig ∈ lhs(msg.Nogood) \Γ⁻ (self) do
                    Update(myAgentView; assig);
3             add(msg:Nogood;myNogoodStore); myValue ← empty;
4             CheckAgentView();
5     else if msg.sender ∈Γ⁺ ( self) ∧ Coherent(msg.Nogood; self) then
            SendMsg:Info(msg.sender;myValue);
End

procedure Backtrack()
1     newNogood ← solve(myNogoodStore);
2     if (newNogood = empty) then
3             end ← true; sendMsg:Stop(system);
4     else
5             sendMsg:Back(newNogood);
6             Update(myAgentView;rhs(newNogood)← unknown);
7             CheckAgentView();
End

function ChooseValue()
1     for each v ∈ D(self) not eliminated by myNogoodStore do
2             if consistent(v; myAgentView) then return (v);
3             else add(x_j = val_j ) self ≠ v;myNogoodStore);
4     return (empty);
End

procedure Update(myAgentView; newAssig)
1     add(newAssig;myAgentV iew);
2     for each ng ∈ myNogoodStore do
3             if not Coherent(lhs(ng);myAgentView) then
remove(ng;myNogoodStore);
End

function Coherent(nogood; agents)
1     for each var ∈ nogood ∪ agents do
2             if nogood[var] ≠ myAgentV iew[var] then return false;
3 return true;
end
```

Figura 3.12. Tehnica ABT kernel pentru căutarea asincronă

Procedura *CheckAgentView* are rolul de a verifica dacă mesajele sunt relevante, aceasta însemnând că se acceptă doar acele mesaje care se referă la o nouă valoare (mesaje de tip info) sau mesaje de tip *back* care apar în situaţia negăsirii de către un agent a unei valori consistente (practic orice valoare este inconsistentă cu *contextul* local).

Mesajele acceptate de către procedura principală ABT_{kernel} sunt folosite la actualizarea unei baze de cunoştinţe locale numită contextul local al agentului (în figura 3.16 apare sub numele myAgentView). Acest lucru presupune actualizarea valorilor din context şi asigurarea coerenţei mulţimii de valori nogood cu această

nouă valoare. Dacă valoarea curentă este validă, ciclul se încheie şi agentul *self* poate primi un nou mesaj. În caz contrar, agentul caută o nouă valoare care să fie consistentă cu datele actuale (linia 4 din procedura CheckAgentView) cu ajutorul funcţiei ChooseValue. Acesta alege o valoare din domeniul lui self care nu a fost eliminată de valorile din *nogood* şi verifică dacă este consistentă, în caz afirmativ returnând aceea valoare. În caz contrar, se elimină aceea valoare printr-un nou *nogood* (liniile 3-4 din ChooseValue). Se observă faptul că valorile *nogood* se referă doar la variabilele din Γ^-(self). Funcţia ChooseValue are două posibilităţi, să returneze o valoare compatibilă, caz în care agentul self trimite un mesaj de informare a copiilor săi (liniile 5-7 din procedura CheckAgentView) sau să ajungă la un domeniu gol pentru agentul self, caz în care va semnala eşecul şi va apela procedura Backtrack(). Această procedură mai întâi rezolvă valorile nogood, apoi actualizează datele locale şi continuă căutarea unei valori compatibile. Acest comportament se remarcă şi în figura 3.13 unde este prezentat fluxul de mesaje.

Trebuie remarcat faptul că în situaţia apariţiei unui eşec fără posibilitate de revenire şi de căutare a altor valori, incoerenţa este dovedită şi căutarea se poate opri (linia 3 din procedura Backtrack). Oprirea se face prin trimiterea mesajului *Stop(System)*, orice agent putând să trimită acest mesaj.

În nucleul din figura 3.12 se remarcă procedura Backtrack(), pentru că ea tratează problemele legate de situaţia de eşec. Conform principiilor de căutare asincronă această procedură este apelată în momentul în care un domeniu devine gol pentru un agent, agentul practic nu poate să facă o asociere pentru variabila sa. În primul rând procedura construieşte un nou nogood pe baza informaţiilor din contextul său. Dacă acest nogood este vid, algoritmul se încheie pentru că nu există soluţie. În schimb, dacă noul nogood nu este vid se identifică agentul cu cea mai mică prioritate dintre agenţii aflaţi în conflict. Noul *nogood* obţinut este trimis la acel agent cu prioritatea cea mai mică (A_{inf}) pentru a-i elimina valoarea curentă. Agentul care a expediat mesajul *nogood* schimbă valoarea curentă a agentului A_{inf} şi actualizează valorile nogood.

Algoritmul ABT kernel este la bază un algoritm ABT, respectând principiile căutării asincrone. Acest lucru se observă şi din analiza fluxului de mesaje prezentat în figura 3.13. El se remarcă prin faptul că nu necesită adăugarea de noi legături între agenţii neconectaţi (spre deosebire de ABT care necesită, în timpul rulării, adăugarea de noi legături). Nucleul ABT este corect dar există posibilitatea de a intra în blocaj. Acest lucru se datorează faptului că anumiţi agenţi pot stoca valori nogood ce conţin informaţii învechite despre agenţii neconectaţi.

În [BM05] au fost propuse mai multe soluţii de eliminare a informaţilor învechite dintre agenţi, soluţii prezentate în continuare.

O prima soluţie prezentată în [BM05] de eliminare a informaţiilor învechite este de a adăuga noi legături pentru a permite agentului ce a recepţionat un mesaj nogood să decidă dacă este sau nu învechit. Aceste legături au fost propuse şi în algoritmul ABT varianta Yokoo [YDIK98], [Yok00], [Yok01].

O a doua soluţie din [BM05] constă în detectarea situaţiei în care un anumit nogood este sau nu învechit. Practic, un nogood ipotetic învechit şi toate valorile recepţionate de la agenţii cu care nu a fost conectat sunt uitate. Aceste două variante permit obţinerea a patru tehnici derivate din nucleul ABT kernel :

- *Adăugarea de noi legături înainte de căutare, în faza de preprocesare.* Varianta aceasta este numită ABT_{all}. Algoritmul adaugă noi legături (care vor deveni permanente) în faza de preprocesare. Această tehnică este apropiată de varianta corectă DIBT.

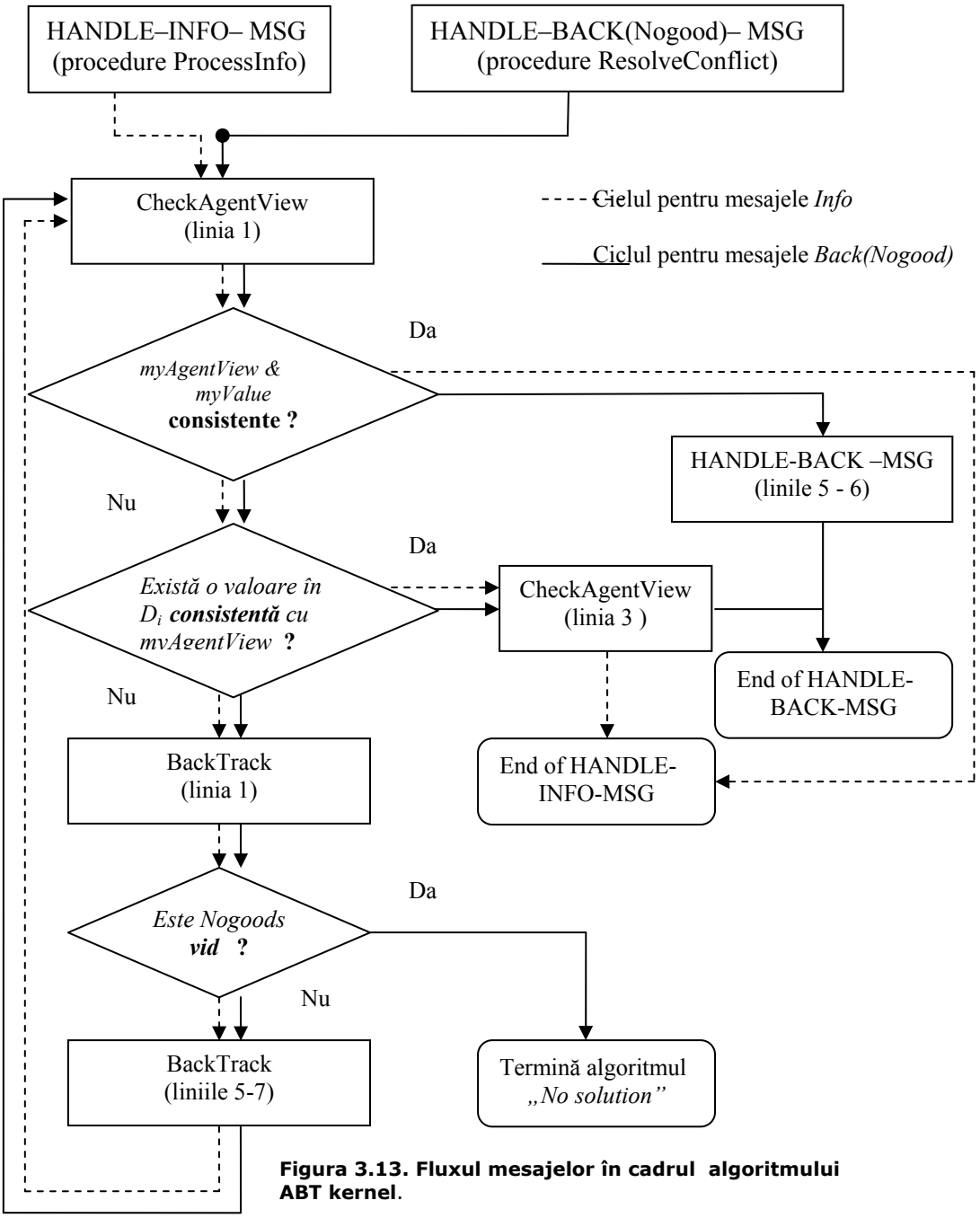


Figura 3.13. Fluxul mesajelor în cadrul algoritmului ABT kernel.

- *Adăugarea de legături în timpul căutării* prin care se obţine varianta de bază *ABT (Yokoo)***.** Legăturile se adaugă în timpul căutării în momentul

recepţionării unui mesaj nogood (Back) care conţine informaţii relative la un agent neconectat, conform ordinii dintre agenţi. Noile legături sunt permanente. .

- *Adăugarea de legături temporare în timpul căutării* prin care se obţine varianta ABT_{temp}**.** Legăturile se adaugă ca şi la varianta ABT Yokoo, în timpul căutării în momentul recepţionării unui mesaj nogood (Back) care conţine informaţii relative la un agent neconectat. Diferenţa este că aceste legături sunt temporare, ele rămânând până când un număr de mesaje între acei agenţi sunt schimbate. Apoi, ele sunt anulate.
- *Eliminarea legăturilor* prin care se obţine varianta ABT_{not}, variantă cunoscută sub numele de Distributed Dynamic Backtracking (DIBT). Această variantă nu necesită legături suplimentare. Practic, informaţiile învechite sunt eliminate în timp finit. Pentru aceasta agenţii care execută backtrack uită toate nogoodurile ce pot conţine ipotetic valori învechite.

3.4. Concluzii.

Cadrul CSP este un cadru de lucru puternic pentru rezolvarea problemelor, în general. Foarte multe probleme se pot modela folosind cadrul CSP şi în mod automat folosind şi DCSP. Această modelare se aplică astăzi în aplicaţii ce variază de la proiectarea maşinilor la alocarea resurselor (în [Wall95] se găsesc descrise variate probleme ce se pot modela şi rezolva în cadrul CSP şi DCSP).

Cercetările din aceşti ultimi zece ani s-au concentrat, în primul rând, pe găsirea de metode de rezolvare pentru cadrul DCSP, plecând de la metodele similare existente în cadrul CSP. Foarte multe metode, bazate pe căutarea sistematică (backtracking şi variante ale sale) au fost adaptate la cadrul DCSP. Adaptările au dus la algoritmii de o complexitate mult mai mare, algoritmii mult mai complicaţi decât cei similari CSP, datorită faptului că ei trebuie să se folosească de cadrul distribuit şi concurent, pentru a se obţine un câştig în ce priveşte eficienţa. Adaptările s-au bazat pe idei diferite, de exemplu de înregistrare a listei de valori care au condus la un eşec (varianta Yokoo) sau salturi în graful constrângerilor (varianta Hamadi), obţinându-se mai multe variante de căutare. Drept exemplu stau cele trei metode de căutare asincronă, backtracking asincron (varianta Yokoo), căutarea asincronă cu agregate (Silaghi) sau backtracking distribuit (varianta Hamadi). S-a încercat îmbunătăţirea acestor algoritmi, cum ar fi, de exemplu, varianta de asynchronous weak-commitment search, ce reprezintă o varianta mult mai bună de căutare asincronă.

Cea mai mare parte din aceşti algoritmi au fost demonstraţi ca fiind algoritmi compleţi, ce pot da un răspuns (afirmativ sau negativ) în ce priveşte existenţa soluţiei. De asemenea, ei au fost evaluaţi, unii comparativ, folosind diverse metode de simulare şi mai puţin în medii de programare reale.

În acest capitol au fost prezentate principalele noţiuni teoretice care stau la baza studiilor din capitolele următoare [BM05], [YDIK98], [Yok00], [Yok01] . A fost prezentat modelul DCSP, urmat de principalele tehnici asincrone pentru care sunt propuse îmbunătăţiri: tehnica „asynchronous backtracking”, tehnicile din familia ABT şi tehnica „asynchronous weak-commitment search”. Pentru aceste tehnici au fost prezentate principalele rutine de tratare a mesajelor şi comportamentul agenţilor în diverse situaţii. Majoritatea acestor tehnici se remarcă prin utilizarea mesajelor de tip nogood, ce anunţă apariţia unei inconsistenţe. Plecând de la aceste informaţii, în

capitolele următoare se va încerca prezentarea unui model de implementare pentru aceste tehnici şi identificarea unor posibile îmbunătăţiri.

Comportamentul agenţilor în cazul recepţionării mesajelor de tip nogood va fi analizat în capitolele următoare pentru a identifica modalităţi de îmbunătăţire a performanţelor. Sunt studiate legăturile tempoare apărute cu scopul limitării fluxului de mesaje ce apare pe acestea. Apoi, se propun metode de înlocuire a stocării acestor valori nogood cu flaguri. Aceste metode permit în capitolele următoare îmbunătăţirea performanţelor tehnicilor asincrone derivate din ABT.

4. PROBLEME ACTUALE ÎN ÎMBUNĂTĂŢIREA PERFORMANŢELOR TEHNICILOR DE CĂUTARE ASINCRONE

Complexitatea tehnicilor asincrone este o temă de studiu foarte importantă pentru cei care realizează cercetări în acest domeniu. Datorită comunicaţiilor asincrone, modelul distribuit este unul non-determinist. Un program distribuit poate avea rulări diferite pentru acelaşi set de date, în funcţie de programarea evenimentelor. Prin urmare, trebuie analizate foarte multe încercări, trebuie verificat cazul cel mai nefavorabil peste toate aceste posibile scheme.

Evaluarea performanţelor acestor tehnici necesită eforturi mari de programare şi implementare. Studiul comportamentelor acestor tehnici, în diverse situaţii, implică multe implementări împreună cu foarte multe rulări. Implementările obişnuite în Java, prin RMI sau Corba, implică eforturi mari şi ineficienţă în studiul acestor tehnici.

În acest capitol sunt analizate condiţiile actuale în care se face implementarea şi evaluarea diverselor tehnici de căutare asincrone. Sunt identificare diverse defecte şi dezavantaje ale tehnicilor de căutare asincrone, plecând de la situaţiile reale ce pot apărea în practică.

Tehnicile de căutare asincrone în cazul aplicării la o problemă modelată DCSP, încearcă să execute o căutare cât mai asincronă astfel încât puterea de procesare a tuturor agenţilor să fie utilizată cât mai eficient. Ideal ar fi ca toţi agenţii să lucreze şi nici unul să nu fie în aşteptare. Din păcate, în cazul majorităţii tehnicilor (familia ABT sau AWCS) doar câţiva agenţi lucrează, cea mai mare parte fiind în aşteptare. Acest lucru este cauzat de mecanismul de lucru al acestor tehnici asincrone, mecanism prezentat în paragraful §3.3.2. Practic un agent îşi schimbă valorile variabilei sale doar când este forţat de alţi agenţi. Un agent, odată ce a găsit o asociere ce este consistentă cu agenţii de ordine superioară, nu mai face nimic şi aşteaptă ca agenţii de prioritate superioară să îşi schimbe valorile sau un nogood de la un agent de prioritate inferioară să fie recepţionat. Această problemă va fi studiată în capitolele următoare.

Acest studiu este baza de plecare pentru identificarea unor soluţii de îmbunătţire a performanţelor tehnicilor de căutare asincrone, îmbunătăţiri propuse în partea a II-a şi în partea a III- a a tezei de faţă.

4.1. Probleme de implementare şi evaluare pentru tehnicile de căutare asincrone.

În acest paragraf sunt identificate diverse probleme, avantaje şi dezavantaje referitoare la soluţiile de implementare şi evaluare existente pentru tehnicile de căutare asincrone.

4.1.1. Implementarea tehnicilor de căutare asincrone.

Studiile existente în literatura CSP şi DCSP se concentrează pe analiza din punct de vedere teoretic a diverselor metode de rezolvare (fie pentru cadrul CSP, fie pentru cadrul DCSP). Multe studii teoretice şi experimentale au fost realizate şi diverse platforme pentru modelul CSP au fost realizate (Ilog Solver, Chip, Choco, Gecode, etc). Aceste platforme au avantajul că utilizatori îşi concentrează efortul în modelarea problemelor reale şi construirea de tehnici eficiente. Scrierea aplicaţiilor distribuite în care se încadrează tehnicile de căutare asincrone specifice modelului DCSP este mult mai dificilă pentru că implică mult mai multe concepte faţă de cazul centralizat CSP. Este vorba de implementarea constrângerilor, programare concurentă şi asincronă pentru agenţi, structuri distribuite, etc. Pentru modelul DCSP singurele platforme de implementare şi studiu existente sunt *Frodo* [Pet06], ce se referă doar la cazul tehnicilor de optimizare existente pentru modelul DCSP şi *Mely* [Gal06].

Nu există foarte multe studii relative la modul în care se pot implementa aceste tehnici de căutare asincrone. Fiecare abordare teoretică necesită şi evaluări experimentale pentru aceste variante de tehnici de căutare asincrone. Conform literaturii de specialitate, autorii folosesc medii de tip JAVA pentru implementarea pe un singur sistem de calcul al tehnicilor de căutare asincrone, dar fără a fi prezentat un model general de implementare şi evaluare pentru aceste tehnici. Sunt folosite diverse moduri de simulare pentru agenţi şi pentru implementarea constrângerilor. Mai mult, fiecare dintre autorii tehnicilor asincrone s-au concentrat pe abordarea teoretică, furnizând demonstraţii teoretice pentru completitudinea algoritmilor. Detaliile de implementare sunt mai puţin cunoscute, eventuale implementări practice necesitând eforturi mari de lucru.

Studiile comparative pentru mai multe tehnici de căutare asincrone sau pentru versiunile unei tehnici de căutare asincrone, presupun rularea în condiţii identice, cum ar fi valori iniţiale identice, graful constrângerilor identic, întârzieri fixe în furnizarea mesajelor, etc.

Plecând de la aceste observaţii, în capitolul următor este propus un model de implementare şi evaluare aplicabil în NetLogo. Acest model, prin aplicarea unei metode de detecţie a terminării, permite obţinerea a două sisteme multi-agent ce permit implementări şi evaluări simultane pentru tehnicile de căutare asincrone în aceleaşi condiţii de rulare. De asemenea, în capitolul următor este prezentată şi o metodologie de implementare pentru tehnicile de căutare asincrone, metodologie ce furnizează detaliile necesare implementării efective în NetLogo.

4.1.2. Evaluarea tehnicilor asincrone.

Având ca ţintă evaluarea efortului depus de agenţi şi a timpului necesar obţinerii soluţiei, studiile existente în literatura au pus în evidenţă diferite criterii particulare folosite de anumiţi autori la evaluarea performanţelor tehnicilor DCSP. Este vorba de alegerea unor unităţi de măsură pentru evaluare şi a unor probleme suport pentru implementarea tehnicilor de căutare asincrone.

În ceea ce priveşte unităţile de măsură, acestea trebuie să respecte câteva principii generale în evaluarea oricărei implementări distribuite sau paralele. În primul rând trebuie eliminate detaliile de implementare sau legate de mediul distribuit sub care se face rularea (sistem de operare şi mediu de programare). Acest lucru este cât se poate de normal. În primul rând, se evită specificarea

limbajelor de programare şi a mediilor de programare ce implementează acele limbaje, pentru că aşa cum se cunoaşte, fiecare limbaj are avantajele şi dezavantajele sale legate de implementarea efectivă şi influenţează subiectiv evaluarea. În al doilea rând, arhitecturile distribuite existente influenţează foarte mult eficienţa tehnicilor.

Din analiza diverselor tehnici asincrone şi a modului de evaluare folosit de autorii acestora, plecând de la lista din [Sil02] completată cu alţi parametri**,** se remarcă utilizarea unei diversităţi de parametrii, după cum urmează:

- **Numărul total de mesaje**, notat cu *TM*. Este o măsură a încărcării globale a reţelei. Este important de analizat pentru medii distribuite ce folosesc Internetul sau reţele Ethernet. Este folosit în [BMM01] la analiza eficienţei algoritmului DisDB şi la analiza algoritmului DIFC.
- **Numărul total de constrângeri**, notat cu *TC*. Este o măsură a timpului global consumat de agenţii implicaţi. Dacă se utilizează protocoale între firele de execuţie a unei aplicaţii ce rulează pe un sistem cu un singur procesor, această unitate de măsură este o bună evaluare a eficienţei timpului. Este folosit la analiza algoritmului DIFC.
- **Numărul total de octeţi transferaţi**, notat cu *TB*. Reprezintă o formă mai exactă a numărului total de mesaje.
- **Numărul total de valori nogood generate**, notat cu *TNG*. De asemenea, vom contoriza **numărul total de valori nogood stocate**, notat cu *TNGS***.** Aceste unităţi sunt folosite în [HY00] şi [Mus05] pentru a vedea eficienţa algoritmilor asincroni din punct de vedere al eficienţei mesajelor nogood.
- **Cea mai lungă secvenţă de mesaje**, notată cu *LSM***.** Este o măsură pentru durata calculelor (numită ceasul logic al lui Lamport, vezi [Lam78]).Oferă o bună estimaţie a eficienţei agenţilor aflaţi la distanţă. Este folosită în [Sil02] la evaluarea unei variante de ABT, pe care introduce consistenţa valorilor nogood.
- **Cea mai lungă secvenţă de constrângeri verificate**, notată cu *LSC***.** Permite măsurarea complexităţii secvenţiale a problemei. Pentru computerele paralele, LSC este o bună măsură a eficienţei de timp a algoritmilor. Este folosită în [SHF00] la evaluarea algoritmului AAS.
- **Cea mai lungă secvenţă echivalentă de constrângeri verificate**, notată cu *LES***.** Se bazează pe asocierea unui cost φ la fiecare mesaj, măsurat în constrângeri verificate (putem vedea măsura LSM ca un LES ce contorizează local constrângerile ca evenimente cu costul 1). În combinaţie cu LSM reprezintă o bună unitate de măsură a eficienţei de timp. Este folosită în [SHF00] pentru evaluarea algoritmului **AAS**. Acolo sunt prezentate câteva posibile valori pentru φ**.**
- **Numărul de rotunjiri realizate de simulator**, notat cu *NSR*. Mulţi algoritmi au fost evaluaţi folosind simulatoare. Astfel de simulatoare rotunjesc numărul de mesaje tratate. Parametrul NSR aproximează parametrul LSM.
- **Numărul de constrângeri verificate de simulator**, notat cu *NSC***.** Este o aproximaţie a parametrului LSC.
- **Numărul de constrângeri echivalente verificate de simulator**, notat cu *NESC*. Pentru algoritmii evaluaţi cu ajutorul unor simulatoare, se poate estima parametrul LES prin însumarea celei mai lungi mulţimi de constrângeri cu numărul de rotunjiri multiplicat cu φ (adică cu costul). Este folosit împreună cu NSR în evaluarea algoritmului ABT**.**

Există tentaţia de a compara complexitatea algoritmilor distribuiţi cu complexitatea algoritmilor secvenţiali. Când se fac astfel de comparaţii, timpul secvenţial nu trebuie comparat cu timpul distribuit, eventual doar relativ la complexitatea mesajelor schimbate. Aşa cum se ştie, în cazul secvenţial timpul este realmente măsurat ca fiind cantitatea totală de timp necesară rulării acelui program, lucru datorat faptului că se contorizează efectiv instrucţiunile executate secvenţial. În schimb, în modelarea distribuită rularea programului implică manipulări de mesaje (recepţionarea lor, extragerea din cozile de mesaje şi tratarea lor). Trebuie subliniat faptul că agenţii specifici modelării DCSP, necesită transmiterea şi recepţionarea mesajelor. Cazul cel mai nefavorabil considerat pentru complexitatea de timp apare când toate mesajele sunt schimbate unul după altul şi astfel complexitatea de timp este egală cu complexitatea mesajelor.

Un alt factor ce poate influenţa evaluarea tehnicilor asincrone este dat de tipurile de probleme folosite la evaluare. Fiecare problemă poate fi rezolvată cu o anumită eficienţă de o anumită tehnică, în funcţie de dificultatea problemei. Pentru modelarea CSP se foloseau anumite tipuri de probleme clasice: problema celor n regine, problema m-colorării unui graf sau problema satisfacerii formulelor propoziţionale (SAT). De asemenea, s-au folosit şi alte probleme cum ar fi problema frecvenţei asocierilor, problema planificări lucrărilor, multe alte probleme de combinatorică. Aceste probleme au fost preluate şi pentru analiza tehnicilor DCSP, dar în formularea distribuită în care fiecare variabilele au fost preluată de agenţi. Pentru aceste probleme există câţiva parametri ce le definesc. Cei mai importanţi sunt *dimensiunea problemei* şi *densitatea grafului de constrângeri* asociat problemei DCSP.

Dintre aceste probleme, literatura DCSP utilizează câteva tipuri de probleme la evaluare:

- problema distribuită a celor n regine, caracterizată prin n=numărul de regine (densitate fixă pentru graful constrângerilor egală cu n*(n-1)/2).
- problema distribuită a m-colorării unui graf generat aleator, caracterizată prin n=numărul de noduri/agenţi, k=3 culori şi m-numărul de legături între noduri/agenţi. În [MJPL92] sunt definite 2 tipuri de grafuri: grafuri cu puţine legături (numite *sparse problems,* având *m=n x 2 legături)* şi grafuri cu un număr special de legături, cunoscute ca fiind probleme dificile (numite *dense problems,* având *m=n x 2.7 legături).*
- problema numită „binary random problems”, caracterizată prin 4-tuple (n, m, p1, p2) unde :
  - n este *numărul de variabile*.
  - m este *dimensiunea domeniului variabilelor*.
  - p1 este o porţiune din graful constrângerilor, generat aleator, p1 este exprimat ca un procent din acest graf al constrângerilor, problema având p1 * n * (n - 1)/2 constrângeri. p1 se mai numeşte *densitatea grafului de constrângeri*.
  - p2 este o porţiune din mulţimea maximală de perechi de valori ce sunt interzise, exprimată ca un procent din totalul de perechi de valori posibile a exista între toate variabilele, p2 * m*m perechi în conflict din cele maxim m*m perechi. P2 se mai numeşte *etanşeitatea constrângerilor*

Trebuie remarcat faptul că problema colorării unui graf este cea mai potrivită pentru evaluare, pentru că permite diferite densităţi pentru graful de constrângeri şi are foarte multe aplicaţii directe în practica reală.

Prin urmare, o evaluare corectă presupune selectarea unei clase cât mai variate de probleme, cu cât mai multe dimensiuni, cu cât mai multe seturi de date alese aleator sau alegerea de seturi de date care să permită densităţii variate pentru graful constrângerilor.

Din nefericire, fiecare rulare asincronă a aceleiaşi probleme poate da diferite rezultate. Această problemă se rezolvă prin mai multe rulări ale aceleiaşi probleme, pentru acelaşi set de date, selectându-se media artitmetică $\overline{M}$ şi dispersia $\sigma(M)$ (M este valoarea totală obţinută pentru un parametru de evaluare).

4.2. Redundanţa mesajelor transmise.

Analiza cozilor de mesaje, arată existenţa unui număr mare de mesaje redundante. Aceste mesaje pot fi redundante sau învechite. Aceste mesaje încarcă inutil costurile obţinerii soluţiei. În subparagrafele următoare sunt identificate mai multe tipuri de mesaje redundante, pentru care se vor încerca soluţii de reducere sau eliminare în capitolul 6.

4.2.1. Mesaje învechite de tip nogood.

În majoritatea tehnicilor de căutare asincrone, orice mesaj de tip *nogood* sau *back* este trimis de un agent A_k copil la un agent părinte A_i pentru a anunţa imposibilitatea selectării unei valori consistente pentru variabila avută în grijă. Recepţionarea unui mesaj de tip nogood presupune verificarea consistenţei acelei liste nogood. Conform definiţiei 3.1.8, lista nogood existentă în mesajul nogood este consistentă dacă conţine aceleaşi valori cu valorile agentului A_i pentru agenţii conectaţi cu el. În cazul majorităţii tehnicilor de căutare, agentul A_i acceptă mesajul nogood (tratându-l în mod corespunzător) doar dacă este consistent cu vederea sa.

Analiza consistenţei mesajelor nogood recepţionate este necesară deorece, datorită întârzierilor existente în furnizarea mesajelor, contextul apariţiei listei nogood s-a schimbat pentru agentul A_i. Cu alte cuvinte, informaţiile din mesajul nogood sunt învechite pentru agentul A_i. Conform definiţiei 3.1.9, mesajul învechit nu este consistent pentru agentul A_i, prin urmare el poate fi ignorat.

Analiza comportamentelor diverselor tehnici arată existenţa pentru tehnicile din familia ABT şi pentru celelalte tehnici ce au la bază un comportament asemănător al unui număr foarte mare de mesaje învechite descoperite de fiecare agent. Majoritatea tehnicilor ce au la bază principiile căutării asincrone resping mesajele învechite şi anunţă agentul ce a transmis lista nogood printr-un alt mesaj ok. Acesta, la rândul lui, va reanaliza contextul său de lucru, fiind posibil să nu mai genereze un nou mesaj nogood. Acest comportament duce la un flux suplimentar de mesaje şi mai ales la un efort local de calcul inutil.

4.2.2. Mesaje redundante de tip ok sau info.

Agenţii folosesc canale de comunicaţie de tip FIFO pentru transmiterea şi recepţionarea mesajelor. Fluxul cel mai mare de mesaje transmise de agenţi este cel de tip ok sau info.

Agenţii, mai ales în cazul tehnicilor din familia ABT, au un flux de mesaje foarte mare cu anumiţi agenţi dintre agenţii copii. Multe dintre acestea sunt mesaje inutile, ce încarcă costurile de obţinere a soluţiei în mod inutil. Mai departe, sunt

prezentate mai multe definiţii pentru identificarea mesajelor acestor mesaje inutile. În capitolul 6 sunt propuse mai multe soluţii de eliminare a acestor mesaje inutile.

Definiţia 4.2.1.- mesaj ok/info învechit. Un mesaj ok(A_k, val_{k1}) recepţionat de un agent A_i este *învechit* relativ la vederea agentului A_i dacă mai există în coada de mesaje un alt mesaj ok(A_k, val_{k2}), recepţionat mai târziu. Cu alte cuvinte mesajul ok(A_k, val_{k1}) a fost recepţionat înaintea mesajului ok(A_k, val_{k2}).

Definiţia 4.2.2.- mesaj ok/info redundant. Un mesaj ok(A_k, val_{k1}) recepţionat de un agent A_i este *redundant* relativ la vederea agentului A_i dacă mai există în coada de mesaje un alt mesaj ok(A_k, val_{k2}), recepţionat ultima dată de la agentul A_k, unde val_{k1}= val_{k1}. Cu alte cuvinte, mesajul ok(A_k, val_{k1}) conţine aceeaşi valoare cu cel mai recent mesaj recepţionat de la agentul A_k.

Aceste două tipuri de mesaje (învechite şi redundante) pot fi ignorate. Un agent A_i transmite mesajul *ok* la agenţii copil. Este posibil, la un moment dat, ca un agent A_k să aibă în coada sa de mesaje mai multe mesaje de tip ok recepţionate în diverse etape. Tratarea secvenţială sau în pachete a acestor mesaje şi întârzierile datorate efortului local de tratare a mesajelor nu permit tratarea completă a acestor mesaje. Mai mult, la un moment dat pot să existe în coada de mesaje două sau mai multe ok de la acelaşi agent. Dintre acestea, ultimul mesaj recepţionat este cel consistent relativ la contextul de lucru, celelalte mesaje fiind fie învechite fie redundante. Fără discuţie, acele mesaje ar trebui ignorate şi netratate. Totuşi, datorită modului de tratare a cozilor de mesaje, ultimul mesaj recepţionat este ultimul tratat, mai întâi fiind tratate cele vechi.

Analiza cozilor de mesaje în cazul tehnicilor din familia AWCS arată existenţa mesajelor învechite şi redundante. În [Yok01] se precizează că în cazul tehnicii AWCS agenţii citesc toate mesajele existente în cozile de mesaje. Totuşi, acolo nu se prezintă nici o modalitate de tratare, adică un protocol care să definească ordinea în care mesajele sunt tratate. Un astfel de protocol trebuie să precizeze modul de tratare a mesajelor, ordinea lor, momentul şi modul de apel pentru rutina de verificare a contextului de lucru, check-agent-view.

În ceea ce priveşte tehnicile din familia ABT, există două studii recente ce oferă o soluţie de management. Astfel, în [BM04] şi [JV05] sunt prezentate variante pentru tehnica ABT în care agenţii citesc toate mesajele existente în cozile de mesaje şi apoi le tratează. Studiile din [BM04] şi [JV05] prezintă pentru prima dată o strategie de tratare a mesajelor, strategie care precizează ordinea şi modul de tratare a mesajelor pe baza unui protocol, dar pentru familia ABT. Aplicarea acestor strategii duce la reducerea costurilor de obţinere a soluţiilor. Analiza acestei startegii este făcută în capitolul 6, folosind cele două sisteme de implementare propuse.

4.3. Explozia valorilor nogood pentru tehnicile asincrone.

Tehnicile existente pentru modelarea DCSP se bazează pe căutarea soluţiei în reţeaua de constrângeri ce se formează. Acestea sunt tehnici de căutare asincrone, cea mai mare parte complete, ce permit rezolvarea unei probleme modelate DCSP. Ele se remarcă prin diversitatea ideilor aplicate pentru ca agenţii să poată lucra asincron şi concurent, asigurându-se o anumită eficienţă.

Tehnicile de căutare asincrone se caracterizează prin apariţia valorilor „nogood” în timpul căutării soluţiei. Aceste tehnici se bazează pe trimiterea de mesaje nogood între agenţi pentru a executa un backtracking inteligent şi a asigura

completitudinea. Tehnica ABT necesită un spaţiu a cărui complexitate pentru fiecare agent este exponenţială [YDIK98]. Complexitatea exponenţială a algoritmului, în cazul nefavorabil, este determinată în primul rând de numărul înregistrărilor de mesaje *nogood.* Tehnica asynchronous backtracking se caracterizează printr-o explozie a valorilor nogood pentru probleme cu dimensiuni mari. Stocarea valorilor nogood, necesară în cadrul tehnicii asynchronous backtracking, necesită o cantitate mare de spaţiu.

De asemenea, şi tehnicile din cadrul familie ABT se bazează pe trimiterea de mesaje de tip nogood (numite mesaje back) între agenţi (conform ordinii dintre agenţi), dar suferă de o explozie a valorilor de tip nogood. Ca o consecinţă, timpul de calcul şi cerinţele pentru resursele hardware cresc mult mai repede decât dimensiunea problemei. Cu siguranţă, în aplicaţiile practice în care se folosesc medii de comunicaţii lente, cum ar fi Internet-ul, tehnica devine total ineficientă datorită numărului mare de valori nogood dezvoltate şi necesităţii comunicării acestora în timpul căutării. Stocarea acestor valori nogood va necesita un spaţiu de memorie foarte mare, implicând şi un efort de căutare mult mai mare.

Apariţia valorilor nogood are ca efect introducerea unor noi constrângeri derivate din cele vechi. Deşi lista nogood indică cauza eşecului şi incorporarea ei ca o nouă constrângere va învăţa agenţii să nu mai repete aceeaşi greşeală, este de dorit ca în timpul derulării algoritmului, să nu apară astfel de situaţii sau ele să fie cât mai puţine, pentru că ele au ca efect mărirea timpului de execuţie al algoritmului (datorită faptului că noi mesaje sunt transmise, noi constrângeri sunt evaluate).

Recent, o tehnică derivată, numită asynchronous backtracking cu flaguri, permite reducerea exploziei valorilor nogood pentru tehnica asynchronous backtracking. Această tehnică este propusă în [GWW03] unde se investighează eficienţa şi cantitatea de valori nogood folosind problema celor n regine, varianta distribuită. Totuşi, din literatura de specialitate se cunoaşte că această problemă nu este potrivită şi suficientă pentru a putea face comparaţii asupra performanţelor unei tehnici asincrone [MJPL92]. Se recomandă folosirea problemei colorării grafurilor, varianta distribuită, problemă ce permite diverse densităţi pentru graful de constrângeri (spre deosebire de problema celor n regine care acceptă o densitate fixă).În literatura de specialitate nu apar referinţe despre studii care să investigheze eficienţa acestei tehnici, comparativ cu tehnica de bază, în aceste condiţii.

Tehnicile din familia ABT se caracterizează printr-o explozie a valorilor nogood, mai ales pentru probleme de dimensiune mare. O altă caracteristică specifică tehnicilor asincrone din familia ABT este aceea că este necesară stocarea valorilor nogood pentru ca agenţii să nu mai repete aceeaşi greşeală. Acest lucru conduce la o cantitate de memorie foarte mare necesară stocării acestor valori nogood, mai ales pentru probleme cu dimensiune mare.

Complexitatea de stocare se referă la cantitatea de memorie utilizată de un algoritm, contorizată cu diferite unităţi de măsură. Această unitate de măsură este foarte importantă în analiza anumitor tehnici de căutare, mare parte dintre tehnicile asincrone existente suferind de o explozie a valorilor nogood. Este posibil ca aceste costuri legate de cantitatea de memorie folosită să facă inutilizabilă o anumită tehnică, chiar dacă din punct de vedere al calculelor este foarte eficientă.

4.4 Învechirea informaţiilor stocate de agenţi.

4.4.1. Agenţi neconectaţi.

Agenţii comunică mesajele la agenţii cu care sunt conectaţi iniţial. Conform ordinii dintre agenţi, o parte dintre agenţi sunt agenţi evaluatori (agenţi copii), iar o parte sunt agenţi părinţi ce trimit mesajele de tip ok. Aceste conexiuni permit construirea unui graf al constrângerilor, graf folosit ca şi suport în transmiterea mesajelor.

Mesajele de tip ok informează agenţii despre schimbarea valorilor, permiţându-le să ia decizii corecte. Totuşi, în cazul a doi agenţi ce nu sunt iniţial conectaţi, deciziile unuia pot avea efecte asupra contextului de lucru pentru cel de-al doilea. În mod tipic, dacă un agent nu reuseşte să găsească nici o valoare consistentă, el transmite un mesaj nogood la cel mai apropiat agent aflat înanintea lui în ordinea de priorităţi, dintre părinţi. Un astfel de mesaj poate conţine vederea sa *agent_view* sau o submulţime a acesteia ce conţine doar valorile agenţilor cu care intră în conflict. Mesajul nogood este recepţionat de un alt agent, ce nu are aceiaşi agenţi vecini. În mesajul nogood el va găsi valorile unor agenţi cu care nu este conectat. Stocarea acelor valori în lista sa agent_view nu este suficientă, pentru că acele valori nu vor putea fi actualizate corespunzător datorită inexistenţei unor canale de comunicaţie cu acei agenţi. Acest lucru conduce la învechirea informaţiilor stocate de agenţi.

Efectul stocării unor informaţii învechite apare la respingerea unor valori nogood ca fiind învechite datorită faptului că un agent a stocat informaţii vechi despre agenţii neconectaţi. În [YDIK92] se propune pentru prima dată o soluţie de adăugare a unor conexiuni suplimentare cu agenţii ce apar în mesajele nogood recepţionate de un agent. Aceste legături spre noi agenţi vor fi folosite pentru transmiterea de mesaje ok de informare a schimbărilor în valorile ataşate agenţilor. Acest lucru permite obţinerea unui algoritm complet. În [HBQ98] este propusă o soluţie de eliminare a acestor legături, prin fixarea unor legături suplimentare între părinţi, dar în faza de preprocesare. Din păcate, în [Yok00] şi [BMM01] se arată că acest lucru duce la necompletitudinea algoritmului obţinut.

4.4.2. Apariţia legăturilor temporare în timpul căutării soluţiei.

În [BM05] este propus un cadru unificator, un nucleu de plecare pentru câteva dintre cele mai importante şi cunoscute tehnici asincrone. Acest ABTkernel nu necesită adăugarea de legături suplimentare cu agenţii prezentaţi anterior. În [BM05] se arată că inexistenţa acelor legături duce la învechirea informaţiilor stocate de un anume agent. Acest lucru are ca efect necompletitudinea nucleului ABT. Autorii propun mai multe soluţii de eliminare a informaţiilor învechite dintre agenţi. O primă soluţie prezentată în [BM05] de eliminare a informaţiilor învechite este de a adăuga noi legături pentru a permite agentului ce a recepţionat un mesaj nogood să decidă dacă este sau nu învechit. Aceste legături au fost propuse în tehnica ABT, varianta Yokoo [YDIK92], [YDIK98].

Legăturile care se adaugă în timpul rulării pot deveni permanente sau temporare. Aceste două tipuri de legături permit obţinerea a două tehnici derivate din nucleul ABT kernel:

- Adăugarea de legături în timpul căutării prin care se obţine varianta ABT de bază (Yokoo). Legăturile se adaugă în timpul căutării în momentul

recepţionării unui mesaj nogood (Back) care conţine informaţii relative la un agent neconectat, conform ordinii dintre agenţi. Noile legături sunt permanente.

- Adăugarea de legături temporare în timpul căutării prin care se obţine varianta ABT_{temp} Legăturile se adaugă ca şi la varianta ABT Yokoo, în timpul căutării în momentul recepţionării unui mesaj nogood (Back) care conţine informaţii relative la un agent neconectat. Legăturile adăugate nu sunt permanente, ele sunt păstrate pentru ca agenţii să schimbă un număr suficient de mesaje, pentru a elimina informaţiile învechite. După transmiterea acelui număr variabil de mesaje, legăturile sunt anulate.

A doua soluţie (numită ABT_{temp}) constă în a păstra temporar acele legături între agenţii care nu pot determina dacă o informaţie este sau nu învechită [BM05]. Practic, se vor păstra temporar acele legături între agenţii care nu pot determina dacă o informaţie este sau nu învechită [BM05].

În literatura de specialitate nu apar referinţe despre studii care să determine o valoare optimă pentru acel număr de mesaje. În [BM05] se analizează rezultatele versiunii ABT_{temp} pentru k=10, valoare dedusă din aproape în aproape pentru cazul particular al problemei cu 16 agenţi folosită la evaluare. Problema determinării unei valori optime pentru numărul de mesaje rămâne deschisă.

4.5. Utilizarea completă a informaţiilor din valorile nogood.

4.5.1. Stocarea valorilor nogood.

O caracteristică a tehnicilor de căutare asincronă se referă la modul de operare cu valorile de tip nogood, care sunt combinaţii de valori inconsistente. Este vorba despre modul în care se construiesc aceste valori, despre metodele pe care le pun în aplicare pentru detectarea valorilor nogood şi modul în care se stochează aceste combinaţii de valori.

În cadrul modelării DCSP, algoritmul asynchronous weak-commitment search (AWCS) joacă un rol fundamental şi de pionierat între algoritmii de căutare asincronă. Algoritmul se remarcă atât prin faptul că suferă de o explozie a valorilor nogood, cât şi prin schimbarea dinamică a ordinii agenţilor, fiind publicat de Yokoo în [YDIK98]. Algoritmul este considerat un algoritm eficient în ce priveşte numărul de ciclii, eficienţă obţinută prin schimbarea dinamică a ordinii agenţilor. Există mai multe studii având ca scop îmbunătăţirea eficienţei acestei tehnici, studii ce se concentrează pe minimizarea şi selecţia valorilor nogood [HY00], [Yok01].

Constrângerile restricţionează intern sau extern comportamentul unui agent. Astfel, studierea constrângerilor poate oferi modalităţi de creştere a eficienţei algoritmilor existenţi în cadrul DCSP. Apariţia valorilor nogood are ca efect introducerea unor noi constrângeri. Valorile nogood indică cauza eşecului şi încorporarea lor ca o nouă constrângere va învăţa agenţii să nu mai repete aceeaşi greşeală [YDIK98], [SHF00a], [BMM01].

In [AD97], Armstrong construieşte o nouă tehnică asincronă în care rezolvarea problemei este divizată în epoci. Există un agent central care este responsabil pentru pornirea procesului de căutare şi un procesor de nogood-uri ce păstrează informaţii despre nogoodurile apărute. Când un nogood este descoperit, agentul va trimite acea valoare unui nogood processor, care o va salva în lista de valori nogood apărute. Mai târziu când un agent încearcă să atribuie o valoare la

propria sa variabilă, se consultă procesorul de nogooduri pentru a fi siguri ca aceea combinaţie de valori nu conduce la un nogood.

Tot în [AD97] se arată că aplicarea tehnicii nogood processor aduce beneficii în ceea ce priveşte reducerea costurilor. În literatură nu apar referinţe despre aplicarea acestei tehnici şi la alte tehnici de căutare asincrone, având ca ţintă reducerea costurilor. Este interesant de văzut dacă această tehnică poate duce la îmbunătăţirea performanţelor anumitor tehnici. De asemenea, este interesant de analizat comportamentul tehnicii nogood processor în situaţia filtrării mesajelor, respectiv în cazul apariţiei de întârzieri în furnizarea mesajelor.

Nerestricţionarea înregistrării valorilor nogood poate deveni în anumite cazuri impracticabilă. Motivul principal este acela că stocarea valorilor nogood consumă excesiv memorie şi poate conduce la epuizarea memoriei disponibile. În mod tipic, numărul de valori nogood creşte cu numărul de conflicte: în cel mai rău caz această creştere poate fi exponenţială în numărul de variabile.

Un alt efect neplăcut al stocării unui număr mare de valori nogood este legat de faptul că verificarea asocierilor curente în lista de valori nogood stocate devine foarte costisitoare, efortul de căutare anulând beneficiile aduse de stocarea valorilor nogood. Aceste probleme sunt probleme deschise ce trebuie analizate pentru a vedea dacă această tehnică nogood processor aduce beneficii asupra eficienţei.

4.5.2. Construirea unor valori nogood eficiente

În [HY00], Hirayama şi Yokoo introduc termenul de *nogood learning* sau *learning,* termen ce se referă la construirea valorilor nogood. În [HY00], se prezintă şi analizează o schemă numită "resolvent-based learning", care, aplicată algoritmului AWCS aduce rezultate foarte bune în ceea ce priveşte numărul de ciclii necesari pentru a rezolva problema. Tehnica se bazează pe construirea unui nou nogood eficient, doar prin verificarea câtorva valori nogood.

Modelarea CSP oferă mai multe tehnici de rezolvare având la bază căutarea. Aceste tehnici pot fi îmbunătăţite prin aplicarea tehnicilor loop-back. Aceste tehnici loop-back se bazează pe folosirea informaţiilor despre căutarea realizată. Există mai multe tehnici loop-back, cunoscute din literatura de specialitate, prezentate în [FD94], [Gin93], [RR98], [BM96], [Dec90].

Tehnica nogood learning, introdusă în [HY00], este o nouă metodă de învăţare pentru valorile nogood, ce se poate aplica la tehnicile DCSP, metodă bazată pe adaptarea tehnicilor loop-back prezentate în [FD94], [Gin93], [RR98], la cadrul DCSP. Ideea de bază este aceea că pentru fiecare posibilă valoare pentru variabila eşec, se selectează un nogood care interzice acea valoare şi apoi se construieşte un nou nogood în afara celui obţinut prin unirea nogood-urilor selectate. Autorii acestei metode compară acest nou nogood ca fiind asemănător cu noţiunea de rezolvent din logica propoziţională, de aici şi denumirea acestei metode *rezolvent bazat pe învăţare.*

Conform literaturii de specialitate nu există nici un studiu care să combine cele două metode prezentate aici: folosirea informaţiilor din nogoodurile stocate cu construirea unor nogooduri eficiente. Acest lucru va fi studiat în această teză, pentru a utiliza complet informaţiile din valorile nogood.

4.6. Concluzii

Analiza din acest capitol a identificat câteva probleme deschise existente la tehnicile de căutare asincrone. Au fost identificare diverse defecte şi dezavantaje ale tehnicilor de căutare asincrone, plecând de la situaţiile reale ce pot apărea în practică.

Tehnicile de căutare asincrone se caracterizează prin apariţia valorilor „nogood" în timpul căutării soluţiei. Tehnica asynchronous backtracking se caracterizează printr-o explozie a valorilor nogood pentru probleme cu dimensiuni mari. Stocarea valorilor nogood, necesară în cadrul tehnicii asynchronous backtracking, necesită o cantitate mare de spaţiu. De asemenea, şi tehnicile din cadrul familie ABT se bazează pe trimiterea de mesaje de tip nogood între agenţi, dar suferă de o explozie a valorilor de tip nogood. Ca o consecinţă, timpul de calcul şi cerinţele pentru resursele hardware cresc mult mai repede decât dimensiunea problemei. Cu siguranţă, în aplicaţiile practice în care se folosesc medii de comunicaţii lente, cum ar fi Internet-ul, tehnica devine total ineficientă datorită numărului mare de valori nogood dezvoltate şi necesităţii comunicării acestora în timpul căutării.

Analiza cozilor de mesaje pentru implementările tehnicilor de căutare asincrone, arată existenţa unui număr mare de mesaje redundante sau învechite. Aceste mesaje încarcă inutil costurile obţinerii soluţiei. Ele apar din cauza întârzierilor în furnizarea mesajelor către agenţi.

Tehnicile derivate din tehnica ABT necesită adăugarea de legături pentru eliminarea informaţiilor învechite dintre agenţi. Aceste noi legături permit unui agent ce a recepţionat un mesaj nogood să decidă dacă este sau nu învechit. Totusi, aceste legături sumplimentare vor duce la creşterea fluxlui de mesaje, multe dintre acestea fiind inutile. Prin urmare, limitarea fluxului de mesaje pentru legăturile temporare, adăugate în timpul căutării, este o problemă deschisă pentru tehnicile din familia ABT.

Tehnicile din familia AWCS necesită stocarea valorilor nogood pentru a asigura completitudinea algoritmului. Nerestricţionarea înregistrării valorilor nogood poate deveni în anumite cazuri impracticabilă. Motivul principal este acela că stocarea valorilor nogood consumă excesiv memorie şi poate conduce la epuizarea memoriei disponibile. Această probleme trebuie analizate pentru a vedea dacă informaţiile din valorile nogood trebuie stocate şi dacă se pot utiliza eficient aceste informaţii, fie pentru căutarea de noi valori, fie pentru construirea unor noi valori nogood eficiente.

SECŢIUNEA II.

CONTRIBUŢII LA IMPLEMENTAREA ŞI EVALUAREA TEHNICILOR DE CĂUTARE ASINCRONE

Capitolele secţiunii de faţă conţin contribuţii aduse în domeniul implementării tehnicilor de căutare asincrone. Secţiunea prezintă două modele de implementare şi evaluare pentru tehnicile de căutare asincrone în NetLogo. Cele două modele sunt utilizate ca suport în studiul comportamentului agenţilor şi identificarea unor îmbunătăţiri în secţiunile următoare.

În capitolul 5 este analizat mediul NetLogo cu scopul construirii unui model general de implementare şi evaluare pentru tehnicile asincrone astfel încât să se poată utiliza mediul NetLogo ca un simulator de bază în studiul tehnicilor asincrone. În acest capitol sunt propuse cele două modele de implementare şi evaluare pentru tehnicile asincrone, împreună cu arhitecturile celor două sisteme distribuite obţinute.

În capitolul 6 este prezentat un studiu comparativ pentru cele mai importante tehnici asincrone, studiu experimental bazat pe implementarea şi evaluarea tehnicilor de căutare asincrone folosind cele două modele prezentate anterior. Scopul principal al acestui studiu este de a studia comportamentul agenţilor în cele două situaţii de implementare: cu sincronizarea execuţiei agenţilor şi execuţie asincronă. Sunt propuse mai multe soluţii de îmbunătăţire a performanţelor pe baza studiului comparativ.

5. MODEL DE IMPLEMENTARE ŞI EVALUARE PENTRU TEHNICILE DE CĂUTARE ASINCRONE.

În cadrul acestui capitol se propune propune un model general de implementare şi evaluare pentru tehnicile de căutare asincrone. Acest model general permite utilizarea mediului NetLogo ca un simulator de bază pentru studiul tehnicilor asincrone. Pe baza acestui model de implementare a tehnicilor de cutare asincrone se poate studia comportamentul agenţilor în diverse situaţii, cum ar fi: ordinea de prioritate a agenţilor, sincronizarea execuţiei agenţilor, valori iniţiale diferite pentru variabilele asociate agenţilor, apariţia de întârzieri în furnizarea mesajelor etc. În felul acesta se pot identifica posibile îmbunătăţiri pentru performanţele tehnicilor asincrone, îmbunătăţiri identificate şi prezentate în capitolele următoare.

Tehnicile asincrone de căutare, existente pentru modelarea DCSP, se încadrează în categoria aplicaţiilor distribuite. Agenţii pot fi procese existente pe un singur sistem de calcul sau pe mai multe sisteme de calcul existente într-o reţea de calculatoare. Implementarea oricărei tehnici de căutare asincrone presupune implementarea agenţilor şi a constrângerilor existente, implementarea legăturilor dintre agenţi şi a canalelor de comunicaţie dintre aceştia, practic implementarea procesului de execuţie a agenţilor.

În acest capitol se încearcă identificarea modului în care se pot simula agenţii, cum se pot implementa constrângerile, cum se pot simula canalele de mesaje, cum se pot contoriza diversele unităţi de măsură a performanţelor tehnicilor asincrone. Pentru aceasta este prezentată o soluţie de modelare şi implementare a procesului de execuţie a agenţilor existenţi în cazul tehnicile de căutare asincrone. Plecând de la modelul de implementare şi evaluare propus, în acest capitol se prezintă două metode de detecţie a terminării tehnicilor de căutare asincone, metode care conduc la obţinerea a două sisteme multi-agent utilizabile la implementarea şi evaluarea tehnicilor asincrone.

Din păcate, nu există un mediu distribuit dedicat modelărilor cu constrângeri distribuite, toate mediile existente sunt medii generale, implementarea agenţilor şi a constrângerilor implicând un anume efort de calcul, mai mare sau mai mic în funcţie de performanţele acelui mediu.

Capitolul începe cu prezentarea mediului NetLogo, a principalelor obiecte Netlogo ce vor fi utilizate drept suport la implementarea şi evaluarea tehnicilor de căutare asincrone. Apoi, este prezentat modelul de implementare a procesului de execuţie a agenţilor şi modul în care se poate face evaluarea tehnicilor asincrone. Pentru acest model sunt propuse două metode de detecţie a terminării algoritmilor obţinându-se două sisteme multi-agent. Pentru cele două sisteme multi-agent sunt propuse două arhitecturi de implementare, arhitecturi utilizate la implementarea şi evaluarea versiunilor tehnicilor de căutare asincrone prezentate în capitolele următoare, în anumite condiţii experimentale. În final, este prezentată o metodologie de implementare a tehnicilor de căutare asincrone, pe baza modelului propus.

5.1. Mediul NetLogo.

NetLogo este un mediu de modelare programabil ce poate fi folosit pentru simularea unor fenomene naturale sau sociale realizat de Uri Wilensky la Northwestern University (SUA), [Wil99]. În particular, mediul oferă o suită de sisteme complexe de modelare dezvoltate în timp. Modelele utilizează sute de agenţi independenţi, toţi agenţii putând opera în paralel.

Netlogo reprezintă următoarea generaţie de limbaje dintr-o serie de limbaje de modelare cu agenţi, ce a început cu StarLogo. NetLogo este un mediu scris integral în Java, astfel el poate fi instalat şi rulat pe cele mai importante platforme (Mac, Windows, Linux), fiind necesară doar instalarea maşinii virtuale Java.

Mediul NetLogo conţine o bibliotecă de modele de simulare predefinite ce pot fi utilizate şi modificate. Aceste modele predefinite acoperă o arie variată din natură sau ştiinţe sociale, incluzând biologie şi medicină, fizică şi chimie, matematică şi ştiinţa calculatoarelor, economie şi psihologie socială [Wil99].

Dintre trăsăturile sale se menţionează:

- Sistem:
  - Rulează pe platforme diverse MacOs, Windows, Linux etc.
  - Modelele pot fi salvate şi rulate ca şi appleturi Java în paginile WEB.
- Limbaj:
  - Limbaj programabil complet.
  - Este o extensie a unui limbaj structurat (Logo) cu suport pentru agenţi şi programare paralelă.
  - Număr nelimitat de agenţi şi variabile.
- Mediu de programare:
  - Suport grafic de afişare pentru obiectele NetLogo.
  - O interfaţă grafică ce oferă un constructor grafic pentru aplicaţii, ce conţine o colecţie de controale grafice.
  - Un sistem de plotare puternic.
  - Un sistem HubNet ce poate fi folosit într-o reţea de calculatoare pentru simulări.
  - Un monitor pentru agenţi ce se poate folosi pentru inspectarea şi controlul agenţilor.
  - O colecţie de instrumente (numită *BehaviorSpac**e***) ce se poate utiliza pentru colectarea datelor din multiple rulări ale unui model. Este o unealtă foarte utilă pentru experimente.
  - Funcţii de export şi import (export de date in anumite formate, salvarea şi restaurarea stărilor unui model).

NetLogo este o extensie importantă a limbajului tradiţional Logo. În primul rând, permite un număr nelimitat de obiecte de tip *turtle*. Pe de altă parte, este o extensie a simulatorului StarLogo, dezvoltat la MIT, prin introducerea unor facilităţi de calcul paralel. În al doilea rând, obiectele de tip *turtles* sunt echipate cu senzori ce permit sesizarea comportamentului fiecărui obiect de tip *turtles* (aceste obiecte pot fi folosite pentru simularea fiecărui agent). Se pot defini reguli de comportament pentru obiectele de tip turtles astfel încât să sesizeze interacţiunea agenţilor cu mediul.

5. 2. Obiectele NetLogo. Programarea distribuită în NetLogo.

"Lumea" NetLogo este alcătuită din agenţi. Fiecare agent poate duce la bun sfârşit o anumită activitate, toţi agenţii rulând simultan şi concurent.

Limbajul NetLogo pune la dispoziţie trei tipuri de agenţi: *turtles*, *patches* şi *observatorul*. Obiectele de tip *turtle* sunt agenţi ce se pot deplasa în lumea NetLogo. Lumea NetLogo este bidimensională şi este împărţită într-un grid de *patch*-uri. Fiecare *patch* este o piesă pătrată ce reprezintă suportul peste care obiectele de tip *turtle* se pot deplasa. *Observatorul* nu are o locaţie fixă, el poate fi imaginat ca aflându-se deasupra lumii de obiecte *turtles* şi *patches.* Observatorul este un agent sistem ce poate iniţia diverse operaţii pentru ceilalţi agenţi. La lansarea mediului agenţii tip turtle nu există, aceştia pot fi creaţi de către observator. De asemenea, fiecare obiect de tip patch poate crea noi obiecte de tip turtles.

NetLogo utilizează comenzi şi raportori pentru a comunica agenţilor ce să facă (comenzile şi raportorii sunt primitive NetLogo). Comenzile sunt acţiuni pentru agenţi. În schimb, raportorii returnează anumite valori. Comenzile şi raportorii se pot grupa în proceduri utilizator (se foloseşte construcţia *to numeprocedură*). Există o comandă specială (numită *ask)* care permite să se specifice agenţilor de tip turtle şi patch ce comenzi să execute. În figura 5.1 se prezintă un exemplu de lucru cu comanda ask.

