FLORENTINA HRISTEA MARIA FLORINA BALCAN

CĂUTAREA ŞI REPREZENTAREA CUNOŞTINŢELOR ÎN INTELIGENŢA ARTIFICIALĂ. TEORIE ŞI APLICAŢII

editura universității din București
2010

„editura universității din București"
Șos. Panduri, 90-92, București – 050663; Telefon/Fax: 021.410.23.84
E-mail: editura_unibuc@yahoo.com
Internet: www.editura.unibuc.ro

Tiparul s-a executat sub c-da nr. 2576/2010 continuare de tiraj, la Tipografia Editurii Universității din București

Biblioteca de Matematica
1100 023 8811

Coperta: arhitect Ana-Maria Popescu

Descrierea CIP a Bibliotecii Naționale a României
HRISTEA, FLORENTINA
Căutarea și reprezentarea cunoștințelor în inteligența artificială. Teorie și aplicații / Florentina Hristea, Maria Florina Balcan – București: Editura Universității din București, 2005
Bibliografie
ISBN 973-575-996-9

I. Balcan, Maria Florentina

004.8(075.8)

CUVÂNT ÎNAINTE

Lucrarea de față abordează două dintre temele esențiale ale inteligenței artificiale, strâns legate între ele, și anume: *tehnicile de căutare* și *reprezentarea cunoștințelor*.

Una dintre cele mai importante concluzii la care s-a ajuns, după scurgerea primelor trei decade de activitate în domeniul inteligenței artificiale, este aceea că inteligența presupune cunoaștere. Această concepție va influența cel mai mult tehnicile specifice domeniului, care, în marea lor majoritate, folosesc pentru implementarea inteligenței cunoștințe reprezentate în mod explicit și algoritmi de căutare. Până la urmă, se poate spune că o tehnică a inteligenței artificiale este o metodă de exploatare și de valorificare a cunoștințelor, care, la rândul lor, trebuie reprezentate într-un anumit mod, și anume conform unor cerințe puse în evidență pe parcursul acestei lucrări.

Structurată în șapte capitole și două anexe, prezenta carte urmărește aspectele de bază referitoare la căutare (ca tehnică de rezolvare a problemelor ce explorează în mod sistematic un spațiu de stări ale problemei) și la reprezentarea cunoștințelor.

Sunt tratate în detaliu principalele tehnici de căutare prin prisma aplicării acestora în inteligența artificială. Mai întâi se prezintă structurile de date specifice (arborii de căutare) după care se discută, pe larg, algoritmii de căutare *neinformată* și *informată*, precum și implementarea acestora în limbajul Prolog. Se acordă o atenție specială analizei proprietăților algoritmilor de căutare și sunt prezentate exemple de aplicări ale acestora (A se vedea Cap. 2 al lucrării).

Departe de a pretinde că epuizează aspectele legate de problematica reprezentării cunoștințelor, lucrarea se oprește, în egală măsură, asupra câtorva dintre cele mai importante sisteme de reprezentare a cunoștințelor, și anume: *sistemele bazate pe reguli*, *structurile de tip slot-and-filler*, *rețelele semantice* și *cadrele* (A se vedea Cap. 4). Cap. 6, intitulat "Raționament statistic", își propune să întregească imaginea referitoare la tehnicile de reprezentare a cunoștințelor, ocupându-se de două tipuri de abordări statistice pentru sisteme de raționament incert (*factorii de certitudine în sistemele bazate pe reguli* și, respectiv, *rețelele Bayesiene*).

Dincolo de prezentarea tehnicilor de căutare și a sistemelor de reprezentare a cunoștințelor clasice, lucrarea își propune tratarea unor aplicații, de naturi complet diferite, ale acestora. Aplicațiile practice discutate aici sunt de o mare diversitate, începând cu cele clasice și aducând discuția în planul celor foarte noi și actuale, dintre care unele (A se vedea Cap. 5) pot constitui subiecte

de reflecţie şi eventuale teme de cercetare cu vaste posibilităţi de aprofundare, extindere şi generalizare.

Exemplul clasic de problemă de căutare prezentat este acela al *jocurilor*. Este însă discutată şi o aplicaţie a căutării în unul dintre cele mai noi subdomenii ale inteligenţei artificiale, şi anume procesarea limbajului natural. Astfel, ultimul capitol se ocupă de *analiza sintactică bazată pe constituenţi*, prezentată ca problemă de căutare.

Principala aplicaţie fundamentală a problematicii reprezentării cunoştinţelor în inteligenţa artificială, la care se referă lucrarea, este aceea a *sistemelor expert* şi a implementării acestora în limbajul Prolog. Sunt tratate pe larg problemele reprezentării cunoştinţelor pe bază de reguli, aşa cum intervin acestea, cel mai adesea, în sistemele expert.

O specială atenţie este acordată, de asemenea, *reţelelor semantice*. Este prezentat un exemplu de reţea semantică extrem de folosită în aplicaţiile inteligenţei artificiale, în general, şi - cu precădere - în subdomeniul procesării limbajului natural. Este vorba de reţeaua semantică **WordNet**, datorită existenţei căreia *procesarea cunoştinţelor* a dobândit astăzi noi dimensiuni. Ei îi este dedicat, în întregime, Cap. 5 al lucrării, care include principalele rezultate originale obţinute de una dintre autoare într-o serie de articole ştiinţifice relativ recente [11], [12], [13], precum şi în timpul stagiului de cercetare ca bursier Fulbright la Universitatea Princeton din S.U.A., Laboratorul de Ştiinţe Cognitive (2004).

Lucrarea se opreşte, prin urmare, asupra celor mai importante tehnici de căutare şi sisteme de reprezentare a cunoştinţelor, remarcându-se printr-o mare diversitate şi permiţând cititorului să-şi formeze o imagine relativ completă asupra problematicii abordate, în special prin varietatea tipurilor de aplicaţii prezentate.

Nu în ultimul rând, considerăm că lucrarea este utilă şi prin numeroasele exemple de programe pe care le oferă şi a căror existenţă este apreciată de noi ca fiind indispensabilă în vederea unei înţelegeri aprofundate a problematicii abordate.

Cartea se adresează, în egală măsură, atât studenţilor de la secţiile "informatică" şi "matematică-informatică" ale diverselor universităţi, cât şi specialiştilor şi tuturor celor care lucrează în domeniul inteligenţei artificiale sau sunt preocupaţi de problematica acestui domeniu.

Nădăjduim că lucrarea noastră îi va ajuta pe cititori să se apropie cu interes de un domeniu atât de vast şi, în acelaşi timp, de fascinant, cum este cel al inteligenţei artificiale.

Autoarele

CAPITOLUL 1
CONSIDERAŢII PRELIMINARE

Inteligenţa artificială poate fi privită ca fiind o ştiinţă sui-generis al cărei obiect de studiu îl constituie modul de programare a calculatoarelor în vederea săvârşirii unor operaţiuni pe care, deocamdată, oamenii le efectuează mai bine. Conform acestei definiţii primare, simplificate (şi care nu este general acceptată) putem gândi inteligenţa artificială ca fiind acel domeniu al informaticii care se ocupă de ***automatizarea comportamentului inteligent***.

O asemenea definiţie[1] priveşte inteligenţa artificială ca fiind un domeniu al informaticii şi, prin urmare, se bazează pe principii teoretice şi practice ale acesteia, cum ar fi:

- *structurile de date* folosite în reprezentarea cunoştinţelor;
- *algoritmii* necesari pentru aplicarea cunoştinţelor;
- *limbajele* şi *tehnicile de programare* folosite la implementarea acestor algoritmi.

În acelaşi timp, aşa cum se arată în [27], definiţia pe care o comentăm aici este deficitară prin faptul că însuşi conceptul de *inteligenţă* nu este foarte bine înţeles şi precizat. Astfel, problema definirii domeniului inteligenţei artificiale devine, până la urmă, una a definirii conceptului însuşi de inteligenţă. Ea presupune ridicarea unor întrebări de tipul: Ce se întâmplă atunci când intervine învăţarea? Ce este creativitatea? Ce este intuiţia? Cum sunt reprezentate cunoştinţele în ţesutul nervos al unei fiinţe vii? Este necesar ca un program de calculator inteligent să fie modelat în conformitate cu ceea ce se cunoaşte despre inteligenţa umană sau este suficientă o abordare strict inginerească a problemei? Este posibil să se achiziţioneze inteligenţă pe un calculator sau o entitate inteligentă necesită bogăţia senzaţiilor şi a experienţei care nu pot fi găsite decât în existenţa biologică? Toate aceste întrebări arată legătura inteligenţei artificiale cu alte domenii, cum ar fi, în primul rând, filozofia. În acelaşi timp, toate aceste întrebări deschise, care nu au încă răspuns, au ajutat la conturarea *problemelor* şi a *soluţiilor*, a *metodologiilor* care constituie miezul inteligenţei artificiale moderne. Inteligenţa artificială, o disciplină încă tânără (prin comparaţie cu altele, cum ar fi matematica sau fizica) ale cărei structură, scopuri, preocupări şi metode sunt mai puţin clar definite decât cele ale unor discipline mult mai mature, pune la dispoziţie instrumente unice şi puternice necesare tocmai în explorarea unor asemenea întrebări.

---

[1] Vezi şi [21], s.v. *artificial intelligence*, p. 37.

1.1. Domeniul inteligenței artificiale. Definiții și un scurt istoric

Domeniul inteligenței artificiale își propune *înțelegerea entităților inteligente*. Dar, spre deosebire de filozofie și psihologie, care își propun, mai mult sau mai puțin, același lucru, inteligența artificială urmărește, în plus, ***construirea*** unor asemenea entități. Ea are drept scop apariția unor calculatoare care să aibă o inteligență de nivel uman sau chiar mai bună.

Așa cum se arată în [27], există mai multe definiții ale domeniului. Acestea variază de-a lungul a două mari dimensiuni. Prima dimensiune este aceea a ***procesului de gândire și a raționamentului***. Cea de-a doua adresează ***comportamentul*** ("behavior"). De asemenea, unele definiții măsoară ***succesul*** în termenii *performanței umane*, în timp ce altele îl măsoară relativ la un concept ideal de inteligență, pe care îl vom numi *rațiune*. Iată câteva definiții ale domeniului inteligenței artificiale, date conform acestor clasificări și preluate de noi din [27]:

- definiții care se concentrează asupra procesului de gândire și a raționamentului și care măsoară succesul în termenii performanței umane – "Automatizarea activităților pe care le asociem cu gândirea umană, activități cum ar fi luarea deciziilor, rezolvarea problemelor, învățarea..." (Bellman, 1978);
- definiții care adresează comportamentul și care măsoară succesul în termenii performanței umane – "Arta de a crea mașini care îndeplinesc funcții ce necesită inteligență atunci când sunt îndeplinite de către oameni" (Kurzweil, 1990);
- definiții care se concentrează asupra procesului de gândire și a raționamentului și care măsoară succesul în termenii unui concept ideal de inteligență, anume acela al rațiunii – "Studiul facultăților mintale prin intermediul modelelor computaționale" (Charniak și McDermott, 1985); "Studiul calculelor care fac posibile percepția, raționamentul și acțiunea" (Winston, 1992);
- definiții care adresează comportamentul și care măsoară succesul în termenii rațiunii – "Un domeniu de studiu care caută să explice și să emuleze comportamentul inteligent în termenii unor procese computaționale" (Schalkoff, 1990); "Acea ramură a informaticii care se ocupă de automatizarea comportamentului inteligent" (Luger și Stubblefield, 1993).

Aceste definiții pot fi grupate în patru mari categorii (care indică și patru țeluri majore urmărite în inteligența artificială):

- sisteme care gândesc ca și ființele umane;
- sisteme care se comportă ca și ființele umane;
- sisteme care gândesc rațional;

- sisteme care se comportă rațional.

În același timp, această varietate de definiții ne face să tragem cel puțin două concluzii clare, și anume: cercetători diferiți gândesc în mod diferit despre inteligența artificială, care apare ca o știință inter-disciplinară, în care își aduc contribuția filozofi, psihologi, lingviști, matematicieni, informaticieni și ingineri.

Prima lucrare recunoscută astăzi ca fiind de inteligență artificială aparține lui Warren McCulloch și Walter Pitts (1943). Aceștia s-au bazat pe trei surse și au conceput un model de neuroni artificiali. Cele trei surse utilizate au fost: cunoștințele despre fiziologia și funcțiile de bază ale neuronilor în creier; analiza formală a logicii propoziționale datorate lui Russel și Whitehead; teoria despre calcul a lui Turing. O altă figură extrem de influentă în inteligența artificială este cea a lui John McCarthy, de la Princeton. După absolvire, McCarthy se mută la Dartmouth College, care va deveni locul oficial în care s-a născut domeniul. De altfel, în timpul workshop-ului organizat la Dartmouth, în vara anului 1956, se hotărăște și adoptarea, pentru noul domeniu, a numelui propus de McCarthy: cel de "inteligență artificială".

Un an istoric în evoluția domeniului este 1958, an în care McCarthy se mută de la Dartmouth la MIT ("Massachusetts Institute of Technology"). Aici el aduce trei contribuții vitale, toate în același an: 1958. McCarthy definește acum limbajul de nivel înalt LISP, care urma să devină limbajul de programare dominant în inteligența artificială și, împreună cu colegii de la MIT, introduce conceptul de *partajare* ("time-sharing"). Tot în 1958 el publică articolul intitulat *Programs with Common Sense* ("Programe cu bun simț") în care descrie un program ipotetic, numit "Advice Taker", care poate fi privit ca reprezentând *primul sistem complet de inteligență artificială*. Programul era proiectat să folosească ***cunoștințe*** pentru a căuta soluții la probleme. Dar, spre deosebire de programele de până atunci, acesta încorpora *cunoștințe generale despre lume*. (Spre exemplu, un grup de axiome simple dădeau programului posibilitatea să genereze un plan pentru a se merge cu mașina la aeroport și a se prinde un avion). Programul era conceput în așa fel încât să poată accepta noi axiome pe parcurs. Prin urmare, i se permitea să dobândească competență în noi domenii, fără a fi reprogramat. Acest program încorpora *principiile de bază ale reprezentării cunoștințelor și ale raționamentului*, și anume:

- este necesar să dispunem de o reprezentare formală a lumii și a felului în care acțiunile unui agent afectează lumea;
- este necesar să putem manipula aceste reprezentări cu ajutorul unor procese deductive.

O mare parte a acestui articol rămâne relevantă și astăzi. Începând din 1958, McCarthy, împreună cu colegul său de la MIT Marvin Minsky, au petrecut ani întregi numai în încercarea de ***a defini*** domeniul inteligenței artificiale.

În timpul primului deceniu de cercetări în inteligența artificială concepția legată de rezolvarea problemelor era aceea a unui mecanism de căutare cu scop general, care încerca să lege laolaltă pași elementari de raționament pentru a găsi soluții complete. Astfel de abordări au fost numite *metode slabe*, întrucât ele folosesc "informații slabe" asupra domeniului. S-a arătat că, pentru domenii complexe, performanța acestora este slabă. Dar discuțiile de început asupra domeniului s-au concentrat mai degrabă asupra *reprezentării problemelor* decât asupra *reprezentării cunoștințelor*. Accentul s-a pus pe formularea problemei care trebuie rezolvată și nu pe formularea resurselor care sunt disponibile programului. Cercetătorii și-au dat ulterior seama că sistemele de inteligență artificială aveau nevoie de cunoștințe din categoria celor din urmă, după efectuarea a două tipuri de cercetări, și anume: ***înțelegerea limbajului natural uman*** și proiectarea așa-numitelor ***sisteme expert*** (cunoscute și sub denumirea de *sisteme bazate pe cunoștințe*), adică a unor sisteme care puteau egala și, în unele cazuri (sarcini de mai mică anvergură și precis definite), depăși performanțele experților umani.

În perioada 1969-1979 apar și se dezvoltă *sistemele expert*, asupra cărora ne vom opri și în lucrarea de față. Caracteristica majoră a sistemelor expert este aceea că ele se bazează pe cunoștințele unui expert uman în domeniul care este studiat; mai exact, pe cunoștințele expertului uman asupra strategiilor de rezolvare a problemelor tipice unui domeniu. Astfel, la baza sistemelor expert se află utilizarea în rezolvarea problemelor a unor mari cantități de cunoștințe specifice domeniului. Tocmai de aceea *reprezentarea cunoștințelor* este atât de importantă în inteligența artificială.

Tot în perioada 1969-1979, numărul mare de aplicații referitoare la problemele lumii reale a determinat creșterea cererii de *scheme de reprezentare a cunoștințelor*. Au fost astfel dezvoltate diferite limbaje de reprezentare. Unele dintre ele se bazau pe *logică*[2], cum este cazul limbajului Prolog[3], extrem de popular în Europa.

---

[2] Limbajele programării logice sunt *limbaje declarative*. Un limbaj de programare declarativ scutește programatorul de a mai menționa procedura exactă pe care trebuie să o execute calculatorul pentru a realiza o funcție. Programatorii folosesc limbajul pentru a descrie o mulțime de *fapte* și de *relații* astfel încât utilizatorul să poată interoga apoi sistemul pentru a obține un anumit rezultat. (Aceasta reprezintă o deosebire esențială față de un limbaj procedural, adică un limbaj care solicită programatorului să specifice procedura pe care trebuie să o urmeze calculatorul pentru a executa o funcție).

[3] Prolog reprezintă o prescurtare de la "**Pro**gramming în **log**ic". Este un limbaj conceput pentru programarea logică (deci un limbaj declarativ). A fost elaborat în cursul anilor 1970 în Europa (mai ales în Franța și Scoția), iar primul compilator de Prolog a fost creat în 1972 de către Philippe Roussel, la Universitatea din Marsilia. Prologul este un limbaj compilat, care utilizează, în locul relațiilor matematice, relații logice între mulțimi de date.

Referindu-ne la un istoric al domeniului, vom mai nota faptul că în perioada 1980-1988 în inteligența artificială începe să se lucreze la nivel industrial. Tot în această perioadă, în care inteligența artificială devine industrie, ea își propune noi țeluri ambițioase, cum ar fi acela al *înțelegerii limbajului natural*.

Începând din 1987 și până în prezent, cu precădere în ultimii ani, atât conținutul, cât și metodologia de cercetare în inteligența artificială au suferit o serie de schimbări. Astfel, se caută aplicații din lumea reală, se construiește pe teoriile existente, dar, mai ales, cercetătorii se bazează mult mai mult pe teoreme riguroase și mai puțin decât până acum pe intuiție. Spre exemplu, domeniul *recunoașterii limbajului* a ajuns să fie dominat de așa-numitele "hidden Markov models" (HMM-uri)[4]. La rândul lor, modelele grafice (care includ, printre altele, rețelele Bayesiene, discutate în Capitolul 6 al lucrării de față) au devenit extrem de importante pentru inteligența artificială. În general, putem spune că domeniul inteligenței artificiale se bazează pe teorii matematice riguroase din ce în ce mai mult.

1.2. Subdomenii ale inteligenței artificiale

Principalele **subdomenii** ale inteligenței artificiale sunt considerate a fi [27] următoarele:

- **jocurile** (bazate pe căutarea efectuată într-un spațiu de stări ale problemei);
- **raționamentul automat și demonstrarea teoremelor** (bazate pe rigoarea și generalitatea logicii matematice. Este cea mai veche ramură a inteligenței artificiale și cea care înregistrează cel mai mare succes. Cercetarea în domeniul demonstrării automate a teoremelor a dus la formalizarea algoritmilor de căutare și la dezvoltarea unor limbaje de reprezentare formală, cum ar fi calculul predicatelor și limbajul pentru programare logică Prolog).
- **sistemele expert** (care pun în evidență importanța cunoștințelor specifice unui domeniu).
- **înțelegerea limbajului natural și modelarea semantică** (Caracteristica de bază a oricărui sistem de înțelegere a limbajului natural o constituie reprezentarea sensului propozițiilor într-un anumit limbaj de reprezentare astfel încât aceasta să poată fi utilizată în prelucrări ulterioare).

---

[4] HMM-urile (*lanțurile Markov ascunse*) sunt generate de un proces de antrenare ("training") pe un mare corpus de date provenind din vorbirea reală.

- **planificarea şi robotica** (Pe scurt, planificarea presupune existenţa unui robot capabil să execute anumite acţiuni atomice, cum ar fi deplasarea într-o cameră plină cu obstacole).
- **învăţarea automată** (datorită căreia se realizează adaptarea la noi circumstanţe, precum şi detectarea şi extrapolarea unor şabloane - "patterns". Învăţarea se realizează, spre exemplu, prin intermediul aşa-numitelor *reţele neurale*[5]. O asemenea reţea reprezintă un tip de sistem de inteligenţă artificială modelat după neuronii - celulele nervoase - dintr-un sistem nervos biologic, în încercarea de a simula modul în care creierul prelucrează informaţiile, învaţă sau îşi aduce aminte[6]).

Toate aceste subdomenii ale inteligenţei artificiale au anumite trăsături în comun [27], şi anume:

- O concentrare asupra problemelor care **nu răspund la soluţii algoritmice**; din această cauză tehnica de rezolvare a problemelor specifică inteligenţei artificiale este aceea de a se baza pe o căutare euristică[7].
- Inteligenţa artificială rezolvă probleme folosind şi informaţie inexactă, care lipseşte sau care nu este complet definită şi utilizează formalisme de reprezentare ce constituie pentru programator o compensaţie faţă de aceste probleme.
- Inteligenţa artificială foloseşte raţionamente asupra trăsăturilor calitative semnificative ale unei situaţii.
- Inteligenţa artificială foloseşte, în rezolvarea problemelor, mari cantităţi de cunoştinţe specifice domeniului investigat.

Majoritatea tehnicilor din inteligenţa artificială folosesc pentru implementarea inteligenţei cunoştinţe reprezentate în mod explicit şi algoritmi de căutare. În inteligenţa artificială există *două abordări majore* complet diferite ale problematicii domeniului: cea ***simbolică*** şi cea ***conecţionistă***. Într-o abordare simbolică cunoştinţele vor fi reprezentate în mod simbolic. O abordare total diferită - cea conecţionistă - este aceea care urmăreşte să construiască programe inteligente folosind modele care fac un paralelism cu structura neuronilor din creierul uman.

Lucrarea de faţă va folosi prima dintre abordările amintite, în cadrul căreia

---

[5] Este acceptat şi termenul de *reţea neuronală*.

[6] Vezi şi [21], s.v. *neural network*, p. 348 şi s.v. *artificial neural network*, p. 37.

[7] Prin *euristică* se înţelege o metodă de dirijare sau optimizare a procesului de rezolvare a unei probleme, pe baza unor reguli derivate din experienţa, intuiţia sau inspiraţia programatorului. Euristica este de neînlocuit în cazul unor probleme deosebit de complexe, cum ar fi, spre exemplu, jocul de şah. Vezi şi [21], s.v. *heuristic*, p. 247.

- ***reprezentarea cunoştinţelor*** adresează problema reprezentării într-un limbaj formal, adică un limbaj adecvat pentru prelucrarea ulterioară de către un calculator, a întregii game de cunoştinţe necesare comportamentului inteligent;
- ***căutarea*** este o tehnică de rezolvare a problemelor care explorează în mod sistematic un spaţiu de stări ale problemei, adică de stadii succesive şi alternative în procesul de rezolvare a acesteia. (Exemple de stări ale problemei pot fi configuraţiile diferite ale tablei de şah în cadrul unui joc sau paşii intermediari într-un proces de raţionament).

1.3. Tehnici ale inteligenţei artificiale

După scurgerea primelor trei decade de activitate în domeniul inteligenţei artificiale, s-a tras concluzia că *inteligenţa presupune cunoaştere*. Aceasta este probabil cea mai importantă concluzie la care s-a ajuns şi ea influenţează cel mai mult tehnicile specifice domeniului. Aşa cum s-a mai arătat, majoritatea tehnicilor din inteligenţa artificială folosesc pentru implementarea inteligenţei cunoştinţe reprezentate în mod explicit şi algoritmi de căutare. Până la urmă, se poate spune [27] că ***o tehnică a inteligenţei artificiale*** este o metodă de exploatare şi valorificare a cunoştinţelor, care, la rândul lor, ar trebui reprezentate într-un anumit mod, şi anume conform următoarelor cerinţe [27]:

- Cunoştinţele trebuie să înglobeze *generalizări*. Cu alte cuvinte, nu este necesară reprezentarea separată a fiecărei situaţii în parte. În schimb, situaţiile care au în comun proprietăţi importante pot fi grupate laolaltă. Dacă cunoştinţele nu ar avea această proprietate, ar fi nevoie de foarte multă memorie şi de multe operaţii de actualizare. Prin urmare, o mulţime de informaţii care nu posedă această proprietate nu va reprezenta cunoştinţe, ci ***date***.
- Cunoştinţele trebuie să poată fi înţelese de către oamenii care le furnizează. Deşi, pentru majoritatea programelor, nucleul de date poate fi achiziţionat în mod automat, în multe domenii ale inteligenţei artificiale este necesar ca majoritatea cunoştinţelor pe care un program trebuie să le aibă să fie furnizate de oameni, în termeni care să poată fi înţeleşi de către aceştia.
- Cunoştinţele trebuie să poată fi uşor modificate pentru a se corecta erori şi pentru a reflecta atât schimbările din lumea înconjurătoare, cât şi schimbările din percepţia şi imaginea noastră despre aceasta.
- Cunoştinţele trebuie să poată fi folosite în foarte multe situaţii, chiar dacă uneori duc lipsă de acurateţe şi nu sunt complete.

- Cunoştinţele trebuie să poată fi folosite astfel încât să ajute la îngustarea gamei de posibilităţi care trebuie luate în consideraţie.

Legat de această problematică, vom mai nota aici doar următorul fapt: *cunoaşterea* are anumite proprietăţi, care ţin de

- Completivitate (Cunoaşterea este întotdeauna incompletă. Cunoştinţele încorporate într-un sistem software al inteligenţei artificiale referitor la o anumită aplicaţie sunt întotdeauna mult mai puţin complete decât ceea ce se cunoaşte despre aplicaţia respectivă).
- Obiectivitate (Cunoaşterea poate fi obiectivă sau subiectivă; este necesar să se facă o distincţie între fapte şi păreri. Acestea din urmă trebuie asociate agenţilor).
- Certitudine (Cunoaşterea poate fi certă sau incertă; tocmai de aceea teoria probabilităţilor joacă un rol esenţial în multe sisteme ale inteligenţei artificiale).
- Formalizare (Cunoştinţele pot fi sau nu uşor de formalizat prin intermediul unui limbaj simbolic. Oamenii operează adesea într-o manieră aproximativă, acordându-şi o marjă de eroare. Aceeaşi posibilitate ar trebui să o aibă şi sistemele inteligente, dar acestea beneficiază numai de cunoştinţele explicite care le-au fost furnizate. Agenţii umani se bazează foarte mult pe experienţă. Sistemele trebuie să beneficieze şi ele de formalizarea experienţei).

Aşa cum s-a mai amintit, comportamentul inteligent presupune cunoaştere. O ipoteză interesantă este aceea a lui Matt Ginsberg, care afirmă că *acţiunea inteligentă* poate fi redusă la *căutare*. Deşi inteligenţa în mod cert nu coincide şi nu poate fi redusă la căutare, aceasta este o ipoteză atractivă, întrucât: căutarea poate fi uşor înţeleasă, este uşor de automatizat, uşor de gestionat etc. De asemenea, cunoştinţele formalizate sunt uşor de utilizat în căutare.

Căutarea este o traversare sistematică a unui spaţiu de soluţii posibile ale unei probleme. Un spaţiu de căutare este de obicei un graf (sau, mai exact, un arbore) în care un nod desemnează o soluţie parţială, iar o muchie reprezintă un pas în construirea unei soluţii. Scopul căutării poate fi acela de

- a găsi un drum în graf de la un nod iniţial la un nod-scop (cu alte cuvinte, de la o situaţie iniţială la una finală);
- a găsi un nod-scop.

Exemplul clasic de problemă de căutare pe care îl vom prezenta aici este acela al jocurilor (a se vedea Capitolul 3 al lucrării de faţă). Este însă interesant de discutat şi alte aplicaţii ale căutării, cum ar fi analiza sintactică a unei propoziţii (a se vedea Capitolul 7), privită ca problemă esenţială de căutare în subdomeniul procesării limbajului natural.

1.4. Un exemplu de problemă tipică în inteligenţa artificială

Un exemplu de problemă tipică tratată de inteligenţa artificială, pe care o vom descrie în cele ce urmează, îl constituie problema furnizării răspunsului la întrebări ("Question Answering").

- **Problema**: Se citeşte de la tastatură text într-un limbaj natural (spre exemplu în limba engleză) şi apoi se răspunde unor întrebări referitoare la acel text, formulate tot în engleză.

- **Observaţie**: În cazul unei astfel de probleme este mai greu de precizat care este răspunsul corect. Oferim, în acest sens, următorul exemplu[8]:

Exemplul nr. 1

Presupunem că textul reprezentând input-ul este dat de unica propoziţie
"*Russia massed troops on the Czech border.*"
Corespunzător unui astfel de input este posibilă existenţa ambelor dialoguri care urmează. Aceste dialoguri au reprezentat efectiv output-ul programului numit POLITICS (Carbonell, 1980):

Dialogul nr. 1

Q: Why did Russia do this?
A: Because Russia thought that it could take political control of Czechoslovakia by sending troops.
Q: What should the United States do?
A: The United States should intervene militarily.

Dialogul nr. 2

Q: Why did Russia do this?
A: Because Russia wanted to increase its political influence over Czechoslovakia.
Q: What should the United States do?
A: The United States should denounce the Russian action in the United Nations.

---

[8]Exemplele din acest subcapitol au fost preluate din [26], iar cel de-al doilea a fost adaptat de noi pentru limba română.

În programul POLITICS răspunsurile au fost construite luând în consideraţie atât textul reprezentând input-ul, cât şi un model separat referitor la concepţiile şi acţiunile diverselor entităţi politice, inclusiv cele ale fostei Uniuni Sovietice. Între cele două dialoguri acest model a fost schimbat. Atunci când modelul se schimbă şi răspunsurile sistemului se schimbă. Astfel, primul dialog a fost produs de programul POLITICS atunci când i s-a dat acestuia un model care corespunde concepţiilor unui conservator american. Cel de-al doilea a fost produs atunci când i s-a dat un model care corespunde concepţiilor unui liberal american. Prin urmare, în acest caz este foarte greu de spus ce înseamnă un răspuns corect. Acest lucru ***depinde de model***.

O problemă din aceeaşi categorie, care utilizează în rezolvare tehnici specifice inteligenţei artificiale, este cea prezentată în

Exemplul nr.2

Fie textul de intrare:
"*Maria s-a dus să cumpere un palton nou. A găsit unul roşu care i-a plăcut mult. Când l-a adus acasă, a descoperit că merge perfect cu rochia ei preferată.*"
Dorim să stabilim un algoritm prin care programul să răspundă următoarelor întrebări:
I1: *Ce s-a dus să cumpere Maria?*
I2: *Ce a găsit Maria care i-a plăcut?*
I3: *A cumpărat Maria ceva?*

Programul nr. 1

Acest program mai întâi converteşte textul de intrare (reprezentând input-ul) într-o *formă structurată internă* care îşi propune să capteze sensul propoziţiilor. De asemenea, converteşte întrebările în acelaşi format. Programul găseşte răspunsurile făcând corespondenţa dintre cele două forme structurate.

Structuri de date folosite în program:

- **CunostinteRomana** - conţine o descriere a cuvintelor, gramaticii şi a interpretărilor semantice adecvate a unei submulţimi din limbă (în acest caz limba română), suficient de mari şi de semnificative, care să poată acoperi textele de intrare pe care sistemul urmează să le primească. Aceste *cunoştinţe* referitoare la limbă sunt folosite atât pentru a face corespondenţa dintre textele de intrare şi o *formă internă, orientată către sens*, precum şi dintre o asemenea formă şi limba recunoscută de sistem. Problema stabilirii unei asemenea *baze de cunoştinţe* pentru o limbă este foarte complicată şi de ea se ocupă subdomeniul procesării limbajului natural.
- **InputText** - reprezintă textul de intrare în format caracter.
- **TextStructurat** - este o reprezentare structurată a conţinutului textului de intrare. Această structură îşi propune să capteze cunoştinţele cele mai importante conţinute în text, independent de modul exact în care aceste cunoştinţe au fost formulate în limba respectivă. Unele lucruri care nu au fost explicitate în acea limbă (spre exemplu la cine se referă pronumele din text) vor fi explicitate în această formă. *Reprezentarea cunoştinţelor* în acest mod este o chestiune esenţială în toate programele inteligenţei artificiale. Există trei familii importante de asemenea *sisteme de reprezentare a cunoştinţelor*:
  - reguli de producţie (de forma "dacă *x*, atunci *y*");
  - structuri de tip *slot-and-filler* (multe reprezentări folosesc perechi de tipul atribut-valoare, numite *sloturi*: engl. *slots*; atributele reprezintă denumirile sloturilor: *slot names*, iar valorile sunt cele care umplu aceste sloturi: *slot fillers*; vezi, de asemenea, p. 123 şi Fig. 4.2).
  - instrucţiuni în formă matematică (de tipul celor dintr-un sistem logic).

Sistemul de reprezentare a cunoştinţelor ales de noi în acest exemplu este o structură de tip "slot-and-filler". În acest sistem, de pildă, propozitia

"*Ea a găsit unul roşu care i-a plăcut mult.*"

se poate reprezenta în felul următor:

```
Eveniment1
        instantiere:          A gasi
        timp:                 Trecut
        agent:                Maria
        obiect:               Lucru1
Lucru1
        instantiere:          Palton
        culoare:              Rosu
Eveniment2
        instantiere:          A place
        timp:                 Trecut
        element modificator:  Mult
        obiect:               Lucru1
```

Aceasta este o descriere simplificată a conţinutului propoziţiei. Ea nu este foarte explicită cu privire la relaţiile temporale (timpul a fost marcat doar ca fiind "trecut"). Una dintre ideile cheie ale unei astfel de reprezentări este aceea că sensul entităţilor din reprezentare derivă din legătura, conexiunile lor cu alte entităţi. De aceea, în reprezentare există şi alte entităţi, corespunzătoare unor

concepte pe care programul le cunoştea înainte de a citi această propoziţie - Maria, palton (conceptul general de *palton*).

- **InputIntrebare** - reprezintă întrebarea care constituie o parte a input-ului, în format caracter.
- **StructIntrebare** - este o reprezentare structurată a conţinutului întrebării puse de utilizator. (Structura este aceeaşi cu cea folosită pentru a reprezenta conţinutul textului care constituie input-ul).

Algoritmul:

Converteşte **InputText** în formă structurată utilizând cunoştinţele conţinute în **CunostinteRomana**. Această operaţie s-ar putea să presupună luarea în consideraţie a *diferite potenţiale structuri*, din variate motive, spre exemplu datorită faptului că multe cuvinte dintr-o limbă pot fi ambigue. De asemenea, anumite structuri gramaticale pot fi ambigue, în procesul comunicării pronumele pot avea diferiţi antecedenţi etc.

După convertirea textului de intrare în formă structurată, pentru a răspunde la o întrebare, se execută următorii ***paşi***:

1. Se converteşte întrebarea în formă structurată, folosind din nou cunoştinţele conţinute în structura de date **CunostinteRomana**. Se foloseşte un marcator special în cadrul structurii pentru a indica acea parte a structurii care ar trebui întoarsă ca răspuns. Acest marcator va corespunde adesea cu ocurenţa unui cuvânt care indică o întrebare în cadrul propoziţiei (cuvânt de tipul *cine* sau *ce*). Felul exact în care se efectuează această marcare depinde de forma aleasă pentru a reprezenta **TextStructurat**. Atunci când, în reprezentare, se foloseşte o structură de tipul "slot-and-filler", ca aici, un marcator special poate fi plasat în unul sau mai multe sloturi. Dacă se foloseşte un sistem logic, marcatorii vor apărea ca variabile în cadrul formulelor logice care reprezintă întrebarea.
2. Se opune această formă structurată celei din **TextStructurat**. (Se împerechează cele două forme).
3. Se întorc ca răspuns acele părţi ale textului care se potrivesc cu segmentul cerut din întrebare.

❑

Iată răspunsurile obţinute în cazul nostru, adică atunci când sistemului i se pun cele trei întrebări luate în consideraţie în *Exemplul nr. 2*:

I1: Răspunsul direct la întrebare este "*un palton nou*".
I2: Răspunsul direct la întrebare este "*un palton roşu*".

I3: La această întrebare **nu** se poate răspunde, întrucât în text nu există un răspuns *direct* la ea.

- **Observaţii:**

1. În această abordare se poate răspunde la majoritatea întrebărilor *pentru care răspunsul este conţinut în text*. Preţul plătit este timpul petrecut pentru a căuta în diversele baze de cunoştinţe (cum ar fi **CunostinteRomana**, **TextStructurat**).

2. Este necesară o abordare mai complexă, adică *mai bogată în cunoştinţe*, întrucât programul nu poate produce un răspuns la a treia întrebare. El poate răspunde numai la primele două întrebări fără a avea cunoştinţe suplimentare despre magazine sau paltoane. Prin urmare, rezultă că nu este suficientă extragerea unei reprezentări structurate a sensului textului care reprezintă input-ul. Sunt necesare cunoştinţe suplimentare, aşa cum se va vedea cu ocazia studierii celui de-al doilea program pe care îl propunem aici.

3. Problema producerii unei baze de cunoştinţe pentru o limbă (de tipul **CunostinteRomana**) este extrem de complicată şi de complexă. De rezolvarea ei se ocupă subdomeniul inteligenţei artificiale numit procesarea limbajului natural. Pe de altă parte, cunoştinţele referitoare la o limbă dată nu sunt suficiente pentru a face ca un program să construiască tipul de reprezentări structurate discutate aici. Sunt necesare cunoştinţe suplimentare despre universul cu care operează textul dat. Astfel de cunoştinţe facilitează dezambiguizarea lexicală şi sintactică, precum şi atribuirea corectă de antecedente pronumelor. Spre exemplu, în textul

"*Maria s-a îndreptat spre vânzătoare. **Ea** a întrebat unde se află raionul de jucării.*"

nu este posibil ca programul să determine la cine se referă pronumele ***ea*** fără a deţine cunoştinţe generale referitoare la rolurile clientului şi respectiv ale vânzătorului într-un magazin.

Programul nr. 2

(care exemplifică ce se înţelege prin tehnică a inteligenţei artificiale)

Acest program converteşte textul de intrare într-o formă structurată care conţine sensurile propoziţiilor din text şi apoi combină această formă cu alte forme structurate, care descriu cunoştinţe prealabile despre obiectele şi situaţiile

la care se referă textul. Programul răspunde la întrebări folosind această *structură de cunoştinţe extinsă*.

Structuri de date folosite în program:

- **ModeluLLumii** - este o reprezentare structurată a cunoştinţelor asupra lumii (universului) din fundal. Această structură conţine cunoştinţe despre obiecte, acţiuni şi situaţii care sunt descrise în textul de intrare. Ea este folosită pentru a construi **TextIntegrat** plecând de la textul de intrare. Figura care urmează ilustrează un exemplu de structură ce reprezintă cunoştinţele sistemului legate de mersul la cumpărături. Asemenea cunoştinţe memorate legate de evenimente stereotipe formează un ***script***[9]. Fig. 1.1 ne arată un script pentru cumpărături. Notaţia folosită aici este simplificată în raport cu cea din literatura existentă, de dragul simplităţii şi al înţelegerii cât mai clare a noţiunii de script. Astfel, notaţia cu "prim" descrie un obiect de acelaşi tip ca cea fără "prim", cele două simboluri (dintre care unul cu "prim") putându-se referi sau nu la obiecte identice. În exemplul nostru, de pildă, **M** este un palton, iar **M'** un palton roşu. Ramificaţiile din figură descriu drumuri alternative prin script.
- **CunostinteLimba** - la fel ca în Programul nr. 1.
- **InputText** - textul de intrare în format caracter.

---

[9] Un *script* este o structură care descrie o secvenţă stereotipă de evenimente, într-un anumit context. Astfel, scripturile îşi propun să reprezinte cunoştinţe despre succesiuni comune de evenimente (i.e. evenimente care nu intervin în mod izolat). Informaţii despre succesiuni tipice de evenimente, aşa cum sunt ele reprezentate în scripturi, pot fi folositoare în interpretarea unei succesiuni particulare de evenimente, care este observată la un moment dat.

Un script constă dintr-o mulţime de *sloturi*. Asociată fiecărui slot se găseşte o informaţie despre tipurile de valori pe care le poate conţine, precum şi o valoare implicită, care poate fi folosită atunci când nu este disponibilă nici o altă informaţie.

Scripturile sunt utile deoarece, în lumea reală, există anumite *şabloane* după care evenimentele se produc. Aceste şabloane se nasc datorită *relaţiilor cauzale* care intervin între evenimente. Evenimentele descrise într-un script formează un uriaş *lanţ cauzal*. Începutul lanţului este mulţimea condiţiilor de intrare care fac posibilă producerea primelor evenimente ale scriptului. Sfârşitul lanţului este mulţimea rezultatelor care pot face posibilă producerea unor evenimente sau a unor lanţuri de evenimente ulterioare. În cadrul lanţului, evenimentele sunt conectate atât la evenimente anterioare, care le-au făcut posibile, cât şi la evenimente ulterioare, a căror producere o înlesnesc.

Dacă se ştie că un anumit script este adecvat într-o situaţie dată, atunci el poate fi foarte util în *prezicerea* producerii unor evenimente care nu au fost menţionate în mod explicit. Scripturile sunt, de asemenea, utile în indicarea felului în care evenimente care au fost menţionate sunt legate unele de altele.

- **TextIntegrat** - o reprezentare structurată a cunoştinţelor conţinute în textul de intrare (similară cu descrierea structurată din Programul nr.1), dar care acum este combinată cu alte cunoştinţe, înrudite, din fundal.
- **InputIntrebare** - întrebarea care constituie o parte a input-ului, în format caracter.
- **StructIntrebare** - o reprezentare structurată a întrebării utilizatorului.

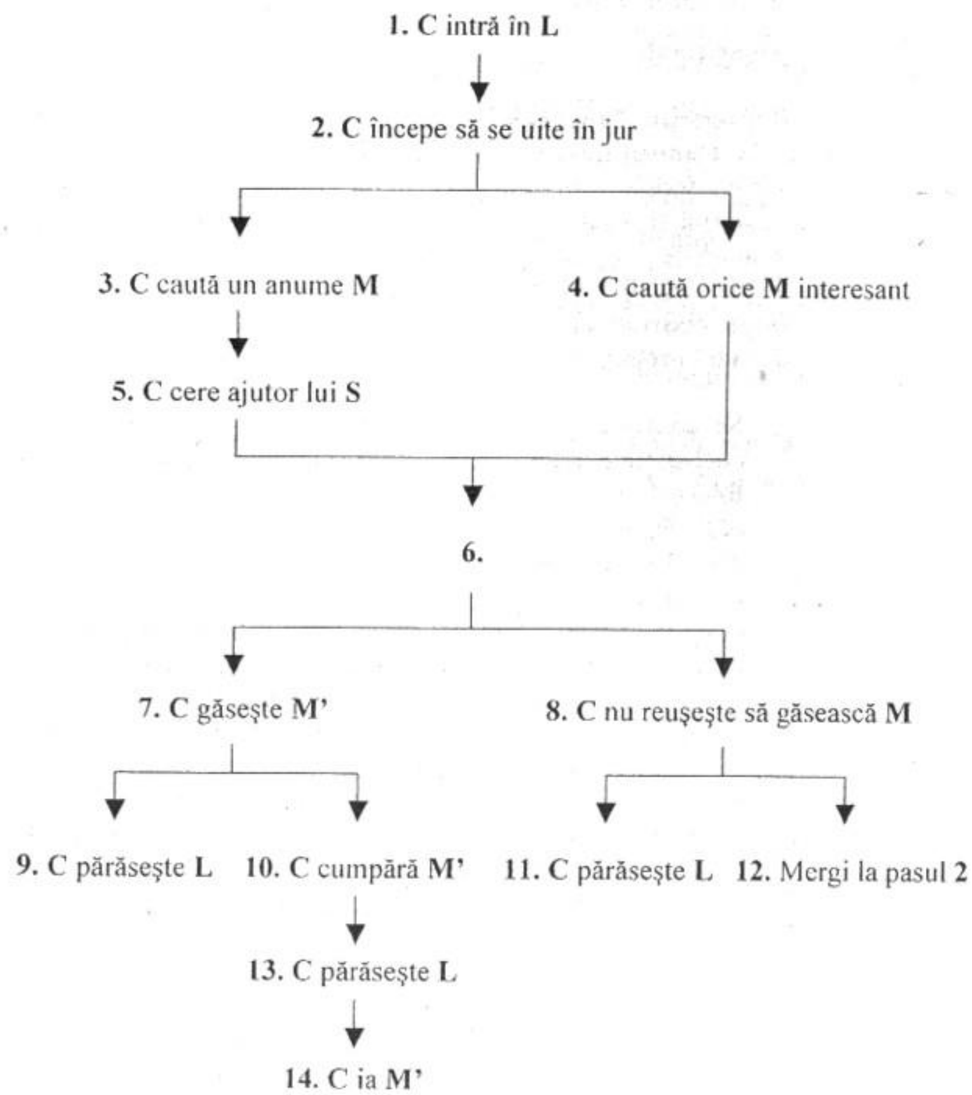


Fig. 1.1

Algoritmul:

Converteşte **InputText** în formă structurată folosind atât cunoştinţele conţinute în **CunostinteLimba**, cât şi pe cele conţinute în **ModelulLumii**. (Numărul structurilor posibile va fi mai mare acum decât în programul anterior, pentru că sunt folosite mult mai multe cunoştinţe. Unele posibilităţi vor fi însă eliminate prin filtrarea lor cu ajutorul cunoştinţelor suplimentare).

După convertirea textului de intrare în formă structurată, pentru a răspunde la o întrebare, se execută următorii ***paşi***:

1. Converteşte întrebarea în formă structurată, ca şi în Programul 1, dar foloseşte, atunci când este necesar, **ModelulLumii**, pentru a rezolva diversele ambiguităţi care pot interveni.
2. Împerechează această formă cu cea din **TextIntegrat**.
3. Întoarce ca răspuns acele părţi din text care se potrivesc cu segmentul de întrebare cerut.

❑

Iată răspunsurile obţinute acum în cazul nostru, adică atunci când sistemului i se pun cele trei întrebări luate în consideraţie în *Exemplul nr. 2*:

I1: La fel ca în Programul nr.1.

I2: La fel ca în Programul nr.1.

I3: Acum se poate răspunde la cea de-a treia întrebare. Scriptul referitor la cumpărături este instanţiat pentru acest text şi, din cauza ultimei propoziţii (*Când l-a adus acasă...*), drumul prin pasul 14 al scriptului este cel folosit în formarea reprezentării acestui text. Atunci când scriptul este instanţiat, **M'** *este limitat la structura reprezentând paltonul roşu* (deoarece scriptul ne spune că **M'** este ceea ce se duce acasă, iar textul afirmă că acasă a fost dus un palton roşu). După ce scriptul a fost instanţiat, **TextIntegrat** conţine o serie de evenimente care provin din script, dar care nu sunt descrise în textul original, inclusiv evenimentul "*Maria cumpără un palton roşu*" (de la pasul 10 al scriptului). În acest fel, folosirea textului integrat ca bază pentru obţinerea răspunsurilor la întrebări permite programului să răspundă, la cea de-a treia întrebare:

"*Ea a cumpărat un palton roşu.*"

- **Observaţii:**

1. Acest program este mai puternic decât primul, întrucât el exploatează mai multe cunoştinţe. Programul exploatează ceea ce reprezintă *o tehnică clasică a inteligenţei artificiale*.

2. Chiar şi tehnicile exploatate în acest program nu sunt adecvate pentru a răspunde tuturor întrebărilor. Cel mai important lucru care îi lipseşte acestui program este un *mecanism general de inferenţă*, care să poată fi folosit atunci când răspunsul cerut nu este conţinut în mod explicit nici chiar în **TextIntegrat**, dar decurge în mod logic din cunoştinţele incluse acolo. Spre exemplu, fiind dat textul "*Duminică dimineaţa Maria s-a dus la cumpărături. Fratele ei a încercat să o sune atunci, dar nu a putut s-o găsească*" ar trebui să se poată răspunde la întrebarea "*De ce nu a putut fratele Mariei să o găsească?*" prin răspunsul "*Pentru că ea nu era acasă*". Dar, pentru a obţine un asemenea răspuns, este necesar să se cunoască faptul că cineva nu poate fi în două locuri în acelaşi timp şi apoi să se folosească acest fapt pentru a trage concluzia că Maria nu ar fi putut fi acasă deoarece era la cumpărături.

Problema inferenţei a fost în mod temporar evitată prin construirea lui **TextIntegrat**, care avea unele inferenţe incluse în el. Dar acestea nu sunt suficiente şi, cum nu este posibil să se anticipeze toate inferenţele normale şi posibile, anumite subdomenii ale inteligenţei artificiale se ocupă de furnizarea unui *mecanism general de inferenţă*.

- **Concluzie:**

Programul nr. 2 reprezintă un exemplu pentru a susţine faptul că o procedură eficientă de "răspunsuri la întrebări" trebuie să se bazeze în mod solid pe *cunoştinţe* şi pe *exploatarea computaţională a acelor cunoştinţe*. De altfel, însuşi scopul tehnicilor inteligenţei artificiale este acela de a putea susţine o folosire eficientă a cunoştinţelor.

CAPITOLUL 2
TEHNICI DE CĂUTARE

Un **agent** este orice entitate care poate fi privită ca *percepând* mediul înconjurător prin *senzori* şi ca *acţionând* asupra acestui mediu prin *efectori*.

În cazul unui *agent uman*, acesta are anumite organe (ochi, urechi etc.) pe post de senzori şi anumite părţi ale corpului (mâini, picioare, gură etc.) pe post de efectori. Există şi agenţi roboţi, caz în care, spre exemplu, efectorii sunt înlocuiţi cu diferite motoare. În general, prin *efector* se înţelege un muşchi, o glandă etc. capabil să răspundă unui stimul, în special unui impuls nervos.

Aşa cum se afirmă în [27], menirea inteligenţei artificiale este să proiecteze un *program agent*, şi anume o funcţie care implementează un agent ce realizează transformarea de la percepţii la acţiuni.

Agenţii inteligenţi au, în mare, acelaşi schelet, ceea ce înseamnă ca acceptă percepţii din mediul înconjurător şi generează acţiuni. Cele mai facile *programe agent* au o formă foarte simplă. Fiecare astfel de program foloseşte nişte *structuri de date interne* care vor fi actualizate pe măsură ce intervin noi percepţii. Asupra acestor structuri de date se operează prin *procedurile de luare de decizii* ale agentului. Aceste proceduri generează alegerea unei acţiuni, alegere care este transmisă arhitecturii (hardware-ului) pentru a fi executată.

În cele ce urmează, vom arăta cum un agent poate acţiona prin stabilirea unor ţeluri sau *scopuri* şi vom lua în consideraţie secvenţa de acţiuni care ar putea duce la îndeplinirea acestor scopuri.

Conform [27], un scop şi o mulţime de modalităţi de atingere a acestuia formează o *problemă*, iar explorarea a ceea ce pot face aceste modalităţi de atingere a scopului constituie procesul de *căutare*.

În continuare, urmând calea din [27], ne propunem să descriem un anumit tip de *agent bazat pe scopuri*, numit *agent rezolvator de probleme*. Agenţii rezolvatori de probleme decid ce trebuie făcut prin găsirea unor secvenţe de acţiuni care conduc la nişte *stări convenabile*. Agentul trebuie să-şi formuleze o viziune adecvată asupra problemei pe care o va rezolva. Tipul de problemă care rezultă din procesul de formulare va depinde de cunoştinţele aflate la îndemâna agentului. În principal, este important dacă el cunoaşte starea curentă şi rezultatele acţiunilor.

Agenţii cu care vom lucra vor adopta un *scop* şi vor urmări *satisfacerea* lui.

2.1. Rezolvarea problemelor prin intermediul căutării

În procesul de rezolvare a problemelor, *formularea scopului*, bazată pe situaţia curentă, reprezintă primul pas.

Vom considera un *scop* ca fiind o mulţime de stări ale universului, şi anume acele stări în care scopul este satisfăcut. *Acţiunile* pot fi privite ca generând tranziţii între stări ale universului. Agentul va trebui să afle care acţiuni îl vor conduce la o stare în care scopul este satisfăcut. Înainte de a face asta el trebuie să decidă ce tipuri de acţiuni şi de stări să ia în consideraţie.

Procesul decizional cu privire la acţiunile şi stările ce trebuie luate în consideraţie reprezintă *formularea problemei*. Formularea problemei urmează după formularea scopului.

Un agent care va avea la dispoziţie mai multe opţiuni imediate va decide ce să facă examinând mai întâi diferite *secvenţe de acţiuni* posibile, care conduc la stări de valori necunoscute, urmând ca, în urma acestei examinări, să o aleagă pe cea mai bună. Procesul de examinare a unei astfel de succesiuni de acţiuni se numeşte *căutare*. Un *algoritm de căutare* primeşte ca *input* o *problemă* şi întoarce ca *output* o *soluţie* sub forma unei succesiuni de acţiuni.

Odată cu găsirea unei soluţii, acţiunile recomandate de aceasta pot fi duse la îndeplinire. Aceasta este *faza de execuţie*. Prin urmare, agentul *formulează*, *caută* şi *execută*. După formularea unui scop şi a unei probleme de rezolvat, agentul cheamă o procedură de căutare pentru a o rezolva. El foloseşte apoi soluţia pentru a-l ghida în acţiunile sale, executând ceea ce îi recomandă soluţia ca fiind următoarea acţiune de îndeplinit şi apoi înlătură acest pas din succesiunea de acţiuni. Odată ce soluţia a fost executată, agentul va găsi un nou scop.

Iată o funcţie, preluată de noi din [27], care implementează un *agent rezolvator de probleme* simplu:

```
function AGENT_REZOLVATOR_DE_PROBLEME(p) return o acţiune
        inputs: p, o percepţie
        static: s, o secvenţă de acţiuni, iniţial vidă
                stare, o descriere a stării curente a lumii (universului)
                g, un scop, iniţial vid
                problemă, o formulare a unei probleme
stare ← ACTUALIZEAZĂ_STARE(stare, p)
if s este vid then
        g ← FORMULEAZĂ_SCOP(stare)
        problemă ← FORMULEAZĂ_PROBLEMĂ(stare, g)
        s ← CAUTĂ(problemă)
        acţiune ← RECOMANDARE(s, stare)
        s ← REST(s, stare)
return acţiune
```


În cele ce urmează nu vom discuta amănunte despre funcţiile ACTUALIZEAZĂ_STARE şi FORMULEAZĂ_SCOP, ci ne vom concentra asupra *procesului de formulare a problemei* şi ne vom ocupa de diferite versiuni ale funcţiei CAUTĂ. Funcţia RECOMANDARE nu face decât să preia prima dintre acţiunile secvenţei de acţiuni, iar funcţia REST întoarce restul.

2.1.1. Tipuri de probleme

Există *patru tipuri de probleme* [27], fundamental diferite: probleme cu o unică stare, probleme cu stări multiple, probleme de contingenţă şi probleme de explorare:

1. Atunci când senzorii agentului îi furnizează acestuia suficientă informaţie pentru ca el să îşi dea seama exact în ce stare se află (spunem că universul îi este accesibil) şi atunci când el ştie în mod exact ce realizează fiecare dintre acţiunile sale, el poate calcula în mod exact în ce stare se va afla după orice secvenţă de acţiuni. Acesta este cazul cel mai simplu, numit *problemă cu o singură stare*.

2. Să presupunem că agentul cunoaşte toate efectele acţiunilor sale, dar are acces limitat la starea universului. Spre exemplu, într-un caz extrem, s-ar putea ca el să nu aibă nici un fel de senzori. În acest caz, el deţine numai o informaţie de tipul: ştie că starea sa iniţială este una din mulţimea de stări {1, 2, 3, 4, 5, 6, 7, 8}. Cu toate acestea, agentul se poate descurca destul de bine, deoarece cunoaşte efectele acţiunilor sale. Spre exemplu, poate calcula că o anumită acţiune îl va duce în una din stările {2, 4, 6, 8}. În concluzie, atunci când universul nu este în întregime accesibil, agentul trebuie să facă raţionamente referitoare la mulţimi de stări în care ar putea ajunge, nu la stări unice. Aceasta se numeşte o *problemă cu stări multiple*.

3. Rezolvarea anumitor probleme (în special în lumea fizică, reală) necesită acţiunea senzorilor chiar în faza de execuţie. Atunci când nu există o secvenţă de acţiuni fixată care să garanteze o soluţie a problemei, agentul va trebui să calculeze un întreg *arbore de acţiuni*, nu numai o unică secvenţă de acţiuni. În general, fiecare ramură a arborelui se ocupă de o posibilă *contingenţă* ce ar putea să apară. Din acest motiv, numim acest gen de probleme - *probleme de contingenţă*[10]. Multe probleme din lumea reală sunt probleme de contingenţă deoarece *predicţia exactă este imposibilă*. (Mersul pe stradă sau condusul maşinii reprezintă, în lumea reală, probleme de contingenţă). Rezolvarea problemelor de contingenţă necesită algoritmi extrem de complecşi. De aceea, în cadrul acestei lucrări, ne vom ocupa numai de problemele cu o singură stare şi cu stări multiple, care pot fi rezolvate utilizând tehnici de căutare similare.

---

[10] Aici *contingenţă* are sensul de *întâmplare*.

Vom lua deci în consideraţie numai cazuri în care *o soluţie garantată constă dintr-o unică secvenţă de acţiuni.*

4. Un agent poate să nu aibă nici o informaţie despre efectele acţiunilor sale. Aceasta este sarcina cea mai grea a unui agent inteligent, identică cu problemele pe care le au nou-născuţii. În acest caz, agentul trebuie *să experimenteze*, descoperind în mod gradual efectele acţiunilor sale, precum şi ce fel de stări există. Aceasta este tot o *căutare*, dar una *în lumea reală, nu în cadrul unui model*. O acţiune efectuată în lumea reală şi nu în cadrul unui model poate fi extrem de periculoasă pentru un agent ignorant. Dacă agentul supravieţuieşte, el *învaţă o "hartă" a mediului înconjurător*, pe care apoi o poate folosi pentru a rezolva probleme ulterioare. Aceasta este aşa-numita *problemă de explorare*.

2.1.2. Probleme şi soluţii corect definite

➢ ***Probleme cu o singură stare***

Elementele de bază ale definirii unei probleme sunt *stările* şi *acţiunile*. Pentru a descrie stările şi acţiunile, din punct de vedere formal, este nevoie de următoarele elemente [27]:

- *Starea iniţială* în care agentul ştie că se află.
- Mulţimea acţiunilor posibile disponibile agentului. Termenul de *operator* este folosit pentru a desemna descrierea unei acţiuni, prin specificarea stării în care se va ajunge ca urmare a îndeplinirii acţiunii respective, atunci când ne aflăm într-o anumită stare. (O formulare alternativă foloseşte o *funcţie succesor S*. Fiind dată o anumită stare *x*, *S(x)* întoarce mulţimea stărilor în care se poate ajunge din *x*, printr-o unică acţiune).
- *Spaţiul de stări* al unei probleme reprezintă mulţimea tuturor stărilor în care se poate ajunge plecând din starea iniţială, prin intermediul oricărei secvenţe de acţiuni.
- Un *drum* în spaţiul de stări este orice secvenţă de acţiuni care conduce de la o stare la alta.
- *Testul scop* este testul pe care un agent îl poate aplica unei singure descrieri de stare pentru a determina dacă ea este o *stare de tip scop*, adică o stare în care scopul este atins (sau realizat). Uneori există o mulţime explicită de stări scop posibile şi testul efectuat nu face decât să verifice dacă s-a ajuns în una dintre ele. Alteori, scopul este specificat printr-o proprietate abstractă şi nu prin enumerarea unei mulţimi de stări. De exemplu, în şah, scopul este să se ajungă la o stare numită "şah mat", în care regele adversarului poate fi capturat la următoarea mutare, orice ar face adversarul. S-ar putea întâmpla ca o soluţie să fie preferabilă alteia, chiar dacă amândouă ating scopul. Spre exemplu, pot fi preferate drumuri cu mai puţine acţiuni sau cu acţiuni mai puţin costisitoare.
- *Funcţia de cost a unui drum* este o funcţie care atribuie un cost unui drum. Ea este adeseori notată prin *g*. Vom considera costul unui drum ca fiind suma costurilor acţiunilor individuale care compun drumul.

Împreună *starea iniţială, mulţimea operatorilor, testul scop* şi *funcţia de cost a unui drum* definesc o ***problemă***.

Tipul de dată prin care putem reprezenta o problemă este [27] următorul:

tip_de_dată PROBLEMĂ
componente: STARE_INIŢIALĂ, OPERATORI, TEST_SCOP, FUNCŢIE_COST_DRUM

Instanţieri ale acestui tip de dată vor reprezenta *input*-ul algoritmilor de căutare. *Output*-ul unui algoritm de căutare este *soluţia*, adică un drum de la starea iniţială la o stare care satisface testul scop.

➢ ***Probleme cu stări multiple***

Pentru definirea unei astfel de probleme trebuie specificate:

- o *mulţime* de stări iniţiale;
- o mulţime de operatori care indică, în cazul fiecărei acţiuni, mulţimea stărilor în care se ajunge plecând de la orice stare dată;
- un test scop (la fel ca la problema cu o singură stare);
- funcţia de cost a unui drum (la fel ca la problema cu o singură stare).

Un operator se aplică unei mulţimi de stări prin reunirea rezultatelor aplicării operatorului fiecărei stări din mulţime. Aici un ***drum*** leagă *mulţimi de stări*, iar o ***soluţie*** este un drum care conduce la o *mulţime de stări*, dintre care *toate sunt stări scop*. Spaţiul de stări este aici înlocuit de *spaţiul mulţimii de stări*.

- **Un exemplu. Problema misionarilor şi canibalilor**

Problema este celebră în inteligenţa artificială deoarece ea a constituit subiectul primului articol care a abordat chestiunea formulării problemelor dintr-un punct de vedere analitic (Amarel, 1968).

Vom considera această problemă într-un caz particular, atunci când numărul misionarilor este acelaşi cu cel al canibalilor, şi anume egal cu trei, iar barca dispune de un număr de două locuri. În acest caz, problema se formulează astfel:

Pe malul stâng al unei ape se găsesc trei misionari şi trei canibali. Aceştia urmează să treacă apa, având la dispoziţie o barcă cu două locuri. Se ştie că, dacă pe unul dintre maluri numărul de canibali este mai mare decât numărul de misionari, atunci misionarii de pe acel mal sunt mâncaţi de canibali. Se cere determinarea unei variante de trecere a apei fără ca misionarii să fie mâncaţi de canibali, dacă aceasta există.

Pentru a formaliza problema, la primul pas ea trebuie abstractizată, prin îndepărtarea tuturor detaliilor care nu au relevanţă asupra soluţiei.

Următorul pas este acela de a decide care este mulţimea de operatori corectă. Se ştie că operatorii vor presupune transportarea a una sau doua persoane de-a lungul râului într-o barcă. Trebuie să decidem dacă este necesară o stare care să reprezinte momentul în care ei se află în barcă sau numai acele momente în care ei ajung pe partea cealaltă a râului. Deoarece barca poate transporta maximum două persoane, nu se pune problema ca în ea numărul de misionari să fie mai mic decât numărul de canibali. Prin urmare, numai capetele unei traversări sunt importante.

Pasul următor realizează o abstractizare asupra indivizilor. Din punctul de vedere al soluţiei, atunci când un canibal trebuie să se urce în barcă, nu contează care dintre ei este acesta (de exemplu cum îl cheamă). Orice permutare a celor trei misionari sau a celor trei canibali conduce la acelaşi rezultat.

Toate aceste consideraţii duc la următoarea definiţie formală a problemei:

- ***Stări***: o stare constă dintr-o secvenţă ordonată de trei numere reprezentând numărul de misionari, de canibali şi de bărci, care se află pe malul râului. Starea de pornire (iniţială) este (3,3,1).
- ***Operatori***: din fiecare stare, posibilii operatori trebuie să ia fie un misionar, fie un canibal, fie doi misionari, fie doi canibali, fie câte unul din fiecare şi să îi transporte cu barca. Prin urmare, există *cel mult cinci operatori*, deşi majorităţii stărilor le corespund mai puţini operatori, intrucât trebuie evitate stările interzise. (*Observaţie*: Dacă am fi ales să distingem între indivizi, în loc de cinci operatori ar fi existat 27).
- ***Testul scop***: să se ajungă în starea (0,0,0).
- ***Costul drumului***: este dat de numărul de traversări.

Acest spaţiu al stărilor este suficient de mic pentru ca problema să fie una trivială pentru calculator.

□

Un aspect deosebit de important îl constituie măsurarea performanţei în rezolvarea de probleme. Astfel, eficacitatea unei căutări poate fi măsurată în cel puţin trei moduri, şi anume conform următoarelor criterii de bază:

- dacă se găseşte o soluţie;

- dacă s-a găsit o soluţie bună (adică o soluţie cu un cost scăzut al drumului);
- care este costul căutării[11] asociat timpului calculator şi memoriei necesare pentru a găsi o soluţie.

2.1.3. Căutarea soluţiilor şi generarea secvenţelor de acţiuni

Rezolvarea unei probleme începe cu starea iniţială. Primul pas este acela de a testa dacă starea iniţială este o stare scop. Dacă nu, se iau în consideraţie şi alte stări. Acest lucru se realizează aplicând operatorii asupra stării curente şi, în consecinţă, generând o mulţime de stări. Procesul poartă denumirea de *extinderea stării*. Atunci când se generează mai multe posibilităţi, trebuie făcută o alegere relativ la cea care va fi luată în consideraţie în continuare, aceasta fiind esenţa căutării. *Alegerea referitoare la care dintre stări trebuie extinsă prima este determinată de strategia de căutare.*

Procesul de căutare construieşte un *arbore de căutare*, a cărui rădăcină este un *nod de căutare* corespunzând stării iniţiale. La fiecare pas, algoritmul de căutare alege un nod-frunză pentru a-l extinde. *Algoritmul de căutare general* este următorul [27]:

```
function CĂUTARE_GENERALĂ(problemă, strategie)
        return soluţie sau eşec
iniţializează arborele de căutare folosind starea iniţială a lui problemă
ciclu do
if nu există candidaţi pentru extindere
        then return eşec
alege un nod-frunză pentru extindere conform lui strategie
if nodul conţine o stare scop
        then return soluţia corespunzătoare
        else extinde nodul şi adaugă nodurile rezultate arborelui de căutare
end
```

□

Este important să facem *distincţia între spaţiul stărilor şi arborele de căutare*. Spre exemplu, într-o problemă de căutare a unui drum pe o hartă, pot exista doar 20 de stări în spaţiul stărilor, câte una pentru fiecare oraş. Dar există un număr infinit de drumuri în acest spaţiu de stări. Prin urmare, arborele de căutare are un număr infinit de noduri. Evident, un bun algoritm de căutare trebuie să evite urmarea unor asemenea drumuri.

---

[11] Costul *total* al unei căutări se defineşte ca fiind suma dintre costul drumului corespunzător şi costul respectivei căutări.

2.1.4. Structuri de date pentru arbori de căutare

Există numeroase moduri de a reprezenta nodurile. În general, se consideră că un nod este o structură de date cu cinci componente:
- starea din spaţiul de stări căreia îi corespunde nodul;
- nodul din arborele de căutare care a generat acest nod (nodul părinte);
- operatorul care a fost aplicat pentru a se genera nodul;
- numărul de noduri aflate pe drumul de la rădăcină la acest nod (*adâncimea nodului*);
- costul drumului de la starea iniţială la acest nod.

Este importantă *distincţia între noduri şi stări*. Astfel, un nod este o structură de date folosită pentru a reprezenta arborele de căutare corespunzător unei anumite realizări a unei probleme, generată de un anumit algoritm. O stare reprezintă însă o configuraţie a lumii înconjurătoare. De aceea, nodurile au adâncimi şi părinţi, iar stările nu le au. Mai mult, este posibil ca două noduri diferite să conţină aceeaşi stare, dacă acea stare este generată prin intermediul a două secvenţe de acţiuni diferite. Funcţia EXTINDERE va fi responsabilă pentru determinarea fiecărei componente a nodului pe care îl generează.

Este necesară, de asemenea, reprezentarea colecţiei de noduri care aşteaptă pentru a fi extinse. Această colecţie de noduri poartă denumirea de *frontieră*. Cea mai simplă reprezentare ar fi aceea a unei mulţimi de noduri, iar strategia de căutare ar fi o funcţie care selectează, din această mulţime, următorul nod ce trebuie extins. Deşi din punct de vedere conceptual această cale este una directă, din punct de vedere computaţional ea poate fi foarte scumpă, pentru că funcţia strategie ar trebui să se "uite" la fiecare element al mulţimii pentru a-l alege pe cel mai bun. De aceea, *vom presupune că această colecţie de noduri este implementată ca o coadă*.

Operaţiile definite pe o coadă vor fi următoarele:
- CREEAZĂ_COADA(*Elemente*) - creează o coadă cu elementele date;
- GOL?(*Coada*) - întoarce „true" numai dacă nu mai există elemente în coadă;
- ÎNLĂTURĂ_DIN_FAŢĂ(*Coada*) - înlătură elementul din faţă al cozii şi îl transmite înapoi (return);
- COADA_FN(*Elemente, Coada*) - inserează o mulţime de elemente în coadă.

Diferite feluri de funcţii de acest tip produc algoritmi de căutare diferiţi.

Cu aceste definiţii, putem da o descriere mai formală a ***algoritmului general de căutare***, conform [27]:

```
function CĂUTARE_GENERALĂ(problemă,COADA_FN) return
        o soluţie sau eşec
 noduri ← CREEAZĂ_COADA(CREEAZĂ_NOD(STARE_INIŢIALĂ
        [problemă]))
 ciclu do
```



```
    if noduri este vidă
            then return eşec
    nod ← ÎNLĂTURĂ_DIN_FAŢĂ(noduri)
    if TEST_SCOP[problemă] aplicat lui STARE(nod) are succes
            then return nod
    noduri ← COADA_FN(noduri, EXTINDERE(nod, OPERATORI
            [problemă]))
end
```

□

2.1.5. Evaluarea strategiilor de căutare

Strategiile de căutare se evaluează conform următoarelor patru criterii:
- Completitudine: dacă, atunci când o soluţie există, strategia dată garantează găsirea acesteia;
- Complexitate a timpului: durata de timp pentru găsirea unei soluţii;
- Complexitate a spaţiului: necesităţile de memorie pentru efectuarea căutării;
- Optimalitate: atunci când există mai multe soluţii, strategia dată să o găsească pe cea mai de calitate dintre ele.

2.2. Căutarea neinformată

Termenul de *căutare neinformată* desemnează faptul că o strategie de acest tip nu deţine nici o informaţie despre numărul de paşi sau despre costul drumului de la starea curentă la scop. Tot ceea ce se poate face este să se distingă o stare-scop de o stare care nu este scop. Căutarea neinformată se mai numeşte şi *căutarea oarbă*.

Să considerăm, de pildă, problema găsirii unui drum de la Arad la Bucureşti, având în faţă o hartă. De la starea iniţială, Arad, există trei acţiuni care conduc la trei noi stări: Sibiu, Timişoara şi Zerind. O căutare neinformată nu are nici o preferinţă între cele trei variante. Un agent mai inteligent va observa însă că scopul, Bucureşti, se află la sud-est de Arad şi că numai *Sibiu* este în această direcţie, care reprezintă, probabil, cea mai bună alegere. Strategiile care folosesc asemenea consideraţii se numesc *strategii de căutare informată* sau *strategii de căutare euristică*.

Căutarea neinformată este mai puţin eficientă decât cea informată. Căutarea neinformată este însă importantă deoarece există foarte multe probleme pentru care nu este disponibilă nici o informaţie suplimentară.

În cele ce urmează, vom aborda mai multe strategii de căutare neinformată, acestea deosebindu-se între ele prin **ordinea** în care nodurile sunt extinse.

2.2.1. Căutarea de tip breadth-first

Strategia de căutare de tip breadth-first extinde mai întâi nodul rădăcină. Apoi se extind toate nodurile generate de nodul rădăcină, apoi succesorii lor şi aşa mai departe. În general, toate nodurile aflate la adâncimea d în arborele de căutare sunt extinse înaintea nodurilor aflate la adâncimea $d+1$. Spunem ca aceasta este o *căutare în lăţime*.

Căutarea de tip breadth-first poate fi implementată chemând algoritmul general de căutare, CĂUTARE_GENERALĂ, cu o funcţie COADA_FN care plasează stările nou generate *la sfârşitul cozii*, după toate stările generate anterior.

Strategia breadth-first este foarte *sistematică* deoarece ia în consideraţie toate drumurile de lungime 1, apoi pe cele de lungime 2 etc., aşa cum se arată în Fig. 2.1. Dacă există o soluţie, este sigur că această metodă o va găsi, iar dacă există mai multe soluţii, căutarea de tip breadth-first va găsi întotdeauna mai întâi *soluţia cel mai puţin adâncă*.

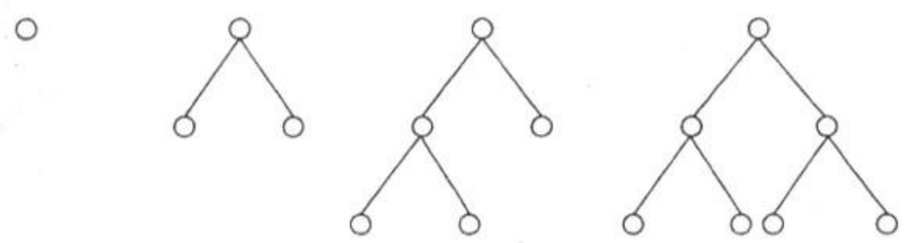


Fig. 2.1

În termenii celor patru criterii de evaluare a strategiilor de căutare, cea de tip breadth-first este *completă* şi este *optimă* cu condiţia ca *costul drumului să fie o funcţie descrescătoare de adâncimea nodului*. Această condiţie este de obicei satisfăcută numai atunci când toţi operatorii au acelaşi cost.

2.2.1.1. Algoritmul de căutare breadth-first

Presupunând că a fost specificată o mulţime de reguli care descriu acţiunile sau operatorii disponibili, ***algoritmul de căutare breadth-first*** se defineşte după cum urmează:

Algoritmul 2.1

1. Creează o variabilă numită LISTA_NODURI şi setează-o la starea iniţială.

2. Până când este găsită o stare-scop sau până când LISTA_NODURI devine vidă, execută:

2.1. Înlătură primul element din LISTA_NODURI şi numeşte-l E. Dacă LISTA_NODURI a fost vidă, STOP.

2.2. Pentru fiecare mod în care fiecare regulă se potriveşte cu starea descrisă în E, execută:

2.2.1. Aplică regula pentru a genera o nouă stare.

2.2.2. Dacă noua stare este o stare-scop, întoarce această stare şi STOP.

2.2.3. Altfel, adaugă noua stare la sfârşitul lui LISTA_NODURI.

2.2.1.2. Implementare Prolog

Pentru a programa în Prolog strategia de căutare breadth-first, trebuie menţinută în memorie *o mulţime de noduri candidate alternative*. Această mulţime de candidaţi reprezintă marginea de jos a arborelui de căutare, aflată în continuă creştere (frontiera). Totuşi, această mulţime de noduri nu este suficientă dacă se doreşte şi extragerea unui drum-soluţie în urma procesului de căutare. Prin urmare, în loc de a menţine o mulţime de noduri candidate, vom menţine *o mulţime de drumuri candidate*[12].

Este utilă, pentru programarea în Prolog, o anumită reprezentare a mulţimii de drumuri candidate, şi anume: *mulţimea va fi reprezentată ca o listă de drumuri, iar fiecare drum va fi o listă de noduri în ordine inversă*. Capul listei va fi, prin urmare, nodul cel mai recent generat, iar ultimul element al listei va fi nodul de început al căutării.

Căutarea este începută cu o mulţime de candidaţi având un singur element:

[[NodIniţial]].

Fiind dată o mulţime de drumuri candidate, căutarea de tip breadth-first se desfăşoară astfel:

- dacă primul drum conţine un nod-scop pe post de cap, atunci acesta este o soluţie a problemei;
- altfel, înlătură primul drum din mulţimea de candidaţi şi generează toate extensiile de un pas ale acestui drum, adăugând această mulţime de

[12] Implementarea în Prolog a strategiei breadth-first propusă aici este preluată din [3].

extensii la sfârşitul mulţimii de candidaţi. Execută apoi căutarea de tip breadth-first asupra mulţimii astfel actualizate.

Vom considera un exemplu în care nodul *a* este nodul de start, *f* şi *j* sunt nodurile-scop, iar spaţiul stărilor este cel din Fig. 2.2:

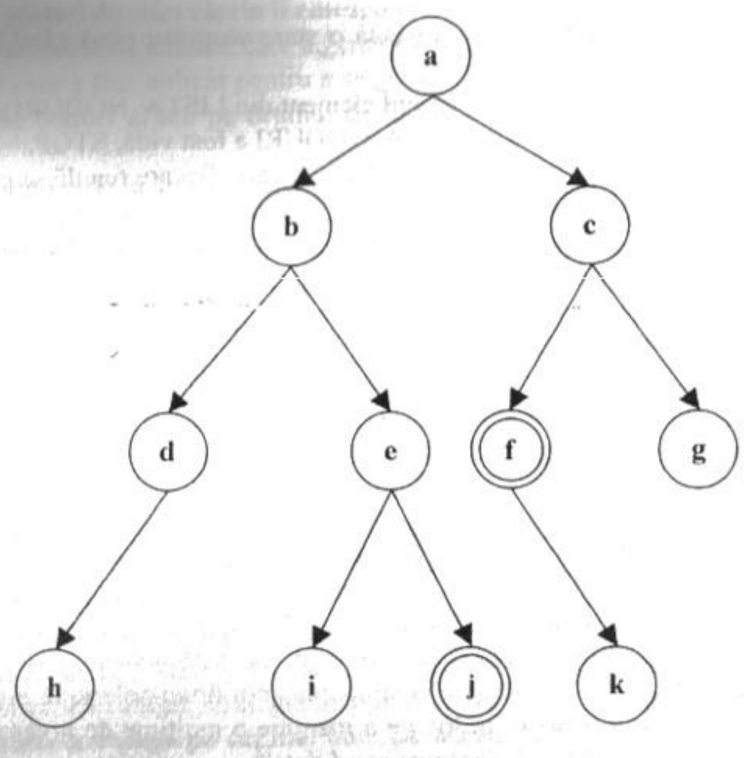


Fig. 2.2

Ordinea în care strategia breadth-first vizitează nodurile din acest spaţiu de stări este: *a, b, c, d, e, f*. Soluţia mai scurtă [a, c, f] va fi găsită înaintea soluţiei mai lungi [a, b, e, j].

Pentru figura anterioară, căutarea breadth-first se desfăşoară astfel:

(1) Se începe cu mulţimea de candidaţi iniţială:

[[a]]

(2) Se generează extensii ale lui [a]:

[[b, a], [c, a]]

(Se observă reprezentarea drumurilor în ordine inversă).

(3) Se înlătură primul drum candidat, [b, a], din mulţime şi se generează extensii ale acestui drum:

[[d, b, a], [e, b, a]]

Se adaugă lista extensiilor la sfârşitul mulţimii de candidaţi:

[[c, a], [d, b, a], [e, b, a]]

(4) Se înlătură [c, a] şi se adaugă extensiile sale la sfârşitul mulţimii de candidaţi, rezultând următoarea mulţime de drumuri:

[[d, b, a], [e, b, a], [f, c, a], [g, c, a]]

La următorii paşi, [d, b, a] şi [e, b, a] sunt extinse, iar mulţimea de candidaţi modificată devine:

[[f, c, a], [g, c, a], [h, d, b, a], [i, e, b, a], [j, e, b, a]]

Acum procesul de căutare întâlneşte [f, c, a], care conţine un nod scop, *f*. Prin urmare, acest drum este returnat ca soluţie.

Programul Prolog care implementează acest proces de căutare va reprezenta mulţimile de noduri ca pe nişte liste, efectuând şi un test care să prevină generarea unor drumuri ciclice. În cadrul acestui program, toate extensiile de un pas vor fi generate prin utilizarea procedurii încorporate **bagof**:

```
rezolva_b(Start,Sol):-breadthfirst([[Start]],Sol).
breadthfirst([[Nod|Drum]|_],[Nod|Drum]):-scop(Nod).
breadthfirst([Drum|Drumuri],Sol):-
        extinde(Drum,DrumuriNoi),
        concat(Drumuri,DrumuriNoi,Drumuri1),
        breadthfirst(Drumuri1,Sol).
extinde([Nod|Drum],DrumuriNoi):-
        bagof([NodNou,Nod|Drum],
        (s(Nod,NodNou),\+(membru(NodNou,[Nod|Drum]))),
        DrumuriNoi),
        !.
extinde(_,[]).
```

Predicatul `rezolva_b(Start,Sol)` este adevărat dacă Sol este un drum (în ordine inversă) de la nodul iniţial Start la o stare-scop, drum obţinut folosind căutarea de tip breadth-first.

Predicatul `breadthfirst(Drumuri,Sol)` este adevărat dacă un drum din mulţimea de drumuri candidate numită Drumuri poate fi extins la o stare-scop; un astfel de drum este Sol.

Predicatul `extinde(Drum,DrumuriNoi)` este adevărat dacă prin extinderea mulţimii de noduri Drum obţinem mulţimea numită DrumuriNoi, el generând mulţimea tuturor extensiilor acestui drum.

Predicatul `concat(Drumuri,DrumuriNoi,Drumuri1)` este adevărat dacă, atunci când concatenăm lista de noduri Drumuri cu lista de noduri DrumuriNoi, obţinem lista de noduri Drumuri1.

Predicatul `membru(NodNou,[Nod|Drum])` este adevărat dacă nodul numit NodNou aparţine listei de noduri [Nod|Drum].

Fapta Prolog `scop(Nod)` arată că Nod este un nod-scop.

Funcţia de succesiune este implementată astfel: `s(Nod,NodNou)` desemnează faptul că NodNou este nodul succesor al nodului Nod.

Prezentăm, în continuare, programul Prolog complet corespunzător exemplului din Fig. 2.2. Programul implementează strategia de căutare breadth-first pentru a găsi soluţiile, cele două drumuri [f, c, a] şi respectiv [j, e, b, a]:

Programul 2.1

```
scop(f).   % specificare noduri-scop
scop(j).
s(a,b).    % descrierea funcţiei succesor
s(a,c).
s(b,d).
s(d,h).
s(b,e).
s(e,i).
s(e,j).
s(c,f).
s(c,g).
s(f,k).

concat([],L,L).
concat([H|T],L,[H|T1]):-concat(T,L,T1).

membru(H,[H|T]).
membru(X,[H|T]):-membru(X,T).

rezolva_b(Start,Sol):-breadthfirst([[Start]],Sol).

breadthfirst([[Nod|Drum]|_],[Nod|Drum]):-scop(Nod).
breadthfirst([Drum|Drumuri],Sol):-
      extinde(Drum,DrumuriNoi),
      concat(Drumuri,DrumuriNoi,Drumuri1),
      breadthfirst(Drumuri1,Sol).

extinde([Nod|Drum],DrumuriNoi):-
      bagof([NodNou,Nod|Drum],(s(Nod,NodNou),
      \+(membru(NodNou,[Nod|Drum]))),DrumuriNoi),
      !.
extinde(_,[]).
```

Interogarea Prologului se face astfel:

```
?- rezolva_b(a,Sol).
```


Răspunsul Prologului va fi:

```
Sol=[f, c, a] ? ;
Sol=[j, e, b, a] ? ;
no
```

Cele două soluţii au fost obţinute ca liste de noduri în ordine inversă, plecându-se de la nodul de start *a*.

Un *neajuns* al acestui program îl reprezintă lipsa de eficienţă a operaţiei de concatenare. Această problemă poate fi rezolvată prin reprezentarea listelor sub forma diferenţei de liste. Pentru un exemplu de implementare a algoritmului breadth-first folosind ***reprezentarea listelor prin diferenţe***, vezi §2.2.1.4.

2.2.1.3. Timpul şi memoria cerute de strategia breadth-first

Pentru a vedea cantitatea de timp şi de memorie necesare completării unei căutări, vom lua în consideraţie un spaţiu al stărilor ipotetic, în care fiecare stare poate fi extinsă pentru a genera *b* stări noi. Se spune că *factorul de ramificare* al acestor stări (şi al arborelui de căutare) este *b*. Rădăcina generează *b* noduri la primul nivel, fiecare dintre acestea generează încă *b* noduri, rezultând un total de b^2 noduri la al doilea nivel ş.a.m.d.. Să presupunem că soluţia acestei probleme este un drum de lungime *d*. Atunci numărul maxim de noduri extinse înainte de găsirea unei soluţii este:

$$1+b+b^2+\cdots+b^d.$$

Prin urmare, algoritmul are o *complexitate exponenţială* de $O(b^d)$. Complexitatea spaţiului este aceeaşi cu complexitatea timpului deoarece toate nodurile frunză ale arborelui trebuie să fie menţinute în memorie în acelaşi timp.

Iată câteva exemple de execuţii cu factor de ramificare b=10, preluate de noi din [27]:

Adâncime	Noduri	Timp	Memorie
2	111	0.1 sec.	11 kilobytes
6	10^6	18 min.	111 megabytes
8	10^8	31 ore	11 gigabytes
12	10^{12}	35 ani	111 terabytes

Se observă că cerinţele de memorie sunt o problemă mai mare, pentru căutarea de tip breadth-first, decât timpul de execuţie. (Este posibil să putem aştepta 18 minute pentru a se efectua o căutare de adâncime 6, dar să nu dispunem de 111

megabytes de memorie). La rândul lor, cerințele de timp sunt majore. (Dacă problema are o soluție la adâncimea 12, o căutare neinformată de acest tip o găsește în 35 de ani). În general, problemele de căutare de *complexitate exponențială* nu pot fi rezolvate decât pe mici porțiuni.

2.2.1.4. Un exemplu. Problema misionarilor și canibalilor

Pe malul estic al unei ape se găsesc N misionari si N canibali. Aceștia urmează să treacă apa, având la dispoziție o barcă cu M locuri. Se știe că, dacă pe unul dintre maluri numărul de canibali este mai mare decât numărul de misionari, atunci misionarii de pe acel mal sunt mâncați de canibali. Se cere determinarea unei variante de trecere a apei fără ca misionarii să fie mâncați de canibali, dacă aceasta există.

În implementarea care urmează vom reprezenta o stare a problemei prin intermediul unui termen Prolog de forma:

```
st(MalBarca,NMisMBarca,NCanMBarca,NMisMOpus,
NCanMOpus,M,N)
```

Primul argument specifică malul pe care se află barca (est sau vest), iar următoarele patru constituie numărul de misionari și de canibali de pe malul pe care se găsește barca și respectiv de pe malul opus. Ultimele două argumente, M și N, se referă la numărul de locuri din barcă și respectiv la numărul de canibali și de misionari aflați inițial pe malul estic al râului.

Programul 2.2

```
%Predicatele specifice problemei misionarilor si
%canibalilor

%Definim relatia de succesiune.

%Predicatul st(S1,S2) este adevarat daca exista un arc
%intre S1 si S2 in spatiul starilor.

s(st(B,NMB,NCB,NMis,NCan,M,N),st(B1,NMisB1,NCanB1,
NMis1,NCan1,M,N)):-
        maluri_opuse(B,B1),
        intre(K1,0,M),
        %determinam K1, numarul de misionari care trec
        %cu barca
        intre(K2,0,M),
```



```
        %determinam K1, numarul de canibali care trec
        %cu barca
        K1+K2=<M,K1+K2>0,
       (K1=0;K1=\=0,K1>=K2),
        NMB>=K1,NCB>=K2,NMisB1 is NMis+K1,NCanB1 is
        NCan+K2,
        NMis1 is NMB-K1,NCan1 is
        NCB-K2,in_regula(NMisB1,NCanB1),
        in_regula(NMis1,NCan1).

membru(X,[X|_]).
membru(X,[_|Y]):-membru(X,Y).
intre(A,A,B):-A=<B.
intre(X,A,B):-A<B,A1 is A+1, intre(X,A1,B).

%In starea initiala toti canibalii si toti misionarii se
%afla pe malul estic.

initial(st(est,N,N,0,0,M,N),M,N).

%In starea finala toti canibalii si toti misionarii se afla
%pe malul vestic.

scop(st(vest,N,N,0,0,_,N)).

%Predicatul maluri_opuse(X,Y) este adevarat daca X si Y
%sunt maluri opuse.

maluri_opuse(est,vest).
maluri_opuse(vest,est).

in_regula(0,_).
in_regula(X,Y):-X>0,X>=Y.

%Implementarea algoritmului breadth-first folosind
%reprezentarea listelor prin diferente

extinde([Node|Path],NewPaths):-
        findall_liste_dif([NewNode,Node|Path],
        (s(Node,NewNode),
        \+ (membru(NewNode,[Node|Path]))),
        NewPaths).

findall_liste_dif(X,Scop,L):-
        call(Scop),assertz(elem(X)),fail;
        assertz(elem(baza)),colectez(L).
colectez(L):-retract(elem(X)),!,
```

```
        (X==baza,!,L=Z-Z;
        colectez(Rest),concat_liste_dif(Rest,
                                [X|A]-A,L)).

concat_liste_dif(A1-Z1,Z1-Z2,A1-Z2).

rezolva_bf_dif(Nod_initial,Solutie):-
        breadthfirst_dif([[Nod_initial]|Z]-Z,
                         Solutie).

breadthfirst_dif([[Nod|Drum]|_]-_,[Nod|Drum]):-
            scop(Nod).
breadthfirst_dif([Drum|Drumuri]-Z1,Solutie):-
           extinde(Drum,LL),
            concat_liste_dif(Drumuri-Z1,LL,A2-Z2),
            A2\==Z2,
            breadthfirst_dif(A2-Z2,Solutie).

%Utilizarea algoritmului breadth-first pentru rezolvarea
%problemei canibalilor si misionarilor

solutie(N,M):-
   tell('C:\\canib_rez.txt'),
   initial(Start,M,N),
   (rezolva_bf_dif(Start,Solutie),lung(Solutie,N1),!,
   NN is N1-1,afisare(Solutie),nl,nl,
   write('Nr mutari: '),write(NN);
   nl,write('Problema nu are solutie !!!')),
   told.

scrie(st(B,MX,CX,MY,CY,_,_)):-
    write('Barca se afla pe malul de '),write(B),
    write('.'),nl,maluri_opuse(B,B1),
    (B=est, write('Pe malul de '),write(B),
    write(' se afla '),write(MX),scrie_nr(MX,m),
    write(' si '),write(CX),scrie_nr(CX,c),
    write('.'),nl,write('Pe malul de '),write(B1),
    write(' se afla '),write(MY),scrie_nr(MY,m),
    write(' si '),write(CY),scrie_nr(CY,c);
    B=vest, write('Pe malul de '),write(B1),
    write(' se afla '),write(MY),scrie_nr(MY,m),
    write(' si '),write(CY),scrie_nr(CY,c),
    write('.'),nl,write('Pe malul de '),write(B),
    write(' se afla '),write(MX),scrie_nr(MX,m),
    write('si'),write(CX),scrie_nr(CX,c)),write('.').

scrie_nr(X,Y):-
```



```
        Y==m,(X==1, write(' misionar');
        X\==1,write(' misionari'));
        Y==c,(X==1, write(' canibal');
        X\==1,write(' canibali')).

afisare([]).
afisare([H|T]):- afisare(T),scrie(H),nl,nl.

lung([],0).
lung([_|T],N):-lung(T,N1),N is N1+1.
```

Interogarea Prologului se face, spre exemplu, după cum urmează:

```
?- solutie(3,2).
```

Răspunsul sistemului va fi `yes` cu semnificația că predicatul a fost satisfăcut. În fișierul `C:\canib_rez.txt` putem vedea și secvența de mutări prin care s-a ajuns din starea inițială în starea finală (în situația în care avem 3 canibali, 3 misionari și 2 locuri în barcă). După interogarea anterioară, conținutul fișierului `C:\canib_rez.txt` va fi următorul:

```
Barca se afla pe malul de est.
Pe malul de est se afla 3 misionari si 3 canibali.
Pe malul de vest se afla 0 misionari si 0 canibali.

Barca se afla pe malul de vest.
Pe malul de est se afla 2 misionari si 2 canibali.
Pe malul de vest se afla 1 misionar si 1 canibal.

Barca se afla pe malul de est.
Pe malul de est se afla 3 misionari si 2 canibali.
Pe malul de vest se afla 0 misionari si 1 canibal.

Barca se afla pe malul de vest.
Pe malul de est se afla 3 misionari si 0 canibali.
Pe malul de vest se afla 0 misionari si 3 canibali.

Barca se afla pe malul de est.
Pe malul de est se afla 3 misionari si 1 canibal.
Pe malul de vest se afla 0 misionari si 2 canibali.

Barca se afla pe malul de vest.
Pe malul de est se afla 1 misionar si 1 canibal.
Pe malul de vest se afla 2 misionari si 2 canibali.
```

Barca se afla pe malul de est.
Pe malul de est se afla 2 misionari si 2 canibali.
Pe malul de vest se afla 1 misionar si 1 canibal.