```
To setup
  ca            ;; se curata ecranul
  crt 10        ;; se creează 10 noi turtles
end

to go
  ask turtles
    [ fd 1      ;; toate obiectele de tip turtles se mută cu un pas
      rt random 10     ;; …şi se rotesc aleator
      lt random 10 ]
end
```

Figura 5.1. Crearea şi execuţia calculelor agenţilor prin intermediul comenzii ask

O mulţime de agenţi se poate grupa într-un *agentset*. Un *agentset* poate conţine oricare din obiectele de tip turtles sau patches, dar nu ambele. Conceptul de mulţime de agenţi a fost introdus pentru a putea aplica comenzi sau proprietăţi.

NetLogo permite definirea de diferite "tipuri" de turtles, numite *breeds*. Odată ce a fost definit un breed, se poate fixa un comportament diferit pentru acesta. Aceste obiecte sunt folosite pentru a simula diversele obiecte existente în problemele DCSP. De exemplu, agenţii din problema celor n-regine pot fi definiţi cu ajutorul obiectelor de tip breed (o construcţie de tipul breeds [queens]). Acest lucru permite fixarea unui comportament special pentru fiecare agent-regină. Când se definesc obiecte de tipul breed, în mod automat se creează un agentset pentru fiecare *breed*. Pentru a implementa agenţii din algoritmul ABT, pentru problema distribuită a celor n-regine, se poate folosi o construcţie de tipul *breeds [queens]*. Secvenţa NetLogo corespunzătoare este prezentată în figura 5.2.

Există o colecţie de controale grafice (limitate ca număr) ce permit realizarea unei interfeţe pentru aplicaţia DCSP. Dintre controalele grafice existente, se menţionează butoanele de comandă ce furnizează cel mai simplu mod de a controla un model. În mod tipic, orice model are cel puţin un buton de tip "setup", buton ce permite iniţializarea aplicaţiei, şi un buton "go" folosit pentru startarea aplicaţiei. La aceste butoane sunt ataşate proceduri de calcul specifice NetLogo care conţin comenzi şi apeluri de raportori.

```
breeds [queens]
globals [no-more-messages tmp]
queens-own [message-queue col current-view nogoods messages-
received_ok messages-received_nogood]
;rows şi  cols  primesc valori între [0..num-queens-1]
```

Figura 5.2. Crearea agenţilor DCSP prin intermediul obiectelor de tip breeds

O problemă foarte importantă este legată de modul de execuţie al procedurilor ataşate unui agent, agent simulat folosind obiecte de tip *breed*. Aplicaţiile DCSP necesită execuţia simultană şi asincronă a codului ataşat fiecărui agent. Acest lucru este posibil în NetLogo pentru că comenzile sunt executate asincron, fiecare obiect de tip "turtle" sau "patche" execută lista sa de comenzi cât poate de repede. Există două moduri de execuţie a comenzilor fiecărui agent. Primul mod constă în "alinierea" comenzilor executate de fiecare turtle, prin depunerea lor în blocul unei comenzi *ask*. În acest mod, paşii executaţi nu vor fi sincronizaţi. Un astfel de exemplu de cod NetLogo este prezentat în figura 5.3

```
ask turtles
  [ fd random 10
    do-calculation
  ]
```

Figura 5.3. Alinierea comenzilor executate de agenţi.

În figura 5.3. obiectele de tip turtle îşi vor începe execuţia comenzii *do-calculation* în momente diferite (nu se face o sincronizare pentru că obiectele de tip turtle se pot muta într-un timp diferit,). În schimb, rescriind codul din exemplul anterior apelând la două comenzi ask se va obţine o sincronizare, fiecare turtle va aştepta până când se vor termina calculele celorlalte obiecte turtle (figura 5.4). Astfel, toate obiectele de tip turtle vor începe simultan execuţia comenzii *do-calculation*.

```
ask turtles
   [ fd random 10 ]
ask turtles
   [ do-calculation ]
```

Figura 5.4. Sincronizarea comenzilor executate de agenţi.

În general, tehnicile DCSP sunt tehnici asincrone ca şi comportament. Prin urmare, este foarte important ca mediul să permită execuţia asincronă a calculelor,

dar să permită şi sincronizarea calculelor. Fiecare agent trebuie să manipuleze anumite tipuri de mesaje (ok, info, back, nogood, etc) asincron. Acest lucru se poate implementa în NetLogo, un exemplu de cod fiind prezentat în figura 5.5.

```
 to handle-message
  if (empty? message-queue) [stop]
  set msg retrieve-message
  show msg
  ...
  if (first msg = "ok")[
     ok xj dj]
  if (first msg = "nogood")[
     nogood item 1 msg
]
end
```

Figura 5.5. Manipularea mesajelor de către agenţi.

O problemă ce poate apărea în cazul tehnicilor asincrone este legată de situaţiile în care este nevoie de execuţia unor comenzi de către un agent, fără a permite celorlalţi agenţi să-i intrerupă execuţia comenzilor. Astfel, ceilalţi agenţi sunt puşi în aşteptare până când comenzile din bloc sunt terminate. Acest lucru se poate realiza în NetLogo, un exemplu fiind prezentat în figura 5.6.

```
crt 5
;; se crează 5 obiecte de tip turtles
ask turtles
  [ without-interruption
     [ type 1 fd 1 type 2 ] ]
=> 1212121212
;; fiecare turtle tipăreşte 1 şi mută, apoi scrie 2.
ask turtles
  [ type 1 fd 1 type 2 ]
=> 1111122222
;; fiecare turtle tipăreşte 1 şi mută simultan,
;; apoi scrie 2 simultan.
```

Figura 5.6. Execuţia unor calcule fără întrerupere

Acestă situaţie apare de foarte multe ori în cazul aplicaţiilor DCSP (de exemplu, pentru a putea manipula corect mesajele, conform ordinii de transmitere).

Ultimul element de limbaj prezentat, necesar implementării aplicaţiilor DCSP, este legat de existenţa unor structuri de date care să permită manipularea datelor. NetLogo oferă implicit o structură de tip listă cu elemente de tip numeric sau string. Această listă se foloseşte pentru reprezentarea mesajelor ok sau mesajelor nogood. Cu ajutorul acestor liste se pot simula structuri de tip coadă (ce permit simularea canalelor de comunicaţii dintre agenţi), în care se vor stoca mesajele recepţionate de fiecare agent, în ordinea în care ele sosesc. De asemenea, cu ajutorul „listelor de liste” se va putea reprezenta graful de constrângeri. Acest lucru va permite să se definească diverse ordini pentru agenţi.

5.3. Modelarea şi implementarea procesului de execuţie a agenţilor

În acest paragraf este prezentată o soluţie de modelare şi implementare a procesului de execuţie a agenţilor existenţi în cazul tehnicile de căutare asincrone. Această modelare, prin aplicarea unei tehnici de detecţie a terminării algoritmilor, permite obţinerea a două sisteme multi-agent ce se pot aplica la implementarea şi evaluarea celor mai reprezentative tehnici de căutare asincrone. Această modelare poate fi utilizată la implementarea majorităţii tehnicilor de căutare asincrone, cum sunt cele din familia AWCS [YDIK98], familia ABT [BM05], DisDB [BMM01], AAS [SHF00a]. Modelarea propusă în acest paragraf permite obţinerea de implementări pentru versiuni derivate de tehnici de căutare asincrone în care se simulează diverse situaţii existente în realitate: întârzieri în furnizarea mesajelor, managementul mesajelor etc.

Exemple de implementări realizate de autor pentru aceste tehnici se găsesc pe site-ul NetLogo ([MASNet-b]) şi în [MASNet-a,c] .

Orice implementare a tehnicilor de căutare asincrone presupune parcurgerea a 2 etape:

- programarea agenţilor astfel încât aceştia să ruleze concurent.
- realizarea interfeţei aplicaţiei respective.

Modelarea procesului de execuţie a agenţilor va fi structurată pe două niveluri, corespunzătore celor două etape de implementare. Definirea modulului în care se vor programa tehnicile asincrone încât agenţii să ruleze concurent şi asincron va reprezenta nivelul intern al modelului. Al doilea nivel se referă la modul de reprezentare a aplicaţiei în NetLogo şi reprezintă nivelul exterior. Primul aspect va fi tratat şi rezolvat folosind obiecte de tip *turtles.* Cel de-al doilea aspect (ce este legat mai mult de problema de rezolvat) se referă la modul de interacţiune cu utilizatorul, la interfaţa de lucru. În ceea ce priveşte acest aspect, NetLogo ne oferă obiecte de tip *patches* şi controale grafice diverse. În orice caz, obiectele de tip patches vor permite simularea suprafeţei aplicaţiei.

5.3.1. Simularea şi iniţializarea agenţilor

Agenţii sunt simulaţi utilizându-se obiecte de tip breed (aşa cum s-a prezentat în paragraful anterior aceştia sunt tipuri de turtles).

În figura 5.7 este prezentat modul de definire a agenţilor împreună cu structurile de date globale şi proprietare agenţilor.

breeds [*agenţi*]
globals [variabile ce simulează memoria partajată de toţi agenţii]
***agent*-own** [message-queue current-view nogoods messages-received_ok messages-received_nogood nr_constraintc, AgentC_Cost]
;message-queue conţine mesajele recepţionate.
;current-view este o listă indexată după numărul agentului de forma [v_0 v_1 v_2...], v_i = -1 dacă nu se cunoaşte valoarea acelui agent.
;current-view poate conţine şi prioritatea sub forma [[$valoare_0$ $prioritate_0$] [$valoare_1$ $prioritate_1$] ...] valoare = -1 şi prioritate = -1 dacă nu sunt cunoscute de agent.
;nogoods este lista de valori inconsistente [0 1 1 0 ...] unde 0 este poziţie bună, iar 1 este inconsistentă.

```
;messages-received_ok şi messages-received_nogood sunt variabile ce
contorizează numărul de mesaje ok şi nogood recepţionate de un agent.
; constraintc şi AgentC_Cost sunt variabile ce permit contorizarea efortului
local depus de agent.
```

Figura 5.7. Definirea agenţilor în cazul tehnicilor asincrone de căutare

Pentru tipurile de probleme alese pentru evaluarea tehnicilor de căutare asincrone se vor defini agenţi folosind construcţii de tipul *breeds [queens]* sau *breeds [vertices]*. Acest lucru va permite programarea fiecărui agent DCSP conform tehnicilor de căutare asincrone alese pentru implementate.

Definirea agenţilor în cazul celor două probleme clasice utilizate la evaluare (problema celor n regine şi problema colorării grafurilor) este prezentată în figurile 5.8 şi 5.9.

```
breeds [queens]
globals [no-more-messages tmp]
queens-own [message-queue col current-view nogoods messages-
received_ok messages-received_nogood nr_constraintc, AgentC_Cost]
;rows şi cols primesc valori între [0..num-queens-1]
;message-queue conţine mesajele care intră. Se prelucrează totdeauna
mesajul din faţa listei
;col este poziţia reginelor pe un anume rând
;current-view este o listă indexată după numărul reginei de forma [col0 col1
col2...], col = -1 dacă este necunoscută.
;nogoods este lista de poziţii inconsistente [0 1 1 0 ... ] unde 0 este poziţie
bună, iar 1 este inconsistentă.
;messages-received_ok şi messages-received_nogood sunt variabile ce
contorizează numărul de mesaje ok şi nogood recepţionate de un agent
;constraintc şi AgentC_Cost sunt variabile ce permit contorizarea efortului
local depus de agent
```

Figura 5.8. Agenţii NetLogo în cazul problemei celor n –regine

```
breeds [vertices ]
;muchiile sunt reprezentate ca o lista de liste: un 2D array indexat prin
"who".Valorile sunt 1 (dacă există link), 0 altfel
;domain este lista de culori permise
globals [edges domain no-more-messages done]
vertices-own [message-queue, current-view, nogoods ... nr_constraintc,
AgentC_Cost]
;message-queue conţine mesajele care intră. Se prelucrează totdeauna
;mesajul din faţa listei.
;current-view is a list indexed by agent number [c0 c1 c2...] color = -1 dacă
este necunoscută.
;nogoods is a list of inconsistent positions [0 1 1 0 ... ] unde 0 este valoare
bună (consistentă), iar 1 este inconsistentă.
```

Figura 5.9. Agenţii NetLogo în cazul problemei colorării grafurilor

Se observă construirea agenţilor cu numele *queens*, respectiv *vertices*. Aceştia vor reprezenta agenţii DCSP ce vor executa procedurile de tratare a mesajelor specifice tehnicilor asincrone. Pentru fiecare agent, se pot defini variabile şi structuri de date proprietare, care vor fi variabile globale pentru agenţi. Posibilitatea de a defini variabile globale sau proprietare permite construirea completă şi corectă a agenţilor ce sunt simulaţi. Există un lucru foarte interesant, un agent simulat printr-un turtle poate accesa variabilele proprietare pentru alt agent, astfel încât agenţii pot coopera şi negocia între ei. Acest lucru permite unui posibil simulator inspectarea valorilor agenţilor de către alţi agenţi. De asemenea, existenţa variabilelor globale permite simularea unor situaţii în care agenţii folosesc zone de memorie comună pentru accesarea anumitor date.

Iniţializarea agenţilor presupune construirea agenţilor şi iniţializarea structurilor de date necesare operării agenţilor. Pentru iniţializare se propune o procedură de iniţializare a fiecărui agent, procedură prezentată în figura 5.10 (procedura se va numi *setup* în toate implementările utilizate în această teză). În mod tipic sunt construiţi agenţii (în număr de *num-agents*) necesari rulării tehnicii de căutare asincrone şi iniţializate structurile de date. Aceste structuri de date vor fi definite în paragraful următor.

```
to setup-agenţii // se folosesc agenţii definiţi cu construcţia breeds [agenţi]
;se creează agenţii în număr de num-agents şi se iniţializează
  create-custom-agenţi num-agents [
    set messages-received_ok 0
    set messages-received_nogood 0
    set current-view get-list num-agents -1
    set nogoods get-list num-agents 0
    set message-queue []
    …
    ]
end
```

Figura 5.10 Procedura de iniţializare a fiecărui agent-setup

5.3.2. Reprezentarea şi manipularea mesajelor.

Orice tehnică de căutare asincronă se bazează pe utilizarea de către agenţi a unor mesaje pentru a comunica diverse informaţii necesare obţinerii soluţiei. Comunicarea între agenţi se face conform modelului de comunicaţie introdus la definirea DCSP. Manipularea mesajelor presupune în primul rând reprezentarea mesajelor. Acest lucru se poate realiza în Netlogo prin utilizarea unor liste indexate. Pentru reprezentarea mesajelor complexe, ce conţin foarte multe informaţii, Netlogo permite folosirea listelor de liste (elementele listei sunt de asemenea liste). Modul de reprezentare a principalelor mesaje întâlnite la tehnicile de căutare asincrone este prezentat în continuare:

- **set** mesaj (**list** "ok" agent valoare costuri_agent) – mesaje de tip *ok sau info;*
- **set** mesaj (**list** "nogood" agent current-view costuri_agent) - mesaje de tip *nogood* sau *back;*
- **set** mesaj (**list** "addl" $agent_1$ $agent_2$ costuri_agent)
- **set** mesaj (**list** "removel" $agent_1$ $agent_2$ costuri_agent)

Se remarcă faptul că definiţia unui mesaj conţine tipul mesajului, agentul ce a transmis şi informaţiile corespunzătoare tipului de mesaj.

Modelul de comunicare existent în cadrul DCSP presupune în primul rând existenţa unor canale de comunicaţie de tip FIFO în care să se stocheze mesajele recepţionate de fiecare agent. Simularea cozilor mesaje pentru fiecare agent se poate face folosind liste Netlogo, pentru care se definesc rutine de tratare corespunzătoare principiilor FIFO. Aceste structuri de date se definesc odată cu definirea agenţilor. În implementările propuse în această teză această structură se va numi *message-queue*. Această structură proprietară fiecărui agent va conţine toate mesajele de tip ok si nogood recepţionate de agentul respectiv (sau orice alt tip de mesaj). Aceste structuri de tip coadă ce au un rol foarte important în funcţionarea tehnicilor de căutare asincrone, trebuie să respecte principiile FIFO.

Manipularea acestor canale poate fi gestionată fie de către un agent central (ce în NetLogo se numeşte *observator*) fie de agenţii respectivi. Pentru aceasta se propune construirea unei rutine numite *update* pentru manipularea globală a canalelor de mesaje. Ea va avea rol şi în detectarea terminării execuţiei tehnicilor de căutare asincrone. Aceasta rutină *update* reprezintă un fel de "program principal, un centru de comandă al agenţilor. Într-o astfel de procedură, ce trebuie să ruleze continuu (până la golirea cozilor de mesaje) se verifică pentru fiecare agent coada de mesaje (pentru a detecta o pauză în transmiterea de mesaje). De asemenea, procedura trebuie să permită operarea cu mesajele ce sunt transmise de agenţi. Pentru aceasta ea va apela pentru fiecare agent o altă procedură care să trateze efectiv fiecare mesaj, corespunzător tipului acestuia. Această procedură va fi numită *handle-message*, ea ocupându-se de manipularea mesajelor specifice fiecărei tehnici de căutare asincrone. În figurile 5.11 şi 5.12 sunt prezentate cele două proceduri principale de manipulare şi tratare a mesajelor, *update* şi *handle-message*. Cele două proceduri sunt cele mai importante din punct de vedere al modului de lucru sincron sau asincron cu mesaje (mod de lucru care defineşte tehnicile asincrone).

```
to update
  set no-more-messages true
  ask agenţi [
    if (not empty? message-queue)[
      set no-more-messages false]
   ]
;dacă toate cozile sunt goale, nu mai există mesaje, se poate opri algoritmul
  if (no-more-messages) [stop]
;se apelează procedura de manipulare a mesajelor din cozile de mesaje, pentru
fiecare agent.

ask agenţi [handle-message]

ask agenţi [
    create-temporary-plot-pen "q" + who
    plot messages-received_nogood
    ….
    ]
;se pot face reprezentări grafice după diverşi parametri (nr. de mesaje, etc )
  end
```

Figura 5.11. Procedura NetLogo update

În procedura *update* din figura 5.11 se observă apelul *"ask agenţi …"*, ce permite execuţia calculelor pentru fiecare agent, dar cu o sincronizare după fiecare rulare a agenţilor. Varianta de mai sus se bazează pe folosirea observatorului în manipularea canalelor cu mesaje. Acest comportament al comenzii ask permite să se construiască un sistem distribuit sincron, aşa cum este definit în [HY00]. În paragraful §5.3.6 se va mai prezenta o variantă pentru update ce va permite obţinerea unui al doilea sistem multi-agent de implementare şi evaluare pentru tehnicile de căutare asincrone.

```
;se manipulează mesajul din coadă şi se identifică tipul său (ok sau nogood,
addl sau removel) apelând procedura de tratare.
to handle-message
  locals [msg … ]

  if (empty? message-queue) [stop]
  set msg retrieve-message

 if (first msg = "ok")
  [
      set messages-received_ok messages-received_ok + 1
      procedure_ok msg
      ;se apelează procedura de tratare a mesajelor de tip ok
  ]

  if (first msg = "nogood")
  [
       set messages-received_nogood messages-received_nogood + 1
      procedure_nogood msg
     ;se apelează procedura de tratare a mesajelor de tip nogood
 ]

 if (first msg = "addl")
  [
      procedure_addlink msg
     ;se apelează procedura de adăugare de noi legături
  ]
…
end
```

Figura 5.12. Procedura de tratare a mesajelor

În figura 5.12, unde este prezentată procedura de tratare a mesajelor, se observă faptul că fiecare agent extrage câte un mesaj din coada de mesaje, identifică tipul de mesaj şi apelează procedura de tratare a mesajului respectiv. Procedurile de tratare a mesajelor de un anume tip se pot implementa conform algoritmului în cauză. De exemplu, pentru tehnica ABT, cele două proceduri sunt prezentate în figura 5.13.

```
;Procedura de tratare a mesajelor ok
;Parametrii sunt notaţi astfel Qj = Xj şi Vj = dj
to ok [Qj Vj]
   ;se actualizează contextul de lucru
   set current-view replace-item Qj current-view Vj
   check-agent-view
end

; Procedura de tratare a mesajelor nogood
to nogood [ nogoodmsg ]
   locals [ i j msg]
  … stocarea şi tratarea mesajului nogood
 end
```

Figura 5.13. Procedurile de tratare a mesajelor de tip ok şi nogood

Un alt aspect analizat este cel relativ la ordinea agenţilor. În capitolul 3 au fost prezentate mai multe tehnici, cum sunt cele din familia ABT, în care ordinea agenţilor este statică. Prin urmare, transmiterea mesajelor ok (care informează asupra apariţiei unei noi valori) se face către anumiţi agenţi. Există tehnici asincrone care necesită sau pot utiliza o ordine dinamică pentru agenţi. În această categorie avem AWCS şi de curând o variantă ABT cu ordine dinamică.

O primă problemă este dată de caracteristicile mediului NetLogo. Comenzile agenţilor sunt executate folosind, în general, comanda ask. Fiecare agent, folosind comanda ask, manipulează mesajele din propria coadă de mesaje. Această comandă este executată pentru obiectele de tip *turtles* sau *breed*. Conform regulilor NetLogo, comanda ask este planificată să execute în ordine crescătoare agenţii, conform ordinii date de numărul de ordine. Ordinea de execuţie este aleasă determinist şi într-un mod reproductibil. Aceste caracteristici existente în NetLogo sunt potrivite pentru cazul ordinilor statice, lexicografice.

În cazul tehnicilor care folosesc ordini dinamice, soluţia constă în construirea de liste ce conţin agenţii cu care trebuie comunicat. De exemplu, în AWCS se pot păstra vecinii fiecărui agent într-o listă, mesajele fiind trimise spre acei agenţi în ordinea indicilor. O astfel de listă poate fi ordonată după criterii date de diverse euristici, astfel încât să se respecte specificul unei anumite tehnici.

5.3.3 Definirea şi reprezentarea interfeţei

În ceea ce priveşte partea de interfaţă, se pot folosi pentru reprezentarea grafică a obiectelor problemei DCSP (regine sau noduri) obiecte de tip *patch*. Este recomandabil să se creeze o procedură de iniţializare a suprafeţei de afişare a valorilor agenţilor. O astfel de procedură este prezentată în figura 5.14.

```
to setup-patches
  ask patches with [(abs (pxcor + pycor)) mod 2 = 0 and
            pxcor < num-queens - screen-edge-x and
           pycor > screen-edge-y - num-queens]
    [set pcolor 8]
end
```

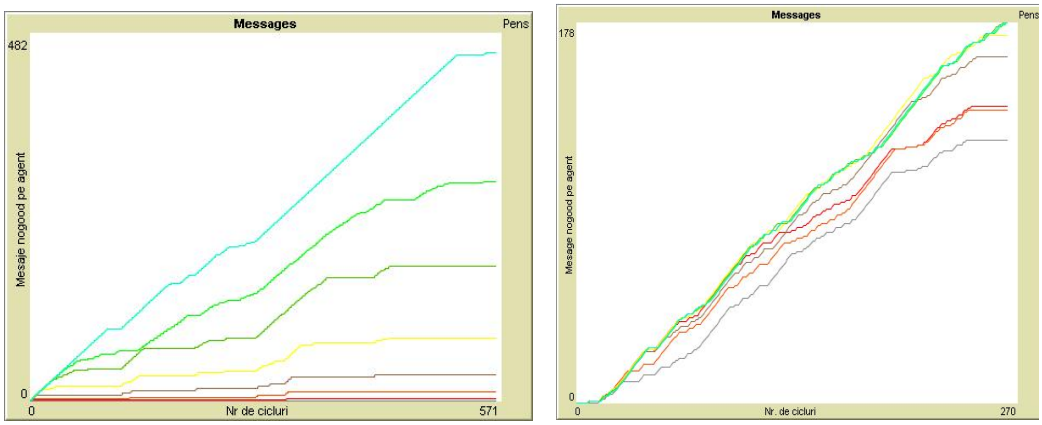
Figura 5.14. Procedura de iniţializare a suprafeţei de lucru a fiecărui agent

În mod similar, pentru cazul problemei colorării grafurilor, se face reprezentarea nodurilor şi a legăturilor ([MASNet-a] şi în [MASNet-b,c]**)**. Cele două proceduri de iniţializare se vor ataşa (folosind o procedură setup) la un buton de start al aplicaţiei, ca în secvenţa din figura 5.15.

```
to setup
  ca
  setup-patches
  setup-queens
  ask queens
       [procedura_iniţializare]
  … alte iniţializări
end
```

Figura 5.15. Procedura de setare a aplicaţiei NetLogo

Mediul NetLogo permite şi afişarea sub formă de grafic a variaţiei numărului de mesaje pentru fiecare agent, lucru foarte util în analiza influenţei ordinii agenţilor asupra fluxului de mesaje sau comportamentului agenţilor relativ la acea unitate de măsură. Acest lucru se poate realiza folosind obiecte grafice de tip *plot* în care se poate trasa, pentru fiecare agent, numărul de valori manipulate la fiecare pas. De exemplu, în figura 5.16 este captată evoluţia numărului de mesaje nogood per agent, la fiecare ciclu pentru tehnicile ABT şi AWCS. Analiza variaţiei fluxului de mesaje din figura 5.16 arată comportamente diferite pentru agenţii din cele două tehnici. În cazul tehnicii ABT (figura 5.16(a)) agenţii de prioritate mică necesită un flux de mesaje mult mai mare. În schimb, în cazul tehnicii AWSC, fluxul de mesaje este asemănător pentru toţi agenţii, acest lucru datorându-se şi ordinii dinamice folosite în această tehnică.



(a) ABT **(b) AWCS**

Figura 5.16. Evoluţia fluxului de mesaje nogood pentru tehnicile ABT şi AWCS în cazul problemei celor n regine (n=8 agenţi)

5.3.4. Detecţia terminării tehnicilor de căutare asincrone în cazul modelului de implementare propus

Pentru majoritatea tehnicilor asincrone soluţia este detectată în general numai după ce apare o perioadă de pauză în transmiterea mesajelor (adică nu se mai transmite nici un fel de mesaj, stare numită *quiescence*). Această stare este uzual recunoscută utilizând mecanisme distribuite generale. Pentru anumite tehnici de căutare asincronă (cum ar fi tehnica AAS), soluţiile pot fi detectate înainte de această pauză. Aceasta înseamnă că terminarea poate fi dedusă mai devreme şi că numărul de mesaje necesare pentru obţinerea soluţiei poate fi redus. În această teză se va considera cazul în care soluţia este detectată după ce apare o pauză în transmiterea mesajelor.

Mai multe tehnici distribuite CSP detectează linişte (o pauză) în transmiterea de mesaje utilizând tehnicile generale din [CL85]. Aceşti algoritmi se remarcă prin faptul că verificarea terminării trebuie să fie iniţiată de agenţii care suspectează apariţia unei pauze în transmisia mesajelor. Totuşi, trebuie precizat faptul că există şi alte tehnici care pot detecta cât mai repede o pauză în transmiterea mesajelor.

Există două situaţii de analizat: situaţia în care există soluţie şi cea în care nu există soluţie. Finalizarea construirii soluţiei, în NetLogo, se bazează pe detectarea unei pauze în transmiterea mesajelor. Această situaţie poate fi rezolvată prin verificarea cozilor de mesaje, cozi ce trebuie să fie goale.

Pentru detecţia terminării tehnicilor de căutare asincrone, implementaţi după modelul NetLogo prezentat, se propun în acest capitol două metode originale de detecţie a terminării execuţiei tehnicilor asincrone [MusBP05]. Cele două metode se bazează pe soluţii diferite pentru detectarea unei pauze în transmiterea mesajelor. Aceste două metode se aplică pentru orice implementare a unor tehnici asincrone, ce foloseşte modelul de implementare prezentat. Aplicarea celor două modele de detecţie a terminării procesului de execuţie a agenţilor conduce la două sisteme multi-agent ce vor fi utilizate în capitolele următoare la analiza tehnicilor de căutare asincrone. Exemple de implementări de tehnici asincrone ce folosesc cele două metode se găsesc pe site-urile [MASNet-a], [MASNet-b,c].

5.3.4.1. Detecţia terminării procesului de execuţie de către agentul central "Observer"

Prima soluţie de detecţie a terminării se bazează pe câteva din facilităţile mediului NetLogo: comanda ask ce permite execuţia calculelor fiecărui agent şi existenţa agentului central *observator.* Manipularea canalelor de comunicaţie va fi realizată de către acest agent central. Pentru aceasta se propune construirea unei rutine numite *update* pentru manipularea globală a canalelor de mesaje. Într-o astfel de procedură, ce trebuie să ruleze continuu (până la golirea cozilor de mesaje) se verifică de către agentul *observator* dacă s-a detectat o pauză în transmiterea de mesaje. De asemenea, procedura trebuie să permită operarea cu mesajele ce sunt transmise de agenţi. Procedura *update* va apela pentru fiecare agent procedura de tratare efectivă a mesajelor. Această procedură va fi numită *handle-message* şi va fi apelată în interiorul procedurii *update* folosind comanda *ask*. Astfel, cele două elemente (apelul procedurilor locale de manipulare a mesajelor cu ask şi detecţia de către *observator*) vor conduce la o variantă de implementare în care se face sincronizarea execuţiei agenţilor.

Practic, după execuţia calculelor, fiecare agent aşteaptă finalizarea calculelor de către ceilalţi agenţi. Comanda ask este controlată la rândul ei de către agentul central *observer.* Acesta, după fiecare sincronizare, verifică dacă toate canalele de mesaje sunt goale (în acel moment). Acest lucru permite întreruperea execuţiei şi tipărirea soluţiei. Această metodă permite obţinerea unui sistem multi-agent cu sincronizarea execuţiei agenţilor. Acest sistem va fi prezentat în paragraful următor.

Prima metodă de detecţie a terminării are în centru agentul sistem „observator", căruia i se predă această sarcină de verificare a unei pauze în transmiterea mesajelor. Acest lucru se realizează prin asocierea procedurii update la „observatorul" NetLogo. Acest observator va apela la comanda ask, pentru a manipula mesajele primite de un agent, pentru că NetLogo nu permite rularea unor rutine ataşate agenţilor de către agentul central. "Observatorul" poate fi privit ca un agent sistem ce iniţiază procedura de terminare a algoritmului. În figura 5.17 este prezentat codul rutinei update, rutină ataşată *observatorului*.

Observatorul foloseşte o variabilă globală (în figura 5.17 numită no-more-messages) prin care se sesizează dacă cozile de mesaje sunt goale, semnal că a apărut o pauză în transmiterea mesajelor. Se remarcă faptul că dacă există mesaje în cel puţin o coadă de mesaje, se apelează procedura de manipulare a mesajelor, folosind instrumentul NetLogo ask.

```
to update
  set no-more-messages true
  set nr_cycles nr_cycles + 1
  ask agenti [
    if (not empty? message-queue)[
       set no-more-messages false]
   ]

 if (no-more-messages)
  [  ifelse Solution
        [WriteSolution]
     [WriteNoSolution]
   stop
    ]
  ask agenti [handle-message]
  …
end
```

Figura 5.17. Procedura update – 1

Această primă metodă conduce la o sincronizare a execuţiei agenţilor, datorită comportamentului comenzii "ask". Această comandă aşteaptă ca fiecare agent să termine de executat comenzile sale. Ca o consecinţă, se poate obţine un sistem cu sincronizare, asemănător cu cel folosit în [HY00] la evaluarea tehnicilor AWCS cu nogood learning.

O altă consecinţă este aceea că se poate contoriza numărul de cicli necesari pentru obţinerea soluţiei. Modul de implementare şi de detecţie a terminării permite reţinerea şi a numărului de ciclii printr-o variabilă globală ce se incrementează de fiecare dată când se execută rutina update (de către "observator").

5.3.4.2. Detecţia terminării procesului de execuţie de către fiecare agent DCSP

Cea de-a doua metodă permite obţinerea de implementări cu un comportament complet asincron pentru agenţi. Această metodă este mult mai generală permiţând evaluarea tehnicilor asincrone în condiţii apropiate de funcţionarea din aplicaţiile practice, în care nu există o sincronizare a execuţiei agenţilor. În acest caz, fiecare agent execută asincron şi concurent calculele sale, neexistând nici o sincronizare. Această situaţie se rezolvă prin introducerea unor indicatoare, care în orice moment păstrează starea canalelor de comunicaţie pentru fiecare agent (true dacă nu există mesaje şi false, în caz contrar). Dacă, la un moment dat, toate indicatoarele sunt true, se poate opri execuţia tehnicilor. Trebuie remarcat faptul că acele indicatoare îşi schimbă valoarea de-a lungul execuţiei, în funcţie de existenţa şi inexistenţa unor mesaje în cozile de mesaje.

A doua metodă de detecţie are ca bază de plecare renunţarea la "observator" pentru a sesiza apariţia unei pauze în transmiterea mesajelor şi execuţia procedurii update de fiecare agent. Detecţia va fi făcută de primul agent (de tip turtle-breed) ce sesizează apriţia acesteia. Soluţia (din NetLogo) este dată de utilizarea unor butoane de tip "turtle forever-buttons", butoane ce nu sunt ataşate observatorului. Practic, nu se va mai apela la comanda "ask" pentru a executa rutinele de manipulare a mesajelor. Acestea vor fi executate pentru fiecare agent (turtle) prin intermediul unui buton (ataşat la turtle) de tip „forever-button" (aşa cum se observă în figura 5.18).

Fiecărui agent i se ataşează un indicator care sesizează starea sa locală. Detectarea terminării algoritmului se va face folosind aceşti indicatori, care vor păstra în orice moment starea canalelor de comunicaţii pentru fiecare agent (true, dacă nu există mesaj în coada de mesaje şi false în caz contrar). Dacă la un moment dat toţi indicatorii sunt true, se poate trece la oprirea algoritmului. Acest lucru este realizat de primul agent turtle ce detectează această stare. În figura 5.18 se prezintă codul procedurii update.

```
to update

 handle-message

 ;detectarea terminării execuţiei este făcută de fiecare turtle
 ;primul care sesizează  că toate cozile sunt goale va trece la oprirea
 ;algorimului
 if (sum (values-from agenti [gt]) = num-agenti)
 [
 ifelse Solution
       [WriteSolution]
    [WriteNoSolution]
  stop
 ]

end
```

Figura 5.18. Procedura update - 2

Folosirea celei de-a doua metode de detecţie a terminării necesită adaptarea rutinei de manipulare a mesajelor, obţinându-se rutina din figura 5.19.

```
to handle-message
  if not empty? message-queue
  [
   set msg retrieve-message
   set gt 0
   if (first msg = "ok?")[
     handle-ok-message item 1 msg]
   if (first msg = "nogood")[
     handle-nogood-message msg]
  ]
  … tratarea altor tipuri de mesaje
   if (empty? message-queue)
  [
   set gt 1
   stop
  ]
 end
```

Figura 5.19. Procedură handle-message – 2

Se observă faptul că dacă coada de mesaje a unui agent nu este goală, indicatorul gt devine 0 (false), în caz contrar el devine 1. Acest lucru este necesar pentru că agenţii trimit mesaje asincron şi concurent la alţi agenţi aşa încât în momente diferite de timp cozile de mesaje trec prin diferite stări. Această a doua soluţie permite construirea unui alt tip de sistem de evaluare în NetLogo cu agenţi, fără sincronizarea execuţiei agenţilor. Acest sistem permite studierea comportamentului agenţilor în condiţii mai apropiate de situaţiile reale.

În ceea ce priveşte cazul în care nu există soluţie, se propune folosirea unei variabile globale "Solution", care reţine starea rezolvării problemei. În primul rând, tehnicile care folosesc o ordine statică, ordine lexicografică şi o parte din cele care folosesc ordine dinamică, detecţia inexistenţei unei soluţii (ceea ce tipic se recunoaşte prin apariţia unui nogood vid) este găsită dacă agentul cu numărul de ordine „0" detectează un nogood. În ceea ce priveşte anumite tehnici ce folosesc o ordine dinamică pentru agenţi, aşa cum este tehnica AWCS, detectarea inexistenţei soluţiei se face dacă s-au înregistrat toate nogood-urile posibile sau s-a detectat un mesaj vid. Apariţia uneia dintre aceste situaţii va duce la setarea variabilei "Solution" pe false.

5.3.5. Evaluarea tehnicilor de căutare asincrone pentru modelul propus.

Pentru evaluarea acestor tehnici există mai multe metrici care asigură o anumită independenţă faţă de limbajele de programare folosite la implementarea acestora. Aceste metrici permit evaluarea tehnicilor asincrone din mai multe puncte de vedere, cum ar fi efortul local şi efortul global depus de agenţi, încărcarea reţelei datorată fluxului de mesaje schimbat.

5.3.5.1. Costuri de comunicare.

Primul criteriu, poate cel mai important în analiza algoritmilor distribuiţi, este acela al *costurilor datorate comunicării de informaţii* între diferite părţi ale algoritmului. Comportamentul asincron întâlnit la tehnicile de căutare asincrone influenţează foarte mult costurile de comunicaţie. Tehnicile de căutare asincrone sunt caracterizate prin folosirea mesajelor de către agenţi în timpul procesului de căutare a soluţiei. Se utilizează mai multe tipuri de mesaje pentru anunţarea şi schimbarea valorilor locale atribuite variabilelor avute în grijă. În această categorie sunt mesajele de tip *ok?* sau *info*, mesaje de tip *nogood* sau *back*, folosite pentru a anunţa apariţia unei inconsistenţe (o combinaţie de valori inconsistente). Alte tehnici (ce obţin performanţe mai bune) folosesc mesaje speciale în scopul îmbunătăţirii soluţiei. Alte tipuri de mesaje sunt *add-link* sau *removel,* prin care se cere adăugarea de noi legături, respectiv anularea unor legături. De asemenea, se mai utilizază mesaje pentru detecţia soluţiei, cum ar fi mesajele *accepted* sau *stop.*

Contorizarea fluxului de mesaje permite evaluarea încărcării globale a reţelei. Este important de analizat pentru medii distribuite ce folosesc Internetul sau reţelele Ethernet. Prin urmare, modelul prezentat trebuie să permită contorizarea şi stocarea costurilor de comunicaţie.

Modelul prezentat în acest capitol permite contorizarea diverselor tipuri de mesaje utilizate de tehnicile de căutare asincrone. Acest lucru se poate face prin folosirea unor variabile globale ataşate agenţilor. Pentru modelul prezentat se propune folosirea a câte unei variabile proprietare fiecărui agent (*messages-received_nogood şi messages-received_ok)*, variabilă ce trebuie incrementată în momentul recepţionării unui mesaj. Această variabilă este incrementată în rutina de manipulare a mesajelor *handle-message***.** Se pot contoriza mesajele recepţionate şi prelucrate, respectiv doar cele transmise.

5.3.5.2. Costuri de timp

Complexitatea de timp este dată de timpul necesar executării calculelor şi se exprimă în termenii *celui mai lent mesaj* implicat în calcul. Această complexitate de timp este comparabilă cu complexitatea de timp a algoritmilor în cazul clasic secvenţial, dar ea nu este realmente o măsurare a timpului, ci o contorizare a operaţiilor consecutive de timp. O altă observaţie importantă, relativă la modul în care se face calculul complexităţii de timp este aceea că se ignoră timpul necesar procesării calculelor, acesta fiind considerat neglijabil în comparaţie cu întârzierile care apar în transmisia mesajelor.

Tehnicile asincrone sunt evaluate de anumiţi autori [YDIK98], [Yok01] folosind pentru complexitatea de timp *ciclu-ul*. Un ciclu este format din acele activităţi necesare ca toţi agenţii să citească mesajele recepţionate, să execute calculele locale şi să trimită mesaje la agenţii corespunzători. Această unitate de măsură asigură o anumită independenţă asupra condiţiilor locale şi a celor existente în mediul distribuit, neputând fi influenţată de întârzierile care apar. Este o unitate de măsură foarte bună pentru acele medii care simulează un mediu distribuit real. Pentru modelul prezentat în acest capitol, numărul de cicluri poate fi determinat folosind o variabilă globală NetLogo (fără ca ea să fie proprietară agenţilor). Această variabilă va fi incrementată în rutina *Update*. De fapt, numărul de apeluri pentru procedurea *Update* reprezintă numărul de cicluri executate de agenţi pentru a obţine

soluţia, respectiv inexistenţa soluţiei. Numărul de ciclii poate fi determinat doar în cazul sistemului multi-agent cu sincronizarea execuţiei agenţilor.

Complexitatea de timp este evaluată şi folosind numărul total de constrângeri verificate de fiecare agent. Este o măsură a timpului global consumat de agenţii implicaţi. De asemenea, permite evaluarea efortului local depus de fiecare agent. Numărul de constrângeri verificate de fiecare agent se poate contoriza folosind variabilele proprietare fiecărui agent (nr_constraintc). Un astfel de contor este incrementat în rutinele *check-agent-view*, la verificarea consistenţei vederii agentului. O problemă mai complicată este legată de contorizarea constrângerilor concurente introduse în [MKRZ02]. Acest lucru se realizează pentru modelul prezentat prin introducerea unei variabile proprietare fiecărui agent, variabilă numită *AgentC_Cost.* Variabila *AgentC_Cost* va reţine numărul de constrângeri concurente pentru agent, valoarea sa fiind transmisă către agenţii cu care este conectat prin intermediul mesajelor. Fiecare agent, la recepţionarea unui mesaj ce conţine o valoare *SenderC_Cost*, îşi actualizează propriul contor *AgentC_Cost* cu noua valoare conform algoritmului din [MKRZ02].

Modelul propus în acest capitol permite pentru ca fiecare agent să obţină valoarea maximă obţinută de ceilalţi agenţi în urma unor calcule. În [MusPP05a] şi [MusPP06] se propune limitarea numărului de mesaje transmise pentru legăturile temporare folosind o valoare maximă a fluxului de mesaje nogood învechite recepţionate de fiecare agent. Practic, această valoare se determină ca fiind cel mai mare număr de mesaje nogood învechite recepţionate la un moment dat. Pentru calcul acestei valori este necesară cunoaşterea de către fiecare agent a numărului maxim de mesaje învechite recepţionate de ceilalţi agenţi. Acest lucru se poate realiza pentru modelul propus prin transmiterea maximelor fiecărui agent la cei cu care este conectat în momentul transmiterii unui mesaj de tip „info" sau „nogood". În felul acesta fiecare agent va cunoaşte, la un moment dat, maximele celorlalţi agenţi.

Îmbunătăţirea performanţelor tehnicilor de căutare asincrone presupune cunoaşterea anumitor informaţii de către toţi agenţii. Este cazul tehnicilor de nogood processor sau limitarea numărului de mesaje transmise de fiecare agent. Aceste informaţii trebuie cunoscute de fiecare agent la un moment dat. Modelul prezentat permite două soluţii. O primă soluţie posibilă constă în folosirea unor variabile globale, neproprietare, accesibile de toţi agenţii. O astfel de soluţie corespunde existenţei unui agent central ce stochează aceste informaţii. Totuşi, o astfel de soluţie nu este apropiată de situaţia practică în care agenţii fie nu pot comunica cu un astfel de agent central sau pot comunica, dar cresc costurile obţinerii soluţiei. A doua soluţie se bazează pe ideea algoritmului de determinare a numărului de constrângeri concurente. Plecând de la această idee în figura 5.20 se propune un algoritm de determinare a valorilor maxime sau minime cunoscute de fiecare agent, folosind doar mesajele transmise de agenţi.

1: Fiecare agent iniţializează variabilele contor *ContorL* cu 0. De asemenea, se iniţializează *MaxContor/MinContor* cu 0.
2: Când un agent transmite un mesaj el include şi valoarea contorului său *MaxContor/MinContor*.
3: Când un agent identifică o valoare de tipul corespunzător pentru un alt agent A_k se actualizează contorul corespunzător din lista *ContorL*.
Replace-item A_k *ContorL* with item A_k *ContorL* + 1