Barca se afla pe malul de vest.
Pe malul de est se afla 0 misionari si 2 canibali.
Pe malul de vest se afla 3 misionari si 1 canibal.

Barca se afla pe malul de est.
Pe malul de est se afla 0 misionari si 3 canibali.
Pe malul de vest se afla 3 misionari si 0 canibali.

Barca se afla pe malul de vest.
Pe malul de est se afla 0 misionari si 1 canibal.
Pe malul de vest se afla 3 misionari si 2 canibali.

Barca se afla pe malul de est.
Pe malul de est se afla 1 misionar si 1 canibal.
Pe malul de vest se afla 2 misionari si 2 canibali.

Barca se afla pe malul de vest.
Pe malul de est se afla 0 misionari si 0 canibali.
Pe malul de vest se afla 3 misionari si 3 canibali.

Nr mutari: 11

2.2.2. Căutarea de tip depth-first

2.2.2.1. Prezentare generală şi comparaţie cu strategia breadth-first

Strategia de căutare de tip depth-first extinde întotdeauna unul dintre nodurile aflate la nivelul *cel mai adânc* din arbore. Căutarea se întoarce înapoi şi sunt extinse noduri aflate la adâncimi mai mici numai atunci când a fost atins un nod care nu reprezintă un nod-scop şi care nu mai poate fi extins. Spunem că aceasta este o *căutare în adâncime*. Modul de a progresa al căutării de tip depth-first este ilustrat în Fig. 2.3.

Implementarea se poate face cu o structură de date de tip *coadă*, care întotdeauna va plasa stările nou generate *în faţa cozii*.

Necesităţile de memorie sunt foarte mici la această metodă. După cum se vede în Fig. 2.3, este necesar să se memoreze un singur drum de la rădăcină la un nod-frunză, împreună cu nodurile-frate rămase neextinse corespunzător fiecărui nod de pe drum.

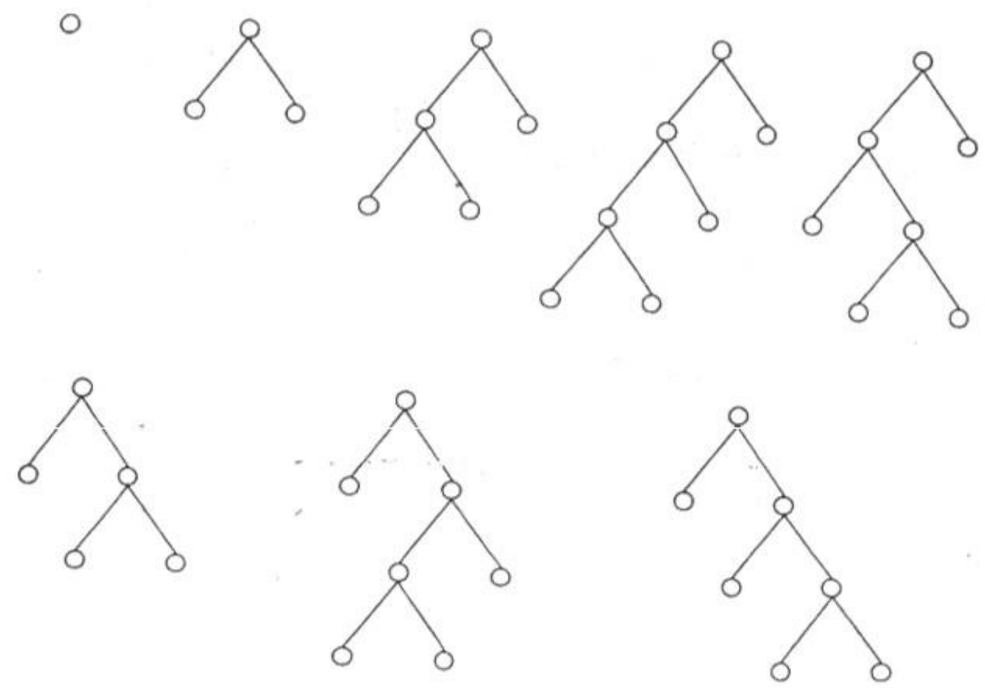


Fig. 2.3

Pentru un spaţiu al stărilor cu factor de ramificare b şi adâncime maximă m, trebuie memorate numai bm noduri, spre deosebire de cele b^d care ar trebui memorate în cazul strategiei de tip breadth-first (atunci când scopul cel mai puţin adânc se află la adâncimea d).

Complexitatea de timp a strategiei depth-first este de $O(b^m)$. Pentru probleme care au foarte multe soluţii, căutarea de tip depth-first s-ar putea să fie mai rapidă decât cea de tip breadth-first, deoarece sunt şanse mari să fie găsită o soluţie după ce se explorează numai o mică porţiune a întregului spaţiu de căutare. Strategia breadth-first trebuie să investigheze toate drumurile de lungime d-1 înainte de a lua în consideraţie pe oricare dintre drumurile de lungime d. Depth-first este de complexitate $O(b^m)$ şi în cazul cel mai nefavorabil.

Principalul *dezavantaj* al acestei strategii este acela că o căutare de tip depth-first va continua întotdeauna în jos, chiar dacă o soluţie mai puţin adâncă există. Această strategie

- poate intra într-un ciclu infinit fără a returna vreodată o soluţie;

- poate găsi un drum reprezentând o soluție, dar care este mai lung decât drumul corespunzător soluției optime; din aceste motive căutarea de tip depth-first nu este nici completă și nici optimală.

Prin urmare, această strategie trebuie evitată în cazul arborilor de căutare de adâncimi maxime foarte mari sau infinite.

Așa cum s-a argumentat deja, principalele *avantaje* ale acestui tip de căutare sunt:

- consumul redus de memorie;
- posibilitatea găsirii unei soluții fără a se explora o mare parte din spațiul de căutare.

2.2.2.2. Implementare Prolog

Strategia depth-first este cea care se potrivește cel mai bine cu stilul de programare *recursiv* din Prolog. Motivul pentru aceasta este faptul că, însuși Prologul, atunci când execută scopuri, explorează diferite alternative în manieră depth-first.

Problema găsirii unui drum-soluție, **Sol**, de la un nod dat, **N**, până la un nod-scop, se rezolvă în felul următor:

- dacă **N** este un nod-scop, atunci **Sol=[N]** sau
- dacă există un nod succesor, **N1**, al lui **N**, astfel încât să existe un drum, **Sol1**, de la **N1** la un nod-scop, atunci **Sol=[N|Sol1]**.

Aceasta se traduce în Prolog astfel[13]:

```
rezolva_d(N,[N]):-
            scop(N).
rezolva_d(N,[N|Sol1]):-
            s(N,N1),
            rezolva_d(N1,Sol1).
```

Interogarea Prologului se va face în felul următor:

```
?- rezolva_d(a,Sol).
```

Pentru exemplificare, vom lua din nou în considerație spațiul stărilor din Fig. 2.2, în care *a* este nodul de start, iar *f* și *j* sunt noduri- scop. Ordinea în care strategia depth-first vizitează nodurile în acest spațiu de stări este: *a, b, d, h, e, i, j*. Soluția găsită este [a, b, e, j]. După efectuarea backtracking-ului este găsită și cealaltă soluție, [a, c, f]. Prezentăm, în continuare, programul Prolog complet, corespunzător acestui exemplu:

[13] Implementarea în Prolog a strategiei depth-first propusă aici este preluată din [3].

Programul 2.3

```
scop(f).   % specificare noduri-scop
scop(j).
s(a,b).    % descrierea funcției succesor
s(a,c).
s(b,d).
s(d,h).
s(b,e).
s(e,i).
s(e,j).
s(c,f).
s(c,g).
s(f,k).

rezolva_d(N,[N]):-scop(N).
rezolva_d(N,[N|Sol1]):-
            s(N,N1),
            rezolva_d(N1,Sol1).
```

Interogarea Prologului se face astfel:

```
?- rezolva_d(a,Sol).
```

Răspunsul Prologului va fi:

```
Sol=[a,b,e,j] ? ;
Sol=[a,c,f] ? ;
no
```

Se observă faptul că forma drumurilor-soluție este cea firească, de la nodul de start la nodul-scop (spre deosebire de cazul strategiei breadth-first, unde nodurile intervin în ordine inversă).

Există însă multe situații în care procedura `rezolva_d` poate să nu lucreze "bine", acest fapt depinzând în exclusivitate de spațiul de stări. Pentru exemplificare, este suficient să adăugăm un arc de la nodul *h* la nodul *d* în spațiul de stări reprezentat de Fig. 2.2. Corespunzător Fig. 2.4, căutarea se va efectua în felul următor: se pleacă din nodul *a*, apoi se coboară la nodul *h*, pe ramura cea mai din stânga a arborelui. În acest moment, spre deosebire de situația anterioară, *h* are un succesor, și anume pe *d*. Prin urmare, de la *h*, execuția **nu** va mai efectua un backtracking, ci se va îndrepta spre *d*. Apoi va fi găsit succesorul lui *d*, adică *h*, ș.a.m.d., rezultând un *ciclu infinit* între *d* și *h*.

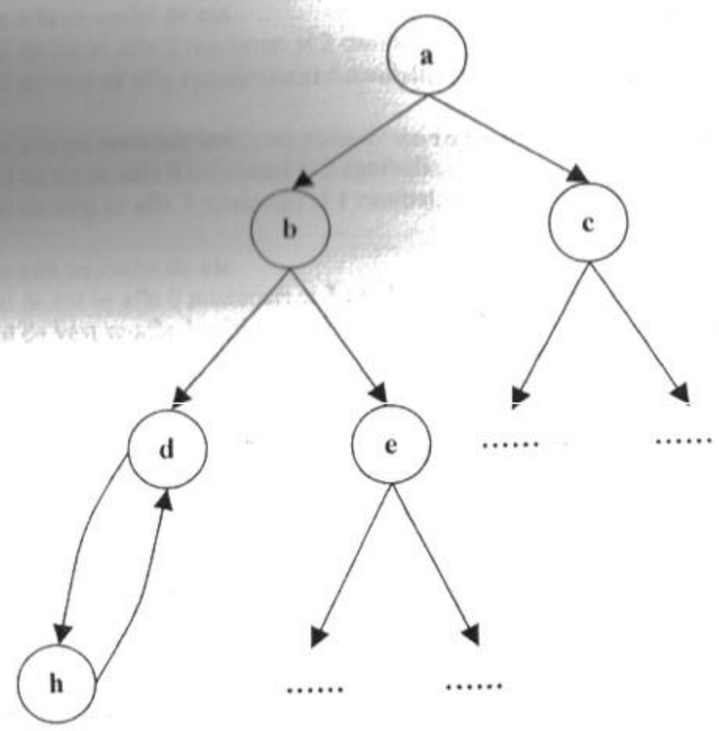


Fig. 2.4

Devine astfel evident faptul că, pentru îmbunătățirea programului, trebuie adăugat **un mecanism de detectare a ciclurilor**. Conform acestuia, orice nod care se află deja pe drumul de la nodul de start la nodul curent nu mai trebuie luat vreodată în considerație. Această cerință poate fi formulată ca o relație:

depthfirst(Drum, Nod, Soluție).

Aici **Nod** este starea pornind de la care trebuie găsit un drum la o stare-scop; **Drum** este un drum, adică o listă de noduri, între nodul de start și **Nod**; **Soluție** este **Drum**, extins via **Nod**, la un nod-scop. Reprezentarea relației este cea din Fig. 2.5.

Argumentul **Drum** poate fi folosit cu două scopuri:

- să împiedice algoritmul să ia în considerație acei succesori ai lui **Nod** care au fost deja întâlniți, adică *să detecteze ciclurile*;
- să construiască un drum-soluție numit **Soluție**.

Predicatul corespunzător,

```
depthfirst(Drum,Nod,Sol),
```

este adevărat dacă, extinzând calea **Drum** via **Nod**, se ajunge la un nod-scop.

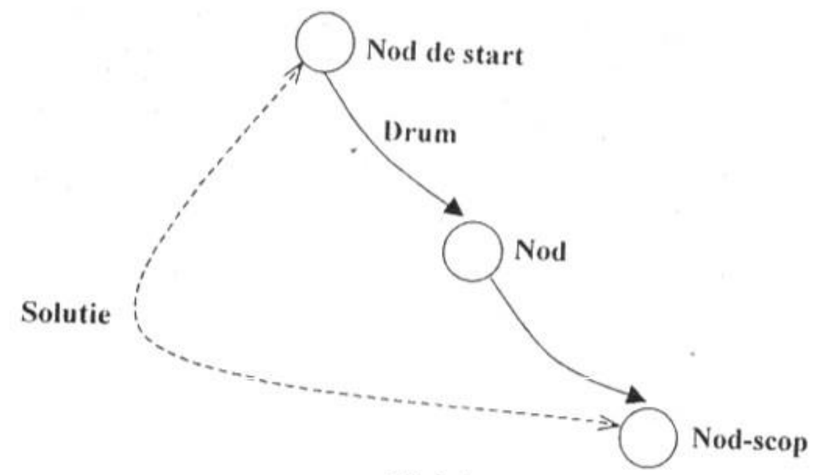


Fig.2.5

În cele ce urmează, prezentăm implementarea în Prolog a strategiei depth-first cu detectare a ciclurilor:

```
rezolval_d(N,Sol):-depthfirst([ ],N,Sol).
depthfirst(Drum,Nod,[Nod|Drum]):-scop(Nod).
depthfirst(Drum,Nod,Sol):-s(Nod,Nod1),
                          \+(membru(Nod1,Drum)),
                          depthfirst([Nod|Drum],Nod1,Sol).
```

Se observă că fragmentul de program anterior verifică dacă anumite noduri au fost luate deja în considerație până la momentul curent, punând condiția de nonapartenență: `\+(membru(Nod1,Drum))`. Drumurile-soluție sunt date ca liste de noduri în ordine inversă, așa cum se va vedea și în exemplul care urmează.

Prezentăm, în continuare, programul Prolog complet corespunzător exemplului din Fig. 2.4. Programul implementează strategia de căutare depth-first cu detectare a ciclurilor, pentru găsirea celor două soluții posibile:

Programul 2.4

```
scop(f).      % specificare noduri-scop
scop(j).
s(a,b).       % descrierea funcției succesor
s(a,c).
s(b,d).
s(d,h).
s(h,d).
```

```
s(b,e).
s(e,i).
s(e,j).
s(c,f).
s(c,g).
s(f,k).

rezolval_d(N,Sol):-depthfirst([ ],N,Sol).

depthfirst(Drum,Nod,[Nod|Drum]):-scop(Nod).
depthfirst(Drum,Nod,Sol):-s(Nod,Nod1),
                          \+(membru(Nod1,Drum)),
                          depthfirst([Nod|Drum],Nod1,Sol).

membru(H,[H|T]).
membru(X,[H|T]):-membru(X,T).
```

Interogarea Prologului se face astfel:

```
?- rezolval_d(a,Sol).
```

Răspunsul Prologului va fi:

```
Sol=[j,e,b,a] ? ;
Sol=[f,c,a] ? ;
no
```

Pentru un exemplu mai complex de problemă a cărei rezolvare necesită o căutare de tip depth-first cu mecanism de detectare a ciclurilor, vezi §2.2.4.

Aşa cum se arată în [3], un program de acest tip nu va lucra totuşi corect atunci când *spaţiul stărilor este infinit*. Într-un astfel de spaţiu algoritmul depth-first poate omite un nod-scop deplasându-se de-a lungul unei ramuri infinite a grafului. Programul poate atunci să exploreze în mod nedefinit această parte infinită a spaţiului, fără a se apropia vreodată de un scop. Pentru a evita astfel de *ramuri aciclice infinite*, procedura de căutare depth-first de bază poate fi, în continuare, rafinată, prin *limitarea adâncimii căutării*. În acest caz, procedura de căutare depth-first va avea următoarele argumente:

depthfirst1(Nod, Soluţie, Maxdepth),

unde Maxdepth este adâncimea maximă până la care se poate efectua căutarea.

Această constrângere poate fi programată prin micşorarea limitei de adâncime la fiecare apelare recursivă, fără a permite ca această limită să devină negativă. Tipul acesta de căutare se numeşte **Depth-limited search** ("căutare cu adâncime limitată"). Predicatul Prolog corespunzător,

```
depthfirst1(Nod,Sol,Max),
```


va fi adevărat dacă, pornind din nodul **Nod**, obţinem un drum-soluţie numit **Sol**, prin efectuarea unei căutări de tip depth-first până la adâncimea maximă notată **Max**. Lăsăm cititorului ca exerciţiu scrierea programului Prolog care va explora cu succes un spaţiu aciclic infinit, prin stabilirea unei limite de adâncime.

Numărul de extinderi într-o căutare depth-limited, până la o adâncime d, cu factor de ramificare b, este:

$$1+b+b^2+\cdots+b^{d-1}+b^d$$

Neajunsul în cazul acestui tip de căutare este acela că trebuie găsită dinainte o limită convenabilă a adâncimii căutarii. Dacă această limită este prea mică, căutarea va eşua. Dacă limita este mare, atunci căutarea va deveni prea scumpă. Pentru a evita aceste neajunsuri, putem executa căutarea de tip depth-limited în mod iterativ, variind limita pentru adâncime. Se începe cu o limită a adâncimii foarte mică şi se măreşte această limită în mod gradat, până când se găseşte o soluţie. Această tehnică de căutare se numeşte **Iterative Deepening Search** ("căutare în adâncime iterativă").

2.2.3. Căutarea în adâncime iterativă

2.2.3.1. Prezentare generală

Căutarea în adâncime iterativă este o strategie care evită chestiunea stabilirii unei adâncimi optime la care trebuie căutată soluţia, prin testarea tuturor limitelor de adâncime posibile: mai întâi adâncimea 0, apoi 1, apoi 2, ş.a.m.d.. Acest tip de căutare combină beneficiile căutării breadth-first şi depth-first, după cum urmează:

- este optimă şi completă ca şi căutarea breadth-first;
- consumă numai cantitatea mică de memorie necesară căutarii depth-first (cerinţa de memorie este liniară).

Ordinea extinderii stărilor este similară cu cea de la căutarea de tip breadth-first, numai că anumite stări sunt extinse de mai multe ori. Această strategie de căutare garantează găsirea nodului-scop de la adâncimea minimă, *dacă* un scop poate fi găsit.

Deşi anumite noduri sunt extinse de mai multe ori, *numărul total de noduri extinse* nu este mult mai mare decât cel dintr-o căutare de tip breadth-first. Vom calcula acest număr de noduri extinse *în cazul cel mai nefavorabil*, în care se caută într-un arbore cu factor de ramificare b, scopul cel mai puţin adânc aflându-se la adâncimea d şi fiind *ultimul* nod care va fi generat la acea adâncime. Reamintim că numărul de noduri extinse de o căutare de tip breadth-first în aceste condiţii poate urca până la:

$$N_{bf}=1+b+b^2+\cdots+b^d=\frac{b^{d+1}-1}{b-1}$$

Pentru a calcula numărul nodurilor extinse de căutarea în adâncime iterativă, vom nota, mai întâi, faptul că numărul nodurilor extinse de o căutare de tip depth-first *completă* până la nivelul j este:

$$N_{dfj} = \frac{b^{j+1} - 1}{b - 1}$$

În cel mai nefavorabil caz, pentru un scop aflat la adâncimea d, căutarea depth-first iterativă trebuie să efectueze căutări de tip depth-first complete, separate, pentru toate adâncimile până la d. Suma nodurilor extinse în această situaţie este:

$$N_{id} = \sum_{j=0}^{d} \frac{b^{j+1} - 1}{b - 1} = \frac{1}{b-1} \sum_{j=0}^{d} [b^{j+1} - 1] = \frac{1}{b-1} [\sum_{j=0}^{d} b^{j+1} - \sum_{j=0}^{d} 1]$$

$$= \frac{1}{b-1} [b(\sum_{j=0}^{d} b^{j}) - \sum_{j=0}^{d} 1] = \frac{1}{b-1} [b \frac{b^{d+1} - 1}{b-1} - (d+1)] =$$

$$= \frac{b(b^{d+1} - 1) - (b-1)(d+1)}{(b-1)^2} = \frac{b^{d+2} - b - bd - b + d + 1}{(b-1)^2} =$$

$$= \frac{b^{d+2} - 2b - bd + d + 1}{(b-1)^2}$$

Spre exemplu [25], atunci când $b=10$, iar scopurile se află la mare adâncime, o căutare în adâncime iterativă de la adâncimea 1 până la adâncimea d extinde cu numai 11% noduri mai mult decât o singură căutare de tip breadth-first sau decât una în adâncime limitată la adâncimea d. Cu cât factorul de ramificare are o valoare mai mare, cu atât se micşorează costurile legate de extinderea repetată a aceloraşi stări. Chiar şi atunci când factorul de ramificare are valoarea 2, căutarea în adâncime iterativă durează numai de aproximativ două ori mai mult decât o căutare de tip breadth-first completă [27]. Aceasta înseamnă că strategia de căutare în adâncime iterativă are tot complexitatea de timp $O(b^d)$, iar complexitatea sa de spatiu este $O(bd)$. În general, căutarea în adâncime iterativă este metoda de căutare preferată atunci când există un spaţiu al căutării foarte mare, iar adâncimea soluţiei nu este cunoscută.

2.2.3.2. Implementare Prolog

Pentru implementarea în Prolog a căutării în adâncime iterative vom folosi predicatul `cale` de forma

```
cale(Nod1,Nod2,Drum)
```


care este adevărat dacă **Drum** reprezintă o cale aciclică între nodurile **Nod1** şi **Nod2** în spaţiul stărilor. Această cale va fi reprezentată ca o listă de noduri date în ordine inversă. Corespunzător nodului de start dat, predicatul `cale` generează toate drumurile aciclice posibile de lungime care creşte cu câte o unitate. Drumurile sunt generate până când se generează o cale care se termină cu un nod-scop.

Implementarea în Prolog a căutării în adâncime iterative este următoarea:

```
cale(Nod,Nod,[Nod]).
cale(PrimNod,UltimNod,[UltimNod|Drum]):-
      cale(PrimNod,PenultimNod,Drum),
      s(PenultimNod,UltimNod),
      \+(membru(UltimNod,Drum)).
depth_first_iterative_deepening(Nod,Sol):-
      cale(Nod,NodScop,Sol),
      scop(NodScop), !.
```

Prezentăm, în continuare, programul Prolog complet corespunzător aceluiaşi exemplu dat de Fig. 2.2:

Programul 2.5

```
scop(f).     % specificare noduri-scop
scop(j).
s(a,b).      % descrierea funcţiei succesor
s(a,c).
s(b,d).
s(d,h).
s(b,e).
s(e,i).
s(e,j).
s(c,f).
s(c,g).
s(f,k).

membru(H,[H|T]).
membru(X,[H|T]):-membru(X,T).

cale(Nod,Nod,[Nod]).
cale(PrimNod,UltimNod,[UltimNod|Drum]):-
      cale(PrimNod,PenultimNod,Drum),
      s(PenultimNod,UltimNod),
      \+(membru(UltimNod,Drum)).
depth_first_iterative_deepening(Nod,Sol):-
```

```
    cale(Nod,NodScop,Sol),
    scop(NodScop),!.
```

Interogarea Prologului se face astfel:

```
?- depth_first_iterative_deepening(a,Sol).
```

Răspunsul Prologului va fi:

```
Sol=[f,c,a] ? ;
no
```

Programul găseşte soluţia cel mai puţin adâncă, sub forma unui drum scris în ordine inversă, după care opreşte căutarea. El va funcţiona la fel de bine şi într-un spaţiu al stărilor conţinând cicluri, datorită mecanismului de verificare `\+(membru(UltimNod,Drum))`, care evită luarea în consideraţie a nodurilor deja vizitate.

Principalul avantaj al acestei metode este acela că ea necesită puţină memorie. La orice moment al execuţiei, necesităţile de spaţiu se reduc la *un singur drum*, acela dintre nodul de început al căutării şi nodul curent.

Dezavantajul metodei este acela că, la fiecare iteraţie, drumurile calculate anterior sunt recalculate, fiind extinse până la o nouă limită de adâncime. Timpul calculator nu este însă foarte afectat, deoarece nu se extind cu mult mai multe noduri.

Pentru un exemplu mai complex de problemă a cărei rezolvare necesită o căutare iterativă în adâncime, vezi §2.2.4.

2.2.4. Strategii de căutare neinformată. Un exemplu

Un exemplu de problemă care poate fi rezolvată prin implementarea oricăreia dintre principalele strategii de căutare neinformată este "Problema mutării blocurilor". Programul care urmează implementează succesiv tehnicile de căutare breadth-first, depth-first cu detectarea ciclurilor şi respectiv iterative deepening, folosind predicatele descrise anterior.

- **Problema Mutarii Blocurilor**

Să considerăm că avem la dispoziţie M blocuri, depozitate pe un teren. Pe suprafaţa terenului există un număr de N locaţii de depozitare. Pentru o mai bună utilizare a spaţiului, blocurile pot fi aşezate unele peste altele, în stive. În fiecare locaţie de depozitare există câte o astfel de stivă, eventual vidă. Un bloc

poate fi mutat din locul său numai dacă el se află în vârful unei stive şi poate fi aşezat numai deasupra unei alte stive (eventual vide).

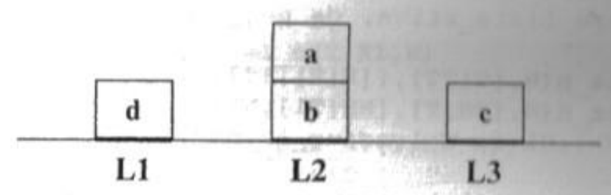
Pentru două configuraţii date, C1 şi C2, ale aşezării celor M blocuri în cele N locaţii, să se stabilească dacă şi cum este posibil să se ajungă în configuraţia C2, plecând din configuraţia C1.

Rezolvare:

În Prolog reprezentăm o stare a problemei (o configuraţie) prin intermediul unei liste de liste C. Fiecare listă din C corespunde unei stive de blocuri şi este ordonată astfel încât blocul din vârful stivei corespunde capului listei. Stivele vide sunt reprezentate prin liste vide. În cazul exemplului din Fig. 2.6, termenii Prolog corespunzători lui C1, respectiv C2 sunt: [[d],[a,b],[c]] şi [[ ],[a,b,c,d],[ ]].

M = 4, N = 3

C1:



C2:

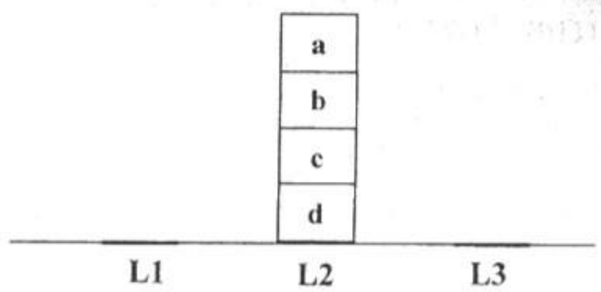


Fig 2.6

Programul 2.6

```
%Predicatele specifice problemei mutarii blocurilor
%Definim relatia de succesiune

s(Lista_stive,Lista_stive_rez):-
```

```
        membru(X,Lista_stive),
        X=[Varf|_],
        det_poz_el(Lista_stive,N,X),
        sterg_la_n(Lista_stive,Lista_stive_inter,N),
        membru(Y,Lista_stive),
        det_poz_el(Lista_stive,N1,Y),N1\==N,
        adaug_la_n(Varf,Lista_stive_inter,
                   Lista_stive_rez,N1),
        \+(permutare(Lista_stive,Lista_stive_rez)).

%Predicatul sterg_la_n(Lista_stive,N,X) este folosit
%pentru stergerea capului stivei de pe pozitia N din lista
%de stive Lista_stive.

sterg_la_n([[_|T]|TT],[T|TT],1).
sterg_la_n([H|T],[H|L],N):-
      N>1,N1 is N-1,sterg_la_n(T,L,N1).

%Predicatul adaug_la_n(Lista_stive,N,X) este folosit
%pentru adaugarea elementului X in capului stivei din lista
%de stive Lista_stive, de pe pozitia N.

adaug_la_n(H,[T|TT],[[H|T]|TT],1).
adaug_la_n(H,[HH|T],[HH|T1],N):-
      N>1,N1 is N-1,adaug_la_n(H,T,T1,N1).

%Prin intermediul predicatelor initial si scop definim
%starile initiale si starile finale.

initial([[d],[a,b],[c]]).

scop([[],[a,b,c,d],[]]).

%Cautare de tip breadth-first

%Predicatul rezolva_b(NodInitial,Solutie) este adevarat
%daca Solutie este un drum (in ordine inversa) de la nodul
%NodInitial la o stare scop, drum obtinut folosind cautarea
%de tip breadth-first.

rezolva_b(NodInitial,Solutie):-
      breadthfirst([[NodInitial]],Solutie).

%Predicatul breadthfirst(Drumuri,Solutie) este adevarat
%daca un drum din multimea de drumuri candidate numita
%"Drumuri" poate fi extins la o stare scop; Solutie este un
%asemenea drum.
```



```
breadthfirst([[Nod|Drum]|_],[Nod|Drum]):-scop(Nod).
breadthfirst([Drum|Drumuri],Solutie):-
            extinde(Drum,DrumNoi),
            concat(Drumuri,DrumNoi,Drumuri1),
            breadthfirst(Drumuri1,Solutie).
extinde([Nod|Drum],DrumNoi):-
            bagof([NodNou,Nod|Drum],
            (s(Nod,NodNou),
            \+(membru(NodNou,[Nod|Drum]))),DrumNoi),
            !.
extinde(_,[]).

%Cautare de tip depth-first cu mecanism de detectare a
%ciclurilor

%Predicatul rezolva(Nod,Solutie) este adevarat daca
%Solutie este un drum aciclic (in ordine inversa) intre
%nodul Nod si o stare scop.

rezolva(Nod,Solutie):-depthfirst([],Nod,Solutie).

%Predicatul depthfirst(Drum,Nod,Solutie) este adevarat
%daca, extinzand calea [Nod|Drum] catre o stare scop,
%obtinem drumul Solutie. Semnificatia argumentelor sale
%este: Nod este o stare de la care trebuie gasita o cale
%catre o stare scop, Drum este o cale (o lista de noduri)
%intre starea initiala si Nod, Solutie este Drum extins via
%Nod catre o stare scop.

depthfirst(Drum,Nod,[Nod|Drum]):-scop(Nod).
depthfirst(Drum,Nod,Solution):-
            s(Nod,Nod1),
            \+ (membru(Nod1,Drum)),
            depthfirst([Nod|Drum],Nod1,Solution).

%Cautare de tip iterative deepening

%Predicatul cale(Nod1,Nod2,Drum) este adevarat daca Drum
%este o cale aciclica intre nodurile Nod1 si Nod2, in
%spatiul starilor; calea Drum este reprezentata ca o lista
%de noduri in ordine inversa.

cale(Nod,Nod,[Nod]).
cale(PrimNod,UltimNod,[UltimNod|Drum]):-
            cale(PrimNod,PenultimNod,Drum),
            s(PenultimNod,UltimNod),
            \+(membru(UltimNod,Drum)).
```

```
depth_first_iterative_deepening(Nod,Solutie):-
            cale(Nod,NodScop,Solutie),
            scop(NodScop),!.

pb_bf:-tell('C:\\bloc_mut_ies_bf.txt'),
      initial(S),rezolva_b(S,Solutie),
      afisare(Solutie),told.

pb_df:-tell('C:\\bloc_mut_ies_df.txt'),
      initial(S),rezolva(S,Solutie),
      afisare(Solutie),told.

pb_df_id:-tell('C:\\block_mut_ies_df_id_.txt'),
          initial(S),
          depth_first_iterative_deepening(S,Solutie),
          afisare(Solutie),told.

%Predicatul lung(L,N) este adevarat daca N este lungimea
%listei L.

lung([],0).
lung([_|T],N):-lung(T,N1),
               N is N1+1.

max(A,B,B):-B>=A,!.
max(A,_,A).

%Predicatul max_lung(L,N) este utilizat pentru
%determinarea celei mai mari lungimi a unei stive din
%lista de stive L.

max_lung([],0).
max_lung([H|T],N):-
          max_lung(T,N1),
          lung(H,N2),
          max(N1,N2,N).

%Predicatul arata(Situatie) este utilizat pentru afisarea
%configuratiei Situatie.

arata(L):-nl,
          max_lung(L,N),
          afis(L,N,N).

%Determinam mai intai cea mai mare lungime a unei stive din
%L si afisam nivel cu nivel, incepand de la ultimul,
%blocurile din stivele aflate in lista L.
```



```
afis(L1,_,0):-af(L1).
afis(L1,N,K):-K>0,
              afis2(L1,N,K),
              nl,
              K1 is K-1,
              afis(L1,N,K1).

af([]):-write(==).
af([_|T]):-write(===),
           af(T).

afis2([],_,_).
afis2([H|T],N,K):-lung(H,L),
          (L>=K,L1 is L-K+1,det_el_n(H,L1,R),write(R);
          L<K,tab(1)),
          tab(2),
          afis2(T,N,K).

det_el_n([H|_],1,H).
det_el_n([_|T],K,H):-K>1,
                     K1 is K-1,
                     det_el_n(T,K1,H).

det_poz_el([H|_],1,H).
det_poz_el([_|T],K,H):-det_poz_el(T,K1,H),
                       K is K1+1.

sterg(X,[X|L],L).
sterg(X,[Y|L],[Y|L1]):-sterg(X,L,L1).

permutare([],[]).
permutare([H|T],L):-permutare(T,T1),
                    sterg(H,L,T1).

membru(X,[X|_]).
membru(X,[_|Y]):-membru(X,Y).

concat([],L,L).
concat([H|T],L,[H|T1]):-concat(T,L,T1).

afisare([]).
afisare([H|T]):-afisare(T),
                arata(H),
                nl.
```

Interogarea Prologului se face apelând predicatele **pb_bf** dacă se dorește o căutare de tip breadth-first, **pb_df** pentru o căutare de tip depth-first cu mecanism de detectare a ciclurilor și respectiv **pb_df_id** pentru o căutare de tip iterative deepening. Iată un exemplu de execuție:

```
?- pb_df_id.
```

Răspunsul sistemului va fi

```
yes
```

cu semnificația că predicatul a fost satisfăcut. În fișierul `C:\block_mut_ies_df_id_.txt` putem vedea secvența de mutări prin care s-a ajuns din configurația inițială în configurația finală. După interogarea anterioară, conținutul fișierului `C:\block_ies_bf.txt` va fi următorul:

```
  a
d b c
==========
 a d
 b c
===========
   d
a b c
==========
b   d
a   c
===========
b
a d c
==========
b c
a d
==========
```



```
  b
  c
a d
===========
 a
 b
 c
 d
==========
```

2.3. Căutarea informată

Căutarea informată se mai numește și *căutare euristică*. Euristica este o metodă de studiu și de cercetare bazată pe descoperirea de fapte noi. În acest tip de căutare vom folosi informația despre spațiul de stări. Se folosesc cunoștințe specifice problemei și se rezolvă probleme de optim.

2.3.1. Căutarea de tip best-first

Tipul de căutare pe care îl discutăm aici se aseamănă cu tehnica breadth-first, numai că procesul nu se desfășoară în mod uniform plecând de la nodul inițial. El înaintează în mod preferențial de-a lungul unor noduri pe care informația euristică, specifică problemei, le indică ca aflându-se pe drumul cel mai bun către un scop. Un asemenea proces de căutare se numește ***căutare euristică*** sau ***căutare de tip best-first***.

Principiile pe care se bazează căutarea de tip best-first sunt următoarele:

1. Se presupune existența unei funcții euristice de evaluare, $\hat{f}$, cu rolul de a ne ajuta să decidem care nod ar trebui extins la pasul următor. Se va adopta *convenția* că valori mici ale lui $\hat{f}$ indică nodurile cele mai bune. Această funcție se bazează pe informație specifică domeniului pentru care s-a formulat problema. Este o funcție de descriere a stărilor, cu valori reale.
2. Se extinde nodul cu cea mai mică valoare a lui $\hat{f}(n)$. *În cele ce urmează, se va presupune că extinderea unui nod va produce toți succesorii acelui nod.*
3. Procesul se încheie atunci când următorul nod care ar trebui extins este un nod-scop.

Fig. 2.7 ilustrează începutul unei căutări de tip best-first:

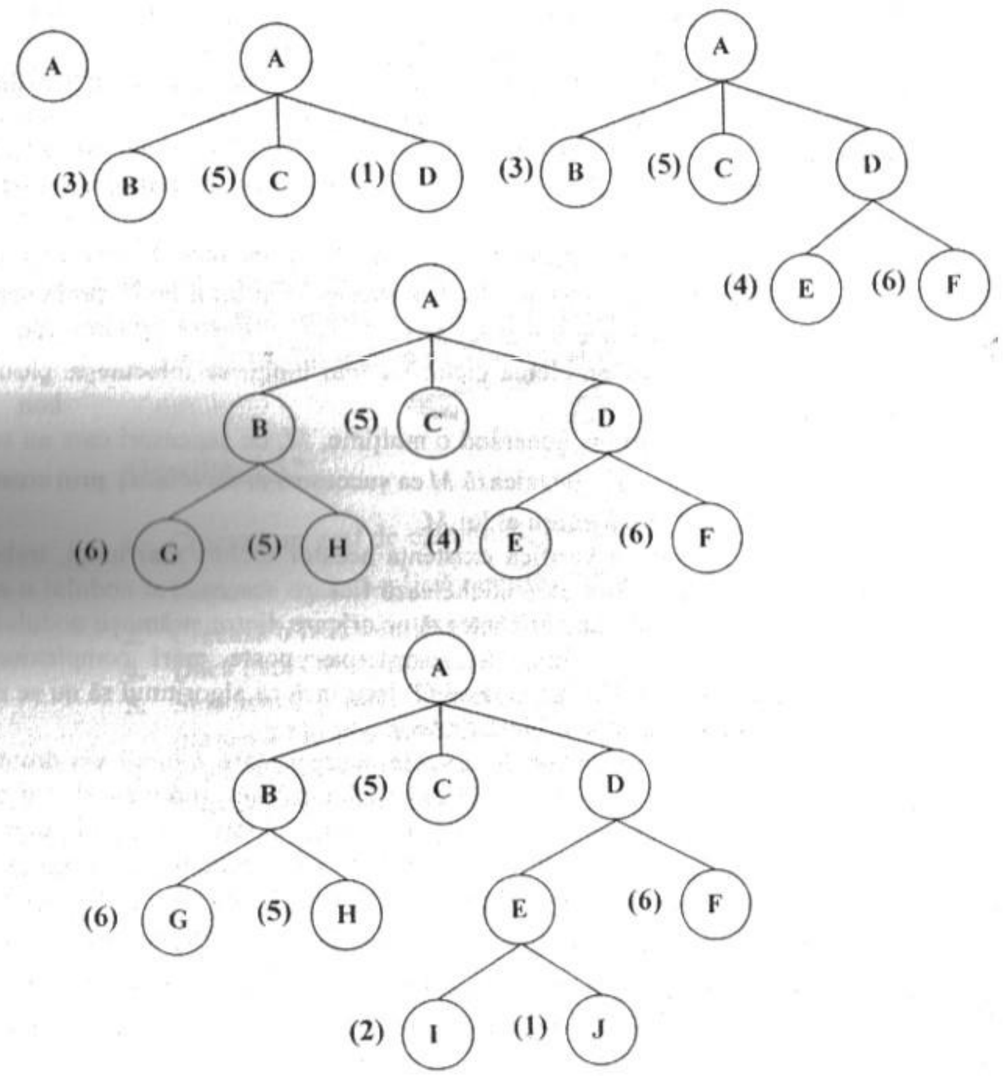


Fig. 2.7

În Fig. 2.7 există inițial un singur nod, A, astfel încât acesta va fi extins. Extinderea lui generează trei noduri noi. Funcția euristică este aplicată fiecăruia dintre acestea. Întrucât nodul D este cel mai promițător, el este următorul nod extins. Extinderea lui va produce două noduri succesor, E și F, cărora li se aplică funcția euristică. În acest moment, un alt drum, și anume acela care trece

prin nodul B, pare mai promițător, astfel încât el este urmat, generându-se nodurile G și H. În urma evaluării, aceste noduri par însă mai puțin promițătoare decât un alt drum, astfel încât este ales, în continuare, drumul prin D la E. E este apoi extins producând nodurile I și J. La pasul următor va fi extins nodul J, întrucât acesta este cel mai promițător. Procesul continuă până când este găsită o soluție.

Pentru a nu fi induși în eroare de o euristică extrem de optimistă, este necesar să înclinăm căutarea în favoarea posibilității de a ne întoarce înapoi, cu scopul de a explora drumuri găsite mai devreme. De aceea, vom adăuga lui $\hat{f}$ *un factor de adâncime*: $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$, unde $\hat{g}(n)$ este *o estimație a adâncimii lui n în graf*, adică reprezintă lungimea celui mai scurt drum de la nodul de start la n, iar $\hat{h}(n)$ este *o evaluare euristică a nodului n*.

Pentru a studia, în cele ce urmează, *aspectele formale ale căutării de tip best-first*[14], vom începe prin a prezenta un algoritm de căutare generală bazat pe grafuri. Algoritmul este preluat din [25] și include versiuni ale căutării de tip best-first ca reprezentând cazuri particulare.

2.3.2. Algoritm de căutare general bazat pe grafuri

Acest algoritm, pe care îl vom numi GraphSearch, este unul general, care permite orice tip de ordonare preferată de utilizator - euristică **sau** neinformată. Iată o *primă variantă* a definiției sale:

GraphSearch

1. Creează un arbore de căutare, T_r, care constă numai din nodul de start n_0. Plasează pe n_0 într-o listă ordonată numită OPEN.
2. Creează o listă numită CLOSED, care inițial este vidă.
3. Dacă lista OPEN este vidă, EXIT cu eșec.
4. Selectează primul nod din OPEN, înlătură-l din lista OPEN și include-l în lista CLOSED. Numește acest nod n.
5. Dacă n este un nod-scop, algoritmul se încheie cu succes, iar soluția este cea obținută prin urmarea în sens invers a unui drum de-a lungul arcelor din arborele T_r, de la n la n_0. (Arcele sunt create la pasul 6).

[14] prezentate, aici, conform [25]

6. Extinde nodul n, generând o mulțime, M, de succesori. Include M ca succesori ai lui n în T_r, prin crearea de arce de la n la fiecare membru al mulțimii M.
7. Reordonează lista OPEN, fie în concordanță cu un plan arbitrar, fie în mod euristic.
8. Mergi la pasul 3.

❑

- **Observație:** Acest algoritm poate fi folosit pentru a efectua căutări de tip best-first, breadth-first sau depth-first. În cazul algoritmului *breadth-first* noile noduri sunt puse la sfârșitul listei OPEN (organizată ca o coadă), iar nodurile nu sunt reordonate. În cazul căutării de tip *depth-first* noile noduri sunt plasate la începutul listei OPEN (organizată ca o stivă). În cazul căutării de tip *best-first*, numită și căutare euristică, lista OPEN este reordonată în funcție de meritele euristice ale nodurilor.

2.3.2.1. Algoritmul A*

Vom particulariza algoritmul GraphSearch la un algoritm de căutare best-first care reordonează, la pasul 7, nodurile listei OPEN în funcție de *valorile crescătoare ale funcției* $\hat{f}$. Această versiune a algoritmului GraphSearch se va numi Algoritmul A*.

Pentru a specifica familia funcțiilor $\hat{f}$ care vor fi folosite, introducem următoarele notații:

- $h(n)$ = costul *efectiv* al drumului de cost minim dintre nodul n și un nod-scop, luând în considerație toate nodurile-scop posibile și toate drumurile posibile de la n la ele;
- $g(n)$ = costul unui drum de cost minim de la nodul de start n_0 la nodul n.

Atunci, $f(n) = g(n) + h(n)$ este costul unui drum de cost minim de la n_0 la un nod-scop, drum ales dintre toate drumurile care trebuie să treacă prin nodul n.

- **Observație:** $f(n_0) = h(n_0)$ reprezintă costul unui drum de cost minim nerestricționat, de la nodul n_0 la un nod-scop.

Pentru fiecare nod n, fie $\hat{h}(n)$, numit *factor euristic*, o estimație a lui $h(n)$ și fie $\hat{g}(n)$, numit *factor de adâncime*, costul drumului de cost minim

până la n găsit de A* până la pasul curent. Algoritmul A* va folosi funcția $\hat{f} = \hat{g} + \hat{h}$.

Așa cum se remarcă în [25], în definirea Algoritmului A* de până acum nu s-a ținut cont de următoarea *problemă*: ce se întâmplă dacă graful implicit în care se efectuează căutarea nu este un arbore? Cu alte cuvinte, există mai mult decât o unică secvență de acțiuni care pot conduce la aceeași stare a lumii plecând din starea inițială. (Există situații în care fiecare dintre succesorii nodului n îl are pe n ca succesor i.e. acțiunile sunt reversibile). Pentru a rezolva astfel de cazuri [25], pasul **6** al algoritmului GraphSearch trebuie înlocuit cu următorul pas **6'** :

6'. Extinde nodul n, generând o mulțime, M, de succesori care nu sunt deja *părinți* ai lui n în T_r. Instalează M ca succesori ai lui n în T_r prin crearea de arce de la n la fiecare membru al lui M.

Pentru a rezolva problema ciclurilor mai lungi, se înlocuiește pasul **6** prin următorul pas **6"**:

6". Extinde nodul n, generând o mulțime, M, de succesori care nu sunt deja *strămoși* ai lui n în T_r. Instalează M ca succesori ai lui n în T_r prin crearea de arce de la n la fiecare membru al lui M.

Desigur, pentru a verifica existența acestor cicluri mai lungi, trebuie văzut dacă structura de date care etichetează fiecare succesor al nodului n este egală cu structura de date care etichetează pe oricare dintre strămoșii nodului n. Pentru structuri de date complexe, acest pas poate mări complexitatea algoritmului. Pasul **6** modificat în pasul **6"** face însă ca algoritmul să nu se mai învârtă în cerc, în căutarea unui drum la scop.

Există încă posibilitatea de a vizita aceeași stare a lumii via drumuri diferite. O modalitate de a trata această problemă este ignorarea ei. Cu alte cuvinte, algoritmul nu verifică dacă un nod din mulțimea M se află deja în listele OPEN sau CLOSED. Algoritmul uită deci posibilitatea de a ajunge în aceleași noduri urmând drumuri diferite. Acest "același nod" s-ar putea repeta în T_r de atâtea ori de câte ori algoritmul descoperă drumuri diferite care duc la el. Dacă două noduri din T_r sunt etichetate cu aceeași structură de date, vor avea sub ele subarbori identici. Prin urmare, algoritmul va duplica anumite eforturi de căutare.

Pentru a preveni duplicarea efortului de căutare atunci când nu s-au impus condiții suplimentare asupra lui $\hat{f}$, sunt necesare niște modificări în algoritmul A*, și anume: deoarece căutarea poate ajunge la același nod de-a lungul unor drumuri diferite, algoritmul A* generează un ***graf de căutare***, notat cu G. G este structura de noduri și de arce generată de A* pe măsură ce

algoritmul extinde nodul inițial, succesorii lui ș.a.m.d.. A* menține și un arbore de căutare, T_r.

T_r, un subgraf al lui G, este arborele cu cele mai bune drumuri (de cost minim) produse până la pasul curent, drumuri până la toate nodurile din graful de căutare. Prin urmare, unele drumuri pot fi în graful de căutare, dar nu și în arborele de căutare. Graful de căutare este menținut deoarece căutări ulterioare pot găsi drumuri mai scurte, care folosesc anumite arce din graful de căutare anterior ce nu se aflau și în arborele de căutare anterior.

Dăm, în continuare, versiunea algoritmului A* care menține graful de căutare. *În practică*, această versiune este folosită mai rar deoarece, de obicei, se pot impune *condiții asupra lui* $\hat{f}$ care garantează faptul că, atunci când algoritmul A* extinde un nod, el a găsit deja drumul de cost minim până la acel nod.

Algoritmul A*

1. Creează un graf de căutare G, constând numai din nodul inițial n_0. Plasează n_0 într-o listă numită OPEN.
2. Creează o listă numită CLOSED, care inițial este vidă.
4. Dacă lista OPEN este vidă, EXIT cu eșec.
5. Selectează primul nod din lista OPEN, înlătură-l din OPEN și plasează-l în lista CLOSED. Numește acest nod n.
5. Dacă n este un nod-scop, oprește execuția cu succes. Returnează soluția obținută urmând un drum de-a lungul pointerilor de la n la n_0 în G. (Pointerii definesc un arbore de căutare și sunt stabiliți la pasul 7).
6. Extinde nodul n, generând o mulțime, M, de succesori ai lui care nu sunt deja strămoși ai lui n în G. Instalează acești membri ai lui M ca succesori ai lui n în G.
7. Stabilește un pointer către n de la fiecare dintre membrii lui M care nu se găseau deja în G (adică nu se aflau deja nici în OPEN, nici în CLOSED). Adaugă acești membri ai lui M listei OPEN. Pentru fiecare membru, m, al lui M, care se află deja în OPEN sau în CLOSED, redirecționează pointerul său către n, dacă cel mai bun drum la m găsit până în acel moment trece prin n. Pentru fiecare membru al lui M care se află deja în lista CLOSED, redirecționează pointerii fiecăruia dintre descendenții săi din G astfel încât aceștia să țintească înapoi de-a lungul celor mai bune drumuri până la acești descendenți, găsite până în acel moment.

8. Reordonează lista OPEN în ordinea valorilor crescătoare ale funcției $\hat{f}$. (Eventuale legături între valori minimale ale lui $\hat{f}$ sunt rezolvate în favoarea nodului din arborele de căutare aflat la cea mai mare adâncime).
9. Mergi la pasul 3.

□

- **Observație:** La pasul 7 sunt redirecționați pointeri de la un nod dacă procesul de căutare descoperă un drum la acel nod care are costul mai mic decât acela indicat de pointerii existenți. Redirecționarea pointerilor descendenților nodurilor care deja se află în lista CLOSED economisește efortul de căutare, dar poate duce la o cantitate exponențială de calcule. De aceea, această parte a pasului 7 de obicei nu este implementată. Unii dintre acești pointeri vor fi până la urmă redirecționați oricum, pe măsură ce căutarea progresează.

2.3.2.1.1. Admisibilitatea Algoritmului A*

Există anumite condiții asupra grafurilor și a lui $\hat{h}$ care garantează că algoritmul A*, aplicat acestor grafuri, găsește întotdeauna drumuri de cost minim. Condițiile asupra *grafurilor* sunt:

1. Orice nod al grafului, dacă admite succesori, are un număr finit de succesori.
2. Toate arcele din graf au costuri mai mari decât o cantitate pozitivă, ε.

Condiția asupra lui $\hat{h}$ este:

3. Pentru toate nodurile n din graful de căutare, $\hat{h}(n) \le h(n)$. Cu alte cuvinte, $\hat{h}$ nu supraestimează niciodată valoarea efectivă h. O asemenea funcție $\hat{h}$ este uneori numită un *estimator optimist*.

- **Observații:**

1. Este relativ ușor să se găsească, în probleme, o funcție $\hat{h}$ care satisface această *condiție a limitei de jos*. De exemplu, în probleme de găsire a drumurilor în cadrul unor grafuri ale căror noduri sunt orașe, distanța de tip linie dreaptă de la un oraș n la un oraș-scop constituie o limită inferioară asupra distanței reprezentând un drum optim de la nodul n la nodul-scop.

2. Cu cele trei condiţii formulate anterior, algoritmul A* garantează găsirea unui drum optim la un scop, în cazul în care există un drum la scop.

În cele ce urmează, formulăm acest rezultat sub forma unei teoreme:

Teorema 2.1

Atunci când sunt îndeplinite condiţiile asupra grafurilor şi asupra lui $\hat{h}$ enunţate anterior şi cu condiţia să existe un drum de cost finit de la nodul iniţial, n_0, la un nod-scop, algoritmul A* garantează găsirea unui drum de cost minim la un scop.

Demonstraţia teoremei:

Cea mai importantă componentă a demonstraţiei este următoarea lemă:

Lema 2.1

Înainte de terminarea Algoritmului A*, la fiecare pas, există întotdeauna un nod, de exemplu n^*, în lista OPEN, cu următoarele proprietăţi:

1. n^* este pe un drum optim la un scop.
2. A* a găsit un drum optim până la n^*.
3. $\hat{f}(n^*) \le f(n_0)$.

Presupunând această lemă ca fiind demonstrată (vezi p. 67), continuăm **demonstraţia teoremei**: vom arăta că algoritmul A* se termină dacă există un scop accesibil şi, mai mult, că el se termină prin găsirea unui drum optim la un scop.

Arătăm că A se termină*: să presupunem că algoritmul nu se termină. În acest caz, A* continuă să extindă noduri la infinit şi, la un moment dat, începe să extindă noduri la o adâncime mai mare în arborele de căutare decât orice limitare finită a adâncimii. (S-a presupus că graful în care se face căutarea are factor de ramificare finit). Întrucât costul fiecărui arc este mai mare decât $\varepsilon > 0$, valorile lui $\hat{g}$ (şi, prin urmare, şi cele ale lui $\hat{f}$) ale tuturor nodurilor din OPEN vor depăşi, până la urmă, pe $f(n_0)$. Dar acest lucru contrazice Lema 2.1.

Arătăm că A se termină cu găsirea unui drum optim*: A* se poate termina numai la pasul 3 (dacă lista OPEN este vidă) sau la pasul 5 (ajungându-se într-un nod-scop).

O terminare la pasul 3 poate interveni numai în cazul unor grafuri finite care nu conţin nici un nod-scop, iar teorema afirmă că este găsit un drum optim la un scop numai dacă un nod-scop există. Prin urmare, A* se termină prin găsirea unui nod-scop.

Să presupunem acum că A* se termină prin găsirea unui scop care nu este optim, de exemplu prin găsirea lui n_{g2} cu $f(n_{g2}) > f(n_0)$ şi în condiţiile în care există un scop optim, $n_{g1} \ne n_{g2}$, cu $f(n_{g1}) = f(n_0)$. Atunci când are loc terminarea în nodul n_{g2}, $\hat{f}(n_{g2}) \ge f(n_{g2}) > f(n_0)$. Dar, chiar înainte ca A* să îl selecteze pe n_{g2} pentru extindere, conform Lemei 2.1, a existat un nod n^* în OPEN şi aflat pe un drum optim, cu $\hat{f}(n^*) \le f(n_0)$. Prin urmare, A* nu ar fi putut să îl selecteze pe n_{g2} pentru extindere, deoarece A* selectează întotdeauna acel nod având cea mai mică valoare a lui $\hat{f}$ şi $\hat{f}(n^*) \le f(n_0) < f(n_{g2})$. Deci teorema a fost demonstrată. □

Demonstraţia lemei:

Demonstraţia Lemei 2.1 se va face prin inducţie. Pentru a demonstra că la fiecare pas al lui A* concluziile lemei sunt valabile, este suficient să demonstrăm că:

(1) sunt valabile la începutul algoritmului;
(2) dacă sunt valabile înainte de extinderea unui nod, vor continua să fie valabile şi după extinderea nodului.

(1) *Cazul de bază*: la începutul căutării, când numai nodul n_0 a fost selectat pentru extindere, nodul n_0 este în OPEN, el este un drum optim la scop şi A* a găsit acest drum. De asemenea, $\hat{f}(n_0) \le f(n_0)$ deoarece $\hat{f}(n_0) = \hat{h}(n_0) \le f(n_0)$. Astfel, nodul n_0 poate fi, în acest stadiu, nodul n^* al lemei.

(2) *Pasul de inducţie*: presupunem adevărate concluziile lemei la momentul la care au fost extinse m noduri ($m \ge 0$) şi, folosind această presupunere, arătăm că ele sunt adevărate la momentul la care au fost extinse $m+1$ noduri.

Fie n^* nodul din ipoteză şi din lista OPEN, nod care se află pe un drum optim găsit de algoritmul A* după ce au fost extinse m noduri.

Cazul I. Dacă n^* *nu* este selectat pentru extindere la pasul $m+1$, n^* are aceleaşi proprietăţi pe care le avea înainte, deci pasul de inducţie este

demonstrat în acest caz. Fig. 2.8 corespunde acestei situaţii şi prezintă Algoritmul A* selectând nodul n_l ca fiind al $(m+1)$-lea nod extins:

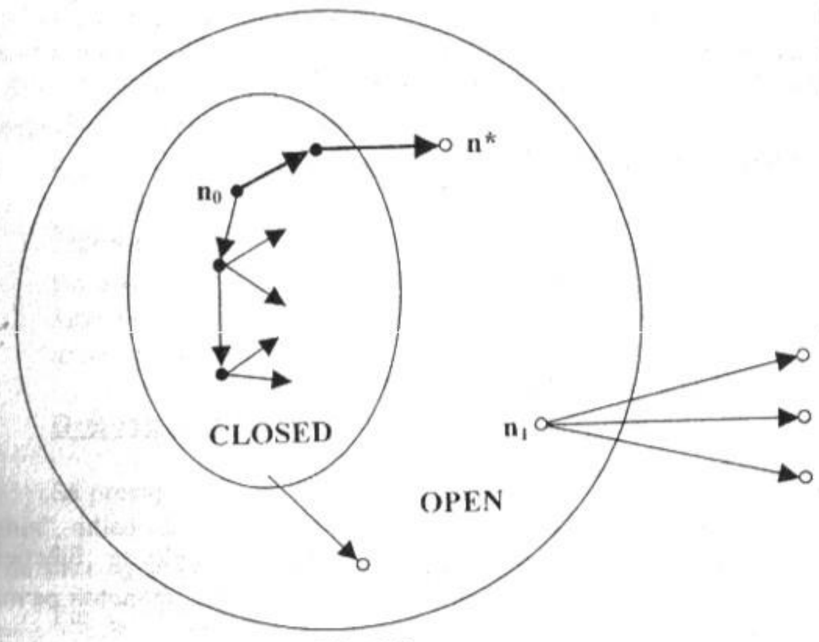


Fig. 2.8

Cazul II. Dacă n^* este selectat pentru extindere, toţi succesorii săi noi vor fi puşi în lista OPEN. Cel puţin unul dintre ei, să presupunem n_p, va fi pe un drum optim la un scop (deoarece, prin ipoteză, un drum optim trece prin n^* şi deci trebuie să continue prin unul dintre succesorii săi). A* a găsit un drum optim la n_p deoarece, dacă ar exista un drum mai bun la n_p, acel drum mai bun ar reprezenta şi un drum mai bun către scop, contrazicând ipoteza că nu există un drum mai bun către scop decât cel găsit de A* ca trecând prin n^*. Deci, în acest caz, permitem lui n_p să fie noul n^* pentru pasul al $(m+1)$-lea. Fig. 2.9 ilustrează situaţia în care A* selectează pe n^* ca fiind al $(m+1)$-lea nod care se extinde. Pasul de inducţie este demonstrat, *cu excepţia proprietăţii* $\hat{f}(n_0) \le f(n_0)$.

Demonstrăm acum proprietatea $\hat{f}(n_0) \le f(n_0)$ pentru toţi paşii m dinaintea terminării algoritmului. Pentru orice nod, n^*, aflat pe un drum optim şi până la care A* a găsit un drum optim, avem:

$\hat{f}(n^*) = \hat{g}(n^*) + \hat{h}(n^*) \le$ (prin definiţie)

$\le g(n^*) + h(n^*) \le$ deoarece $\begin{cases} \hat{g}(n^*) = g(n^*) \\ \hat{h}(n^*) \le h(n^*) \end{cases}$

$\le f(n^*) \le$ deoarece $g(n^*) + h(n^*) = f(n^*)$ prin definiţie

$\le f(n_0)$ deoarece $f(n^*) = f(n_0)$, întrucât n^* se află pe un drum optim, *ceea ce completează demonstraţia lemei.*

□

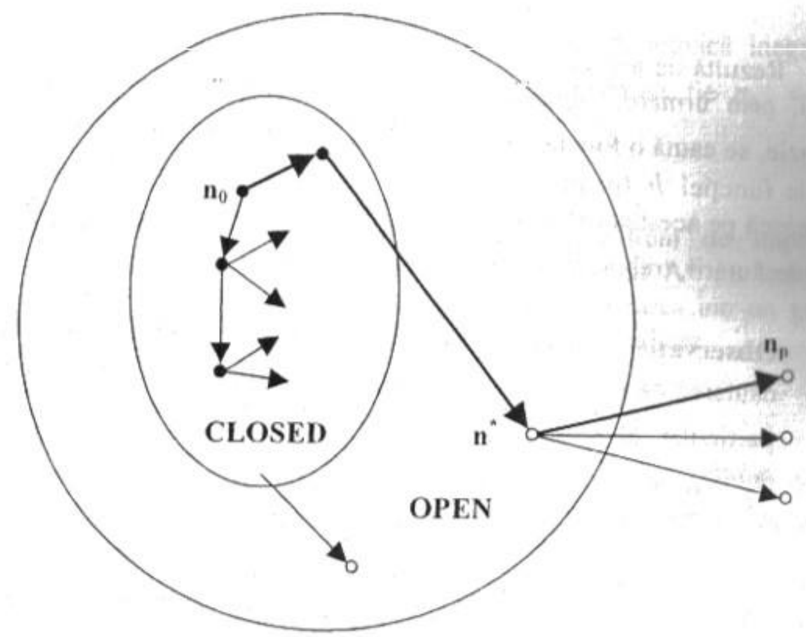


Fig. 2.9

Definiţia 2.1
Orice algoritm care garantează găsirea unui drum optim la scop este un algoritm *admisibil*.

Prin urmare, atunci când cele trei condiţii ale Teoremei 2.1 sunt îndeplinite, A* este un algoritm admisibil. Prin extensie, vom spune că orice funcţie $\hat{h}$ care nu supraestimează pe h este *admisibilă*.

În cele ce urmează, atunci când ne vom referi la Algoritmul A*, vom presupune că cele trei condiţii ale Teoremei 2.1 sunt verificate.

Dacă *două versiuni ale lui A**, A*$_1$ şi A*$_2$, diferă între ele numai prin aceea că $\hat{h}_1 < \hat{h}_2$ pentru toate nodurile care nu sunt noduri-scop, vom spune că A*$_2$ *este mai informat decât* A*$_1$. Referitor la această situaţie, formulăm următoarea teoremă, fără a intra în detaliile demonstrării ei:

Teorema 2.2

Dacă algoritmul A*$_2$ este mai informat decât A*$_1$, atunci la terminarea căutării pe care cei doi algoritmi o efectuează asupra oricărui graf având un drum de la n_0 la un nod-scop, fiecare nod extins de către A*$_2$ este extins şi de către A*$_1$.

❑

Rezultă de aici că A*$_1$ extinde cel puţin tot atâtea noduri câte extinde A*$_2$ şi, prin urmare, algoritmul mai informat A*$_2$ este şi mai eficient. În concluzie, se caută o funcţie $\hat{h}$ ale cărei valori sunt cât se poate de apropiate de cele ale funcţiei h (pentru o cât mai mare eficienţă a căutării), dar fără să le depăşească pe acestea (pentru admisibilitate). Desigur, pentru a evalua eficienţa *totală* a căutării, trebuie luat în consideraţie şi costul calculării lui $\hat{h}$.

- **Observaţie:** Atunci când $\hat{f}(n) = \hat{g}(n) =$ adâncime(n), se obţine căutarea de tip *breadth-first*. Algoritmul breadth-first reprezintă un caz particular al lui A* (cu $\hat{h} \equiv 0$), prin urmare el este un algoritm *admisibil*.

2.3.2.1.2. Condiţia de consistenţă

Fie o pereche de noduri (n_i, n_j) astfel încât n_j este un succesor al lui n_i.

Definiţia 2.2

Se spune că $\hat{h}$ *îndeplineşte condiţia de consistenţă* dacă, pentru orice astfel de pereche (n_i, n_j) de noduri din graful de căutare,

$$\hat{h}(n_i) - \hat{h}(n_j) \le c(n_i, n_j),$$

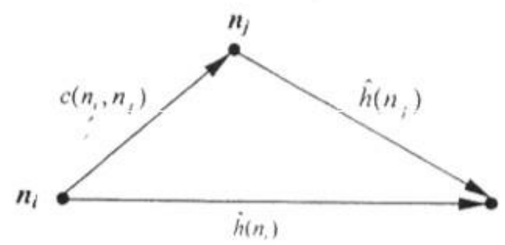
unde $c(n_i, n_j)$ este costul arcului de la n_i la n_j.

Condiţia de consistenţă mai poate fi formulată şi sub una din formele următoare:

$$\hat{h}(n_i) \le \hat{h}(n_j) + c(n_i, n_j) \text{ sau } \hat{h}(n_j) \ge \hat{h}(n_i) - c(n_i, n_j),$$

ceea ce conduce la următoarea *interpretare* a ei: de-a lungul oricărui drum din graful de căutare, estimaţia făcută asupra costului optim rămas pentru a atinge scopul nu poate descreşte cu o cantitate mai mare decât costul arcului de-a lungul acelui drum. Se spune că funcţia euristică este *local consistentă* atunci când se ia în consideraţie costul cunoscut al unui arc.

Condiţia de consistenţă:



$$\hat{h}(n_i) \le c(n_i, n_j) + \hat{h}(n_j)$$

Condiţia de consistenţă implică faptul că valorile funcţiei $\hat{f}$ corespunzătoare nodurilor din arborele de căutare descresc monoton pe măsură ce ne îndepărtăm de nodul de start.

➢ Fie n_i şi n_j două noduri în *arborele de căutare* generat de algoritmul A*, cu n_j succesor al lui n_i. Atunci, dacă condiţia de consistenţă este satisfăcută, avem:

$$\hat{f}(n_j) \ge \hat{f}(n_i).$$

Demonstraţie:

Pentru a demonstra acest fapt, se începe cu condiţia de consistenţă:

$$\hat{h}(n_j) \ge \hat{h}(n_i) - c(n_i, n_j)$$

Se adună apoi $\hat{g}(n_j)$ în ambii membri ai inegalităţii anterioare ($\hat{g}$ este factor de adâncime, adică o estimaţie a adâncimii nodului):

$$\hat{h}(n_j) + \hat{g}(n_j) \ge \hat{h}(n_i) + \hat{g}(n_j) - c(n_i, n_j)$$

Dar $\hat{g}(n_j) = \hat{g}(n_i) + c(n_i, n_j)$, adică adâncimea nodului n_j este adâncimea lui n_i plus costul arcului de la n_i la n_j. Dacă egalitatea nu ar avea loc, n_j nu ar fi un succesor al lui n_i în *arborele* de căutare. Atunci:

$$\hat{h}(n_j)+\hat{g}(n_j)\geq \hat{h}(n_i)+\hat{g}(n_i)+c(n_i,n_j)-c(n_i,n_j),$$

deci $\hat{f}(n_j)\geq \hat{f}(n_i)$.

□

Din această cauză, *condiția de consistență asupra lui $\hat{h}$* este adesea numită *condiție de monotonie asupra lui $\hat{f}$*.

Există următoarea teoremă referitoare la condiția de consistență:

Teorema 2.3

Dacă este satisfăcută condiția de consistență asupra lui $\hat{h}$, atunci, în momentul în care algoritmul A* extinde un nod *n*, el a găsit deja un drum optim până la *n*.

Demonstrație:

Să presupunem că A* se pregătește să extindă un nod *n* (*n* este un "nod deschis", adică un nod din lista OPEN), în căutarea unui drum optim de la un nod de start n_0 la un nod-scop, într-un graf implicit G.

Fie $\xi=(n_0,n_1,.....,n_l,n_{l+1},....,n=n_k)$ o secvență de noduri din G care constituie un drum optim de la n_0 la *n*. Fie n_l ultimul nod din ξ extins de A*. Întrucât n_l este ultimul "nod închis" din ξ, adică ultimul din lista CLOSED, știm că n_{l+1} este în lista OPEN și deci reprezintă un candidat pentru extindere.

Pentru orice nod n_i și succesorul său n_{i+1} din ξ, avem[15]

$$g(n_{i+1})+\hat{h}(n_{i+1})=g(n_i)+c(n_i,n_{i+1})+\hat{h}(n_{i+1})\geq g(n_i)+\hat{h}(n_i),$$

atunci când condiția de consistență este satisfăcută.

Tranzitivitatea relației $\geq$ ne dă:

$$g(n_j)+\hat{h}(n_j)\geq g(n_i)+\hat{h}(n_i) \text{ pentru } \forall n_i,n_j \text{ de pe } \xi \text{ dacă } i<j.$$

În particular,

$$g(n)+\hat{h}(n)\geq g(n_{l+1})+\hat{h}(n_{l+1})=\hat{f}(n_{l+1})$$

deoarece A* a găsit un drum optim până la n_{l+1}, făcând $\hat{g}(n_{l+1})=g(n_{l+1})$.

---

[15] deoarece $g(n_i)+c(n_i,n_{i+1})=g(n_{i+1})$ și din condiția de consistență $\hat{h}(n_i)\leq \hat{h}(n_j)+c(n_i,n_j)$, cu n_j succesor; aici $n_j\equiv n_{i+1}$.

Dar, întrucât A* se pregătește să extindă nodul *n* în locul nodului n_{l+1}, înseamnă că :

$$\hat{f}(n)=\hat{g}(n)+\hat{h}(n)\leq \hat{f}(n_{l+1}) \qquad (1)$$

S-a stabilit deja că

$$\hat{f}(n_{l+1})\leq g(n)+\hat{h}(n) \qquad (2)$$

Din (1) și (2) rezultă

$$\hat{g}(n)+\hat{h}(n)\leq g(n)+\hat{h}(n)$$

Prin urmare,

$$\hat{g}(n)\leq g(n).$$

Dar, întrucât metoda de calcul a funcțiilor $\hat{g}$ implică inegalitatea $\hat{g}(n)\geq g(n)$, înseamnă că $\hat{g}(n)=g(n)$, ceea ce arată că, fie $n_{l+1}=n$, fie a fost deja găsit un alt drum optim până la nodul *n*.

□

- **Observație:** Condiția de consistență este extrem de importantă deoarece, atunci când este satisfăcută, algoritmul A* nu trebuie să redirecționeze niciodată pointeri la pasul 7. Căutarea într-un graf nu diferă atunci prin nimic de căutarea în cadrul unui arbore.

2.3.2.1.3. Optimalitatea Algoritmului A*

Fie G o stare-scop optimală cu un cost al drumului notat f^*. Fie G_2 o a doua stare-scop, suboptimală, care este o stare-scop cu un cost al drumului

$$g(G_2)>f^*$$

Presupunem că A* selectează din coadă, pentru extindere, pe G_2. Întrucât G_2 este o stare-scop, această alegere ar încheia căutarea cu o soluție suboptimală. Vom arăta că acest lucru nu este posibil.

Fie un nod *n*, care este, la pasul curent, un nod frunză pe un drum optim la G. (Un asemenea nod trebuie să existe, în afara cazului în care drumul a fost complet extins, caz în care algoritmul ar fi returnat G). Pentru acest nod *n*, întrucât *h* este admisibilă, trebuie să avem:

$$f^*\geq f(n) \qquad (1)$$

Mai mult, dacă *n* nu este ales pentru extindere în favoarea lui G_2, trebuie să avem:

$$f(n)\geq f(G_2) \qquad (2)$$

Combinând (1) cu (2) obținem:

$$f^*\geq f(G_2) \qquad (3)$$

Dar, deoarece G_2 este o stare-scop, avem $h(G_2) = 0$. Prin urmare,

$$f(G_2) = g(G_2) \quad (4)$$

Cu presupunerile făcute, conform (3) şi (4) am arătat că

$$f^* \geq g(G_2)$$

Această concluzie contrazice faptul că G_2 este suboptimal. Ea arată că A* nu selectează niciodată pentru extindere un scop suboptimal. A* întoarce o soluţie numai după ce a selectat-o pentru extindere, de aici rezultând faptul că *A* este un algoritm optim.*

2.3.2.1.4. Completitudinea Algoritmului A*

Întrucât A* extinde noduri în ordinea valorilor crescătoare ale lui *f*, în final va exista o extindere care conduce la o stare-scop. Acest lucru este adevărat, în afara cazului în care există un număr foarte mare de noduri, număr care tinde la infinit, cu

$$f(n) < f^*.$$

Singura modalitate în care ar putea exista un număr infinit de noduri ar fi aceea în care:

- există *un nod* cu *factor de ramificare infinit*;
- există *un drum* cu un *cost finit*, dar care are un *număr infinit de noduri*. (Acest lucru ar fi posibil conform paradoxului lui Zeno, care vrea să arate că o piatră aruncată spre un copac nu va ajunge niciodată la acesta. Astfel, se imaginează că traiectoria pietrei este împărţită într-un şir de faze, fiecare dintre acestea acoperind jumătate din distanţa rămasă până la copac. Aceasta conduce la un număr infinit de paşi cu un cost total finit).

Prin urmare, *exprimarea corectă* este aceea că A* este complet relativ la *grafuri local finite*, adică grafuri cu un factor de ramificare finit, cu condiţia să existe o constantă pozitivă δ astfel încât fiecare operator să coste cel puţin δ.

2.3.2.1.5. Complexitatea Algoritmului A*

S-a arătat (demonstraţie pe care nu o vom reproduce în cadrul relativ restrâns al acestei lucrări) că o creştere exponenţială va interveni, în afara cazului în care eroarea în funcţia euristică nu creşte mai repede decât logaritmul costului efectiv al drumului. Cu alte cuvinte, *condiţia pentru o creştere subexponenţială* este:

$$\left|h(n) - h^*(n)\right| \leq O(\log h^*(n)),$$

unde $h^*(n)$ este *adevăratul* cost de a ajunge de la *n* la scop.

În afară de timpul mare calculator, algoritmul A* consumă şi mult spaţiu de memorie deoarece *păstrează în memorie toate nodurile generate*.

Algoritmi de căutare mai noi, de tip "memory-bounded" (cu limitare a memoriei), au reuşit să înlăture neajunsul legat de problema spaţiului de memorie folosit, fără a sacrifica optimalitatea sau completitudinea. Unul dintre aceştia este algoritmul IDA*.

2.3.2.2. Iterative Deepening A* (IDA*)

Algoritmul IDA* se referă la o căutare iterativă în adâncime de tip A* şi este o extensie logică a lui Iterative Deepening Search care foloseşte, în plus, informaţia euristică.

În cadrul acestui algoritm fiecare iteraţie reprezintă o căutare de tip depth-first, iar căutarea de tip depth-first este modificată astfel încât ea să folosească *o limită a costului* şi nu o limită a adâncimii.

Faptul că în cadrul algoritmului A* *f* nu descreşte niciodată de-a lungul oricărui drum care pleacă din rădăcină ne permite să trasăm, din punct de vedere conceptual, *contururi* în spaţiul stărilor. Astfel, în interiorul unui contur, toate nodurile au valoarea *f(n)* mai mică sau egală cu o aceeaşi valoare. În cazul algoritmului IDA* fiecare iteraţie extinde toate nodurile din interiorul conturului determinat de costul *f* curent, după care se trece la conturul următor.

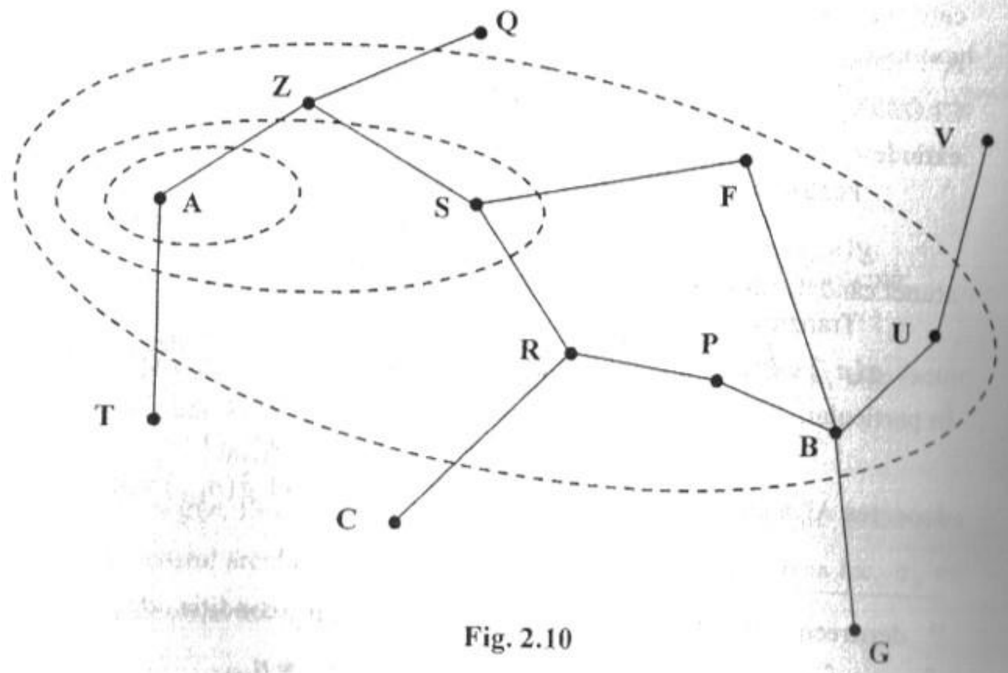


Fig. 2.10

De îndată ce căutarea în interiorul unui contur dat a fost completată, este declanşată o nouă iteraţie, folosind un nou cost f, corespunzător următorului contur. Fig. 2.10 prezintă căutări iterative în interiorul câte unui contur.

Algoritmul IDA* este complet şi optim cu aceleaşi amendamente ca şi A*. Deoarece este de tip depth-first *nu necesită decât un spaţiu proporţional cu cel mai lung drum pe care îl explorează.*

Dacă δ este cel mai mic cost de operator, iar f^* este costul soluţiei optime, atunci, în cazul cel mai nefavorabil, IDA* va necesita spaţiu pentru memorarea a $\frac{bf^*}{\delta}$ noduri, unde b este acelaşi factor de ramificare.

Complexitatea de timp a algoritmului depinde în mare măsură de numărul valorilor diferite pe care le poate lua funcţia euristică.

2.3.3. Implementarea Prolog a căutării de tip best-first

Pentru o mai bună înţelegere a metodei, în cele ce urmează, prezentarea căutării de tip best-first, în vederea implementării ei în Prolog, se va face conform abordării din [3].

Astfel, vom imagina căutarea de tip best-first funcţionând în felul următor: căutarea constă dintr-un număr de subprocese "concurente", fiecare explorând alternativa sa, adică propriul subarbore. Subarborii au subarbori, care vor fi la rândul lor explorati de subprocese ale subproceselor, ş.a.m.d.. Dintre toate aceste subprocese doar unul este activ la un moment dat şi anume cel care se ocupă de alternativa cea mai promiţătoare (adică alternativa corespunzătoare celei mai mici $\hat{f}$ - valori). Celelalte procese aşteaptă până când $\hat{f}$ - valorile se schimbă astfel încât o altă alternativă devine mai promiţătoare, caz în care procesul corespunzător acesteia devine activ. Acest mecanism de activare-dezactivare poate fi privit după cum urmează: procesului corespunzător alternativei curente de prioritate maximă i se alocă un buget şi, atâta vreme cât acest buget nu este epuizat, procesul este activ. Pe durata activităţii sale, procesul îşi extinde propriul subarbore, iar în cazul atingerii unei stări-scop este anunţată găsirea unei soluţii. Bugetul acestei funcţionări este determinat de $\hat{f}$ - valoarea corespunzătoare celei mai apropiate alternative concurente.

Ilustrăm această idee în exemplul următor. Considerăm oraşele **s**, **a**, **b**, **c**, **d**, **e**, **f**, **g**, **t** unite printr-o reţea de drumuri ca în Fig. 2.11. Aici fiecare drum direct între două oraşe este etichetat cu lungimea sa; numărul din căsuţa alăturată unui oraş reprezintă distanţa în linie dreaptă între oraşul respectiv şi oraşul **t**. Ne punem problema determinării celui mai scurt drum între oraşul **s** şi oraşul **t** utilizând strategia best-first. Definim în acest scop funcţia $\hat{h}$

bazându-ne pe distanţa în linie dreaptă între două oraşe. Astfel, pentru un oraş X, definim

$$\hat{f}(X) = \hat{g}(X) + \hat{h}(X) = \hat{g}(X) + dist(X,t)$$

unde $dist(X,t)$ reprezintă distanţa în linie dreaptă între **X** şi **t**.

În acest exemplu, căutarea de tip best-first este efectuată prin intermediul a două procese, P_1 si P_2, ce explorează fiecare câte una din cele două căi alternative. Calea de la **s** la **t** via nodul **a** corespunde procesului P_1, iar calea prin nodul **e** corespunde procesului P_2.

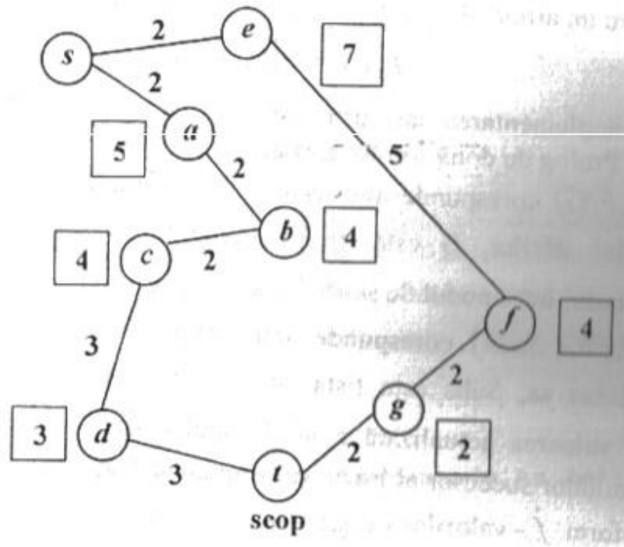


Fig. 2.11

În stadiile iniţiale, procesul P_1 este mai activ, deoarece $\hat{f}$ - valorile de-a lungul căii corespunzătoare lui sunt mai mici decât $\hat{f}$ - valorile de-a lungul celeilalte căi. Atunci când P_1 explorează **c**, iar procesul P_2 este încă la **e**, $\hat{f}(c) = \hat{g}(c) + \hat{h}(c) = 6 + 4 = 10$, $\hat{f}(e) = \hat{g}(e) + \hat{h}(e) = 2 + 7 = 9$ şi deci $\hat{f}(e) < \hat{f}(c)$. În acest moment, situaţia se schimbă: procesul P_2 devine activ, iar procesul P_1 intră în aşteptare. În continuare, $\hat{f}(c) = 10$, $\hat{f}(f) = 11$, $\hat{f}(c) < \hat{f}(f)$ şi deci P_1 devine activ şi P_2 intră în aşteptare. Pentru că $\hat{f}(d) = 12 > 11$, procesul P_1 va reintra în aşteptare, iar procesul P_2 va rămâne activ până când se va atinge starea scop **t**.

Căutarea schiţată mai sus porneşte din nodul iniţial şi este continuată cu generarea unor noduri noi, conform relaţiei de succesiune. În timpul acestui

proces, este generat un arbore de căutare, a cărui rădăcină este nodul de start. Acest arbore este extins în direcția cea mai promițătoare conform $\hat{f}$ - valorilor, până la găsirea unei soluții.

În vederea implementării în Prolog, vom extinde definiția lui $\hat{f}$, de la noduri în spațiul stărilor, la arbori, astfel:

- pentru un arbore cu un singur nod N, avem egalitate între $\hat{f}$ - valoarea sa și $\hat{f}(N)$;
- pentru un arbore T cu rădăcina N și subarborii $S_1, S_2, \ldots$ definim

$$\hat{f}(T) = \min_i \hat{f}(S_i)$$

În implementarea care urmează, vom reprezenta arborele de căutare prin termeni Prolog de două forme, și anume:

- $l(N,F/G)$ corespunde unui arbore cu un singur nod N; N este nod în spațiul stărilor, G este $\hat{g}(N)$ (considerăm $\hat{g}(N)$ ca fiind costul drumului între nodul de start și nodul N), $F = G + \hat{h}(N)$.
- $t(N,F/G, Subs)$ corespunde unui arbore cu subarbori nevizi; N este rădăcina sa, Subs este lista subarborilor săi, G este $\hat{g}(N)$, F este $\hat{f}$ - valoarea actualizată a lui N, adică este $\hat{f}$ - valoarea celui mai promițător succesor al lui N; de asemenea, Subs este ordonată crescător conform $\hat{f}$ - valorilor subarborilor constituenți.

Recalcularea $\hat{f}$ - valorilor este necesară pentru a permite programului să recunoască cel mai promițător subarbore, la fiecare nivel al arborelui de căutare (adică arborele care conține cel mai promițător nod terminal).

În exemplul anterior, în momentul în care nodul s tocmai a fost extins, arborele de căutare va avea 3 noduri: rădăcina **s** și copiii **a** si **e**. Acest arbore va fi reprezentat, în program, prin intermediul termenului Prolog $t(s,7/0,[l(a,7/2),l(e,9/2)])$. Observăm că $\hat{f}$ - valoarea lui **s** este 7, adică $\hat{f}$ - valoarea celui mai promițător subarbore al său. În continuare va fi extins subarborele de rădăcină **a**. Cel mai apropiat competitor al lui **a** este **e**; cum $\hat{f}(e) = 9$, rezultă că subarborele de rădăcină **a** se poate extinde atâta timp cât $\hat{f}$ - valoarea sa nu va depăși 9. Prin urmare, sunt generate **b** si **c**. Deoarece $\hat{f}(c) = 10$, rezultă că limita de extindere a fost depășită și alternativa **a** nu mai poate "crește". În acest moment, termenul Prolog corespunzător subarborelui de căutare este următorul:

$$t(s,9/0,[l(e,9/2),t(a,10/2,[t(b,10/4,[l(c,10/6)])])])$$

În implementarea care urmează, predicatul cheie va fi predicatul expandeaza:

```
expandeaza(Drum,Arb,Limita,Arb1,Rez,Solutie)
```

Argumentele sale au următoarele semnificații:

- **Drum** reprezintă calea între nodul de start al căutării și **Arb**;
- **Arb** este arborele (subarborele) curent de căutare;
- **Limita** este $\hat{f}$ - limita pentru extinderea lui **Arb**;
- **Rez** este un indicator a cărui valoare poate fi "da", "nu", "imposibil";
- **Solutie** este o cale de la nodul de start ("prin **Arb1**") către un nod-scop (în limita **Limita**), dacă un astfel de nod-scop există.

Drum, Arb și **Limita** sunt parametrii de intrare pentru **expandeaza** (în sensul că ei sunt deja instanțiati atunci când **expandeaza** este folosit). Prin utilizarea predicatului **expandeaza** se pot obține trei feluri de rezultate, rezultate indicate prin valoarea argumentului **Rez**, după cum urmează:

- **Rez=da,** caz în care **Solutie** va unifica cu o cale soluție găsită expandând **Arb** în limita **Limita** (adică fără ca $\hat{f}$ - valoarea să depășească limita **Limita**); **Arb1** va rămâne neinstanțiat;
- **Rez=nu,** caz în care **Arb1** va fi, de fapt, **Arb** expandat până când $\hat{f}$ - valoarea sa a depășit **Limita; Solutie** va rămâne neinstanțiat;
- **Rez=imposibil,** caz în care argumentele **Arb1** și **Solutie** vor rămâne neinstanțiate; acest ultim caz indică faptul că explorarea lui **Arb** este o alternativă "moartă", deci nu trebuie să i se mai dea o șansă pentru reexplorare în viitor; acest caz apare atunci când $\hat{f}$ - valoarea lui **Arb** este mai mică sau egală decât **Limita**, dar arborele nu mai poate fi extins, fie pentru că nici o frunză a sa nu mai are succesori, fie pentru că un astfel de succesor ar crea un ciclu.

Vom prezenta în continuare o implementare a unei variante a metodei best-first [3], în SICStus Prolog, implementare care folosește considerentele anterioare.

Strategia best-first

```
%Predicatul bestfirst(Nod_initial,Solutie) este adevarat
%daca Solutie este un drum (obtinut folosind strategia
%best-first) de la nodul Nod_initial la o stare-scop.

bestfirst(Nod_initial,Solutie):-
          expandeaza([],l(Nod_initial,0/0),9999999,_,
          da,Solutie).
```

```
expandeaza(Drum,l(N,_),_,_, da,[N|Drum]):-scop(N).

%Caz 1: daca N este nod-scop, atunci construim o
%cale-solutie.

expandeaza(Drum,l(N,F/G),Limita,Arbl,Rez,Sol):-
      F=<Limita,
      (bagof(M/C,(s(N,M,C), \+ (membru(M,Drum))),Succ),!,
      listasucc(G,Succ,As),
      cea_mai_buna_f(As,Fl),
      expandeaza(Drum,t(N,Fl/G,As),Limita,Arbl, Rez,Sol);
      Rez=imposibil).

%Caz 2: Daca N este nod-frunza a carui f̂-valoare este mai
%mica decat Limita, atunci ii generez succesorii si ii
%extind in limita Limita.

expandeaza(Drum,t(N,F/G,[A|As]),Limita,Arbl,Rez,
Sol):-
      F=<Limita,
      cea_mai_buna_f(As,BF),
      min(Limita,BF,Limita1),
      expandeaza([N|Drum],A,Limita1,A1,Rez1,Sol),
      continua(Drum,t(N,F/G,[A1|As]),Limita,Arbl,
      Rez1,Rez,Sol).

%Caz 3: Daca arborele de radacina N are subarbori nevizi si
%f̂-valoarea este mai mica decat Limita, atunci extindem
%cel mai "promitator" subarbore al sau; in functie de
%rezultatul obtinut, Rez, vom decide cum anume vom continua
%cautarea prin intermediul procedurii (predicatului)
%continua.

expandeaza(_,t(_,_,[]),_,_,imposibil,_):-!.

%Caz 4: pe aceasta varianta nu o sa obtinem niciodata
%o solutie.

expandeaza(_,Arb,Limita,Arb,nu,_):-
      f(Arb,F),
      F>Limita.

%Caz 5: In cazul unor f̂-valori mai mari decat Bound,
%arborele nu mai poate fi extins.

continua(_,_,_,_,da,da,Sol).
```



```
continua(P,t(N,F/G,[A1|As]),Limita,Arbl,nu,Rez,Sol):-
      insereaza(A1,As,NAs),
      cea_mai_buna_f(NAs,Fl),
      expandeaza(P,t(N,Fl/G,NAs),Limita,Arbl,Rez,Sol).
continua(P,t(N,F/G,[_|As]),Limita,Arbl,imposibil,Rez,
Sol):-
      cea_mai_buna_f(As,Fl),
      expandeaza(P,t(N,Fl/G,As),Limita,Arbl,Rez,Sol).

listasucc(_,[],[]).
listasucc(G0,[N/C|NCs],Ts):-
      G is G0+C,
      h(N,H),
      F is G+H,
      listasucc(G0,NCs,Ts1),
      insereaza(l(N,F/G),Ts1,Ts).

%Predicatul insereaza(A,As,As1) este utilizat pentru
%inserarea unui arbore A intr-o lista de arbori As,
%mentinand ordinea impusa de f̂-valorile lor.

insereaza(A,As,[A|As]):-
      f(A,F),
      cea_mai_buna_f(As,Fl),
      F=<Fl,!.
insereaza(A,[A1|As],[A1|As1]):-insereaza(A,As,As1).

min(X,Y,X):-X=<Y,!.
min(_,Y,Y).

f(l(_,F/_),F).      % f-val unei frunze
f(t(_,F/_,_),F).    % f-val unui arbore

%Predicatul cea_mai_buna_f(As,F) este utilizat pentru
%a determina cea mai buna f̂-valoare a unui arbore din
%lista de arbori As, daca aceasta lista este nevida; lista
%As este ordonata dupa f̂-valorile subarborilor
%constituenti.

cea_mai_buna_f([A|_],F):-f(A,F).
cea_mai_buna_f([],999999).

%In cazul unei liste de arbori vide, f̂-valoarea
%determinata este foarte mare.
```

Pentru aplicarea programului anterior la o problemă particulară, trebuie adăugate anumite relații specifice problemei. Aceste relații definesc de fapt problema particulară ("regulile jocului") și, de asemenea, adaugă o anumită informație euristică despre cum anume s-ar putea rezolva aceasta. Predicatele specifice problemei sunt:

- `s(Nod,Nod1,Cost)`

% acest predicat este adevărat dacă există un arc între Nod1 și Nod în spațiul stărilor

- `scop(Nod)`

% acest predicat este adevărat dacă Nod este stare-scop în spațiul stărilor

- `h(Nod,H)`

% H este o estimație euristică a costului celui mai ieftin drum între Nod și o stare-scop.

Exemplificăm, în cele ce urmează, folosirea strategiei generale best-first în cazul problemei "eight puzzle", precum și al problemei misionarilor și canibalilor.

2.3.3.1. Exemple

- **Problema 8-puzzle**

Se consideră un tablou pătratic de dimensiune 3x3, împărțit în 9 câmpuri pătrate de latură 1. Opt dintre aceste câmpuri conțin câte o plăcuță etichetată cu un număr între 1 și 8, în timp ce al nouălea câmp nu conține nimic. O plăcuță poate fi deplasată de pe câmpul sau pe un câmp vecin, dacă astfel nu se suprapune peste altă plăcuță, deci dacă acest ultim câmp nu conține nici o plăcuță. Fiind date două configurații C1, C2, ale plăcuțelor pe tabloul 3x3, se cere să se spună dacă și cum este posibil să se ajungă din configurația C1 în configurația C2 printr-un număr minim de mutări.