```
4: if an agent receives a message with a counter SenderMx  then
      Set  MaxContor = max { ContorL} //      Set  MinContor = min { ContorL}
      if MaxContor < SenderMx then     //      if MinContor < SenderMx then
          Set MaxContor  SenderMx      //          Set MinContor  SenderMx
      end if                           //      end if
   end if
```

Figura 5.20. Determinarea valorii maxime/minime recepţionate de fiecare agent

Fiecare agent foloseşte o listă de variabile contor în care memorează valorile recepţionate de la fiecare agent (de exemplu nogoodurile recepţionate, numărul de valori schimbate etc). Apoi, fiecare agent calculează într-o variabilă proprietară maximul sau minimul din structura sa de date. La recepţionarea unui mesaj, actualizează acest maxim/minim cu valoarea transmisă de alt agent. Din faptul că agenţii au legături pe baza constrângerilor, ei vor recepţiona maximele/minimele de la agenţii vecini, iar aceştia de alţi vecini, s.a.m.d.

5.4. Sisteme multi-agent pentru implementarea şi evaluarea tehnicilor de căutare asincrone.

Plecând de la modelarea procesului de execuţie a agenţilor propusă în paragraful anterior, prin aplicarea celor metode de detecţie a terminării acestui proces se pot obţine două sisteme multi-agent utilizabile la implementarea şi evaluarea tehnicilor de căutare asincrone. Arhitecturile celor două sisteme distribuite aplicabile pentru tehnicile de căutare asincrone, sunt prezentate în figurile 5.21 şi 5.22.

Cele două sisteme sunt caracterizate prin câteva elemente comune. În primul rând simularea agenţilor este realizată în acelaşi mod, fiecare agent fiind simulat prin folosirea obiectelor de tip breed. În al doilea rând, agenţilor li se poate ataşa o rutină generală de tratare a mesajelor (în figurile 5.21 şi 5.22 handle-message-1/2). În al treilea rând, procedurile de tratare a mesajelor (care de obicei diferenţiază fiecare tehnică) sunt implementate în acelaşi mod pentru cele două sisteme.

Principala diferenţă dintre cele două sisteme este legată de detecţia terminării fiecărei tehnici implementate. Primul sistemul multi-agent foloseşte agentul sistem *observator*, iar cel de-al doilea sistem agenţii simulaţi prin obiecte de tipul *breed.*

5.4.1. Sistem multi-agent cu sincronizarea execuţiei agenţilor-SIES.

Primul sistem multi-agent de implementare şi evaluare se obţine prin aplicarea primei metode de detecţie a terminării tehnicilor de căutare asincrone. Acest sistem va fi numit *sistem de implementare şi evaluare cu sincronizare*-SIES. Ideea principală este aceea a utilizării agentului sistem *observator* în identificarea pauzei în transmiterea mesajelor. Acest lucru este realizat prin ataşarea rutinei update la un buton ataşat observatorului.

Al doilea element care diferenţiază acest sistem este acela al utilizării comenzii ask pentru execuţia rutinei handle-message a fiecărui agent (în rutina update). Acestă soluţie duce la sincronizarea execuţei agenţilor.

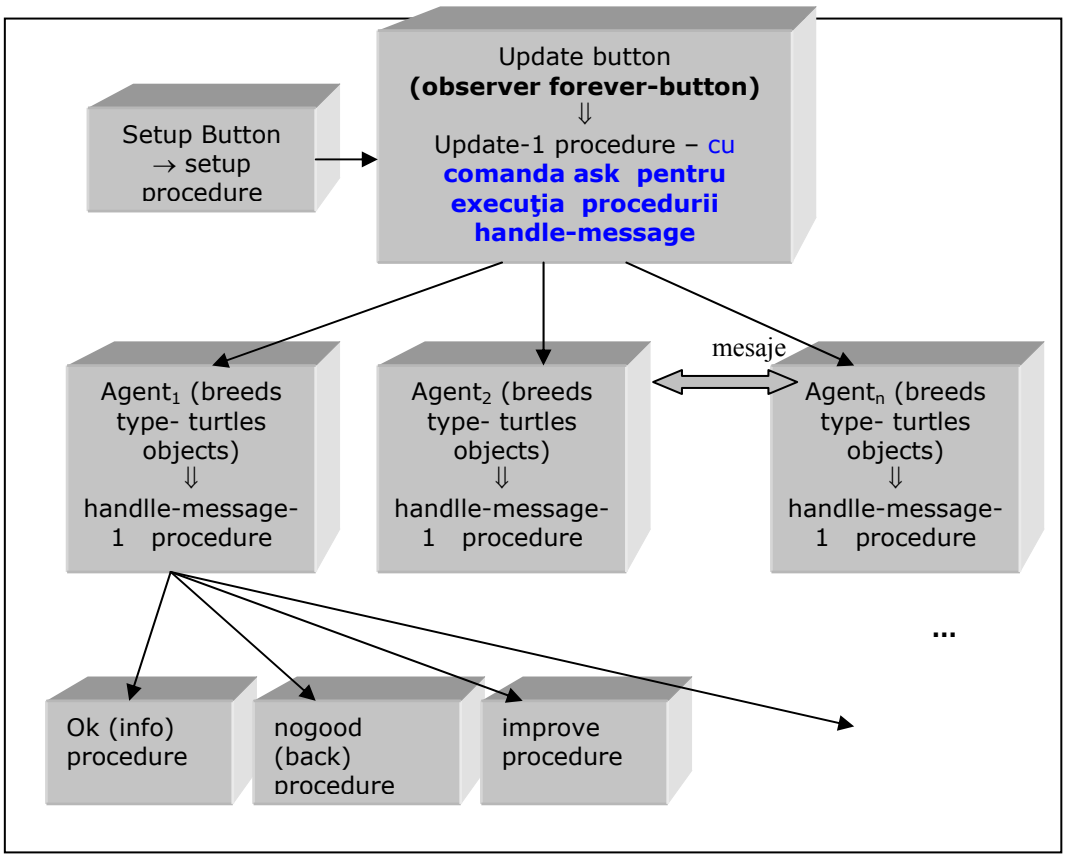


Figura 5.21. Arhitectura unui sistem multi-agent cu sincronizarea execuţiei- SIES

În figura 5.21 este prezentată arhitectura acestui sistem multi-agent, sistem ce va fi folosit în capitolele următoare la implementarea şi evaluarea mai multor variante de tehnici de căutare asincrone.

În arhitectura de mai sus, observatorul este cel ce iniţiază procedura de oprire a rulării algoritmului.

5.4.2. Sistem multi-agent cu operarea asincronă a agenţilor-SIEAS

Al doilea sistem multi-agent se obţine prin aplicarea celei de-a doua metode de detecţie a terminării execuţiei agenţilor la modelul de implementare şi evaluare propus anterior. Acest sistem va fi numit *sistem de implementare şi evaluare cu operarea asincronă a agenţilor*-SIEAS. Sistemul multi-agent se remarcă prin renunţarea la observator şi ataşarea rutinei update fiecărui agent de tip breed. În felul acesta execuţia rutinei handlle-message nu se mai face prin intermediul comenzii ask, aceasta fiind executată pentru fiecare agent (turtle) de către butonul de tip turtle cu proprietatea „forever-button” setată. Renunţarea la comanda ask are ca efect rularea asincronă a agenţilor, fiecare agent prelucrând mesajele sale fără a aştepta o sincronizare cu ceilalţi agenţi.

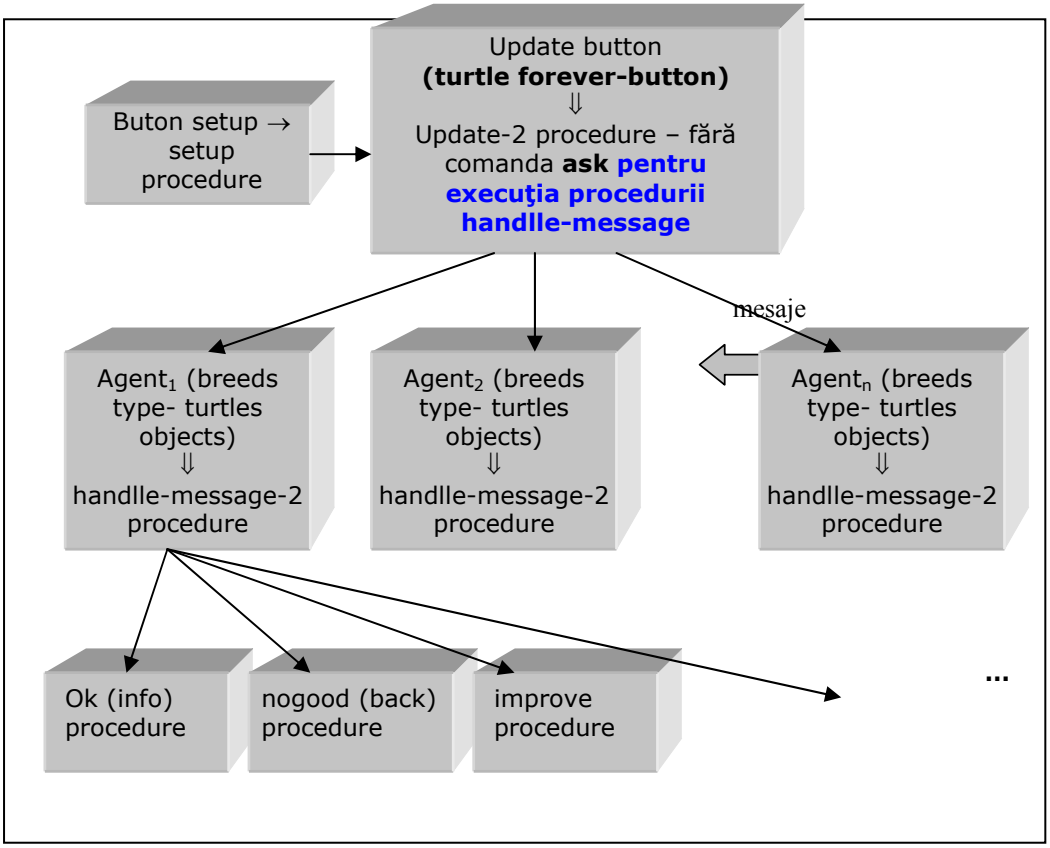


Figura 5.22. Arhitectura unui sistem multi-agent cu operarea asincronă a agenţilor-SEIAS

5.4.3. Diferenţa dintre un sistem cu sincronizare şi unul complet asincron.

În mod obişnuit, cele mai multe dintre simulatoare rulează mai multe fire de execuţie pe acelaşi sistem de calcul, câte un fir de execuţie pentru fiecare agent. Aplicaţia rulează în mai multe cicluri, un ciclu constând în aceea perioadă necesară pentru ca fiecare agent să recepţioneze mesaje, să execute calculele locale şi să trimită mesaje. Practic, după fiecare ciclu are loc o sincronizare a agenţilor, se aşteaptă ca fiecare să termine ciclul său. Un altfel de comportament este definit ca un sistem distribuit sincron în [HY00].

Acest lucru se întâlneşte şi în NetLogo. Un apel al procedurii principale, numită în această lucrare *update,* corespunde noţiunii de ciclu. Pentru ca această rutină să ruleze până la determinarea soluţiei sau inexistenţei acesteia, ea este ataşată la un buton de tip "Observer" cu proprietatea „forever-button" setată. În principiu, ea cere prin intermediul comenzii ask, manipularea mesajelor din cozile de mesaje. Fiecare agent, concurent şi asincron, extrage, trimite mesaje şi execută calculele sale locale, dar comanda ask, face la final o sincronizare. Acest mecanism

permite definirea unui comportament de sistem distribuit sincron şi este obţinut pentru primul sistem multi-agent.

Acest sistem se poate transforma într-un model care să funcţioneze cât mai asincron, fără a fi nevoie să se extindă NetLogo prin module Java. Soluţia este dată de folosirea celei de-a doua soluţii de detectare a terminării algoritmului şi de folosirea unor butoane ataşate obiectelor de tip turtle, butoane ce nu mai sunt ataşate observatorului. În acelaşi timp se renunţă la folosirea comenzii ask pentru a executa rutina de manipulare a mesajelor, aceasta fiind executată pentru fiecare agent (turtle) de către butonul de tip turtle „forever-button". Aceste elemente conduc la arhitectura prezentată în figura 5.22.

5.5. Metodologie de implementare şi evaluare pentru tehnicile asincrone

În acest paragraf se prezintă o metodologie de implementare a tehnicilor de căutare asincrone în NetLogo, folosind modelul prezentat în paragraful anterior 5.3. Această metodologie presupune identificarea obiectelor aplicaţiei, construirea agenţilor şi a suprafeţei de lucru pentru aplicaţie. De asemenea, sunt construite canalele de comunicaţie dintre agenţi, rutinele de manipulare a mesajelor şi programul principal al aplicaţiei. Metodologia conţine mai multe elemente specifice NetLogo necesare pentru finalizarea implementării tehnicilor asincrone de căutare. Orice implementare realizată pe baza modelului prezentat, va necesita parcurgerea următorilor paşi.

P1. Definirea obiectelor aplicaţiei DCSP.

Plecând de la tipul de problemă ce se implementează se vor defini obiectele aplicaţiei DCSP. În figura 5.23 se prezintă o soluţie de modelare a agenţilor şi a suprafeţei de lucru a aplicaţiei. Ca şi exemple de modelare se propun *breeds [queens]* (pentru modelarea agenţilor asociaţi reginelor din problema celor n regine) sau *breeds [vertices]* (pentru modelarea agenţilor asociaţi fiecărui nod din problema colorării grafurilor).

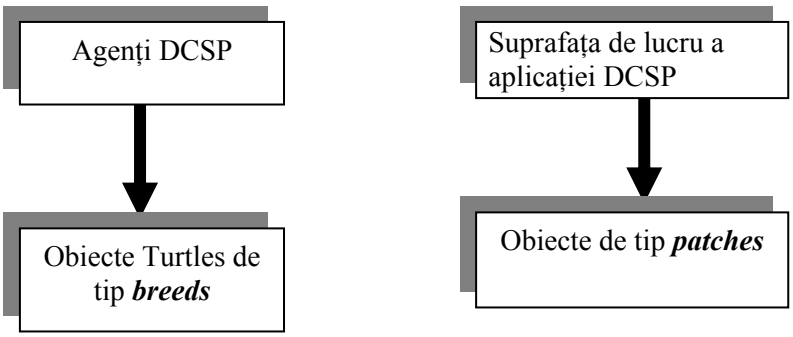
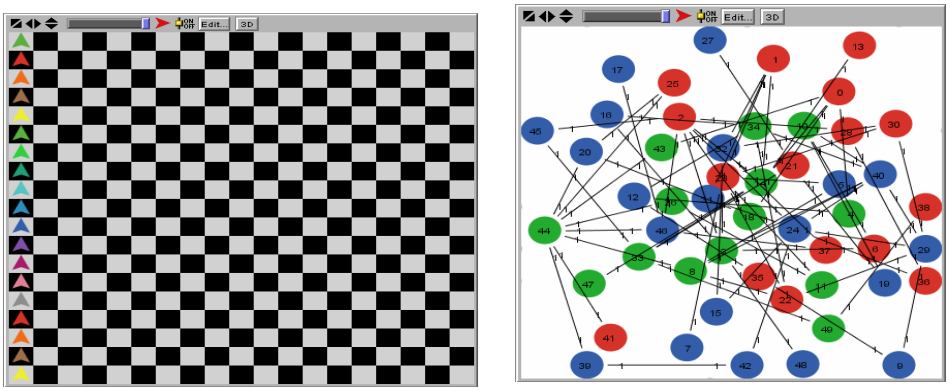


Figura 5.23. Identificarea obiectelor aplicaţiei DCSP

În schimb, pentru modelarea suprafeţei aplicaţiei sunt folosite obiecte de tip *patches.* În funcţie de ce reprezintă aceşti agenţi ei sunt reprezenaţi pe suprafaţa Netlogo. În figura 5.24 sunt prezentate două modalităţi NetLogo de reprezentare a agenţilor de tip *regine*, respectiv *noduri*.



(a) regine (b) noduri

Figura 5.24. Exemple de reprezentare a agenţilor pe suprafaţa NetLogo

P2. Manipularea mesajelor. Canalul de tip FIFO pentru mesaje.

Orice agent păstrează contextul de lucru sub forma a cel puţin două structuri proprietare: *current_view* şi lista sa *nogood*. Acest context este folosit pentru a lua decizii, inclusiv la construirea mesajelor. Pentru modelul propus, structurile de date ce memorează contextul de lucru al fiecărui agent se pot simula prin intermediul unor liste. O soluţie este prezentată în figura 5.25 (a)

Manipularea mesajelor presupune în primul rând reprezentarea acestora. Acest lucru se poate realiza prin utilizarea unor liste indexate. Pentru repezentarea mesajelor complexe, ce conţin foarte multe informaţii, Netlogo permite folosirea listelor de liste (elementele listei sunt de asemenea liste). În figura 5.25.(b) este prezentant modul de reprezentare a principalelor mesaje întâlnite la tehnicile asincrone. Simularea cozilor de mesaje pentru fiecare agent se poate face folosind liste Netlogo, pentru care se definesc rutine de tratare corespunzătoare principiilor FIFO (figura 5.25.(c)). Aceste structuri păstrează mesajele recepţionate de agent.

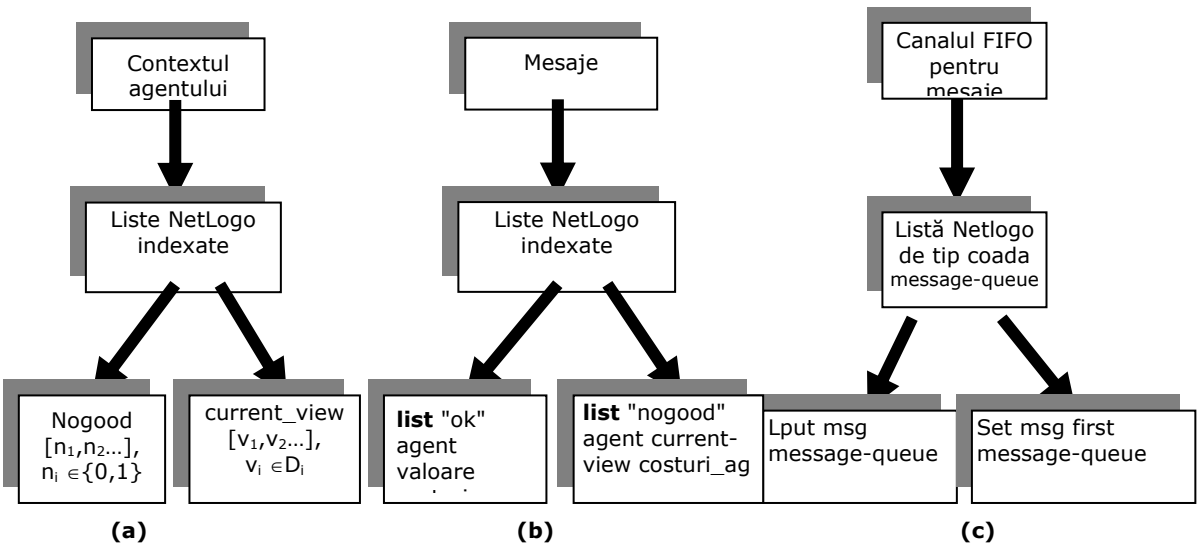


Figura 5.25. Structurile necesare manipulării mesajelor

Plecând de la elementele NetLogo prezentate în figura 5.25, se pot construi trei proceduri de manipulare a mesajelor din coada de mesaje, rutine prezentate în figura 5.26.

```
to receive-message [msg]
  without-interruption [
    set message-queue lput msg
message-queue]
end
```

```
to-report retrieve-message
   without-interruption [
    set msg first message-queue
    set message-queue butfirst
message-queue]
  report msg
```

```
to handle-message
  if (empty? message-queue) [stop]
  set msg retrieve-message
  if (first msg = "ok")[
  ok msg ]
  . . .
end
```

Figure 5.26. Manipularea mesajelor.

Prima rutină *receive-message* (figura 5.26) este utilizată pentru recepţionarea unui nou mesaj. A doua rutină *retrieve-message* are ca scop extragerea unui mesaj din coada de aşteptare, ea fiind apelată în rutina de tratare a mesajelor.
Ultima rutină (prezentată şi în paragraful 5.3.) *handle-message* identifică tipul de mesaj, apelând rutina corespunzătoare de tratare a acestuia.

P3. Iniţializarea aplicaţiei şi a fiecărui agent. "Programul principal" al aplicaţiei.

Iniţializarea aplicaţiei presupune construirea agenţilor şi a suprafeţei de rulare pentru aceştia. Odată cu construirea agenţilor se fac şi iniţializările necesare. În mod obişnuit se iniţializează contextul de lucru al agentului (current-view), cozile de mesaje, variabile ce contorizează efortul depus de agent.

În figura 5.27 sunt prezentate cele două rutine de iniţializare a aplicaţiei şi de iniţializare a agenţilor.

Suprafaţa aplicaţie trebuie să conţină obiecte NetLogo prin intermediul cărora să se controleze parametrii de definire a fiecărei probleme: numărul de agenţi, densitatea grafului constrângerilor, numărul de culori. Pentru aceasta se pot folosi obiecte Netlogo de tip *slider.* Aceste obiecte permit definirea şi monitorizarea parametrilor fiecărei probleme.

Pentru rularea aplicaţiei se propune introducerea unui obiect grafic de tip button şi setarea proprietăţii *forever*. În felul acesta, codul ataşat sub forma unei proceduri NetLogo (ce se aplică la fiecare agent) va rula continuu, până la golirea cozilor de mesaje şi întâlnirea comenzii *Stop* (care în NetLogo opreşte execuţia unui agent). Soluţia prezentată în figura 5.28 se bazează pe utilizarea comenzii *ask*. Această comandă NetLogo execută o sincronizare a execuţiei fiecărui agent.

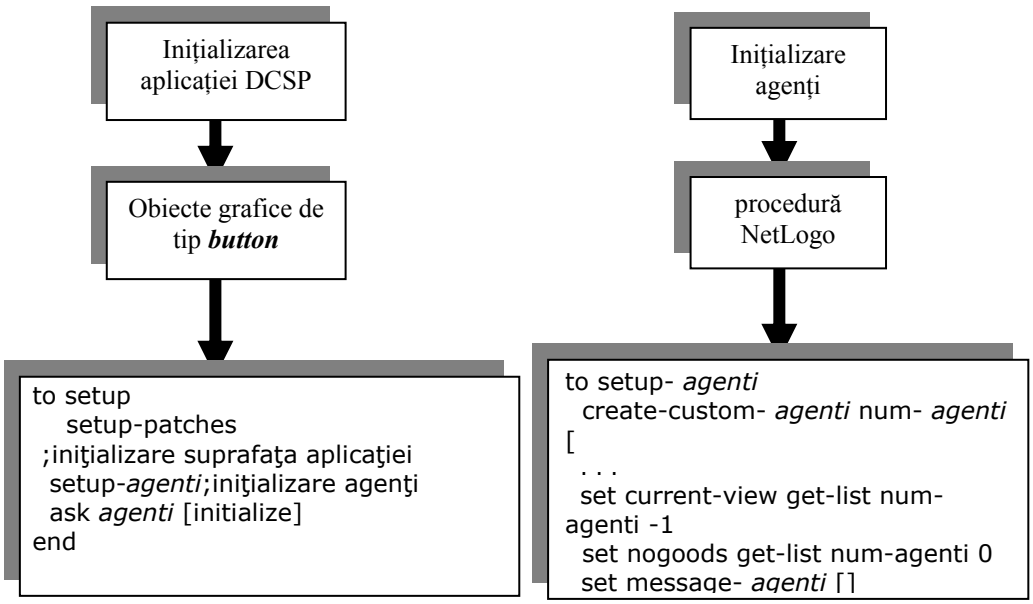


Figura 5.27. Iniţializarea aplicaţiei DCSP

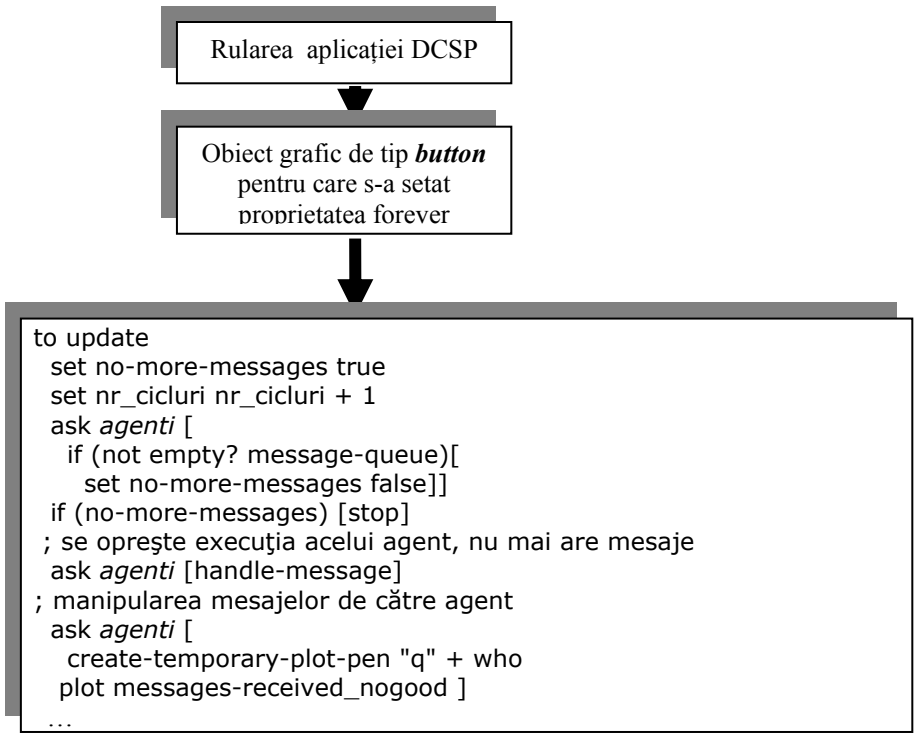


Figura 5.28. Procedura de rulare a aplicaţiei DCSP pentru sistemul SEIS.

O altă observaţie importantă este legată de ataşarea butonului grafic *observatorului* (figura 5.28). Utilizarea acestei soluţii permite obţinerea unei soluţii de implementare cu sincronizarea execuţiei agenţilor. În acest caz, *observatorul* va fi cel care va iniţia oprirea execuţiei aplicaţiei DCSP. Acest lucru va fi realizat în momentul în care primeşte comanda *stop* de la fiecare agent. În figura 5.28 procedura *update* este ataşată şi manipulată de *observator.* Aceste elemente conduc la sistemul multi-agent cu sincronizarea execuţiei agenţilor.

Dacă se doreşte obţinerea unui sistem cu operarea asincronă se va utiliza cea de-a doua metodă de detecţie ce presupune o altă rutină update. Această nouă rutină update va fi ataşată la un obiect grafic de tip *buton* ce este ataşat si manipulat de agenţii de tip turtle (agenţii simulaţi erau de tip breed ce reprezintă un caz particular de turtle). O astfel de soluţie este prezentată în figura 5.29.

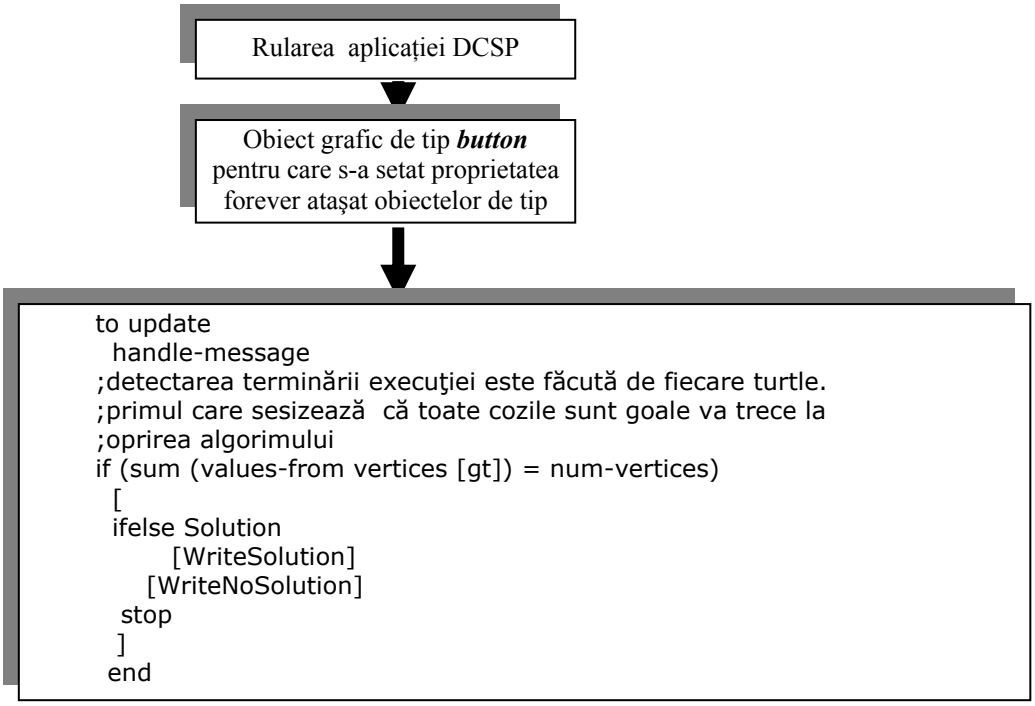


Figura 5.29. Procedura de rulare a aplicaţiei DCSP pentru sistemul SIEAS.

P4. Monitorizarea parametrilor de evaluare .

Modelul prezentat în acest capitol permite contorizarea diverselor costuri pentru obţinerea soluţiei de către tehnicile de căutare asincrone. Acest lucru se poate face prin folosirea unor variabile globale ataşate agenţilor. De exemplu, pentru contorizarea fluxului de mesaje principal se poate folosi câte o variabilă proprietară fiecărui agent (*messages-received_nogood şi messages-received_ok)*, variabilă ce trebuie incrementată în momentul generării şi trimiterii unei mesaj. Această variabilă este incrementată în rutina de manipulare a mesajelor *handle-message***.** De asemenea, pentru contorizarea efortului de lucru depus de agenţi se folosesc două variabile nr_constraintc şi c-ccks. Aceste variabile reţin costurile

necesare pentru fiecare agent. Prin urmare, vor trebui totalizate aceste costuri. Un exemplu este prezentat în figura 5.30.

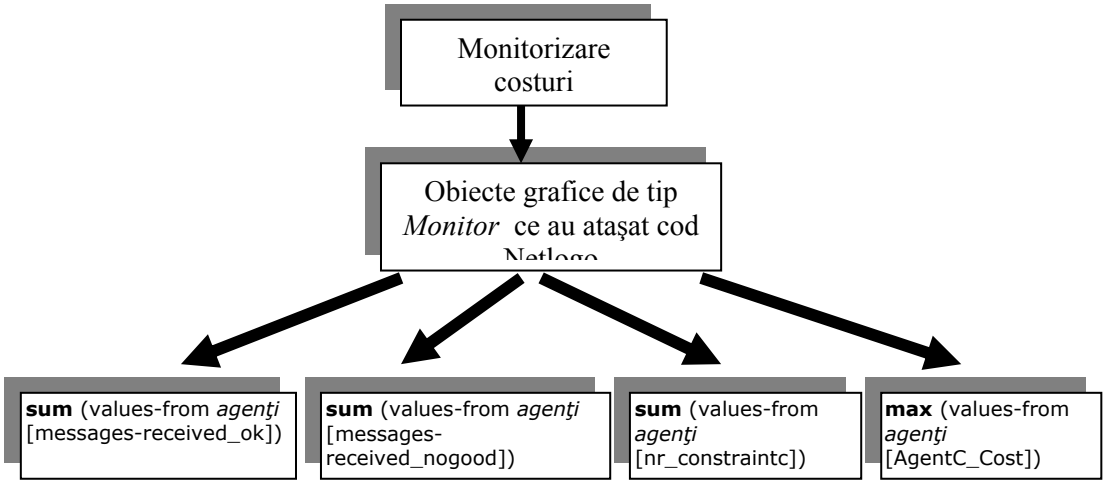


Figura 5.30. Monitorizarea parametrilor de evaluare pentru cele două sisteme

Aplicarea metodologiei prezentate anterior permite implementarea şi evaluarea oricărei tehnici de căutare asincrone.

În figura 5.31 este prezentată captura unei implementări pentru tehnica AWCS ce foloseşte sistemul multi-agent SIEAS.

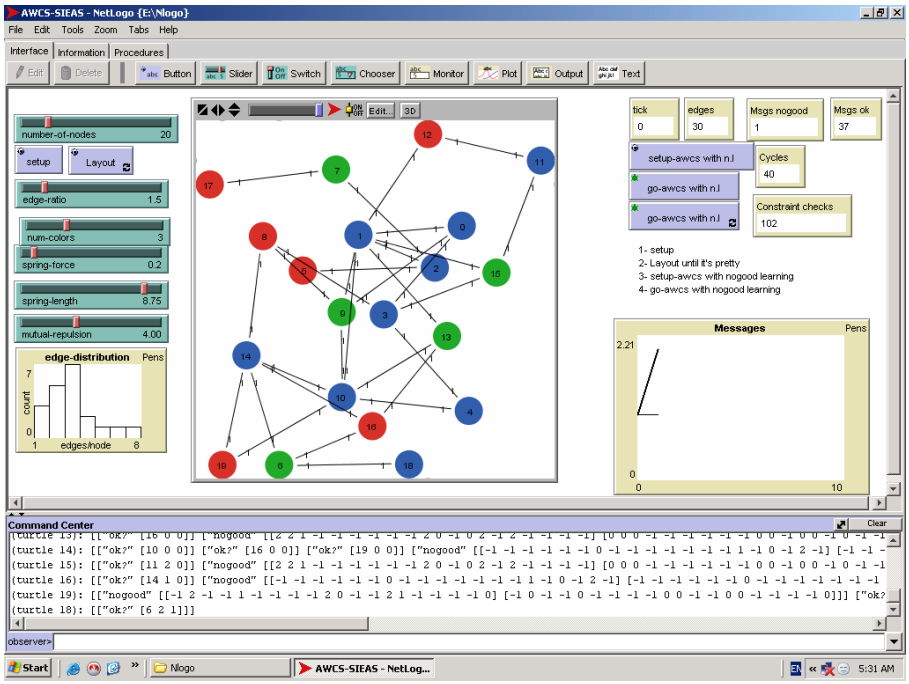


Figura 5.31. Sistemul multi-agent SIEAS utilizat la implementarea tehnicii AWCS

5.7. Concluzii.

În acest capitol, pornind de la facilităţile oferite de mediul NetLogo, se propune construirea unui model general de implementare şi evaluare pentru tehnicile asincrone, model care utilizează mediul NetLogo ca suport de simulare în studiul tehnicilor asincrone.

În consecinţă s-a propus un model general de implementare şi evaluare pentru tehnicile asincrone. Modelul propus presupune identificarea obiectelor NetLogo necesare pentru implementarea tehnicilor asincrone (agenţi, mesaje, cozi de mesaje, ordinea agenţilor) şi a interfeţei de interacţiune cu utilizatorul. S-au propus soluţii pentru simularea obiectelor oricărei aplicaţii DCSP. De asemenea, s-au propus soluţii de contorizare a costurilor obţinerii unei soluţii utilizând diverse metrici. Acest lucru permite evaluarea performanţelor tehnicilor asincrone şi a eventualelor îmbunătăţiri ale acestora. Se poate considera că modelul propus este primul model în NetLogo care permite implementarea şi evaluarea tehnicilor asincrone după mai multe criterii de evaluare. De asemenea, modelul permite studierea comportamentului agenţilor pentru diverse tehnici, studierea costurilor fiecărui agent.

Sunt propuse două metode de detecţie a terminării algoritmilor, metode ce permit construirea a două sisteme multi-agent (SEIS-sistem de implementare şi evaluare cu sincronizarea execuţiei agenţilor şi SEIAS-sistem de implementare şi evaluare cu operarea asincronă a agenţilor) pentru evaluarea corectă a performanţelor tehnicilor asincrone. Este posibil să se facă comparaţii pentru tehnicile asincrone, pentru a vedea care variantă de implementare este de preferat: sincron sau complet asincron. Se poate studia comportamentul agenţilor în diverse situaţii şi identificarea unor posibile îmbunătăţiri ale performanţelor tehnicilor asincrone.

O altă facilitate foarte importantă, este legată de faptul că sistemul NetLogo, prin intermediul sistemelor multi-agent definite, permite şi simularea altor situaţii din practică, cum ar fi apariţia întârzierilor în transmisia mesajelor. Se pot introduce pauze în prelucrarea mesajelor prin intermediul rutinelor handle-message şi studia comportamentul agenţilor în cazul apariţiilor de întârzieri în furnizarea mesajelor. De asemenea, se pot simula situaţii în care este nevoie de controlul canalelor de comunicaţii, cum ar fi filtrarea mesajelor (folosită în capitolele următoare).

Aceste două sisteme multi-agent vor fi utilizate în capitolele următoare la implementarea şi identificarea unor îmbunătăţiri pentru câteva din tehnicile asincrone.

Ca o concluzie generală, se poate afirma că modelul realizat împreună cu cele două sisteme multi-agent pot fi folosite pentru studiul şi analiza tehnicilor asincrone.

Elementele evidenţiate în cadrul acestui capitol au fost publicate în [MusBP05] şi [MusJP06].

6. STUDIU COMPARATIV AL TEHNICILOR DE CĂUTARE ASINCRONE

Complexitatea tehnicilor asincrone este o temă de studiu foarte importantă pentru cei care realizează cercetări în acest domeniu. Datorită comunicaţiilor asincrone, modelul distribuit este unul non-determinist. Într-adevăr, un program distribuit poate avea rulări diferite pentru acelaşi set de date, în funcţie de programarea evenimentelor. Prin urmare, trebuie analizate foarte multe încercări, trebuie verificat cazul cel mai nefavorabil peste toate aceste posibile scheme.

În această capitol este prezentat un studiu comparativ pentru cele mai importante tehnici asincrone, studiu experimental bazat pe implementarea şi evaluarea acestor tehnici folosind cele două modele prezentate anterior. Scopul principal al acestui studiu este de a studia comportamentul agenţilor în cele două situaţii de implementare: cu sincronizarea execuţiei agenţilor şi execuţia asincronă a agenţilor. De asemenea, sunt analizate comparativ performanţele tehnicilor din cele două familii identificându-se pentru fiecare clasă modelul potrivit de implementare şi evaluare, în funcţie de costurile de obţinere a soluţiei. Studiul investighează eficienţa tehnicilor asincrone, în raport cu fluxul de mesaje schimbate de agenţi, numărul de ciclii consumaţi pentru găsirea soluţiei şi constrângerile verificate, comparativ pentru cele două modele prezentate anterior.

În literatura de specialitate nu există referinţe despre studii care să investigheze comparativ comportamentul tehnicilor asincrone în cele două situaţii. Prin urmare, este interesant de investigat oportunitatea sincronizarii executiei agenţilor in cazul tehnicilor asincrone. Sunt investigate comportamentele mai multor tehnici asincrone în doua cazuri: agentii executa asincron prelucrarea mesajelor recepţionate (situaţia reală din practică) si cazul sincron, în care se face o sincronizarea a executiei agentilor. Cu alte cuvinte, în cazul sincron agenţii executa un ciclu de calcul în care prelucrează un mesaj din coada de mesaje (dacă există), după care se face o sincronizarea aşteptându-se finalizarea de către ceilalţi agenţi a prelucrării mesajelor lor.

Plecând de la comportamentul acestor tehnici, în acest capitol sunt propuse mai multe soluţii de îmbunătăţire a performanţelor prin introducerea sincronizării parţiale a execuţiei agenţilor şi introducerea managementului mesajelor pentru aceste tehnici.

În partea de început a capitolului se definesc condiţiile experimentale în care s-au desfăşurat experimentele, apoi se prezintă rezultatele experimentale pentru două mari familii asincrone: familia ABT şi familia AWCS (pe baza celor două modele). Pe baza acestor experimente, se propune o soluţie de introducere a sincronizării execuţiei agenţilor, soluţie ce permite reducerea costurilor în căutarea soluţiei. Apoi, este studiată redundanţa mesajelor şi sunt oferite mai multe soluţii de eliminare sau reducere a acesteia.

Acest propuneri permit îmbunătăţirea performanţelor pentru câteva din tehnicile de căutare asincrone analizate.

6.1. Condiţii experimentale.

În acest paragraf sunt prezentate condiţiile experimentale utilizate în această teză la realizarea tuturor experimentelor, pe baza modelului prezentat anterior în capitolul 5. Cele două sisteme multi-agent prezentate în capitolul 5 au permis implementarea tehnicilor asincrone şi studiul acestor tehnici în diferite situaţii. Pe baza acestor studii, ce vor fi prezentate în capitolele următoare, s-au identificat anumite îmbunătăţiri.

Evaluarea performanţelor tehnicilor asincrone presupune utilizarea unor unităţi de măsură care să asigure o anumită independenţă faţă de limbajele de programare folosite la implementarea acestora. Aceste unităţi de măsură permit evaluarea tehnicilor asincrone din mai multe puncte de vedere, cum ar fi efortul local şi efortul global depus de agenţi, încărcarea reţelei datorată fluxului de mesaje schimbat [MusPO04]. În paragraful 3.3.2. au fost prezentate mai multe unităţi de măsură ce pot fi folosite la evaluarea tehnicilor asincrone.

Îmbunătăţirile propuse în capitolele următoare au fost evaluate folosind anumite unităţi de măsură dintre cele prezentate în 3.3.2. Pentru a face evaluarea performanţelor tehnicilor asincrone analizate, aceste tehnici au fost implementate în mediul NetLogo 3.0.2, folosind limbajul NetLogo ([Wil99], [MAS Netlogo Models-a,b]) pe baza modelului prezentat în capitolul 5. În cele mai multe situaţii s-a utilizat sistemul multi-agent SEIS, acesta permiţând şi contorizarea ciclurilor.

Suportul de calcul a fost dat de o reţea client-server bazată pe Red Hat Linux, formată dintr-un server IBM şi 14 staţii de lucru. Platforma NetLogo împreună cu cele două sisteme SEIS şi SEIAS a fost instalată pe fiecare dintre calculatoarele din reţea. Fiecare calculator a permis rulări pentru anumite tehnici, în anumite condiţii, selectate pentru fiecare experiment.

Tehnicile asincrone au fost aplicate pentru problema colorării unui graf în varianta distribuită. Pentru problema colorării grafurilor s-au considerat patru tipuri de probleme definite ca în [MJPL92] în funcţie de parametrii n–numărul de noduri/agenţi, k=3 culori şi m- numărul de legături între agenţi.

În funcţie de densitatea grafului de constrângeri (parametrul m) s-au utilizat 2 clase de probleme:

- grafuri cu puţine legături, numite *sparse problems,* având *m=n x 2 legături.*
- grafuri cu un număr special de legături, numite *dificult problems,* având *m=n x 2.7 legături.*

Pentru fiecare clasă de problemă s-au generat aleator câte 10 grafuri (având densităţile precizate anterior) cu dimensiunile n=15, n=20, n=30 şi n=40 noduri. La fiecare generare s-au reţinut doar grafurile conexe, în caz contrar generându-se alt graf. Practic, în funcţie de dimensiune s-au selectat două categorii de probleme:

- dimensiune mică n=15, n=20 noduri/agenţi.
- dimensiune mare n=30, n=40 noduri/agenţi.

Astfel s-au obţinut patru clase de probleme, variind în funcţie de dimensiune şi densitate.

Pentru fiecare variantă s-au executat un număr de 100 încercări, reţinându-se media valorilor măsurate. Cu alte cuvinte, pentru fiecare din cele patru clase probleme, au fost generate aleator 10 grafuri cu densităţile precizate anterior, pentru fiecare graf generându-se aleator 10 valori iniţiale. Fiecare dintre versiunile de tehnici obţinute au fost evaluate cu aceleaşi valori iniţiale.

Pentru a putea face evaluarea tehnicilor asincrone de căutare (variantele de bază şi variantele cu îmbunătăţiri), pentru fiecare rulare s-au selectat mai mulţi parametrii dintre cei prezentaţi în paragraful §4.1.2, după cum urmează:

- **Numărul total de mesaje-***TM***.** Acest parametru permite evaluarea încărcării reţelei. Contorizarea fluxului de mesaje a constat în numărarea mesajelor de tip ok/info şi nogood/back schimbate de agenţi. Folosind modelul de implementare propus, la fiecare agent s-a reţinut numărul de mesaje schimbate. În final, s-a totalizat cantitatea de mesaje de tip ok/info (Tok), respectiv nogood/back(Tnogood), reţinută la fiecare agent.
- **Numărul total de valori nogood stocate-***TNGS*. Această unitate de măsură permite evaluarea algoritmilor asincroni din punct de vedere al eficienţei mesajelor nogood. La fiecare agent s-a reţinut numărul de mesaje nogood stocate, în final totalizându-se cantitatea de mesaje de nogood/back, reţinută la fiecare agent.
- **Numărul total de constrângeri-***TCcks*. Acest parametru o măsură a timpului global consumat de agenţii implicaţi. Pentru fiecare agent s-a reţinut numărul de constrângeri verificate. Practic s-a evaluat pentru fiecare agent efortul local depus de el. În final s-a totalizat această cantitate de constrângeri verificate.
- **Numărul de constrângeri concurente verificate**-*TC-ccks*. Pentru fiecare agent s-a reţinut numărul de constrângeri concurente verificate. Pentru a obţine acest număr s-a aplicat celor două sisteme SEIS şi SEIAS algoritmul din [MKRZ02]. În final s-a totalizat cantitate de constrângeri concurente verificate.
- **Numărul de ciclii necesari pentru obţinerea soluţiei**-*Tciclii*. Pentru cazul utilizării sistemului SEIS s-au numărat ciclii neceari obţinerii soluţiei, practic numărul de apeluri pentru procedura *update*.

6.2. Analiza experimentală a tehnicilor din familia ABT.

În acest paragraf sunt evaluate tehnicile din familia ABT. Cu alte cuvinte, sunt evaluate cele două mari tehnici derivate din nucleul ABT kernel, prin eliminarea informaţiilor învechite dintre agenţi:

- tehnica ABT - obţinută prin adăugarea legăturilor între agenţii neconectaţi, legături devenite permanente.
- tehnica DisDB – care se remarcă prin faptul că nu necesită legături suplimentare. Informaţiile învechite dintre agenţi sunt eliminate în timp finit. Pentru aceasta agenţii care execută backtrack, uită toate nogoodurile ce pot conţine ipotetic valori învechite.

Fiecare din variantele ABT şi DisDB a fost implementată apelând la cele modelul prezentat în capitolul anterior, fiecare agent tratând secvenţial un singur mesaj din coada de mesaje, la un ciclu de calcul. S-au utilizat cele două sisteme de implementare şi evaluare, obţinându-se două tipuri de rulări :

- versiuni implementate în sistemul SEIS, notate cu ABT_1 şi $DisDB_1$. Practic s-a rulat în condiţiile sincronizării execuţiei agenţilor.
- versiuni implementate în sistemul SEIAS, notate cu ABT_2 şi $DisDB_2$. Practic s-a rulat în condiţiile operării asincrone a agenţilor.

În tabelul 6.1 sunt prezentate valorile parametrilor prezentaţi anterior, valori obţinute pentru cele patru versiuni. În tabelul 6.1 s-a reţinut cantitatea totală de mesaje ok şi nogood, şi cantitaţile de constrângeri verificate. Nu s-a contorizat numărul de ciclii.

		n=20		n=30	
		m=n*2	m=n * 2.7	m=n*2	m=n*2.7
ABT_1	TNogood	336.23	481.88	3883.12	3936.20
	TOk	580.45	1615.48	8578.23	12343.88
	Tcck	23176.09	80834.37	957934.98	938790.08
	Tc-ccks	6543.07	16572.45	147128.29	123501.58
ABT_2	TNogood	203.15	366.94	3181.57	2196.52
	TOk	343.72	1438.48	6518.74	9943.55
	Tcck	17417.20	69332.91	763992.16	698240.00
	Tc-ccks	4468.28	7593.00	90601.22	45264.01
$DisDB_1$	TNogood	138.76	316.94	2803.34	3104.44
	TOk	300.04	722.89	6855.19	7235.88
	Tcck	18548.68	40841.66	603288.94	778016.08
	Tc-ccks	4526.52	8112.34	124728.67	154409.43
$DisDB_2$	TNogood	83.10	216.41	2785.65	2983.69
	TOk	213.72	556.03	6518.74	6878.05
	Tcck	12117.12	29850.88	585992.16	655243.39
	Tc-ccks	2468.48	3963.02	70605.97	82785.79

Tabelul 6.1. Rezultatele experimentale pentru versiunile ABT şi DisDB

Analizând valorile din tabelul 6.1. se remarcă faptul că varintele în care agenţii operează asincron au necesitat costuri mai mici relativ la fluxul de mesaje, dar şi relativ la efortul depus de agenţi (constrângeri verificate). Variantele cu sincronizarea execuţiei, indiferent de tehnica ABT sau DisDB, au necesitat costuri mult mai mari. Acest lucru arată că sincronizarea nu este recomandată în cazul acestor tehnici bazate pe o ordine statică a agenţilor.

Este posibil să existe o legătură între ordinea statică a agenţilor şi sincronizarea execuţiei agenţilor. Practic, aceste tehnici ce folosesc o ordine statică, au necesitat costuri globale mult mai mari în cazul sincronizării, relativ la cazul asincron.

6.3. Analiza experimentală a tehnicilor din familia AWCS.

În acest paragraf sunt analizate rezultatele experimentale pentru tehnicile din familia AWCS. În acest studiu s-a implementat o varianta de bază introdusă în [Yok98] care s-a aplicat la tehnica nogood learning din [Hiro2000], obţinându-se o variantă îmbunătăţită (notată cu AWCS-nl). Pentru fiecare variantă s-au implementat câte două versiuni corespunzător celor două modele obţinute:

- varianta de bază AWCS [YDIK98] cu sincronizarea execuţiei agenţilor: $AWCS\text{-}nl_1$. S-a utilizat sistemul SEIS.
- varianta de bază AWCS [YDIK98] cu operarea sincronă a agenţilor: $AWCS\text{-}nl_2$. S-a utilizat sistemul SEIAS.

Aşa cum se ştie, constrângerile verificate evaluează efortul local depus de fiecare agent, dar numărul de constrângeri concurente verificate permite evaluarea efortului fără a lua în calcul faptul că agenţii lucrează concurent. Analizând rezultatele din tabelul 6.2, se remarcă faptul că sincronizarea execuţiei agenţilor a redus efortul local. În schimb, odată cu creşterea dimensiunii problemelor (40 noduri), varianta asincronă AWCS-nl_2 a necesitat eforturi mult mai mari faţă de varianta cu sincronizare. Diferenţe mari de efort local apar mai ales pentru problemele de densitate mare (cunoscute şi sub numele de probleme dificile).

		n = 30		n = 40	
		m=n x2	m=n x2.7	m=n x2	m=n x2.7
AWCS-nl_1	TNogood	575.84	3639.52	713.60	24652.61
	TOk	1864.50	7624.27	2403.25	51132.39
	Tcck	2913.94	14482.23	3747.34	102388.82
	Tc-ccks	802.74	4351.37	824.10	21888.16
AWCS-nl_2	TNogood	502.77	2181.62	733.93	23259.95
	TOk	1625.23	23964.88	2422.19	85845.00
	Tcck	2628.08	45813.89	4163.19	387355.71
	Tc-ccks	662.13	2512.22	793.45	15099.00

Tabelul 6.2. Rezultatele experimentale pentru versiunile AWCS

Se remarcă totuşi, că pentru probleme de densitate rară (m=n x 2.0) variantele asincrone au avut costuri apropiate de variantele sincrone, chiar eforturi mai mici. Acelaşi comportament a fost întâlnit şi relativ la fluxul de mesaje, varianta cu sincronizarea execuţiei a necesitat un flux de mesaje mai mic decât varianta asincronă.

6.4. Analiza experimentală a tehnicii DB.

În acest paragraf sunt analizate rezultatele experimentale pentru tehnica Distributed Breakout. În acest studiu s-a implementat varianta revizuită din [Yok00]. Pentru această variantă s-au realizat câte două implementări corespunzător celor două sisteme MAS obţinute:

- varianta de bază DB [Yok00] cu sincronizarea execuţiei agenţilor: DB_1. S-a utilizat sistemul SEIS.
- varianta de bază DB [Yok00] cu operarea sincronă a agenţilor: DB_2. S-a utilizat sistemul SEIAS.

		n = 20		n = 30		n = 40	
		m=n x2	m=n x2.7	m=n x2	m=n x2.7	m=n x2	m=n x2.7
DB_1	TM	1868	2561	5458	8900	8962	20848
	Timprove	914	1259	2687	4412	4439	10369
	Tok	951	1299	2757	4484	4519	10475
	Tcck	1433	1873	4127	7883	6952	21820
DB_2	TM	1868	2564	5458	8903	8963	20852
	Timprove	916	1264	2701	4418	4445	10376
	Tok	951	1300	2757	4485	4518	10476
	Tcck	1449	1889	4160	7952	7003	21989

Tabelul 6.3. Rezultatele experimentale pentru versiunile DB

Tehnica DB are anumite particularităţi comparativ cu celellalte tehnici de căutare asincrone. Din acest motiv s-a contorizat numărul total de mesaje transmise (TM), numărul de mesaje de tip ok (Tok) şi numărul total de mesaje improve recepţionate (Timprove). De asemenea, pentru evaluarea efortului local s-a contorizat numărul total de constrângeri verificate *TCcks*.

Analizând rezultatele din tabelul 6.3, se remarcă faptul că cele două variante DB_1 şi DB au necesitat costuri apropiate în ceea ce priveşte obţinerea soluţiei. Indiferent de dimensiunea problemei şi densitatea grafului constrângerilor, sincronizarea execuţiei agenţilor nu a redus efortul local. Totuşi, se observă reduceri mici de costuri în obţinerea soluţiei (mai ales în ceea ce priveşte efortul local depus de agenţi), pentru cazul cu sincronizarea execuţiei agenţilor.

Spre deosebire de celelalte tehnici de cautare asincrone analizate, tehnica DB s-a comportat identic în cele două cazuri de rulare. O explicaţie este legată de modul de comportament al agenţilor. Fiecare agent rulează alternativ în două moduri de operare:modul *ok* şi modul *improve*.

6.5. Introducerea sincronizării în cazul tehnicilor de căutare asincrone.

Tehnicile de căutare asincrone au comportamente diferite în cazul apariţiei întărzierilor în furnizarea mesajelor. Acest lucru are ca efect comportamente diferite în cazul sincronizării execuţiei agenţilor. Analiza rezultatelor din paragrafele anterioare şi în special din paragraful §6.3 arată că sincronizarea poate conduce la o reducere a costurilor pentru anumite tehnici.

În literatura de specialitate nu există referinţe despre studii care să investigheze comparativ comportamentul tehnicilor asincrone în cele două situaţii şi nici soluţii de obţinere a acestei sincronizări. Experimentele anterioare au arătat că sincronizarea execuţiei agenţilor reduce costurile obţinerii soluţiei pentru anumite familii de tehnici de căutare asincrone. Plecând de la aceste observaţii, în acest paragraf sunt introduse două soluţii de obţinere a sincronizării exeuţiei agenţilor pentru toate tehnicile de căutare asincrone, dar în special pentru tehnicile ce folosesc o ordine dinamică pentru agenţii (familia AWCS).

Sistemul cu sincronizare SEIS, prezentat în capitolul 5 şi utilizat la evaluarea din paragrafele §6.2 şi 6.3, se bazează pe elementele Netlogo, utilizându-se comanda *ask* pentru execuţia procedurilor de tratare a mesajelor agenţilor. Această comandă face o sincronizare a comenzilor ataşate agenţilor astfel încât se obţine automat sincronizarea execuţiei agenţilor. În realitate, agenţii rulează concurent şi asincron, fiecare agent tratând mesajele sale din coada de mesaje în ordinea în care sosesc, fără a aştepta finalizarea calculelor de către ceilalţi agenţi. În paragrafele următoare sunt propuse două soluţii de sincronizare generale, care nu depind de un anume limbaj de programare.

6.5.1 Sincronizarea completă a execuţiei agenţilor.

Soluţia propusă presupune folosirea unei zone comune de memorie la care au acces agenţii şi în care să fie stocate anumite informaţii necesare tuturor agenţilor. În această zonă comună de memorie se stochează valoarea unei variabile globale *Nragents*, accesibilă tuturor agenţilor. Iniţial, această variabilă este iniţializată cu numărul de agenţi. Fiecare agent va marca starea execuţiei calculelor sale în variabila Nragents. Practic, fiecare agent A_i în momentul lansării rutinei de

tratare a mesajelor decrementează valoarea variabilei *Nragents* cu 1. În schimb, în momentul în care agentul finalizează de tratat mesajele din coada sa de mesaje (secvenţial sau în pachete) incrementează valoarea variabilei *Nragents* cu 1. Cu alte cuvinte variabila *Nragents* permite identificarea în orice moment a situaţiei execuţiei agenţilor. Dacă această variabilă, la un moment dat are ca valoare numărul de agenţii înseamnă că toţi agenţii au terminat de tratat mesajele din coada de mesaje. Acesta poate deveni un moment de sincronizare, care se poate retine prin intermediul unei variabile *Sincronizare*.

Implementarea oricărei tehnici asincrone presupune construirea unor rutine de execuţie a calculelor de către fiecare agent. În modelele de implementare propuse în capitolul 5 această rutină este denumită *update*. De asemenea, fiecare agent are nevoie de o altă rutină de tratare a mesajelor sale (fiecare tehnică diferă prin faptul că tratează diferit acele mesaje) numită în modelele din capitolul 5 *handle-message*. Pentru prezentarea acestei metode se vor păstra aceste notaţii folosite în capitolul 5 la prezentarea celor două sistem multi-agent.

În figura 6.1 este prezentată nouă rutină *update-3* ce este executată de fiecare agent. Fiecare agent verifică dacă s-a atins starea de sincronizare, în caz afirmativ rulează propria procedură de manipulare a mesajelor (handle-message). Apoi, agentul intră în starea de aşteptare, până ce variabila *Nragents* devine egală cu numărul agenţilor, pentru a putea trece la un nou ciclu de execuţie a calculelor sale. În felul acesta se face o sincronizare a execuţiei agenţilor (marcată în figura 6.1 prin *Sincronizare* cu true).

```
to update
 if Sincronizare
 [
    handle-message
 ]
  ifelse Solution
  [  WriteSolution]
  [  WriteNoSolution]
   stop
  ]
  while [Nragents != number-of-agent]
  [
    wait
    set Sincronizare false
 ]
set Sincronizare true
end
```

Figura 6.1 Procedură update -3

În figura 6.2 este prezentată noua rutină de tratare a mesajelor *handle-message – 3,* pe care fiecare agent o apelează pentru prelucrarea mesajelor sale. În această rutină se observă faptul că agentul decrementează variabila globală *Nragents* la intrarea în procedură. În final, după ce agentul a tratat un mesaj sau mai multe mesaje (în pachete), se face incrementarea variabilei *Nragents.* Varianta din figura 6.2 prelucrează un singur mesaj la un ciclu.

În figura 6.2 se observă apelul celor două proceduri tipice de manipulare a mesajelor ok (info în cazul nucleului ABT) sau nogood (back în cazul nucleului ABT).

Pentru anumite tehnici pot fi apelate şi alte rutine de tratare a mesajelor (de exemplu pentru ABT tratarea mesajelor de tip add-link).

```
to handle-message
  set  Nragents  Nragents - 1
  if not empty? message-queue
  [
   set msg retrieve-message
   set gt 0
   if (first msg = "ok?")
    [
      handle-ok-message  msg
    ]
   if (first msg = "nogood")
    [
      handle-nogood-message  msg
    ]
  ]
  if (empty? message-queue)
  [
     set gt 1
     stop
 ]
  set  Nragents  Nragents  + 1
 end
```

Figura 6.2 Procedură handle-message–3

Cele două proceduri de manipulare a mesajelor se pot aplica în orice limbaj ales la implementare. În particular, aceste două rutine se pot aplica pentru modelul asincron propus în capitolul 5 obţinându-se o sincronizare a execuţiei agenţilor. Astfel, plecând de la elementele de implementare prezentate în capitolul 5 se poate obţine un al treilea model de implementare şi evaluare pentru tehnicile asincrone, in NetLogo.

6.5.2 Sincronizarea parţială a execuţiei agenţilor.

A doua soluţie propusă în acest capitol constă în sincronizarea agenţilor vecini. Spre deosebire de prima variantă, fiecare agent va aşteapta finalizarea calculelor agenţilor vecini cu care este conectat, aflaţi înaintea lui în ordinea lexicografică. Această soluţie permite o sincronizare parţială a execuţiei agenţilor. Ea nu necesită existenţa agentului central.

Această a doua soluţie de sincronizare parţială necesită introducerea unui mesaj de sincronizare. Acest mesaj este asemănător unui jeton pe care trebuie să îl recepţioneze fiecare agent pentru a putea să treacă la execuţia ciclului său de calcul. Pentru aceasta, fiecare agent utilizează un al doilea canal de comunicaţie pentru recepţionarea mesajelor de sincronizare (primul canal este folosit pentru recepţionarea mesajelor de tip ok sau nogood).

În figura 6.3 este prezentată nouă rutină *update-4* ce este executată de fiecare agent, pentru această a doua soluţie. Fiecare agent verifică dacă s-a atins starea de sincronizare, verificând dacă a recepţionat mesajele de sincronizare de la

agenţii vecini. Pentru verificarea stării de sincronizare, în figura 6.5 este prezentată o funcţie în cod NetLogo. În figura 6.4 este prezentată noua procedură de mânuire a mesajelor, *handle-message–4*. Spre deosebire de prima soluţie, după procesarea mesajelor, se trimite mesajul de sincronizare la agenţii vecini.

Protocolul de lucru presupune parcurgerea de fiecare agent a două etape:

- fiecare agent procesează toate mesajele din canalul său principal de comunicatii executând un ciclu de calcul. În momentul în care canalul principal de mesaje este gol, trimite un mesaj de tip „sincron” la agenţii vecini aflaţi înaintea lui în ordinea lexicografică.

- după fiecare ciclu, agenţii verifică dacă au recepţionat mesajele de sincronizare de la toţi vecinii săi, aflaţi după el în ordinea lexicografică, aşteptând în caz contrar recepţionarea acestor mesaje.