Rezolvare:

Vom trata această problemă ca pe una de căutare euristică. Un nod în spațiul stărilor este de fapt o configurație a plăcuțelor în tablou. Deoarece ne interesează rezolvarea problemei printr-un număr minim de mutări (adică ne propunem să minimizăm lungimea soluțiilor), este firesc să considerăm că lungimile arcelor din spațiul stărilor sunt egale cu 1.

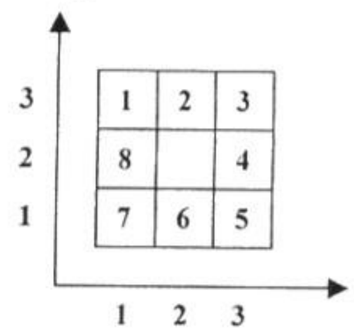
Pentru rezolvarea problemei pot fi luate în considerație mai multe funcții euristice [3]. Astfel, pentru o stare (configurație) S, putem defini:

- $\hat{h}_1(S)$= numărul de plăcuțe care nu sunt la locul lor în starea S față de starea finală;
- $\hat{h}_2(S)$= suma distanțelor pe verticală și pe orizontală de la poziția fiecărei plăcuțe în configurația curentă (S), la poziția respectivei plăcuțe în configurația finală.

Vom reprezenta o configurație în Prolog printr-o listă a pozițiilor curente corespunzătoare locului liber, respectiv plăcuțelor, fiecare poziție fiind specificată printr-o pereche de coordonate, de forma X/Y. Ordinea acestor obiecte în listă este următoarea:

(1) poziția curentă a locului liber (neocupat de nici o plăcuță),
(2) poziția curentă a plăcuței etichetate cu 1,
(3) poziția curentă a plăcuței etichetate cu 2 etc.

Exemplu:



[2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]

Fig. 2.12

În cele ce urmează, vom considera configurația ilustrată în exemplul de mai sus ca fiind configurația finală C2 din enunțul problemei, indiferent de alegerea configurației inițiale. Prin urmare, predicatul scop este definit astfel:

scop ([2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]).

În acest caz, putem defini o nouă euristică, $\hat{h}_3$, după cum urmează:

$$\hat{h}_3(S) = \hat{h}_2(S) + 3 * \sec v(S),$$

pentru o stare S, unde $\sec v(S)$ este un indice al secvențialității, adică al gradului în care plăcuțele sunt deja ordonate în configurația actuală, în raport cu ordinea lor în configurația finală. Acest indice este calculat ca și suma indicilor corespunzători fiecărei plăcuțe, astfel:

- indicele plăcuței aflate în centrul tabloului este 1;

- indicele unei plăcuţe aflate într-o poziţie diferită de centru este 0, dacă acea plăcuţă este urmată (în sens invers-trigonometric) de succesorul său din configuraţia finală;
- în orice altă situaţie, indicele este egal cu 2.

De exemplu, pentru configuraţia S_a din Fig. 2.13 a), $\sec v(S_a) = 6$. Fig. 2.13 prezintă trei exemple de configuraţii iniţiale:

1	3	4
8		2
7	6	5

a)

2	8	3
1	6	4
7		5

b)

2	1	6
4		8
7	5	3

c)

Fig. 2.13

Funcţiile euristice $\hat{h}_1$ şi $\hat{h}_2$ sunt admisibile, iar $\hat{h}_2$ este mai informată decât $\hat{h}_1$. Funcţia euristică $\hat{h}_3$ funcţionează bine, în sensul că ea dirijează foarte eficient căutarea spre atingerea scopului. De exemplu, rezolvând problema pentru configuraţiile a) şi b) din Fig. 2.13, cu această euristică, nici un nod din afara drumului celui mai scurt nu va fi extins înainte de găsirea primei soluţii. Acest lucru înseamnă că drumurile cele mai scurte în aceste cazuri sunt generate fără backtracking. Chiar şi pentru configuraţia c) din Fig. 2.13, problema este rezolvată aproape direct.

În ciuda calităţilor ei, euristica $\hat{h}_3$ nu este totuşi admisibilă. Cu alte cuvinte, ea nu garantează că soluţia cea mai scurtă este găsită întotdeauna înaintea uneia mai lungi. Motivul este acela că funcţia aleasă nu satisface condiţia de admisibilitate pentru toate nodurile din spaţiul stărilor. Spre exemplu, pentru configuraţia iniţială S_a, avem:

$$\hat{h}_3(S_a) = 4 + 3*6$$
$$h(S_a) = 4$$

Implementarea în Prolog a predicatelor specifice problemei "8-puzzle" va folosi relaţia auxiliară

mandist (P1, P2, D)

pentru determinarea distanţei Manhattan D între poziţiile (pătratele) P1 şi P2, definită ca fiind suma dintre distanţa pe orizontală şi cea pe verticală între cele două pătrate. Prezentăm, în continuare, programul Prolog complet corespunzător acestei probleme:

Programul 2.7

```
%Procedurile (predicatele) specifice problemei eight-puzzle

%Definim relatia de succesiune; consideram ca toate arcele
%au costul 1.

s([Gol|Placute],[Placuta|Placute1],1):-
                interschimba(Gol,Placuta,Placute,Placute1).

%Predicatul modul(A,B,D) este adevarat daca D este |A-B|
modul(A,B,D):-D is A-B,D>=0,!;
              D is B-A.

distman(X/Y,X1/Y1,D):-
            modul(X,X1,Dx),modul(Y,Y1,Dy), D is Dx+Dy.

interschimba(Gol,Placuta,[Placuta|Ts],[Gol|Ts]):-
            distman(Gol,Placuta,1).
interschimba(Gol,Placuta,[T1|Ts],[T1|Ts1]):-
            interschimba(Gol,Placuta,Ts,Ts1).

membru(H,[H|_]).
membru(X,[_|T]):-membru(X,T).

h([Gol|Placute],H):-
            scop([Gol1|Placute_final]),
            disttot(Placute,Placute_final,D),
            secv(Placute,S),
            H is D+3*S.

secv([Prima|Placute],S):-
            secv([Prima|Placute],Prima,S).
secv([Placuta1,Placuta2|Placute],Prima,S):-
            scor(Placuta1,Placuta2,S1),
            secv([Placuta2|Placute],Prima,S2),
            S is S1+S2.
secv([Ultima],Prima,S):-scor(Ultima,Prima,S).

scor(2/2,_,1):-!.
%Placuta plasata in centru are scorul 1.
scor(1/3,2/3,0):-!.
scor(2/3,3/3,0):-!.
scor(3/3,3/2,0):-!.
scor(3/2,3/1,0):-!.
scor(3/1,2/1,0):-!.
scor(2/1,1/1,0):-!.
```

```
scor(1/1,1/2,0):-!.
scor(1/2,1/3,0):-!.
scor(_,_,2).

disttot([],[],0).
disttot([T|Ts],[S|Ss],D):-
            distman(T,S,D1),
            disttot(Ts,Ss,D2),
            D is D1+D2.

scop([2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2]).

initial([2/1,1/2,1/3,3/3,3/2,3/1,2/2,1/1,2/3]).

arata_solutie([]).
arata_solutie([P|L]):-
            arata_solutie(L),nl,
            write('-------'),
            arata_poz(P).

arata_poz([S0,S1,S2,S3,S4,S5,S6,S7,S8]):-
            membru(Y,[3,2,1]),nl,membru(X,[1,2,3]),
            membru(P-X/Y,[' '-S0,1-S1,2-S2,3-S3,4-S4,5-S5,
            6-S6,7-S7,8-S8]),
            write(P),tab(2),fail;
            true.

pb:-tell('C:\\best_first_output.txt'),
    initial(Poz),bestfirst(Poz,Sol),
    arata_solutie(Sol),told.

%Predicatul bestfirst(Nod_initial,Solutie) este adevarat
%daca Solutie este un drum (obtinut folosind strategia
%best-first) de la nodul Nod_initial la o stare-scop.

bestfirst(Nod_initial,Solutie):-
            expandeaza([],l(Nod_initial,0/0),9999999,_,
                        da,Solutie).

expandeaza(Drum,l(N,_),_,_, da,[N|Drum]):-scop(N).

%Caz 1: daca N este nod-scop, atunci construim o cale-
%solutie.

expandeaza(Drum,l(N,F/G),Limita,Arbl,Rez,Sol):-
      F=<Limita,
      (bagof(M/C,(s(N,M,C), \+ (membru(M,Drum))),Succ),!,
```



```
      listasucc(G,Succ,As),
      cea_mai_buna_f(As,F1),
      expandeaza(Drum,t(N,F1/G,As),Limita,Arbl, Rez,Sol);
      Rez=imposibil).

%Caz 2: Daca N este nod-frunza a carui f̂-valoare este mai
%mica decat Limita, atunci ii generez succesorii si ii
%expandez in limita Limita.

expandeaza(Drum,t(N,F/G,[A|As]),Limita,Arbl,Rez,
Sol):-
      F=<Limita,
      cea_mai_buna_f(As,BF),
      min(Limita,BF,Limita1),
      expandeaza([N|Drum],A,Limita1,A1,Rez1,Sol),
      continua(Drum,t(N,F/G,[A1|As]),Limita,Arbl,
               Rez1,Rez,Sol).

%Caz 3: Daca arborele de radacina N are subarbori nevizi si
%f̂-valoarea este mai mica decat Limita, atunci expandam
%cel mai "promitator" subarbore al sau; in functie de
%rezultatul obtinut, Rez, vom decide cum anume vom continua
%cautarea prin intermediul procedurii (predicatului)
%continua.

expandeaza(_,t(_,_,[]),_,_,imposibil,_):-!.

%Caz 4: pe aceasta varianta nu o sa obtinem niciodata
%o solutie.

expandeaza(_,Arb,Limita,Arb,nu,_):- f(Arb,F),
                                    F>Limita.

%Caz 5: In cazul unor f̂-valori mai mari decat Bound,
%arborele nu mai poate fi extins.

continua(_,_,_,_,da,da,Sol).
continua(P,t(N,F/G,[A1|As]),Limita,Arbl,nu,Rez,Sol):-
     insereaza(A1,As,NAs),
     cea_mai_buna_f(NAs,F1),
     expandeaza(P,t(N,F1/G,NAs),Limita,Arbl,Rez,Sol).
continua(P,t(N,F/G,[_|As]),Limita,Arbl,imposibil,Rez,
Sol):-
      cea_mai_buna_f(As,F1),
      expandeaza(P,t(N,F1/G,As),Limita,Arbl,Rez,Sol).

listasucc(_,[],[]).
```

```
listasucc(G0,[N/C|NCs],Ts):-
                        G is G0+C,
                        h(N,H),
                        F is G+H,
                        listasucc(G0,NCs,Ts1),
                        insereaza(l(N,F/G),Ts1,Ts).

%Predicatul insereaza(A,As,As1) este utilizat pentru
%inserarea unui arbore A intr-o lista de arbori As,
%mentinand ordinea impusa de f̂-valorile lor.

insereaza(A,As,[A|As]):-f(A,F),
                        cea_mai_buna_f(As,F1),
                        F=<F1,!.
insereaza(A,[A1|As],[A1|As1]):-insereaza(A,As,As1).

min(X,Y,X):-X=<Y,!.
min(_,Y,Y).

f(l(_,F/_),F).      % f-val unei frunze
f(t(_,F/_,_),F).    % f-val unui arbore

%Predicatul cea_mai_buna_f(As,F) este utilizat pentru
%a determina cea mai buna f̂-valoare a unui arbore din
%lista de arbori As, daca aceasta lista este nevida; lista
%As este ordonata dupa f̂-valorile subarborilor
%constituenti.

cea_mai_buna_f([A|_],F):-f(A,F).
cea_mai_buna_f([],999999).

%In cazul unei liste de arbori vide, f̂-valoarea
%determinata este foarte mare.
```

Interogarea Prologului se face după cum urmează:

```
?- pb.
```

Răspunsul sistemului va fi

```
yes
```


cu semnificația că predicatul a fost satisfăcut. În fișierul `C:\best_first_output.txt` putem vedea și secvența de mutări prin care s-a ajuns din starea inițială

2	8	3
1	6	4
7		5

în starea finală

1	2	3
8		4
7	6	5

După interogarea anterioară, conținutul fișierului `C:\best_first_output.txt` va fi următorul:

```
-------
2 8 3
1 6 4
7   5
-------
2 8 3
1   4
7 6 5
-------
2   3
1 8 4
7 6 5
-------
  2 3
1 8 4
7 6 5
-------
1 2 3
  8 4
7 6 5
-------
```

1 2 3
8 4
7 6 5

- **Problema misionarilor şi canibalilor**

Pe malul estic al unei ape se găsesc N misionari şi N canibali. Aceştia urmează să treacă apa, având la dispoziţie o barcă cu M locuri. Se ştie că dacă pe unul dintre maluri numărul de canibali este mai mare decât numărul de misionari, atunci misionarii de pe acel mal sunt mâncaţi de canibali. Se cere determinarea unei variante de trecere a apei fără ca misionarii să fie mâncaţi de canibali.

Rezolvare:

În §2.2.1.4. am expus deja o implementare în Prolog a unei modalităţi de rezolvare a acestei probleme folosind mijloacele specifice căutării neinformate (algoritmul breadth-first). Fără a modifica esenţial predicatele din programul 2.2 folosite pentru descrierea spaţiului stărilor, precum şi predicatele specifice implementării strategiei best-first, definite anterior (§2.3.3), putem construi o rezolvare a acestei probleme folosind strategia best-first. Considerăm că singurul aspect asupra căruia este necesar să ne mai concentrăm atenţia este acela al stabilirii unor funcţii euristice adecvate.

Prezentăm, în continuare, două exemple de funcţii euristice. Pentru o stare dată S, notăm cu $n_C(\text{est})$ numărul de canibali de pe malul de est şi cu $n_M(\text{est})$ numărul de misionari de pe acelaşi mal. Definim:

$$\hat{h}_1(S) = \left[\frac{n_C(\text{est}) + n_M(\text{est})}{M}\right]$$

$$\hat{h}_2(S) = \begin{cases} 2\left[\dfrac{n_C(\text{est}) + n_M(\text{est}) - 2}{M-1}\right] + 1, & \text{daca } n_C(\text{est}) + n_M(\text{est}) \neq 0 \\ & \text{si barca se afla pe malul de est} \\ 2\left[\dfrac{n_C(\text{est}) + n_M(\text{est}) - 1}{M-1}\right] + 2, & \text{daca } n_C(\text{est}) + n_M(\text{est}) \neq 0 \\ & \text{si barca se afla pe malul de vest} \\ 0, & \text{daca } n_C(\text{est}) + n_M(\text{est}) = 0 \end{cases}$$

Funcţia euristică $\hat{h}_2$ descrie situaţia în care un transport est-vest se face cu cât mai mulţi pasageri (eventual cu M - capacitatea totală a bărcii), iar unul vest-est se face cu cât mai puţini (adică unul singur), fără a ţine cont de relaţia dintre numărul de canibali şi cel de misionari de pe cele două maluri. Dacă, în plus, intervine şi constrângerea privitoare la această relaţie numerică între canibali şi misionari, cu siguranţă că transportul va fi făcut într-un număr de drumuri mai mare sau egal cu cel specificat de euristica h_2. Prin urmare, această funcţie euristică este admisibilă. Se observă uşor că şi funcţia euristică $\hat{h}_1$ este admisibilă, dar şi că aceasta este mai puţin informată decât $\hat{h}_2$.

2.3.4. Concluzii

1. Până acum, *cunoştinţele* au fost folosite numai în procesul formulării unei probleme în termeni de *stări* şi de *operatori*. Fiind însă dată o problemă bine definită, singurul loc în care putem aplica cunoştinţele este în funcţia ce organizează coada. Această funcţie trebuie să determine următorul nod care va fi extins. De obicei, cunoştinţele care permit determinarea acestuia sunt furnizate de către o *funcţie de evaluare* care întoarce un număr menit să descrie cât de oportună sau neoportună este extinderea unui anumit nod la pasul următor. Atunci când nodurile sunt ordonate astfel încât cel care este cel mai bine evaluat să fie primul extins, strategia care rezultă poartă denumirea de *căutare de tip best-first*.

2. Una dintre cele mai simple strategii de căutare de tip best-first constă în minimizarea costului estimat pentru a fi atins scopul. Cu alte cuvinte, nodul a cărui stare este considerată ca fiind cea mai apropiată de starea-scop este întotdeauna primul extins. Pentru majoritatea problemelor, costul atingerii scopului plecându-se dintr-o anumită stare nu poate fi determinat în mod exact,

dar poate fi estimat. O funcţie care calculează astfel de estimări ale costului se numeşte *funcţie euristică* şi este, de obicei, notată cu *h*:

h(n) = costul estimat al celui mai ieftin drum de la starea din nodul *n* la o stare-scop.

O căutare de tip best-first care foloseşte această funcţie *h* pentru a selecta următorul nod ce va fi extins se numeşte o *căutare de tip Greedy*.

3. *Căutarea de tip Greedy* minimizează costul estimat până la scop, *h(n)* şi, prin urmare, reduce considerabil costul căutării. Ea nu este însă nici optimă şi nici completă. Pe de altă parte, *căutarea de cost uniform* minimizează costul, *g(n)*, al drumului găsit până la momentul curent. Ea este şi optimă şi completă, dar poate fi foarte ineficientă. Este necesară combinarea celor două strategii pentru a se obţine avantajele oferite de amândouă. Cele două funcţii de evaluare folosite de ele pot fi combinate prin simpla sumare:

$$f(n) = g(n) + h(n)$$

Întrucât *g(n)* furnizează costul drumului de la nodul de start la nodul *n*, iar *h(n)* reprezintă costul estimat al celui mai ieftin drum de la *n* la scop, avem că *f(n)* este costul estimat al celei mai ieftine soluţii care trece prin *n*. Prin urmare, pentru a găsi cea mai ieftină soluţie, se încearcă mai întâi extinderea nodului care are cea mai mică valoare a lui *f*. Această strategie se demonstrează că este şi completă şi optimă, dacă se impune o restricţie simplă asupra funcţiei *h*. Restricţia asupra lui *h* este aceea de a se alege întotdeauna *o funcţie h care nu supraestimează costul* atingerii scopului. O asemenea funcţie *h* se numeşte *euristică admisibilă*.

4. Euristicile admisibile sunt întotdeauna optimiste, întrucât presupun costul rezolvării unei probleme mai scăzut decât este el în realitate. Acest optimism se transferă şi asupra funcţiei *f* în felul următor:

- dacă *h* este admisibilă, *f(n)* niciodată nu supraestimează costul efectiv al celei mai bune soluţii care trece prin *n*.

Căutarea de tip best-first care foloseşte acest *f* ca funcţie de evaluare şi o funcţie *h* admisibilă se numeşte *căutare de tip A**. A* este *complet* şi *optimal*.

5. S-a demonstrat (Dechter şi Pearl, 1985) că algoritmul A* este *optim eficient*. Aceasta înseamnă că nici un alt algoritm optim nu extinde mai puţine noduri decât A*.

6. Pentru a folosi algoritmul A* în rezolvarea unei probleme concrete, trebuie definite, spre a fi utilizate de algoritm: un spaţiu al stărilor, un predicat scop şi o funcţie euristică adecvată fiecărei probleme.

7. Aproape toate euristicile admisibile au proprietatea de *monotonie*. Dacă euristica nu este monotonă, se poate face o corecţie minoră care restaurează monotonia, după cum urmează: considerăm două noduri, *n* şi *n'*, unde *n* este părintele lui *n'*. De fiecare dată când generăm un nou nod, vom verifica dacă costul funcţiei *f* asociate lui este mai mic decât costul funcţiei *f* asociate nodului părinte. Dacă da, folosim funcţia de cost *f* a nodului părinte:

$$f(n') = max(f(n), g(n')+h(n'))$$

Dacă se foloseşte această ecuaţie, atunci *f* va fi întotdeauna *nedescrescătoare* de-a lungul oricărui drum de la rădăcină, cu condiţia ca *h* să fie admisibilă.

8. Algoritmi de căutare de tip "memory-bounded", cum ar fi IDA*, înlătură neajunsurile legate de problema spaţiului de memorie folosit, fără a sacrifica optimalitatea sau completitudinea.

CAPITOLUL 3
JOCURILE CA PROBLEME DE CĂUTARE

Jocurile au reprezentat întotdeauna o arie de aplicaţie interesantă pentru algoritmii euristici. Jocurile de două persoane sunt în general complicate datorită existenţei unui oponent ostil şi imprevizibil. De aceea ele sunt interesante din punctul de vedere al dezvoltării euristicilor, dar aduc multe dificultăţi în dezvoltarea şi aplicarea algoritmilor de căutare.

Prezenţa unui oponent face ca problema de decizie să devină una mai complexă şi mai complicată decât problemele de căutare discutate anterior. Oponentul introduce *incertitudinea*, întrucât nu se ştie niciodată ce va face acesta la pasul următor. În esenţă, toate programele referitoare la jocuri trebuie să trateze aşa numita *problemă de contingenţă*. Incertitudinea care intervine nu este de aceeaşi natură cu cea introdusă, de pildă, prin aruncarea unui zar sau cu cea determinată de starea vremii. Oponentul va încerca, pe cât posibil, să facă mutarea cea mai puţin benignă, în timp ce zarul sau vremea sunt presupuse a nu lua în consideraţie scopurile agentului.

Dar ceea ce face ca jocurile să constituie probleme cu adevărat diferite este faptul că, de regulă, ele sunt extrem de greu de rezolvat. Jocul de şah, spre exemplu, are un factor de ramificare mediu de valoare circa 35, iar partidele constau adesea din câte 50 de mutări corespunzătoare fiecărui jucător, astfel încât arborele de căutare are circa 35^{100} noduri.

Complexitatea jocurilor introduce *un tip de incertitudine complet nou*. Astfel, incertitudinea se naşte nu datorită faptului că există informaţie care lipseşte, ci datorită faptului că jucătorul nu are timp să calculeze consecinţele exacte ale oricărei mutări. Din acest punct de vedere, jocurile se aseamănă infinit mai mult cu lumea reală decât problemele de căutare standard.

Întrucât, în cadrul unui joc, există, de regulă, limite de timp, jocurile *penalizează ineficienţa* extrem de sever. Astfel, dacă o implementare a căutării de tip A*, care este cu 10% mai puţin eficientă, este considerată satisfăcătoare, un program pentru jocul de şah care este cu 10% mai puţin eficient în folosirea timpului disponibil va duce la pierderea partidei. Din această cauză, studiul nostru se va concentra asupra *tehnicilor de alegere a unei bune mutări atunci când timpul este limitat. Tehnica de "retezare"* ne va permite să ignorăm porţiuni ale arborelui de căutare care nu pot avea nici un rol în stabilirea alegerii finale, iar *funcţiile de evaluare euristice* ne vor permite să aproximăm utilitatea reală a unei stări fără a executa o căutare completă.

În cele ce urmează, ne vom referi la tehnici de joc corespunzătoare unor *jocuri de două persoane cu informaţie completă*, cum ar fi şahul. În cazul

jocurilor interesante, arborii rezultați sunt mult prea complecși pentru a se putea realiza o căutare exhaustivă, astfel încât sunt necesare abordări de o natură diferită. Una dintre metodele clasice se bazează pe „principiul minimax", implementat în mod eficient sub forma Algoritmului Alpha-Beta (bazat pe așa-numita *tehnică de alpha-beta retezare*).

3.1. O definire formală a jocurilor

Tipul de jocuri la care ne vom referi în continuare este acela al jocurilor de două persoane cu informație perfectă sau completă. În astfel de jocuri există doi jucători care efectuează mutări în mod alternativ, ambii jucători dispunând de informația completă asupra situației curente a jocului. (Prin aceasta, este exclus studiul majorității jocurilor de cărți). Jocul se încheie atunci când este atinsă o poziție calificată ca fiind "terminală" de către regulile jocului - spre exemplu, "mat" în jocul de șah. Aceleași reguli determină care este rezultatul jocului care s-a încheiat în această poziție terminală. Un asemenea joc poate fi reprezentat printr-un *arbore de joc* în care nodurile corespund situațiilor (stărilor), iar arcele corespund mutărilor. Situația inițială a jocului este reprezentată de nodul rădăcină, iar frunzele arborelui corespund pozițiilor terminale.

Vom lua în considerație cazul general al unui joc cu doi jucători, pe care îi vom numi MAX și respectiv MIN. MAX va face prima mutare, după care jucătorii vor efectua mutări pe rând, până când jocul ia sfârșit. La finalul jocului vor fi acordate puncte jucătorului câștigător (sau vor fi acordate anumite penalizări celui care a pierdut).

Un joc poate fi definit, în mod formal, ca fiind *un anumit tip de problemă de căutare având următoarele componente*:

- *starea inițială*, care include poziția de pe tabla de joc și o indicație referitoare la cine face prima mutare;
- *o mulțime de operatori*, care definesc mișcările permise ("legale") unui jucător;
- *un test terminal*, care determină momentul în care jocul ia sfârșit;
- *o funcție de utilitate* (numită și *funcție de plată*), care acordă o valoare numerică rezultatului unui joc; în cazul jocului de șah, spre exemplu, rezultatul poate fi *câștig, pierdere* sau *remiză*, situații care pot fi reprezentate prin valorile 1, -1 sau 0.

Dacă un joc ar reprezenta o problemă standard de căutare, atunci acțiunea jucătorului MAX ar consta din căutarea unei secvențe de mutări care conduc la o stare terminală reprezentând o stare câștigătoare (conform funcției de utilitate) și din efectuarea primei mutări aparținând acestei secvențe. Acțiunea lui MAX interacționează însă cu cea a jucătorului MIN. Prin urmare, MAX trebuie să găsească *o strategie* care va conduce la o stare terminală

câștigătoare, indiferent de acțiunea lui MIN. Această strategie include mutarea corectă a lui MAX corespunzătoare fiecărei mutări posibile a lui MIN. În cele ce urmează, vom începe prin a arăta cum poate fi găsită strategia optimă (sau rațională), deși în realitate nu vom dispune de timpul necesar pentru a o calcula.

3.2. Algoritmul Minimax

Oponenții din cadrul jocului pe care îl vom trata prin aplicarea Algoritmului Minimax vor fi numiți, în continuare, MIN și respectiv MAX. MAX reprezintă jucătorul care încearcă să câștige sau să își maximizeze avantajul avut. MIN este oponentul care încearcă să minimizeze scorul lui MAX. Se presupune că MIN folosește aceeași informație și încearcă întotdeauna să se mute la acea stare care este cea mai nefavorabilă lui MAX.

Algoritmul Minimax este conceput pentru a determina strategia optimă corespunzătoare lui MAX și, în acest fel, pentru a decide care este cea mai bună primă mutare:

Algoritmul Minimax

1. Generează întregul arbore de joc, până la stările terminale.

2. Aplică funcția de utilitate fiecărei stări terminale pentru a obține valoarea corespunzătoare stării.

3. Deplasează-te înapoi în arbore, de la nodurile-frunze spre nodul-rădăcină, determinând, corespunzător fiecărui nivel al arborelui, valorile care reprezintă utilitatea nodurilor aflate la acel nivel. Propagarea acestor valori la niveluri anterioare se face prin intermediul nodurilor-părinte succesive, conform următoarei reguli:

- dacă starea-părinte este un nod de tip MAX, atribuie-i maximul dintre valorile avute de fiii săi;
- dacă starea-părinte este un nod de tip MIN, atribuie-i minimul dintre valorile avute de fiii săi.

4. Ajuns în nodul-rădăcină, alege pentru MAX acea mutare care conduce la valoarea maximă.

□

- **Observație**: Decizia luată la pasul 4 al algoritmului se numește ***decizia minimax***, întrucât ea maximizează utilitatea, în ipoteza că oponentul joacă perfect cu scopul de a o minimiza.

Un arbore de căutare cu valori minimax determinate conform Algoritmului Minimax este cel din Fig. 3.1:

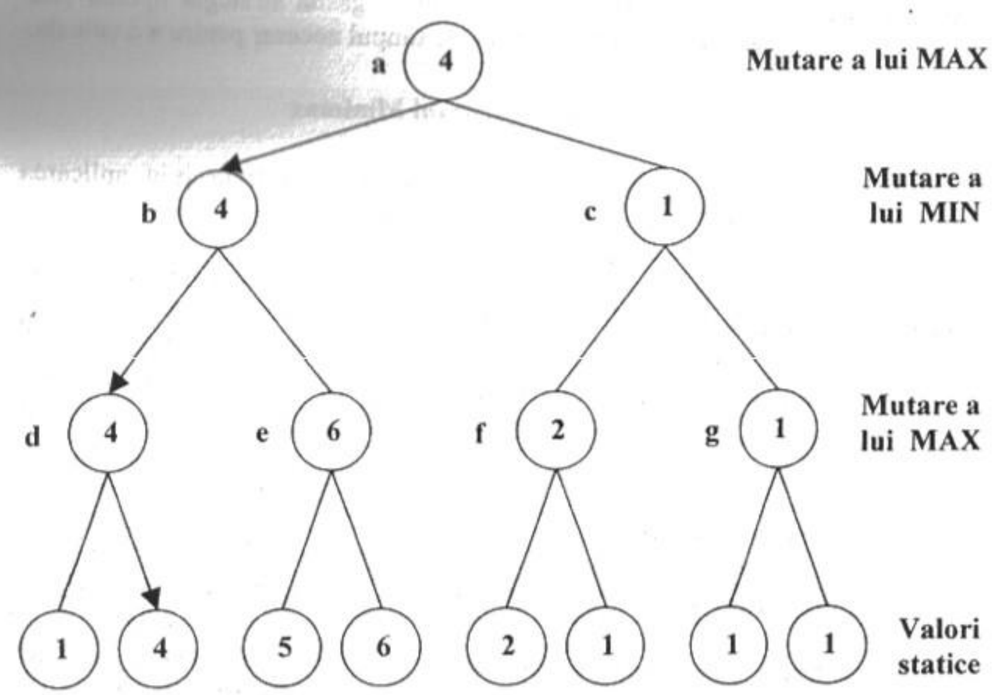


Fig. 3.1

Valorile poziţiilor de la ultimul nivel sunt determinate de către funcţia de utilitate şi se numesc *valori statice*. Valorile minimax ale nodurilor interne sunt calculate în mod dinamic, în manieră bottom-up, nivel cu nivel, până când este atins nodul-rădăcină. Valoarea rezultată, corespunzătoare acestuia, este 4 şi, prin urmare, cea mai bună mutare a lui MAX din poziţia *a* este *a-b*. Cel mai bun răspuns al lui MIN este *b-d*. Această secvenţă a jocului poartă denumirea de *variaţie principală*. Ea defineşte jocul optim de tip minimax pentru ambele părţi. Se observă că valoarea poziţiilor de-a lungul variaţiei principale nu variază. Prin urmare, mutările corecte sunt cele care *conservă valoarea jocului*.

Dacă adâncimea maximă a arborelui este *m* şi dacă există *b* "mutări legale" la fiecare punct, atunci complexitatea de timp a Algoritmului Minimax este $O(b^m)$. Algoritmul reprezintă o căutare de tip depth-first (deşi aici este sugerată o implementare bazată pe recursivitate şi nu una care foloseşte o coadă de noduri), astfel încât cerinţele sale de spaţiu sunt numai liniare în *m* şi *b*.

În cazul jocurilor reale, cerinţele de timp ale algoritmului sunt total nepractice, dar acest algoritm stă la baza atât a unor metode mai realiste, cât şi a analizei matematice a jocurilor.

Întrucât, pentru majoritatea jocurilor interesante, arborele de joc nu poate fi alcătuit în mod exhaustiv, au fost concepute diverse metode care se bazează pe căutarea efectuată numai într-o anumită porţiune a arborelui de joc. Printre acestea se numără şi tehnica Minimax, care, în majoritatea cazurilor, va căuta în arborele de joc numai până la *o anumită adâncime*, de obicei constând în numai câteva mutări. Ideea este de a evalua aceste poziţii terminale ale căutării, fără a mai căuta dincolo de ele, cu scopul de a face economie de timp. Aceste estimări se propagă apoi în sus de-a lungul arborelui, conform principiului Minimax. Mutarea care conduce de la poziţia iniţială, nodul-rădăcină, la cel mai promiţător succesor al său (conform acestor evaluări) este apoi efectuată în cadrul jocului.

Este de observat distincţia care se face, în acest cadru, între ***arborele de joc*** şi ***arborele de căutare***. Acesta din urmă este de obicei o parte a arborelui de joc (cea superioară), şi anume acea parte care este în mod explicit generată de către procesul de căutare. Prin urmare, poziţii terminale ale căutării nu reprezintă, neapărat, poziţii terminale ale jocului.

Jocurile reprezintă unul dintre primele obiecte de studiu ale inteligenţei artificiale. Încă din anul 1950 au fost scrise primele programe referitoare la jocul de şah. Autorii lor sunt Claude Shannon şi Alan Turing. Chiar în cadrul primului articol al lui Shannon, referitor la jocul de şah, se face propunerea ca, în loc de a merge până la stările terminale şi a folosi funcţia de utilitate, programul să încheie căutarea mult mai devreme şi să aplice *o funcţie de evaluare euristică* nodurilor-frunză ale arborelui rezultat. Algoritmul general Minimax a fost amendat în două moduri: funcţia de utilitate a fost înlocuită cu o funcţie de evaluare, iar testul terminal a fost înlocuit de către un aşa-numit *test de tăiere*.

Cea mai directă abordare a problemei deţinerii controlului asupra cantităţii de căutare care se efectuează este aceea de a fixa o limită a adâncimii, astfel încât testul de tăiere să aibă succes pentru toate nodurile aflate la sau sub adâncimea *d*. Limita de adâncime va fi aleasă astfel încât cantitatea de timp folosită să nu depăşească ceea ce permit regulile jocului. O abordare mai robustă a acestei probleme este aceea care aplică „iterative deepening". În acest caz, atunci când timpul expiră, programul întoarce mutarea selectată de către cea mai adâncă căutare completă.

3.2.1. Funcţii de evaluare

O *funcţie de evaluare* întoarce o *estimaţie*, realizată dintr-o poziţie dată, a utilităţii aşteptate a jocului. Ea are la bază evaluarea şanselor de câştigare a jocului de către fiecare dintre părţi, pe baza calculării caracteristicilor unei poziţii. În mod evident, performanţa unui program referitor la jocuri este extrem de dependentă de calitatea funcţiei de evaluare utilizate.

Funcţia de evaluare trebuie să îndeplinească anumite condiţii evidente: ea trebuie să concorde cu funcţia de utilitate în ceea ce priveşte stările terminale, calculele efectuate nu trebuie să dureze prea mult şi ea trebuie să reflecte în mod corect şansele efective de câştig. O valoare a funcţiei de evaluare acoperă mai multe poziţii diferite, grupate laolaltă într-o *categorie* de poziţii etichetată cu o anumită valoare. Spre exemplu, în jocul de şah, fiecare *pion* poate avea valoarea 1, un *nebun* poate avea valoarea 3 şamd.. În poziţia de deschidere evaluarea este 0 şi toate poziţiile până la prima captură vor avea aceeaşi evaluare. Dacă MAX reuşeşte să captureze un nebun fără a pierde o piesă, atunci poziţia rezultată va fi evaluată la valoarea 3. Toate poziţiile de acest fel ale lui MAX vor fi grupate într-o *categorie* etichetată cu "3". Funcţia de evaluare trebuie să reflecte şansa ca o poziţie aleasă la întâmplare dintr-o asemenea categorie să conducă la câştig (sau la pierdere sau la remiză) pe baza experienţei anterioare.

Funcţia de evaluare cel mai frecvent utilizată presupune că valoarea unei piese poate fi stabilită independent de celelalte piese existente pe tablă. Un asemenea tip de funcţie de evaluare se numeşte *funcţie liniară ponderată*, întrucât are o expresie de forma

$$w_1f_1 + w_2f_2 + \ldots + w_nf_n,$$

unde valorile w_i, $i = \overline{1,n}$ reprezintă ponderile, iar f_i, $i = \overline{1,n}$ sunt caracteristicile unei anumite poziţii. În cazul jocului de şah, spre exemplu w_i, $i = \overline{1,n}$ ar putea fi valorile pieselor (1 pentru pion, 3 pentru nebun etc.), iar f_i, $i = \overline{1,n}$ ar reprezenta numărul pieselor de un anumit tip aflate pe tabla de şah.

În construirea formulei liniare trebuie mai întâi alese caracteristicile, operaţie urmată de ajustarea ponderilor până în momentul în care programul joacă suficient de bine. Această a doua operaţie poate fi automatizată punând programul să joace multe partide cu el însuşi, dar alegerea unor caracteristici adecvate nu a fost încă realizată în mod automat.

3.2.2. Implementare Prolog

Un program Prolog care calculează valoarea minimax a unui nod intern dat va avea ca relaţie principală pe


```
minimax(Poz,SuccBun,Val)
```

unde **Val** este valoarea minimax a unei poziţii **Poz**, iar **SuccBun** este cea mai bună poziţie succesor a lui **Poz** (i.e. mutarea care trebuie făcută pentru a se obţine **Val**). Cu alte cuvinte, avem:

minimax(Poz, SuccBun, Val) :
Poz este o poziţie, **Val** este valoarea ei de tip minimax;
cea mai bună mutare de la **Poz** conduce la poziţia **SuccBun**.

La rândul ei, relaţia

```
mutari(Poz,ListaPoz)
```

corespunde regulilor jocului care indică mutările "legale" (admise): **ListaPoz** este lista poziţiilor succesor legale ale lui **Poz**. Predicatul **mutari** va eşua dacă **Poz** este o poziţie de căutare terminală (o frunză a arborelui de căutare). Relaţia

```
celmaibun(ListaPoz,PozBun,ValBuna)
```

selectează cea mai bună poziţie **PozBun** dintr-o listă de poziţii candidate **ListaPoz**. **ValBuna** este valoarea lui **PozBun** şi, prin urmare, şi a lui **Poz**. „Cel mai bun" are aici sensul de maxim sau de minim, în funcţie de partea care execută mutarea.

O implementare a principiului Minimax este realizată de următorul program Prolog, preluat din [3]:

Principiul Minimax

```
minimax(Poz,SuccBun,Val):-
            % mutările legale de la Poz produc ListaPoz
            mutări(Poz,ListaPoz),!,
            celmaibun(ListaPoz,SuccBun,Val);
            %Poz nu are succesori şi este evaluat
            %în mod static
            staticval(Poz,Val).
celmaibun([Poz],Poz,Val):-
            minimax(Poz,_,Val),!.
celmaibun([Poz1|ListaPoz],PozBun,ValBuna):-
            minimax(Poz1,_,Val1),
            celmaibun(ListaPoz,Poz2,Val2),
            maibine(Poz1,Val1,Poz2,Val2,PozBun,ValBuna).

%Poz0 mai bună decât Poz1

maibine(Poz0,Val0,Poz1,Val1,Poz0,Val0):-
```

```
        %Min face o mutare la Poz0
        %Max preferă valoarea maximă
        mutare_min(Poz0),
        Val0>Val1,!
        ;
        %Max face o mutare la Poz0
        %Min preferă valoarea mai mică
        mutare_max(Poz0),
        Val0<Val1,!.

%Altfel, Poz1 este mai bună decât Poz0

maibine(Poz0,Val0,Poz1,Val1,Poz1,Val1).
```

3.3. O implementare eficientă a principiului Minimax: Algoritmul Alpha-Beta

Tehnica pe care o vom examina, în cele ce urmează, este numită în literatura de specialitate **alpha-beta prunning** ("alpha-beta retezare"). Atunci când este aplicată unui arbore de tip minimax standard, ea va întoarce aceeaşi mutare pe care ar furniza-o şi Algoritmul Minimax, dar într-un timp mai scurt, întrucât realizează *o retezare a unor ramuri ale arborelui care nu pot influenţa decizia finală*. Principiul general al acestei tehnici constă în a considera un nod oarecare *n* al arborelui, astfel încât jucătorul poate alege să facă o mutare la acel nod. Dacă acelaşi jucător dispune de o alegere mai avantajoasă, *m*, fie la nivelul nodului părinte al lui *n*, fie în orice punct de decizie aflat mai sus în arbore, atunci *n* nu va fi niciodată atins în timpul jocului. Prin urmare, de îndată ce, în urma examinării unora dintre descendenţii nodului *n*, ajungem să deţinem suficientă informaţie relativ la acesta, îl putem înlătura.

Ideea tehnicii de alpha-beta retezare este aceea de a găsi o mutare "suficient de bună", nu neapărat cea mai bună, dar suficient de bună pentru a se lua decizia corectă. Această idee poate fi formalizată prin introducerea a *două limite*, *alpha* şi *beta*, reprezentând limitări ale valorii de tip minimax corespunzătoare unui nod intern. Semnificaţia acestor limite este următoarea: *alpha* este *valoarea minimă* pe care este deja garantat că o va obţine MAX, iar *beta* este *valoarea maximă* pe care MAX poate spera să o atingă. Din punctul de vedere al jucătorului MIN, beta este valoarea cea mai nefavorabilă pentru MIN pe care acesta o va atinge. Prin urmare, *valoarea efectivă* care va fi găsită se află *între alpha şi beta*. Valoarea alpha, asociată nodurilor de tip MAX, nu poate niciodată să descrească, iar valoarea beta, asociată nodurilor de tip MIN, nu poate niciodată să crească. Dacă, spre exemplu, valoarea alpha a unui nod intern de tip MAX este 6, atunci MAX nu mai trebuie să ia în cosideraţie nici o valoare internă mai mică sau egală cu 6 care este asociată oricărui nod de tip

MIN situat sub el. Alpha este scorul cel mai prost pe care îl poate obţine MAX, presupunând că MIN joacă perfect. În mod similar, dacă MIN are valoarea beta 6, el nu mai trebuie să ia în consideraţie nici un nod de tip MAX situat sub el care are valoarea 6 sau o valoare mai mare decât acest număr. Cele două reguli pentru încheierea căutării, bazată pe valori alpha şi beta, pot fi formulate după cum urmează [19]:

1. Căutarea poate fi oprită dedesubtul oricărui nod de tip MIN care are o valoare beta mai mică sau egală cu valoarea alpha a oricăruia dintre strămoşii săi de tip MAX.
2. Căutarea poate fi oprită dedesubtul oricărui nod de tip MAX care are o valoare alpha mai mare sau egală cu valoarea beta a oricăruia dintre strămoşii săi de tip MIN.

Dacă, referitor la o poziţie, se arată că valoarea corespunzătoare ei se află în afara intervalului alpha-beta, atunci această informaţie este suficientă pentru a şti că poziţia respectivă nu se află de-a lungul *variaţiei principale*, chiar dacă nu este cunoscută valoarea exactă corespunzătoare ei. Cunoaşterea valorii exacte a unei poziţii este necesară numai atunci când această valoare se află între alpha şi beta.

Din punct de vedere formal, putem defini o *valoare de tip minimax* a unui nod intern, P, V(P, alpha, beta), ca fiind "*suficient de bună*" dacă satisface următoarele cerinţe:

$$\begin{aligned} &V(P, alpha, beta) < alpha, && \text{dacă } V(P) < alpha \\ &V(P, alpha, beta) = V(P), && \text{dacă } alpha \le V(P) \le beta \qquad (1) \\ &V(P, alpha, beta) > beta, && \text{dacă } V(P) > beta, \end{aligned}$$

unde prin V(P) am notat valoarea de tip minimax corespunzătoare unui nod intern.

Valoarea exactă a unui nod-rădăcină P poate fi întotdeauna calculată prin setarea limitelor după cum urmează:

$$V(P, -\infty, +\infty) = V(P).$$

Corespunzător arborelui din Fig. 3.2 procesul de căutare decurge după cum urmează:

1. Începe din poziţia *a*.
2. Mutare la *b*.
3. Mutare la *d*.
4. Alege valoarea maximă a succesorilor lui *d*, ceea ce conduce la V(*d*) = 4.
5. Întoarce-te în nodul *b* şi execută o mutare de aici la *e*.
6. Ia în consideraţie primul succesor al lui *e* a cărui valoare este 5. În acest moment, MAX, a cărui mutare urmează, are garantată, aflându-se în poziţia *e*, cel puţin valoarea 5, indiferent care ar fi celelalte alternative plecând din *e*. Această informaţie este suficientă pentru ca MIN să realizeze că, la nodul *b*, alternativa *e* este inferioară alternativei *d*.

Această concluzie poate fi trasă fără a cunoaşte valoarea *exactă* a lui *e*. Pe această bază, cel de-al doilea succesor al lui *e* poate fi neglijat, iar nodului *e* i se poate atribui valoarea *aproximativă* 5.

Fig. 3.2 ilustrează acţiunea Algoritmului Alpha-Beta în cazul arborelui din Fig. 3.1. Aşa cum se vede în figură, unele dintre valorile de tip minimax ale nodurilor interne sunt aproximative. Totuşi, aceste aproximări sunt suficiente pentru a se determina în mod exact valoarea rădăcinii. Se observă că Algoritmul Alpha-Beta reduce complexitatea căutării de la 8 evaluări statice la numai 5 evaluări de acest tip:

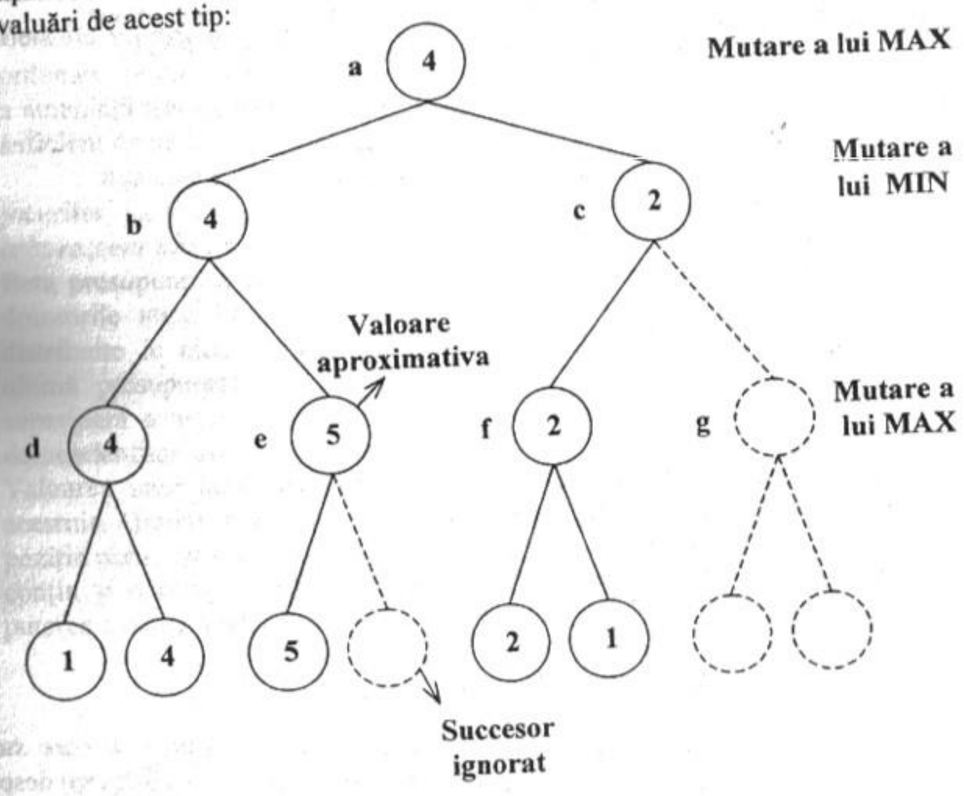


Fig. 3.2

Căutarea de tip alpha-beta retează nodurile figurate în mod discontinuu. Ca rezultat, câteva dintre valorile intermediare nu sunt exacte (nodurile *c*, *e*), dar aproximările făcute sunt suficiente pentru a determina atât valoarea corespunzătoare rădăcinii, cât şi variaţia principală, în mod exact. Un alt exemplu [19] de aplicare a Algoritmului Alpha-Beta este cel din Fig. 3.3, alpha-beta retezarea produce corespunzător spaţiului de stări din Fig. 3.3, alpha-beta retezarea produce arborele de căutare din Fig. 3.4, în care stările fără numere nu sunt evaluate.