```
to update
 if Sincronizare
 [
  handle-message
  set Sincronizare false
 ]
While [ nu a recepţionat mesajul sincron de la toţi vecinii săi aflaţi după el în
ordine lexicografică – functia VerificareSincronizare
[  wait
   set Sincronizare false
]
set Sincronizare true
end
```

Figura 6.3 Procedura update-4

```
to handle-message
while [not empty? message-queue]
  [
   set msg retrieve-message
   set gt 0
   if (first msg = "ok?")[
     handle-ok-message msg]
   if (first msg = "nogood")[
     handle-nogood-message msg]
  ]
set msg list "sincron" who
send msg to AgentiiVecini  // se trimite doar la vecinii aflaţi înainte în ordinea
                                lexicografică
if (empty? message-queue)
  [
   set gt 1
   stop   ]
end
```

Figura 6.4 Procedură handle-message–4

```
to VerificareSincronizare
   while [not empty?VeciniAg and (not empty? message-queue1)]
   [ set lstmsg []
     foreach message-queue1
     [
       set msg ?
       set tip first  msg
       if (tip = "sincron" )
        [
          set lstmsg lput ? lstmsg
          set ag item 1 msg
          set VeciniAg remove ag VeciniAg
        ]
      ]
     foreach lstmsg
     [
       set message-queue1 remove ? message-queue1
     ]
   ]
 if empty? VeciniAg
 [
    set continuare true
   ;set message-queue1 []
 ]
end
```

Figura 6.5 Funcţia de verificare a stării de sincronizare

Există câteva diferenţe între cele două soluţii, diferenţe ce apar în cele două rutine update (figurile 6.1 şi 6.3) şi handle-message (figurile 6.2 şi 6.4). Pentru prima soluţie, agenţii întră în starea de aşteptare până când variabila Nragents devine egală cu numărul de agenţi. În schimb, pentru a doua soluţie de sincronizare, agenţii aşteaptă să recepţioneze mesajul sincron de la toţi vecinii săi aflaţi înaintea lui în ordinea lexicografică. Aceste prime modificări apar în rutina update. Relativ la cealaltă rutină de tratare a mesajelor, pentru a doua soluţie se trimite mesajul sincron la vecinii săi aflaţi înainte în ordine lexicografică (la finalul procesării mesajelor).

6.5.3 Analiza experimentală a efectului sincronizării parţiale şi complete a execuţiei agenţilor.

În acest paragraf sunt prezentate rezultatele experimentale pentru acest studiu, obţinute în urma implementării şi evaluării tehnicilor asincrone în condiţiile introducerii sincronizării parţiale a execuţiei agenţilor. Implementarea şi evaluarea s-a realizat pe baza modelului prezentat în capitolul 5 şi a soluţiilor din paragrafele §6.5.1 şi §6.5.2.

Experimentele s-au efectut în aceleaşi condiţii prezentate în paragraful §6.1. Pentru a putea face evaluarea celor două familii (variantele de bază şi variantele cu sincronizare), s-a contorizat fluxul de mesaje (cantitatea de mesaje info şi back schimbate de agenţi), numărul de constrângeri verificate şi numărul de constrângeri concurente verificate. Evaluările au fost făcute pentru cele două familii de tehnici.

6.5.3.1 Familia AWCS.

În cadrul familiei AWCS există mai multe variante ce se bazează pe construirea unor nogooduri eficiente (nogood learning) sau pe stocarea şi folosirea acestor nogooduri in procesul de selectie al valorilor (nogood processor). În acest studiu s-a implementat varianta de bază introdusă în [YDIK98] la care s-a aplicat tehnica nogood learning din [HY00] obţinându-se o varianta îmbunătăţită (notată cu AWCS-nl). Pentru această variantă s-a considerat că fiecare agent tratează complet mesajele existente în coada sa de mesaje) şi s-au făcut câte trei implementări corespunzător celor trei modele obţinute:

- Variante implementate în sistemul SEIS: AWCS-nl_1. În aceste cazuri sincronizarea este realizată cu ajutorul comenzii "ask".
- Variante implementate în sistemul SEIAS: AWCS-nl_2. Nu există sincronizare, agenţii operează asincron.
- Variante implementate în sistemul SEIAS modificat prin aplicarea metodelor de sincronizare introduse în paragrafele §6.4.1 şi §6.4.2. : AWCS-nl_3 şi AWCS-nl_4. Practic agenţii rulează în condiţiile unor sincronizări complete, respectiv parţiale.

În tabelul 6.4 sunt prezentate valorile obţinute pentru versiunile AWCS analizate.

		n = 30		n = 40		n = 50	
		m=n x2	m=n x2.7	m=n x2	m=n x2.7	m=n x2	m=n x2.7
AWCS-nl_1	TNogood	126.66	902.57	575.84	3639.52	713.60	24652.61
	TOk	470.48	1918.56	1864.50	7624.27	2403.25	51132.39
	Tcck	856.43	4475.58	2913.94	14482.23	3747.34	102388.82
	Tc-ccks	307.12	1608.16	802.74	4351.37	824.10	21888.16
AWCS-nl_2	TNogood	129.86	1412.17	502.77	12181.62	733.93	93259.95
	TOk	456.06	2881/41	1625.23	23964.88	2422.19	185845.00
	Tcck	900.47	6659.52	2628.08	45813.89	4163.19	387355.71
	Tc-ccks	294.81	2243.28	662.13	12512.22	793.45	73099.00
AWCS-nl_3	TNogood	137.31	1137.93	534.27	3439.96	755.42	22394.24
	TOk	493.51	2374.80	1738.56	7129.70	2512.65	47181.16
	Tcck	894.40	5507.58	2618.45	13858.95	3943.26	93429.74
	Tc-ccks	319.11	1992.80	759.39	4109.20	865.93	20065.21
AWCS-nl_4	TNogood	133.24	1211.12	567.23	3672.32	723.15	23371.67
	TOk	471.54	2417.56	1799.98	7889.14	2579.42	50083.72
	Tcck	931.18	5863.72	2751.92	15219.11	4081.18	109657.25
	Tc-ccks	328.38	2101.82	795.28	4310.87	876.12	21011.11

Table 6.4. Rezultatele experimentale pentru versiunile AWCS în cazul introducerii sincronizării parţiale şi complete a agenţilor

Analizând rezultatele din tabelul 6.4, se remarcă faptul că sincronizarea execuţiei agenţilor a redus efortul local, indiferent de varianta folosită (cea bazată pe comanda ask sau cea generală introdusă în acest articol). În schimb, odată cu creşterea dimensiunii problemelor (40 noduri), varianta asincronă AWCS-nl_2 a necesitat eforturi mult mai mari faţă de variantele cu sincronizare. Diferenţe mari de efort local apar mai ales pentru problemele de densitate mare (probleme cunoscute sub numele de probleme dificile). Se remarcă faptul că pentru probleme de

densitate rară (m= n x 2.0) variantele asincrone au avut costuri apropiate de variantele sincrone, chiar eforturi mai mici. Comparând cele două variante sincrone, se remarcă eforturi apropiate pentru obţinerea soluţiei, astfel încât soluţia practică propusă în acest articol necesită costuri apropiate de varinta simulată prin comanda ask.

În cazul fluxului de mesaje, comportamentul remarcat la efortul de calcul s-a păstrat aproximativ la fel, variantele sincrone necesitând un flux de mesaje mai mic decât variantele asincrone. Fluxul de mesaje a crescut pentru variantele asincrone odată cu creşterea dimensiunii problemelor rezolvate.

6.5.3.2. Familia ABT.

Plecând de la nucleul ABT, prin eliminarea informaţiilor învechite se poate ajunge la două mari tehnici: ABT ([YDIK98]) şi DisDB ([BM05]). Aceste două tehnici bazate pe o ordine statică sunt analizate pentru a vedea efectul sincronizării agenţilor. Spre deosebire de tehnica AWCS, aici s-au implementat versiunile în care fiecare agent tratează la fiecare ciclu doar un mesaj din coada sa de mesaje (notate cu ABT_k şi $DisDB_k$).

Pentru fiecare din tehnicile ABT şi DisDB s-au făcut câte trei implementări corespunzătoare celor trei sisteme obţinute:

- Variante implementate în sistemul SEIS: ABT_1 şi $DisDB_1$. În aceste cazuri sincronizarea este realizată cu ajutorul comenzii "ask".
- Variante implementate în sistemul SEIAS: ABT_2 şi $DisDB_2$. Nu există sincronizare, agenţii operează asincron.
- Variante implementate în sistemul SEIAS modificat prin aplicarea metodelor de sincronizare introduse în paragrafele §6.5.1 şi §6.5.2: $ABT_{3,4}$ şi $DisDB_{3,4}$.

În tabelul 6.5 sunt prezentate valorile obţinute pentru versiunile ABT, iar în tabelul 6.6 sunt cele pentru versiunile DisDB.

		n = 20		n = 30	
		m=n x2	m=n x2.7	m=n x2	m=n x2.7
ABT_1	TNogood	359.70	481.88.74	3883.12	3936.20
	TOk	1214.93	1615.48	8578.23	12343.88
	Tcck	67500.30	80834.37	957934.98	938790.08
	Tc-ccks	14502.26	16572.45	147128.29	123501.58
ABT_2	TNogood	305.70	366.94	3181.57	2196.52
	TOk	1124.16	1438.48	6518.74	9943.55
	Tcck	60382.63	69332.91	763992.16	698240.00
	Tc-ccks	8993.06	7593.00	90601.22	45264.01
ABT_3	TNogood	360.41	376.99	3713.65	3764.15
	TOk	1237.79	1495.00	8065.23	10209.04
	Tcck	68321.27	76823.12	85234.16	800193.03
	Tc-ccks	13401.26	12312.34	112675.05	89000.45
ABT_4	TNogood	371.12	395.18	3945.18	3799.55
	TOk	1323.11	1578.06	8245.32	11311.71
	Tcck	67001.81	75121.12	83555.23	753712.20
	Tc-ccks	12182.11	11133.56	109129.82	90101.34

Tabelul 6.5. Rezultatele obţinute pentru versiunile ABT în cazul introducerii sincronizării parţiale şi complete a agenţilor.

		n = 20		n = 30	
		m=n x2	m=n x2.7	m=n x2	m=n x2.7
$DisDB_1$	TNogood	138.76	316.94	2803.34	3104.44
	TOk	300.04	722.89	6855.19	7235.88
	Tcck	18548.68	40841.66	603288.94	778016.08
	Tc-ccks	4526.52	8112.34	124728.67	154409.43
$DisDB_2$	TNogood	83.10	216.41	2785.65	2983.69
	TOk	213.72	556.03	6518.74	6878.05
	Tcck	12117.12	29850.88	585992.16	655243.39
	Tc-ccks	2468.48	3963.02	70605.97	82785.79
$DisDB_3$	TNogood	125.38	271.25	2813.03	3016.96
	TOk	277.64	646.82	7005.56	7029.70
	Tcck	16837.14	35589.37	603192.16	712024.78
	Tc-ccks	3791.24	6289.09	115059.67	90786.78
$DisDB_4$	TNogood	117.21	273.67	2767.28	2994.39
	TOk	272.40	651.71	6981.90	7002.81
	Tcck	15002.91	36111.52	523023.73	679821.34
	Tc-ccks	34230.74	6421.90	102011.72	88564.20

Table 6.6. Rezultatele obţinute pentru versiunile DisDB în cazul introducerii sincronizării parţiale şi complete a agenţilor.

Cele două tehnici bazate pe o ordine statică s-au comportat diferit faţă de tehnica AWCS, aşa cum s-a remarcat şi în paragraful 6.2. Efortul local cel mai mic s-a depus pentru varianta asincronă, variantele cu sincronizare necesitând verificarea unei număr mai mare de constrângeri. De asemenea, cel mai mic flux de mesaje s-a obţinut pentru varintele asincrone. Cele două tehnici s-u comportat la fel pentru ambele tipuri de probleme: probleme cu densitate rară sau probleme dificile.

Soluţia de sincronizare propusă în acest paragraf a fost superioară celei oferite de comanda ask din NetLogo. Din păcate, tehnicile din familia ABT bazate pe această ordine dinamică, indiferent ca necesită sau nu adăugarea de legături suplimentare, se comportă mai bine în cazul asincron, sincronizarea nefiind o soluţie de reducere a costurilor.

Rezultatele acestui studiu au fost publicate în [MVCPP07].

6.6. Metode de eliminare a mesajelor redundante.

6.6.1 Mesaje învechite şi redundante de tip ok.

Tehnicile de căutare asincrone folosesc mesaje de tip ok pentru a comunica schimbarea valorilor fiecărui agent. Aceste mesaje sunt transmise şi recepţionate de agenţi folosind canale de comunicaţie de tip FIFO. Analiza comportamentului agenţilor, pe baza celor două modele, arată că anumiţi agenţi, mai ales în cazul tehnicilor din familia ABT, au un flux de mesaje foarte mare cu anumiţi agenţii copii. Multe dintre acestea sunt mesaje inutile, având ca efect încărcarea costurilor de obţinere a soluţiei în mod inutil.

În capitolul 4 au fost definite şi prezentate mesajele redundante de tip ok/info. Conform definiţiilor 4.2.1 şi 4.2.2 ăn cazul tehnicilor de căutare asincrone sunt întâlnite mesaje de tip ok/info *învechite* şi mesaje de tip ok/info *redundante***.** În

paragrafele următoare sunt prezentate mai multe soluţii de reducere sau eliminare a acestor mesaje inutile, soluţii publicate în [MusPO05], [MusVP06]. Anumite soluţii, cum sunt cele de management, permit şi reducerea efortului local depus de agenţi, pe lângă reducerea fluxului de mesaje.

6.6.2 Filtrarea mesajelor învechite sau redundante.

Cele două modele de implementare şi evaluare au permis studiul cozilor de mesaje pentru majoritatea tehnicilor de căutare studiate. Analizând cozilor de mesaje pentru implementările tehnicilor de căutare din cele două mari familii (ABT şi AWCS), după ce acestea au fost implementate folosind cele două modele propuse în capitolul 5 şi aplicate la problema colorării grafurilor, varianta distribuită, s-a observat o cantitate foarte mare de mesaje de tip ok redundante. În primul rând s-au identificat mesaje de tip ok/info de la acelaşi agent, trimise în diverse momente ale rulării, mesaje conţinând aceeaşi valoare sau valori diferite. Aşa cum s-a prezentat în capitolul 4 acestea sunt mesaje de tip ok redundante, iar unele învechite.

Plecând de la aceste observaţii, în acest paragraf se propune o tehnică simplă de filtrare, ce se aplică asupra cozilor de mesaje (ce sunt canale de comunicaţii de tip FIFO). Practic, când se extrage din coada de mesaje un mesaj de tip *ok/info*, acesta nu este tratat imediat de rutina sa, ci se verifică dacă acesta nu este cumva redundant sau învechit. Dacă apare una din aceste situaţii, se ignoră mesajul (eliminându-l din coada de mesaje), altfel se apelează rutina normală de tratare a mesajelor ok. Acest lucru are ca efect reducerea numărului de apeluri pentru procedura check-agent-view.

În figura 6.6 este prezentat algoritmul de filtrare aplicat pentru fiecare coadă de mesaje.

Procedure OK-messages-filtering
Extract Msg(ok, x_i)
If *there is no other ok message in the message-queue received from* x_i
[
Apelează procedura de tratatre a mesajelor ok (ok, x_i) // **Ok(msg)**
]

Figura 6.6 Filtrarea mesajelor

Algoritmul de filtrare prezentat în figura 6.6, este aplicat de fiecare agent în momentul în care extrage un mesaj de tip ok, înainte de tratarea acestuia. Aplicarea acestui algoritm va permite reducerea efortului local depus de agent, prin reducerea cantităţii de constrângeri verificate. Aplicarea acestui algoritm de filtrare nu are efect asupra completitudinii.

6.6.3 Managementul mesajelor.

6.6.3.1. Managementul mesajelor în cazul tehnicii AWCS.

Aşa cum a fost prezentat, tehnicile de căutare asincrone din familia AWCS (ca şi toate tehnicile de căutare asincrone) folosesc mesaje de tip ok pentru a comunica schimbarea valorilor fiecărui agent, respectiv mesaje de tip nogood pentru a anunţa o inconsistenţă. Tehnicile asincrone se caracterizează prin existenţa unor proceduri de tratare a mesajelor. În mod tipic există proceduri de tratare pentru mesajele ok (info), nogood (back), addl sau removel. Aceste rutine tratează secvenţial mesajele existente în cozile de mesaje. În mod tipic, fiecare agent extrage câte un mesaj din coada sa de mesaje, identifică tipul de mesaj şi apelează rutina de tratare corespunzătoare.

În [Yok01] se precizează că în cazul tehnicii AWCS agenţii citesc toate mesajele existente în cozile de mesaje. Acolo nu este prezentată şi modalitatea de tratare, adică un protocol care să definească ordinea în care mesajele sunt tratate. În acest paragraf se propune o metodă de management pentru mesaje, astfel încât fiecare agent poate trata complet sau parţial mesajele existente la un moment dat în coada lui de mesaje. Această metodă este aplicată pentru tehnica asynchrounous weak-commitment search, permiţându-se eliminarea din cozile de mesaje a mesajelor învechite şi redundante [MusVP06]. Acest lucru are ca efect obţinerea unei tehnici derivate mult mai eficientă în ceea ce priveşte costurile obţinerii soluţiei.

Strategia de management a mesajelor va permite reducerea sau eliminarea mesajelor redundante în cazul tehnicilor din familia AWCS. Este vorba de tehnica de bază AWCS şi versiunile la care se aplică tehnicile de tip "nogood learning". Folosind această metodă de management, este extins modelul NetLogo de implementare a tehnicilor asincrone cu suportul pentru managemntul mesajelor, model prezentat în capitolul 5.

Varianta propusă pleacă de la soluţia clasică în care mesajele sunt citite şi prelucrate secvenţial, unul câte unul. Această soluţie presupune existenţa unei rutine de tratare a mesajelor, care extrage secvenţial fiecare mesaj, identifică tipul acestuia şi apelează rutina de prelucrare corespunzătoare. În figura 6.6. este prezentată procedura de tratare a mesajelor secvenţial, valabilă pentru tehnicile din familia AWCS. Această procedură este derivată din modelul prezentat în capitolul 5.

In această procedură de tratare a mesajelor, procedurile *handle-ok-message* şi *handle-nogood-message* sunt rutinele de tratare a mesajelor ok şi nogood, prezentate la algoritmul AWCS. Fiecare din aceste rutine se caracterizează prin faptul că apelează procedura check-agent-view având ca scop verificarea consistenţelor din lista agent-view şi selectarea unei noi valori pentru agentul curent. Soluţia pentru managementul mesajelor, ce se poate aplica în Netlogo sau orice alt limbaj ales pentru implementare, va permite citirea fie a tuturor mesajelor din coada de mesaje, fie citirea parţială a mesajelor, în funcţie de valoarea variabilei *msize***.**

Soluţia presupune existenţa unui protocol de tratare a mesajelor pentru cazul tehnicilor din familia AWCS. Acest protocol stabileşte ordinea în care sunt tratate mesajele şi momentul în care se încearcă asocierea unei noi valori. De asemenea, acest protocol permite tratarea completă sau partială a mesajelor, prin introducerea parametrului msize ce reprezintă numărul de mesaje citite la un moment dat din coada de mesaje. Parametrul msize poate lua valori între 1 şi lungimea cozii de mesaje. În cazul în care *msize* este 1 se obţine soluţia de tratare secvenţială a mesajelor, prezentată în capitolul 5.

```
to handle-message
   if (empty? message-queue) [stop]
   set msg retrieve-message
    if (first msg = "ok?")
    [
       set xj item 0 (item 1 msg)
       if (remove-old-ok-message xj = 0 )
       [
         set messages-received_ok messages-received_ok + 1
         handle-ok-message msg
      ]
    ]
   if (first msg = "nogood")
   [
     set messages-received_nogood messages-received_nogood + 1
     handle-nogood-message msg
  ]
end
```

Figura 6.7. Procedura de tratare secvenţială a mesajelor pentru tehnica AWCS

În figura 6.8 este prezentată procedura de management a mesajelor, procedură ce este aplicată de fiecare agent la coada sa de mesaje. Mesajele vor fi extrase în ordinea sosirii în structurile de tip coada asociate agenţilor şi tratate astfel:

- în cazul în care mesajul este de tip ok?, se actualizează contextul de lucru al agentului cu valoarea recepţionată, fără a se verifica consistenţa valorilor existente acolo sau a încerca selectarea unei noi valori.
- Dacă mesajul este de tip nogood, se salvează valoarea nogood şi se actualizează contextul de lucru cu valorile noilor agenţi neconectati iniţial cu agentul curent.

După parcurgerea şi extragerea completă sau parţială a mesajelor, se apelează o singură dată rutina check-agent-view cu scopul verificării consistenţelor din lista agent-view şi selectarea (eventual) a unei noi valori consistente.

Această soluţie presupune câteva schimbări în algoritmul de bază AWCS prezentat în figura 3.4 (capitolul 3). Prima modificare presupune renunţarea completă la rutina *handle-ok-message*. Tratarea mesajelor ok se face acum global, în rutina *handle-message*. Pentru fiecare mesaj de tip ok (ce are forma "ok?" [A_k current-value priority]) extras din coada de mesaje, se actualizează contextul de lucru al agentului adică ceea ce se numeşte agent-view. Se remarcă faptul că actualizarea se face în ordinea în care mesajele au fost recepţionate. În cazul existenţei a două mesaje ok recepţionate de la acelaşi agent A_k, în lista agent-view va rămâne ultima valoare transmisă de agentul A_k (primul mesaj ok sosit este fie învechit, fie redundat). Apelarea procedurii check-agent-view, ce are rol în verificarea consistenţei listei agent-view şi a selecţiei unei noi valori, se face ulterior,

după parcurgerea tuturor mesajelor din coada de mesaje. A doua modificare presupune adaptarea rutinei rutinei *handle-nogood-message* astfel încât aceasta să nu mai apeleze procedura check-agent-view .

```
to message-manage [ msize ]
 set n 0
1.  while [not empty? message-queue and n < msize] or
1′   while [not empty? message-queue] ***
  [
    set msg retrieve-message
    if (first msg = "stop")
    [
       set done  true
       stop
    ]
    if (first msg = "ok?")
    [
      set messages-received_ok messages-received_ok + 1
      Update agent-view cu msg
    ]
    if (first msg = "nogood")
    [
      set messages-received_nogood messages-received_nogood + 1
      handle-nogood-message msg
    ]
   set n n + 1
  ]
 if n != 0
 [check-agent-view]
End
```

Figura 6.8. Procedura de management a mesajelor pentru tehnica AWCS

În ce priveşte tratarea completă sau parţială a mesajelor, acest lucru se poate rezolva prin intermediul variabilei msize sau renunţând la condiţia a doua de limitare a pachetelor (linia etichetată cu ***). Aceasta are rol de decide numărul de mesaje extrase şi tratate din coada de mesaje. Dacă msize este egal cu numărul de elemente din coada de mesaje sau dacă lipseşte această condiţie, procedura prezentată în figura 6.8 permite tratarea tuturor mesajelor. În schimb, daca msize este 1 se obţine tratarea secvenţială a mesajelor prezentată în figura 6.7. În paragraful de experimente sunt investigate şi analizate performanţelor pentru trei variante ce se obţin pentru msize =1 (tratarea secvenţială), length (message-queue) (tratarea completă a mesajelor relativ la momentul începerii prelucrării mesajelor) şi tratarea tuturor mesajelor inclusiv a celor recepţionate ulterior.

În figura 6.8 există două posibilităţi de terminare a tratatării mesajelor. Prima variantă presupune introducerea liniei 1 în locul liniei 1′. În acest caz, fiecare agent se opreşte în momentul în care fie nu mai există mesaje fie s-au tratat *msize*

mesaje. A doua variantă presupune introducerea liniei 1′. Această variantă va forţa agenţii să se oprească dacă nu mai există mesaje în coada lor de mesaje. Trebuie remarcată o diferenţă de comportament între variantele cu tratarea completă a mesajelor obţinute prin folosirea celor două linii (prima variantă presupune *msize = length (message-queue)*). Tratarea mesajelor presupune un efort şi prin urmare apariţia unei întârzieri. Este posibil să mai sosească alte mesaje faţă de momentul iniţial. În cazul celei de-a doua variante (obţinută prin introducerea liniei 1′) dacă apar ulterior mesaje acestea sunt totuşi tratate astfel depăşind numărul msize de mesaje permise. În schimb, pentru prima variantă (obţinută prin introducerea liniei etichetate cu 1) se tratează doar acele mesaje existente la inceput în coada de mesaje.

Soluţia de mai sus presupune apelarea o singură dată a procedurii check-agent-view astfel reducându-se efortul de calcul local depus de agent. Tratarea globală a tuturor mesajelor (eventual tratarea parţială a lor) duce la reducerea dramatică a numărului de cicluri, lucru confirmat şi de experimentele prezentate în paragraful următor.

6.6.4. Analiza experimentală a metodelor de reducere a redundanţei mesajelor.

Pentru a vedea comportamentul tehnicilor relativ la metodele de reducere a mesajelor redundante s-a utilizat sistemul SEIS. În acest sistem au fost implementate şi analizate mai multe versiuni pentru tehnica AWCS, după cum urmează:

- versiunea de bază din [YDIK98] notată $AWCS_1$ (msize =1)
- versiunea de bază la care se aplică tehnica filtrării propusă în paragraful 6.5.2, notată $AWCS_2$.
- versiunea AWCS cu tratarea completă a mesajelor, notată cu $AWCS_3$ (msize=length (message-queue)).
- versiunea AWCS cu tratarea completă a mesajelor, notată cu $AWCS_4$ dar folosind condiţia de oprire bazată pe linia 1′ din figura 3.

Prima unitate de măsură utilizată a fost ciclul. Aceasta permite evaluarea efortului global depus de agenţi. Din analiza valorilor din tabelul 1 se observă diferenţe foarte mari între versiunile secvenţiale şi cele cu management. Aplicarea managementului mesajelor a redus dramatic numărul de ciclii necesari pentru obţinerea soluţiei. Cele două versiuni cu managementul mesajelor ($AWCS_3$ şi $AWCS_4$) au necesitat un număr apropiat de ciclii în obţinerea soluţiei, la mare distanţă de variantele secvenţiale.

În ceea ce priveşte efortul local depus de agenţi (constrângeri), analizând valorile din tabelul 6.7 se observă o reducere a efortului de calcul în cazul tehnicilor cu managementul mesajelor. Cu toate că aplicarea tehnicii filtrării mesajlor a redus la jumătate efortul local, valorile sunt destul de departe de valorile obţinute de variantele $AWCS_3$ şi $AWCS_4$. În schimb, dintre acestea se remarcă varianta $AWCS_3$ cu limitarea tratării mesajelor la numărul existent înainte de începerea prelucării mesajelor.

		n = 20		n = 30	
		m=n x 2	m=n x2.7	m=n x2	m=n x2.7
$AWCS_1$	TNogood	632.01	13893.43	2888.42	129444.52
	Tok	2076.54	31660.36	9485.79	295212.48
	Ciclii	152.76	2292.86	439.40	14232.83
	TCcks	5096.36	91552.0	22020.45	799855.65
	Tc-ccks	545.69	8956.43	1607.70	56470.13
$AWCS_2$	TNogood	361.60	3067.68	1836.76	28480.76
	Tok	992.34	5558.84	4837.10	49612.60
	Ciclii	93.47	517.86	275.02	3074.26
	TCcks	2726.40	19373.27	12312.96	166011.07
	Tc-ccks	308.51	1986.67	952.75	12135.51
$AWCS_3$	TNogood	184.08	1507.90	619.51	21777.40
	Tok	648.88	3561.98	2127.59	52013.80
	Ciclii	12.02	29.33	17.95	166.17
	TCcks	748.49	3359.61	1706.13	31129.40
	Tc-ccks	155.99	579.77	275.65	3882.74
$AWCS_4$	TNogood	187.13	2723.84	764.86	34863.76
	Tok	674.12	6461.79	2628.86	84097.27
	Ciclii	11.13	43.27	18.91	225.20
	TCcks	697.12	5532.81	1852.22	46817.96
	Tc-ccks	187.31	1137.02	372.91	6633.67

Table 6.7. Rezultatele pentru versiunile AWCS în cazul introducerii mangementului mesajelor.

În cazul fluxului de mesaje, se remarcă reducerea acestui flux în cazul aplicării filtrării mesajelor (varianta $AWCS_2$), dar mai ales pentru variantele cu management. Dintre cele două variante de sincronizare propusă, varianta $AWCS_3$ a necesitat un flux de mesaje ceva mai mic.

6.7. Concluzii.

Modelul prezentat în capitolul anterior a permis implementarea şi studiul tehnicilor asincrone din două mari familii: familia ABT şi familia AWCS. Pe baza acestor studii (în diferite situaţii) s-au identificat anumite îmbunătăţiri.

Tehnicile asincrone se caracterizează prin diverse proprietăţi, ele sunt o mixtură de tehnici de căutare, tehnici de consistenţă, tehnici de reordonare pentru agenţi, tehnici de backjumping sau tehnici de învăţare. Prin urmare ele au avut comportamente diferite în situaţiile studiate relatativ la efortul depus pentru obţinerea soluţiei. În tabelul 6.8 sunt sintetizate comportamentele tehnicilor de căutare asincrone din cele două mari familii, evidenţiindu-se situaţiile în care ele se comportă relativ bine.

Tehnicile din familia AWCS, bazate pe o ordine dinamică pentru agenţi, au necesitat costuri mai mici în obţinerea soluţiei în cazul sincronizării execuţei agenţilor. Soluţia de sincronizare propusă în acest capitol permite reducere a efortului local şi a fluxului de mesaje comparativ cu variantele asincrone.

A doua categorie de tehnici, cele din familia ABT s-au comportat diferit, ele necsitând costuri mai mici în cazul asincron decât în cazul sincronizării execuţiei agenţilor. Toate aceste tehnici folosesc o ordine statică pentru agenţi.

O a treia categorie de tehnici de căutare asincrone evaluate, cele de îmbunătăţire iterativă a soluţiei–DB, au necesitat costuri apropiate în ceea ce priveşte obţinerea soluţiei, indiferent de modul de rulare: cu sau fără sincronizarea execuţiei agenţilor.

Tabelul 6.8 Sintetizarea comportamentului tehnicilor de căutare asincrone

Tehnica de căutare asincronă	**Autori Publicaţie**	**Situaţii studiate**					
		Sincroniz. completă	Sincroniz. parţială	Operarea asincronă	Filtrarea mesajelor	Tratarea completă a mesajelor	Întârzieri mesaje
ABT	Yokoo- [YDIK92]	-	-	+	+	++	--
DisDB	Bessiere- [BMM01]	-	-	+	+	++	--
AWCS	Yokoo- [YDIK98]	++	+	-	-	++	+
AWCS-nogood learning	Yokoo,- [YDIK98], Hirayama [HY00]	++	+	-	-	++	+
DB	Yokoo – [YH96]	+	+	+			+

Legenda: - - = costuri foarte mari; - costuri mari;
+ = costuri mici; ++ = costuri foarte mici.

Introducerea managementului mesajelor a permis reducerea numărului de mesaje redundante şi învechite. Astfel s-au obţinut versiuni pentru tehnicile din familia AWCS în care s-a redus dramatic costurile necesare obţinerii soluţiei.

Se remarcă diferenţe de performanţe foarte mari între variantele secvenţiale şi cele cu tratarea completă a mesajelor. Filtrarea mesajelor de tip ok a redus costurile, dar totuşi, implică un efort global mare (exprimat în ciclii sau constrângeri concurente).

Dintre variantele cu managementul mesajelor se remarcă aceea variantă în care se limitează tratarea mesajelor existente la începutul unui ciclu de calcul. Acest lucru este valabil pentru familia AWCS. Apariţia ulterioară a altor mesaje, duce la mărirea costurilor. Variantele cu limitare au necesitat costuri mai mici în ceea ce priveşte numărul de constrângeri concurente, dar un flux de mesaje mai mare. Ca o concluzie, se remarcă faptul că introducerea managementului, indiferent de variantele propuse, permite obţinerea unor performanţe mult mai bune.

Rezultatele evidenţiate în acest capitol au fost publicate în [MusCP06], [MJP06] şi [MVCPP07].

SECŢIUNEA III.

CONTRIBUŢII LA ÎMBUNĂTĂŢIREA PERFORMANŢELOR TEHNICILOR DE CĂUTARE ASINCRONE

Capitolele secţiunii de faţă conţin contribuţii aduse în domeniul îmbunătăţirii performanţelor tehnicilor de căutare asincrone. Secţiunea prezintă mai multe soluţii legate de eliminarea efectului exploziei valorilor nogood, limitarea fluxului de mesaje pentru legăturile temporare şi utilizarea eficientă a listelor nogood.

În capitolul 7 este propusă o soluţie de aplicare a tehnicii cu flaguri în cazul familiei ABT. Este înlocuită stocarea valorilor nogood cu etichetarea valorilor din domeniu prin două tipuri de flaguri. Spaţiul de memorie necesar se reduce considerabil în cazul aplicării tehnicii cu flaguri.

În capitolul 8 este investigat comportamentul tehnicii ABT_{temp} pentru diferite valori ale numărului de mesaje transmise pentru legăturile temporare. Pe baza acestei analize, în acest capitol sunt identificate şi prezentate mai multe soluţii originale de determinare a numărului de mesaje schimbate de agenţii cu legături temporare, număr optim de mesaje pentru care trebuie păstrată o legătură temporară.

În capitolul 9 se propune un membru hibrid pentru familia ABT, membru derivat din nucleul ABT prin eliminarea informaţiilor învechite dintre agenţi, combinând două metode mai vechi de eliminare a informaţiilor învechite dintre agenţi: adăugarea de legături permanente între agenţi, respectiv adăugarea de legături temporare.

În capitolele 10 şi 11 sunt propuse soluţii de adaptare a tehnicii „nogood processor” pentru tehnica AWCS şi de combinarea a acestei tehnici cu tehnicile de construire a unor nogooduri eficiente, obţinându-se o variantă hibridă cu costuri mai mici în obţinerea soluţiei.

7. TEHNICI DE CĂUTARE ASINCRONE CU FLAGURI.

Tehnicile de căutare asincrone se caracterizează prin apariţia valorilor „nogood" în timpul căutării soluţiei. Tehnica ABT necesită un spaţiu a cărui complexitate pentru fiecare agent este exponenţială [YDIK98]. Complexitatea exponenţială a algoritmului, în cazul nefavorabil, este determinată în primul rând de numărul înregistrărilor de mesaje *nogood.* De asemenea, şi tehnicile din cadrul familiei ABT se bazează pe trimiterea de mesaje de tip nogood (numite mesaje back) între agenţi şi suferă de o explozie a valorilor de tip nogood. Stocarea acestor valori nogood va necesita un spaţiu de memorie foarte mare, implicând şi un efort de căutare mult mai mare.

Recent, o tehnică derivată, numită asynchronous backtracking cu flaguri, permite reducerea exploziei valorilor nogood. Această tehnică este propusă în [GWW03] şi se bazează pe utilizarea flagurilor în locul stocării valorilor nogood, fiind aplicată doar pentru tehnica ABT.

În acest capitol se propune o soluţie originală de aplicare a tehnicii cu flaguri în cazul familiei ABT. Este înlocuită stocarea valorilor nogood cu etichetarea valorilor din domeniu prin două tipuri de flaguri. Această soluţie este aplicată pentru nucleul ABT obţinându-se o variantă originală numită *ABT kernel cu flaguri*. Plecând de la acest nucleu se poate ajunge la două tehnici cunoscute: Asynchronous Backtracking şi Distributed Dynamic Backtracking (Bessiere) în variantele cu flaguri [Mus05]. În felul acesta se obţine o generalizare a tehnicii cu flaguri pentru cazul familiei ABT, soluţie originală propusă în acest capitol. Spaţiul de memorie alocat este considerabil redus în cazul variantelor cu flaguri.

Studiul investighează eficienţa noilor tehnici din familia ABT, în raport cu fluxul de mesaje schimbate de agenţi, numărul de cicli consumaţi pentru găsirea soluţiei şi constrângerile verificate, comparativ cu tehnicile de bază derivate din nucleul ABT. Redundanţa mesajelor în cazul acestor tehnici cu flaguri este de asemenea analizată. De asemenea, în acest capitol se investighează eficienţa tehnicii asynchronous backtracking cu flaguri (propusă în [GWW03]), în raport cu fluxul de mesaje, efortul de calcul necesar si ciclii consumaţi pentru găsirea soluţiei. Sunt propuse câteva îmbunătăţiri asupra tehnicii asynchronous backtracking cu flaguri (restartarea căutării şi eliminarea redundanţei mesajelor).

Capitolul începe cu o introducere asupra contextului de lucru. În paragraful al doilea este prezentată tehnica etichetării cu flaguri introdusă în [GWW03] pentru tehnica ABT, urmată de câteva propuneri de îmbunătăţire a perfomanţelor acestei tehnici. În cel de-al patrulea paragraf se propune o soluţie originală de aplicare a tehnicii flagurilor în cazul nucleului ABT. În paragraful de rezultate experimentale sunt analizate tehnicile de căutare asincrone obţinute prin aplicarea tehnicii flagurilor. Sunt utilizate pentru implementare şi evaluare modelele prezentate anterior în capitolul 3. Pe baza acestor rezultate experimentale se prezintă un studiu comparativ pentru tehnicile prezentate. Ultimul paragraf prezintă concluziile aplicării tehnicii flagurilor în cazul familiei ABT.

7.1. Algoritmul Asynchronous Backtracking cu flaguri(ABTWF)

Tehnica asynchronous backtracking (varianta din [YDIK98]) conţine o rutină de tratare a mesajelor nogood recepţionate, rutină prezentată în capitolul 3. Aceste mesaje de tip nogood sunt analizate pentru a vedea dacă sunt sau nu învechite (consistente cu vederea agentului, conform definiţiei 3.3.4). Orice agent ce recepţionează un mesaj nogood verifică dacă valoarea nogood este consistentă cu vederea sa, în caz afirmativ acceptând mesajul şi stocându-l pentru ca mai târziu agentul să nu mai asocieze aceea valoare.

În [GWW03] sunt identificate mai multe tipuri de valori nogood ce sunt recepţionate de un agent (nu neapărat acceptate), în cazul tehnicii de bază ABT. De fapt, sunt identificate patru clase de valori nogood care sunt recepţionate de fiecare agent prin intermediul mesajelor de tip nogood, după cum urmează:

Definiţia 7.1 - ***nogood de clasă (1)***. Un nogood recepţionat de agentul A_i se numeşte de clasă 1 dacă lista nogood conţine o asociere de tipul $\{(x_i, d_i)\}$, unde valoarea d_i este aceeaşi cu valoarea curentă a agentului curent A_i.

Definiţia 7.2 - ***nogood de clasă (2)***. Un nogood recepţionat de agentul A_i se numeşte de clasă 2 dacă lista nogood conţine o asociere de tipul $\{(x_i, d_i)\}$, unde valoarea d_i este diferită de valoarea curentă a agentului curent A_i.

Definiţia 7.3 - ***nogood de clasă (3)***. Un nogood recepţionat de agentul A_i se numeşte de clasă 3 dacă lista nogood conţine o submulţime de asocieri de valori $\{(x_j,d_j),\ldots,(x_i, d_i)\}$ care sunt consistente cu vederea agentului A_i (agent_view) şi valoarea curentă a acestui agent (x_i, current _value).

Definiţia 7.4 - ***nogood de clasă (4)***. Un nogood recepţionat de agentul A_i se numeşte de clasă 4 dacă lista nogood conţine o submulţime de asocieri de valori $\{(x_j,d_j),\ldots,(x_i, d_i)\}$ care nu sunt consistente cu vederea agentului A_i (agent_view) şi valoarea curentă a acestui agent (x_i, current _value).

Analizând tehnica asynchronous backtracking se remarcă faptul că un agent A_i îşi poate schimba valoarea în două situaţii:

(1). la recepţionarea unui mesaj ok?;

(2). la recepţionarea unui mesaj nogood consistent cu vederea agentului A_i.

Plecând de la această observaţie, sunt propuse două propoziţii relative la efectul celor patru clase de nogooduri.

Propoziţia 7.1. Orice nogood de clasă (2) şi (4) recepţionat de un agent nu necesită stocarea, fără a avea efect asupra completitudinii.

Demonstraţie.

Conform definiţiilor 7.2 şi 7.4, un nogood de clasă (2) sau (4) conţine o valoare diferită de valorile existente în lista agent_view a agentului A_i. Din această cauză în momentul generării şi transmiterii mesajului nogood către agentul A_i (bănuit a fi vinovat de apariţia situaţiei de conflict), acesta şi-a schimbat cel puţin acea valoare. Ca o consecinţă, nogoodul este învechit şi nu

mai are relevanţă. Prin urmare el poate fi respins şi nu mai necesită stocarea lui pentru mai târziu.

Propoziţia 7.2. Dacă un agent recepţionează un nogood de clasă (1) sau (3) va trebui să îşi schimbe valoarea curentă asociată variabilei sale.

Demonstraţie.
Când agentul A_i recepţionează un nogood de clasă (1) de la agentul A_k, înseamnă că valoarea curentă a agentului A_i nu satisface cel puţin o constrângere între A_k şi A_i. Mai departe, când se recepţionează un nogood de clasă (3) de la un agent A_k, înseamnă că lista nogood este consistentă şi cu vederea agentului şi cu valoarea sa curentă. Concluzia este că pentru cele două clase de nogood-uri, valoarea asociată agentului curent (x_i, d_i) nu este bună (dintre cei doi agenţi aflaţi în conflict, vinovat de apariţia eşecului este A_i).

Ca o consecinţă a propoziţiei 7.2, agentul curent A_i va încerca să selecteze o nouă valoare şi va stoca acel nogood pentru a nu mai repeta această greşeală (adică să mai asocieze această valoare variabilei sale în contextul vederii sale).

Lema 7.3. Orice nogood de clasă (3) poate fi stocat temporar.

Demonstraţie.
Conform definiţiei 7.3. un nogood de clasă (3) conţine aceleaşi valori pentru agenţii vecini cunoscuţi. Totuşi, de-a lungul rulării algoritmului, vederea (agent-view) a agentului A_i se schimbă şi, în consecinţă, nogoodul stocat devine învechit. Din acest motiv stocarea permanentă a valorii nogood de clasă (3) nu este necesară. Practic este suficientă stocarea temporară a acelei valori cât vederea agentului conţine aceleaşi valori cu agentul A_k. Trebuie remarcat faptul că definiţia de consistenţă este relativă la momentul recepţionării nogoodului şi nu ulterior.

Pe baza observaţiilor anterioare în [GWW03] se propune o tehnică derivată, tehnică numită "asynchronous backtracking cu flaguri". Tehnica aceasta este obţinută pornind de la tehnica de bază ABT, prin înlocuirea stocării valorilor nogood cu indexarea valorilor locale din domeniul cu flaguri. Acest lucru presupune utilizarea a două tipuri de flaguri, corespunzător claselor (1) şi (3) de nogood-uri:

(1) flag permanent (PF): când se recepţionează un nogood de clasă (1), agentul A_i va seta flagul valorii curente cu PF. Un astfel de flag arată că valoarea curentă nu va mai fi asociată niciodată în timpul rulării. Un astfel de flag nu va fi şters niciodată.

(2) flag temporar (TF): când se recepţionează un nogood de clasă (3), agentul A_i va seta flagul ataşat valorii curente cu TF. Acest tip de flag se va schimba de-a lungul rulării algoritmului.

Practic, când agentul A_i recepţionează un mesaj ok? sau transmite un mesaj nogood, înseamnă că un agent cu prioritate mai mare decât A_i şi-a schimbat valoarea sa sau că A_i trebuie să îşi schimbe valoarea curentă. În astfel de situaţii, flagul temporar asociat valorii curente trebuie şters astfel încât acea valoare (etichetată cu TF) să mai poată fi selectată mai târziu.

Utilizarea acestor două tipuri de flaguri a impus şi schimbarea testelor de consistenţă utilizate la verificarea soluţiei parţiale, după cum urmează:

(1) Lista *agent_view* şi (x_i, valoare curentă) satisfac toate constrângerile problemei;

(2) Valoarea curentă nu a fost etichetată cu un flag local (fie de tip PF sau de tip TF).

7.2. Îmbunătăţirea performanţelor tehnicii Asynchronous Backtracking cu flaguri

Pentru îmbunătăţirea perfomanţelor tehnicii derivate obţinute sunt propuse în acest capitol câteva modificări asupra flagurilor stocate în locul nogood-urilor.

O primă propunere constă în a reseta flagurile permanente în momentul recepţionării unui mesaj de tip ok. În momentul recepţionării unui mesaj de tip ok, nu se şterg doar flagurile de tip TF, ci se face o resetare a listei cu flaguri ataşate valorilor din domeniul unui agent. Acest lucru va permite reducerea numărului de mesaje de tip nogood neconsistente recepţionate de acel agent.

Resetarea listei de flaguri, temporare sau permanente, va permite în momentul recepţionării unei valori de tip ok şi apariţiei unei inconsistenţe să se repornească procesul de căutare a unor noi valori consistente. În schimb, în momentul generării unui mesaj de tip nogood sunt curăţate doar flagurile temporare, nu şi cele permanente.

Rezultatele experimentale ce vor fi prezentate în paragrafele următoare, arată o reducere a costurilor obţinerii soluţiei, faţă de varianta cu flaguri prezentată în [GWW03].

A doua propunere este relativă la existenţa mesajelor redundante de tip ok şi în cazul tehnicii ABT cu flaguri, varianta din [GWW03]. Analizând cozile cu mesaje pentru această tehnică asincronă, după ce acestea au fost implementate folosind cele două sisteme propuse în capitolul 5 şi aplicate la problema colorării grafurilor, varianta distribuită s-a observat o cantitate foarte mare de mesaje de tip ok redundante. În primul rând s-au identificat mesaje de tip ok de la acelaşi agent, trimise în diverse momente ale rulării, mesaje conţinând aceeaşi valoare sau valori diferite. Aceste mesaje de tip ok sunt redundante, iar unele învechite.

Algoritmul de filtrare prezentat în capitolul 6, figura 6.6, este aplicat de fiecare agent în momentul în care extrage un mesaj de tip ok, înainte de tratarea acestuia. Aplicarea acestui algoritm va permite reducerea efortului local depus de agent, prin reducerea cantităţii de constrângeri verificate.

7.3. Introducerea flagurilor în cazul tehnicilor de căutare din familia ABT.

În acest paragraf se propune o soluţie originală de aplicare a tehnicii flagurilor în cazul tehnicilor din familia ABT. Această soluţie conduce la o generalizare a tehnicii flagurilor. În primul rând se pleacă de la nucleul ABT, pentru care se înlocuieşte stocarea valorilor nogood prin etichetarea valorilor nogood. Apoi, aplicând metodele de eliminare a informaţiilor învechite, se poate ajunge la tehnicile cunoscute: ABT, DisDB, DIBT.

7.3.1. Introducerea tehnicii flagurilor în cazul nucleului ABT.

Analizând fluxul de mesaje back (mesaje ce transportă valorile nogood) ce apar în cazul tehnicilor din familia ABT, au fost identificate aceleaşi tipuri de valori nogood, asemănătoare celor identificate în [GWW03] şi prezentate în paragraful 7.3. Nucleul ABT se comportă asemănător tehnicii asynchronous backtracking în ceea ce priveşte tratarea mesajelor nogood recepţionate. Prin urmare, doar valorile nogood de clasă 1 şi 3 trebuie stocate, cele de clasa 2 şi 4 fiind învechite din aceleaşi motive. Mai mult, valorile nogood de clasă 3 necesită să fie stocate doar temporar.

Plecând de la aceste observaţii, se înlocuieşte stocarea valorilor nogood cu indexarea valorilor locale cu flaguri. Sunt folosite două tipuri de flaguri, corespunzătoare celor două clase de valori nogood:

(1) flag permanent (PF): când se recepţionează un nogood de clasă (1), agentul A_i setează flagul valorii curente cu PF;
(2) flag temporar (TF):când se recepţionează un nogood de clasă (3), agentul A_i setează flagul ataşat valorii curente cu TF.

Din punct de vedere tehnic, fiecare agent va păstra o listă de flaguri asociate valorilor din domeniul său, flaguri de tipurile prezentate anterior. Această listă de flaguri va fi folosită de fiecare agent la selecţia unei valori din domeniul său. Lista de flaguri va fi în permanenţă consistentă cu contextul fiecărui agent. Condiţiile de consistenţă vor consta în verificarea listei myAgentView (care reprezintă contextul fiecărui agent) cu valoarea curentă pentru a satisface toate constrângerile, respectiv flagul asociat valorii curente să nu fie etichetat cu nici una din valorile PF sau TF. Se remarcă faptul că s-a adăugat o condiţie suplimentară pentru consistenţa valorilor, aceea ca flagul corespunzător valorii curente să nu fie etichetat cu una din valorile PF sau TF.

Aceste modificări sunt propuse a se efectua asupra procedurii ABT kernel, procedură din care ulterior se poate obţine, prin eliminarea valorilor învechite, tehnici cunoscute, dar în variantele cu flaguri. Acest lucru va permite generalizarea tehnicii cu flaguri la mai multe tehnici.

În figura 7.1 se propune o variantă originală a algoritmul ABT kernel cu flaguri. Modificările faţă de varianta iniţială ABT kernel prezentată în capitolul 3, sunt etichetate cu caracterul *, iar porţiunile de cod anulate încep cu // şi sunt etichetate cu **.