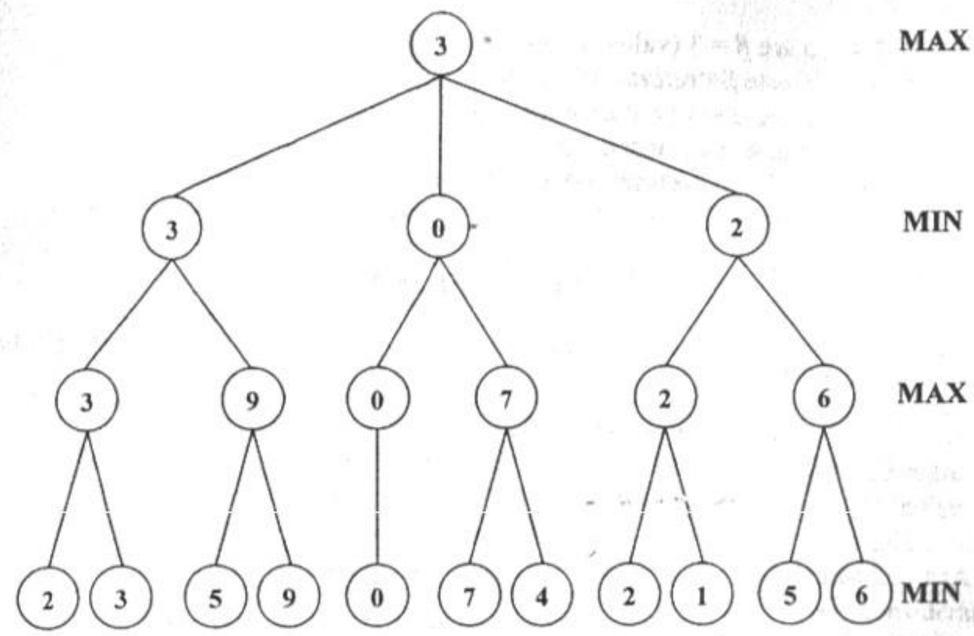


Fig. 3.3

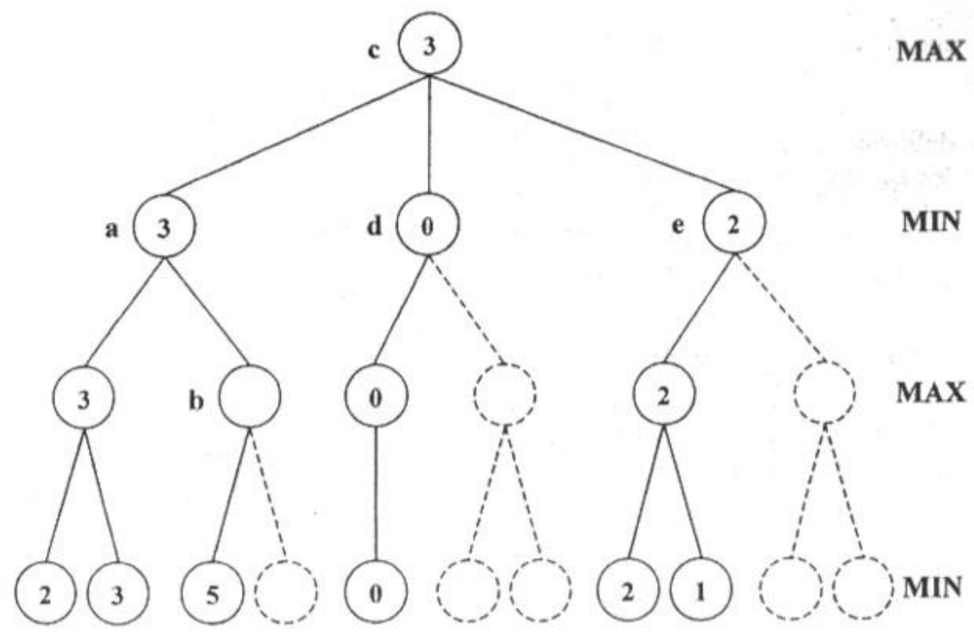


Fig. 3.4

În Fig. 3.4: a are $\beta = 3$ (valoarea lui a nu va depăşi 3);
b este β - retezat, deoarece $5 > 3$;
c are $\alpha = 3$ (valoarea lui c nu va fi mai mică decât 3);
d este α – retezat, deoarece $0 < 3$;
e este α – retezat, deoarece $2 < 3$;
c este 3.

3.3.1. Implementare Prolog

În cadrul unei implementări în Prolog a Algoritmului Alpha-Beta, relaţia principală este

```
alphabeta(Poz,Alpha,Beta,PozBuna,Val)
```

unde **PozBuna** reprezintă un succesor "suficient de bun" al lui **Poz**, astfel încât valoarea sa, **Val**, satisface cerinţele (i):

Val = V(Poz, Alpha, Beta)

Procedura

```
limitarebuna(ListaPoz,Alpha,Beta,PozBuna,Val)
```

găseşte, în lista **ListaPoz**, o poziţie suficient de bună, **PozBuna**, astfel încât valoarea de tip minimax, **Val**, a lui **PozBuna**, reprezintă o aproximaţie suficient de bună relativ la **Alpha** şi **Beta**.

Intervalul alpha-beta se poate îngusta (dar niciodată lărgi) în timpul apelărilor recursive, de la o mai mare adâncime, ale procedurii.

Relaţia

```
limitenoi(Alpha,Beta,Poz,Val,AlphaNou,BetaNou)
```

defineşte noul interval [**AlphaNou**, **BetaNou**]. Acesta este întotdeauna mai îngust sau cel mult egal cu vechiul interval [**Alpha**, **Beta**]. Îngustarea intervalului conduce la operaţii suplimentare de retezare a arborelui.

O implementare a Algoritmului Alpha-Beta este realizată de următorul program Prolog, preluat, de asemenea, din [3]:

<u>Algoritmul Alpha-Beta</u>

```
alphabeta(Poz,Alpha,Beta,PozBuna,Val ):-
      mutari(Poz,ListaPoz),!,
      limitarebuna(ListaPoz,Alpha,Beta,PozBuna,Val);
      %valoare statică a lui Poz
      staticval(Poz,Val).

limitarebuna([Poz|ListaPoz],Alpha,Beta,PozBuna,
ValBuna):-
      alphabeta(Poz,Alpha,Beta,_,Val),
      destuldebun(ListaPoz,Alpha,Beta,Poz,Val,PozBuna,
                  ValBuna).

%nu există alt candidat

destuldebun([ ],_,_,Poz,Val,Poz,Val):-!.
destuldebun(_,Alpha,Beta,Poz,Val,Poz,Val):-
      %atingere limită superioară
      mutare_min(Poz),Val > Beta,!;
      %atingere limită inferioară
      mutare_max(Poz),Val < Alpha,!.
destuldebun(ListaPoz,Alpha,Beta,Poz,Val,PozBuna,
ValBuna):-
      %rafinare limite
      limitenoi(Alpha,Beta,Poz,Val,AlphaNou,BetaNou),
      limitarebuna(ListaPoz,AlphaNou,BetaNou,Poz1,
                   Val1),
      maibine(Poz,Val,Poz1,Val1,PozBuna,ValBuna).

limitenoi(Alpha,Beta,Poz,Val,Val,Beta):-
      %creştere limită inferioară
      mutare_min(Poz),Val > Alpha,!.
limitenoi(Alpha,Beta,Poz,Val,Alpha,Val):-
      %descreştere limită superioară
      mutare_max(Poz),Val < Beta,!.
%altfel,limitele nu se schimbă
limitenoi(Alpha,Beta,_,_,Alpha,Beta).

%Poz mai bună ca Poz1
maibine(Poz,Val,Poz1,Val1,Poz,Val):-
      mutare_min(Poz),Val > Val1,!;
      mutare_max(Poz),Val < Val1,!.
%altfel, Poz1 mai bună
maibine(_,_,Poz1,Val1,Poz1,Val1).
```

3.3.2. Consideraţii privitoare la eficienţă

Eficienţa Algoritmului Alpha-Beta depinde de *ordinea în care sunt examinaţi succesorii.* Este preferabil să fie examinaţi mai întâi succesorii despre care se crede că ar putea fi cei mai buni. În mod evident, acest lucru nu poate fi realizat în întregime. Dacă el ar fi posibil, funcţia care ordonează succesorii ar putea fi utilizată pentru a se juca un joc perfect. În ipoteza în care această ordonare ar putea fi realizată, s-a arătat că Algoritmul Alpha-Beta nu trebuie să examineze, pentru a alege cea mai bună mutare, decât $O(b^{d/2})$ noduri, în loc de $O(b^d)$, ca în cazul Algoritmului Minimax. Aceasta arată că factorul de ramificare efectiv este $\sqrt{b}$ în loc de b – în cazul jocului de şah 6, în loc de 35. Cu alte cuvinte, Algoritmul Alpha-Beta poate "privi înainte" la o adâncime dublă faţă de Algoritmul Minimax, pentru a găsi acelaşi cost.

În cazul unei ordonări neprevăzute a stărilor în spaţiul de căutare, Algoritmul Alpha-Beta poate dubla adâncimea spaţiului de căutare (Nilsson 1980). Dacă există o anumită ordonare nefavorabilă a nodurilor, acest algoritm nu va căuta mai mult decât Algoritmul Minimax. Prin urmare, în cazul cel mai nefavorabil, Algoritmul Alpha-Beta nu va oferi nici un avantaj în comparaţie cu căutarea exhaustivă de tip minimax. În cazul unei ordonări favorabile însă, dacă notăm prin N numărul poziţiilor de căutare terminale evaluate în mod static de către Algoritmul Minimax, s-a arătat că, în cazul cel mai bun, adică atunci când mutarea cea mai puternică este prima luată în consideraţie, Algoritmul Alpha-Beta nu va evalua în mod static decât $\sqrt{N}$ poziţii. În practică, o funcţie de ordonare relativ simplă (cum ar fi încercarea mai întâi a capturilor, apoi a ameninţărilor, apoi a mutărilor înainte, apoi a celor înapoi) ne poate apropia suficient de mult de rezultatul obţinut în cazul cel mai favorabil.

Aşa cum se remarcă în [27], toate rezultatele cu privire la complexitatea jocurilor, ca şi a problemelor de căutare, în general, se bazează pe un *model arborescent idealizat*. Spre exemplu, modelul folosit pentru Algoritmul Alpha-Beta presupune că toate nodurile au acelaşi factor de ramificare, b; că toate drumurile ating limita de adâncime fixată, d; că evaluările frunzelor sunt distribuite în mod aleator de-a lungul ultimului nivel al arborelui. Această ultimă presupunere are cel mai serios neajuns: dacă o mutare din partea superioară a arborelui reprezintă o greşeală dezastruoasă, atunci majoritatea descendenţilor săi vor părea neavantajoşi jucătorului care a făcut greşeala. Valoarea unui nod pare a fi strâns corelată cu valorile nodurilor frate ale acestuia. Gradul de corelare depinde foarte mult de fiecare joc în parte şi de poziţia particulară corespunzătoare rădăcinii. Astfel, cercetările legate de jocuri conţin şi o componentă de ştiinţă empirică, care în mod inevitabil eludează puterea analizei matematice.

3.4. Un exemplu. Jocul X şi 0

Pentru a ilustra folosirea celor doi algoritmi Minimax si Alpha-Beta, considerăm drept exemplu aşa-numitul **Joc X şi 0**, care face parte din categoria jocurilor între două persoane, cu informaţie perfectă (completă).

Jocul X şi 0

Se consideră un tablou pătratic de dimensiune 3×3, ale cărui câmpuri sunt marcate alternativ de către doi jucători. Unul dintre jucători marchează cu X, iar celălalt cu 0. Câştigă acel jucător care marchează primul, cu simbolul său şi în întregime, fie o linie orizontală, fie una verticală (coloană), fie o linie diagonală a tabloului. Să se determine jucătorul câştigător şi configuraţia finală a tabloului.

Pentru rezolvarea acestei probleme vom defini funcţia de evaluare conform [25]. Pentru o poziţie p, definim $e(p)$ astfel:

- $e(p) = M$ (M - un număr foarte mare), dacă p este poziţie câştigătoare pentru MAX;
- $e(p) = -M$ (M - un număr foarte mare), dacă p este poziţie câştigătoare pentru MIN;
- $e(p) =$ numărul de linii deschise pentru MAX - numărul de linii deschise pentru MIN, dacă p nu este poziţie câştigătoare pentru nici unul dintre cei doi jucători; prin linie deschisă pentru MAX, respectiv pentru MIN înţelegem o linie orizontală, verticală sau diagonală care a fost parţial completată numai cu simbolul corespunzător lui MAX (MIN); o linie ce conţine numai câmpuri nemarcate poate fi considerată deschisă pentru ambii jucători.

Dacă jucătorul MAX este cel care marchează cu 0 (deci jucătorul MIN este cel care marchează cu X) şi dacă poziţia p este cea din Fig. 3.5, atunci $e(p) = 4 - 6 = -2$.

	0	
	x	

Fig. 3.5

În implementarea care urmează vom reprezenta o stare a problemei prin intermediul unui termen Prolog de forma

st(J,Tablou,N,MAX)

Aici **J** este jucătorul care face mutarea curentă (**J** poate fi X sau 0), **N** reprezintă numărul maxim de mutări care se pot face în adâncime (în algoritmii Minimax si Alpha-Beta), iar **Tablou** este o listă în care se memorează tabloul de joc, pe linii. **MAX** este jucătorul (X sau 0) corespunzător calculatorului, deci oponentul jucătorului uman.

Programul 3.1

```
%predicate specifice pentru jocul X si 0

juc_opus(x,0).
juc_opus(0,x).
```

```
%initial, tabloul este complet nemarcat

tablou_initial([gol,gol,gol,gol,gol,gol,gol,gol,
                gol]).

%Predicatul tablou_final(T) este adevarat fie daca T are
%o linie orizontala, o coloana sau o diagonala marcata in
%intregime de catre unul din jucatori, fie daca T este
%complet marcat, dar nici unul dintre jucatori nu este
%castigator (caz care corespunde unei situatii de remiza).

tablou_final([C,C,C|_],C):-C\==gol,!.
tablou_final([_,_,_,C,C,C|_],C):-C\==gol,!.
tablou_final([_,_,_,_,_,_,C,C,C],C):-C\==gol,!.
tablou_final([C,_,_,C,_,_,C|_],C):-C\==gol,!.
tablou_final([_,C,_,_,C,_,_,C|_],C):-C\==gol,!.
tablou_final([_,_,C,_,_,C,_,_,C],C):-C\==gol,!.
tablou_final([C,_,_,_,C,_,_,_,C],C):-C\==gol,!.
tablou_final([_,_,C,_,C,_,C|_],C):-C\==gol,!.
tablou_final(Tablou,gol):- \+ (membru(gol,Tablou)).

membru(X,[X|_]).
membru(X,[_|T]):-membru(X,T).

mutari(Poz,ListaPoz):-
  bagof(Poz1,mutare(Poz,Poz1),ListaPoz).

mutare(st(Jucator,T,N,M),st(Jucator_opus,Tablou1,
      N1,M)):-
            N>0,\+(tablou_final(T,_)),
            juc_opus(Jucator,Jucator_opus),N1 is N-1,
            marcheaza_pozitie(T,Jucator,Tablou1).

marcheaza_pozitie([gol|T],Jucator,[Jucator|T]).
marcheaza_pozitie([H|T],Jucator,[H|T1]):-
            marcheaza_pozitie(T,Jucator,T1).

%definim functia de evaluare statica

staticval(st(_,Tabl
ou,_,MAX),Val):-
            tablou_final(Tablou,T),!,juc_opus(MAX,MIN),
            (T==MIN, Val is -99;
            T==MAX, Val is  99;
            T==gol,Val is 0).
staticval(st(_,Tablou,_,MAX),Val):-
            juc_opus(MAX,MIN),
```



```
            nr_lin_deschise(MAX,L_D_2,Tablou),
            %determinam numarul de linii deschise
            %pentru MAX
            nr_lin_deschise(MIN,L_D_1,Tablou),
            %determinam numarul de linii deschise
            %pentru MIN
            Val is L_D_2-L_D_1.

%Predicatul nr_lin_deschise(XO,L_D,Tablou) este utilizat
%pentru determinarea numarului de linii deschise (L_D)
%pentru jucatorul XO, in cazul configuratiei (tabloului)
%Tablou.

nr_lin_deschise(XO,L_D,Tablou):-
            linia1(Tablou,L1,XO),
            linia2(Tablou,L2,XO),
            linia3(Tablou,L3,XO),
            coloana1(Tablou,C1,XO),
            coloana2(Tablou,C2,XO),
            coloana3(Tablou,C3,XO),
            diagonala1(Tablou,D1,XO),
            diagonala2(Tablou,D2,XO),
            L_D is L1+L2+L3+C1+C2+C3+D1+D2.

membru_l([],_).
membru_l([H|T],XO):-
            membru(H,[XO,gol]),membru_l(T,XO).

%Predicatul linia1(T,R,XO) este folosit pentru a determina
%daca prima linie orizontala a tabloului T este linie
%deschisa pentru jucatorul XO; daca da, atunci R=1, altfel
%R=0.

linia1([X,Y,Z|_],1,XO):-membru_l([X,Y,Z],XO),!.
linia1(_,0,_).

linia2([_,_,_,X,Y,Z|_],1,XO):-membru_l([X,Y,Z],XO),!.
linia2(_,0,_).

linia3([_,_,_,_,_,_,X,Y,Z],1,XO):-
            membru_l([X,Y,Z],XO),!.
linia3(_,0,_).

%Predicatul linia1(T,R,XO) este folosit pentru a determina
%daca prima coloana a tabloului T este linie deschisa
%pentru jucatorul XO; daca da, atunci R=1, altfel R=0.
```

```
coloana1([X,_,_,Y,_,_,Z|_],1,XO):-
            membru_l([X,Y,Z],XO),!.
coloana1(_,0,_).

coloana2([_,X,_,_,Y,_,_,Z|_],1,XO):-
            membru_l([X,Y,Z],XO),!.
coloana2(_,0,_).

coloana3([_,_,X,_,_,Y,_,_,Z],1,XO):-
            membru_l([X,Y,Z],XO),!.
coloana3(_,0,_).

%Predicatul   diagonala1(T,R,XO)   este   folosit   pentru
%a determina daca prima diagonala a tabloului T este linie
%deschisa pentru jucatorul XO; daca da, atunci R=1, altfel
%R=0

diagonala1([X,_,_,_,Y,_,_,_,Z],1,XO):-
            membru_l([X,Y,Z],XO),!.
diagonala1(_,0,_).

diagonala2([_,_,X,_,Y,_,Z|_],1,XO):-
            membru_l([X,Y,Z],XO),!.
diagonala2(_,0,_).

mutare_max(st(MAX,_,_,MAX)).

mutare_min(st(MIN,_,_,MAX)):-juc_opus(MAX,MIN).

%algoritmul minimax

minimax(Poz,SuccBun,Val):-
            mutari(Poz,ListaPoz),!,
            celmaibun(ListaPoz,SuccBun,Val);
            staticval(Poz,Val).

celmaibun([Poz],Poz,Val):-minimax(Poz,_,Val).
celmaibun([Poz1|ListaPoz],PozBun,ValBuna):-
            minimax(Poz1,_,Val1),
            celmaibun(ListaPoz,Poz2,Val2),
            maibine(Poz1,Val1,Poz2,Val2,PozBun,ValBuna).

maibine(Poz0,Val0,Poz1,Val1,Poz0,Val0):-
            mutare_min(Poz0),Val0>Val1,!;
            mutare_max(Poz0),Val0<Val1,!.
maibine(Poz0,Val0,Poz1,Val1,Poz1,Val1).
```



```
%algoritmul alpha-beta

alphabeta(Poz,Alpha,Beta,PozBuna,Val):-
            mutari(Poz,ListaPoz),!,
            limitarebuna(ListaPoz,Alpha,Beta,PozBuna,
                         Val);
            staticval(Poz,Val).

limitarebuna([Poz|ListaPoz],Alpha,Beta,PozBuna,
              ValBuna):-
            alphabeta(Poz,Alpha,Beta,_,Val),
            destuldebun(ListaPoz,Alpha,Beta,Poz,Val,
            PozBuna,ValBuna).

destuldebun([],_,_,Poz,Val,Poz,Val):-!.
destuldebun(_,Alpha,Beta,Poz,Val,Poz,Val):-
            mutare_min(Poz),Val>Beta,!;
            mutare_max(Poz),Val<Alpha,!.
destuldebun(ListaPoz,Alpha,Beta,Poz,Val,PozBuna,
            ValBuna):-
            limitenoi(Alpha,Beta,Poz,Val,AlphaNou,BetaNou),
            limitarebuna(ListaPoz,AlphaNou,BetaNou,Poz1,
                         Val1),
            maibine(Poz,Val,Poz1,Val1,PozBuna,ValBuna).

limitenoi(Alpha,Beta,Poz,Val,Val,Beta):-
            mutare_min(Poz),Val>Alpha,!.
limitenoi(Alpha,Beta,Poz,Val,Alpha,Val):-
            mutare_max(Poz),Val<Beta,!.
limitenoi(Alpha,Beta,_,_,Alpha,Beta).

%predicate specifice pentru jocul x si 0

x_si_o_minimax:-
            initializari(MAX,N),
            tablou_initial(Tablou),
            afis_tablou(Tablou),
            joc_minimax(st(x,Tablou,N,MAX)).
            %jucatorul care marcheaza cu X incepe jocul

x_si_o_alpha_beta:-
            initializari(MAX,N),
            tablou_initial(Tablou),
            afis_tablou(Tablou),
            joc_alpha_beta(st(x,Tablou,N,MAX)).
            %jucatorul care marcheaza cu X incepe jocul
```

```
%Predicatul    jucator_MAX(M)    este    folosit    pentru
%determinarea jucatorului care va face prima mutare; el
%este,   de   asemenea,   utilizat   pentru   determinarea
%jucatorului MAX.

jucator_MAX(M):-
        write('Incepe jocul ... '),nl,
        repeat,
        write('Vrei sa fii cu X ? (da/nu)'),nl,
        read(T),
        (T=da,M=0 ;T=nu, M=x;T=d,M=0;T=n,M=x),!.

initializari(M,N):-
        jucator_MAX(M),
        nl,repeat,
        write('Adancimea: '),
        read(N),integer(N),!,nl.

joc_minimax(st(_,Tablou,_,MAX)):-
            tablou_final(Tablou,J),!,
            anunt_castigator(J,MAX).
joc_minimax(st(J,Tablou,N,MAX)):-
            MAX\==J,!,juc_opus(J,J_O),
            mutare_MIN(Tablou,Tablou_urm,J),nl,
            afis_tablou(Tablou_urm),
            joc_minimax(st(J_O,Tablou_urm,N,MAX)).
joc_minimax(st(J,Tablou,N,MAX)):-
            MAX==J,juc_opus(J,J_O),
            minimax(st(J,Tablou,N,MAX),
            st(J_O,Tablou_urm,_,MAX),_),
            nl,write('Mutarea mea este: '),nl,nl,
            afis_tablou(Tablou_urm),nl,
            joc_minimax(st(J_O,Tablou_urm,N,MAX)).

joc_alpha_beta(st(_,Tablou,_,MAX)):-
            tablou_final(Tablou,J),!,
            anunt_castigator(J,MAX).
joc_alpha_beta(st(J,Tablou,N,MAX)):-
            MAX\==J,!,juc_opus(J,J_O),
            mutare_MIN(Tablou,Tablou_urm,J),nl,
            afis_tablou(Tablou_urm),
            joc_alpha_beta(st(J_O,Tablou_urm,N,MAX)).
joc_alpha_beta(st(J,Tablou,N,MAX)):-
            MAX==J,juc_opus(J,J_O),
            alphabeta(st(J,Tablou,N,MAX),
                      -500,500,st(J_O,Tablou_urm,_,MAX),_),
            nl,write('Mutarea mea este: '),nl,nl,
```



```
            afis_tablou(Tablou_urm),nl,
            joc_alpha_beta(st(J_O,Tablou_urm,N,MAX)).

anunt_castigator(J,MAX):-
            J==MAX,write('Ai pierdut!'),nl;
            juc_opus(J,J_O),J_O==MAX,
            write('Ai castigat!'),nl,
            write('Bravo !!');
            J==gol, write('Jocul s-a incheiat cu remiza '),nl.

mutare_MIN(Tablou,Tablou_urm,J):-
            nl,write('Trebuie sa marchezi. '),nl,
            write('Specifica campul pe care il vei marca.'),
            nl,repeat,det_coord_1(L),
            det_coord_2(C),
            N is L*3+C,
            liber_marcheaza(N,Tablou,Tablou_urm,J),!.

det_coord_1(L):-
            repeat,write('Linia: '),
            read(L1),integer(L1),!,
            L is L1-1.

det_coord_2(C):-
            repeat,write('Coloana: '),
            read(C1),integer(C1),!,
            C is C1-1.

liber_marcheaza(0,[gol|R],[J|R],J):-!.
liber_marcheaza(N,[H|T],[H|T1],J):-
            N1 is N-1,liber_marcheaza(N1,T,T1,J).

afis_tablou([C1,C2,C3,C4,C5,C6,C7,C8,C9]):-
            write('   1 2 3 '),nl,nl,
            write('1   '),scrie_elem(C1),
            tab(1),scrie_elem(C2),
            tab(1),scrie_elem(C3),nl,
            write('2   '),scrie_elem(C4),
            tab(1),scrie_elem(C5),
            tab(1),scrie_elem(C6),nl,
            write('3   '),scrie_elem(C7),
            tab(1),scrie_elem(C8),
            tab(1),scrie_elem(C9),nl.

scrie_elem(X):-X==gol,!,write('.');
            X==x,!,write('X');
            write('0').
```

Interogarea Prologului se face apelând predicatele **x_si_o_minimax** dacă se doreşte folosirea algoritmului minimax şi respectiv **x_si_o_alpha_beta** pentru folosirea algoritmului alpha-beta. Iată nişte exemple de execuţie ale programului:

1. Unul dintre jucători câştigă

```
?- x_si_o_minimax.
Incepe jocul...
Vrei sa fii cu X? (da/nu)
|: nu.
Adancimea: 6.

  1 2 3
1 . . .
2 . . .
3 . . .

Mutarea mea este:

  1 2 3
1 . . .
2 . X .
3 . . .

Trebuie sa marchezi.
Specifica campul pe care il vei marca.
Linia: 1.
Coloana: 2.

  1 2 3
1 . 0 .
2 . X .
3 . . .

Mutarea mea este:

  1 2 3
1 . 0 .
2 . X .
3 . . X

Trebuie sa marchezi.
Specifica campul pe care il vei marca.
Linia: 1.
Coloana: 1.
```



```
  1 2 3
1 0 0 .
2 . X .
3 . . X

Mutarea mea este:

  1 2 3
1 0 0 X
2 . X .
3 . . X

Trebuie sa marchezi.
Specifica campul pe care il vei marca.
Linia: 3.
Coloana: 1.

  1 2 3
1 0 0 X
2 . X .
3 0 . X

Mutarea mea este:

  1 2 3
1 0 0 X
2 . X X
3 0 . X

Ai pierdut!

Yes
```

2. Remiză

```
?- x_si_o_minimax.
Incepe jocul...
Vrei sa fii cu X? (da/nu)
|: da.
Adancimea: 6.

  1 2 3
1 . . .
2 . . .
3 . . .
```

Trebuie sa marchezi.
Specifica campul pe care il vei marca.
Linia: 2.
Coloana: 2.

```
  1 2 3
1 . . .
2 . X .
3 . . .
```

Mutarea mea este:

```
  1 2 3
1 . . .
2 . X .
3 . . 0
```

Trebuie sa marchezi.
Specifica campul pe care il vei marca.
Linia: 1.
Coloana: 1.

```
  1 2 3
1 X . .
2 . X .
3 . . 0
```

Mutarea mea este:

```
  1 2 3
1 X . .
2 . X .
3 0 . 0
```

Trebuie sa marchezi.
Specifica campul pe care il vei marca.
Linia: 3.
Coloana: 2.

```
  1 2 3
1 X . .
2 . X .
3 0 X 0
```

Mutarea mea este:


```
  1 2 3
1 X 0 .
2 . X .
3 0 X 0
```

Trebuie sa marchezi.
Specifica campul pe care il vei marca.
Linia: 2.
Coloana: 1.

```
  1 2 3
1 X 0 .
2 X X .
3 0 X 0
```

Mutarea mea este:

```
  1 2 3
1 X 0 .
2 X X 0
3 0 X 0
```

Trebuie sa marchezi.
Specifica campul pe care il vei marca.
Linia: 1.
Coloana: 3.

```
  1 2 3
1 X 0 .
2 X X 0
3 0 X 0
```

Jocul s-a incheiat cu remiza

yes

CAPITOLUL 4
ASPECTE ALE REPREZENTĂRII CUNOŞTINŢELOR

Pentru a rezolva problemele complexe întâlnite în domeniul inteligenţei artificiale este nevoie atât de o mare cantitate de cunoştinţe, cât şi de mecanisme specifice pentru manipularea acestor cunoştinţe cu scopul găsirii de soluţii ale unor noi probleme. Cele două tipuri diferite de entităţi pe care le tratează diversele modalităţi de reprezentare a cunoştinţelor sunt:

- *Faptele* - reprezentând adevăruri într-o lume relevantă. Acestea sunt entităţile pe care dorim să le reprezentăm.
- *Reprezentări ale faptelor într-un anumit formalism.* Acestea sunt entităţile care vor putea fi efectiv manipulate.

  Structurarea acestor entităţi se poate face la două niveluri diferite:
  - *Nivelul cunoştinţelor*, nivel la care sunt descrise faptele (inclusiv comportamentul fiecărui agent şi scopurile lui curente).
  - *Nivelul simbolurilor*, la care reprezentările obiectelor de la nivelul cunoştinţelor sunt definite în termenii unor simboluri care pot fi manipulate de către programele de calculator.

Un bun sistem de reprezentare a cunoştinţelor într-un anumit domeniu ar trebui să posede [8] următoarele patru proprietăţi:

- *Adecvare în reprezentare* - abilitatea de a reprezenta toate tipurile de cunoştinţe care sunt necesare într-un anumit domeniu.
- *Adecvare inferenţială* - abilitatea de a manipula structurile reprezentaţionale într-un asemenea mod încât să poată fi derivate structuri noi, corespunzătoare cunoştinţelor noi deduse din cele vechi.
- *Eficienţa inferenţială* - abilitatea de a încorpora în structura de cunoştinţe informaţii adiţionale care să poată fi folosite pentru a canaliza atenţia mecanismului de inferenţă în direcţiile cele mai promiţătoare.
- *Eficienţa în achiziţie* - abilitatea de a achiziţiona uşor informaţii noi. În mod ideal, însusi programul ar trebui să poată controla achiziţia de cunoştinţe.

Nici un sistem unic nu poate optimiza toate aceste caracteristici, nu în ultimul rând datorită tipurilor extrem de diverse de cunoştinţe care există, majoritatea programelor bazându-se pe tehnici multiple.

4.1. Tipuri de cunoştinţe

4.1.1. Cunoştinţe relaţionale simple

Cea mai simplă modalitate de reprezentare a faptelor declarative constă în folosirea unei mulţimi de relaţii de acelaşi tip cu cele utilizate în sistemele de baze de date. Un exemplu de sistem relaţional este cel din Fig. 4.1. Cunoştinţele relaţionale din acest tabel corespund unei mulţimi de atribute şi de valori asociate, care împreună descriu obiectele bazei de cunoştinţe.

Student	Vârstă	An de studiu	Note la informatică
Popescu Andrei	18	I	8-9
Ionescu Maria	18	I	9-10
Hristea Oana	20	I	7-8
Pârvu Ana	19	II	8-9
Savu Andrei	19	II	7-8
Popescu Ana	20	III	9-10

Fig. 4.1

Această reprezentare este simplă deoarece, în sine, nu posedă şi nu furnizează capacitatea de inferenţă. Dar cunoştinţele reprezentate în acest mod pot constitui input-ul adecvat pentru motoare de inferenţă mult mai puternice. Sistemele de baze de date sunt proiectate tocmai cu scopul de a furniza suportul necesar cunoştinţelor relaţionale.

4.1.2. Cunoştinţe care se moştenesc

Cunoştinţele despre obiecte, atributele lor şi valorile acestora sunt adesea mult mai complexe decât permite modul de reprezentare din Fig. 4.1. În particular, este posibil ca reprezentarea de bază să fie îmbogăţită cu mecanisme de inferenţă care operează asupra structurii reprezentării. Pentru ca această modalitate de reprezentare să fie eficientă, structura trebuie proiectată în aşa fel încât ea să corespundă mecanismelor de inferenţă dorite. Una dintre cele mai utilizate forme de inferenţă este ***moştenirea proprietăţilor***, prin care elemente aparţinând anumitor clase *moştenesc atribute şi valori* provenite de la clase mai generale, în care sunt incluse. Pentru a admite moştenirea proprietăţilor, *obiectele* trebuie să fie organizate în *clase*, iar clasele trebuie să fie aranjate în cadrul unei ***ierarhii***.

În Fig. 4.2 sunt reprezentate cunoştinţe legate de jocul de fotbal, cunoştinţe organizate într-o structură de acest tip. În această reprezentare, liniile desemnează atribute. Nodurile figurate prin dreptunghiuri reprezintă obiecte şi valori ale atributelor obiectelor. Aceste valori pot fi, la rândul lor, privite ca obiecte având atribute şi valori ş.a.m.d.. Săgeţile corespunzătoare liniilor sunt orientate de la un obiect la valoarea lui (de-a lungul liniei desemnând atributul corespunzător).

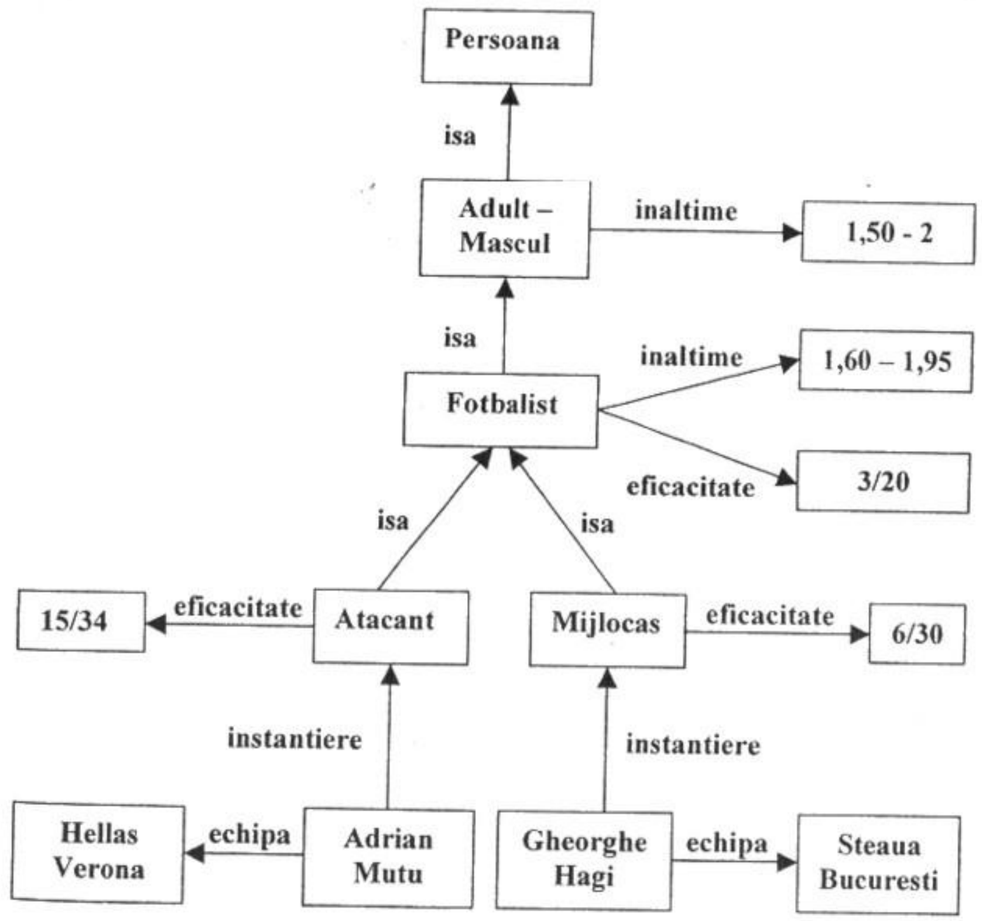


Fig. 4.2

În acest exemplu, structura corespunzătoare reprezintă o *structură de tip "slot-and-filler"*. Ea mai poate fi privită şi ca o *reţea semantică* sau ca o colecţie de *cadre*. În cel din urmă caz, fiecare cadru individual reprezintă colecţia atributelor şi a valorilor asociate cu un anumit nod. O diferenţiere exactă

a acestor tipuri de reprezentări este greu de făcut datorită marii flexibilităţi care există în reprezentarea cunoştinţelor. În general, termenul de sistem de cadre implică existenţa unei mai mari structurări a atributelor şi a mecanismelor de inferenţă care le pot fi aplicate decât în cazul reţelelor semantice. Vom reveni asupra ambelor modalităţi de reprezentare a cunoştinţelor.

În exemplul din Fig. 4.2., toate obiectele şi majoritatea atributelor care intervin corespund domeniului sportiv al jocului de fotbal şi nu au o semnificaţie generală. Singurele două excepţii sunt atributul *isa*, utilizat pentru a desemna *incluziunea între clase* şi atributul *instanţiere*, folosit pentru a arăta *apartenenţa la o clasă*. Aceste două atribute, extrem de folosite, se află la baza *moştenirii proprietăţilor* ca tehnică de inferenţă.

Utilizând această tehnică de inferenţă, *baza de cunoştinţe* poate asigura atât regăsirea faptelor care au fost memorate în mod explicit, precum şi a faptelor care derivă din cele memorate în mod explicit, ca în următorul exemplu:

eficacitate(Adrian_Mutu) = 15/34

Pentru a găsi răspunsul la această interogare, întrucât nu există nici o valoare pentru eficacitate memorată în mod explicit corespunzător lui Adrian_Mutu, a fost urmat atributul *instanţiere* până la *Atacant* şi a fost extrasă valoarea memorată acolo. Se poate acum observa una dintre caracteristicile negative ale moştenirii proprietăţilor, şi anume aceea că această tehnică de inferenţă poate produce valori implicite care nu sunt garantate a fi corecte, dar care reprezintă cele mai bune aproximări în lipsa unor informaţii exacte. Această tehnică de inferenţă continuă să fie printre cele mai folosite.

4.1.3. Cunoştinţe inferenţiale

Moştenirea proprietăţilor este o formă de inferenţă puternică, dar nu este suficientă pentru a construi un sistem de reprezentare complet, care, de cele mai multe ori, combină mai multe tehnici de reprezentare a cunoştinţelor.

Puterea logicii tradiţionale este adesea utilă pentru a se descrie toate inferenţele necesare. Desigur, astfel de cunoştinţe nu sunt utile decât în prezenţa unei proceduri de inferenţă care să le poată exploata. Procedura de inferenţă necesară în acest caz este una care implementează regulile logice de inferenţă standard. Există multe asemenea proceduri, dintre care unele fac raţionamente de tipul "*înainte*", de la fapte date către concluzii, iar altele raţionează "*înapoi*", de la concluziile dorite la faptele date. Una dintre procedurile cele mai folosite de acest tip este *rezoluţia*, care foloseşte strategia contradicţiei.

În general, logica furnizează o structură puternică în cadrul căreia sunt descrise legăturile dintre valori. Ea se combină adesea cu un alt limbaj puternic de descriere, cum ar fi o ierarhie de tip *isa*.

4.1.4. Cunoştinţe procedurale

Reprezentarea cunoştinţelor descrisă până în prezent s-a concentrat asupra *faptelor statice, declarative*. Un alt tip de cunoştinţe extrem de utile sunt *cunoştinţele* ***procedurale*** sau *operaţionale*, care specifică ce anume trebuie făcut şi când.

Cunoştinţele procedurale pot fi reprezentate în programe în diverse moduri. Cea mai simplă modalitate este, desigur, cea sub formă de cod, într-un anumit limbaj de programare. În acest caz, maşina foloseşte cunoştinţele atunci când execută codul pentru a efectua o anumită sarcină. Acest mod de reprezentare a cunoştinţelor procedurale nu este însă cel mai fericit din punctul de vedere al adecvării inferenţiale, precum şi al eficienţei în achiziţie. Din această cauză s-au căutat alte modalităţi diferite, de reprezentare a cunoştinţelor procedurale, astfel încât acestea să poată fi manipulate relativ uşor atât de către oameni, cât şi de către alte programe.

Cea mai folosită tehnică de reprezentare a cunoştinţelor procedurale în programele de inteligenţă artificială este aceea a utilizării *regulilor de producţie*. Atunci când sunt îmbogăţite cu informaţii asupra felului în care trebuie să fie folosite, regulile de producţie sunt mai procedurale decât alte metode existente de reprezentare a cunoştinţelor.

Regulile de producţie, numite şi reguli de tip **if-then**, sunt instrucţiuni condiţionale, care pot avea diverse interpretări, cum ar fi:

- **if** precondiţie P **then** concluzie C
- **if** situaţie S **then** acţiune A
- **if** condiţiile C1 şi C2 sunt verificate **then** condiţia C nu este verificată.

Regulile de producţie sunt foarte utilizate în proiectarea *sistemelor expert*. Vom reveni asupra lor, precum şi a implementării lor în limbajul Prolog.

A face o distincţie clară între cunoştinţele declarative şi cele procedurale este foarte dificil. Diferenţa esenţială între ele este dată de modul în care cunoştinţele sunt folosite de către procedurile care le manipulează.

4.2. Clase de metode pentru reprezentarea cunoştinţelor

Principalele tipuri de reprezentări ale cunoştinţelor sunt reprezentările *bazate pe logică* şi cele de tip "*slot-filler*" ("deschizătură-umplutură").

Reprezentările bazate pe logică aparţin unor două mari categorii, în funcţie de instrumentele folosite în reprezentare, şi anume:

- Logica - mecanismul principal îl constituie inferenţa logică.
- Regulile (folosite, de pildă, în sistemele expert) - principalele mecanisme sunt "*înlănţuirea înainte*" şi "*înlănţuirea înapoi*". O regulă

este similară unei implicații logice, dar nu are o valoare proprie (regulile sunt *aplicate*, ele nu au una dintre valorile true sau false).

Reprezentările de tip slot-filler folosesc două categorii diferite de structuri:

- Rețele semantice și grafuri conceptuale - o reprezentare "distribuită" (concepte legate între ele prin diverse relații). Principalul mecanism folosit este ***căutarea***.
- Cadre și scripturi - o reprezentare structurată (grupuri de concepte și relații); sunt foarte utile în reprezentarea tipicității. Principalul mecanism folosit este *împerecherea* (potrivirea) *șabloanelor* (tiparelor).

4.3. Reprezentarea cunoștințelor în sistemele expert

Un *sistem expert* este un program care se comportă ca un expert într-un domeniu relativ restrâns. Caracteristica majoră a sistemelor expert, numite și *sisteme bazate pe cunoștințe*, este aceea că ele se bazează pe cunoștințele unui expert uman în domeniul care este studiat. Mai exact, ele se sprijină pe cunoștințele expertului uman asupra strategiilor de rezolvare a problemelor specifice domeniului. Astfel, la baza sistemelor expert se află utilizarea în rezolvarea problemelor a unor mari cantități de cunoștințe specifice domeniului.

Sistemele expert[16] trebuie să poată rezolva probleme care necesită cunoștințe într-un anumit domeniu. Prin urmare, ele trebuie să posede aceste cunoștințe într-o anumită formă. Din această cauză ele se mai numesc și sisteme bazate pe cunoștințe. Nu toate sistemele bazate pe cunoștințe constituie însă sisteme expert. Un sistem expert trebuie, în plus, să fie capabil *să explice* utilizatorului comportamentul său și deciziile luate, la fel cum o fac experții umani. Această caracteristică referitoare la generarea explicațiilor este necesară în special în domeniile în care intervine *incertitudinea*, cum ar fi diagnosticarea medicală. Numai în acest fel utilizatorul poate detecta o posibilă fisură în raționamentul sistemului.

O caracteristică suplimentară care este adesea cerută sistemelor expert este, prin urmare, abilitatea de a trata incertitudinea sau starea de a fi incomplet. Astfel, informațiile asupra problemei care trebuie rezolvată pot fi incomplete sau de natură a nu inspira încredere, iar relațiile din domeniul problemei pot fi aproximative. Spre exemplu, un anumit medicament poate genera anumite probleme, dar *cel mai adesea* nu o face.

Pentru a construi un sistem expert este necesară, în general, definirea următoarelor două funcții:

[16] A căror tratare se face aici conform [3].

- o *funcție de rezolvare a problemei*, funcție capabilă să folosească cunoștințele specifice domeniului și care trebuie să poată trata incertitudinea.
- o *funcție de interacțiune cu utilizatorul*, care asigură și generarea de explicații asupra intențiilor sistemului și a deciziilor luate de acesta în timpul și după procesul de rezolvare a problemei.