```
procedure ABTkernel()
1  myValue←empty; end ← false;
2     CheckAgentView();
3     while (not end) do
4       msg ← getMsg();
5       switch(msg:type)
6               Info : ProcessInfo(msg);
7               Back : ResolveConflict(msg);
8               Stop : end← true;
End
procedure CheckAgentView(msg)
1     if not consistent(myValue;myAgentView) then
2               myValue ← ChooseValue();
```

```
3            if (myValue) then for each child ∈ Γ+ (self) do
                    sendMsg:Info(child;myValue);
4            else Backtrack();
End

procedure ProcessInfo(msg)
1     add(newAssig;myAgentView);
// remove Update(myAgentView; msg.Assig); **
2   Clear TF Flags on domain values;        *
2'    CheckAgentView();
End

procedure ResolveConflict(msg)
1     if Coherent(msg.Nogood; Γ- (self) ∪ {self}) then
2     for each assig ∈ lhs(msg.Nogood) \Γ- (self) do
          add(newAssig;myAgentV iew);   *
        //remove  Update(myAgentView; assig);   **
3     //remove add(msg:Nogood;myNogoodStore);   **
4    when nogood only contains {(xi,di)} and is consistent with (xi,current_value)
       do *
            label current_value Flag PF
      end do
5    when nogood is consisten with agent_view and (xi ,current_value) do  *
           label current_value Flag TF
     end do;
6    myValue ← empty;
4     CheckAgentView();
5     else if msg.sender ∈Γ+ ( self) ∧ Coherent(msg.Nogood; self) then
    SendMsg:Info(msg.sender; myValue);
End

procedure Backtrack()
1   newNogood ← {V|V=inconsistent subset of  myAgentView}   *
     //remove newNogood ←   newNogood ← solve(myNogoodStore);  **
2     if (newNogood = empty) then
3            end ←  true; sendMsg:Stop(system);
4     else
5            sendMsg:Back(newNogood, xj);  /* where xj has the lowest priority in V*/
6    Update(myAgentView; xj←  unknown);
6'     Clear TF Flags on domain values;        *
    Label  value-of xj Flag  unknown
7            CheckAgentView();
End

function ChooseValue()
1     for each v ∈ D(self) not eliminated by myFlagList do  *
        /*flagul asociat nu are valoarea PF sau TF */
2       if consistent(v; myAgentView) then return (v);
3       else    label current_value Flag TF      *    /*v is inconsistent with xj 's value */
4         //  remove add(xj  = valj  ) self  ≠ v;myNogoodStore);    **
```

```
5     return (empty);
End

procedure Update(myAgentView; newAssig)
1     add(newAssig;myAgentV iew);
2     for each v ∈ D(self) do
3              if  not consistent (v; myAgentView) then
4             label v Flag TF;
5      else  label v Flag   unknown
End

function Coherent(nogood; agents)
1     for each var ∈ nogood ∪ agents do
2              if   nogood[var] ≠ myAgentView[var] then return false;
3 return true;
end
```

Figura 7.1. Algoritmul ABT kernel cu flaguri aplicabil pentru căutarea asincronă.

În figura 7.1 se observă câteva modificări esenţiale în codul iniţial al nucleului ABT, modificări propuse în acest studiu pentru a putea aplica tehnica cu flaguri. Analiza acestor modificări este prezentată în continuare.

În primul rând, la recepţionarea unui mesaj de tip "info" care informează agentul "self" de schimbarea valorii unui alt agent se face actualizarea contextului local al agentului "self". Aceasta presupune actualizarea listei *myAgentView* şi a listei de flaguri, prin curăţirea flagurilor temporare. Se remarcă faptul că nu mai este nevoie de actualizat lista de nogood-uri, aceasta fiind înlocuită de lista de flaguri.

În al doilea rând, în momentul recepţionării unui mesaj nogood acesta este acceptat doar dacă este coerent cu Γ^- (self) $\cup$ {self}. Apoi se actualizează lista *myAgentView* (adică contextul local) cu valorile agenţilor care sunt în mesajul nogood dar nu sunt în $\backslash\Gamma^-$ (self). Practic, în acest moment se identifică clasa mesajului nogood (fie de clasă (1), fie de clasă (3)), etichetându-se corespunzător flagurile asociate (nogoodurile de clasă (2) şi (4) sunt respinse ca învechite).

O altă modificare apare în corpul procedurii BackTrack, rutină ce este apelată în momentul negăsirii nici unei valori consistente cu contextul agentului "self". În primul rând, noul nogood nu este de tipul "Nogood resolution", ci este construit ca o submultime inconsistentă a contextului agentului (în algoritmul din figura 7.1, V=mulţime inconsistentă a lui myAgentView). Agentul spre care se trimite mesajul nogood este determinat ca fiind A_j, unde A_j are cea mai mică prioritate din V. Cu această ocazie se actualizează flagurile asociate valorilor din domeniul variabilei curente, resetându-se flagurile temporare. În al doilea rând, se resetează flagul ataşat valorii avute de agentul A_j, pentru ca această valoare să poată fi din nou aleasă.

Trebuie făcute câteva observaţii relativ la actualizarea listei cu flaguri. În [GWW03] se propune ştergerea flagurilor temporare, păstrându-se flagurile permanente. În [MusB05] este propusă o îmbunătăţire care constă în restartarea procesului de căutare şi prin resetarea flagurilor permanente, nu numai a celor temporare, idee prezentată anterior. Această idee este aplicată aici, resetându-se şi flagurile permanente şi cele temporare.

Ultima modificare necesară este relativă la funcţia *ChooseValue*, în care trebuie actualizată lista de flaguri şi nu lista de valori nogood stocate.

Aplicarea nucleului ABT la rezolvarea unei probleme modelate DCSP poate conduce la apariţia de informaţii învechite între agenţi [BM05]. Acest lucru are ca şi consecinţă intrarea într-un ciclu infinit a algoritmului, pentru anumite clase şi instanţe de probleme. Plecând de la nucleul ABT, în [BM05], se elimină informaţiile învechite dintre agenţi, prin diverse metode. Se poate pleca de la această procedură şi ajunge la versiuni cunoscute sau apropiate (cum ar fi ABT, DIBT şi DisDB).

Aplicând aceleaşi metode de eliminare a informaţiilor asupra nucleului ABT cu flaguri, se poate ajunge la diverse variante de tehnici, variante ce folosesc flaguri în locul stocării nogood-urilor. În continuare sunt analizate două tehnici foarte cunoscute, la care se poate ajunge. Variantele obţinute vor fi numite "asynchronous backtracking cu flaguri" (ABTWFL), "distributed dynamic backtracking cu flaguri "(DisDBWFL).

7.4.2. Asynchronous Backtracking cu flaguri (ABTWFL).

O primă soluţie de eliminare a informaţiilor învechite este de a adăuga noi legături pentru a permite agentului ce a recepţionat un mesaj nogood să decidă dacă este sau nu învechit [BM05]. Aceste legături au fost propuse în algoritmul original ABT. În urma aplicării acestei metode de eliminare a informaţiilor învechite nucleului ABT varianta cu flaguri prezentată anterior, se obţine o nouă tehnică hibridă numită „asynchronous backtracking cu flaguri"-ABTWFL.

Noile legături trebuie adăugate în timpul căutării. O astfel de legătură este necesară a fi adăugată între doi agenţi A_i şi A_k, neconectaţi iniţial. Această situaţie apare în cazul unui agent A_i în momentul recepţionării unui mesaj de tip Back (nogood) ce conţine informaţii relative la un alt agent A_k cu care nu este conectat, agent aflat înaintea lui conform ordinii iniţiale. Noile legături vor deveni permanente permiţând agenţilor să recepţioneze mesaje info şi în felul acesta să elimine informaţiile învechite dintre agenţi. Modificările necesare în ABTkernel cu flaguri, pentru a elimina informaţiile învechite, sunt aceleaşi cu cele prezentate în [BM05], pentru a obţine tehnica asynchronous backtracking. Aplicarea acestor modificări conduc la variantă hibridă ABTWFL.

7.4.3. Distributed Dynamic Backtracking cu flaguri (DisDBWFL).

O a doua soluţie, prezentată în [BM05] constă în detectarea situaţiei în care un anumit nogood este sau nu învechit. Practic, un nogood ipotetic învechit şi toate valorile recepţionate de la acei agenţi cu care nu a fost conectat sunt uitate.

În loc să încerce să se informeze când un agent neconectat şi-a schimbat valoarea, agentul curent poate studia cursul evenimentelor şi actualiza, dacă este cazul, informaţiile sale. Mai exact, când toate valorile sale au fost încercate, un nou nogood este construit şi trimis la agentul cel mai apropiat, agent vinovat de apariţia acestei situaţii. Acest mesaj va forţa agentul ce va recepţiona mesajul să-şi schimbe valoarea. Pentru acele variabile ce sunt în Γ^- (self), acest lucru nu trebuie neapărat realizat pentru că agenţii responsabili sunt legaţi prin legături ce le permit recepţionarea de mesaje info. În schimb, pentru alţi agenţi cu care nu este conectat, aceste acţiuni din backtracking pot conduce la învechirea nogoodurilor apărute. Din acest motiv, în momentul apelării procedurii backtracking, se renunţă la acele nogooduri şi la valorilor asociate acestor variabile [BM05].

Aplicând această metodă la nucleul ABT (varianta cu flaguri), se ajunge la tehnica Distributed Dynamic Backtracking cu flaguri (DisDBWFL). Trebuie amintit că există o anumită deosebire faţă de varianta derivată din [BM05]: noul nogood nu este de tipul "nogood resolution", ci este construit ca o submulţime inconsistentă a contextului agentului (lista myAgentView).

7.4. REZULTATE EXPERIMENTALE.

În acest paragraf sunt prezentate rezultatele experimentale obţinute în urma implementării şi evaluării tehnicilor asincrone prezentate. Pentru a face evaluarea performanţelor tehnicilor construite, aceste tehnici au fost implementate folosind modelul cu sincronizarea execuţiei agenţilor (SEIS), model ce permite şi contorizarea numărului de ciclii necesari obţinerii soluţiei. Tehnicile asincrone au fost aplicate pentru problema colorării unui graf în varianta distribuită (n=15, respectiv n=20 noduri). S-a utilizat cadrul experimental prezentat în paragraful 6.1.

Dintre unităţile de măsură prezentate în paragraful 6.1, s-a contorizat fluxul de mesaje (adică numărul de mesaje info şi back schimbate de agenţi), numărul de constrângeri verificate, numărul de constrângeri concurente verificate (notat cu c-ccks) şi numărul de cicli necesari pentru obţinerea soluţiei. S-a contorizat numărul de valori nogood stocate (respectiv listele cu flaguri pentru versiunile cu flaguri). Evaluările au fost făcute pentru cele şapte tehnici prezentate (variantele iniţiale cu stocarea nogoodurilor şi variantele cu flaguri).

7.4.1. Analiza rezultatelor experimentale în cazul tehnicii Asynchronous Backtracking cu flaguri (varianta Yokoo)

Cele trei versiuni evaluate (derivate din varianta de bază Yokoo) au fost notate cu ABT, ABTWFL şi ABTWFL-îmbunătăţită (variantă propusă în paragraful 7.3). Valorile obţinute pentru cele două clase de grafuri (cu 15, respectiv 20 de noduri) sunt prezentate în tabelul 7.1.

Prima unitate de măsură folosită pentru analiza performanţelor celor trei versiuni de tehnici a fost ciclul. Această unitate de măsură permite evaluarea efortului global depus de agenţi. Modelul utilizat la implementare a permis contorizarea numărului de cicli. Din analiza valorilor, se observă că cele trei versiuni au valori apropiate în ce priveşte numărul de cicli necesari. Varianta cu flaguri îmbunătăţită se detaşează pentru grafuri de dimensiune mai mare şi densităţi mari. Acest fapt arată un comportament bun în cazul problemelor dificile cu dimensiuni mari.

În ceea ce priveşte efortul local depus de fiecare agent analizând valorile din tabel, se remarcă o diferenţă mare între tehnica de bază ABT (Yokoo) şi cele două variante cu flaguri, acestea necesitând un efort local mult mai mic (aproape jumătate). Varianta propusă se detaşează prin faptul că ea implică cel mai mic efort în găsirea unei soluţii.

În ce priveşte fluxul de mesaje, cele trei variante necesită un număr apropiat de mesaje pentru probleme cu dimensiune şi densitate mică. În schimb, pentru cazul problemelor cu dimensiune mai mare şi densitate mare, variantele cu flaguri au necesitat un flux de mesaje mai mic. Trebuie remarcat faptul că numărul de valori nogood stocate, respectiv numărul de flaguri pentru variantele cu flaguri sunt apropiate, dar cantitatea de memorie necesară este mult mai mică pentru ultimele variante.

		n = 15		n = 20	
		m=n x2	m=n x2.7	m=n x2	m=n x2.7
ABT	TNogood	35.75	201.24	190.58	976.81
	TOk	161.87	601.15	705.89	3423.86
	TNGS	16.54	52.02	83.87	273.61
	Tciclii	36.49	139.98	148.95	506.74
	TCcks	8162.67	34063.76	50940.36	246900.42
	Tc-ccks	1995.91	7882.72	11082.42	47391.03
ABTWFL	TNogood	34.10	161.83	192.15	1098.30
	TOk	158.47	503.78	712.89	3856.38
	TNGS	15.57	42.37	81.57	297.68
	Tciclii	34.46	114.20	144.39	566.46
	TCcks	3506.95	13451.21	22367.05	113534.07
	Tc-ccks	1055.13	3686.45	6356.22	28211.31
ABTWFL-îmbunătăţită	TNogood	33.88	161.44	192.12	851.41
	TOk	157.49	500.28	708.89	2914.48
	TNGS	15.52	41.72	83.88	242.51
	Tciclii	33.24	113.71	144.47	441.92
	TCcks	3281.64	13084.59	22520.98	86663.49
	Tc-ccks	979.64	3664.35	6327.31	22021.41

Tabelul 7.1. Rezultatele experimentale pentru versiunile de Asynchronous Backtracking (versiunea de bază -[YDIK98]) în cazul introducerii flagurilor.

Aplicând algoritmul de filtrare propus în paragraful 6.5.2 s-au obţinut alte trei variante de tehnici asincrone pentru care s-a executat un număr de 100 de rulări (în condiţiile utilizării modelului cu sincronizare). Valorile obţinute pentru cele trei versiuni de tehnici în condiţiile filtrării mesajelor sunt prezentate în tabelul 7.2.

Din analiza valorilor din tabelul 7.2 se remarcă faptul că cele două variante cu flaguri, dar în special varianta propusă în acest studiu, necesită costuri de calcul mai mici Se remarcă faptul că pentru probleme de dimensiune şi dificultate mare varianta propusă în acest studiu necesită costuri mult mai mici faţă de varianta de bază şi chiar faţă de varianta iniţială cu flaguri.

		n = 15		n = 20	
		m=n x2	m=n x2.7	m=n x2	m=n x2.7
ABT cu filtrare	TNogood	28.84	71.92	151.33	520.12
	TOk	127.71	235.78	542.53	1764.26
	TNGS	18.06	34.91	100.35	263.15
	Tciclii	33.42	65.1	137.72	348.98
	TCcks	6305.53	12110.44	38298.7	117686.39
	Tc-ccks	1358.14	2632.87	7312.32	19630.17
ABTWFL cu Filtrare	TNogood	24.79	61.26	113.34	368.43
	TOk	117.93	206.82	438.50	1268.3
	TNGS	15.06	28.88	72.05	181.03
	Tciclii	30.07	55.97	102.52	242.87
	TCcks	2686.15	5581.6	13551.87	40239.12
	Tc-ccks	722.58	1421.56	3578.68	8879.37

ABTWFL-îmbunătăţită şi cu filtrare	TNogood	22.31	59.09	105.69	343.23
	TOk	110.49	203.7	401.21	1179.68
	TNGS	13.31	28.02	64.54	167.48
	Tciclii	27.49	54.2	92.46	222.07
	TCcks	2282.42	5066.64	12362.26	35693.79
	Tc-ccks	627.22	1309.52	3306.3	8079.38

Tabelul 7.2. Rezultatele experimentale pentru versiunile de Asynchronous Backtracking (versiunea [YDIK98]) cu filtrarea mesajelor.

7.5.2. Analiza rezultatelor experimentale în cazul tehnicilor din familia ABT cu flaguri.

Pe baza modelului cu sincronizare s-a implementat nucleul ABT în ambele variante (varianta de bază şi cea cu flaguri), nucleu prezentat în paragraful 7.4.1. La acest nucleu s-au aplicat cele două metode de eliminare a informaţiilor învechite obţinând patru variante de tehnici. Cele patru variante evaluate din familia ABT (Bessiere) au fost numite ABT (Bessiere), ABTWFL, DisDB şi DisDBWFL. Pentru evaluare s-au ales trei clase de grafuri cu dimensiune mai mare. Valorile obţinute pentru trei clase de grafuri (cu 20, 25 respectiv 30 de noduri) sunt prezentate în tabelul 7.3 (versiunile ABT) şi tabelul 7.4. (versiunile DisDB).

Suplimentar, s-a contorizat şi numărul de constrângeri concurente verificate de fiecare agent. Acesta permite evaluarea efortului local, fără a lua în calcul faptul că agenţii lucrează concurent (informal, numărul de constrângeri concurente verificate, aproximează cea mai lungă secvenţă de constrângeri verificate neexecutate concurent).

Din analiza valorilor din cele două tabele se remarcă faptul că cele două versiuni (versiunea de bază şi cea cu flaguri) au valori apropiate în ce priveşte numărul de cicli necesari obţinerii soluţiei, remarcând totuşi performanţe mai bune pentru versiunile cu flaguri. Versiunea cu flaguri se detaşează pentru grafuri cu dimensiune mare şi densitate mare. Acest lucru arată un comportament bun în cazul problemelor dificile.

		n = 20		n = 25		n = 30	
		m=n x2	m=n x2.7	m=n x2	m=n x2.7	m=n x2	m=n x2.7
ABT	TNogood	279.70	928.05	1213.23	1393.19	4152.59	5498.03
	TOk	1012.81	3242.26	4205.35	5389.66	14648.67	23741.29
	TNgs	137.55	289.61	592.42	477.98	1719.31	1546.79
	Ciclii	211.52	490.74	634.17	736.62	2294.85	2427.14
	TCcks	59372	186777	368304	368304	1328756	1892342
	Tc-ccks	12393	34358	48201	56843	270280	240830
ABTWFL	TNogood	242.51	621.54	694.04	972.76	2859.54	3004.56
	TOk	908.85	2218.59	2684.29	3677.43	10091.74	13021.04
	TNgs	119.20	194.80	338.98	343.35	1207.56	882.73
	Ciclii	183.09	329.63	456.06	472.31	1585.63	1313.81
	TCcks	24535	67371	91516	115722	426475	406482
	Tc-ccks	7349	18365	32865	27624	119371	79695

Tabelul 7.3. Rezultatele experimentale pentru versiunile ABT- (Bessiere) în cazul introducerii flagurilor.

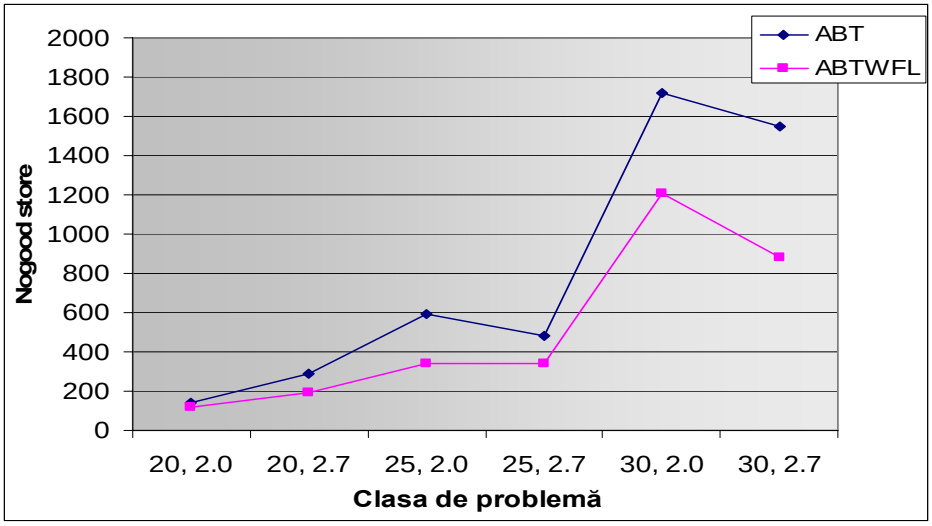


Figura 7.2 Rezultate comparative pentru versiunile ABT(Bessiere) în raport cu numărul de mesaje de tip nogood stocate

În ceea ce priveşte efortul local depus de fiecare agent (evaluat prin contorizarea constrângerilor verificate, respectiv numărul de constrângeri concurente verificate), analizând aceste valori, se observă aceleaşi diferenţe între tehnicile de bază şi cele cu flaguri, acestea din urmă necesitând un efort local aproape la jumătate.

În figurile 7.2. şi 7.3. sunt prezentate evoluţiile fluxului de mesaje nogood stocate de fiecare agent, respectiv numărul de flaguri setate pentru versiunile cu flaguri. Folosirea acestei tehnici a permis reducerea fluxului de mesaje nogood şi în plus reducerea cantităţii de memorie necesară pentru stocarea informaţiilor din nogood-uri, pentru ambele tehnici derivate.

		n = 20		n = 25		n = 30	
		m=n x2	m=n x2.7	m=n x2	m=n x2.7	m=n x2	m=n x2.7
DisDB	TNogood	288.12	824.89	697.92	1066.35	3397.69	5956.57
	TOk	629.39	1927.99	1393.93	2728.78	6980.20	13620.4
	TNGS	168.88	330.75	402.67	474.35	1951.72	2442.77
	Tciclii	158.05	403.42	400.58	534.55	1814.48	1770.23
	TCcks	38258.7	112446.8	122533	178043	719449	1196721
	Tc-ccks	9364.67	25255.1	31541.9	33845.2	184218	186897
DisDBWFL	TNogood	215.38	648.08	446.74	775.30	1976.57	6434.89
	TOk	480.37	1502.50	839.75	1974.85	4601.43	14589.8
	TNGS	127.51	269.74	238.72	346.38	1046.07	2862.95
	Tciclii	115.50	284.23	233.32	327.37	1053.68	1912.38
	TCcks	24723.5	73457.67	52843	103938.2	401034	1081615.
	Tc-ccks	6357.37	16447.37	13569.4	19069	107099	169949

Tabelul 7.4. Rezultatele experimentale pentru versiunile DisDB în cazul introducerii flagurilor.

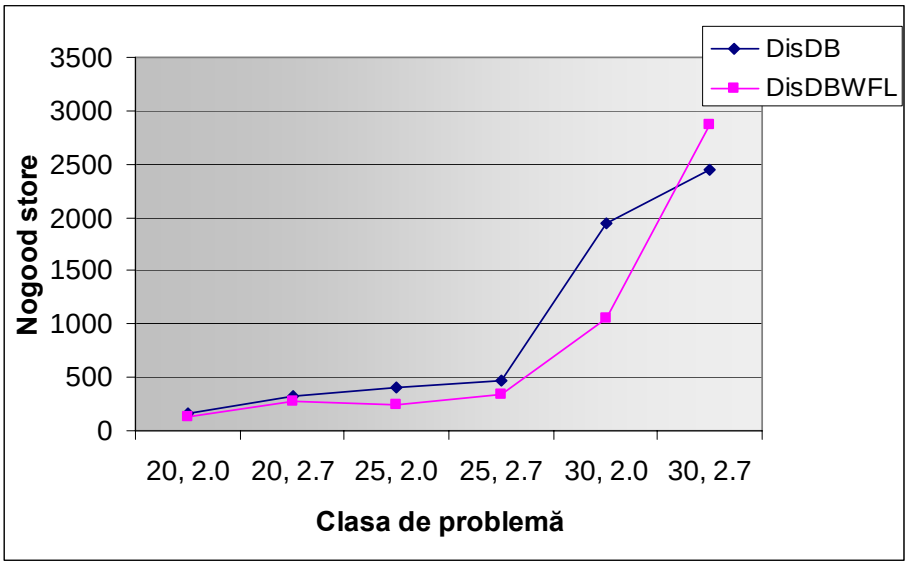


Figura 7.3. Rezultate comparative pentru versiunile DisDB în raport cu numărul de mesaje de tip nogood stocate

În ceea ce priveşte fluxul de mesaje, numărul de mesaje schimbate a fost foarte apropiat pentru problemele cu dimensiune mică şi densitate rară. Dar, în cazul problemelor cu densitate şi dimensiune mare, versiunile cu flaguri necesită un flux mai mic de mesaje. Relativ la numărul de mesaje nogood stocate (respectiv liste de tip flag), cele două variante au valori apropiate, dar cantitatea de memorie pentru versiunile cu flaguri este mult mai mică.

7.5. Concluzii.

Creşterea numărului de mesaje de tip nogood înregistrate determină o creştere a complexităţii algoritmilor de căutare asincroni în cazul cel mai nefavorabil. Ca o consecinţă, timpul de calcul şi necesităţile hardware cresc foarte repede odată cu dimensiunea problemelor, studiul prezentat propunându-şi să elimine stocarea valorilor nogood prin introducerea etichetării valorilor.

Tehnicile din familia ABT se caracterizează printr-o explozie a valorilor nogood, mai ales pentru probleme de dimensiune mare. O altă caracteristică specifică tehnicilor asincrone din familia ABT, este aceea că este necesară stocarea valorilor nogood pentru ca agenţii să nu mai repete aceeaşi greşeală. Acest lucru conduce la o cantitate de memorie foarte mare, mai ales pentru probleme cu dimensiune mare. O soluţie de eliminare a exploziei valorilor nogood este aceea de etichetare a valorilor din domeniul fiecărei variabile cu flaguri.

Tehnica ABT cu flaguri este o tehnică derivată din tehnica de bază ABT (varianta Yokoo). Ea implică etichetarea valorilor locale din domeniul fiecărui agent, cu flaguri. Acest lucru permite eliminarea stocării valorilor nogood şi, prin urmare, pentru probleme de dimensiune mare, evitarea exploziei valorilor nogood. Spaţiul de

memorie necesar se reduce considerabil în cazul aplicării tehnicii cu flaguri. Această tehnică cu flaguri poate fi îmbunătăţită prin restartarea procesului de căutare, în cazul recepţionării unui mesaj de tip ok, îmbunătăţire propusă în acest studiu. Experimentele prezentate arată o eficienţă mărită în cazul acestei variante propuse, dar fără a aduce rezultate spectaculoase.

În acest studiu este propusă o soluţie originală de înlocuire a stocării valorilor nogood cu etichetarea valorilor din domeniu prin două tipuri de flaguri. Această soluţie este aplicată pentru nucleul ABT, obţinându-se o nouă variantă ce nu mai necesită stocarea valorilor nogood. Plecând de la acest nucleu se poate ajunge la două tehnici cunoscute: asynchronous backtracking cu flaguri (varianta Bessiere) şi Distributed Dynamic Backtracking cu flaguri.

Variantele de tehnici propuse în acest studiu (ABT şi DisDB cu flaguri) sunt obţinuţi din tehnica ABT kernel prin eliminarea informaţiilor învechite dintre agenţi şi indexarea valorilor locale din domeniul unui agent cu flaguri. Acest lucru elimină stocarea valorilor nogood şi, prin urmare, pentru probleme de dimensiune mare, eliminarea exploziei valorilor nogood. Spaţiul de memorie alocat este considerabil redus în cazul varintelor cu flaguri.

Tehnicile cu flaguri pot fi îmbunătăţite prin restartarea procesului de căutare când sunt recepţionate mesaje de tip info sau back, soluţie propusă în acest capitol. Experimentele arată o îmbunătăţire, dar fără a avea rezultate experimentale spectaculoase. Aceste experimente, realizate în condiţiile unor variate probleme, cu densităţi variate, cu dimensiuni mici şi mari, cu valori iniţiale aleatoare, arată că tehnicile cu flaguri necesită mai puţin spaţiu de stocare decât tehnicile de bază. Mai mult, o reducere a costurilor este de notat în cazul căutării soluţiei.

Ca o concluzie generală, introducerea flagurilor a permis eliminarea efectului exploziei valorilor nogood.

Rezultatele acestui capitol au fost publicate în [Mus05] şi [MusB05]

8. DETERMINAREA NUMĂRULUI OPTIM DE MESAJE TRANSMISE PENTRU LEGĂTURILE TEMPORARE ÎN CAZUL TEHNICILOR ASINCRONE DIN FAMILIA ABT

Scopul acestui studiu este de a investiga comportamentul tehnicii ABT_{temp} pentru diferite valori ale numărului de mesaje transmise pentru legăturile temporare Pe baza acestei analize, în acest capitol sunt identificate şi prezentate mai multe soluţii originale de determinare a numărului de mesaje schimbate de agenţii cu legături temporare, număr optim de mesaje pentru care trebuie păstrată o legătură temporară. Este propusă o soluţie dinamică de determinare a numărului de mesaje necesare pentru păstrarea unei legături, experimentele arătând o reducere a costurilor relativ la varianta de bază ABT.

Capitolul începe cu o prezentare a contextului de lucru, apoi sunt prezentate două tipuri de soluţii pentru numărul de mesaje transmise pentru legăturile tempoare: soluţii statice şi soluţii dinamice. Folosind modelul cu sincronizare şi soluţiile de evaluare propuse în capitolul 3 sunt implementate şi evaluate variantele obţinute pentru tehnica ABT cu legături temporare. În final sunt prezentate concluziile determinării numărului de mesaje.

8.1 Analiza comportamentului tehnicii ABT cu legături temporare.

Având ca bază de plecare algoritmul asynchronous backtracking (ABT) în [BMM05] a fost propus un cadru unificator, un nucleu de plecare pentru câteva dintre cele mai importante şi cunoscute tehnici asincrone. Din acest nucleu au fost derivate mai multe tehnici, tehnici cunoscute sub numele de familia ABT. Ele diferă prin modul în care adaugă legături între agenţii neconectaţi cu scopul de a detecta şi elimina informaţiile învechite dintre agenţi.

Există mai multe soluţii de eliminare a informaţiilor învechite dintre agenţi. O primă soluţie prezentată în [BMM05] de eliminare a informaţiilor învechite este de a adăuga noi legături pentru a permite agentului ce a recepţionat un mesaj nogood să decidă dacă este sau nu învechit. Legăturile care se adaugă în timpul rulării pot deveni permanente sau temporare. Păstrarea temporară a legăturilor conduce la un algorim derivat numit ABT_{temp}. În literatura de specialitate nu apar referinţe despre studii care să determine o valoare optimă pentru acel număr de mesaje. Acest studiu investighează diverse valori pentru numărul de mesaje, valori ce se determină fie static (înainte de rulare), fie dinamic în timpul rulării. Folosind modelul de implementare cu sincronizare este investigat comportamentul tehnicii ABT_{temp} pentru diferite valori ale numărului de mesaje având ca ţintă identificarea unei valori optime ce permite reducerea costurilor de găsire a soluţiei.

Pentru a putea analiza comportamentul acestei tehnici cu legături temporare, tehnică derivată din nucleul ABT, s-au efectuat mai multe experimente cu diverse valori ale numărului de mesaje [MusPP05a]. Pentru aceasta, s-au ales

două clase de probleme de dimensiune mare (30 şi 40 de noduri/agenţi), la fiecare clasă executându-se rulări pentru diverse valori ale numărului de mesaje. În figura 8.1. se prezintă valorile obţinute pentru diverse valori ale numărului de mesaje (s-a contorizat numărul de mesaje schimbate şi numărul de constrângeri concurente verificate de agenţi).

Experimentele arată că pentru fiecare clasă de probleme există o valoare a numărului de mesaje, valoare pentru care tehnica ABT kernel cu legături temporare se comportă asemănător cu tehnica de bază ABT. De asemenea, din analiza experimentală a valorilor, se observă că există o altă valoare a numărului maxim de mesaje, pentru care se obţin performanţe mai bune decât varianta de bază ABT, variantă derivată din ABT kernel.

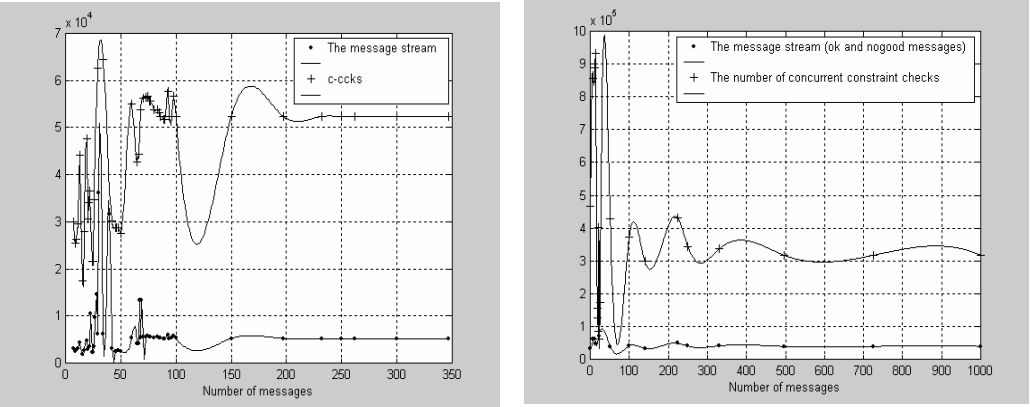


Figura 8.1. Efortul de calcul pentru diverse valori ale numărului de mesaje, în cazul tehnicii ABT cu legături temporare

8.2. Determinarea valorii numărului de mesaje ce trebuie transmise pentru legăturile temporare.

În acest paragraf se vor prezenta mai multe soluţii de determinare a numărului de mesaje schimbate de agenţii cu legături temporare, număr de mesaje pentru care trebuie păstrată o legătură temporară.

Plecând de la comportamentul identificat în paragraful anterior sunt propuse două tipuri de soluţii: soluţii statice (în care numărul de mesaje este fixat şi nu se schimbă de-a lungul rulării algoritmului) şi soluţii dinamice (în care valoarea numărului de mesaje variază de la începutul rulării până la final).

8.2.1. Soluţii statice pentru determinarea numărului optim de mesaje.

Soluţiile statice propuse în acest studiu se bazează pe determinarea pentru fiecare agent, înainte de rulare, a valorii numărului de mesaje ce trebuie transmise pentru o legătură. Acest număr poate fi determinat în mai multe moduri şi este folosit ca număr maxim de mesaje transmise.

Soluţiile statice presupun construirea grafului indus asociat problemei (într-o fază de preprocesare). Fiecărei probleme DCSP i se poate asocia un graf al constrângerilor, în care nodurile sunt agenţi/variabile şi arcele depind de existenţa

constrângerilor dintre agenţi/variabile. Din acest graf al constrângerilor noi se poate obţine *graful indus*, relativ la ordinea existentă între agenţi, prin adăugarea unor legături între părinţii fiecărui nod, dacă acestea nu există. Acest graf indus se construieşte ca şi în [DP88] şi a fost prezentat în capitolul 2, remarca 2.2.2. Pe baza acestui graf se poate determina un număr de mesaje fixe pentru fiecare agent, după cum urmează:

- prima soluţie consideră că numărul de mesaje va fi egal cu numărul de vecini în graful indus, pentru fiecare agent. Fiecare agent va păstra o anumită legătură temporară până când va schimba un număr de mesaje egal cu numărul său de vecini. Această soluţie va fi denumită *#S1.*
- a doua soluţie se bazează pe calculul unei valori globale comune pentru toţi agenţii, valoare obţinută ca cea mai mare valoare dintre numărul vecinilor fiecărui agent. Această soluţie va fi denumită *#S2*

Experimentele ce vor fi prezentate în paragraful următor vor arăta că pentru probleme cu dimensiune mică soluţiile sunt eficiente, dar odată cu creşterea dimensiunii problemelor, varianta de bază este mai performantă decât variantele cu număr temporar de legături (cu soluţii statice).

8.2.2. Soluţii dinamice pentru determinarea numărului optim de mesaje.

Prin urmare, cea mai bună soluţie, observată şi experimental, este de a determina dinamic numărul maxim de mesaje în timpul rulării, acesta neavând o valoare fixă. El trebuie să fie variabil, pentru fiecare agent în parte, o valoare maximă generală nefiind cea mai bună alegere. Practic, se deduce dinamic în timpul rulării valoarea numărului de mesaje şi ea este ajustată în funcţie de evoluţia algoritmului. Această soluţie va fi propusă mai departe la prezentarea soluţiilor dinamice.

Variantele dinamice propuse se bazează pe folosirea informaţiilor legate de fluxul de mesaje nogood învechite. Aceste informaţii se schimbă de-a lungul rulării algoritmului. După cum se cunoaşte din [BMM05] când un agent A_k recepţionează un mesaj nogood el este acceptat dacă este consistent cu vederea sa, altfel este considerat ca fiind învechit. Fluxul de mesaje învechite creşte şi datorită faptului că agenţii nu sunt informaţi (datorită inexistenţei acelor legături suplimentare). Prin urmare, fiecare agent foloseşte o structură de date suplimentară în care reţine numărul de mesaje învechite recepţionate de la fiecare agent cu care este conectat. Aceste valori sunt folosite pentru a determina numărul de mesaje schimbate pentru fiecare legătură temporară. Practic, această valoare se determină ca fiind cel mai mare număr de mesaje nogood învechite recepţionate la un moment dat. Aceasta este prima soluţie dinamică propusă în acest studiu, denumită #D1.

A doua soluţie dinamică propusă în acest studiu, denumită #D2, se obţine din prima soluţie dinamică, prin folosirea informaţiilor legate de numărul de vecini ai fiecărui agent. În cazul primei soluţii, în prima parte a rulării algoritmului, fluxul de mesaje nogood învechite este foarte mic, având ca efect o valoare mică pentru numărul de mesaje pentru care se păstrează o legătură. Prin urmare, se porneşte cu o valoare fixă pentru numărul de mesaje, egală cu cel mai mare număr de vecini din graful indus. Această valoare iniţială este actualizată pe parcurs, folosind cea mai mare valoare a numărului de mesaje învechite a tuturor agenţilor.

Pentru a îmbunătăţi eficienţa celor două variante dinamice propuse, fiecare agent încearcă să păstreze cât mai mult anumite legături temporare. Anumite

legături temporare cu acei vecini cu care a schimbat un flux de mesaje nogood învechite maxim sunt păstrate suplimentar. Practic, pentru fiecare agent se determină agentul cu care a avut un flux de mesaje învechite maxim (în procedura *CheckAgentView*). Legătura cu acesta este păstrată suplimentar, până se identifică un alt agent cu care a avut un flux de mesaje învechite maxim.

8.2.3 Determinarea numărului maxim de mesaje învechite recepţionate de agenţi.

Cele două soluţii dinamice presupun cunoaşterea de către fiecare agent a numărului maxim de mesaje învechite recepţionate de fiecare agent. O soluţie ar consta în existenţa unui agent central, care să stocheze informaţii legate de numărul maxim de mesaje învechite receţionate de fiecare agent. Acest agent ar putea păstra o listă cu maximele transmise de fiecare agent. Din păcate această soluţie măreşte costurile soluţiei pentru că necesită ca fiecare agent să transmită agentului central maximele relative la fluxul de mesaje învechite.

O a doua soluţie propusă în acest capitol, mult mai practică, este bazată pe transmiterea maximelor fiecărui agent la cei cu care este conectat în momentul transmiterii unui mesaj de tip „info" sau „nogood". Ideea este asemănătoare celei din algoritmul de determinare a costului cumulativ din [MKRZ02].

În figura 8.2 se prezintă un algoritm original de determinare a „maximului" dintre maximele fluxului de mesaje învechite. Fiecare agent păstrează o listă locală de variabile contor (C_OldNogood) având rol de a reţine numărul de mesaje învechite recepţionate de la agenţi. De asemenea, fiecare agent păstrează şi un contor *MaxOldNogood* ce va reţine maximul numărului de mesaje dintre vecini.

Determinarea numărului maxim de mesaje învechite recepţionate de agenţi

1: Fiecare agent iniţializează variabilele contor *C_OldNogood* cu 0.
De asemenea, se iniţializează *MaxOldNogood* cu 0.
2: Când un agent transmite un mesaj el include şi valoarea contorului său *MaxOldNogood*.
3: Când un agent recepţionează o valoare nogood învechită de la un agent *Sender* se actualizează contorul corespunzător din lista *C_OldNogood*.
Replace-item Sender *C_OldNogood* with item Sender *C_OldNogood* + 1
4: if an agent receives a message with a counter *SenderMx* then
Set MaxOldNogood = max { *C_OldNogood* }
if *MaxOldNogood* < *SenderMx* then
Set *MaxOldNogood* = *SenderMx*
end if
end if

Figura 8.2 Determinarea numărului maxim de mesaje învechite recepţionate de agenţi

Acest algoritm se aplică asupra tehnicii ABT, permiţând fiecărui agent să reţină valoarea maximă a numărului de mesaje nogood învechite schimbate de agenţi.

8.2.4 Adaptarea tehnicii ABT cu legături temporare

Aplicarea algoritmului din figura 2 şi limitarea fluxului de mesaje a implicat anumite modificări asupra tehnicii ABT cu legături temporare.

În figura 8.3. se prezintă acele modificări necesare în codul tehnicii ABT (varinta derivată din ABT kernel), pentru a putea aplica metodele prezentate anterior pentru numărul de mesaje, în special pentru aplicarea soluţiilor dinamice de determinare a numărului de mesaje. Modificările sunt marcate cu ***.

```
procedure ABTkernel()
1  myValue←empty; end ← false;
1.1.  Set Γe+ (self) ← ∅                          ***
1.2.  Set Γe- (self) ← ∅                           ***
2      CheckAgentView();
3      while (not end) do
4        msg ← getMsg();
5        switch(msg:type)
6                Info : ProcessInfo(msg);
7                Back : ResolveConflict(msg);
8                Stop : end← true;
9.1.            AddL : SetLink(msg);
9.2.            RemoveL: RemoveLink(msg);                ***
End
procedure ProcessInfo(msg)
1      Update(myAgentView; msg.Assig);
2      CheckAgentView();
End
procedure CheckAgentView(msg)
1      if not consistent(myValue;myAgentView) then
2                myValue ← ChooseValue();
3                if (myValue) then
                   for each child ∈ Γ+ (self) do
                       sendMsg:Info(child;myValue);
3.1                    CheckRemoveLink(Self, Child)          ***
4                else
                   Backtrack();
End
procedure ResolveConflict(msg)
1      if Coherent(msg.Nogood; Γ- (self) ∪ {self}) then
2.1              CheckAddLink(msg);
3                add(msg:Nogood;myNogoodStore); myValue ← empty;
4                CheckAgentView();
5.1    else if  Coherent(msg.Nogood; self) then
                    SendMsg:Info(msg.sender; myValue);
5.2.     replace item Sender C_OldNogood with item Sender C_OldNogood+1 ***
End
procedure SetLink(msg)
1      add(msg.sender; Γ+ (self)); add(msg.sender; Γe+ (self));
2      sendMsg:Info(msg.sender;myV alue);
End
procedure CheckAddLink(msg)
1      for each (var ∈ lhs(msg.Nogood))
2            if (var ∉ Γ- (self)) then
```

```
3               sendMsg:AddL(var, self);
4               add(var; Γ⁻ (self)); add(var; Γe⁻ (self));
5               Update(myAgentView; var ← varValue);
End
procedure RemoveLink(msg)                    ***
1    remove(msg.sender; Γ⁻ (self));
2    remove(msg.sender; Γe⁻ (self));
End
procedure CheckRemoveLink(msg)               ***
1   for each child ∈ Γe⁺ (self) do
2         if (item child CMessageTempLink)= MaxOldNogood then
              remove(child; Γ⁺ (self)); remove(child; Γe⁺ (self));
3              sendMsg:RemoveL(child, self);
4              Update(myAgentView; child ← unknown);
End
procedure Backtrack()
1     newNogood ← solve(myNogoodStore);
2     if (newNogood = empty) then
3              end ←  true; sendMsg:Stop(system);
4     else
5              sendMsg:Back(newNogood);
6              Update(myAgentView;rhs(newNogood)←  unknown);
7              CheckAgentView();
End
function ChooseValue()
1     for each v ∈ D(self) not eliminated by myNogoodStore do
2              if consistent(v; myAgentView) then
                 return (v);
3              else
                 add(xj  = valj  ) self  ≠ v;myNogoodStore);
                            /*v is inconsistent with   xj 's value */
4     return (empty);
End
procedure Update(myAgentView; newAssig)
1     add(newAssig;myAgentV iew);
2     for each ng ∈ myNogoodStore do
3              if  not Coherent(lhs(ng);myAgentView) then
                 remove(ng;myNogoodStore);
End
function Coherent(nogood; agents)
1     for each var ∈ nogood ∪ agents do
2              if   nogood[var] ≠ myAgentV iew[var] then return false;
3 return true;
end
```

Figura 8.3. Algoritmul ABT cu legături temporare.

Câteva precizări trebuie făcute, pentru o bună înţelegere a modificărilor necesare în codul ABT (variantă derivată din ABT kernel), modificări realizate pentru obţinerea variantei ABT cu legături temporare.

În primul rând, fiecare agent va utiliza două mulţimi suplimentare Γ_e^+ (self) şi Γ_e^- (self), pentru identificarea agenţilor copil şi părinte ce apar în urma legăturilor temporare. În procedura ABTkernel(), în liniile 1.1. şi 1.2. ei sunt determinaţi. De asemenea, este nevoie de introducerea a două structuri de date *CMessageTempLink* şi *C_OldNogood***.** Prima structură va fi folosită de un agent A_i pentru a reţine numărul de mesaje info transmise pentru fiecare legătură temporară. A doua structură se foloseşte la contorizarea fluxului de mesaje învechite.

În al doilea rând, este necesară introducerea unui al patrulea mesaj *RemoveL*, care anunţă un agent "copil" de anularea legăturii temporare dintre cei doi agenţi. Acest lucru implică şi ştergerea agentului ce a transmis mesajul din lista de părinţi a agentului copil Modificările necesare apar în linia 8.3. din procedura ABTkernel() şi în procedura RemoveLink(), (aceasta este o nouă rutină adăugată în codul iniţial ABT kernel).

În al treilea rând, după selectarea unei noi valori şi anunţarea copiilor despre noua selecţie, este necesară verificarea legăturilor temporare pentru a determina câte dintre din ele mai rămân actuale. Acest lucru este făcut în procedura *CheckAgentView(msg)* linia 3.1, prin apelarea unei noi proceduri cu numele *CheckRemoveLink(Self, Child)***.** Această rutină verifică, pentru agenţii copil din Γ_e^+ (self), dacă s-a atins numărul maxim de mesaje transmise pentru aceea legătură. Acest număr maxim se poate determina cu oricare dintre metodele propuse în acest articol. În caz că s-a atins valoarea maximă *MaxOldNogood* se anulează acea legătură şi se actualizează contextul de lucru.

În figura 8.4 este prezentat fluxul mesajelor, în care apar noile mesaje (mesaje de tip *AddL* şi *RemoveL* ce permit gestiunea legăturilor noi). Sunt evidenţiate porţiunile de cod ce sunt parcurse suplimentar pentru varianta ABT cu legături temporare (porţiunile de culoare albastră şi roşie), comparativ cu cele de la varianta ABT (doar porţiunile de culoare roşie).

Porţiunile de culoare albastră prezintă traseul parcurs pentru verificrea legăturilor temporare. În primul rând acestea sunt verificate doar în momentul în care s-a identificat o valoare consistentă pentru variabila agentului, după informarea agenţilor copii prin mesaje de tip *info.* În momentul transmiterii numărului maxim de mesaje se parcurge procesul de anulare a legăturii temporare (rutina CheckRemoveLink). Procesul de anulare a legăturii este parcurs de cei doi agenţi între care există legătura temporară. Agentul care transmite mesajul *RemoveL* actualizează mulţimea sa de copii, iar celălalt agent actualizează mulţimea de părinţi.

Trebuie subliniat faptul că procesul de verificare este pornit în rutina CheckAgentView şi prin urmare poate apărea fie la recepţionarea unui mesaj *info* fie la recepţionarea unui mesaj *back*.

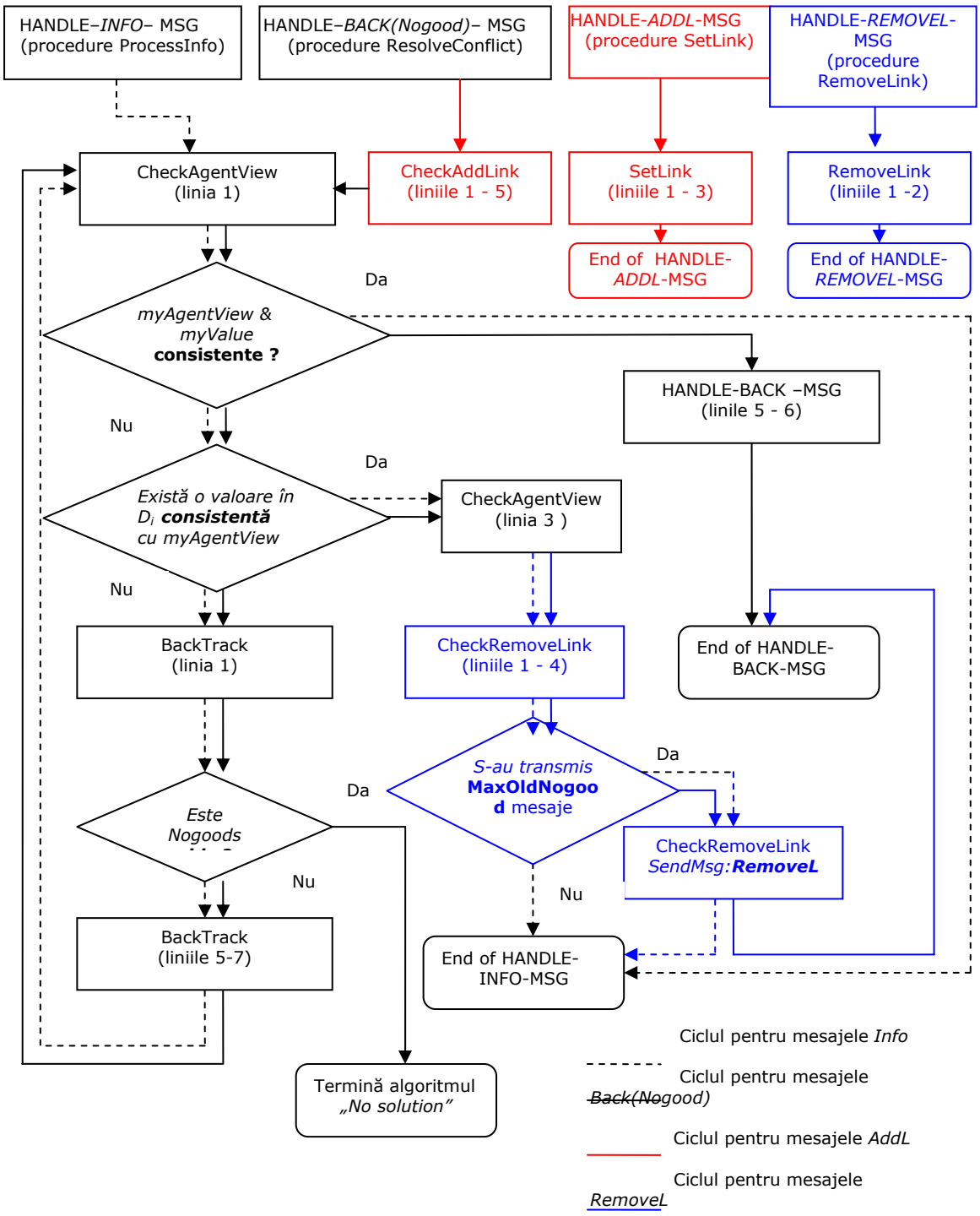


Figura 8.4. Fluxul mesajelor în cadrul algoritmului ABT cu legături temporare

8.3. REZULTATE EXPERIMENTALE.

În acest paragraf sunt prezentate rezultatele experimentale pentru acest studiu, obţinute în urma implementării şi evaluării tehnicilor asincrone prezentate. Implementarea şi evaluarea s-a realizat pe baza modelului cu sincronizare SEIS prezentat în capitolul 5.

Experimentele s-au efectuat în aceleaşi condiţii ca şi la studiile anterioare. Este vorba de aplicarea tehnicilor asincrone pentru problema colorării unui graf, în varianta distribuită şi utilizarea acelor două tipuri de probleme. Condiţiile au fost prezentate în capitolul 6, paragraful §6.1.

Pentru a putea face evaluarea celor cinci tehnici (varianta de bază Yokoo şi tehnicile obţinute), s-a contorizat fluxul de mesaje (adică cantitatea de mesaje info şi back schimbate de agenţi), numărul de constrângeri verificate şi numărul de constrângeri concurente verificate. Evaluările au fost făcute pentru cele cinci tehnici prezentate.

Analizând cozile cu mesaje pentru tehnicile asincrone, după ce acestea au fost implementate pe baza modelului cu sincronizare şi aplicate la problema colorării grafurilor, varianta distribuită, s-a observat şi la aceste tehnici o cantitate foarte mare de mesaje de tip info redundante. Trebuie precizat faptul că fiecare agent rulează concurent şi asincron, prin urmare în fiecare coadă de mesaje se adună mai multe mesaje de tip *info* sau *back*. În primul rând, s-au identificat mesaje de tip info de la acelaşi agent, trimise în diverse momente ale rulării. Aceste mesaje de tip info sunt redundante, iar unele învechite.

Plecând de la aceste observaţii, s-a apelat la algoritmul de filtrare prezentat în §6.5.2. Practic, când se extrage din coada de mesaje un mesaj de tip *info*, acesta nu este tratat imediat de rutina *ProcessInfo*, ci se verifică dacă acesta nu este cumva redundant sau învechit. Dacă apare una din aceste situaţii, se ignoră mesajul (eliminându-l din coada de mesaje), altfel se apelează rutina normală de tratare a mesajelor info.

Prin urmare, pentru fiecare versiune s-au executat 100 de rulări, dar în condiţiile aplicării algoritmului de filtrare a mesajelor de tip info. Cele două clase de tehnici evaluate au fost numite ABT (varianta de bază ABT derivată din nucleul ABT), ABTTL (ABT cu legături temporare). Valorile experimentale obţinute pentru trei clase de grafuri, grafuri de dimensiune mare (cu 30, 40 respectiv 50 noduri/agenţi) au fost stocate în tabelul 5.1.

Spre deosebire de studiul din capitolul anterior unde s-au utilizat grafuri cu dimensiune mai mică, evaluarea tehnicilor cu legături temporare a necesitat probleme de dimensiune mult mai mare pentru că tehnica de bază şi cea cu legături temporare necesitau costuri apropiate pentru probleme de dimensiune mică. Utilizarea modelului cu sincronizare la implementarea tehnicilor a permis o evaluare pentru probleme de dimensiune mare.

Pentru cazul versiunii cu număr temporar de legături s-au implementat şi evaluat două variante de algoritmi cu legături temporare, corespunzător celor două tipuri de soluţii propuse pentru determinarea numărului de mesaje:

- versiuni ce determină static numărul de mesaje (numite ABTTL#s1 şi ABTL#s2, corespunzator celor două soluţii statice propuse în paragraful §8.2.1);

- versiuni ce determină dinamic numărul de mesaje (numite ABTTL#d1 şi ABTTL#d2, corespunzator celor două soluţii dinamice propuse în paragraful §8.2.2).

Efortul local depus de fiecare agent a fost evaluat folosind cele două unităţi de măsură propuse în capitolul 5: numărul constrângerile verificate şi numărul de constrângeri concurente verificate (notat c-ccks). Analizând rezultatele din tabelul 8.1, se remarcă eforturi de calcul mici pentru probleme de dimensiune mică faţă de varianta de bază ABT. În schimb, odată cu creşterea dimeniunii problemelor (40 şi 50 noduri), doar variantele dinamice au performanţe bune, remarcându-se varianta ABTTL#d2, care a necesitat cel mai mic efort de calcul în raport cu varianta ABT.

Variantele statice au necesitat eforturi locale mai bune decât varianta de bază doar pentru probleme de dimensiune mică, indiferent de densitatea problemei. Pentru probleme de dimensiune mare, costurile au crescut dramatic pentru aceste versiuni, spre deosebire de cele dinamice.

		n = 30		n = 40		n = 50	
		m=n x2	m=n x2.7	M=n x2	m=n x2.7	m=n x2	m=n x2.7
ABT	TNogood	1621.99	2632.30	3950.03	7248.24	35231.29	33008.18
	TOk	5560.30	9348.11	14208.47	28235.65	177244.79	140240.47
	TCcks	476188	739461	1690134	2946308	21316191	17631276
	Tc-ccks	87704.	118282	299993.73	352171.15	2182343.71	1929763
ABTTL #s1	TNogood	877.16	2411.86	3648.17	5374.55	29475.00	28405.00
	TOk	2793.26	7503.00	10972.33	16153.90	104635.50	92471.00
	TCcks	260786	628925	1403769	1878704	14572460	13476611
	Tc-ccks	69076	110646	304527.83	333460.15	3041334.50	2560670
ABTTL #s2	TNogood	1191.12	3014.29	3017.25	6521.36	25931.00	25741.50
	TOk	3592.03	8973.00	10771.38	21284.82	99043.00	78725.00
	TCcks	339751	778830	1184052	2432875	13386381	11243364
	Tc-ccks	96029.22	144736.43	197051.88	423334.82	2373663.50	2216387.50
ABTTL #d1	TNogood	1063.78	2825.27	2779.32	8218.54	20665.00	34548.20
	TOk	3948.45	8977.92	9797.11	28642.23	83434.50	121561.80
	TCcks	375523.61	754445.5	1124637.9	3121796.6	10636911.0	16671921.0
	Tc-ccks	99501.02	155294.85	200040.21	545574.08	1870391.67	1999188.00
ABTTL #d2	TNogood	1054.65	2581.87	2893.17	7262.12	10930.21	18948.86
	TOk	3743.49	8185.80	10217.28	23654.44	44167.43	64503.64
	TCcks	356951.37	691387.06	1167595.10	2665277.12	5701223.86	9050695.86
	Tc-ccks	93656.01	136670.46	220346.34	472663.35	1007322.86	1596578.36

Tabelul 8.1. Rezultatele experimentale pentru versiunile ABT în cazul limitării fluxului de mesaje pentru legăturile temporare

În ceea ce priveşte fluxul de mesaje, comportamentul remarcat anterior s-a păstrat, variantele dinamice necesitând un flux de mesaje mult mai mic decât

varianta de bază ABT cu număr fix de legături. Dar, în cazul problemelor cu dimensiune şi densitate mare, varianta dinamică #D2 a necesitat cel mai mic flux de mesaje.

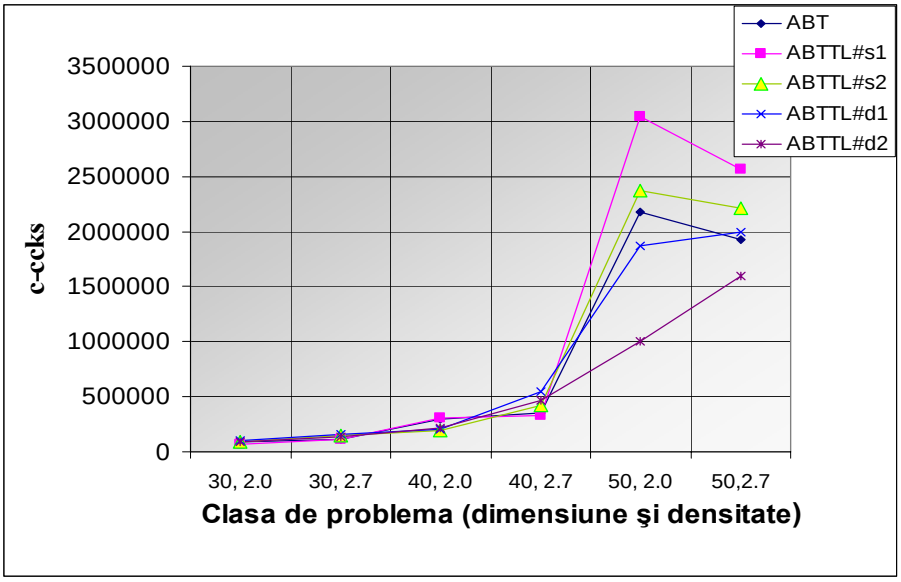


Figura 8.5. Rezultate comparative pentru versiunile ABT în raport cu numărul de constrângeri concurente verificate

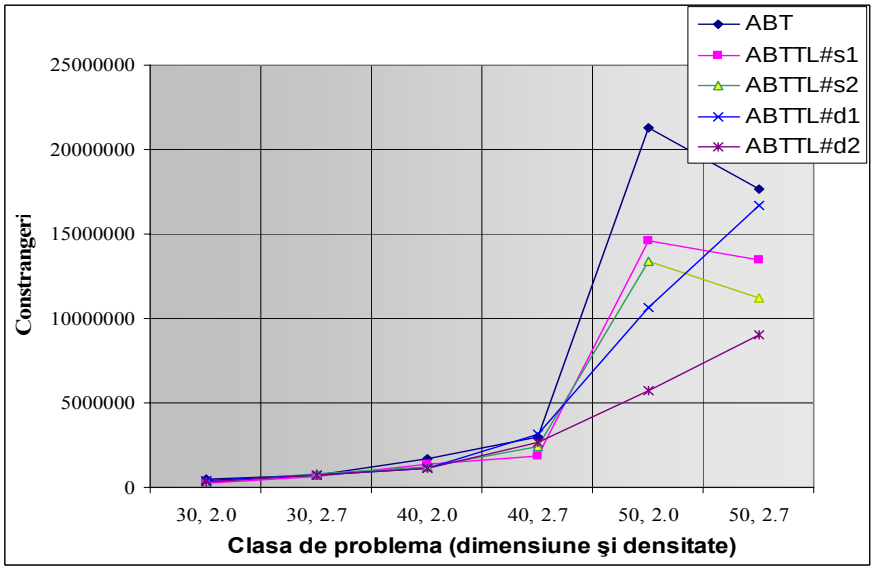


Figura 8.6. Rezultate comparative pentru versiunile ABT în raport cu numărul de constrângeri verificate

Soluţiile dinamice s-au remarcat ca fiind cele mai eficiente. Toate cele trei soluţii dinamice au presupus că agenţii cunosc în orice moment maximele relative la

fluxul de mesaje învechite obţinute de ceilalţi agenţi. În paragraful anterior s-au propus două soluţii relative la cunoaşterea acestor maxime. Mediul NetLogo permite agenţilor să aibă acces la zone de memorie comună, prin urmare soluţiile dinamice au putut fi implementate foarte uşor în acest simulator fără a folosi algoritmul din figura 8.2. Din experimentele efectuate a rezultat că nu sunt foarte mari diferenţe faţă de varianta dinamică propusă în acest studiu (variantă care nu necesită existenţa acelor zone comune de memorie şi care presupune folosirea algoritmului 8.3). Trebuie subliniat faptul că sunt de preferat variantele dinamice propuse împreună cu algoritmul din fig. 8.2. pentru că în practică, existenţa acelor zone comune de memorie nu este posibilă întotdeauna.

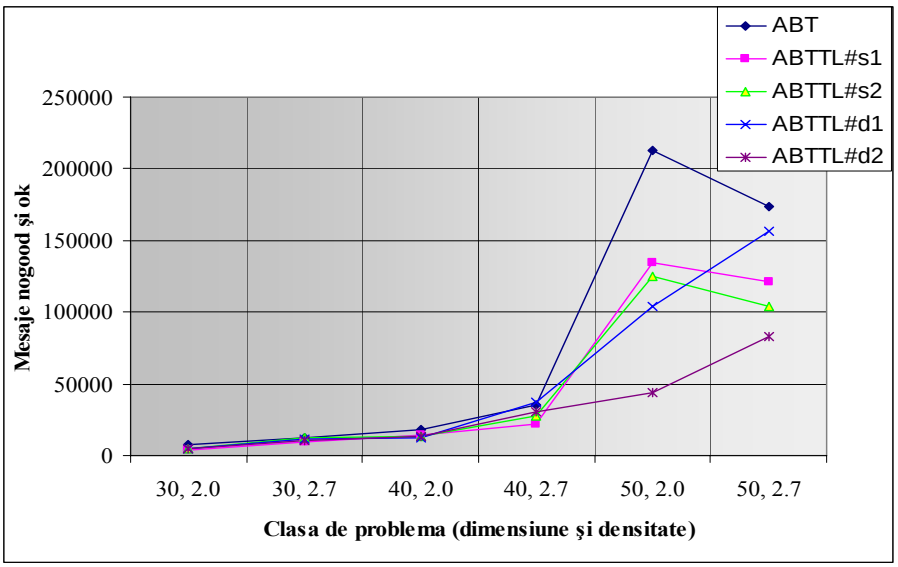


Figura 8.7. Rezultate comparative pentru versiunile ABT în raport cu numărul de mesaje schimbate

8.4. Concluzii.

În acest studiu a fost investigat comportamentul tehnicii ABT cu legături temporare pentru diferite valori ale numărului de mesaje transmise pentru legăturile temporare. Experimentele realizate arată că fiecare clasa de probleme are o valoare fixă a numărului de mesaje, valoare pentru care tehnica ABT kernel cu legături temporare se comportă identic cu tehnica de bază ABT. Din analiza experimentală a valorilor se observă că există o valoare apropiată de numărul maxim de mesaje, pentru care se obţin performanţe mai bune decât varianta de bază ABT. Au fost propuse mai multe soluţiii pentru limitarea numărului de mesaje transmise.

O primă soluţie propusă este aceea de a determina pentru fiecare agent, înainte de rulare, un număr fix de mesaje ce trebuie transmise pentru o legătură. Acest număr este determinat în mai multe moduri, fiind propuse două variante statice #S1 şi #S2. Acest număr va fi folosit ca şi număr maxim de mesaje transmise. Experimentele arată că pentru probleme cu dimensiune mică soluţia este eficientă, dar pentru dimensiune mare varianta de bază depăşeşte varianta cu număr temporar de legături ce utilizează cele două soluţii #S1 şi #S2.

O a două categorie de soluţii propuse, mult mai eficientă, constă în a determina dinamic în timpul rulării numărul maxim de mesaje. Practic, se deduce dinamic în timpul rulării, valoarea maximă a numărului de mesaje şi ea este ajustată în funcţie de evoluţia algoritmului, obţinându-se soluţiile #D1 şi #D2. Pentru aceste soluţii, un algoritm de determinare a numărului maxim de mesaje este propus, fără a necesita zone comune de memorie. Acest algoritm se poate aplica pentru sistemele de implementare SEIS şi SEIAS prezentate în capitolul 5.

Evaluările experimentale realizate în condiţiile unor probleme variate, cu valori iniţiale alese aleator, pentru probleme cu densităţi variate, arată că soluţia dinamică #D2 este mai eficientă decât tehnica de bază, pentru ambele tipuri de probleme (cu densitate mică şi dificilă), pentru dimensiuni mici şi pentru dimensiuni mari ale problemelor.

Rezultatele evidenţiate în acest capitol au fost publicate în [MusPP05a].

9. ASYNCHRONOUS BACKTRACKING CU LEGĂTURI TEMPORARE ŞI PERMANENTE

În acest studiu se propune un membru hibrid pentru familia ABT, membru derivat din nucleul ABT prin eliminarea informaţiilor învechite dintre agenţi, combinând două metode de eliminare a informaţiilor învechite dintre agenţi: adăugarea de legături permanente între agenţi, respectiv adăugarea de legături temporare. Noul membru se obţine prin identificarea şi transformarea anumitor legături temporare în legături permanente. Pentru aceasta se propune o metodă de identificare a legăturilor temporare bazată pe informaţiile relative la fluxul de mesaje învechite recepţionate de fiecare agent. Capitolul de faţă continuă studiul legăturilor temporare cu scopul de a găsi soluţii de îmbunătăţire a performanţelor tehnicilor de căutare asincrone, studiul având ca punct de plecare soluţiile dinamice propuse în capitolul 8.

Capitolul începe cu o prezentare a cadrului de lucru, urmată de prezentarea modului în care se obţine noua tehnică derivată, prin transformarea anumitor legături temporare în legături permanente. În paragraful al treilea sunt făcute mai multe evaluări experimentale, prin implementarea acestei tehnici folosind modelul cu sincronizare. În final sunt prezentate concluziile relative la noul membru hibrid din familia ABT.

9.1. Eliminarea informaţiilor învechite în cazul familiei ABT.

Tehnica ABT kernel este la bază o tehnică derivată din tehnica ABT introdusă în [YDIK92], dar se remarcă prin faptul că nu necesită adăugarea de noi legături între agenţii neconectaţi iniţial (spre deosebire de ABT care necesită, în timpul rulării, de adăugarea de noi legături). În capitolul 3 au fost prezentate mai multe soluţii de eliminare a informaţilor învechite dintre agenţi, soluţii introduse în [BM05]. Aceste soluţii vor fi sintetizate pentru a putea explica locul noii tehnici derivate propuse aici.

O primă soluţie de eliminare a informaţiilor învechite este de a adăuga noi legături pentru a permite agentului ce a recepţionat un mesaj nogood să decidă dacă este sau nu învechit. Această soluţie conduce la algoritmul clasic ABT al lui Yokoo, respectând principiile căutării asincrone definite de Yokoo în [YDIK98].

O a doua soluţie constă în detectarea situaţiei în care un anumit nogood este sau nu învechit. Practic, un nogood ipotetic învechit şi toate valorile recepţionate de la acei agenţi cu care nu a fost conectat sunt uitate. Aceste două variante permit obţinerea a patru tehnici derivate din nucleul ABT kernel:

- *Adăugarea de noi legături înainte de căutare, în faza de preprocesare*. Varianta aceasta este numită ABT_{all}. Algoritmul adaugă noi legături (care vor deveni permanente) în faza de preprocesare. Această tehnică este apropiată de o variantă corectă DIBT.
- *Adăugarea de legături în timpul căutării-* se obţine varianta ABT de bază (Yokoo). Legăturile se adaugă în timpul căutării în momentul recepţionării

unui mesaj nogood (Back) care conţine informaţii relative la un agent neconectat, conform ordinii dintre agenţi. Noile legături sunt permanente. .

- *Adăugarea de legături temporare în timpul căutării*- se obţine varianta ABT_{temp} Legăturile se adaugă ca şi la varianta ABT Yokoo, în timpul căutării în momentul recepţionării unui mesaj nogood (Back) care conţine informaţii relative la un agent neconectat. Diferenţa este că aceste legături sunt temporare, ele rămânând până când un număr de mesaje între acei agenţi sunt schimbate (după care ele sunt anulate).
- *Eliminarea legăturilor*:-se obţine varianta ABT_{not}, variantă cunoscută sub numele de Distributed Dynamic Backtracking (DIBT). Această variantă nu necesită legături suplimentare. Practic, informaţiile învechite sunt eliminate în timp finit. Pentru aceasta agenţii care execută backtrack uită toate nogoodurile ce pot conţine ipotetic valori învechite.

În capitolul anterior au fost investigate diverse valori pentru numărul de mesaje, valori ce se determină fie static, fie dinamic în timpul rulării. În capitolul 8 şi în [MusPP05a] este propusă o soluţie dinamică de determinare a numărului de mesaje necesare pentru păstrarea unei legături, experimentele arătând o eficienţă mai bună (în raport cu efortul de calcul făcut) decât pentru varianta de bază ABT. Soluţia dinamică se bazează pe determinarea fluxului maxim de mesaje nogood şi folosirea acestor informaţii în determinarea numărului de mesaje.

Plecând de la soluţia dinamică propusă în capitolul 8 pentru determinarea numărului de mesaje necesare păstrării unei legături tempoare, în acest capitol se propune o nouă metodă hibridă de eliminare a informaţiilor învechite dintre agenţi. Această soluţie constă în a transforma anumite legături temporare în legături permanente, pe baza informaţiilor legate de fluxul de mesaje învechite. Aplicând această metodă la nucleul *ABTkernel*, se poate obţine o nouă tehnică hibridă, tehnică ce foloseşte ce este mai bun de la cele două tehnici derivate: ABT şi ABT temporary link.

- *Adăugarea de legături permanente şi temporare*: ABT cu legături permanente şi temporare. Acest nou algoritm adaugă noi legături în timpul căutării. O parte din aceste legături sunt temporare, ele fiind păstrate până ce un număr de mesaje este schimbat (număr determinat dinamic în timpul rulării). În schimb, anumite legături temporare sunt transformate în legături fixe, pe baza unor informaţii relative la fluxul maxim de valori nogood învechite.

Experimentele arată o mai bună eficienţă (în raport cu efortul de calcul) în comparaţie cu varianta asynchronous backtracking derivată din nucleul ABT.

9.2 Asynchronous Backtracking cu legături temporare şi fixe.

Analiza valorilor experimentale prezentate în capitolul 8 arată că cea de-a doua soluţie dinamică de determinare a numărului de mesaje este cea mai eficientă. Această soluţie va fi punctul de plecare în construirea unei noi tehnici derivate hibride. Prin urmare, se vor folosi informaţiile legate de fluxul de mesaje nogood învechite. Astfel, asemănător metodei dinamice de determinare a numărului de mesaje, se vor determina perioadele de rulare pentru care anumite legături temporare sunt păstrate. Spre deosebire de soluţia prezentată în capitolul anterior, anumite legături vor fi transformate în legături fixe.Aceasta va fi ideea de bază ce va sta la construirea unei noi tehnici derivate.

Pentru a creşte eficienţa celor două variante dinamice propuse în capitolul anterior, fiecare agent încearcă să păstreze cât mai mult legăturile temporare. Plecând de la aceste observaţii, soluţia propusă în capitolul de faţă constă în transformarea anumitor legături temporare în legături fixe. De fapt, legăturile temporare cu acei agenţi cu care a schimbat un flux de mesaje nogood învechite maxim sunt transformate în legături fixe. Pentru fiecare agent se determină agentul cu care a avut loc un flux de mesaje învechite maxim (dintre cei cu care au legături temporare). Legătura temporară ce există cu acesta este transformată într-o legătură permanentă (va rămâne până la finalul găsirii soluţiei).

Această soluţie presupune cunoaşterea de către fiecare agent, a numărului maxim de mesaje învechite recepţionate de fiecare agent. O soluţie, propusă şi folosită în acest capitol, este bazată pe transmiterea maximelor fiecărui agent la cei cu care este conectat, în momentul transmiterii unui mesaj de tip info sau nogood. Ideea este asemănătoare celei din algoritmul de determinare a costului cumulativ din [MKRZ02].

Determinarea numărului maxim de mesaje învechite recepţionate de agenţi

1: Fiecare agent iniţializează variabilele contor *C_OldNogood* cu 0. De asemenea, se iniţializează *MaxOldNogood* cu 0.
2: Când un agent transmite un mesaj el include şi valoarea contorului său *MaxOldNogood*.
3: Când un agent recepţionează o valoare nogood învechită de la un agent *Sender* se actualizează contorul corespunzător din lista *C_OldNogood*.
Replace-item Sender *C_OldNogood* with item Sender *C_OldNogood* + 1
4 if un agent recepţionează un mesaj cu contorul *SenderMx* de la agentul *Sender*
if item *Sender C_OldNogood* < *SenderMx* then
Replace-item Sender *Sender C_OldNogood* with *SenderMx*
end if
Set MaxOldNogood = max { *C_OldNogood* }
end if

Fig. 9.1. Determinarea numărului maxim de mesaje învechite recepţionate de fiecare agent

În figura 9.1. se prezintă un algoritm original de determinare a maximului dintre maximele fluxului de mesaje învechite. Fiecare agent păstrează o listă locală de variabile contor (C_OldNogood) având rol de a contoriza numărul de mesaje învechite recepţionate. De asemenea, fiecare agent păstrează şi un contor *MaxOldNogood* ce va reţine maximul numărului de mesaje dintre vecini. Acest algoritm este asemănător cu cel folosit în capitolul anterior, dar există anumite diferenţe la pasul 4. În urma experimentelor s-a observat că această variantă permite obţinerea unor rezultate mai bune, în acest context al legăturilor temporare transformate în fixe.

Acest algoritm se aplică asupra tehnicii ABT (derivată din nucleul ABT), permiţând fiecărui agent să reţină valoarea maximă a numărului de mesaje nogood învechite schimbate de agenţi.

În figura 9.2 se prezintă acele modificări necesare în codul tehnicii ABT Yokoo (variantă derivată din nucleul ABTkernel), pe baza metodei de determinare a legăturilor temporare şi a legăturilor temporare transformate în legături permanente, astfel încât să se obţină o nouă tehnică hibridă, tehnică ce foloseşte ce este mai bun de la cele două tehnici derivate: ABT şi ABT temporary link. Modificările sunt marcate cu ***.