Fiecare dintre aceste funcții poate fi foarte complicată și depinde în mod nemijlocit de domeniul de aplicație și de cerințele practice care pot să apară.

4.3.1. Structura de bază a unui sistem expert

Un sistem expert conține trei module principale, și anume:

- o bază de cunoștințe;
- un motor de inferență;
- o interfață cu utilizatorul.

Baza de cunoștințe cuprinde cunoștințele specifice unui domeniu, inclusiv fapte simple referitoare la domeniul de aplicație, reguli care descriu relațiile și fenomenele proprii domeniului și, eventual, metode, euristici și idei pentru rezolvarea problemelor domeniului.

Motorul de inferență utilizează în mod activ cunoștințele din baza de cunoștințe.

Interfața cu utilizatorul asigură comunicarea dintre utilizator și sistem, oferind utilizatorului o privire asupra procesului de rezolvare a problemei executat de către motorul de inferență. De cele mai multe ori motorul de inferență și interfața sunt privite ca un unic modul, numit *shell* (înveliș).

Structura de bază a unui sistem expert este prezentată în Fig. 4.3:

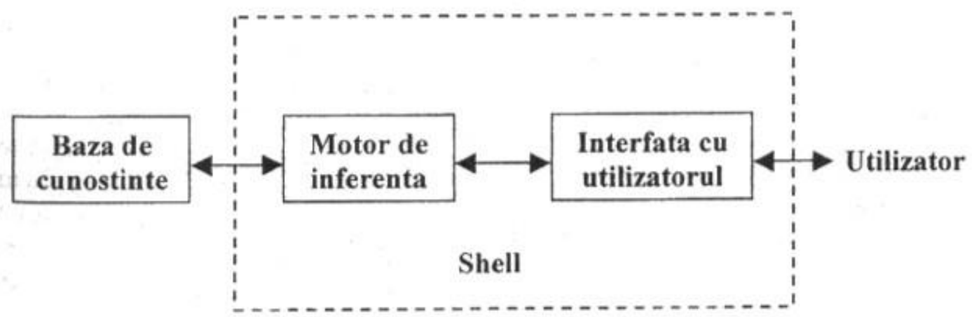


Fig. 4.3

Schema din Fig. 4.3 separă cunoștințele de algoritmii care folosesc aceste cunoștințe. Această separare este convenabilă din următoarele motive: baza de cunoștințe depinde în mod clar de aplicație. Pe de altă parte, învelișul

este, în principiu, independent de domeniu. O modalitate de a dezvolta sisteme expert pentru diverse aplicații, în aceste condiții, constă din dezvoltarea unui *shell* care poate fi folosit în mod universal, cu utilizarea unei baze de cunoștințe diferite corespunzător fiecărei aplicații. Desigur, toate bazele de cunoștințe utilizate vor trebui să se conformeze unui același formalism care este înțeles de către *shell*. În practică, această abordare nu va avea succes decât în cazul în care domeniile de aplicație sunt extrem de similare, dar, chiar dacă modificări ale învelișului vor fi necesare, atunci când se trece de la un domeniu la altul, principiile de bază pot fi reținute.

4.3.2. Reprezentarea cunoștințelor cu reguli if-then

Regulile de tip ***if-then***, numite și *reguli de producție*, constituie o formă naturală de exprimare a cunoștințelor și au următoarele caracteristici suplimentare:

- *Modularitate*: fiecare regulă definește o cantitate de cunoștințe relativ mică și independentă de celelalte.
- *Incrementabilitate*: noi reguli pot fi adăugate bazei de cunoștințe în mod relativ independent de celelalte reguli.
- *Modificabilitate* (ca o consecință a modularității): regulile vechi pot fi modificate relativ independent de celelalte reguli.
- Susțin *transparența* sistemului.

Această ultimă proprietate este o caracteristică importantă a sistemelor expert. Prin *transparența sistemului* se înțelege abilitatea sistemului de a explica deciziile și soluțiile sale. Regulile de producție facilitează generarea răspunsului pentru următoarele două tipuri de întrebări ale utilizatorului:

- întrebare de tipul "*cum*": ***Cum*** ai ajuns la această concluzie?
- întrebare de tipul "*de ce*": ***De ce*** te interesează această informație?

Regulile de tip **if-then** adesea definesc relații logice între conceptele aparținând domeniului problemei. Relațiile pur logice pot fi caracterizate ca aparținând așa-numitelor *cunoștințe categorice*, adică acelor cunoștințe care vor fi întotdeauna adevărate. În unele domenii însă, cum ar fi diagnosticarea în medicină, predomină cunoștințele "moi" sau *probabiliste*. În cazul acestui tip de cunoștințe, regularitățile empirice sunt valide numai până la un anumit punct (adesea, dar nu întotdeauna). În astfel de cazuri, regulile de producție pot fi modificate prin adăugarea la interpretarea lor logică a unei calificări de verosimilitate, obținându-se reguli de forma următoare:

if conditie A **then** concluzie B **cu certitudinea** F

Pentru a exemplifica folosirea regulilor de producție, vom lua în considerație baza de cunoștințe din Fig. 4.4, care își propune să trateze problema scurgerii de apă în apartamentul din aceeași figură. O scurgere de apă poate interveni fie în baie, fie în bucătărie. În ambele situații, scurgerea provoacă

o problemă (inundație) și în hol. Această bază de cunoștințe[17] simplistă nu presupune decât existența defectelor unice: problema poate fi la baie sau la bucătărie, dar nu în ambele locuri în același timp. Ea este reprezentată în Fig. 4.4 sub forma unei *rețele de inferență*:

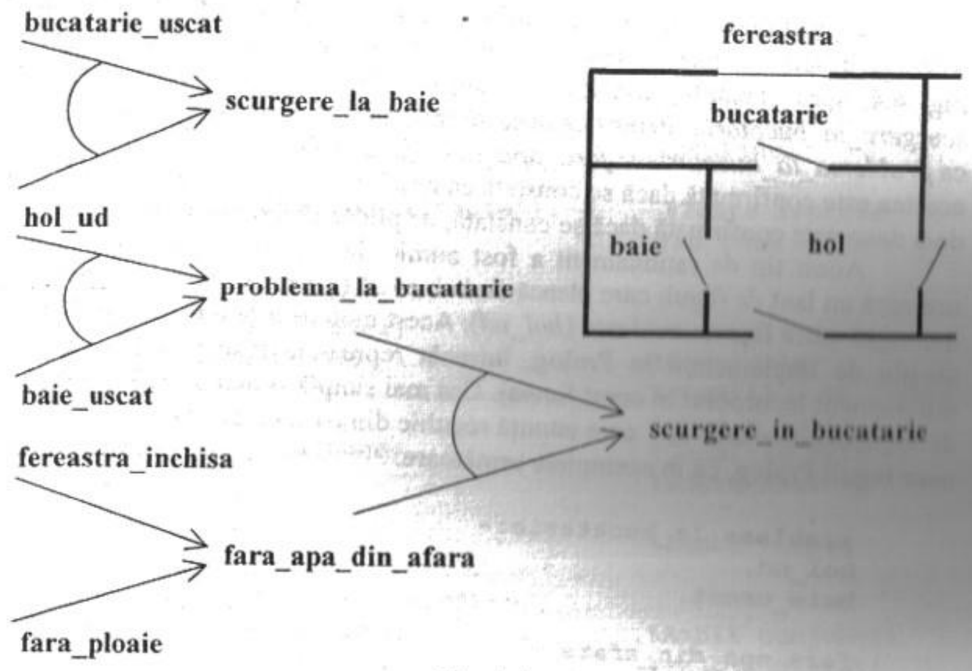


Fig. 4.4

Nodurile rețelei corespund propozițiilor, iar legăturile corespund regulilor din baza de cunoștințe. Arcele care conectează unele dintre legături indică conexiunea conjunctivă existentă între propozițiile corespunzătoare. În consecință, regula referitoare la existența unei probleme în bucătărie, în cadrul acestei rețele, este:

if hol_ud si baie_uscat **then** problema_la_bucatarie.

4.3.3. Înlănțuire înainte și înapoi în sistemele bazate pe reguli

Atunci când cunoștințele sunt reprezentate într-o anumită formă, este nevoie de o procedură de raționament care să tragă concluzii derivate din baza de cunoștințe. În cazul regulilor de tip **if-then**, există două modalități de a raționa, ambele extrem de ușor de implementat în Prolog [3], și anume:

[17] Preluată de noi, pentru exemplificare, din [3].

- înlănţuire înapoi ("backward chaining");
- înlănţuire înainte ("forward chaining").

4.3.3.1. Înlănţuirea înapoi

Raţionamentul de tip "înlănţuire înapoi" pleacă de la o ipoteză şi apoi parcurge în sensul "înapoi" reţeaua de inferenţă. În cazul bazei de cunoştinţe din Fig. 4.4, spre exemplu, una dintre ipotezele de la care se poate pleca este *scurgere_in_bucatarie*. Pentru ca această ipoteză să fie confirmată, este nevoie ca *problema_la_bucatarie* şi *fara_apa_din_afara* să fie adevărate. Prima dintre acestea este confirmată dacă se constată că holul este ud şi baia este uscată. Cea de-a doua este confirmată dacă se constată, de pildă, că fereastra este închisă.

Acest tip de raţionament a fost numit "înlănţuire înapoi" deoarece el urmează un lanţ de reguli care pleacă de la ipoteză (*scurgere_in_bucatarie*) şi se îndreaptă către faptele evidente (*hol_ud*). Acest mod de a raţiona este extrem de simplu de implementat în Prolog, întrucât reprezintă însuşi mecanismul de raţionament încorporat în acest limbaj. Cea mai simplă şi mai directă modalitate de implementare este cea care enunţă regulile din baza de cunoştinţe sub forma unor reguli Prolog, ca în exemplele următoare:

```
problema_la_bucatarie:-
hol_ud,
baie_uscat.

fara_apa_din_afara:-
fereastra_inchisa
;
fara_ploaie.
```

Faptele observate ca fiind evidente pot fi reprezentate sub forma unor fapte Prolog de tipul:

```
hol_ud.
baie_uscat.
fereastra_inchisa.
```

Ipoteza de la care s-a plecat poate fi acum verificată printr-o interogare a Prologului de tipul următor:

```
?- scurgere_in_bucatarie.
yes
```


În această implementare, în cazul regulilor, a fost folosită însăşi sintaxa limbajului Prolog. Această abordare prezintă anumite dezavantaje, printre care faptul că expertul în domeniu care utilizează baza de cunoştinţe trebuie să fie familiarizat cu limbajul Prolog, întrucât el trebuie să citească regulile, să le poată modifica şi să poată specifica reguli noi. Un al doilea dezavantaj al acestei implementări este faptul că baza de cunoştinţe nu se poate distinge, din punct de vedere sintactic, de restul programului. Pentru a face distincţia dintre baza de cunoştinţe şi restul programului mai clară, sintaxa regulilor expert poate fi modificată prin folosirea unor operatori definiţi de utilizator. Spre exemplu, pot fi folosiţi ca operatori *if*, *then*, *and* şi *or*, declaraţi în felul următor:

```
:- op(800,fx,if).
:- op(700,xfx,then).
:- op(300,xfy,or).
:- op(200,xfy,and).
```

Regulile pot fi scrise atunci sub forma:

```
if
      hol_ud and bucatarie_uscat
then
      scurgere_la_baie.
if
      fereastra_inchisa or fara_ploaie
then
      fara_apa_din_afara.
```

Faptele observate pot fi enunţate sub forma unei proceduri pe care o vom numi `fapta`:

```
fapta (hol_ud).
fapta (baie_uscat).
fapta (fereastra_inchisa).
```

În această nouă sintaxă este nevoie de un nou interpretor pentru reguli. Un astfel de interpretor poate fi definit sub forma procedurii

`este_adevarat(P)`

unde propoziţia `P` este fie dată prin intermediul procedurii `fapta`, fie poate fi derivată prin utilizarea regulilor. Noul interpretor, aşa cum este el propus în [3], este următorul:

```
:- op(800,fx,if).
:- op(700,xfx,then).
:- op(300,xfy,or).
```

```
:- op(200,xfy,and).

este_adevarat(P):-
      fapta(P).

este_adevarat(P):-
      if Conditie then P,
      este_adevarat(Conditie).

este_adevarat(P1 and P2):-
      este_adevarat(P1),
      este_adevarat(P2).

este_adevarat(P1 or P2):-
      este_adevarat(P1)
      ;
      este_adevarat(P2).
```

Se observă că acest interpretor pentru reguli **if-then** de tip "înlănţuire înapoi" continuă să lucreze înapoi în maniera depth-first.
Interogarea Prologului se face acum în felul următor:

```
? : -este_adevarat(scurgere_in_bucatarie).
yes
```

Principalul dezavantaj al procedurilor de inferenţă relativ simple prezentate până acum constă în faptul că utilizatorul trebuie să enunţe toate informaţiile relevante de la început, înaintea declanşării raţionamentului, sub formă de fapte. Există, prin urmare, pericolul ca utilizatorul să enunţe prea puţine sau prea multe fapte. De aceea, este de preferat ca el să furnizeze informaţiile în mod interactiv, în cadrul unui dialog, atunci când acest lucru devine necesar. O asemenea abordare poate fi văzută în [3].

4.3.3.2. Înlănţuirea înainte

Înlănţuirea înainte nu începe cu o ipoteză, ci face un raţionament în direcţie opusă, de la partea cu **if** la partea cu **then**. În cazul exemplului studiat, de pildă, după ce se observă că holul este ud iar baia este uscată, se trage concluzia că există o problemă la bucătărie.

Interpretorul pentru înlănţuirea înainte pe care îl vom prezenta aici şi care este preluat din [3], presupune că regulile sunt, ca şi înainte, de forma

if Conditie **then** Concluzie

unde **Conditie** poate fi o expresie de tipul **AND/OR**. Pentru simplitate vom continua să presupunem că regulile nu conţin variabile. Interpretorul începe cu

ceea ce este deja cunoscut (şi enunţat prin intermediul relaţiei **fapta**), trage toate concluziile posibile şi adaugă aceste concluzii (folosind **assert**) relaţiei **fapta**:

```
inainte:-
      fapta_noua_dedusa(P),                  %o noua fapta
      !,
      write('Dedus:'),write(P),nl,
      assert(fapta (P)),
      inainte                                %continua
      ;
     write('Nu mai exista fapte').           %Toate faptele
                                             %au fost deduse

fapta_noua_dedusa(Concl):-
      if Cond then Concl,                    %o regula
      not fapta(Concl),                      %Concluzia
                                             %regulii nu este
                                             %inca o fapta
      fapta_compusa(Cond).                   %Conditia este
                                             %adevarata?

fapta_compusa(Cond):-                        %fapta simpla
      fapta(Cond).
fapta_compusa(Cond1 and Cond2):-
      fapta_compusa(Cond1),                  %Ambii conjuncti
      fapta_compusa(Cond2).                  %sunt adevarati

fapta_compusa(Cond1 or Cond2):-
      fapta_compusa(Cond1)
      ;
      fapta_compusa(Cond2).
```

Interogarea Prologului se face în felul următor:

```
?-inainte.
Dedus: problema_la_bucatarie
Dedus: fara_apa_din_afara
Dedus: scurgere_in_bucatarie
Nu mai exista fapte
```

4.3.3.3. Concluzii

Regulile **if-then** formează lanţuri de forma

informatie input →...→ informatie dedusa

Aceste două tipuri de informaţie sunt cunoscute sub diverse denumiri în literatura de specialitate, denumiri care depind, în mare măsură, de contextul în care ele sunt folosite. Informaţia de tip input mai poartă denumirea de *date* sau *manifestări*. Informaţia dedusă constituie *ipotezele* care trebuie demonstrate sau *cauzele* manifestărilor sau *diagnostice* sau *explicaţii*.

Atât înlănţuirea înainte, cât şi cea înapoi presupun *căutare*, dar direcţia de căutare este diferită pentru fiecare în parte. Înlănţuirea înapoi execută o căutare de la scopuri înspre date, din care cauză se spune despre ea că este *orientată către scop*. Prin contrast, înlănţuirea înainte caută pornind de la date înspre scopuri, fiind *orientată către date*.

Întrebarea firească care se ridică este cea referitoare la care tip de raţionament este preferabil. Răspunsul depinde în mare măsură de problema dată. În general, dacă se doreşte verificarea unei anumite ipoteze, atunci înlănţuirea înapoi, pornindu-se de la respectiva ipoteză, pare mai naturală. Dacă însă există o multitudine de ipoteze şi nu există o anumită motivaţie pentru testarea cu prioritate a uneia dintre ele, atunci înlănţuirea înainte va fi preferabilă. Această metodă se recomandă şi în cazul sistemelor în care datele se achiziţionează în mod continuu, iar sistemul trebuie să detecteze apariţia oricărei situaţii reprezentând o anomalie. În acest caz, fiecare schimbare în datele de intrare poate fi propagată înainte, pentru a se constata dacă ea indică vreo eroare a procesului monitorizat sau o schimbare a nivelului de performanţă.

În alegerea metodei de raţionament poate fi utilă însăşi forma reţelei în cauză. Astfel, un număr mic de noduri de date şi unul ridicat al nodurilor scop pot sugera ca fiind mai adecvată înlănţuirea înainte. Un număr redus al nodurilor scop şi multe noduri corespunzătoare datelor indică înlănţuirea înapoi ca fiind preferabilă. Este totuşi de notat faptul că majoritatea sistemelor expert sunt infinit mai complexe şi necesită o combinare a celor două tipuri de raţionament, adică o combinare a înlănţuirii în ambele direcţii.

4.3.4. Generarea explicaţiilor

Una dintre caracteristicile regulilor de producţie care fac din acestea o modalitate naturală de exprimare a cunoştinţelor în cadrul sistemelor expert este faptul că ele *susţin transparenţa sistemului*. Prin transparenţa sistemului se înţelege abilitatea acestuia de a explica deciziile şi soluţiile sale. Regulile de producţie facilitează generarea răspunsului pentru următoarele două tipuri de întrebări ale utilizatorului:

- Întrebare de tipul "cum": ***Cum*** ai ajuns la această concluzie?
- Întrebare de tipul "de ce": ***De ce*** te interesează această informaţie?

În cele ce urmează, ne vom ocupa de primul tip de întrebare. O tratare a întrebărilor de tipul "de ce", care necesită interacţiunea utilizatorului cu procesul de raţionament, poate fi consultată în [3].

În cazul întrebărilor de tipul "cum", explicaţia pe care sistemul o furnizează cu privire la modul în care a fost dedus răspunsul său constituie un *arbore de demonstraţie* a modului în care concluzia finală decurge din regulile şi faptele aflate în baza de cunoştinţe.

Fie "<=" un operator infixat. Atunci arborele de demonstraţie al unei propoziţii poate fi reprezentat în una dintre următoarele forme, în funcţie de necesităţi:

1. Dacă **P** este o faptă, atunci arborele de demonstraţie este **P**.
2. Dacă **P** a fost dedus folosind o regulă de forma

   if Cond **then P**

   atunci arborele de demonstraţie este

   P <= DemCond

unde **DemCond** este un arbore de demonstraţie a lui **Cond**.

3. Fie **P1** şi **P2** propoziţii ale căror arbori de demonstraţie sunt **Dem1** şi **Dem2**. Dacă **P** este de forma **P1 and P2**, atunci arborele de demonstraţie corespunzător este **Dem1 and Dem2**. Dacă **P** este de forma **P1 or P2**, atunci arborele de demonstraţie este fie **Dem1**, fie **Dem2**.

Construcţia arborilor de demonstraţie în Prolog este directă şi se poate realiza prin modificarea predicatului `este_adevarat`, introdus în §4.3.3.1, în conformitate cu cele trei cazuri enunţate mai sus. Un astfel de predicat `este_adevarat` modificat ar putea fi următorul, preluat de noi din [3]:

```
%este_adevarat(P,Dem) daca Dem constituie o
%demonstratie a faptului ca P este adevarat

:-op(800,xfx,<=).

este_adevarat(P,P):-
      fapta(P).
este_adevarat(P,P<= DemCond):-
      if Cond then P,
      este_adevarat(Cond,DemCond).
este_adevarat(P1 and P2, Dem1 and Dem2):-
      este_adevarat(P1,Dem1),
      este_adevarat(P2,Dem2).
este_adevarat(P1 or P2, Dem):-
      este_adevarat(P1,Dem)
      ;
      este_adevarat(P2,Dem).
```

4.3.5. Introducerea incertitudinii

Reprezentarea cunoştinţelor luată în discuţie până acum pleacă de la presupunerea că domeniile problemelor sunt *categorice*. Aceasta înseamnă că răspunsul la orice întrebare este fie adevărat, fie fals. Regulile care interveneau erau de aceeaşi natură, reprezentând aşa-numite implicaţii categorice. Totuşi, majoritatea domeniilor expert nu sunt categorice. Atât datele referitoare la o anumită problemă, cât şi regulile generale pot să nu fie certe. Incertitudinea poate fi modelată prin atribuirea unei calificări, alta decât adevărat sau fals, majorităţii aserţiunilor. Gradul de adevăr poate fi exprimat prin intermediul unui număr real aflat într-un anumit interval - spre exemplu, un număr între 0 şi 1 sau între -5 şi +5. Astfel de numere cunosc, în literatura de specialitate, o întreagă varietate de denumiri, cum ar fi *factor de certitudine*, *măsură a încrederii* sau *certitudine subiectivă*.

În cele ce urmează, vom exemplifica prin extinderea reprezentării bazate pe reguli de până acum cu o schemă simplă de incertitudine, preluată din [3]. Fiecărei propoziţii i se va adăuga un număr între 0 şi 1 ca *factor de certitudine*. Reprezentarea folosită va consta dintr-o pereche de forma:

Propoziţie: FactorCertitudine

Această notaţie va fi aplicată şi regulilor. Astfel, următoarea formă va defini o regulă şi gradul de certitudine până la care acea regulă este validă:

If Condiţie then Concluzie: Certitudine.

În cazul oricărei reprezentări cu incertitudine este necesară specificarea modului în care se combină certitudinile propoziţiilor şi ale regulilor. Spre exemplu, să presupunem că sunt date două propoziţii **P1 şi P2** având certitudinile **c(P1)** şi respectiv **c(P2)**. Atunci putem defini

c(P1 and P2) = min(c(P1), c(P2))

c(P1 or P2) = max(c(P1), c(P2))

Dacă există regula

if P1 then P2: C

cu C reprezentând factorul de certitudine, atunci

c(P2) = c(P1)*C

Pentru simplitate, vom presupune, în cele ce urmează, că nu există mai mult de o regulă susţinând o aceeaşi afirmaţie. Dacă ar exista două astfel de reguli în baza de cunoştinţe, ele ar putea fi transformate, cu ajutorul operatorului OR, în reguli echivalente care satisfac această presupunere. Implementarea în Prolog a unui interpretor de reguli corespunzător schemei de incertitudine descrise aici, preluată din [3], va presupune specificarea de către utilizator a estimaţiilor de certitudine corespunzătoare datelor observate (nodurile cel mai din stânga ale reţelei) prin relaţia

dat(Propozitie, Certitudine).

Iată un asemenea interpretor pentru reguli cu factor de certitudine:

```
% certitudine (Propozitie, Certitudine)

certutudine(P,Cert):-
      dat(P,Cert).

certitudine(Cond1 and Cond2, Cert):-
      certitudine(Cond1,Cert1),
      certitudine(Cond2,Cert2),
      minimum(Cert1,Cert2,Cert).

certitudine(Cond1 or Cond2, Cert):-
      certitudine(Cond1,Cert1),
      certitudine(Cond2,Cert2),
      maximum(Cert1,Cert2,Cert).

certitudine(P,Cert):-
      if Cond then P:C1,
      certitudine(Cond,C2),
      Cert is C1*C2.
```

Regulile bazei de cunoştinţe studiate anterior pot fi acum rafinate ca în următorul exemplu :

```
      if hol_ud and baie_uscat
then
      problema_la_bucatarie: 0.9.
```

O situaţie în care holul este ud, baia este uscată, bucătăria nu este uscată, fereastra nu este închisă şi credem - dar nu suntem siguri - că afară nu plouă, poate fi specificată după cum urmează:

```
      dat(hol_ud,1).
      dat(baie_uscat,1).
      dat(bucatarie_uscat,0).
      dat(fara_ploaie,0.8).
      dat(fereastra_inchisa,0).
```

Interogarea Prologului referitoare la o scurgere în bucătărie se face astfel:

```
?- certitudine(scurgere_in_bucatarie, C).
C = 0.8
```

Factorul de certitudine **C** este obţinut după cum urmează: faptul ca holul este ud iar baia este uscată indică o problemă în bucătărie cu certitudine 0.9. Întrucât a existat o oarecare posibilitate de ploaie, factorul de certitudine corespunzător lui *fara_apa_din_afara* este 0.8. În final, factorul de certitudine al lui *scurgere_in_bucatarie* este calculat ca fiind min(0.8, 0.9) = 0.8.

Chestiunea manevrării incertitudinii în sistemele expert a fost îndelung dezbătută în literatura de specialitate, cele prezentate aici, constituind, fără îndoială, o schemă mult simplificată de tratare a acestei probleme. Abordări matematice bazate pe teoria probabilităţilor există, în egală măsură. Ele vor fi amintite de noi în §6.1.1 al lucrării de faţă. Ceea ce li se reproşează, cel mai adesea, este faptul că abordări corecte din punct de vedere probabilistic necesită fie informaţie care nu este întotdeauna disponibilă, fie anumite presupuneri simplificatoare care, de regulă, nu sunt justificate în aplicaţiile practice şi care fac, din nou, ca abordarea să nu fie suficient de riguroasă din punct de vedere matematic.

Una dintre cele mai cunoscute şi mai utilizate scheme cu factori de certitudine este cea dezvoltată pentru sistemul MYCIN[18], un sistem expert folosit în diagnosticarea infecţiilor bacteriene. Factorii de certitudine MYCIN au fost concepuţi pentru a produce rezultate care păreau corecte experţilor, din punct de vedere intuitiv. Alţi cercetători au argumentat conceperea unor factori de certitudine bazaţi într-o mai mare măsură pe teoria probabilităţilor, iar alţii au experimentat scheme mult mai complexe, proiectate pentru a modela mai bine lumea reală. Factorii MYCIN continuă însă să fie utilizaţi cu succes în multe aplicaţii cu informaţie incertă.

4.3.6. Un exemplu de sistem expert. Implementare Prolog

Pentru exemplificarea noţiunilor descrise anterior, prezentăm, în continuare, implementarea în Prolog a unui sistem expert (SEPPCO) destinat asistării clienţilor unei firme de turism, în alegerea unei oferte pentru petrecerea concediului de odihnă, din totalitatea celor existente.

În cadrul acestui sistem, cunoştinţele sunt reprezentate cu ajutorul regulilor de tip **if-then** (dacă-atunci), iar inferenţa utilizată este de tip înlănţuire înapoi cu incertitudine (în prezenţa incertitudinii). Incertitudinea este modelată prin intermediul factorilor de certitudine/incertitudine de tip MYCIN (o descriere mai formală a factorilor MYCIN este realizată în §6.1.1). În exemplul din această secţiune, considerăm că factorii de certitudine sunt cuprinşi între **-100** şi **100**; -100 corespunde valorii cu siguranţă "fals", iar 100 corespunde valorii cu siguranţă "adevărat". Să mai remarcăm faptul că aceşti factori nu reprezintă probabilităţi.

[18] Pentru care vezi § 6.1.1 al lucrării de faţă.

Shell-ul sistemului SEPPCO este o versiune modificată a shell-ului unui sistem expert cu inferenţă de tip înlănţuire înapoi prezentat în [20] (capitolele 3 şi 4 şi Anexa B).

Prezentăm, în continuare, baza de cunoştinţe a sistemului SEPPCO. După cum se va vedea, conţinutul bazei apare într-o formă foarte apropiată de cea a limbajului natural.

Baza de cunoştinţe sistem SEPPCO

```
scopul este loc_concediu.

regula 1
daca buget_disponibil este redus
atunci in_romania fc 90.

regula 2
daca buget_disponibil este mediu
atunci in_romania fc 70.

regula 3
daca buget_disponibil este mare
atunci in_romania fc 50.

regula 4
daca departare este aproape
atunci in_romania.

regula 5
daca departare este departe
atunci in_romania fc 40.

regula 6
daca in_romania si
     la_mare si
     tip_oferta este sejur_1_luna si
     buget_disponibil este mare si
     anotimp este vara
atunci loc_concediu este neptun fc 80.

regula 7
daca in_romania si
     la_mare si
     tip_oferta este sejur_2_saptamani si
     buget_disponibil este mare si
     anotimp este vara
atunci loc_concediu este mamaia fc 90.
```

```
regula 8
daca in_romania si
     la_mare si
     tip_oferta este sejur_2_saptamani si
     anotimp este vara
atunci loc_concediu este costinesti fc 60.

regula 9
daca in_romania si
     not la_mare si
     tip_oferta este excursie si
     anotimp este vara
atunci loc_concediu este manastiri_oltenia fc 70.

regula 10
daca in_romania si
     not la_mare si
     tip_oferta este excursie si
     anotimp este vara
atunci loc_concediu este manastiri_moldova fc 60.

regula 11
daca not la_mare si
     anotimp este vara
atunci loc_concediu este delta_dunarii.

regula 12
daca not la_mare si
     tip_oferta este sejur_1_luna si
     anotimp este vara
atunci loc_concediu_vara este busteni.

regula 13
daca la_mare si
     departare este foarte_departe si
     buget_disponibil este mare si
     anotimp este vara
atunci loc_concediu_vara este bahamas fc 80.

regula 14
daca not la_mare si
     departare este foarte_departe si
     buget_disponibil este mare si
     tip_oferta este excursie si
     anotimp este vara
atunci loc_concediu este valea_loirei.
```



```
regula 15
daca departare este aproape si
     not la_mare si
     buget_disponibil este mediu si
     anotimp este vara
atunci loc_concediu_vara este sinaia fc 70.

regula 16
daca la_mare si
     buget_disponibil este mare si
     anotimp este iarna
atunci loc_concediu este rio_de_janeiro.

regula 17
daca buget_disponibil este mare si
     not la_mare si
     departare este foarte_departe si
     tip_oferta este excursie si
     anotimp este iarna
atunci loc_concediu este austria_germania_franta fc
90.

regula 18
daca departare este foarte_departe si
     not la_mare si
     tip_oferta este sejur_2_saptamani si
     buget_disponibil este mare si
     anotimp este iarna
atunci loc_concediu este chamonix fc 60.

regula 19
daca departare este aproape si
     not la_mare si
     tip_oferta este sejur_2_saptamani si
     buget_disponibil este mare si
     anotimp este iarna
atunci loc_concediu este poiana_brasov.

regula 20
daca in_romania si
     not la_mare si
     tip_oferta este sejur_2_saptamani si
     anotimp este iarna
atunci loc_concediu este busteni fc 70.

intreaba anotimp
optiuni (vara iarna)
```

```
afiseaza 'In ce anotimp preferati sa va petreceti
concediul?'.

intreaba tip_oferta
optiuni (sejur_2_saptamani sejur_1_luna excursie)
afiseaza 'Preferati sa mergeti intr-o excursie, ori
sa petreceti un sejur intr-o statiune?'.

intreaba la_mare
optiuni (da nu)
afiseaza 'Preferati sa petreceti concediul la mare?'.

intreaba departare
optiuni (aproape departe foarte_departe)
afiseaza 'Preferati ca locul de petrecere a
concediului sa fie mai aproape, ori mai departe de
localitatea unde locuiti?'.

intreaba buget_disponibil
optiuni (redus mediu mare)
afiseaza 'Ce tip de buget alocati pentru petrecerea
concediului?'.
```

Din punctul de vedere al implementării în Prolog se poate spune că, în general, există două modalități de a lucra cu o bază de cunoștințe exprimată printr-un limbaj aproape natural. Prima dintre ele se bazează pe definirea unor operatori și aceasta este abordarea folosită în implementarea sistemului prezentat în capitolul 15 din [3]. Această abordare a fost folosită și de noi în descrierea făcută în paragraful anterior. Cea de-a doua modalitate (exemplificată aici în implementarea sistemului SEPPCO) folosește, în esență, abilitatea Prolog-ului de a lucra cu gramatici de tip DCG ("definite clause grammar" - o extensie a gramaticilor independente de context). În această abordare, din punctul de vedere al programatorului, problema analizei sintactice (parsing) se reduce doar la specificarea unei gramatici DCG și, de aceea, am putea spune că această variantă oferă mai multă flexibilitate programatorului. O descriere amănunțită a acestei modalități de lucru există în [4].

Prezentăm, în continuare, câteva caracteristici ale sistemului SEPPCO, precum și principalele predicate ale implementării sale în Prolog.

Înainte de toate, vom face specificarea că, ori de câte ori o valoare a unui atribut este determinată, fie cu ajutorul unei reguli, fie cu ajutorul utilizatorului, o pereche de tipul (Atribut,Valoare) este salvată în spațiul de lucru, împreună cu factorul de certitudine *FC* asociat. Mai clar, informațiile referitoare la faptele cunoscute la un moment dat (informații care sunt, în esență, perechi de tipul (atribut, valoare)) sunt menținute în spațiul de lucru prin

intermediul unor termeni Prolog de tipul: `fapt(av(Atr, Val), FC, Reguli)`. Aici `Atr` și `Val` reprezintă un atribut și respectiv o valoare asociată acestuia, iar `FC` reprezintă factorul de certitudine asociat faptului nostru. Argumentul `Reguli` specifică lista regulilor care au dus în mod direct la derivarea faptului și acest argument este util pentru generarea explicațiilor la întrebările de tipul **cum**. Formatul intern al regulilor specificate în baza de cunoștințe este următorul:

`regula(N, premise(Lista), concluzie(Scop,FC))`

Primul argument, `N`, reprezintă un atom, care identifică în mod unic regula (mai exact, `N` este un întreg pozitiv). `Lista` este o listă de elemente de tipul `av(Atr, Val)`, iar `Scop` este și el un element de forma `av(Atr, Val)`.

Menționăm, de asemenea, că prin intermediul unor propoziții de tipul `Scopul este Atribut` putem specifica în baza de cunoștințe întrebările generale (scopurile principale) la care dorim ca sistemul nostru să răspundă. În formatul intern Prolog, o propoziție `Scopul este Atribut` se traduce/se reprezintă prin **scop(Atribut)**.

O caracteristică importantă a sistemului SEPPCO (în contextul prezenței incertitudinii) este aceea că inferența nu este oprită atunci când este găsit primul răspuns posibil la o întrebare; dimpotrivă, toate variantele posibile/"rezonabile" sunt explorate și raportate. În general, există două posibilități prin care incertitudinea poate pătrunde/apărea într-un sistem expert: prin intermediul regulilor cu concluzie incertă și prin intermediul răspunsurilor utilizatorului. În aceste condiții, situațiile în care trebuie specificată modalitatea prin care incertitudinea este propagată în sistem se referă, în esență, la următoarele cazuri:

- reguli cu concluzia incertă;
- reguli cu premise incerte;
- date incerte introduse de utilizator;
- combinarea premiselor incerte cu concluzii incerte;
- actualizarea unor fapte incerte în spațiul de lucru folosind, de asemenea, fapte incerte;
- impunerea unui prag de incertitudine pentru o premisă a unei reguli, prag care determină aplicarea/neaplicarea regulii respective.

Specificăm, în continuare, pe scurt, modalitățile în care toate aceste situații sunt tratate în SEPPCO. Menționăm, mai întâi, că lucrăm în ipoteza că membrul stâng al unei reguli este întotdeauna o conjuncție de "scopuri elementare". Remarcăm faptul că, în general, într-un sistem expert, membrul stâng al unei reguli poate avea o formă mai generală decât cea permisă în SEPPCO, spre exemplu poate fi de tipul conjuncții de disjuncții de scopuri elementare; pentru aceste situații au fost propuse diverse modalități de a se lucra cu factorii de (in)certitudine [6].

După cum se poate vedea în baza de cunoştinţe a sistemului SEPPCO, anumite reguli au un factor de (in)certitudine asociat lor; aceste reguli sunt numite reguli cu concluzia incertă. Semnificaţia factorului FC asociat unei reguli R este următoarea: dacă premisele regulii R sunt adevărate cu certitudinea 100, atunci se poate deduce că concluzia este cunoscută cu certitudinea FC. De exemplu, dacă concluzia unei reguli R are certitudine FC =50 şi dacă premisele sale sunt cunoscute cu certitudinea 100, atunci faptul din concluzia lui R este dedus cu certitudinea 50. În situaţia în care există premise în membrul stâng al unei reguli R, care au factorul de certitudine diferit de 100, factorul de certitudine al membrului stâng al regulii R este ales ca fiind minimul factorilor asociaţi premiselor lui R.

Dacă membrul stâng al unei reguli R este incert (şi factorul său asociat este $StFC$) şi dacă regula este „cu concluzie incertă" având factorul asociat RFC, atunci factorul de incertitudine FC al faptului determinat prin aplicarea regulii (adică al concluziei) este calculat astfel:

$$FC = \frac{RFC \cdot StFC}{100}$$

Pentru a evita folosirea regulilor al căror membru stâng au un factor de certitudine pozitiv, dar foarte mic, considerăm valoarea 20 ca fiind valoarea pozitivă minimă a factorului de certitudine asociat membrului stâng al unei reguli aplicabile R. Există cazuri în care mai multe reguli conduc la aceeaşi concluzie. În aceste situaţii este de dorit ca fiecare dintre ele să contribuie la factorul de incertitudine FC al faptului rezultat. De aceea folosim următoarea strategie: dacă o regula R ce conduce la o concluzie deja existentă în spaţiul de lucru (prin intermediul unui fapt F) se poate aplica la un moment dat, atunci noul factor de certitudine asociat faptului F se calculează astfel:

$$nou_FC(X,Y) = X + \frac{Y \cdot (100 - X)}{100}, \text{ dacă } X, Y > 0$$

$$nou_FC(X,Y) = 100 \cdot \frac{(X+Y)}{(100 - \min(X,Y))}, \text{ dacă } X > 0, Y \le 0$$

$$nou_FC(X,Y) = 100 \cdot \frac{(X+Y)}{(100 - \min(X,Y))}, \text{ dacă } X \le 0, Y > 0$$

$$nou_FC(X,Y) = -nou_FC(-X,-Y), \text{ dacă } X, Y < 0$$

În formulele anterioare, X reprezintă factorul de certitudine al faptului existent (F), Y este factorul de certitudine al faptului determinat prin aplicarea regulii

R, iar $nou_FC(X,Y)$ este noul factor de certitudine asociat faptului considerat (F).

Pe scurt, motorul de inferenţă al sistemului SEPPCO trebuie să îndeplinească următoarele funcţii: să combine factorii de certitudine în modul specificat anterior, să menţină în spaţiul de lucru faptele derivate şi să actualizeze informaţiile relevante. În plus, este nevoie ca toate informaţiile referitoare la un anume atribut să fie determinate (atunci când acest lucru este necesar) şi adăugate în spaţiul de lucru.

Trebuie remarcat faptul că, în implementarea de faţă, presupunem că, iniţial, în spaţiul de lucru nu avem nici un fapt cunoscut, ci numai reguli şi alte informaţii referitoare la întrebările pe care utilizatorul le poate pune (toate fiind exact informaţiile care se regăsesc în baza de cunoştinţe).

Principalul predicat al programului, predicat care reprezintă de fapt „motorul" inferenţei, este `realizare_scop`:

```
realizare_scop(Scop, FC, Istorie)
```

Argumentele sale au următoarele semnificaţii:

- `Scop` are fie forma `av(Atr,Val)`, fie forma `not av(Atr,Val)`; `Av` este atributul despre care dorim să aflăm informaţii la un moment dat şi, de obicei, atunci când acest predicat este folosit, `Atr` este deja instanţiat.
- `FC` reprezintă factorul de incertitudine ce va fi asociat faptului nostru în final.
- `Istorie` reţine "istoria" şi este folosit pentru a răspunde la întrebarea "de ce?"

Predicatul tratează, în esenţă, trei cazuri, după cum urmează:

- dacă în baza de cunoştinţe există un fapt care ne oferă informaţii referitoare la `Atr`, atunci problema este rezolvată şi alte posibilităţi (reguli, informaţii de la utilizator) sunt eliminate; deoarece în baza de cunoştinţe apare o informaţie referitoare la `Atr`, înseamnă că toate variantele de răspuns pentru `Atr` au fost explorate deja (acest lucru se întamplă deoarece, aşa cum s-a specificat, lucrăm în ipoteza că, iniţial, baza de cunoştinţe nu conţine nici un fapt cunoscut);
- altfel, dacă utilizatorul poate fi interogat referitor la `Atr`, atunci acesta este întrebat despre `Atr` şi răspunsul său este înregistrat (adăugat în spaţiul de lucru); predicatul folosit pentru interogare este `interogheaza`;
- altfel, se determină pe rând toate regulile care au atributul `Atr` în concluzie şi se încearcă demonstrarea premiselor din regulile respective; dacă acestea pot fi demonstrate, atunci "incertitudinea

premiselor" este combinată cu (in)certitudinea concluziei pentru a se determina factorul de certitudine al rezultatului datorat aplicării fiecărei reguli în parte.

Remarcăm faptul că, în cel de-al treilea caz, se iau în calcul toate regulile în care `Atr` intervine în concluzie, chiar dacă acestea se referă la valori diferite pentru `Atr`. Specificăm, de asemenea, faptul că sistemul nostru permite ca premisa unei reguli să fie şi de forma `not Scop`; în această situaţie, predicatul `realizare_scop` este utilizat pentru `Scop`, iar factorul de certitudine pentru `not Scop` este definit/considerat ca fiind opusul factorului de certitudine pentru `Scop`.

Menţionăm că, prin intermediul unor termeni de tipul `interogabil(Atr,Mesaj,Optiuni)`, sistemul SEPPCO menţine în spaţiul de lucru informaţii referitoare la atributele despre care putem obţine informaţii de la utilizator. Aceste informaţii sunt preluate din baza de cunoştinţe în momentul în care ea este încărcată.

Predicatul `interogheaza` este utilizat în cel de-al doilea caz prezentat în descrierea predicatului `realizare_scop`, şi anume atunci când nu există în spaţiul de lucru fapte referitoare la un anume atribut, dar se pot obţine informaţii referitoare la `Atr` de la utilizator. Forma sa este:

```
interogheaza(Atr,Mesaj,Optiuni,Istorie)
```

Argumentele sale au următoarele semnificaţii:

- `Atr` este atributul despre care dorim să obţinem informaţii;
- `Mesaj` reprezintă mesajul/întrebarea care va fi adresată utilizatorului atunci când acesta va fi interogat; acest mesaj poate fi obţinut prin intermediul predicatului `interogabil`;
- `Optiuni` este de fapt lista în care sunt specificate valorile posibile ale atributului `Atr`; ca şi în cazul atributului `Mesaj`, `Optiuni` poate fi determinat prin intermediul predicatului `interogabil`;
- Argumentul `Istorie` reprezintă istoria şi este folosit pentru a răspunde la întrebările de tipul "**de ce?**"

Specificăm faptul că răspunsul dorit este fie de forma `M` (unde `M` este un membru din lista de opţiuni `Optiuni`), fie de forma `M fc Numar` (unde `Numar` este un factor de certitudine diferit de 100). Mai remarcăm doar că în prima situaţie se consideră că factorul de certitudine al răspunsului asociat este maxim.

Cel de-al treilea caz prezentat în descrierea predicatului `realizare_scop` se referă la situaţia în care sunt efectiv utilizate reguli pentru a determina informaţii despre `Atr`. Predicatul folosit în această situaţie este predicatul `fg`, predicat care forţează folosirea tuturor regulilor care au `Atr` în

concluzie, precum şi combinarea factorilor de certitudine ale faptelor derivate, în modul dorit şi descris anterior. După cum se va vedea în textul programului, predicatele principale folosite în definiţia predicatului `fg` sunt: `demonstreaza`, `ajusteaza`, `actualizeaza`. Predicatul `demonstreaza` este utilizat pentru a considera toate premisele din lista de premise a fiecărei reguli de interes, apelând `realizare_scop` pentru fiecare dintre premise; factorul de certitudine final asociat membrului stâng al fiecărei reguli este calculat ca fiind minimul factorilor de certitudine din lista de premise asociate regulii respective. Predicatul `ajusteaza` este folosit pentru a combina incertitudinea asociată unei reguli cu incertitudinea premiselor sale. În final, predicatul `actualizeaza` tratează situaţia în care concluzia datorată aplicării unei reguli se regăseşte deja în spaţiul de lucru prin intermediul unui fapt; acest predicat este folosit în cazul în care concluziile mai multor reguli se referă la aceeaşi pereche (atribut, valoare).

Principalul predicat al shell-ului este predicatul `pornire`, predicat care determină un ciclu de execuţie, în acesta din urmă opţiunile fiind `incarca`, `consulta`, `reinitiaza`, `afisare_fapte`, `iesire`, `cum`. Opţiunile `incarca`, `afisare_fapte`, `consulta`, `cum` sunt implementate prin intermediul predicatelor `incarca`, `scopuri_princ`, `cum`, `afiseaza_fapte`. Predicatul `scopuri_princ` încearcă satisfacerea, pe rând, a fiecărui scop care a fost specificat în baza de cunoştinţe; el reprezintă punctul de pornire al inferenţei (de tip înlănţuire înapoi). Predicatul `incarca` este utilizat pentru încărcarea bazei de cunoştinţe în spaţiul de lucru; cel mai important predicat care intervine în descrierea predicatului `incarca` este predicatul `incarca_reguli`, care este definit în felul următor:

```
incarca_reguli :-
        repeat, citeste_propozitie(L),
        proceseaza(L),L == [end_of_file],nl.
```

Aşa cum arată definiţia sa, predicatul `incarca_reguli` forţează citirea, pe rând, a fiecărei propoziţii din fişierul de intrare (fişier determinat în interiorul predicatului `incarca`). Fiecărei propoziţii ii este asociată o listă de atomi (prin intermediul predicatului `citeste_propozitie`) şi fiecare astfel de listă de atomi este "transformată" într-un termen Prolog, prin intermediul predicatului `proceseaza`. Definirea predicatului `citeste_propozitie` se bazează pe o implementare sugerată în [4]. Predicatul `proceseaza` este definit astfel:

```
proceseaza([end_of_file]):-!.
proceseaza(L) :-trad(R,L,[]),assertz(R), !.
```

Argumentul său, `L`, reprezintă o listă de atomi, despre care considerăm că este validă dacă ea corespunde cuvintelor unei fraze din fişierul în care este specificată baza de cunoştinţe a sistemului SEPPCO. În esenţă, predicatul `trad` este folosit pentru traducerea unei fraze valide din baza de cunoştinţe (frază specificată prin lista de atomi corespunzătoare) în formatul intern al Prologului. Aşa cum am specificat deja, această traducere este implementată folosind gramatici de tip DCG. Dacă lista de atomi `L` nu reprezintă o frază validă, atunci predicatul `trad` şi, prin urmare, şi predicatul `proceseaza`, eşuează. Remarcăm faptul că, în definirea predicatului `trad`, se specifică în mod clar ce înseamnă fraza validă.