```
 procedure ABTkernel()
1  myValue←empty; end ← false;
1.1.  Set Γe+ (self) ← ∅                          ***
1.2.  Set Γe- (self) ← ∅                           ***
2     CheckAgentView();
3     while (not end) do
4       msg ← getMsg();
5       switch(msg:type)
6             Info : ProcessInfo(msg);
7             Back : ResolveConflict(msg);
8             Stop : end← true;
9.1.        AddL : SetLink(msg);
9.2.        RemoveL: RemoveLink(msg);                ***
End

procedure ProcessInfo(msg)
1     Update(myAgentView; msg.Assig);
2     CheckAgentView();
End

procedure CheckAgentView(msg)
1     if not consistent(myValue;myAgentView) then
2           myValue ← ChooseValue();
3           if (myValue) then
                  for each child ∈ Γ+ (self) do  sendMsg:Info(child;myValue);
3.1               CheckRemoveLink(Self, Child)          ***
4           else Backtrack();
end

procedure ResolveConflict(msg)
1     if Coherent(msg.Nogood; Γ- (self) ∪ {self}) then
2.1         CheckAddLink(msg);
3           add(msg:Nogood;myNogoodStore); myValue ← empty;
4           CheckAgentView();
5.1   else if Coherent(msg.Nogood; self) then
              SendMsg:Info(msg.sender; myValue);
5.2.   replace item Sender C_OldNogood with item Sender C_OldNogood+1
***
end

procedure SetLink(msg)
1     add(msg.sender; Γ+ (self)); add(msg.sender; Γe+ (self));
2     sendMsg:Info(msg.sender;myV alue);
End

procedure CheckAddLink(msg)
1     for each (var ∈ lhs(msg.Nogood))
2         if (var ∉ Γ- (self)) then
3             sendMsg:AddL(var, self);
```

```
4              add(var; Γ⁻ (self)); add(var; Γe⁻ (self));
5              Update(myAgentView; var ← varValue);
End

procedure RemoveLink(msg)                    ***
1    remove(msg.sender; Γ⁻ (self));
2    remove(msg.sender; Γe⁻ (self));
End
procedure CheckRemoveLink(msg)               ***
1    for each child ∈ Γe⁺ (self) do
2         if (item child COldNogood)= MaxOldNogood then
               replace item child FlagList with 1
3        if (item child CMessageTempLink)≥MaxOldNogood and
                                          (item child FlagList=0) then
              remove(child; Γ⁺ (self)); remove(child; Γe⁺ (self));
4             sendMsg:RemoveL(child, self);
5             Update(myAgentView; child ← unknown);
End
procedure Backtrack()
1     newNogood ← solve(myNogoodStore);
2     if (newNogood = empty) then
3             end ← true; sendMsg:Stop(system);
4     else
5             sendMsg:Back(newNogood);
6             Update(myAgentView;rhs(newNogood)← unknown);
7             CheckAgentView();
End

function ChooseValue()
1     for each v ∈ D(self) not eliminated by myNogoodStore do
2             if consistent(v; myAgentView) then return (v);
3             else add(xj = valj) self ≠ v;myNogoodStore);
4     return (empty);
End

procedure Update(myAgentView; newAssig)
1     add(newAssig;myAgentV iew);
2     for each ng ∈ myNogoodStore do
3             if  not Coherent(lhs(ng);myAgentView) then
remove(ng;myNogoodStore);
End

function Coherent(nogood; agents)
1     for each var ∈ nogood ∪ agents do
2             if   nogood[var] ≠ myAgentV iew[var] then return false;
3 return true;
end
```

Figura 9.2. Algoritmul ABT cu legături temporare şi permanente.

Obţinerea acestei variante derivate din ABTkernel a presupus mai multe schimbări în algoritmul de bază ABT kernel, derivat în asynchronous backtracking.

Se vor prezenta acele modificări realizate în codul ABT, care conduc la varianta ABT cu legături temporare şi permanente.

Prima modificare a constat în introducerea pentru fiecare agent a două mulţimi suplimentare Γ_e^+ (self) şi Γ_e^- (self), pentru identificarea agenţilor copil şi părinte ce apar în urma legăturilor temporare. În procedura *ABTkernel()*, în liniile 1.1. şi 1.2. ei sunt determinaţi. De asemenea, este nevoie de introducerea a două structuri de date *CMessageTempLink* şi *C_OldNogood***.** Prima structură va fi folosită de un agent A_i pentru a reţine numărul de mesaje info transmise pentru fiecare legătură temporară. A doua structură se foloseşte la contorizarea fluxului de mesaje învechite.recepţionate.

Evidenţa de fiecare agent a legăturilor temporare ce pot fi transformate în legături permanente, este realizată folosind o listă de flaguri (în algoritmul din figura 9.2. este numită FlagList), indexată după numărul de ordine al agenţilor copil. Această listă memorează valoarea 1 pentru o legătură temporară devenită permanentă şi 0 în caz contrar. Existenţa valorii 1 în lista de flaguri a unui agent, pe o poziţie k, va permite păstrarea legăturii spre agentul $A_{k.}$

Noul algoritm necesită introducerea unui al patrulea mesaj *RemoveL*, care anunţă un agent copil de anularea legăturii temporare dintre doi agenţi. Practic, agentul copil va anula agentul *Sender* din lista părinţilor săi. Modificările necesare apar în linia 9.2. din procedura *ABTkernel()* şi procedura RemoveLink().

O altă modificare faţă de algoritmul ABT este aceea că după selectarea unei noi valori şi anunţarea copiilor despre noua selecţie, este necesară verificarea legăturilor temporare pentru a determina câte dintre ele mai rămân actuale. Acest lucru este făcut în procedura *CheckAgentView(msg)* linia 3.1, prin apelarea unei noi proceduri cu numele *CheckRemoveLink(Self, Child).* Această rutină verifică pentru agenţii copil din Γ_e^+ (self) dacă s-a atins numărul maxim de mesaje transmise pentru acea legătură. Acest număr maxim se poate determina cu metoda propusă în figura 9.1. În caz că s-a atins valoarea maximă *MaxOldNogood* se anulează acea legătură şi se actualizează contextul de lucru. În schimb, dacă agentul *child* este cel care a atins maximul de mesaje nogood recepţionate, este etichetat cu 1, astfel încât legătura sa cu agentul curent devine permanentă şi nu va mai fi anulată.

În figura 9.3 este prezentat fluxul mesajelor, în care sunt evidenţiate noile mesaje (mesaje de tip *AddL* şi *RemoveL* ce permit gestiunea legăturilor noi). Sunt evidenţiate porţiunile de cod ce sunt parcurse suplimentar pentru varianta ABT cu legături temporare şi permanente (porţiunile de culoare albastră şi roşie), comparativ cu cele de la varianta ABT Yokoo (doar porţiunile de culoare roşie).

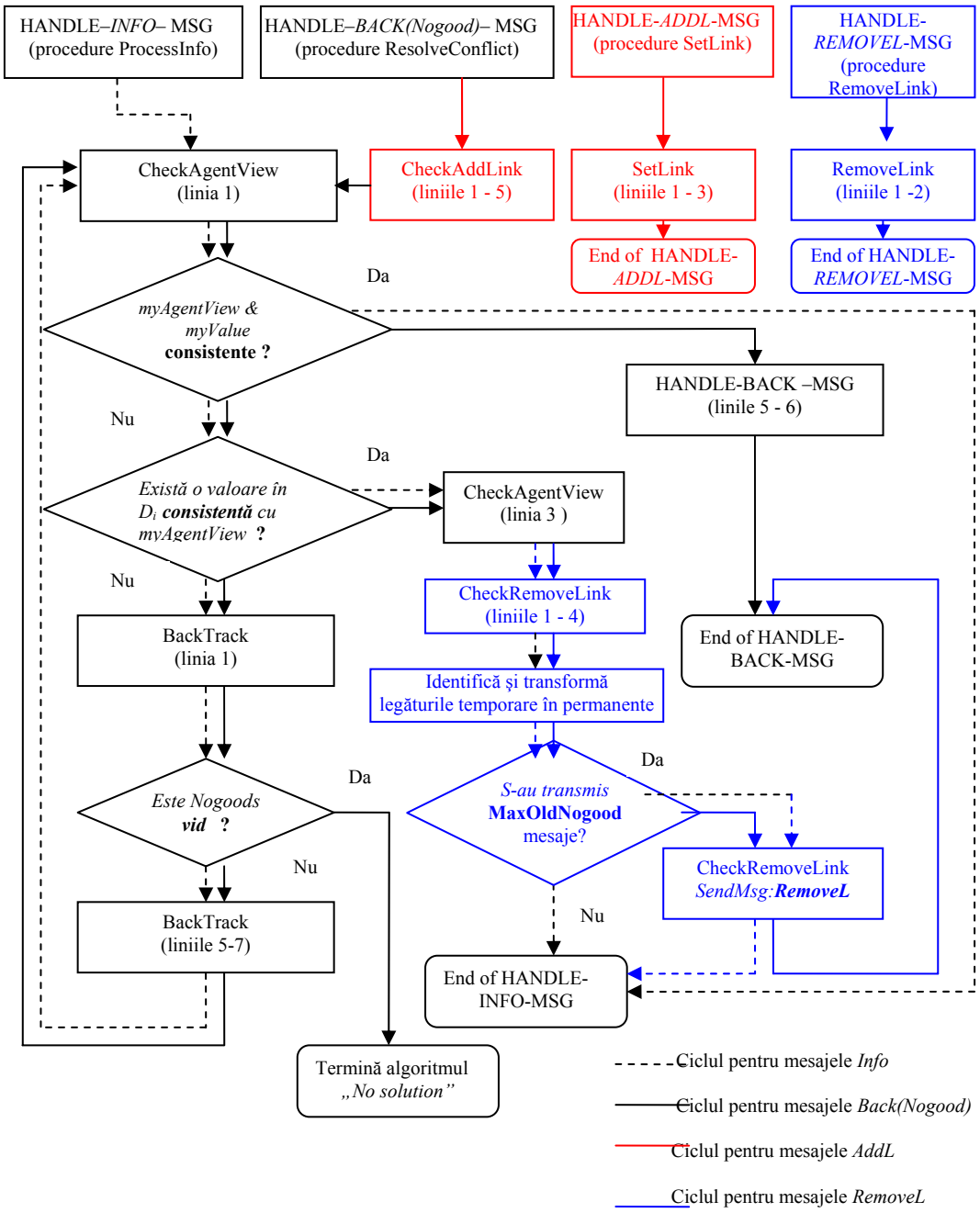


Figura 9.3. Fluxul mesajelor în cadrul algoritmului ABT cu legături fixe şi temporare

9.3. REZULTATE EXPERIMENTALE.

În acest paragraf sunt prezentate rezultatele experimentale, obţinute în urma implementării şi evaluării tehnicilor asincrone analizate pe baza modelului cu sincronizare.

Un ecran de captură exemplificator este prezentat în figura 9.4.

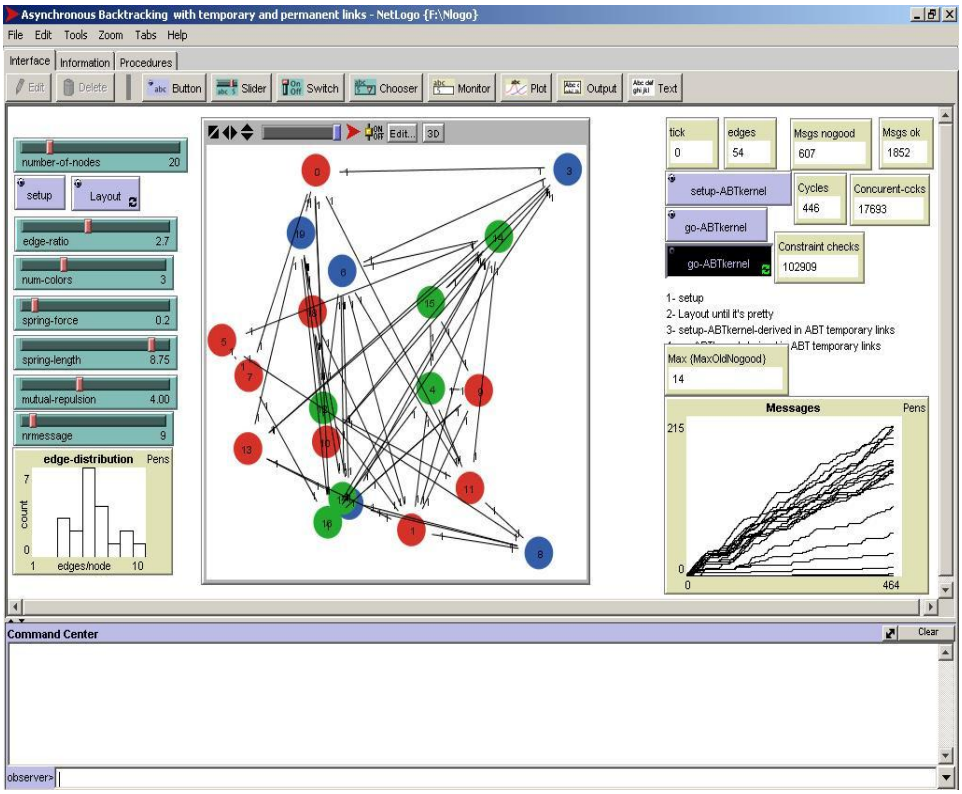


Figura 9.4. Implementarea NetLogo pentru algoritmului ABT cu legături fixe şi temporare

Experimentele s-au efectuat în aceleaşi condiţii ca şi în capitolul anterior. Tehnicile asincrone au fost aplicate pentru problema colorării unui graf, în varianta distribuită, folosind cele două clase de densităţi. Pentru a putea face evaluarea celor două variante (varianta de bază Yokoo şi tehnica hibridă obţinută), s-a contorizat fluxul de mesaje (adică cantitatea de mesaje info şi back schimbate de agenţi), numărul de constrângeri verificate şi numărul de constrângeri concurente verificate.

Pentru implementare şi evaluare s-a utilizat modelul cu sincronizarea execuţiei agenţilor prezentat în capitolul 5. Ţinând cont de analiza prezentată în capitolul 6 relativă la fluxul de mesaje redundante, s-a aplicat şi aici algoritmul de filtrare prezentat în §6.5.2.

Prin urmare, pentru fiecare versiune s-au executat 100 de rulări, dar în condiţiile aplicării algoritmului de filtrare a mesajelor de tip info. Cele două clase de tehnici evaluate au fost numite ABT (varianta de bază ABT-derivată din ABT kernel),

ABTTPL (ABT cu legături temporare şi permanente). Valorile experimentale obţinute pentru trei clase de grafuri, grafuri de dimensiune mare (cu 30, 40 respectiv 50 noduri/agenţi) sunt stocate în tabelul 9.1. Pentru cazul versiunii cu legături temporare şi permanente s-au evaluat două variante de algoritmi, corespunzând la două modalităţi de determinare a numărului de mesaje învechite recepţionate de fiecare agent:

- o versiune în care numărul maxim de mesaje învechite se determină cu metoda prezentată în figura 9.1, versiune notată cu ABTTFL#1. Practic se obţine maximul fluxului de mesaje învechite pentru toţi agenţii.
- o versiune în care numărul maxim de mesaje învechite se determină doar dintre vecinii fiecărui agent, fără a fi nevoie să se aplice metoda din figura 9.1. Practic, fiecare agent îşi determină *MaxOldNogood* din lista sa *C_OldNogood* (acesta este un maxim local). Aceasta versiune a fost notată cu ABTTPL#2.

		n = 30		n = 40		n = 50	
		m=n x2	m=n x2.7	m=n x2	m=n x2.7	m=n x2	m=n x2.7
ABT	TNogood	1427.20	1691.20	2920.43	8820.10	11008.18	21130.77
	TOk	5407.34	6142.25	11323.40	32946.23	44184.56	71094.45
	TCcks	443073.7	484444.9	1276261.8	3496629.7	8003935.9	11706566.6
	Tc-ccks	77567.37	76658.00	187313.21	407892.71	839876.36	1562254.00
ABTTPL #s1	TNogood	940.22	1741.61	1873.33	6541.21	8287.04	19858.26
	TOk	3053.32	5669.61	5922.09	20512.21	26908.60	69642.81
	TCcks	264994.6	466694.0	671172.3	2332383.6	3899996.2	9641356.5
	Tc-ccks	62819.55	87131.03	127092.26	415689.09	777720.36	1735377.4
ABTTPL #s2	TNogood	981.12	1614.23	2163.13	6623.51	11745.71	23185.81
	TOk	3338.19	5551.78	6819.81	21231.85	46174.70	83198.28
	TCcks	223118.5	434123.8	731456.7	2715132.6	4328765.7	1267431.3
	Tc-ccks	58791.95	81332.32	138292.2	479856.25	866721.3	2100675.2

Tabelul 9.1. Rezultatele experimentale pentru versiunile ABT în cazul variantelor cu legături temporare şi permanente.

În ceea ce priveşte evaluarea efortului local depus de fiecare agent (concurent sau neconcurent), analizând rezultatele din tabelul 9.1, se remarcă eforturi de calcul mai mici, pentru dimensiune mică, faţă de varianta de bază Yokoo. Mai mult, versiunea a doua ABTTPL#2 a necesitat un efort mai mic în obţinerea soluţiei decât prima variantă. Se remarcă faptul că odată cu creşterea dimensiunii problemelor (40 şi 50 noduri), varianta ABTTPL#1 este mai eficientă. În ceea ce priveşte numărul de constrângeri concurente verificate, din păcate variantele propuse necesită valori mai apropiate de varianta de bază ABT.

În ce priveşte fluxul de mesaje s-a păstrat acelaşi comportament remarcat anterior (relativ la efortul de calcul), variantele hibride propuse necesitând un flux de mesaje mai mic decât varianta de bază ABT cu număr fix de legături. Dar, în

cazul problemelor cu dimensiune şi densitate mare, varianta ABTTPL#1 a necesitat un flux de mesaje mai mic decât varianta ABTTPL#2.

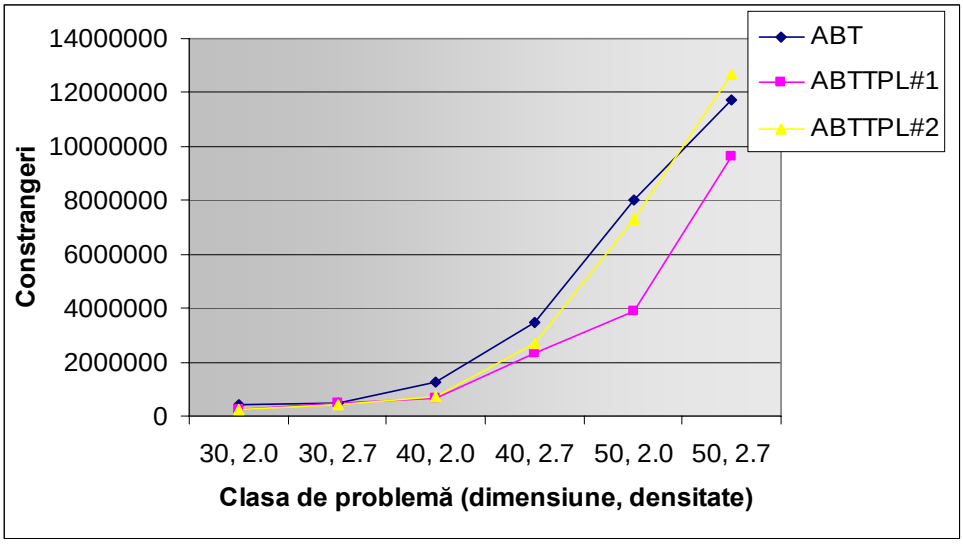


Figura 9.5. Rezultate comparative pentru versiunile ABT în raport cu numărul de constrângeri verificate

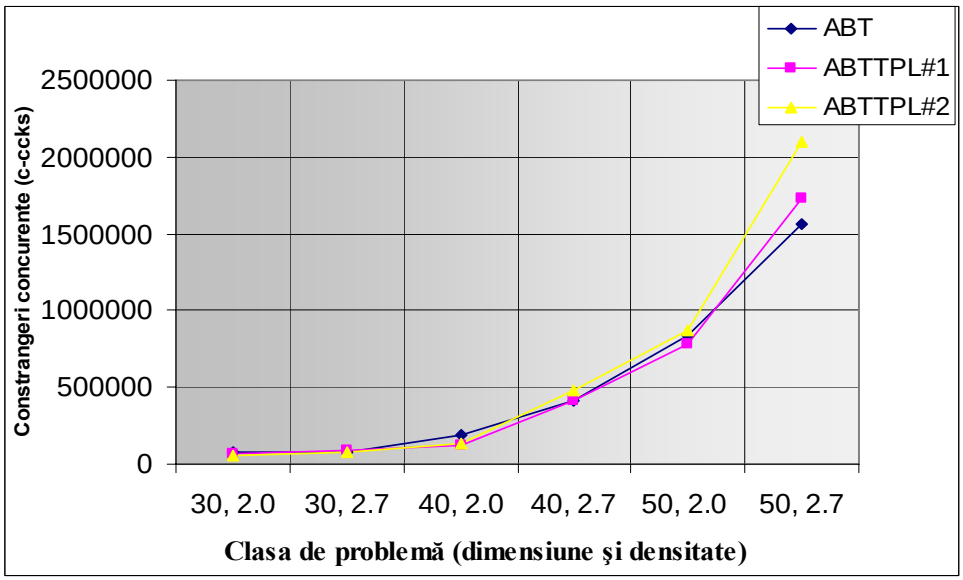


Figura 9.6. Rezultate comparative pentru versiunile ABT în raport cu numărul de constrângeri concurente verificate

Analiza valorilor experimentale şi a reprezentărilor grafice din figurile 9.5, 9.6 şi 9.7. arată că dintre cele două variante propuse, prima varianta este cea mai eficientă, comparativ cu a doua, care pentru probleme de dimensiune mare

necesită costuri mult mai mari. Prin urmare, este recomandată determinarea valoriii MaxOldNogood (folosită ca valoare maximă pentru numărul de mesaje transmise pentru legăturile temporare) prin aplicarea algoritmului prezentat în fig. 9.1.

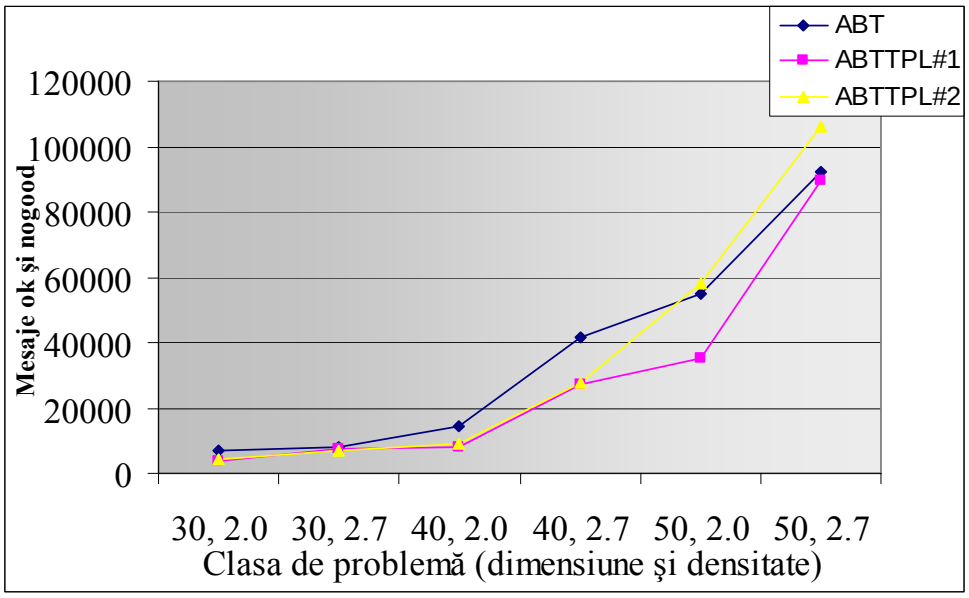


Figura 9.7 Rezultate comparative pentru versiunile ABT în raport cu numărul de mesaje schimbate

O altă observaţie este relativă la cele două clase de probleme analizate: probleme cu densitate mare şi cele cu densitate mică. Valorile obţinute sunt mai apropiate de varianta de bază în cazul problemelor dificile (cele cu densitate mare 2.7). În schimb, pentru probleme cu densitate rară se remarcă diferenţe mai mari între valorile măsurate pentru cele două variante hibride comparativ cu cea de bază.

9.4. Concluzii.

În acest capitol se propune un nou membru hibrid pentru familia ABT, membru derivat din nucleul ABT prin eliminarea informaţiilor învechite dintre agenţi, combinând două metode mai vechi de eliminare a informaţiilor învechite dintre agenţi: adăugarea de legături permanente între agenţi, respectiv adăugarea de legături temporare. Noul membru presupune transformarea anumitor legături temporare în legături permanente, pe baza informaţiilor relative la fluxul de mesaje învechite recepţionate de fiecare agent.

Studiul din acest capitol este o continuare a studiului legăturilor temporare prezentat în capitolul 8.

Pentru obţinerea acestei tehnici hibride se propune o metodă de determinare a numărului de mesaje ce se transmit pentru legăturile temporare şi o soluţie de determinare a legăturilor temporare ce sunt transformate în legături permanente. Evaluările experimentale realizate în condiţiile unor probleme variate, cu valori iniţiale alese aleator, pentru probleme cu densităţi variate, arată că soluţia

hibridă este mai eficientă decât tehnica de bază, pentru ambele tipuri de probleme (cu densitate mică şi dificilă), pentru dimensiuni mici şi pentru dimensiuni mari ale problemelor.

Rezultatele evidenţiate în acest capitol au fost publicate în [MusPP06].

10. ANALIZA EXPERIMENTALĂ A IMPACTULUI TEHNICII „NOGOOD PROCESSOR" ASUPRA EFICIENŢEI TEHNICILOR DE CĂUTARE ASINCRONE DIN FAMILIA AWCS.

În acest capitol se propune o soluţie de adaptare a tehnicii „nogood processor" introdusă de Armstrong, pentru tehnica AWCS [AD97]. Această tehnică constă în stocarea valorilor nogood şi folosirea ulterioară a informaţiilor din nogood-uri în procesul de selectare a unei noi valori pentru variabilele asociate agenţilor. Sunt analizate situaţiile în care se foloseşte un „nogood processor" centralizat şi situaţiile în care se distribuie nogood-urile la mai multe „nogood processor" ce sunt mânuite de anumiţi agenţi. Se propune o soluţie de distribuire a procesoarelor de nogood-uri către agenţi, în funcţie de ordinea agenţilor, cu scopul de a reduce costurile de căutare şi de stocare.

Capitolul începe cu o prezentare a contextului de lucru. În paragraful al doilea se prezintă o soluţie de implementare a tehnicii nogood processor în cadrul tehnicii AWCS. În acest paragraf sunt propuse mai multe soluţii de distribuire a nogoodurilor către procesoare: versiuni cu un singur nogood processor centralizat şi versiuni cu mai multe procesoare distribuite. Paragraful al treilea prezintă o analiză experimentală a versiunilor obţinute, comparându-se performanţele relativ la tehnica de bază. În final sunt prezentate concluziile aplicării tehnicii nogood processor în cazul tehnicii AWCS.

10.1 Tehnica "nogood processor".

În cadrul modelării DCSP, algoritmul asynchronous weak-commitment search (AWCS) joacă un rol fundamental şi de pionierat între algoritmii de căutare asincronă. Algoritmul se remarcă atât prin faptul că suferă de o explozie a valorilor nogood, cât şi prin schimbarea dinamică a ordinii agenţilor, fiind publicat de Yokoo în [YDIK98]. Apariţia valorilor nogood are ca efect introducerea unor noi constrângeri. Valorile nogood indică cauza eşecului şi încorporarea lor ca o nouă constrângere va învăţa agenţii să nu mai repete aceeaşi greşeală [YDIK98], [SHF00], [BMM01].

In [AD97], Armstrong construieşte o nouă tehnică diferită de AWCS şi ABT, în care rezolvarea problemei este divizată în epoci. Există un agent central, ce este responsabil pentru pornirea procesului de căutare şi un procesor de nogood-uri (nogood processor) ce păstrează informaţii despre nogoodurile apărute. Procesul de stocare şi utilizare ulterioară a valorilor nogood este numit ca fiiind tehnica „nogood processor" [AD97].

În [AD97] se arată experimental că se obţin reduceri ale costurilor prin aplicarea tehnicii „nogood processor". În literatură nu apar referinţe la modul în care trebuie aplicată tehnica "nogood processor" pentru tehnica AWCS, modul de distribuire a procesoarelor de nogood-uri şi care sunt beneficiile aduse asupra eficienţei. Acest lucru va fi investigat în acest studiu. Astfel, este analizat modul de

distribuire al valorilor nogood la agenţi şi modul de utilizare a informaţiilor stocate în nogood, denumită tehnica “nogood processoor”. Sunt analizate avantajele pe care le aduce tehnica nogood processor la îmbunătăţirea performanţelor tehnicii AWCS, identificând câteva moduri de distribuire ce aduc îmbunătăţiri ale eficienţei. De asemenea, se analizează comportamentul tehnicii nogood processor în situaţia filtrării mesajelor, respectiv în cazul apariţiei de întârzieri în furnizarea mesajelor.

Nerestricţionarea înregistrării valorilor nogood poate deveni în anumite cazuri impracticabilă. Motivul principal este acela că stocarea valorilor nogood consumă excesiv memorie şi poate conduce la epuizarea memoriei disponibile. În mod tipic, numărul de valori nogood creşte cu numărul de conflicte: în cel mai rău caz această creştere poate fi exponenţială în numărul de variabile. Un alt efect neplăcut al stocării unui număr mare de valori nogood este legat de faptul că verificarea asocierilor curente în lista de valori nogood stocate devine foarte costisitoare, efortul de căutare anulând beneficiile aduse de stocarea valorilor nogood. Aceste elemente sunt analizate penmtru a vedea dacă tehnică nogood processor aduce beneficii asupra eficienţei.

10.2. ADAPTAREA TEHNICII “NOGOOD PROCESSOR” ÎN CAZUL TEHNICII AWCS.

10.2.1 Implementarea tehnicii „nogood processor”.

O caracteristică a nogood-urilor este că ele sunt esenţiale pentru asigurarea completitudinii, dar au efecte asupra spaţiului necesar pentru orice algoritm DCSP. În algoritmul AWCS agenţii redescoperă şi stochează multe copii ale aceluiaşi nogood, în momente de timp diferite. Un nogood processor poate reduce costurile prin stocarea unei singure copii şi reducerea timpului de căutare.

Adaptarea şi aplicarea tehnicii nogood processor definite în [AD97] pentru tehnica AWCS a presupus găsirea unor răspunsuri la problemele apărute: cum se stochează valorile nogood, cum se distribuie stocarea şi evaluarea valorilor nogood. O altă problemă foarte importantă a fost dacă toţi agenţii sau o parte dintre ei, să apeleze procesoarele de nogooduri, pentru ca nu cumva costurile necesare evaluării nogood-urilor stocate să depăşească beneficiile aduse de tehnica nogood processor. Verificarea nogod-urilor poate induce întârzieri asupra evaluării mesajelor de către agenţi şi prin urmare influenţează costurile globale. În acest paragraf sunt propuse soluţii legate de stocarea valorilor nogood şi distribuirea acestora.

În acest studiu se consideră că fiecare agent are acces la rezultatele propriului nogood processor sau ale unui nogood processor centralizat (în mod tipic folosind mesaje). Mai mult, fiecare agent trimite (stochează) valorile nogood recepţionate nogood processor-ului asociat. Se vor considera două situaţii: un singur „nogood processor”, coordonat de un agent central şi mai multe „nogood processor”, coordonate de anumiţi agenţi. În cazul folosirii unui singur procesor, procedura de tratare a mesajelor *nogood* va stoca într-o zonă de memorie comună (dacă aceasta poate exista) sau va transmite valorile agentului central ce mânuieşte „nogood processorul”. Se va considera că valorile nogood sunt stocate în lista *nogoods_store.*

În figura 10.1. este prezentată varianta de AWSC modificată pentru a putea aplica tehnica nogood processor. În figura 10.1 sunt marcate cu ** liniile unde se apelează fiecare nogood processor.

```
when received (ok?, (xj,dj, prioritate))  do
   add (xj , dj , priority) to agent_view;
   check_agent_view;
end do;

when received (nogood, xj, nogood)  do
   add nogood to nogood_list;       *
   when (xk,dk,prioritate), where xk is not in neighbors is contained in nogood  do
     add xk to neighbors
     add (xk, dk, priority) to agent_view;
   end do
   check_agent_view;
end do;

procedure check_agent_view
  when  agent_view şi curent_value  are not consistent  do
    if   no value in Di is consistent with agent_view then
         backtrack
    else
        select d∈Di where agent_view and d minimizes the number of constraint
        violations  with lower priority agents;                                  **
         current_value ← d
         send (ok?, (xi, d, current _priority)) to neighbors
    end if
  end do
end procedure

Procedure Backtrack
  nogoods ← {V/ V= inconsistent subset of  agent_view}
  when  an empty set is an element of  nogoods  do
    broadcast to other agents that there is no solution, terminate this algorithm;
  end do
  when  no element of nogoods  is included noogod_sent   do
    for each  V ∈ nogoods  do
      add V to nogood_sent
      for  each (xj,dj,pj) in  V
         send  (nogood, xi, V ) to xj
      end do
    end do
    Pmax←max(xj,dj,pj) ∈agent:view (pj)
    current_priority ← 1+Pmax
    select  d∈Di where d minimizes the number of constraint violations
    with lower priority agents;                                              **
    current_value ← d
     send (ok?, (xi, d, current _priority)) to  neighbors
  end do
end procedure
```

Figura 10.1 Algoritmul AWCS cu nogood processor (procedurile pentru recepţionarea şi tratarea mesajelor)

Informaţiile stocate de fiecare nogood processor vor fi folosite în procesul de căutare a unei noi valori pentru fiecare variabilă avută în grijă de agent. Pentru aceasta, fiecare nogood processor va verifica (la cererea unui agent), prin intermediul subrutinei *check_inconsistent_value_nogood_processor***,** dacă valoarea selectată de agent nu a mai existat în combinaţie cu valorile agenţilor de prioritate superioară. În figura 10.2 este prezentată rutina de verificare a inconsistenţei unei noi valori. Apelul rutinei de verificare a inconsistenţei *(check_inconsistent_value_ nogood_processor)* este marcat cu ****** în algoritmul AWCS prezentat în figura 10.1.

```
function check_inconsistent_value_nogood_processor [Ai]
foreach  Nogood ∈ nogoods_store  do
 foreach x∈Nogood with prioritatea curentă din current_view *mai mare decât
          a  agentului Ai
    pos ←  position x  in Nogood
     if  x != item pos current_value
        return  consistent
   end do
  if  curent_value  != item Ai in nogood
     return  consistent
   endif
 end do
 return  inconsistent
end procedure
```

Figura 10.2 Rutina de verificare a valorilor nogood, apelată în cadrul algoritmului AWCS.

În figura 10.2. se consideră că *item pos current_value* înseamnă valoarea din lista current_value de pe poziţia pos.

Pentru a mări eficienţa procesoarelor de noogoduri în momentul în care se primeşte un mesaj de tip ok, care informează de atribuirea unei noi valori de către un anumit agent, se face verificarea de către nogood processor a valorilor pentru agenţii de prioritate superioară. În felul acesta, dacă aceea combinaţie a mai existat ca şi nogood, se încearcă căutarea unei noi valori pentru agentul curent.

O întrebare la care s-a căutat un răspuns a constat în identificarea modului de distribuire a procesoarelor de noogoduri. Sunt propuse în acest studiu două soluţii de distribuire a acestora:

- un singur nogood processor operat de un agent central. Variantele propuse în acest studiu bazate pe folosirea unui singur „nogood processor” vor fi notate cu $AWCS_{2k}$.
- mai multe „nogood processor” distribuite la anumiţi agenţi (după anumite reguli ce vor fi prezentate) Variantele cu mai multe nogood processor vor fi notate cu $AWCS_{3k}$.

În paragraful de experimente tehnica de bază AWCS va fi notată cu $AWCS_{1.}$ [YDIK98].

10.2.2. Versiuni de AWCS cu un singur "nogood processor".

Prima soluţie propusă în acest studiu constă în a adapta tehnica AWCS astfel încât să stocheze centralizat valorile nogood. Versiunile acestea vor fi notate cu $AWCS_{2k}$. Fiecare agent în momentul recepţionării unui nogood, îl transmite unui agent central ce are un nogood processor centralizat. Acest agent va stoca valoarea recepţionată într-o listă *nogood_store* (toate processoarele de nogooduri salvează doar noile valori nogood, eliminând copiile). Informaţiile de aici vor fi folosite mai târziu în procesul de căutare a unei noi valori. Prin urmare, procedura *check_agent_view* va selecta o nouă valoare consistentă cu lista agent_view şi cu lista de valori nogood stocate de „nogood processoor". Spre deosebire de varianta de bază AWCS, noua variantă $AWCS_{2k}$ va selecta o valoare nouă dacă, suplimentar, subrutina *check_inconsistent_value_nogood _processor* va returna valoarea *consistent*. De asemenea, se va proceda la apelul rutinei de verificare a valorilor nogood şi în procedura backtrack, în momentul selecţiei unei noi valori.

Pentru această primă soluţie cu un singur nogood processor, sunt propuse mai multe versiuni. Versiunile diferă prin identificarea agenţilor care apelează „nogood processorul" şi prin identificarea agenţilor pentru care se face compararea combinaţiei de valori. O primă versiune presupune aplicarea rutinei check_inconsistent_value-nogood_processor prezentată în figura 10.2, versiune notată cu $AWCS_{21}$. În cadrul rutinei *check_inconsistent_value-nogood_processor* sunt verificaţi doar agenţii ce au prioritatea curentă din current_view mai mare decât a agentului A_i. Această versiune va reprezenta varianta de bază cu "nogood processor" centralizat.

A doua versiune propusă, notată cu $AWCS_{22}$, s-a obţinut prin adaptarea rutinei *check_inconsistent_value_nogood_processor* astfel încât să verifice valorile agenţilor de prioritate superioară, prioritate avută în momentul stocării valorii nogood. Cu alte cuvinte, identificarea agenţilor de prioritate superioară faţă de agentul A_i, nu se face folosind prioritatea lor actuală (aflată în current-view) ci se face relativ la prioritatea stocată de nogood processor. Modificarea corespunzătoare s-a marcat cu * în rutina de verificare. Valorile acestor agenţi sunt verificate pentru a vedea dacă au mai existat în această combinaţie.

Următoarele versiuni se bazează pe identificarea agenţilor care vor apela procesoarele de nogooduri asociate. Sunt propuse mai multe versiuni ce diferă prin faptul că nu toţi agenţii folosesc informaţiile procesoarelor de nogooduri.

A treia versiune propusă, notată cu $AWCS_{23}$, se obţine plecând de la varianta $AWCS_{21}$ astfel încât apelarea rutinei *check_inconsistent_value_nogood_ processor* să se facă de toţi agenţii, cu excepţia celui de prioritate maximă dintre vecinii din momentul acela. A patra versiune, notată cu $AWCS_{24}$**,** se obţine de la varianta $AWCS_{21}$ astfel încât apelarea rutinei *check_inconsistent_value_nogood_ processor* să se facă doar de agentul cu prioritatea cea mai mare dintre vecini (prioritatea curentă din lista *current-view*, respectiv cea veche stocată).

O ultimă versiune, notată cu $AWCS_{25}$**,** se obţine plecând de la varianta $AWCS_{22}$ astfel încât verificarea să se facă doar de agentul cu prioritatea cea mai mare dintre vecini (prioritatea curentă din lista *current-view*, respectiv cea veche stocată). Această ultimă versiune bazată pe un nogood processor centralizat se remarcă prin faptul că identificarea agenţilor de prioritate superioară se face relativ la prioritatea lor din momentul stocării şi faptul că nu toţi agenţii apelează nogood processorul..

10.2.3.Versiuni de AWCS cu mai multe “nogood processor”.

A doua soluţie s-a bazat pe distribuirea valorilor nogood la mai multe „nogood processor”, câte unul pentru fiecare agent. Aceste versiuni vor fi notate cu $AWCS_{3k}$. În momentul recepţionării unei valori nogood, ea este stocată doar de nogood processor-ul asociat. Valoarea nogood nu mai este transmisă la un agent central, ci este stocată local. Ca şi în cazul versiunilor $AWCS_{2k,}$ informaţiile de aici vor fi folosite mai târziu în procesul de căutare.

Selectarea unei noi valori presupune şi verificarea valorilor nogood stocate. Verificarea se face în mod similar soluţiei cu un singur nogood processor, obţinându-se două versiuni notate $AWCS_{31}$ şi $AWCS_{32}$ (ultima versiune foloseşte modalitatea de identificarea a agenţilor de prioritate superioară folosind priorităţile vechi). Versiunea $AWCS_{31}$ va reprezenta varianta de bază cu “nogood processor” distribuit.

10.3. REZULTATE EXPERIMENTALE.

10.3.1 Condiţii experimentale

În acest paragraf sunt prezentate rezultatele experimentale obţinute în urma implementării şi evaluării tehnicilor asincrone cu nogood processor, pe baza modelului cu sincronizare.

Experimentele s-au desfăşurat în condiţiile prezentate în capitolul 6, paragraful §6.1. Variantele de AWCS cu nogood processor au fost aplicate pentru problema colorării unui graf, în varianta distribuită, pentru fiecare variantă s-au executat un număr de 100 încercări, reţinându-se media valorilor măsurate (pentru fiecare clasă de probleme, au fost generate aleator 10 grafuri cu acele densităţi, pentru fiecare graf generându-se aleator 10 valori iniţiale).

Pentru a putea face evaluarea acestor variante de algoritmi s-a contorizat fluxul de mesaje (cantitatea de mesaje ok şi nogood schimbate de agenţi), numărul de constrângeri verificate şi numărul de cicli necesari pentru obţinerea soluţiei.

10.3.2 Analiza rezultatelor experimentale pentru cazul aplicării tehnicilor nogood processor.

În figurile 10.3, 10.4 şi 10.5 sunt prezentate grafic rezultatele experimentale pentru cele 8 versiuni de AWCS, în funcţie de numărul de cicli, constrângeri şi fluxul de mesaje. În figura 10.3 este prezentat un studiu comparativ pentru efortul global depus de agenţi pentru obţinerea soluţiei.

Analizând graficele din figura 10.3, unde s-a reprezentat numărul de ciclii necesari obţinerii soluţiei, se remarcă faptul că tehnica nogood processor a avut efect asupra performanţelor tehnici de bază AWCS. Dintre variantele analizate se remarcă performanţele variantelor $AWCS_{22}$, $AWCS_{24}$ şi $AWCS_{25}$. Cele trei variante au avut un număr mai mic de cicli, comparativ cu celelalte variante, pentru ambele clase de probleme (cu densitate rară sau densitate mare). Totuşi, se remarcă pentru cea mai dificilă problemă (n=30, m=2.7) comportamentul cel mai bun l-a avut varianta $AWCS_{25}$, în care operarea procesorului de nogooduri s-a făcut doar de agentul cu prioritatea cea mai mare dintre vecini, comparaţiile făcându-se în raport cu priorităţile vechi.

În ceea ce priveşte efortul local depus de fiecare agent, în figura 10.4 sunt prezentate rezultatele comparative pentru cele 8 variante, rezultate obţinute prin

compararea numărului de constrângeri verificate. Se remarcă versiunile $AWCS_{22}$, $AWCS_{24}$ şi $AWCS_{25}$.

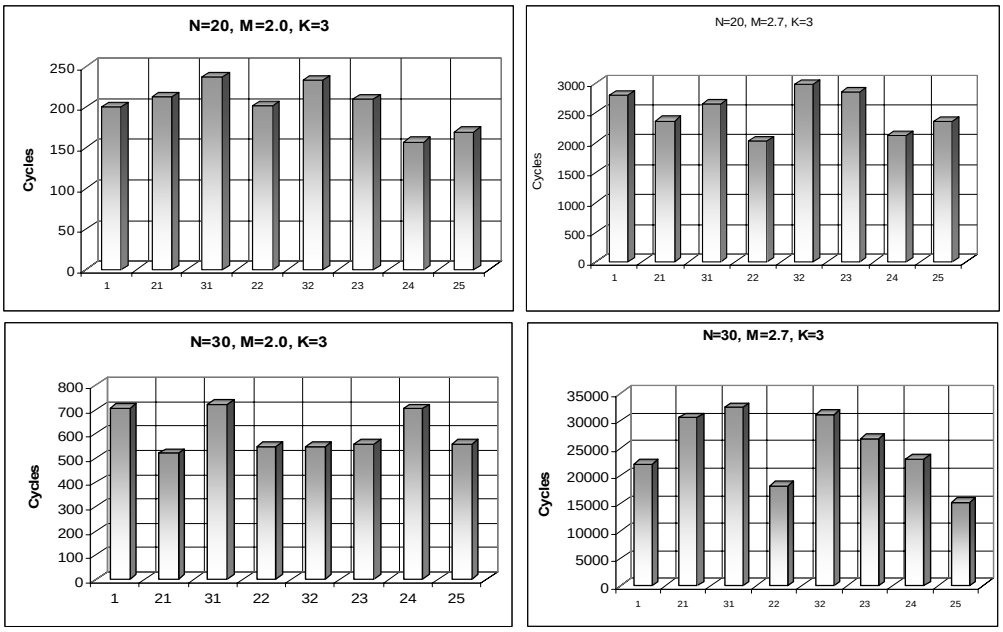


Figura 10.3 Studiu comparativ pentru versiunile de AWCS - ciclii

În ceea ce priveşte variantele de nogood processor cu distribuirea completă a acestora la fiecare agent (cu prioritate curentă sau prioritate veche), acestea nu au avut rezultate deosebite faţă de tehnica de bază.

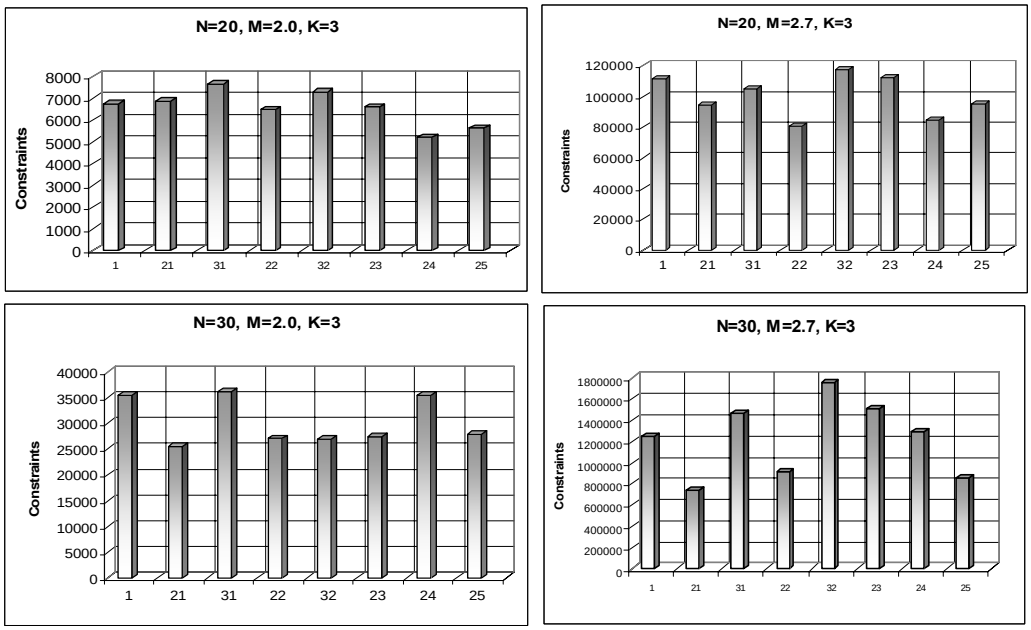


Figura 10.4. Studiu comparativ pentru versiunile de AWCS - constrângeri.

În graficul din figura 10.5 sunt prezentate rezultatele experimentale şi studiul comparativ pentru cele 8 variante în funcţie de fluxul de mesaje. Au fost contorizate mesajele de tip ok şi nogood (ce caracterizează tehnica AWCS). Se remarcă faptul că nu s-au contorizat decât mesajele nogood schimbate de agenţi pentru construirea soluţiei, fără a lua în calcul mesajele necesare pentru a transmite valorile nogood la procesoarele de nogood-uri.

Observaţiile anterioare, făcute la evaluarea în funcţie de numărul de cicli rămân valabile, remarcând performanţele tehnicilor $AWCS_{22}$, $AWCS_{24}$ şi $AWCS_{25}$.

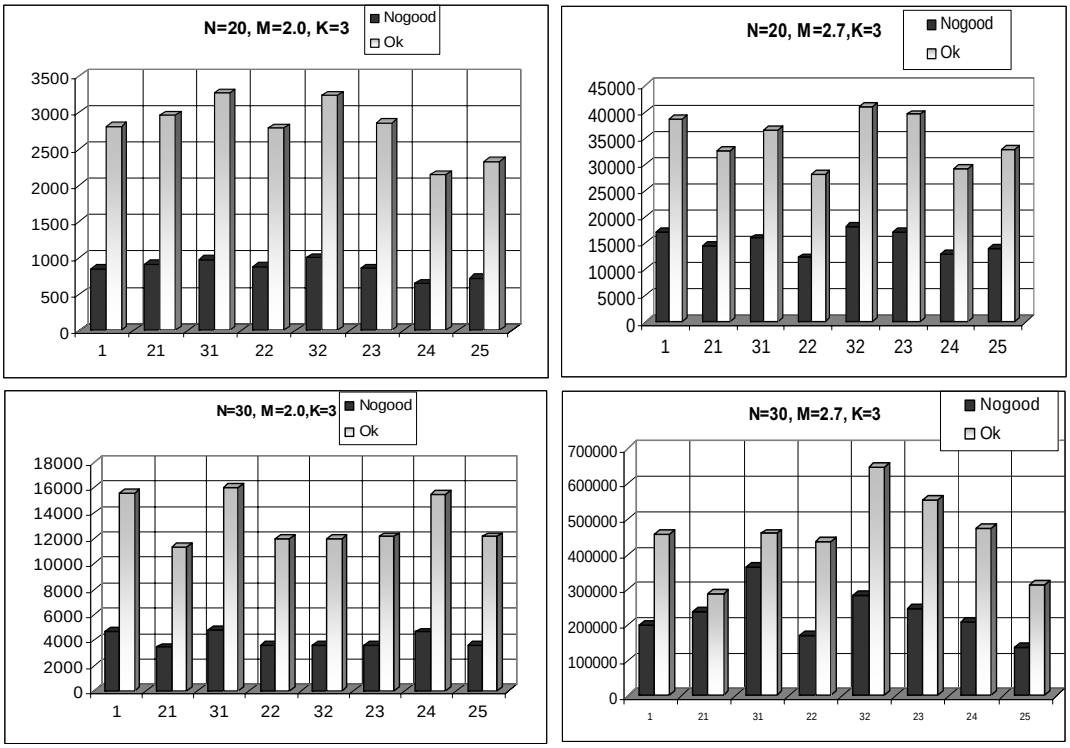


Figura 10.5 Studiu comparativ pentru versiunile de AWCS - mesaje nogood and ok

În concluzie, analizând rezultatele experimentale obţinute, se observă faptul că cele mai bune rezultate s-au obţinut pentru variantele derivate din cazul $AWCS_2$ cu un singur nogood processor centralizat. În aceste cazuri s-au obţinut îmbunătăţiri în ceea ce priveşte perfomanţele, comparativ cu tehnica de bază $AWCS_1$ cât şi cu variantele cu nogood processor distribuit.

10.3.3 Analiza rezultatelor experimentale în cazul apariţiilor întârzierilor în furnizarea mesajelor.

În acest studiu s-a dorit să se observe efectele pe care le au întârzierile în furnizarea mesajelor asupra performanţelor tehnicilor asincrone. Pentru a vedea efectul întârzierilor în furnizarea mesajelor şi pentru cazul variantelor obţinute prin aplicarea tehnicii nogood processor s-au făcut evaluări prin introducerea forţată a

unor întârzieri în furnizarea mesajelor (introduse aleator). Pentru aceasta, pentru fiecare agent s-a forţat analiza mesajelor prin introducerea unei pauze forţate. Modelul cu sincronizare SEIS utilizat la implementare a permis introducerea acestor întârzieri în rutinele de manipulare a mesajelor. Pentru analiza comportamentului la întârzieri s-a ales tehnica de bază $AWCS_1$ şi cele mai performante variante obţinute $AWCS_{22}$, $AWCS_{24}$ şi $AWCS_{25}$.

Rezultatele experimentale sunt prezentate în figurele 10.6, 10.7 şi 10.8.

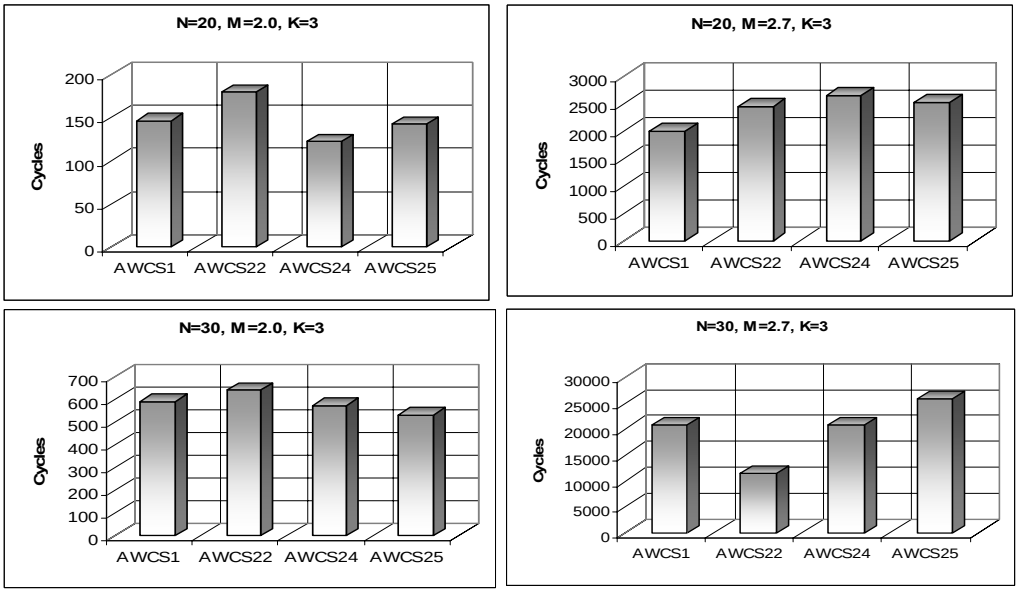


Figura 10.6. Studiu comparativ pentru versiunile de AWCS cu întârzieri aleatoare -

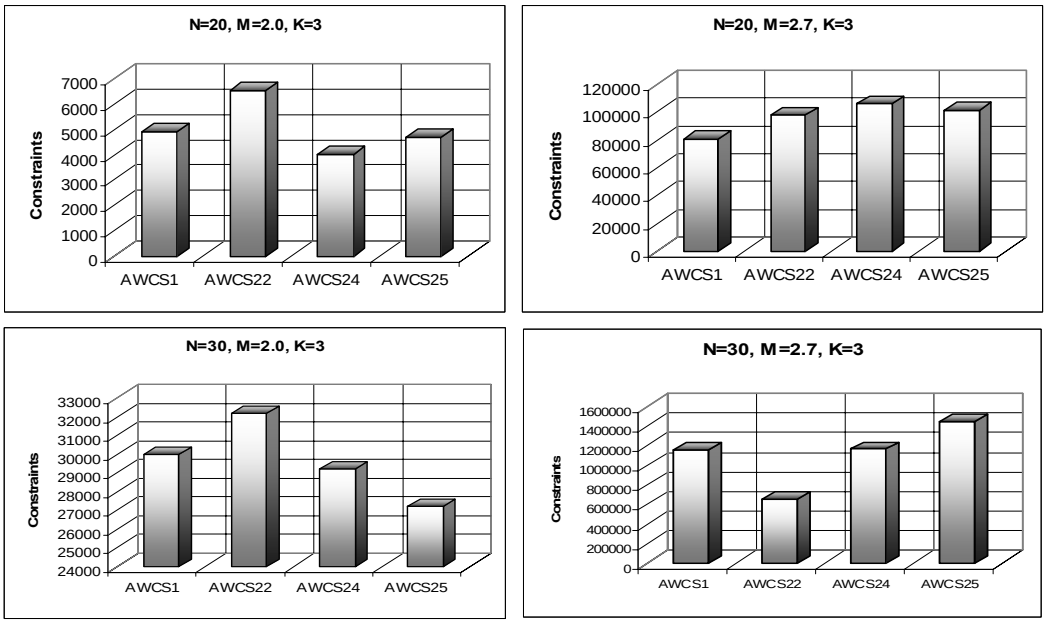


Figura 10.7 Studiu comparativ pentru versiunile de AWCS cu întârzieri aleatoare -

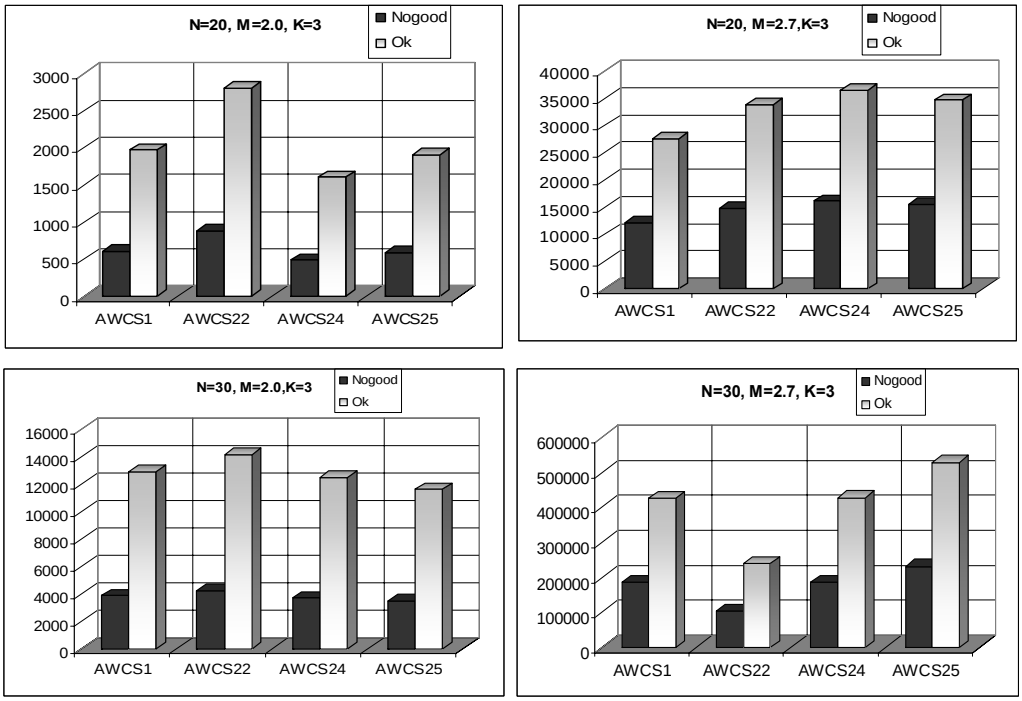


Figura 10.8 Studiu comparativ pentru versiunile de AWCS cu întârzieri

Se remarcă un comportament diferit pentru tehnicile derivate. Tehnicile $AWCS_{24}$ şi $AWCS_{25}$ au avut rezultate bune pentru probleme de densitate mică şi costuri mari în cazul problemelor de densitate mare. În schimb, tehnica $AWCS_{22}$ a necesitat costuri mai mari pentru clasele de probleme rare (sparse) si un comportament bun pentru cazul problemelor dificile.

Comportamentul anterior s-a păstrat relativ la fluxul de mesaje şi cantitatea de constrângeri verificate. Totuşi, se observă rezultate mai bune pentru tehnica de bază AWCS şi variantele derivate. Practic, nu s-au obţinut rezultate spectaculoase în ce priveşte performanţele, tehnica de bază având costuri comparabile cu celelalte variante. Aceasta arată că variantele cu nogood processor sunt sensibile la întârzieri în furnizarea mesajelor.

10.3.4 Analiza rezultatelor experimentale în cazul filtrării mesajelor.

În capitolul 6, paragraful §6.5, a fost prezentată o metodă de filtrare a mesajelor redundante şi învechite. Şi în cazul variantelor de AWCS există o cantitate de mesaje ok redundante şi învechite. Prin urmare a fost aplicată această metodă de filtrare a mesajelor ok pentru variantele cele mai performante analizate anterior ($AWCS_{22}$, $AWCS_{24}$, $AWCS_{25}$). Rezultatele sunt prezentate în figurile 10.9, 10.10 şi 10.11.

Se remarcă faptul că filtrarea a redus foarte mult costurile pentru tehnica de bază în cazul problemelor de densitate rară. În schimb, variantele cu nogood processor au avut costuri mai mici decât tehnica de bază (după aplicarea filtrării mesajelor) pentru clasele de probleme dense.

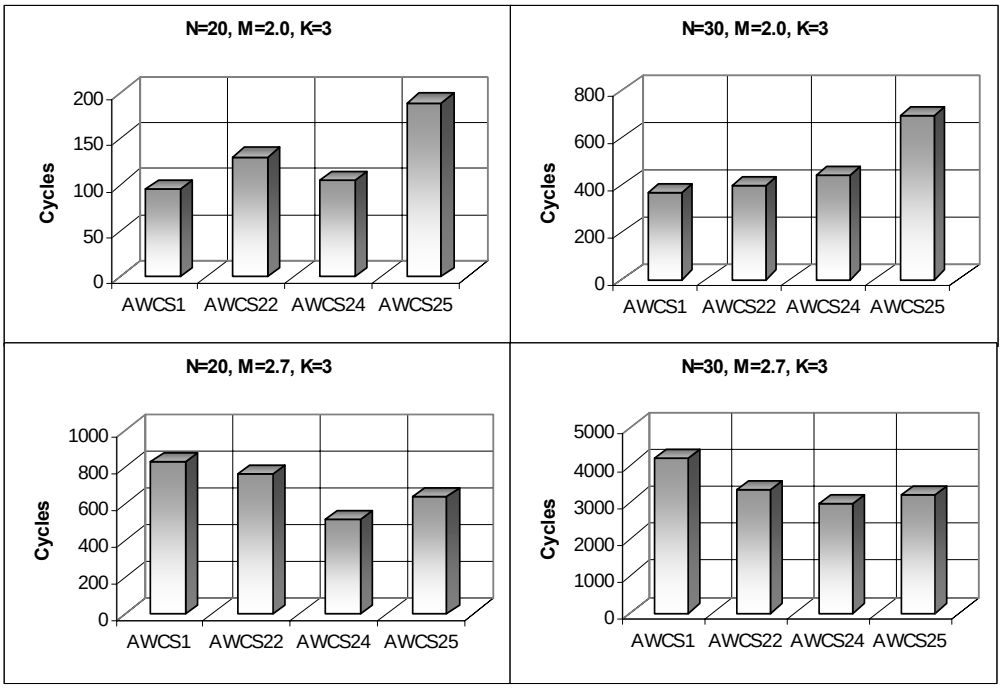


Figura 10.9 Studiu comparativ pentru versiunile de AWCS cu filtarea mesajelor - ciclii

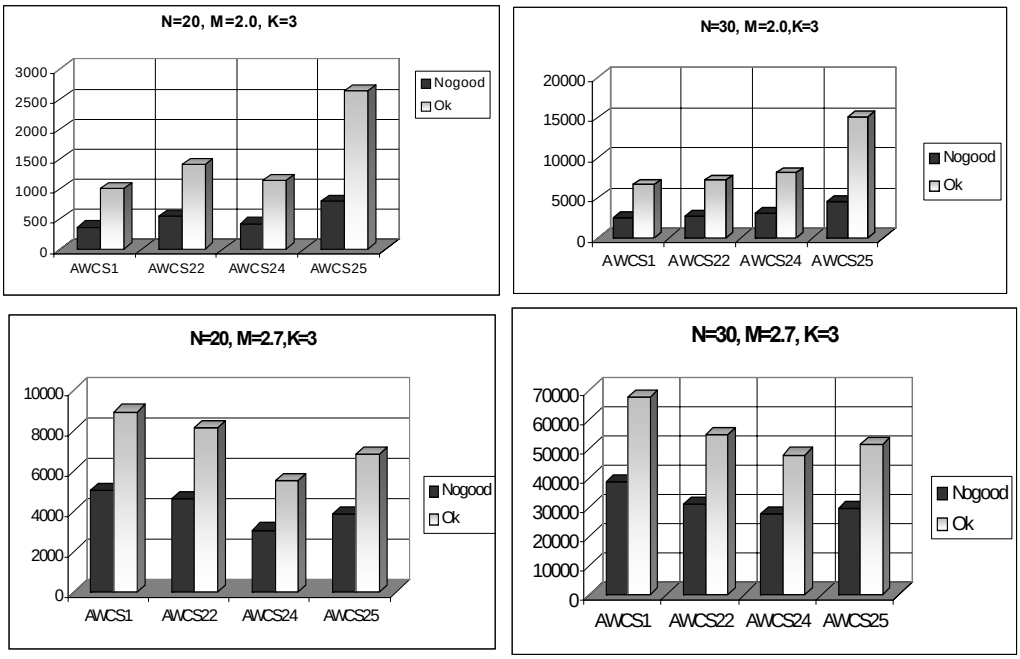


Figure 10.10. Studiu comparativ pentru versiunile de AWCS cu filtarea mesajelor – mesaje nogood şi ok

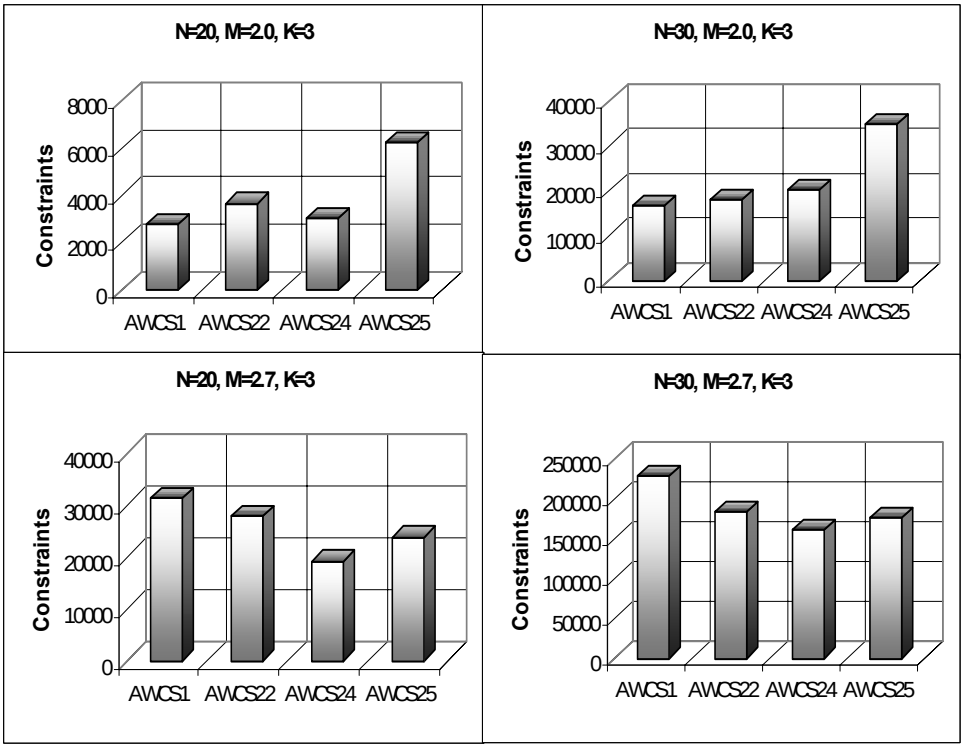


Figura 10.11. Studiu comparativ pentru versiunile de AWCS cu filtarea mesajelor – constrângeri verificate

Ca o concluzie generală, tehnica de bază AWCS a obţinut cele mai bune performanţe pentru “sparse problem”. În schimb, pentru “difficult problem”, variantele bazate de tehnica nogood processor au obţinut cele mai bune rezultate, filtrele aplicate reducând efortul de calcul (efortul local, fluxul de mesaje şi efortul global de calcul).

10.4 Concluzii.

În acest studiu s-a încercat adaptarea tehnicii nogood processor pentru tehnica AWCS. Când un nogood este descoperit în timpul procesului de căutare a soluţiei, această valoare este trimisă şi stocată de un nogood processor. Mai târziu, când se încearcă căutarea unei noi valori pentru un anume agent, se verifică suplimentar dacă aceea asociere împreună cu valorile agenţilor de prioritate superioară nu a mai existat şi prin urmare este eliminată din arborele de căutare.

Se propune o soluţie de distribuire a procesoarelor de nogooduri către agenţi, în funcţie de ordinea agenţilor, cu scopul de a reduce costurile de căutare şi de stocare. Sunt analizate experimental avantajele pe care le aduce tehnica nogood processor la îmbunătăţirea performanţelor tehnicii AWCS, identificând câteva moduri de distribuire ce aduc îmbunătăţiri ale performanţelor. De asemenea, se analizează efectul aplicării tehnicii nogood processor în situaţia filtrării mesajelor şi în cazul apariţiei de întârzieri în furnizarea mesajelor.

S-au analizat mai multe variante obţinute prin distribuirea valorilor nogood la un singur nogood processor, respectiv la mai multe nogood processor, câte unul pentru fiecare agent. Pentru identificarea agenţilor de prioritate superioară s-a luat în calcul prioritatea curentă a agentului (aflată în current-view) şi priorităţile vechi stocate în listele nogood-store.

Cele mai performante variante s-au obţinut prin distribuirea valorilor nogood la un nogood processor centralizat şi în care valorile de tip nogood sunt folosite doar la agentul cu prioritatea cea mai mare dintre vecini (raportat la prioritatea curentă sau la cea veche stocată). Îmbunătăţiri ale performanţelor s-au obţinut şi în cazul unui nogood processor centralizat ce foloseşte priorităţile vechi pentru identificarea agenţilor. În acest caz, experimentele au arătat o reducere a fluxului de mesaje, a numărului de constrângeri verificate, dar şi a numărului de cicli necesari obţinerii soluţiei. Prin distribuirea valorilor nogood şi a procesoarelor de nogooduri s-au obţinut îmbunătăţiri ale performanţelor.

Trebuie subliniat un anumit aspect relativ la costurile suplimentare pe care le poate induce aplicarea tehnicii nogood processor. Transmiterea valorilor nogood către procesoare presupune un flux de mesaje suplimentar care încarcă costurile găsirii unei soluţii. În analiza realizată aici, acest flux de mesaje a fost ignorat. Totuşi, efectul acestui flux de mesaj suplimentar s-a regăsit în numărul de cicli şi de constrângeri verificate, deci într-un nume fel a fost contorizat.

O altă situaţie analizată a fost aceea a apariţiei întârzierilor în transmiterea mesajelor. Şi aici anumite variante de AWCS cu nogood processor au avut comportamente bune, dar nu spectaculoase, tehnica de bază având costuri comparabile cu celelalte variante pentru această situaţie. Aceasta arată că variantele cu nogood processor sunt sensibile la întârzieri în furnizarea mesajelor, deşi sunt mai performante în cazul clasic în care mesajele au întârzieri fixe.

Ca o concluzie generală, tehnica nogood processor poate aduce beneficii importante asupra perfomanţelor tehnicilor asincrone, ducând la reducerea efortului de căutare a soluţiei.

Rezultatele evidenţiate în acest capitol au fost publicate în [MusCPP06] şi [MusC06].

11. COMBINAREA TEHNICILOR NOGOOD PROCESSOR ŞI NOGOOD LEARNING ÎN CAZUL TEHNICILOR DE CĂUTARE ASINCRONE DIN FAMILIA AWCS.

Plecând de la rezultatele din studiul anterior, în acest studiu se încearcă combinarea variantelor de nogood processor cu tehnicile de învăţare, cu scopul de a găsi o soluţie de îmbunătăţire a performanţelor tehnicii AWCS.

Capitolul începe cu o prezentare a contextului de lucru. În paragraful al doilea se prezintă tehnica "resolvent-based learning", introdusă în [HY00]. În al treilea capitol este prezentată o soluţie de combinare a celor două tehnici. Paragraful al patrulea prezintă o analiză experimentală a versiunilor obţinute, comparându-se performanţele relativ la tehnica de bază. În final sunt prezentate concluziile combinării celor două tehnicii în cazul tehnicii AWCS.

11.1 Introducere.

Tehnicile asincrone existente în cadrul programării cu constrângeri distribuite se caracterizează prin apariţia valorilor „nogood" în timpul căutării soluţiei. Mesajele de tip nogood sunt transmise între agenţi cu scopul de a efectua un backtracking inteligent şi de a asigura completitudinea algoritmilor. Valorile nogood arată cauza eşecului şi incorporarea lor ca o constrângere va permite agenţilor să nu mai repete aceeaşi greşeală. Aşa cum a fost prezentat în capitolele anterioare, algoritmul asynchronous weak-commitment search (AWCS) se remarcă prin faptul că suferă de o explozie a valorilor nogood, dar şi prin schimbarea dinamică a ordinii agenţilor [YDIK98]. Tehnica AWCS este considerată o tehnică eficientă obţinută prin schimbarea dinamică a ordinii agenţilor. În studiul anterior s-a văzut care este efectul stocării nogooduri şi utilizării ulterioare a informaţiilor stocate în nogooduri în cazul acestei tehnici.

În acest capitol se analizează modul în care se poate combina o tehnică de obţinerea a unor valori nogood eficiente cu o tehnică de stocare a acestora. Astfel se încearcă combinarea tehnicii "resolvent-based learning" introdusă de Yokoo, cu tehnica „nogood processor" pentru cazul tehnicii AWCS. Aceaste tehnici se referă la modul de obţinere a unor nogood-uri eficiente, respectiv la modul în care se face stocarea valorilor nogood şi folosirea ulterioară a informaţiilor furnizate de nogood-uri în procesul de selectare a unei noi valori pentru variabilele asociate agenţilor. Plecând de la această analiză, se propun anumite modificări pentru cele două tehnici cunoscute, cu scopul de a obţine o variantă mai performantă.

În [HY00], Hirayama şi Yokoo introduc termenul de *nogood learning* sau *learning,* termen ce se referă la construirea valorilor nogood. În [HY00], se prezintă şi analizează o schemă numită "resolvent-based learning", care, aplicată algoritmului AWCS permite reducerea costurilor necesare pentru a rezolva problema. Tehnica se bazează pe construirea unui nou nogood eficient, doar prin verificarea câtorva valori nogood.

În capitolul anterior este prezentată o soluţie de distribuire a procesoarelor de nogooduri identificându-se mai multe solutii de creştere a eficienţei tehnicii AWCS. În acel studiu se încearcă adaptarea tehnicii nogood processor pentru tehnica AWCS. Această tehnică constă în stocarea valorilor nogood şi folosirea informaţiilor din nogooduri la selectarea unei noi valori. Analiza din capitolul anterior a permis identificarea modului în care se distribuie nogood-urile: se propune o soluţie de distribuire a nogood procesoarelor între agenţi, conform cu ordinea agenţilor, astfel încât să se obţină o reducere a costurilor de căutare şi de stocare .

Plecând de la rezultatele din studiul anterior, în acest studiu se încearcă combinarea variantelor de nogood processor cu tehnicile de învăţare, cu scopul de a găsi o soluţie de mărire a performanţelor tehnicii AWCS. Sunt analizate beneficiile pe care le aduce combinarea tehnicii nogood processor împreună cu tehnica „resolvent-based learning", la îmbunătăţirea performanţelor tehnicii AWCS printr-un set de experimente. De asemenea, se analizează comportamentul tehnicilor obţinute în cazul aplicării tehnicii filtrării mesajelor, tehnică introdusă în capitolele anterioare.

11.2. Tehnica "resolvent-based learning".

Modelarea CSP oferă mai multe tehnici de rezolvare având la bază căutarea. Aceste tehnici pot fi îmbunătăţite prin aplicarea tehnicilor loop-back. Aceste tehnici loop-back se bazează pe folosirea informaţiilor despre căutarea realizată. Există mai multe tehnici loop-back, cunoscute din literatura de specialitate, prezentate în [FD94], [Gin93], [RR98], [BM96], [Dec90]**.**

Tehnica nogood learning, introdusă în [HY00], este o nouă metodă de învăţare pentru valorile nogood, ce se poate aplica la tehnicile DCSP, metodă bazată pe adaptarea tehnicilor loop-back prezentate în [FD94], [Gin93], [RR98], la cadrul DCSP. Ideea de bază este aceea că pentru fiecare posibilă valoare pentru variabila eşec, se selectează un nogood care interzice acea valoare şi apoi se construieşte un nou nogood în afara celui obţinut prin unirea nogood-urilor selectate. Autorii acestei metode compară acest nou nogood ca fiind asemănător cu noţiunea de rezolvent din logica propoziţională, de aici şi denumirea acestei metode *rezolvent bazat pe învăţare.*

Pentru a putea explica mai bine modul de funcţionare a acestei metode [HY00] se consideră un agent A_i, cu domeniul D_i şi fiecare valoare din acest domeniu intră în conflict cu câteva valori nogood de prioritate superioară, pentru lista curentă agent_view. Această metodă presupune construirea nogoodului pe care îl va transmite un agent A_i folosind nogoodurile stocate. Acest agent selectează o valoare nogood pentru fiecare valoare $d \in D_i$ astfel:

- agentul identifică acele nogood-uri de prioritate superioară ce sunt inconsistente relativ la contextul curent agent_view şi avem $x_i=d$;
- selectează cel mai mic dintre aceste nogood-uri.
-

În final, agentul A_i construieşte un nou nogood prin ştergerea tuturor elementelor incluzând x_i din mulţimea de nogood-uri selectate. Agentul selectează cel mai mic nogood din perspectiva priorităţilor din cauză că se doreşte să se obţină un nou nogood cât mai mic posibil.

11.3 Combinarea tehnicilor de "nogood processor" şi "nogood learning"

Combinarea celor două tehnici nu s-a putut face în mod direct. Au fost necesare mai multe adaptări pentru a putea obţine o tehnică derivată completă şi eficientă.

După cum s-a prezentat anterior, tehnica "nogood learning" intervine în cazul în care un agent A_i (cu domeniul D_i) pentru fiecare valoare din domeniul său intră în conflict cu câteva valori nogood de prioritate superioară, pentru lista curentă agent_view. De fapt, aceasta este situaţia de eşec (numită la aceste tehnici backtrack), în care se apelează rutina Backtracking. Introducerea tehnicii nogood processor are ca efect folosirea informaţiilor stocate de fiecare nogood processor în procesul de căutare a unei noi valori pentru fiecare variabila avută în grijă de agent. Pentru aceasta, fiecare nogood processor verifică (la cererea unui agent), prin intermediul subrutinei *check_inconsistent_value_nogood_ processor***,** dacă valoarea selectată de agent nu a mai existat în combinaţie cu valorile agenţilor de prioritate superioară. Prin urmare, este posibil ca la apelul rutinei de construire a unui nou nogood, pentru o anumită valoare să nu existe nici un nogood de prioritate superioară care să fie inconsistent sub lista curentă agent_view şi să apară x_i=d. În această situaţie se propune restartarea procesului de construire a valorilor nogood, selectând pentru nogood lista curentă agent_view.

Dintre variantele de nogood processor propuse în capitolul anterior s-a ales varianta cu mai multe "nogood processor", în care fiecare agent are propriul său nogood processor, apelat în momentul selectării unei noi valori. În acest studiu sunt propuse două variante, notate cu $AWCS_3$ şi $AWCS_4$, care combină tehnica nogood learning cu prima versiune de nogood processor, varianta cu mai multe procesoare. Varianta $AWCS_4$ diferă de varianta $AWCS_3$ prin faptul că verificarea se face doar de agentul cu prioritatea cea mai mare dintre vecini. O altă observaţie legată de adaptarea variantelor din capitolul anterior este aceea că nu se mai apelează suplimentar processoarele în cazul recepţionării unui mesaj ok. În paragraful următor de experimente se va considera $AWCS_1$ ca fiind tehnica de bază, iar $AWCS_2$ varianta cu „resolvent-based learning".

În figura 11.1 sunt prezentate procedurile pentru recepţionarea mesajelor pentru noua variantă, evidenţiind acele diferenţe faţă de varianta de bază AWCS (notate cu *).

```
when received (ok?, (xj,dj, prioritate))  do
   add (xj , dj , priority) to agent_view;
   check_agent_view;
end do;

when received (nogood, xj, nogood)  do
   add nogood to nogood_list;
  *
   when (xk,dk,prioritate), where xk is not in neighbors is contained in nogood  do
      add xk to neighbors
      add (xk, dk, priority) to agent_view;
   end do
   check_agent_view;
end do;
```

```
procedure check_agent_view
  when  agent_view şi curent_value  are not consistent  do
    if   no value in Di is consistent with agent_view then
         backtrack
    else
        select d∈Di where agent_view and d minimizes the number of constraint
          violations   with lower priority agents;                          **
         current_value ← d
         send (ok?, (xi, d, current _priority)) to neighbors
    end if
  end do
end procedure


Procedure Backtrack
  nogoods ←{V/V=inconsistent subset of  agent_view}//construit cu tehnica
"nogood learning"
  when  an empty set is an element of  nogoods  do
     broadcast to other agents that there is no solution,
     terminate this algorithm;
  end do
  when  no element of nogoods  is included noogod_sent   do
   for each  V ∈ nogoods  do
     add V to nogood_sent
     for  each (xj,dj,pj) in  V
        send  (nogood, xi, V ) to xj
     end do
   end do
   Pmax←max(xj,dj,pj) ∈agent:view (pj)
   current_priority ← 1+Pmax
   select  d∈Di where d minimizes the number of constraint violations
   with lower priority agents;                                          **
   current_value ← d
    send (ok?, (xi, d, current _priority)) to  neighbors
  end do
end procedure
```

Figura 11.1. Algoritmul asynchronous weak-commitment search cu nogood learning şi nogood processor (procedurile pentru recepţionarea mesajelor)

Informaţiile stocate de fiecare nogood processor vor fi folosite în procesul de căutare a unei noi valori pentru fiecare variabila avută în grijă de agent. Pentru aceasta, fiecare nogood processor va verifica (la cererea unui agent), prin intermediul subrutinei *check_inconsistent_value_nogood_processor*, dacă valoarea selectată de agent nu a mai existat în combinaţie cu valorile agenţilor de prioritate superioară. În figura 11.2 este prezentată rutina de verificare a inconsistenţei unei noi valori. Apelul rutinei de verificare a inconsistenţei *(check_inconsistent_value_nogood_processor)* este marcat cu ** în algoritmul AWCS prezentat în figura 11.1.

```
function check_inconsistent_value_nogood_processor [Ai]

foreach Nogood ∈ nogoods_store do
  foreach x ∈ Nogood with the current priority from current_view * bigger
                    than the agent's Ai
    pos ← position x in Nogood
     if x != item pos current_value
        return consistent
   end do
  if curent_value != item Ai in nogood
     return consistent
   endif
 end do
 return inconsistent
end procedure
```

Figura 11.2 Rutina de verificare a valorilor nogood

11.4. REZULTATE EXPERIMENTALE.

11.4.1 Condiţii experimentale

În acest paragraf se prezintă rezultatele experimentale, obţinute în urma implementării şi evaluării tehnicilor asincrone prezentate. Variantele AWCS (cele iniţiale şi cele obţinute prin combinarea celor două metode "nogood processor" şi „nogood learning") au fost implementate folosind modelul cu sincronizarea execuţiei agenţilor prezentat în capitolul 5. Experimentele s-au desfăşurat în condiţiile prezentate în paragraful §6.1.

Pentru a putea face evaluarea acestor variante de algoritmi s-au contorizat mesajele de tip ok şi nogood schimbate de agenţi, numărul de constrângeri verificate şi numărul de ciclii necesari pentru obţinerea soluţiei.

11.4.2 Analiza rezultatelor experimentale pentru cazul de bază.

În figurile 11.3, 11.4 şi 11.5 sunt prezentate rezultatele experimentale pentru cele 4 versiuni de AWCS, în funcţie de numărul de cicli, constrângeri şi fluxul de mesaje.

În cazul numărului de cicli necesari obţinerii soluţiei analizând graficele din figura 11.3 se remarcă performanţe foarte bune pentru variantele obţinute $AWCS_3$ şi $AWCS_4$, comparativ cu varianta de bază $AWCS_1$ şi performanţe bune comparativ cu varianta cu nogood learning–$AWCS_2$. Cele două variante au avut un număr mai mic de ciclii, comparativ cu celelalte variante, pentru ambele clase de probleme (cu densitate rară şi densitate mare). Pentru cea mai dificilă problemă (n=30,m=2.7) comportamentul cel mai bun l-a avut varianta $AWCS_4$, când manipularea procesorului de nogooduri s-a făcut doar de agentul cu prioritatea cea mai mare dintre vecini. O altă observaţie este că cele două variante propuse au costuri apropiate, cu execepţia clasei n=30, m=2.7 .

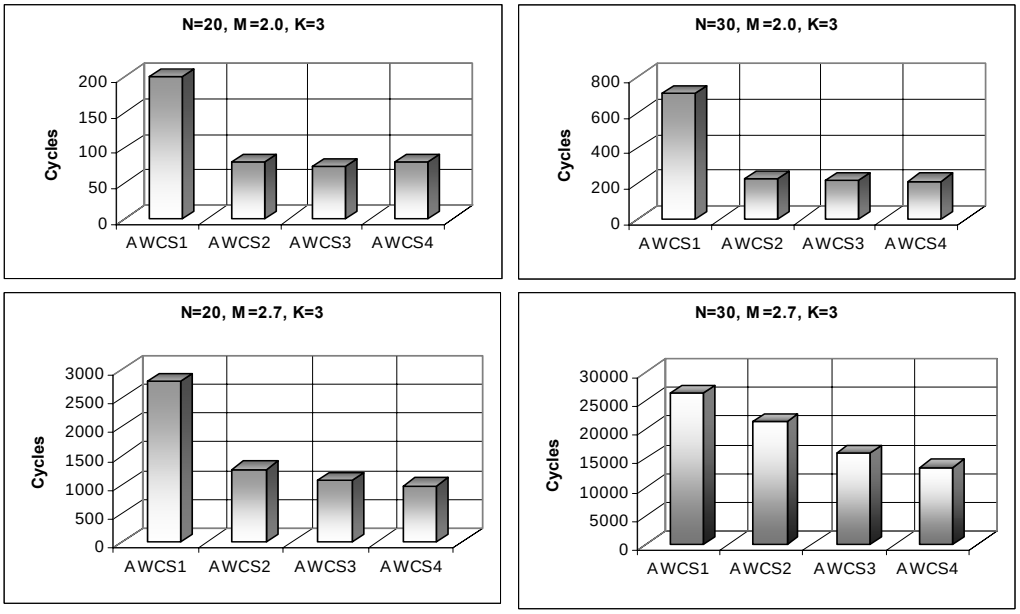


Figura 11.3. Studiu comparativ pentru versiunile de AWCS - ciclii.

În ceea ce priveşte numărul de constrângeri verificate, prezentate în figura 11.4, se remarcă faptul că variantele propuse, inclusiv varianta cu nogood learning, au necesitat un număr mai mare de constrângeri. Un efort local mai mic s-a obţinut pentru problemele de densitate mare, remarcându-se varianta $AWCS_4$.

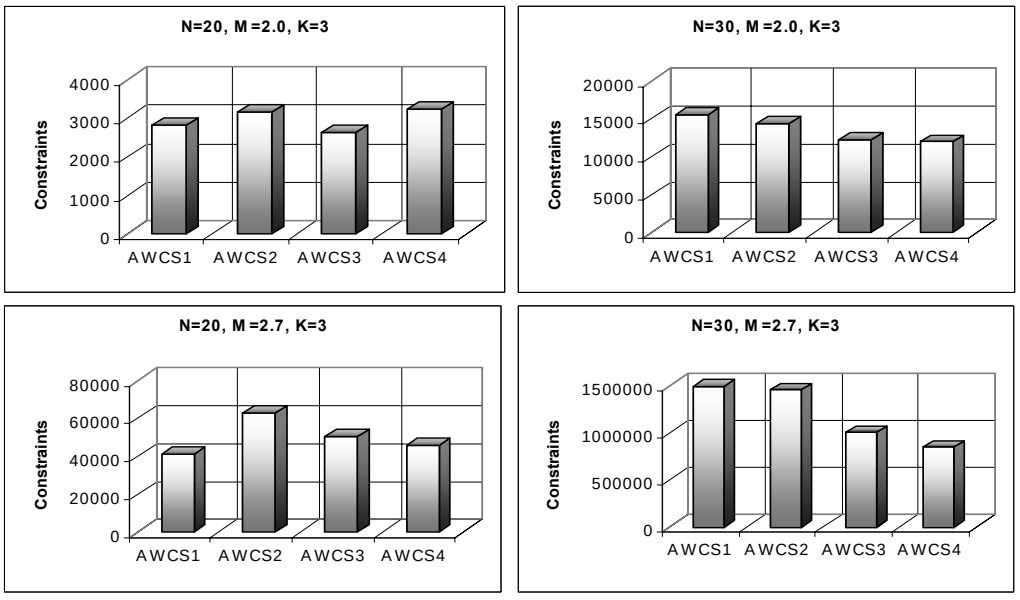


Figura 11.4. Studiu comparativ pentru versiunile de AWCS-constrângeri verificate

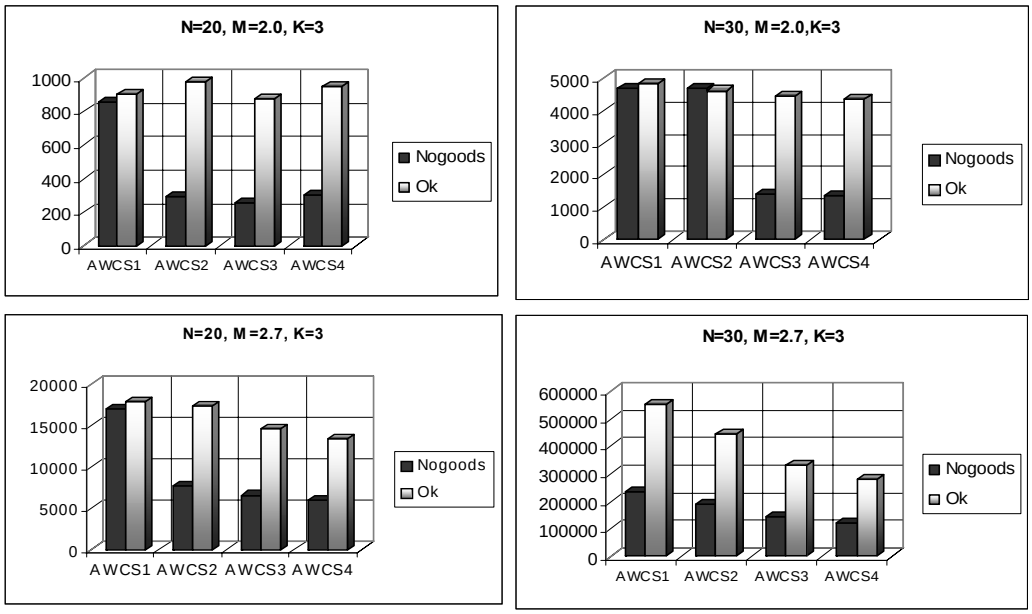


Figura 11.5 Studiu comparativ pentru versiunile de AWCS – mesaje nogood şi ok.

În ultimul grafic, cel din figura 11.5 sunt prezentate rezultatele experimentale pentru cele 4 variante, dar în funcţie de fluxul de mesaje (mesaje de tip ok şi nogood). În cadrul experimentelor realizate s-au contorizat doar mesajele nogood schimbate de agenţi, fără a lua în calcul mesajele necesare pentru a transmite valorile nogood la procesoarele de nogood-uri, ca şi în cazul analizei tehncilor nogood processor. Variantele propuse aici au avut comportamente diferite pentru cele două clase de probleme. În cazul problemelor cu densitate rară, fluxul de mesaje a fost apropiat de tehnicile de bază. În schimb, pentru probleme cu densitate mare ele se remarcă printr-un flux de mesaje mult mai mic, mai ales pentru problemele de dimensiune mare (n= 30 noduri).

11.4.3. Analiza rezultatelor experimentale în cazul filtrării mesajelor.

In capitolul 6, paragraful §6.5.2 a fost prezentată o metodă de filtrare a mesajelor redundante şi învechite. Analiza implementărilor pentru variantele de AWCS, pe baza modelului cu sincronizare, a arătat că există o cantitate de mesaje ok redundante şi învechite. Prin urmare, a fost aplicată această metodă de filtrare a mesajelor ok pentru cele patru variante evaluate. Rezultatele sunt prezentate în figurile 11.6, 11.7 şi 11.8.

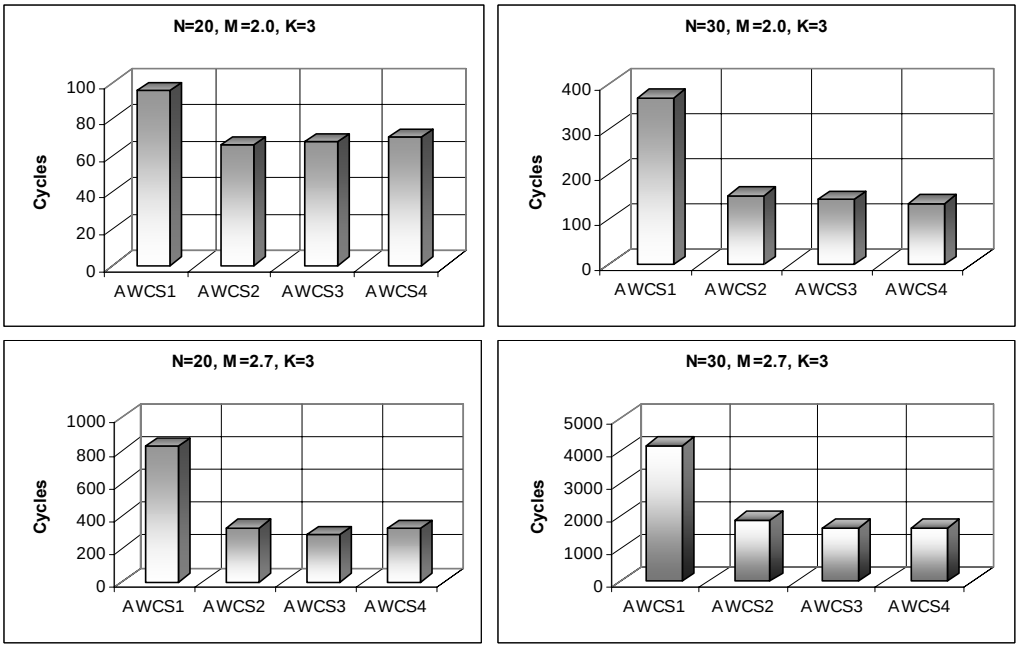


Figura 11.6 Studiu comparativ pentru versiunile de AWCS cu filtrarea mesajelor- cicli.

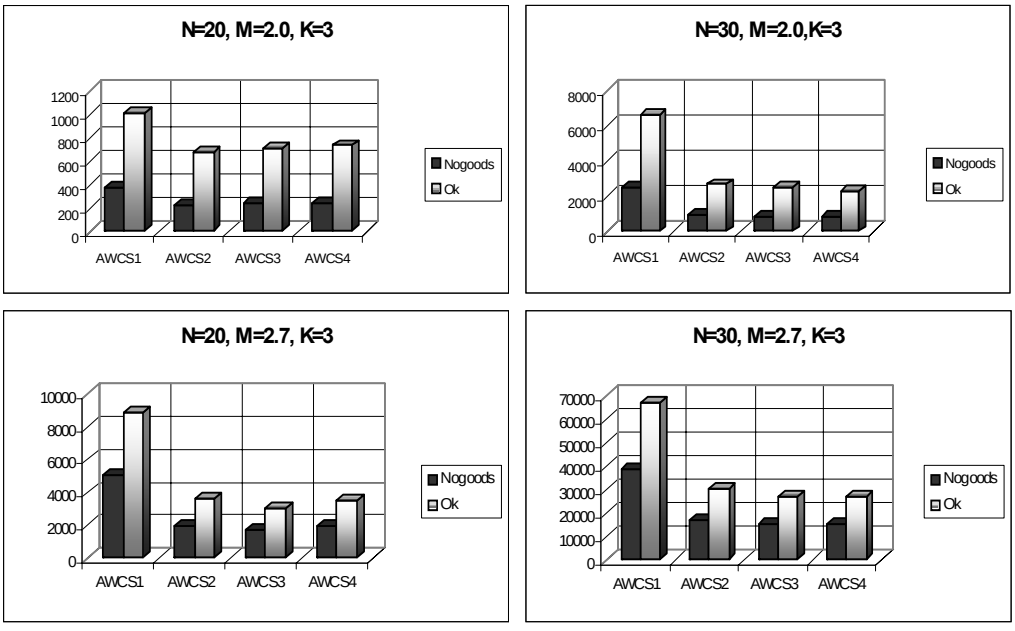


Figura 11.7 Studiu comparativ pentru versiunile de AWCS cu filtrarea mesajelor – mesaje nogood şi ok.

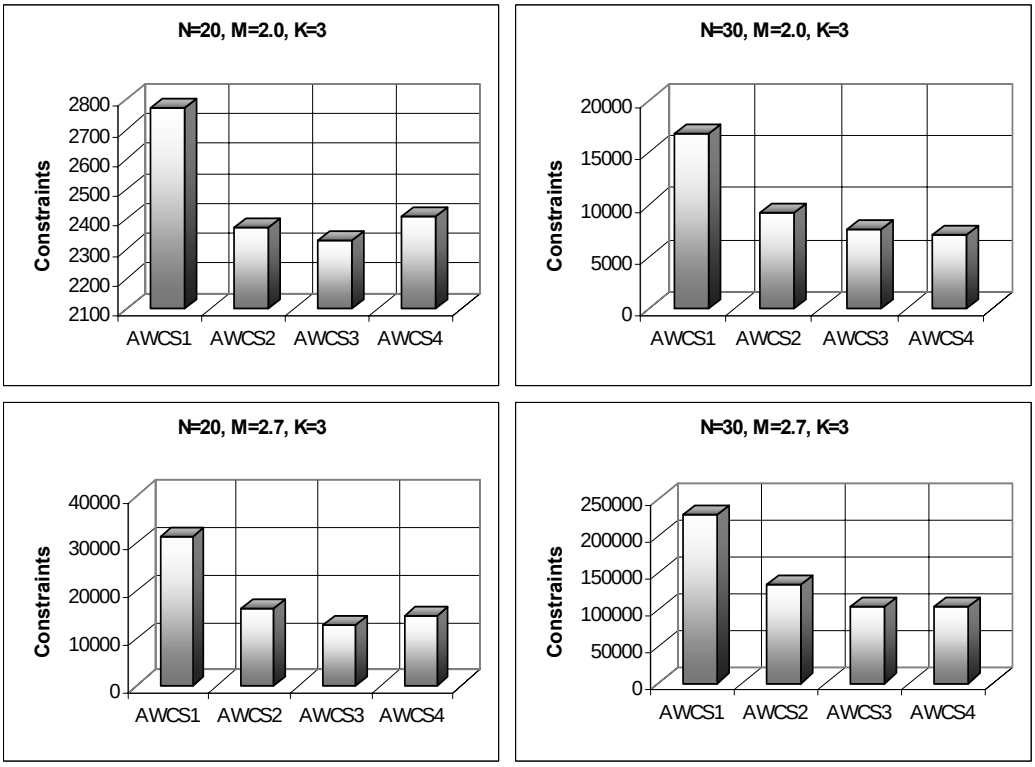


Figura 11.8 Studiu comparativ pentru versiunile AWCS cu filtrarea mesajelor–constrângeri

Se remarcă reducerea costurilor în cazul aplicării tehnicii filtrării, indiferent în ce au fost contorizate ele. Filtrarea mesajelor a permis reducerea numărului de cicli necesari obţinerii unei soluţii, dar şi a fluxului de mesaje, respectiv a numărului de constrângeri verificate. Prin urmare, rămân valabile concluziile din capitolele anterioare relative la aplicarea tehnicii filtrării pentru orice tehnică asincronă.din familiile ABT şi AWCS.

11.5. Concluzii.

În acest studiu s-a încercat combinarea a două tehnici „nogood processor" şi "resolvent-based learning" pentru cadrul AWCS. Aceste tehnici se referă la modul de obţinere a unor nogood-uri eficiente (resolvent-based learning), respectiv la modul în care se face stocarea valorilor nogood şi folosirea ulterioară a informaţiilor furnizate de nogood-uri în procesul de selectare a unei noi valori. Plecând de la aceste tehnici, sunt propuse în acest capitol anumite modificări pentru cele două tehnici cunoscute, obţinând două tehnici derivate. Scopul combinării celor două tehnici (tehnici ce se concentrează pe minimizarea şi selecţia valorilor nogood.) a fost de îmbunătăţire a eficienţei algoritmului AWCS.

Adaptarea la cadrul AWCS a necesitat câteva modificări asupra modului de construire a noului nogood, având la bază restartarea procesului de construire a

valorilor nogood. Cea mai performantă variantă s-a obţinut prin distribuirea valorilor nogood doar la agentul cu prioritatea cea mai mare dintre vecini. Evaluările experimentale au arătat o reducere a fluxului de mesaje şi a numărului de cicli necesari obţinerii soluţiei. Distribuirea valorilor nogood la anumite procesoare, împreună cu tehnica resolvent-based learning, a permis obţinerea unor îmbunătăţiri pentru performanţele tehnicii AWCS.

Filtrarea mesajelor redundante şi învechite, introdusă în capitolul 6, aplicată la variantele propuse, a condus la reducerea costurilor în găsirea soluţiei.

Ca o concluzie generală, combinarea celor două tehnici, sub forma celor două variante propuse, poate aduce beneficii importante la performanţa tehnicilor asincrone din familia AWCS.

Rezultatele acestui capitol au fost publicate în [MusC06].

SECŢIUNEA IV.
CONCLUZII FINALE ŞI PERSPECTIVE

Secţiunea finală conţine concluziile finale desprinse pe parcursul tratării temelor propuse în lucrare, rezumatul contribuţiilor şi perspectivele de continuare a cercetării în această arie tematică.

12. CONCLUZII FINALE ŞI PERSPECTIVE

12.1. Concluzii.

Teza de faţă abordează *problematica sistemelor multi-agent bazate pe tehnicile de căutare asincrone existente în cadrul programării cu constrângeri distribuite.* Principalele idei pe care se bazează teza sunt:

- Utilizarea mediului NetLOGO ca un simulator de bază în studiul tehnicilor de căutare asincrone existente în cadrul programării bazate pe constrângeri distribuite;
- Utilizarea completă şi eficientă a informaţiilor oferite de listele de tip *nogood*, inclusiv renunţarea la stocarea acestora în anumite cazuri identificate în lucrarea de faţă;
- Limitarea fluxului de mesaje pentru legăturile temporare;
- Introducerea managementului mesajelor pe baza analizei cozilor de mesaje împreună cu sincronizarea execuţiei agenţilor;
- Problemele modelate cu constrângeri fiind în general probleme NP-complete, presupun analize într-o fază de preprocesare care să identifice tehnica de căutare potrivită, versiunea de implementare şi condiţiile de rulare.

Obiectivele propuse în cadrul programului de cercetare sunt strâns legate de problemele curente ale domeniului programării cu constrângeri distribuite şi se aliniază la ideile prezentate anterior. O prezentare detaliată a modului în care au fost tratate şi atinse aceste obiective se va regăsi în acest paragraf.

(1) Conceperea şi dezvoltarea unui model pentru implementarea tehnicilor de căutare asincrone.

Analiza mediului NetLogo a permis construirea unui model general de implementare şi evaluare pentru tehnicile de căutare asincrone astfel încât să se poată utiliza mediul NetLogo ca un simulator de bază în studiul tehnicilor asincrone.

Modelul propus a presupus identificarea mai multor obiecte NetLogo necesare pentru implementarea tehnicilor de căutare asincrone:

- Implementarea agenţilor problemei DCSP;
- Implementarea mesajelor şi a canalelor de comunicaţie pentru mesaje;
- Implementarea ordinii agenţilor;
- Implementarea interfeţei de interacţiune cu utilizatorul;
- Contorizarea costurilor obţinerii soluţiei.

Sunt propuse două metode de detecţie a terminării algoritmilor, metode ce permit construirea a două sisteme multi-agent (unul sincron distribuit şi celălalt asincron distribuit) pentru evaluarea corectă a performanţelor tehnicilor asincrone. Este posibil să se facă comparaţii pentru tehnicile asincrone, pentru a vedea care variantă de implementare este de preferat: sincron sau complet asincron. Se poate studia comportamentul agenţilor în diverse situaţii şi identificarea unor posibile

îmbunătăţiri ale performanţelor tehnicilor asincrone. Sistemul NetLogo, prin intermediul modelului şi a celor două sisteme multi-agent prezentate, permite simularea altor situaţii din practică, cum ar fi apariţia întârzierilor în transmisia mesajelor. Se pot introduce pauze în prelucrarea mesajelor prin intermediul rutinelor handle-message şi studia comportamentul agenţilor în cazul apariţiilor de întârzieri în furnizarea mesajelor. De asemenea, se pot simula situaţii în care este nevoie de controlul canalelor de comunicaţii, cum ar fi filtrarea mesajelor. Aceste două sisteme multi-agent (SEIS şi SEIAS) sunt utilizate în secţiunea a III-a la implementarea şi identificarea unor îmbunătăţiri pentru câteva din tehnicile de căutare asincrone.

Ca o concluzie se poate afirma că sistemele multi-agent realizate pot fi folosite pentru studiul şi analiza tehnicilor asincrone, sistemele permiţând evaluări complete ale acestora. Aceste rezultate au fost publicate în [MusBP05] şi [MusJP06].

(2) *Conceperea unei metodologii unitare de implementare şi evaluare a tehnicilor de căutare asincrone pe baza modelului identificat anterior. Această metodologie a permis construirea celor două sisteme multi-agent pentru studiul tehniciilor de căutare asincrone.*

Plecând de la modelarea procesului de execuţie a agenţilor, în această teză este prezentată o metodologie de implementare a algoritmilor asincroni în NetLogo, exemplificată pentru cele două sisteme multi-agent. Această metodologie presupune parcurgerea mai multor etape în implementarea tehnicilor de căutare asincrone. Aceste etape presupun identificarea obiectelor aplicaţiei, construirea agenţilor şi a suprafeţei de lucru pentru aplicaţie, construirea canalelor de comunicaţie dintre agenţi, implementarea rutinelor de manipulare a mesajelor şi programul principal al aplicaţiei:

- modelarea şi construirea agenţilor DCSP folosind obiecte *turtles de tip breeds*;
- modelarea suprafeţei de lucru a aplicaţiei folosind obiecte de tip *patches*;
- reprezentarea mesajelor utilizând liste indexate. Pentru repezentarea mesajelor complexe, ce conţin foarte multe informaţii, Netlogo permite folosirea listelor de liste;
- simularea canalelor de comunicaţie pentru fiecare agent se poate face folosind liste Netlogo, pentru care se definesc rutine de tratare corespunzătoare principiilor FIFO. Aceste structuri păstrează mesajele recepţionate de fiecare agent;
- iniţializarea aplicaţiei presupune construirea agenţilor şi a suprafeţei de rulare pentru aceştia. Odată cu construirea agenţilor se iniţializează contextul de lucru al agentului (current-view), cozile de mesaje, variabile ce contorizează efortul depus de agent;
- rularea aplicaţiei presupune existenţa unui obiect grafic de tip button şi setarea proprietăţii *forever*. În felul acesta, codul ataşat sub forma unei proceduri NetLogo (ce se aplică la fiecare agent) va rula continuu, până la golirea cozilor de mesaje şi întâlnirea mesajului *stop*. Soluţia prezentată pentru sistemul cu sincronizare se bazează pe utilizarea comenzii *ask*.. Celălalt sistem, în care agenţii operează asincron, se bazează pe renunţarea la comanda ask si utilizarea unor butoane grafice ataşate la obiecte de tip *turtle*;

- ataşarea butonului grafic *observatorului*. Utilizarea acestei soluţii permite obţinerea unei soluţii de implementare cu sincronizarea execuţiei agenţilor. În acest caz, *observatorul* va fi cel care va iniţia oprirea execuţiei aplicaţiei DCSP. Acest lucru va fi realizat în momentul în care primeşte comanda *stop* de la fiecare agent. Procedura principală *update* este ataşată şi manipulată de *observator.* Aceste elemente conduc la un sistem de implementare cu sincronizarea execuţiei agenţilor -SIES.
- Un astfel de sistem multi-agent cu sincronizarea execuţiei poate fi transformat într-unul cu operarea asincronă prin ataşarea butonului grafic obiectelor de tip turtle ce simulează agenţii. Acest sistem (SIEAS) necesită utilizarea unei noi metode de detecţie a terminării execuţiei, metodă prezentată în această teză.

Rezultatele prezentate anterior au fost publicate în [MusBP05] şi [MusJP06].

(3) Studiul detaliat al comportamentului mai multor tehnici de căutare asincrone în diverse situaţii cum ar fi sincronizarea execuţiei agenţilor, apariţia întârzierilor în furnizarea mesajelor, existenţa unui graf al constrângerilor foarte dens, memorie de lucru disponibilă limitată, procesarea în pachete a mesajelor.

Modelul prezentat anterior a permis implementarea şi studiul tehnicilor asincrone din două mari familii: familia ABT şi familia AWCS. Pe baza acestor studii (în diferite situaţii) s-au identificat anumite îmbunătăţiri. Evaluarea performanţelor tehnicilor asincrone a presupus utilizarea unor unităţi de măsură care să asigure o anumită independenţă faţă de limbajele de programare folosite la implementare, în particular faţă de NetLogo. Unitaţile de măsură selectate şi implementate în cele două arhitecturi multi-agent, au permis evaluarea tehnicilor asincrone din mai multe puncte de vedere, cum ar fi efortul local şi efortul global depus de agenţi, încărcarea reţelei datorată fluxului de mesaje schimbat. Tehnicile asincrone au fost aplicate pentru problema colorării unui graf în varianta distribuită. Pentru problema colorării grafurilor s-au considerat două tipuri de probleme: grafuri cu puţine legături (numite *sparse problems, având m=n x 2 legături) şi* grafuri cu un număr special de legături, cunoscute ca fiind probleme dificile (numite *dificult problems, având m=n x 23, respectiv m=n x 2.7 legături).*

Pentru familia ABT au fost investigate şi evaluate cele două mare tehnici derivate din nucleul ABT kernel, prin eliminarea informaţiilor învechite dintre agenţi:

- tehnica ABT- obţinută prin adăugarea legăturilor între agenţii neconectaţi, legături devenite permanente.
- tehnica DisDB- se remarcă prin faptul că nu necesită legături suplimentare. Informaţiile învechite dintre agenţi sunt eliminate în timp finit.

Valorile experimentale arată că cele două tehnici asincrone necesită costuri mult mai mici pentru cazul în care agenţii operează asincron. Efortul local şi cel global depus de agenţi, inclusiv fluxul de mesaje a fost mai mic cu cu aproximativ 20-30 % pentru cazul asincron faţă de cazul cu sincronizarea execuţiei agenţilor. Comportamentul s-a păstrat pentru ambele tipuri de probleme (sparse şi dificult problems). Prin urmare, nu este recomandată sincronizarea execuţiei agenţilor pentru cele două mari tehnici (cu şi fără legături suplimentare).

Cele două sisteme SIES şi SIEAS au permis introducerea de întârzieri în furnizarea mesajelor, întârzieri introduse aleator. Indiferent de sistemul de implementare, costurile obţinerii soluţiei s-au mărit o dată cu apriţia acestor întârzieri. Cele două tehnici, bazate pe o ordine statică a agenţilor, au avut necesitat costuri mult mai mari, relativ la efortul depus de agenţi cât şi relativ la fluxul de mesaje, odată cu creşterea timpului necesar furnizării mesajelor.

A doua familie implementată şi analizată a fost familia AWCS. În cazul acestei familii există mai multe variante ce se bazează pe construirea unor nogooduri eficiente (nogood learning) sau pe stocarea şi folosirea acestor nogooduri in procesul de selectie al valorilor (nogood processor). Au fost analizate variantele de bază pentru tehnica AWCS, dar şi variantele îmbunătăţite la care s-a aplicat tehnica nogood learning sau nogood processor.

Valorile experimentale arată că în cazul variantelor AWCS, cu sau fără nogood learning, cu sau fără nogood processor, cele două tehnici asincrone necesită costuri mult mai mici pentru cazul în care agenţii operează sincron. Efortul local şi cel global depus de agenţi, inclusiv fluxul de mesaje a fost mai mic pentru cazul în care se face sincronizarea execuţiei agenţilor. Comportamentul s-a păstrat mai ales pentru probleme cu densitate mare (dificult problems), pentru cealaltă clasă de probleme costurile fiind mai apropiate. Prin urmare, în cazul acestor tehnici din familia AWCS, ce se bazează pe o ordine dinamică a agenţilor, este recomandată sincronizarea execuţiei agenţilor.

În ceea ce priveşte comportamentul la apariţia întârzierilor, surprinzător, tehnicile din familia AWCS nu au necesitat costuri mai mari, ci, din contră, în unele cazuri aceste costuri au scazut. Prin urmare, aceste tehnici, sunt recomandate pentru cazurile reţelelor în care există întârzieri în furnizarea mesajelor, mai ales în varianta cu sincronizarea execuţiei agenţilor.

Plecând de la analiza comportamentului tehnicilor din familia AWCS, în această teză este propusă o metodă de sincronizare a execuţiei agenţilor, metodă bazată pe folosirea unui mesaj jeton. Modelul cu sincronizare s-a bazat pe elementele limbajului NetLogo. Prin urmare, o metodă generală de sincronizare, care să nu depinde de un anume limbaj de implementare, a fost necesară. Această metodă a fost publicată în [MVCPP07].

Cele două sisteme multi-agent au permis analiza detaliată a cozilor de mesaje şi identificarea de mesaje redundante şi învechite, identificare realizată la recepţionarea acestor mesaje. Pentru reducerea şi eliminarea acestor mesaje, ce încarcă inutil costurile de obţinere a unei soluţii, sunt propuse şi analizate două soluţii. O primă soluţie, foarte simplă, constă în filtrarea mesajelor şi ignorarea mesajelor redundante. O a doua soluţie, mult mai bună, a constat în introducerea managementului mesajelor, astfel încât s-au putut trata complet sau în pachete mesajele recepţionate. Aceste rezultate au fost publicate în [MusCP06].

Tehnicile din familia AWCS au avut comportamente diferite faţă de cele din familia ABT. Acest lucru este probabil legat de ordinea dinamică a agenţilor.

(4) *Identificarea şi propunerea unor îmbunătăţiri ale performanţelor tehnicilor de căutare asincrone*.

- *eliminarea efectului exploziei valorilor nogood:*

Creşterea numărului de mesaje de tip nogood înregistrate determină o creştere a complexităţii algoritmilor de căutare asincroni în cazul cel mai nefavorabil. Ca o consecinţă, timpul de calcul şi necesităţile hardware cresc foarte repede, odată cu dimensiunea problemelor.

Analiza tehnicilor din familia ABT în condiţiile date de cele două sisteme multi-agent arată că aceste variante de algoritmi se caracterizează printr-o explozie a valorilor nogood, mai ales pentru probleme de dimensiune mare. Tehnicile de căutare asincrone din familia ABT necesită stocarea valorilor nogood pentru ca agenţii să nu mai repete aceeaşi greşeală, acest lucru conducând la o cantitate de memorie foarte mare, mai ales pentru probleme cu dimensiune mare.

Analiza listelor nogood arată că nu este necesară stocarea acestor valori, o soluţie de eliminare a exploziei valorilor nogood fiind aceea de etichetare a valorilor din domeniul fiecărei variabile cu flaguri. Plecând de la această analiză, este propusă o nouă tehnică. Ea implică etichetarea valorilor locale din domeniul fiecărui agent, cu flaguri. Acest lucru permite eliminarea stocării valorilor nogood şi, prin urmare, pentru probleme de dimensiune mare, evitarea exploziei valorilor nogood. Spaţiul de memorie necesar se reduce considerabil în cazul aplicării tehnicii cu flaguri.

În teză este propusă o soluţie originală de înlocuire a stocării valorilor nogood cu etichetarea valorilor din domeniu prin două tipuri de flaguri. Această soluţie este aplicată pentru nucleul ABT, obţinându-se o nouă variantă ce nu mai necesită stocarea valorilor nogood. Plecând de la acest nucleu se poate ajunge la două tehnici cunoscute: asynchronous backtracking cu flaguri (varianta Bessiere) şi Distributed Dynamic Backtracking cu flaguri. Variantele de tehnici propuse în acest studiu (ABT şi DisDB cu flaguri) sunt obţinuţi din tehnica ABT kernel prin eliminarea informaţiilor învechite dintre agenţi şi indexarea valorilor locale din domeniul unui agent cu flaguri. Acest lucru elimină stocarea valorilor nogood şi, prin urmare, pentru probleme de dimensiune mare, eliminarea exploziei valorilor nogood. Spaţiul de memorie alocat este considerabil redus în cazul variantelor cu flaguri.

Tehnicile cu flaguri pot fi îmbunătăţite prin restartarea procesului de căutare când sunt recepţionate mesaje de tip info sau back. Experimentele realizate în condiţiile unor variate probleme arată că tehnicile cu flaguri necesită mai puţin spaţiu de stocare decât tehnicile de bază. Aceste rezultate au fost publicate în [Mus05] şi [MusB05]

- *limitarea fluxului de mesaje pentru legăturile temporare ce apar în timpul căutării soluţiei:*

Analiza comportamentului tehnicii ABT_{temp} pentru diferite valori ale numărului de mesaje transmise pentru legăturile temporare a permis identificarea şi prezentarea în teză a mai multor soluţii originale de determinare a numărului de mesaje schimbate de agenţii cu legături temporare, număr optim de mesaje pentru care trebuie păstrată o legătură temporară. Experimentele realizate arată că fiecare clasă de probleme are o valoare fixă a numărului de mesaje, valoare pentru care tehnica ABT kernel cu legături temporare se comportă identic sau mai bine decât tehnica de bază ABT. Din analiza experimentală a valorilor s-a observat că există o valoare apropiată de numărul maxim de mesaje, pentru care se obţin performanţe mai bune decât varianta de bază ABT. Plecând de la această analiză, au fost propuse mai multe soluţii pentru limitarea numărului de mesaje transmise.

O primă soluţie propusă este aceea de a determina pentru fiecare agent, înainte de rulare, un număr fix de mesaje ce trebuie transmise pentru o legătură. Acest număr este determinat în mai multe moduri, fiind propuse două variante statice. Experimentele arată că pentru probleme cu dimensiune mică soluţia este eficientă, dar pentru dimensiune mare varianta de bază depăşeşte varianta cu număr temporar de legături.

O a două soluţie propusă, mult mai performantă, constă în a determina dinamic numărul maxim de mesaje în timpul rulării. Practic, se deduce dinamic în

timpul rulării, valoarea maximă a numărului de mesaje şi ea este ajustată în funcţie de evoluţia algoritmului. Pentru această soluţie, s-a propus un algoritm de determinare a numărului maxim de mesaje care nu necesită zone comune de memorie. Evaluările experimentale realizate în condiţiile unor probleme variate, cu valori iniţiale alese aleator, pentru probleme cu densităţi variate, arată că soluţia dinamică este mai eficientă decât tehnica de bază. Rezultatele prezentate mai sus au fost publicate în [MusPP05a].

- *combinarea legăturilor temporare cu cele permanente:*

Un nou membru hibrid pentru familia ABT, membru derivat din nucleul ABT prin eliminarea informaţiilor învechite dintre agenţi, combinând două metode mai vechi de eliminare a informaţiilor învechite dintre agenţi: adăugarea de legături permanente între agenţi, respectiv adăugarea de legături temporare, este propus. Noul membru presupune transformarea anumitor legături temporare în legături permanente, pe baza informaţiilor relative la fluxul de mesaje învechite recepţionate de fiecare agent. Noul membru este obţinut pe baza studiului legăturilor temporare.

Pentru obţinerea acestei tehnici hibride se propune o metodă de determinare a numărului de mesaje ce se transmit pentru legăturile temporare şi o soluţie de determinare a legăturilor temporare ce sunt transformate în legături permanente. Evaluările experimentale realizate în condiţiile unor probleme variate, cu valori iniţiale alese aleator, pentru probleme cu densităţi variate, arată că soluţia hibridă este mai eficientă decât tehnica de bază, pentru ambele tipuri de probleme (cu densitate mică şi dificilă), pentru dimensiuni mici şi pentru dimensiuni mari ale problemelor. Aceste rezultate au fost publicate în [MusPP06].

- *aplicarea tehnicii nogood processor la tehnica AWCS:*

Tehnica nogood processor este aplicată în cazul familiei AWCS. Când un nogood este descoperit în timpul procesului de căutare a soluţiei, această valoare este trimisă şi stocată de un nogood processor. Mai târziu, când se încearcă căutarea unei noi valori pentru un anume agent, se verifică suplimentar dacă acea asociere împreună cu valorile agenţilor de prioritate superioară nu a mai existat şi prin urmare este eliminată din arborele de căutare. Pentru aceasta, se propune o soluţie de distribuire a procesoarelor de nogooduri către agenţi, în funcţie de ordinea agenţilor, cu scopul de a reduce costurile de căutare şi de stocare. Sunt analizate experimental avantajele pe care le aduce tehnica nogood processor la îmbunătăţirea performanţelor tehnicii AWCS, identificând câteva moduri de distribuire ce aduc îmbunătăţiri ale performanţelor.

Cele mai performante variante s-au obţinut prin distribuirea valorilor nogood la un nogood processor centralizat şi în care valorile de acolo sunt folosite doar la agentul cu prioritatea cea mai mare dintre vecini (raportat la prioritatea curentă sau la cea veche stocată). Îmbunătăţiri ale performanţelor s-au obţinut şi în cazul unui nogood processor centralizat ce foloseşte prioritătile vechi pentru identificarea agenţilor. În acest caz, experimentele au arătat o reducere a fluxului de mesaje, a numărului de constrângeri verificate, dar şi a numărului de cicli necesari obţinerii soluţiei. Prin distribuirea valorilor nogood şi a procesoarelor de nogooduri s-au obţinut îmbunătăţiri ale performanţelor.

Ca o concluzie, tehnica nogood processor poate aduce beneficii importante asupra perfomanţelor tehnicilor asincrone, ducând la reducerea efortului de căutare a soluţiei. Rezultatele evidenţiate mai sus au fost publicate în [MCPP06] şi [MusC06].

- *Combinarea tehnicii nogood processor cu tehnicile de învăţare:*

Având ca punct de plecare tehnica *nogood processor* se propune combinarea acestei tehnici cu tehnicile de învăţare pentru cadrul AWCS. Aceste tehnici se referă la modul de obţinere a unor nogooduri eficiente (resolvent-based learning), respectiv la modul în care se face stocarea valorilor nogood şi folosirea ulterioară a informaţiilor furnizate de nogooduri în procesul de selectare a unei noi valori. Pentru a putea combina cele două tehnici sunt propuse în această teză anumite modificări pentru cele două tehnici cunoscute, obţinând două tehnici derivate. Scopul combinării celor două tehnici (tehnici ce se concentrează pe minimizarea şi selecţia valorilor nogood.) a fost de îmbunătăţire a performanţelor algoritmului AWCS.

Adaptarea la cadrul AWCS a necesitat câteva modificări referitoare la modul de construire a noului nogood, având la bază restartarea procesului de construire a valorilor nogood. Cea mai performantă variantă s-a obţinut prin distribuirea valorilor nogood doar la agentul cu prioritatea cea mai mare dintre vecini. Evaluările experimentale au arătat o reducere a fluxului de mesaje şi a numărului de cicli necesari obţinerii soluţiei. Distribuirea valorilor nogood la anumite procesoare, împreună cu tehnica resolvent-based learning, a permis obţinerea unor îmbunătăţiri pentru performanţele tehnicii AWCS.

Combinarea celor două tehnici, sub forma celor două variante propuse, poate aduce beneficii importante la performanţa tehnicilor asincrone din familia AWCS. Aceste ultime rezultate au fost publicate în [MusC06].

12.2. Rezumat al contribuţiilor

Lucrarea de faţă aduce o serie de contribuţii în domeniul dezvoltării sistemelor multi-agent bazate pe constrângeri distribuite. Un scurt rezumat al acestora este prezentat în continuare:

Capitolul 5:

- conceperea şi prezentarea unui model de implementare şi evaluare în NetLogo, ce permite modelarea şi simuarea procesului de execuţie pentru agenţi;
- prezentarea a două metode de detecţie a terminării procesului de execuţie a algoritmilor asincroni, pentru modelul de simulare propus anterior.
- definirea unui sistem multi-agent de implementare şi evaluare pentru tehnicile de căutare asincrone cu sincronizarea execuţiei agenţilor-SIES;
- definirea unui sistem multi-agent de implementare şi evaluare pentru tehnicile de căutare asincrone cu operarea asincronă a agenţilor-SIEAS.
- conceperea şi prezentarea unei metodologii de implementare pentru tehnicile de căutare asincrone utilizând sistemele SIES şi SIEAS.

Capitolul 6:

- analiza comportamentului tehnicilor din familia ABT;
- analiza comportamentului tehnicilor din familia AWCS;
- introducerea managementului mesajelor în cazul tehnicilor din familia AWCS;
- definirea unui protocol de tratare completă a mesajelor din canalele de comunicaţie pentru familia AWCS;

- introducerea sincronizării parţiale şi complete în cazul tehnicilor de căutare asincrone prin prezentarea unei soluţii;
- definirea unui metode de filtrare a mesajelor redundante şi învechite.

Capitolul 7:

- analiza comportamentului tehnicilor din familia ABT în raport cu fluxul de mesaje nogood;
- identificarea tipurilor de mesaje nogood în cazul tehnicilor din familia ABT;
- eliminarea stocării valorilor nogood prin introducerea flagurilor în cazul familiei ABT;
- o soluţie de aplicare a tehnicii cu flaguri în cazul familiei ABT. Este înlocuită stocarea valorilor nogood cu etichetarea valorilor din domeniu prin două tipuri de flaguri, obţinându-se o tehnică originală cu flaguri;
- îmbunătăţirea performanţelor tehnicilor cu flaguri prin restartarea procesului de căutare când sunt recepţionate mesaje de tip info sau back.

Capitolul 8:

- analiza comportamentului tehnicii ABT_{temp} pentru diferite valori ale numărului de mesaje transmise pentru legăturile temporare;
- propunerea a două soluţii statice de limitare a numărului de mesaje transmise;
- propunerea a două soluţii dinamice de determinare a numărului maxim de mesaje în timpul rulării. Pentru aceasta, se deduce dinamic în timpul rulării, valoarea maximă a numărului de mesaje şi ea este ajustată în funcţie de evoluţia algoritmului;
- un algoritm de determinare a numărului maxim de mesaje este propus, fără a necesita zone comune de memorie;
- o analiză experimentală ce investighează comportamentul variantelor cu limitarea fluxului de mesaje, analiză ce arată că soluţiile dinamice propuse necesită costuri mai mici.

Capitolul 9:

- o metodă de determinare a numărului de mesaje ce se transmit pentru legăturile temporare şi o soluţie de determinare a legăturilor temporare ce sunt transformate în legături permanente;
- un membru hibrid pentru familia ABT, membru derivat din nucleul ABT prin eliminarea informaţiilor învechite dintre agenţi, combinând două metode mai vechi de eliminare a informaţiilor învechite dintre agenţi: adăugarea de legături permanente între agenţi, respectiv adăugarea de legături temporare, este propus;
- o analiză experimentală în condiţiile unor probleme variate, cu valori iniţiale alese aleator, pentru probleme cu densităţi variate, ce arată comportamentul bun al noii tehnici hibride propuse.

Capitolul 10:

- o soluţie de aplicare a tehnicii nogood processor în cazul tehnicii AWCS;
- o metodă de distribuire a valorilor nogood la un singur procesor de nogooduri;
- o metodă de distribuire a valorilor nogood la mai multe procesoare de nogooduri;
- o analiză experimentală a efectelor aplicării soluţiilor de distribuire a valorilor nogood şi de folosire în cazul selecţiei unei noi valori;

Capitolul 11:

- o soluţie de combinare a tehnicii nogood processor cu tehnicile de învăţare pentru cadrul AWCS;
- o analiză experimentală a efectelor aplicării celor două metode de utilizare a informaţiilor din valorile nogood.

12.3. Concluzii finale

Reţelele cu constrângeri distribuite şi-au dovedit succesul în modelarea problemelor reale. Multe probleme reale, cum ar fi cele de planificare, configurare, diagnoză, teoria grafurilor, designul circuitelor, planificare genetică, designul maşinilor, înţelegerea limbajelor etc. pot fi văzute ca şi categorii de probleme de satisfacere a constrângerilor. Cele mai multe dintre problemele abordate de programarea cu constrângeri sunt probleme NP-complete.

Teza de faţă demonstrează că aplicarea acestor tehnici de căutare asincrone presupune analize într-o fază de preprocesare pentru a identifica tehnica potrivită şi condiţiile de rulare. Modelele propuse în această teză permit implementarea şi evaluarea tehnicilor ce vor fi aplicate la anumite probleme reale şi identificarea condiţiilor optime de rulare practică.

Utilizarea metodologiei de implementare şi evaluare pentru tehnicile de căutare asincrone permite reducerea efortului de implementare. Variantele de implementare obţinute pot fi adaptate cu un efort mic, se pot simula situaţii complexe din practică, cum ar fi existenţa întârzierilor în furnizarea mesajelor, existenţa unor grafuri pentru constrângeri dense sau rare (corespunzătoare unor probleme dificile sau simple). Studiile arată că cele două mari familii de tehnici de căutare asincrone, au comportamente total diferite în apariţia întârzierilor sau introducerea sincronizării execuţiei agenţilor.

Analiza a două mari familii de tehnici de căutare asincrone arată că există diverse comportamente pentru agenţi, comportamente ce pot influenţa în bine sau în rău costurile de obţinere a soluţiei. Prin urmare, o analiză într-o fază de preprocesare este necesară. O astfel de analiză se poate face prin implementarea acestor tehnici în NetLogo şi identificarea acelei tehnici ce necesită costuri mici, pentru situaţiile dorite. Se pot selecta tehnicile care necesită un flux de mesaje mai mic, dar eforturi de calcul mai mari, pentru situaţiile de canale de comunicaţie lente şi sisteme de calcul puternice. În schimb, în cazul existenţei unor canale de comunicaţie rapide, dar cu sisteme de calcul mai puţin puternice, se pot selecta acele tehnici care necesită un flux de mesaje mai mare, dar eforturi în constrângeri mai mici.

Studiile din această teză arată că pot apărea situaţii, cum ar fi explozia valorilor nogood, ce pot face impracticabilă o anumită tehnică, chiar dacă ea este eficientă. În astfel de situaţii soluţia propusă în această teză de etichetare a valorilor din domeniu prin flaguri poate deveni o soluţie.

O altă situaţie ce poate apărea în practică este aceea în care nu este posibilă adăugarea de noi legături între agenţi din motive de securitate sau pentru că canalele de comunicaţie nu permit un flux rapid de mesaje. În aceste situaţii, trebuie identificate alte tehnici sau aplicate metodele prezentate în această lucrare de limitare a fluxului de mesaje.

O ultimă concluzie este legată de folosirea completă a informaţiilor din mesajele nogood. Acest lucru permite reducerea costurilor în obţinerea soluţiei, mai ales pentru probleme dificile cu un graf al constrângerilor foarte dens.

12.3. Perspective de cercetare şi dezvoltare

Problemele abordate în această teză, modelele, tehnicile şi algoritmii prezentaţi permit continuarea preocupărilor şi deschid noi perspective în domeniul de mare interes al sistemelor multi-agent bazate pe constrângeri distribuite.

Se doreşte continuarea cercetărilor întreprinse în cadrul acestui program de doctorat, urmând două direcţii:

(a) a îmbunătăţirii şi a extinderii modelelor NetLogo folosite la implementarea tehnicilor asincrone. Este vorba de extinderea acestor modele NetLogo pentru a permite cadrul multivariabilă, managementul mesajelor şi prelucrarea simultană a tuturor mesajelor din cozile de mesaje. De asemenea, se doreşte introducerea noilor probleme pentru analiza tehnicilor asincrone, probleme numite „binary random problems", care permit studierea mult mai exactă a comportamentelor tehnicilor asincrone, problema colorării grafurilor fiind un caz particular;

(b) a identificării altor soluţii de îmbunătăţire a performanţelor tehnicilor asincrone.

O primă idee de îmbunătăţire a performanţelor este legată de studiul mesajelor redundante, în sensul identificării cauzelor care duc la apariţia acestora şi găsirea altor soluţii de reducere sau eliminare a acestora, dar înainte de a fi transmise către agenţi.

A doua idee este legată de studiul tehnicilor în cazul existenţei ordinilor dinamice asupra agenţilor. Recent, a fost dezvoltată o variantă ABT cu ordine dinamică, variantă foarte performantă ce se bazează pe folosirea unor euristici speciale. Se doreşte continuarea acestor studii având ca ţintă identificarea altor euristici pentru determinarea ordinii dinamice şi găsirea unei soluţii de aplicare la cadrul mult mai general al familiei ABT.

O a treia temă de studiu (relativ la îmbunătăţirea performanţelor) se referă la încercarea de extindere a familiei ABT pentru cazul în care agenţii operează cu mai multe variabile.

13. BIBLIOGRAFIE

[AD97] A. Armstrong and E. Durfee. "Dynamic Prioritization of Complex Agents in Distributed Constraint Satisfaction Problems", in *Proceedings of the 15th IJCAI*, Nagoya, Japan, 1997, pp. (620-625).

[BM03] I. Brito and P. Meseguer. "Distributed forward checking", in *Proceedings of the CP 2003*, Kinsale, Ireland, 2003, pp. (801-806).

[BM04] I. Brito and P. Meseguer. "Synchronous, asynchronous and hybrid algorithm for DisCSP", in *Proceedings of the Workshop on Distributed Constraints Reasoning (DCR-04)*, CP-2004, Toronto, 2004, pp. (80-94).

[BM05] C. Bessière, A. Maestre, I. Brito and P. Meseguer. "Asynchronous Backtracking without adding links: a new member in the ABT Family", *Artificial Intelligence*, vol. 161, no. 1-2, 2005, pp. (7-24).

[BM96] R. J. Bayardo and D.P. Miranker. "A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraint Satisfaction Problem", in *Proceedings of the Thirteenth National Conference on Artificial Intelligence,* 1996, pp. (298– 304).

[BMM01] C. Bessière, A. Maestre and P. Meseguer. "Distributed Dynamic Backtracking", in *Proceedings of the IJCAI'01 Workshop on Distributed Constraint Reasoning*, Seattle WA, 2001, pp. (9-16).

[Box78] F. Box. "A heuristic technique for assignment frequencies to mobile radio nets", *IEEE Transactions on Vehicular technology, vol.* 27, no.2, pp.(57-64).

[BR75] J.R. Bitner and E.M. Reingold. "Backtrack programming technique", *Comunications of the ACM*, vol. 18, no. 11, 1975, pp. (651-656).

[CDK91] Z. Collin, R. Dechter and S. Katz. "On feasibility of distributed constraint satisfaction", in *Proceedings of the Twelfth International Joint Conferance on Artificial Inteligence*, 1991, pp. (318-324).

[CL85] K.M. Chandy and L. Lamport. "Distributed snapshots: Determining global states of distributed systems", *ACM Transactions on Computer Systems*, vol. 3, no. 1, 1995, pp. (63–75).

[Coo90] M.C. Cooper. "An optimal k-consistency algorithm", *Artificial Inteligence,* vol. 41, no. 1, 1990, pp. (89-95).

[Dec90] R. Dechter. "Enhancement Schemes for Constraint Proccessing: Backjumping, Learnig and CutSet Decomposition", *Artificial Inteligence*, vol. 41, no. 1, 1990, pp. (273-312).

[DF99] R. Dechter and D. Frost. "Backtracking algorithms for constraint satisfaction problems", Technical Report, Information and Computer Science Department, UC Irvine, 1999.

[DM94] R. Dechter and I. Meiri. "Experimental evaluation of preprocessing algorithms for constraint satisfaction problems", *Artificial Inteligence*, vol. 68, 1994, pp. (211-241).

[DP88] R. Dechter and J. Pearl. "Network-based heuristics for constraint satisfaction problems". *Artificial Intelligence*, vol. 34, 1998, pp. (1-38).

[DP89] R. Dechter and J. Pearl. "Tree clustering schemes for constraint processing". *Artificial Inteligence,* vol.38, 1989, pp. (353-366).

[FD94] D. Frost and R. Dechter. "Dead-end driven Learning", in *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994, pp. (294–300).

[FQ85] E. Freuder and M. Quinn. "Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems", in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985, pp. (1076-1078).

[Fro97] D. Frost. "Algorithms and Heuristics for Constraint Satisfaction Problems", *PhD thesis*, Information and Computer Science, University of California, Irvine, CA, 1997.

[Gal06] M. Galley. "Distributed constraint programming platform using sJavap". Available: http://cs.fit.edu/Projects/asl/#MELY, 2006.

[Gas79] Gaschnig, J. Performance Measurement and Analysis of certain Search Algorithms *. PhD thesis*, Carnegie Mellon University, Pittsburgh, 1979.

[Gin93] M. Ginsberg. "Dynamic backtracking"*. Journal of Artificial Intelligence Research,* vol. 1, 1993, pp. (25-46) *.*

[GWW03] C. Gwen-Hua, L. Wei-Li, C.L Wang**.** "Asynchronous Backtracking Algorithm with no effect of nogood explosion", in *Proceedings of the International Conference on Computer, Communication and Control Technologies - CCCT2003*, Orlando, Florida, Vol. IV, 2003, pp. (272-279).

[Ham99] Y. Hamadi. "Traitement des problemes de satisfaction de constraintes distributes", *PhD theisi*, Universitatea Motpellier II, 1999.

[Hav97] W. Havens. "Nogood caching for multiagent backtrack search", in *Proceedings of AAAI Constraints and Agents Workshop (W5)*, 1997.

[HBQ98] Y. Hamadi, C. Bessiere and J. Quinqueton. "Backtracking in Distributed Constraint Networks", in *Proceedings of the 13th European Conference on Artificial Intelligence- ECAI'98* , Brighton, UK, 1998, pp. (219-223).

[HE80] M. Haralick, G.L. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, vol. 14, 1980, pp. (263-313).

[HT95] K. Hirayama and J. Toyoda. "Forming coalitions for breaking deadlocks" *Proceedings of the First international Conference on Multi-agent System*, pp.155–162, MIT Press, 1995.

[HY00] K. Hirayama and M. Yokoo. "The Effect of Nogood Learning in Distributed Constraint Satisfaction", in *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, 2000, pp. (169–177).

[JL94] J. Jaffar and J. Lassez. "Constraint logic programming: A survery", *Journal of Logic Programming*, vol. 19, no. 20, 1994, pp. (503-581).

[JV05] H. Jiang, J.M. Vidal. "Reducing Redundant Messages in the Asynchronous Backtracking Algorithm", in *Proceedings of the Sixth International Workshop on Distributed Constraint Reasoning*, 2005.

[KS92] H. Kautz and B. Selman. "Planning as satisfiability", in *Proceedings of the Tenth European Conference on Artificial Intelligence*, 1992, pp. (360-363).

[Ku92] V. Kumar. "Algorithms for Constraint Satisfaction Problems: A survey", *Artificial Intelligence*, vol. 13, no. 1, 1992, pp. (32-44).

[Lam78] L. Lamport. "Time, clocks and the ordering of events in a distributed system"*, Communications of the ACM,* vol. 21, no. 7, 1978, pp. (558-565).

[Mac77] A.K. Mackworth. "Consistency in networks of relations", *Artificial Inteligence*, vol. 8, 1977, pp. (90-118).

[MASNet-a] MAS Netlogo Models-a. Available: http://jmvidal.cse.sc.edu/netlogomas/.

[MASNet-b] MAS Netlogo Models-b. Available: http://ccl.northwestern.edu/netlogo/models/community

[MF85] A.K. Mackworth and E.C. Freuder. "The complexity of some polynomial network consistency algoritms for constraint satisfaction problems", *Artificial Inteligence*, vol. 25, 1995, pp. (65-74).

[MJ00] P. Meseguer and M. A. Jimenez. "Distributed Forward Checking", in *Proceedings of the CP'00 Workshop on Distributed Constraint Satisfaction* Problems, Singapore, Thailand, 2000.

[MJPL90] S. Minton, M.D. Johnston, A.B. Philips and P. Laird. "Solving large scale constraint satisfaction and scheduling problems using heuristic repair method", in *Proceedings of the Eighth National Conference on Artificial Inteligence (AAAI-90)*, Boston 1990, pp. (17-24).

[MJPL92] S. Minton, M.D. Johnston, A.B. Philips and P. Laird. "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems", *Artificial Intelligence*, vol. 58, no. 1-3, 1992, pp. (161-205).

[MKRZ02] M. Meisels, E. Kaplansky, I. Razgon and R. Zivan. "Comparing Performance of Distributed Constraints Processing Algorithms", in *Proceedings of the. Workshop on Distributed Constraint Reasoning, AAMAS-2002*, Bologna, Italy, 2002, pp. (86-93).

[ML98] D.L. Mammen and V. R. Lesser. "Problem structure and subproblem sharing in multi-agent systems", in *Proceedings of the Third International Conference on Multi-Agent Systems*, 1998, pp. (174-181).

[Mor93] P. Morris. "The breakout method for escaping form local minima", in *Proceedings of the Eleventh National Conference on Artificial Inteligence*, 1993, pp. (40-45).

[Mus05] I. Muscalagiu. "The Effect of Flag Introduction on the Explosion of Nogood Values in the Case of ABT Family Techniques", in *Proceedings of the 4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'05), Lecture Notes in Computer Science*, Vol.3690, Spring–Verlang, 2005, pp. (286–295).

[MusB05] I. Muscalagiu and M. Bălan. "The Effect of Flag Introduction on the Explosion of Nogood Values when using The Asynchronous Backtracking Technique", in *Proceedings of the 15th International Conference on Control Systems and Computer Science (CCSCS15)*, Bucuresti, Romania, 2005, pp. (604-610).

[MusBP05] I. Muscalagiu, M. Balan and M., Panoiu. "Detection Techniques for the termination of Asynchronous Algorithms in Netlogo", in *Proceedings of the Second Balkan Conference in Informatics (BCI 2005)*, Muntenegru, 2005, pp. (389-396).

[MusCP06] I. Muscalagiu, V. Cretu, H.E. Popa. "Message management in the case of AWCS family techniques", in *Proceedings of the 7th International Conference on Technical Informatics (CONTI 2006)*, Timişoara, Romania, 2006, pp. (215- 220).

[MusCPP06] I. Muscalagiu, V. Cretu, M. Pănoiu, C. Pănoiu. "The Experimental Analysis of the Impact of the Nogood Processor Technique on the Efficiency of the Asynchronous Techniques", in *Proceedings of ICCCC2006*, Băile Felix - Oradea, Romania, 2006, Vol. 2, pp. (326-331).

[MusJP06] I. Muscalagiu, H. Jiang, H.E. Popa. "Implementation and evaluation model for the asynchronous techniques: from a synchronously distributed system to a asynchronous distributed system", in *Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2006)*, Timisoara, Romania, *IEEE Computer Society Press*, 2006, pp. (209-216).

[MusMPI04] I. Muscalagiu, D. Muscalagiu, M. Panoiu and A. Iordan. "Educational Software Meant for the Implementation And Evaluation of the Asynchronous Techniques of the ABT Family, created In NetLogo", in *Annals of The Faculty of Engineering Hunedoara*, Tom II, Fascicula 2, 2004, pp.(193-200).

[MusPO05] I. Muscalagiu, M. Panoiu and Osaci. "The Redundance of Messages in the Nogood Learning Techniques Case", in *Proceedings of the 15th International Conference on Control Systems and Computer Science (CCSCS15)*, Bucharest, Romania, 2005, pp. (611-616).

[MusPP05a] I. Muscalagiu, H.E. Popa and M. Panoiu. "Determining the number of messages transmitted for the temporary links in the case of ABT Family Techniques", in *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005)*, Timisoara, Romania, *IEEE Computer Society Press*, 2005, pp. (215-222).

[MusPP05b] I. Muscalagiu, H.E. Popa and M. Panoiu. "The Behaviour of Asynchronous Techniques Regarding the Occururence of Delays in the Sending of Messages: An empirical study", in *Proceedings of the Second Balkan Conference in Informatics (BCI 2005)*, Muntenegru, 2005, pp. (379-388).

[MusPP06] I. Muscalagiu, H.E. Popa, M. Panoiu. "Asynchronous Backtracking with temporary and fixed links: A New Hybrid Member in the ABT Family", *Journal of Computer Science INFOCOMP*, Vol. 5, no 2, 2006, pp. (29-37).

[MusV06] I. Muscalagiu, and V. Cretu. "Improving the Performances of Asynchronous Algorithms by Combining the Nogood Processors with the Nogood Learning Techniques", *Journal "INFORMATICA" Lithuania,* Vol. 17, Nr 1, 2006, pp (39-54).

[MusVCPP07] I. Muscalagiu, J. Vidal, V. Cretu, H.E. Popa, M. Panoiu. "The Efects of Agent Synchronization in Asynchronous Search Algorithms", in *Proceedings of the 1st KES Symposium on Agent and Multi-Agent Systems –*

Technologies and Applications (KES AMSTA 2007), *Lecture Notes in Artificial Intelligence*, Vol. 4496, Spring–Verlang, 2007, pp. (53–62).

[Nad89] B. Nadel. "Constraint satisfaction algorihms", *Computational Intelligence*, vol. 5, 1989, pp. (188-224).

[NetLogo3.0] Available: http://ccl.northwestern.edu/netlogo/.

[Pet06] A. Petcu. "Frodo: a framework for Open/Distributed Optimization", Technical Report EPFL:2006/001, EPFL, CH-1015 Lausanne, Available: http://liawww.epfl.ch/ frodo/, 2006.

[Pros93a] P. Prosser. "Forward checking with backmarking". Technical Report AISL–48-93, University of Strathclyde, 1993 .

[Pros93b] P. Prosser. "Hybrid algorithms for constraint satisfaction problems", *Computational Intelligence*, vol. 9, no. 3, 1993, pp. (268-299).

[Pur83] P.W. Purdom. "Search rearrangement backtracking and polynomial average time", *Artificial Intelligence*, vol. 21, 1983, pp. (117-133).

[RR98] E.T. Richards and B Richards. "Non-systematic Search and Learning: An empirical study", in *Proceedings of the Principles and Practice of Constraint Programming–CP98*, Lecture Notes in Computer Science, Vol.1520. Springer-Verlag, 1998, pp. 370–384.

[SGM96] G. Solotorevsky, E. Gudes and A. Meisels. "Modeling and solving distributed constraint satisfaction problems (dcsps)", in *Proceedings of Constraint Processing-96*, New Hamphshire, October 1996.

[SHF00] M.C. Silaghi, D. Sam-Haroud and B. Faltings. "Asynchronous Search with Aggregations", in *Proceedings of the 17th National Conference on Artificial Intelligence- AAAI'00,* Austin, Texas, 2000, pp. (917-922).

[Sil02] M.C. Silaghi. "Asynchronously Solving Distributed Problems With Privacy Requirements", *PhD theisis*, Lausanne , EPFL Elvetia, 2002.

[SLM92] B. Selman, H. Levesque and D. Mithcell. "A new method for solving hard satisfiability problems", in *Proceedings of the European Conference on Artificial Inteligence (AAAI-92)*, 1992, pp. (440-444).

[SS77] R.M. Stallman, G.S. Sussman. "Forward reasoning and dependency-directed backtracking in a system for computer–aided circuit analysis", *Artificial Inteligence*, vol. 9, 1977, pp. (135-196).

[Tsa93] E.Tsang. "Foundation of Constraint Satisfaction". *Academic Press*, 1993.

[VHDT92] P. Van Hentenryck, Y. Deville and C.M. Teng. "A generic arc consistency algorithm and its specializations", *Artificial Inteligence,* vol. 57, 1992, pp. (291-321).

[Wall95] M. Wallace. "Practical Applications of Constraint Programming", in *Proceedings Conference on Practical Applications of Constraints Technology*, Paris, 1995.

[Walt75] D. Waltz. "Understanding line drawings of scene with shadows", *The Psychology of Computer Vision*, McGraw-Hill, 1975, pp. (19-91).

[Wil99] U. Wilensky. "NetLogo itself". Available: NetLogo: http://ccl.northwestern.edu/netlogo/, *Center for Connected Learning and Computer-Based Modeling*, Northwestern University. Evanston, IL, 1999.

[YDIK92] M. Yokoo, E. H. Durfee, T. Ishida and K. Kuwabara. "Distributed constraint satisfaction for formalizing distributed problem solving", in.*Proceedings of the twelfth IEEE International Conference on Distributed Computing Systems*, 1992, pp. (614-621).

[YDIK98] M. Yokoo, E.H. Durfee, T. Ishida and K. Kuwabara. "The distributed constraint satisfaction problem: formalization and algorithms", *IEEE Transactions on Knowledge and Data Engineering*, vol. 10, no. 5, 1998, pp. (673-685).

[Yok00] M. Yokoo. "Algorithms for distributed constraint satisfaction problems: A review", *Autonomous Agents & Multi-Agent System*, vol. 3, no. 2, 2000, pp.(198-212).

[Yok00p] M. Yokoo. Private communication, 2000.

[Yok01] M. Yokoo. "Distributed Constraint Satisfaction- Foundation of Cooperation in Multi-agent Systems", in *Springer Verlang*, 2001.

[ZM04] R. Zivan and A. Meisels. "Concurrent Backtrack Search on DisCSPs", in *Proceedings of the FLAIRS-04*, Miami Beach, May, 2004.

[ZM05] R. Zivan and A. Meisels. "Dynamic ordering for asynchronous backtracking on DisCSPs", in *Proceedings of the CP2005*, Barcelona, 2005, pp. (32-46).

14. LISTA LUCRĂRILOR PUBLICATE

[Mus05]	I. Muscalagiu. "The Effect of Flag Introduction on the Explosion of Nogood Values in the Case of ABT Family Techniques", in *Proceedings of the 4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'05),* Lecture Notes in Computer Science, Vol.3690*,* Spring–Verlang, 2005, pp. (286–295).	**ISI**, INSPEC COMPEDEX, DBLP, SCOPUS
[MusB05]	I. Muscalagiu and M. Bălan. "The Effect of Flag Introduction on the Explosion of Nogood Values when using The Asynchronous Backtracking Technique", in *Proceedings of the 15th International Conference on Control Systems and Computer Science (CCSCS15)*, Bucharest, Romania, 2005, pp. (604-610).	
[MusBP05]	I. Muscalagiu, M. Balan and M., Panoiu. "Detection Techniques for the termination of Asynchronous Algorithms in Netlogo", in *Proceedings of the Second Balkan Conference in Informatics (BCI 2005)*, Muntenegru, 2005, pp. (389-396).	
[MusCP06]	I. Muscalagiu, V. Cretu, H.E. Popa. "Message management in the case of AWCS family techniques", in *Proceedings of the 7th International Conference on Technical Informatics*, Timişoara, Romania, 2006, pp. (215- 220).	
[MusCPP06]	I. Muscalagiu, V. Cretu, M. Pănoiu, C. Pănoiu. "The Experimental Analysis of the Impact of the Nogood Processor Technique on the Efficiency of the Asynchronous Techniques", in *Proceedings of ICCCC2006*, Băile Felix - Oradea, Romania, 2006, Vol. 2, pp. (326-331).	
[MusJP06]	I. Muscalagiu, H. Jiang, H.E. Popa. "Implementation and evaluation model for the asynchronous techniques: from a synchronously distributed system to a asynchronous distributed system", in *Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2006)*, Timisoara, Romania, *IEEE Computer Society Press*, 2006, pp. (209-216).	**IEE** INSPEC, COMPEDEX SCI -**ISI Proceeding**, DBLP
[MusMPI04]	I. Muscalagiu, D. Muscalagiu, M. Panoiu and A. Iordan. "Educational Software Meant for the Implementation And Evaluation of the Asynchronous Techniques of the ABT Family,	

	created In NetLogo", in *Annals of The Faculty of Engineering Hunedoara*, Tomul II, Fascicula 2, 2004, pp.(193-200).	
[MusPO05]	I. Muscalagiu, M. Panoiu and Osaci. "The Redundance of Messages in the Nogood Learning Techniques Case", in *Proceedings of the 15th International Conference on Control Systems and Computer Science (CCSCS15)*, Bucharest, Romania, 2005, pp. (611-616).	
[MusPP05a]	I. Muscalagiu, H.E. Popa and M. Panoiu. "Determining the number of messages transmitted for the temporary links in the case of ABT Family Techniques", in *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005)*, Timisoara, Romania, *IEEE Computer Society Press*, 2005, pp. (215-222).	**IEE** INSPEC, COMPEDEX SCI -**ISI Proceeding**, DBLP, SCOPUS
[MusPP05b]	I. Muscalagiu, H.E. Popa and M. Panoiu. "The Behaviour of Asynchronous Techniques Regarding the Occururence of Delays in the Sending of Messages: An empirical study", in *Proceedings of the Second Balkan Conference in Informatics (BCI 2005)*, Muntenegru, 2005, pp. (379-388).	
[MusPP06]	I. Muscalagiu, H.E. Popa, M. Panoiu. "Asynchronous Backtracking with temporary and fixed links: A New Hybrid Member in the ABT Family", *Journal of Computer Science INFOCOMP*, Vol. 5, no 2, 2006, pp. (29-37).	IEE INSPEC, IS, DEST, EBSCO
[MusV06]	I. Muscalagiu, and V. Cretu. "Improving the Performances of Asynchronous Algorithms by Combining the Nogood Processors with the Nogood Learning Techniques", *Journal "INFORMATICA" Lithuania,* Vol. 17, Nr 1, 2006, pp (39-54).	**ISI**, INSPEC, COMPEDEX, DBLP, SCOPUS
[MusVCPP07]	I. Muscalagiu, J. Vidal, V. Cretu, H.E. Popa, M. Panoiu. "The Efects of Agent Synchronization in Asynchronous Search Algorithms", in *Proceedings of the 1st KES Symposium on Agent and Multi-Agent Systems – Technologies and Applications (KES AMSTA 2007)*, Lecture Notes in Artificial Intelligence, Vol. 4496, Spring–Verlang, 2007, pp. (53–62).	**ISI Proceeding** INSPEC COMPEDEX, DBLP, SCOPUS
[MASNet-a]	MAS Netlogo Models-a. http://jmvidal.cse.sc.edu/netlogomas/.	
[MASNet-b]	MAS Netlogo Models-b. http://ccl.northwestern.edu/netlogo/models/ community	
[MASNet-c]	MAS Netlogo Models-c. http://discsp-netlogo.fih.upt.ro/ .	