Reamintim că informaţiile referitoare la faptele cunoscute la un moment dat sunt menţinute în spaţiul de lucru prin intermediul unor termeni Prolog de tipul `fapt(av(Atr,Val), FC, Reguli)`, unde argumentul `Reguli` specifică lista regulilor care au dus în mod direct la derivarea faptului nostru. Folosind această caracteristică, implementarea predicatului **cum** (predicat folosit pentru generarea explicaţiilor la întrebările de tipul „cum?") este foarte naturală (a se vedea textul programului).

Predicatul `reinitiaza` este utilizat pentru a readuce spaţiul de lucru la o fomă identică cu cea imediat de după încărcarea bazei de cunoştinţe (o formă în care, în spaţiul de lucru, se regăsesc numai reguli şi informaţii referitoare la întrebările pe care utilizatorul le poate pune, dar nu şi fapte cunoscute (deduse/derivate)).

Prezentăm, în continuare, o implementare în Prolog a sistemului nostru, implementare ce foloseşte predicatele şi considerentele anterioare:

```
:-use_module(library(lists)).
:-use_module(library(system)).
:-op(900,fy,not).
:-dynamic fapt/3.
:-dynamic interogat/1.
:-dynamic scop/1.
:-dynamic interogabil/3.
:-dynamic regula/3.

not(P):-P,!,fail.
not(_).

scrie_lista([]):-nl.
scrie_lista([H|T]) :-
        write(H), tab(1),
        scrie_lista(T).

afiseaza_fapte :-
        write('Fapte existente în baza de cunostinte:'),
```



```
        nl,nl, write(' (Atribut,valoare) '), nl,nl,
        listeaza_fapte,nl.

listeaza_fapte:-
        fapt(av(Atr,Val),FC,_),
        write('('),write(Atr),write(','),
        write(Val), write(')'),
        write(','), write(' certitudine '),
        FC1 is integer(FC),write(FC1),
        nl,fail.
listeaza_fapte.

lista_float_int([],[]).
lista_float_int([Regula|Reguli],[Regula1|Reguli1]):-
        (Regula \== utiliz,
        Regula1 is integer(Regula);
        Regula == utiliz, Regula1=Regula),
        lista_float_int(Reguli,Reguli1).

pornire :-
   retractall(interogat(_)),
   retractall(fapt(_,_,_)),
   repeat,
   write('Introduceti una din urmatoarele optiuni: '),
   nl,nl,
   write(' (Incarca Consulta Reinitiaza  Afisare_fapte  Cum
   Iesire) '),
   nl,nl,write('|: '),citeste_linie([H|T]),
   executa([H|T]), H == iesire.

executa([incarca]) :-
        incarca,!,nl,
        write('Fisierul dorit a fost incarcat'),nl.
executa([consulta]) :-
        scopuri_princ,!.
executa([reinitiaza]) :-
        retractall(interogat(_)),
        retractall(fapt(_,_,_)),!.
executa([afisare_fapte]) :-
        afiseaza_fapte,!.
executa([cum|L]) :- cum(L),!.
executa([iesire]):-!.
executa([_|_]) :-
        write('Comanda incorecta! '),nl.

scopuri_princ :-
   scop(Atr),determina(Atr), afiseaza_scop(Atr),fail.
```

```
scopuri_princ.

determina(Atr) :-
        realizare_scop(av(Atr,_),_,[scop(Atr)]),!.
determina(_).

afiseaza_scop(Atr) :-
        nl,fapt(av(Atr,Val),FC,_),
        FC >= 20,scrie_scop(av(Atr,Val),FC),
        nl,fail.
afiseaza_scop(_):-nl,nl.

scrie_scop(av(Atr,Val),FC) :-
        transformare(av(Atr,Val), X),
        scrie_lista(X),tab(2),
        write(' '),
        write('factorul de certitudine este '),
        FC1 is integer(FC),write(FC1).

realizare_scop(not Scop,Not_FC,Istorie) :-
        realizare_scop(Scop,FC,Istorie),
        Not_FC is - FC, !.
realizare_scop(Scop,FC,_) :-
        fapt(Scop,FC,_), !.
realizare_scop(Scop,FC,Istorie) :-
        pot_interoga(Scop,Istorie),
        !,realizare_scop(Scop,FC,Istorie).
realizare_scop(Scop,FC_curent,Istorie) :-
        fg(Scop,FC_curent,Istorie).

fg(Scop,FC_curent,Istorie) :-
        regula(N, premise(Lista), concluzie(Scop,FC)),
        demonstreaza(N,Lista,FC_premise,Istorie),
        ajusteaza(FC,FC_premise,FC_nou),
        actualizeaza(Scop,FC_nou,FC_curent,N),
        FC_curent == 100,!.
fg(Scop,FC,_) :- fapt(Scop,FC,_).

pot_interoga(av(Atr,_),Istorie) :-
        not interogat(av(Atr,_)),
        interogabil(Atr,Optiuni,Mesaj),
        interogheaza(Atr,Mesaj,Optiuni,Istorie),nl,
        asserta( interogat(av(Atr,_)) ).

cum([]) :-
        write('Scop? '),nl,
        write('|:'),citeste_linie(Linie),nl,
```



```
        transformare(Scop,Linie), cum(Scop).
cum(L) :-
        transformare(Scop,L),nl, cum(Scop).
cum(not Scop) :-
        fapt(Scop,FC,Reguli),
        lista_float_int(Reguli,Reguli1),
        FC < -20,transformare(not Scop,PG),
        append(PG,[a,fost,derivat,cu,ajutorul, 'regulilor:
        '|Reguli1],LL),
        scrie_lista(LL),nl,afis_reguli(Reguli),fail.
cum(Scop) :-
        fapt(Scop,FC,Reguli),
        lista_float_int(Reguli,Reguli1),
        FC > 20,transformare(Scop,PG),
        append(PG,[a,fost,derivat,cu,ajutorul,'regulilor:
        '|Reguli1],LL),
        scrie_lista(LL),nl,afis_reguli(Reguli),
        fail.
cum(_).

afis_reguli([]).
afis_reguli([N|X]) :-
        afis_regula(N),
        premisele(N),
        afis_reguli(X).

afis_regula(N) :-
        regula(N, premise(Lista_premise),
        concluzie(Scop,FC)),NN is integer(N),
        scrie_lista(['regula  ',NN]),
        scrie_lista(['  Daca']),
        scrie_lista_premise(Lista_premise),
        scrie_lista(['  Atunci']),
        transformare(Scop,Scop_tr),
        append(['   '],Scop_tr,L1),
        FC1 is integer(FC),append(L1,[FC1],LL),
        scrie_lista(LL),nl.

scrie_lista_premise([]).
scrie_lista_premise([H|T]) :-
        transformare(H,H_tr),
        tab(5),scrie_lista(H_tr),
        scrie_lista_premise(T).

transformare(av(A,da),[A]) :- !.
transformare(not av(A,da), [not,A]) :- !.
transformare(av(A,nu),[not,A]) :- !.
```

```
transformare(av(A,V),[A,este,V]).

premisele(N) :-
        regula(N, premise(Lista_premise), _),
        !, cum_premise(Lista_premise).

cum_premise([]).
cum_premise([Scop|X]) :-
        cum(Scop),
        cum_premise(X).

interogheaza(Atr,Mesaj,[da,nu],Istorie) :-
        !,write(Mesaj),nl,
        de_la_utiliz(X,Istorie,[da,nu]),
        det_val_fc(X,Val,FC),
        asserta( fapt(av(Atr,Val),FC,[utiliz]) ).
interogheaza(Atr,Mesaj,Optiuni,Istorie) :-
        write(Mesaj),nl,
        citeste_opt(VLista,Optiuni,Istorie),
        assert_fapt(Atr,VLista).

citeste_opt(X,Optiuni,Istorie) :-
        append(['('],Optiuni,Opt1),
        append(Opt1,[')'],Opt),
        scrie_lista(Opt),
        de_la_utiliz(X,Istorie,Optiuni).

de_la_utiliz(X,Istorie,Lista_opt) :-
        repeat,write(': '),citeste_linie(X),
        proceseaza_raspuns(X,Istorie,Lista_opt).

proceseaza_raspuns([de_ce],Istorie,_) :-
        nl,afis_istorie(Istorie),!,fail.
proceseaza_raspuns([X],_,Lista_opt):-
        member(X,Lista_opt).
proceseaza_raspuns([X,fc,FC],_,Lista_opt):-
        member(X,Lista_opt),float(FC).

assert_fapt(Atr,[Val,fc,FC]) :-
        !,asserta( fapt(av(Atr,Val),FC,[utiliz]) ).
assert_fapt(Atr,[Val]) :-
        asserta( fapt(av(Atr,Val),100,[utiliz])).

det_val_fc([nu],da,-100).
det_val_fc([nu,FC],da,NFC) :- NFC is -FC.
det_val_fc([nu,fc,FC],da,NFC) :- NFC is -FC.
det_val_fc([Val,FC],Val,FC).
```



```
det_val_fc([Val,fc,FC],Val,FC).
det_val_fc([Val],Val,100).

afis_istorie([]) :- nl.
afis_istorie([scop(X)|T]) :-
        scrie_lista([scop,X]),!,
        afis_istorie(T).
afis_istorie([N|T]) :-
        afis_regula(N),!,afis_istorie(T).

demonstreaza(N,ListaPremise,Val_finala,Istorie) :-
   dem(ListaPremise,100,Val_finala,[N|Istorie]),!.

dem([],Val_finala,Val_finala,_).
dem([H|T],Val_actuala,Val_finala,Istorie) :-
        realizare_scop(H,FC,Istorie),
        Val_interm is min(Val_actuala,FC),
        Val_interm >= 20,
        dem(T,Val_interm,Val_finala,Istorie).

actualizeaza(Scop,FC_nou,FC,RegulaN) :-
   fapt(Scop,FC_vechi,_),
   combina(FC_nou,FC_vechi,FC),
   retract( fapt(Scop,FC_vechi,Reguli_vechi) ),
   asserta( fapt(Scop,FC,[RegulaN | Reguli_vechi]) ),!.
actualizeaza(Scop,FC,FC,RegulaN) :-
   asserta( fapt(Scop,FC,[RegulaN]) ).

ajusteaza(FC1,FC2,FC) :-
        X is FC1 * FC2 / 100,
        FC is round(X).

combina(FC1,FC2,FC) :-
        FC1 >= 0,FC2 >= 0,
        X is FC2*(100 - FC1)/100 + FC1,
        FC is round(X).
combina(FC1,FC2,FC) :-
        FC1 < 0,FC2 < 0,
        X is - ( -FC1 -FC2 * (100 + FC1)/100),
        FC is round(X).
combina(FC1,FC2,FC) :-
        (FC1 < 0; FC2 < 0),
        (FC1 > 0; FC2 > 0),
        FCM1 is abs(FC1),FCM2 is abs(FC2),
        MFC is min(FCM1,FCM2),
        X is 100 * (FC1 + FC2) / (100 - MFC),
        FC is round(X).
```

```
incarca :-
        write('Introduceti numele fisierului care doriti
        sa fie incarcat: '),nl, write('|:'),read(F),
        file_exists(F),!,incarca(F).
incarca :-write('Nume incorect de fisier! '),nl,fail.

incarca(F) :-
   retractall(interogat(_)),retractall(fapt(_,_,_)),
   retractall(scop(_)),retractall(interogabil(_,_,_)),
   retractall(regula(_,_,_)),
   see(F),incarca_reguli,seen,!.

incarca_reguli :-
        repeat,citeste_propozitie(L),
        proceseaza(L),L == [end_of_file],nl.

proceseaza([end_of_file]):-!.
proceseaza(L) :-
        trad(R,L,[]),assertz(R), !.

trad(scop(X)) --> [scopul,este,X].
trad(scop(X)) --> [scopul,X].
trad(interogabil(Atr,M,P)) -->
        [intreaba,Atr],lista_optiuni(M),afiseaza(Atr,P).
trad(regula(N,premise(Daca),concluzie(Atunci,F))) -->
        identificator(N),daca(Daca),atunci(Atunci,F).
trad('Eroare la parsare'-L,L,_).

lista_optiuni(M) --> [optiuni,'('],lista_de_optiuni(M).

lista_de_optiuni([Element]) --> [Element,')'].
lista_de_optiuni([Element|T]) -->
        [Element],lista_de_optiuni(T).

afiseaza(_,P) --> [afiseaza,P].
afiseaza(P,P) --> [].

identificator(N) --> [regula,N].
daca(Daca) --> [daca],lista_premise(Daca).

lista_premise([Daca]) --> propoz(Daca),[atunci].
lista_premise([Prima|Celalalte]) -->
        propoz(Prima),[si],lista_premise(Celalalte).
lista_premise([Prima|Celalalte]) -->
        propoz(Prima),[','],lista_premise(Celalalte).

atunci(Atunci,FC) --> propoz(Atunci),[fc],[FC].
```



```
atunci(Atunci,100) --> propoz(Atunci).

propoz(not av(Atr,da)) --> [not,Atr].
propoz(av(Atr,Val)) --> [Atr,este,Val].
propoz(av(Atr,da)) --> [Atr].

citeste_linie([Cuv|Lista_cuv]) :-
        get0(Car),
        citeste_cuvant(Car, Cuv, Car1),
        rest_cuvinte_linie(Car1, Lista_cuv).

% -1 este codul ASCII pt EOF

rest_cuvinte_linie(-1, []):-!.
rest_cuvinte_linie(Car,[]) :-(Car==13;Car==10), !.
rest_cuvinte_linie(Car,[Cuv1|Lista_cuv]) :-
        citeste_cuvant(Car,Cuv1,Car1),
        rest_cuvinte_linie(Car1,Lista_cuv).

citeste_propozitie([Cuv|Lista_cuv]) :-
        get0(Car),citeste_cuvant(Car, Cuv, Car1),
        rest_cuvinte_propozitie(Car1, Lista_cuv).

rest_cuvinte_propozitie(-1, []):-!.
rest_cuvinte_propozitie(Car,[]) :-Car==46, !.
rest_cuvinte_propozitie(Car,[Cuv1|Lista_cuv]) :-
        citeste_cuvant(Car,Cuv1,Car1),
        rest_cuvinte_propozitie(Car1,Lista_cuv).

citeste_cuvant(-1,end_of_file,-1):-!.
citeste_cuvant(Caracter,Cuvant,Caracter1) :-
        caracter_cuvant(Caracter),!,
        name(Cuvant, [Caracter]),get0(Caracter1).
citeste_cuvant(Caracter, Numar, Caracter1) :-
        caracter_numar(Caracter),!,
        citeste_tot_numarul(Caracter, Numar, Caracter1).

citeste_tot_numarul(Caracter,Numar,Caracter1):-
        determina_lista(Lista1,Caracter1),
        append([Caracter],Lista1,Lista),
        transforma_lista_numar(Lista,Numar).

determina_lista(Lista,Caracter1):-
        get0(Caracter),
        (caracter_numar(Caracter),
        determina_lista(Lista1,Caracter1),
        append([Caracter],Lista1,Lista);
```

```
        \+(caracter_numar(Caracter)),
        Lista=[],Caracter1=Caracter).

transforma_lista_numar([],0).
transforma_lista_numar([H|T],N):-
        transforma_lista_numar(T,NN),
        lungime(T,L), Aux is exp(10,L),
        HH is H-48,N is HH*Aux+NN.

lungime([],0).
lungime([_|T],L):-
        lungime(T,L1),
        L is L1+1.

% 39 este codul ASCII pt '

citeste_cuvant(Caracter,Cuvant,Caracter1) :-
        Caracter==39,!,
        pana_la_urmatorul_apostrof(Lista_caractere),
        L=[Caracter|Lista_caractere],
        name(Cuvant, L),get0(Caracter1).

pana_la_urmatorul_apostrof(Lista_caractere):-
        get0(Caracter),
        (Caracter == 39,Lista_caractere=[Caracter];
        Caracter\==39,
        pana_la_urmatorul_apostrof(Lista_caractere1),
        Lista_caractere=[Caracter|Lista_caractere1]).

citeste_cuvant(Caracter,Cuvant,Caracter1) :-
        caractere_in_interiorul_unui_cuvant(Caracter),!,
        ((Caracter>64,Caracter<91),!,
        Caracter_modificat is Caracter+32;
        Caracter_modificat is Caracter),
        citeste_intreg_cuvantul(Caractere,Caracter1),
        name(Cuvant,[Caracter_modificat|Caractere]).

citeste_intreg_cuvantul(Lista_Caractere,Caracter1) :-
   get0(Caracter),
   (caractere_in_interiorul_unui_cuvant(Caracter),
   ((Caracter>64,Caracter<91),!,
   Caracter_modificat is Caracter+32;
   Caracter_modificat is Caracter),
   citeste_intreg_cuvantul(Lista_Caractere1, Caracter1),
   Lista_Caractere=[Caracter_modificat|Lista_Caractere1];
   \+(caractere_in_interiorul_unui_cuvant(Caracter)),
   Lista_Caractere=[], Caracter1=Caracter).
```



```
citeste_cuvant(_,Cuvant,Caracter1):-
   get0(Caracter),
   citeste_cuvant(Caracter,Cuvant,Caracter1).

caracter_cuvant(C):-member(C,[44,59,58,63,33,46,41,40]).

% am specificat codurile ASCII pentru , ; : ? ! . ) (

caractere_in_interiorul_unui_cuvant(C):-
   C>64,C<91;C>47,C<58;
   C==45;C==95;C>96,C<123.

caracter_numar(C):-C<58,C>=48.
```

Prezentăm, în continuare, un exemplu complex de execuţie a programului. În acest exemplu, sunt evidenţiate cele mai importante caracteristici ale sistemului SEPCCO, prin intermediul unor situaţii special alese pentru a releva modul în care sistemul propus tratează incertitudinea. Un exerciţiu util (chiar recomandabil) ar putea fi rularea exemplului împreună cu urmărirea îndeaproape a mecanismului de producere a răspunsurilor, în fiecare caz în parte.

Exemplu de utilizare

| ?- pornire.
Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: incarca
Introduceti numele fisierului care doriti sa fie incarcat:
|:'C:/ sist_seppco.txt'.

Fisierul dorit a fost incarcat

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: consulta
'Ce tip de buget alocati pentru petrecerea concediului?'
(redus mediu mare)
: mare

'Preferati ca locul de petrecere a concediului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'
(aproape departe foarte_departe)
: departe

'Preferati sa petreceti concediul la mare?'
: da

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'
(sejur_2_saptamani sejur_1_luna excursie)
: sejur_2_saptamani

'In ce anotimp preferati sa va petreceti concediul?'
(vara iarna)
: vara

loc_concediu este costinesti
factorul de certitudine este 42
loc_concediu este mamaia
factorul de certitudine este 63

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: afisare_fapte
Fapte existente în baza de cunostinte:

(Atribut,valoare)

(loc_concediu,costinesti), certitudine 42
(loc_concediu,mamaia), certitudine 63
(anotimp,vara), certitudine 100
(tip_oferta,sejur_2_saptamani), certitudine 100
(la_mare,da), certitudine 100
(in_romania,da), certitudine 70
(departare,departe), certitudine 100
(buget_disponibil,mare), certitudine 100

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum
Scop?
|:loc_concediu este costinesti

loc_concediu este costinesti a fost derivat cu ajutorul regulilor: 8

regula 8
Daca
in_romania
la_mare
tip_oferta este sejur_2_saptamani
anotimp este vara
Atunci
loc_concediu este costinesti 60

in_romania a fost derivat cu ajutorul regulilor: 5 3

regula 5
Daca
departare este departe
Atunci
in_romania 40

departare este departe a fost derivat cu ajutorul regulilor: utiliz

regula 3
Daca
buget_disponibil este mare
Atunci
in_romania 50

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

la_mare a fost derivat cu ajutorul regulilor: utiliz

tip_oferta este sejur_2_saptamani a fost derivat cu ajutorul regulilor: utiliz

anotimp este vara a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: reinitiaza
Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: consulta
'Ce tip de buget alocati pentru petrecerea concediului?'

(redus mediu mare)
: mare

'Preferati ca locul de petrecere a concediului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'
(aproape departe foarte_departe)
: aproape

'Preferati sa petreceti concediul la mare?'
: da

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'
(sejur_2_saptamani sejur_1_luna excursie)
: sejur_1_luna

'In ce anotimp preferati sa va petreceti concediul?'
(vara iarna)
: vara

loc_concediu este neptun
factorul de certitudine este 80

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum
Scop?
|:loc_concediu este neptun

loc_concediu este neptun a fost derivat cu ajutorul regulilor: 6

regula 6
Daca
in_romania
la_mare
tip_oferta este sejur_1_luna
buget_disponibil este mare
anotimp este vara
Atunci
loc_concediu este neptun 80

in_romania a fost derivat cu ajutorul regulilor: 4 3

regula 4
Daca

departare este aproape
Atunci
in_romania 100

departare este aproape a fost derivat cu ajutorul regulilor: utiliz

regula 3
Daca
buget_disponibil este mare
Atunci
in_romania 50

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

la_mare a fost derivat cu ajutorul regulilor: utiliz

tip_oferta este sejur_1_luna a fost derivat cu ajutorul regulilor: utiliz

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

anotimp este vara a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: reinitiaza
Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: consulta
'Ce tip de buget alocati pentru petrecerea concediului?'
(redus mediu mare)
: mare

'Preferati ca locul de petrecere a concediului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'
(aproape departe foarte_departe)
: aproape fc 90

'Preferati sa petreceti concediul la mare?'
: da

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'
(sejur_2_saptamani sejur_1_luna excursie)

: sejur_1_luna fc 60

'In ce anotimp preferati sa va petreceti concediul?'
(vara iarna)
: vara fc 70

loc_concediu este neptun
factorul de certitudine este 48

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum loc_concediu este neptun

loc_concediu este neptun a fost derivat cu ajutorul regulilor: 6

regula 6
Daca
in_romania
la_mare
tip_oferta este sejur_1_luna
buget_disponibil este mare
anotimp este vara
Atunci
loc_concediu este neptun 80

in_romania a fost derivat cu ajutorul regulilor: 4 3

regula 4
Daca
departare este aproape
Atunci
in_romania 100

departare este aproape a fost derivat cu ajutorul regulilor: utiliz

regula 3
Daca
buget_disponibil este mare
Atunci
in_romania 50

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

la_mare a fost derivat cu ajutorul regulilor: utiliz

tip_oferta este sejur_1_luna a fost derivat cu ajutorul regulilor: utiliz

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

anotimp este vara a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: afisare_fapte
Fapte existente în baza de cunostinte:

(Atribut,valoare)

(loc_concediu,neptun), certitudine 48
(anotimp,vara), certitudine 70
(tip_oferta,sejur_1_luna), certitudine 60
(la_mare,da), certitudine 100
(in_romania,da), certitudine 95
(departare,aproape), certitudine 90
(buget_disponibil,mare), certitudine 100

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: reinitiaza
Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: consulta
'Ce tip de buget alocati pentru petrecerea concediului?'
(redus mediu mare)
: mare

'Preferati ca locul de petrecere a concediului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'
(aproape departe foarte_departe)
: departe

'Preferati sa petreceti concediul la mare?'
: nu fc 30

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'
(sejur_2_saptamani sejur_1_luna excursie)
: excursie

'In ce anotimp preferati sa va petreceti concediul?'
(vara iarna)
: vara

loc_concediu este delta_dunarii
  factorul de certitudine este 30
loc_concediu este manastiri_oltenia
  factorul de certitudine este 21

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum loc_concediu este delta_dunarii

loc_concediu este delta_dunarii a fost derivat cu ajutorul regulilor: 11

regula 11
  Daca
    not la_mare
    anotimp este vara
  Atunci
    loc_concediu este delta_dunarii 100

not la_mare a fost derivat cu ajutorul regulilor: utiliz

anotimp este vara a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum loc_concediu este manastiri_oltenia

loc_concediu este manastiri_oltenia a fost derivat cu ajutorul regulilor: 9

regula 9
  Daca
    in_romania
    not la_mare
    tip_oferta este excursie
    anotimp este vara

  Atunci
    loc_concediu este manastiri_oltenia 70

in_romania a fost derivat cu ajutorul regulilor: 5 3

regula 5
  Daca
    departare este departe
  Atunci
    in_romania 40

departare este departe a fost derivat cu ajutorul regulilor: utiliz

regula 3
  Daca
    buget_disponibil este mare
  Atunci
    in_romania 50

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

not la_mare a fost derivat cu ajutorul regulilor: utiliz

tip_oferta este excursie a fost derivat cu ajutorul regulilor: utiliz

anotimp este vara a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: afisare_fapte
Fapte existente în baza de cunostinte:

(Atribut,valoare)

(loc_concediu,delta_dunarii), certitudine 30
(loc_concediu,manastiri_moldova), certitudine 18
(loc_concediu,manastiri_oltenia), certitudine 21
(anotimp,vara), certitudine 100
(tip_oferta,excursie), certitudine 100
(la_mare,da), certitudine -30
(in_romania,da), certitudine 70
(departare,departe), certitudine 100
(buget_disponibil,mare), certitudine 100

Introduceti una din urmatoarele optiuni:
(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: reinitiaza
Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: consulta
'Ce tip de buget alocati pentru petrecerea concediului?'
(redus mediu mare)
: mare

'Preferati ca locul de petrecere a concediului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'
(aproape departe foarte_departe)
: departe

'Preferati sa petreceti concediul la mare?'
: da fc 90

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'
(sejur_2_saptamani sejur_1_luna excursie)
: sejur_1_luna

'In ce anotimp preferati sa va petreceti concediul?'
(vara iarna)
: iarna fc 50

loc_concediu este rio_de_janeiro
factorul de certitudine este 50

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum loc_concediu este rio_de_janeiro

loc_concediu este rio_de_janeiro a fost derivat cu ajutorul regulilor: 16

regula 16
Daca
la_mare
buget_disponibil este mare
anotimp este iarna
Atunci

loc_concediu este rio_de_janeiro 100

la_mare a fost derivat cu ajutorul regulilor: utiliz

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

anotimp este iarna a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: afisare_fapte
Fapte existente în baza de cunostinte:

(Atribut,valoare)

(loc_concediu,rio_de_janeiro), certitudine 50
(anotimp,iarna), certitudine 50
(tip_oferta,sejur_1_luna), certitudine 100
(la_mare,da), certitudine 90
(in_romania,da), certitudine 70
(departare,departe), certitudine 100
(buget_disponibil,mare), certitudine 100

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: reinitiaza
Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: consulta
'Ce tip de buget alocati pentru petrecerea concediului?'
(redus mediu mare)
: mare fc 35

'Preferati ca locul de petrecere a concediului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'
(aproape departe foarte_departe)
: foarte_departe

'Preferati sa petreceti concediul la mare?'
: nu fc 90

'In ce anotimp preferati sa va petreceti concediul?'
(vara iarna)
: iarna fc 80

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'
(sejur_2_saptamani sejur_1_luna excursie)
: sejur_2_saptamani

loc_concediu este chamonix
factorul de certitudine este 21

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum loc_concediu este chamonix

loc_concediu este chamonix a fost derivat cu ajutorul regulilor: 18

regula 18
Daca
departare este foarte_departe
not la_mare
tip_oferta este sejur_2_saptamani
buget_disponibil este mare
anotimp este iarna
Atunci
loc_concediu este chamonix 60

departare este foarte_departe a fost derivat cu ajutorul regulilor: utiliz

not la_mare a fost derivat cu ajutorul regulilor: utiliz

tip_oferta este sejur_2_saptamani a fost derivat cu ajutorul regulilor: utiliz

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

anotimp este iarna a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: afisare_fapte
Fapte existente în baza de cunostinte:

(Atribut,valoare)

(loc_concediu,chamonix), certitudine 21
(tip_oferta,sejur_2_saptamani), certitudine 100
(anotimp,iarna), certitudine 80
(la_mare,da), certitudine -90
(departare,foarte_departe), certitudine 100
(in_romania,da), certitudine 18
(buget_disponibil,mare), certitudine 35

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: iesire

yes

4.4. Rețele semantice și cadre

Sistemele bazate pe reguli nu constituie singurul cadru de lucru pentru reprezentarea cunoștințelor. Alte două moduri de lucru posibile constau în utilizarea așa-numitelor ***rețele semantice*** și, respectiv, ***cadre***, la care ne vom referi, pe scurt, în cele ce urmează. Acestea diferă de reprezentările bazate pe reguli în primul rând prin faptul că se concentrează asupra reprezentării, într-un mod structurat, a unor mulțimi foarte mari de fapte. Mulțimea de fapte reprezentată este *structurată* și, în măsura în care este posibil, *comprimată*: o serie de fapte nu sunt reprezentate în mod explicit atunci când ele pot fi reconstruite prin mecanismul de inferență. Atât rețelele semantice, cât și cadrele sunt ușor de reprezentat în Prolog, așa cum se va vedea în cele ce urmează.

4.4.1. Rețele semantice

O ***rețea semantică*** constă din entități și relații între acestea. Ea se reprezintă sub forma unui *graf orientat* ale cărui arce leagă vârfuri etichetate. Cel mai adesea nodurile grafului corespund entităților, în timp ce relațiile sunt figurate ca legături între noduri etichetate cu denumirile relațiilor. În Fig. 4.5 este prezentată, spre exemplificare, o rețea semantică. Exemplul a fost preluat de noi din [3], iar rețeaua este folosită pentru reprezentarea unor fapte de tipul următor:

- O pasăre este un anumit tip de animal.
- Metoda uzuală de mișcare a păsărilor este zborul.
- Un albatros este o pasăre.

- Mihai şi Tudor sunt albatroşi.

Se observă că relaţia de tip **isa** uneori realizează legătura dintre *o clasă* de obiecte şi o altă clasă având o sferă mai largă şi reprezentând un element supraordonat (*animal* este o clasă supraordonată a lui *pasăre*), iar alteori face legătura între *o instanţiere* a unei clase şi însăşi clasa respectivă (*Mihai* este un *albatros*).

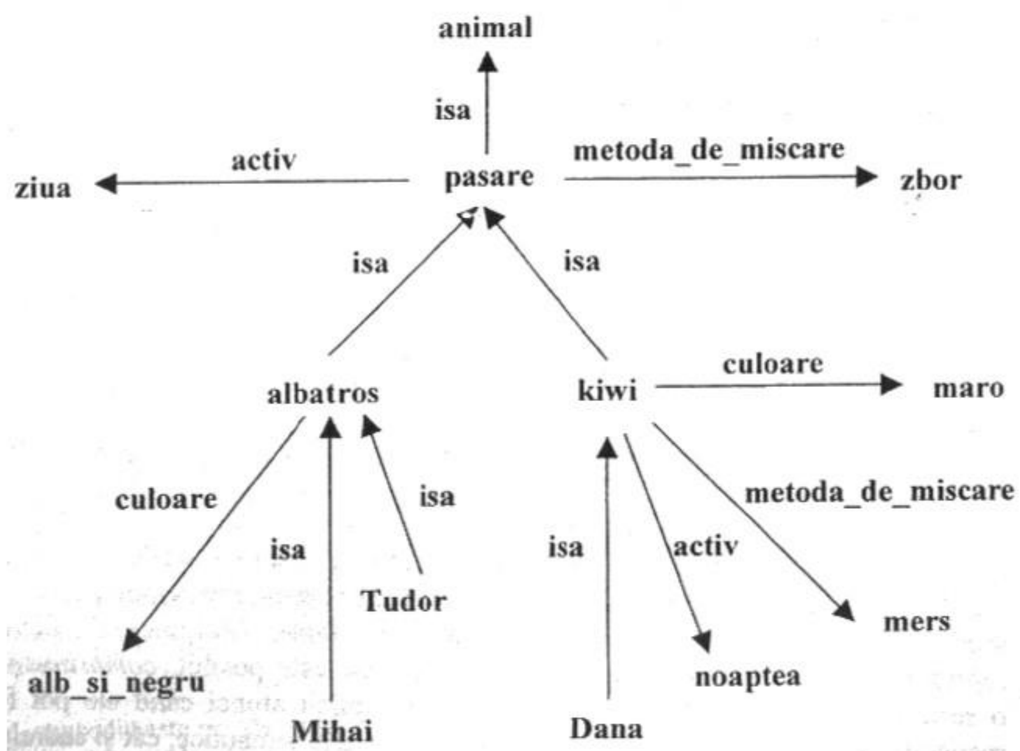


Fig. 4.5

O astfel de reţea este imediat "tradusă" sub forma unor fapte Prolog de tipul:

```
isa(pasare,animal).
isa(mihai,albatros).
metoda_de_miscare(pasare,zbor).
metoda_de_miscare(kiwi,mers).
```

Pe lângă faptele de acest tip, care sunt enunţate în mod explicit, alte fapte pot fi deduse pe baza aceleiaşi reţele. *Principiul tipic de inferenţă* încorporat în reţea este, aşa cum s-a mai menţionat, *moştenirea proprietăţilor*. Faptele sunt moştenite prin intermediul relaţiei **isa**. În Prolog putem enunţa

faptul că metoda de mişcare este moştenită, urcând de-a lungul ierarhiei isa, sub forma următoare:

```
metoda_de_miscare(X,Metoda):-
        isa(X,SuperX),
        metoda_de_miscare(SuperX,Metoda).
```

Pentru a nu enunţa câte o regulă de moştenire distinctă corespunzător fiecărei relaţii care poate fi moştenită, se va enunţa o regulă mult mai generală referitoare la fapte, fie ele enunţate în mod explicit în cadrul reţelei, fie moştenite:

```
fapta(Fapta):-                  %Fapta nu este o variabila;
                                %Fapta=Rel(Ar1,Arg2)
      Fapta,!.                  %Fapta explicita in retea

fapta(Fapta):-
      Fapta=..[Rel,Arg1,Arg2],
      isa(Arg1,SuperArg),                 %urcand de-a lungul
                                          %ierarhiei isa
      SuperFapta=..[Rel,SuperArg,Arg2].
```

Reţeaua semantică din Fig. 4.5 poate fi acum interogată în modul următor:

```
?-fapta(metoda_de_miscare(dana,Metoda)).
Metoda=mers
```

Această metodă de mişcare a fost moştenită de la faptul că păsările kiwi merg, aceasta fiind o faptă dată în mod explicit în cadrul reţelei. Pe de altă parte, metoda de mişcare a lui Mihai este moştenită de la *clasa pasăre* şi ea este, prin urmare, zborul:

```
?-fapta(metoda_de_miscare(mihai,Metoda)).
Metoda= zbor
```

Unul dintre subdomeniile inteligenţei artificiale în care este extrem de folosită reprezentarea cunoştinţelor prin intermediul reţelelor semantice este acela al *prelucrării limbajului natural*. Astfel, una dintre cele mai utile reprezentări ale *cunoştinţelor lexicale* este cea sub formă de reţea semantică. Avantajele oferite de reţelele semantice în reprezentarea cunoştinţelor lexicale sunt multiple. Printre acestea amintim faptul că reţelele semantice uşurează *construcţia lexiconului*, permiţând moştenirea proprietăţilor. În acelaşi timp, ele furnizează o mulţime foarte bogată de legături între sensurile cuvintelor, ceea ce

facilitează *dezambiguizarea*. Aceasta din urmă este, de regulă, realizată printr-o reprezentare ierarhică a sensurilor cuvintelor, cel mai adesea captată tocmai prin intermediul rețelelor semantice. Folosindu-se astfel de ierarhii pot fi definite restricții selecționale și pot fi utilizate aceste constrângeri pentru a reduce numărul de sensuri posibile ale unui cuvânt.

- **Un alt exemplu de rețea semantică. WordNet**

Un exemplu celebru de rețea semantică (din domeniul prelucrării limbajului natural) este WordNet, datorită căreia ***procesarea cunoștințelor*** a dobândit noi dimensiuni.

WordNet reprezintă în primul rând o *bază de date lexicală interactivă*, dezvoltată în ultimii 15 ani, pentru limba engleză, la Universitatea Princeton, de către un grup de cercetători condus de profesorul George Miller. În același timp, WordNet poate fi privită ca un *dicționar semantic*, deoarece cuvintele sunt localizate pe baza *afinităților conceptuale* cu alte cuvinte, spre deosebire de cazul dicționarelor clasice, unde cuvintele sunt ordonate alfabetic. Deși este similară unui tezaur, WordNet este mult mai utilă aplicațiilor inteligenței artificiale, întrucât este înzestrată cu o bogată mulțime de relații între cuvinte și sensuri ale cuvintelor. WordNet este implementată în limbajele Prolog, C și Java.

WordNet conține majoritatea substantivelor, verbelor, adjectivelor și adverbelor limbii engleze, organizate în mulțimi de sinonime numite **synset**-uri. Fiecare *synset* reprezintă un *concept*.

Rețeaua WordNet structurează informația lexicală în termeni de sensuri ale cuvintelor. Ea face corespondența dintre formele tip ale cuvintelor și sensurile acestora utilizând categoria sintactică ca parametru. Astfel, cuvintele aparținând aceleiași categorii sintactice, care pot fi folosite pentru a exprima același înțeles, sunt grupate într-un același synset. Cuvintele polisemantice aparțin mai multor synset-uri. Spre exemplu, cuvântul englezesc *computer* are două sensuri definite în WordNet, ceea ce face ca el să aparțină la doua synset-uri diferite, după cum urmează:

(1) {computer, data processor, electronic computer, information processing system}

și

(2) {calculator, reckoner, figurer, estimator, computer}.

În versiunea sa curentă (versiunea 2.0), WordNet conține 152059 cuvinte organizate în 115424 synset-uri. Cuvintele și conceptele sunt legate între ele prin ***relații semantice***. Există în total 279930 asemenea relații (219418 pentru substantive, 28163 pentru verbe, 28684 pentru adjective și 3665 pentru

adverbe). Toate aceste numere sunt însă aproximative, întrucât WordNet continuă să crească. Versiunea 2.0 este accesibilă la adresa:

http://www.cogsci.princeton.edu/~wn/obtain/

Relațiile semantice se stabilesc între cuvinte, între cuvinte și synset-uri, precum și între synset-uri. Fiecare cuvânt ținteşte către unul sau mai multe synset-uri, fiecare dintre acestea corespunzând unui anumit sens al cuvântului respectiv. Prin urmare, diferite cuvinte pot ținti către un același sens (synset). Bogăția mulțimii de relații stabilite între synset-uri este ceea ce face ca rețeaua semantică WordNet să fie atât de puternică și de interesantă pentru diverse tipuri de aplicații. Exemple de relații semantice existente în WordNet sunt **sinonimia** (*synonymy*), folosită pentru a forma synset-urile, **hiperonimia** (*hypernymy*) și **hiponimia** (*hyponymy*), corespunzând relației de tip *isa* și respectiv relației inverse (*reverse isa*), **meronimia** (*meronymy*), corespunzând relației *parte-din*, relația **cauzală** referitoare la verbe și altele. O importanță deosebită este atașată relațiilor de hiperonimie și de hiponimie ca relații între synset-uri.

Cu ajutorul relației de hiperonimie (sau de tip ***isa***) conceptele de ***substantiv*** și de ***verb*** sunt structurate sub formă de ***ierarhii***. Cele de ***adjectiv*** și de ***adverb*** au o structură diferită (***cluster***). Semantica relației de tip isa permite unui concept să moștenească toate proprietățile hiperonimelor sale. În plus, proprietățile tipice ale unui concept sunt enunțate sub formă de **glosă** atașată fiecărui concept în parte. Fiecare glosă include o definiție, una sau mai multe explicații suplimentare și unul sau mai multe exemple.

WordNet reprezintă o bază de date lexicală a limbii engleze care a fost adoptată pe scară largă pentru o întreagă varietate de aplicații practice din domeniul inteligenței artificiale, cu precădere din subdomeniul procesării limbajului natural. Mulți cercetători care utilizează WordNet, în special în domeniul inteligenței artificiale, consideră că aceasta reprezintă o ***bază de cunoștințe*** lexicală și o valorifică ca atare. *Procesarea cunoștințelor* a dobândit noi dimensiuni în S.U.A. datorită existenței WordNet. În același timp, comunitatea științifică internațională se arată extrem de interesată de dezvoltarea unor baze de date lexicale de tip WordNet pentru cât mai multe limbi, în încercarea de a crea *o infrastructură ontologică uniformă*. Astfel, întrucât mulțimea de bază a relațiilor care leagă între ele conceptele rămâne aceeași, indiferent de limbă, *algoritmii de inferență* pentru extragerea informației pot rămâne aceiași. Existența unei asemenea infrastructuri ontologice uniforme corespunzător diverselor limbi va simplifica traducerea automată și va facilita utilizarea aceloraşi scheme de raționament și a aceloraşi algoritmi concepuți pe baza rețelei americane WordNet.

Posibilele *aplicații ale WordNet* în cele mai variate domenii (***regăsirea informației, extragerea informației, dezambiguizarea, generarea limbajului natural, învățarea, dicționarele electronice, achiziția de cunoștințe*** ș.a.) sunt citate în peste 300 de lucrări științifice.

Este de menţionat faptul că, la mijlocul anilor '90, datorită multiplelor aplicaţii dezvoltate pe baza WordNet, a fost puternic resimţită nevoia de a se crea baze de date asemănătoare şi pentru alte limbi, în special pentru cele europene. Un imens efort ştiinţific şi financiar a fost lansat în Europa Occidentală, pentru a se crea aşa-numita **EuroWordNet**, utilizând varianta americană WordNet ca model. Acest efort ştiinţific s-a concretizat în anul 1996, în cadrul proiectului de cercetare - dezvoltare **EuroWordNet**, sub conducerea Universităţii din Amsterdam:

http://www.hum.uva.nl/~ewn/

În prezent există câte o bază de date lexicală de tip WordNet pentru limbile daneză, italiană şi spaniolă (fiecare aflată în continuă îmbunătăţire) şi se lucrează la unele similare pentru limbile germană, franceză, cehă şi estoniană. Tot în prezent se pune problema creării unor astfel de baze de date lexicale interactive pentru limbile din Europa Centrală şi de Est, folosindu-se varianta WordNet a limbii engleze ca model şi adaptând-o specificului fiecărei limbi în parte. Proiectul **BalkaNet**, finanţat de Comisia Europeană, se ocupă în prezent de aceste limbi:

http://www.ceid.upatras.gr/Balkanet/

Eforturile cercetătorilor (informaticienilor) se concentrează şi asupra problemei generării automate a unor baze de date de tip WordNet corespunzătoare diverselor limbi, generare care să pornească de la reţeaua semantică WordNet a limbii engleze. În cazul limbii române acest studiu a fost realizat, referitor la substantivele şi adjectivele româneşti, de către echipa RORIC-LING de la Universitatea din Bucureşti, în cadrul proiectului **BALRIC-LING** (finanţat tot de Comisia Europeană) şi este descris în pagina de web a acestui proiect:

http://phobos.cs.unibuc.ro/roric

4.4.2. Cadre

În reprezentarea bazată pe cadre faptele sunt concentrate în jurul obiectelor. Cuvântul "obiect" are aici fie sensul unui obiect fizic concret, fie al unui concept mai abstract, cum ar fi o clasă de obiecte sau chiar o situaţie.

Un ***cadru*** este o structură de date ale cărei componente sunt numite *slot*-uri (tăietură, deschizătură). Slot-urile pot găzdui informaţii de diverse naturi. Aici se pot găsi valori simple, referinţe la alte slot-uri sau chiar proceduri care pot calcula valoarea slot-ului pe baza altor informaţii. Un slot poate să fie vid, el putând fi umplut prin mecanismul de inferenţă. Ca şi în cazul reţelelor semantice, cel mai frecvent principiu de inferenţă este moştenirea proprietăţilor. Atunci când un cadru reprezintă o clasă de obiecte (albatros) iar un alt cadru reprezintă o superclasă a acestei clase (pasăre), cadrul corespunzător clasei poate moşteni valori de la cadrul reprezentând superclasa.

Un cadru, prin urmare, este o colecţie de atribute (slot-uri) şi de valori asociate (eventual şi de constrângeri asupra acelor valori), care descriu o anumită entitate a lumii. Uneori, un cadru descrie o entitate într-un sens absolut, alteori el reprezintă acea entitate numai dintr-un anumit punct de vedere. Un cadru unic este arareori util. În schimb, se pot construi întregi *sisteme de cadre* cu ajutorul unor colecţii de cadre care sunt conectate între ele prin faptul că valoarea unui atribut al unui anumit cadru este un alt cadru. În particular, este util să se atribuie cât mai multă structură atât nodurilor, cât şi legăturilor unei reţele. Deşi actualmente nu se face o distincţie clară între o reţea semantică şi un sistem de cadre, putem spune că, pe masură ce sistemul dobândeşte mai multă structură, el se apropie din ce în ce mai mult de un aşa-numit *sistem de cadre*.

Preluând acelaşi exemplu din [3] vom plasa o serie de cunoştinţe despre păsări în cadre, după cum urmează:

CADRU: pasare
un_fel_de: animal
metoda_de_miscare: zbor
activ: ziua

Acest cadru reprezintă clasa păsărilor. Iată cadrele pentru două subclase ale acestei clase - albatros şi kiwi:

CADRU: albatros
un_fel_de: pasare
culoare: alb_si_negru
marime: 115

CADRU: kiwi
un_fel_de: pasare
metoda_de_miscare: mers
activ: noaptea
culoare: maro
marime: 40

Albatros, ca element tipic al clasei păsărilor, moşteneşte metoda de mişcare (zborul), precum şi faptul că este activ în timpul zilei, de la cadrul pasăre. Din această cauză, nu se menţionează nimic relativ la **metoda_de_miscare** şi **activ** în cadrul referitor la albatros. Pe de altă parte, kiwi este o pasăre atipică şi nu respectă valorile pentru **metoda_de_miscare** şi **activ** ale clasei păsărilor.

Un cadru se poate referi şi la o anumită *instanţiere* a unei clase, ca în cazul albatrosului numit Mihai:

CADRU: Mihai
instantiere_pentru: albatros
marime: 120

Se observă faptul că, în cazul primelor cadre, relaţia **un_fel_de** este o relaţie între o clasă şi o superclasă, în timp ce relaţia **instantiere_pentru** din ultimul

exemplu este o relaţie între un membru (element) al unei clase şi clasa respectivă.

Informaţia inserată în cadre poate fi reprezentată în Prolog ca o mulţime de fapte, câte o faptă Prolog corespunzând fiecărei valori a unui slot. Formatul pentru aceste fapte ales în [3] şi pe care îl prezentăm aici este următorul:

```
Nume_cadru(Slot,Valoare).
```

Avantajul acestui format este acela că toate faptele referitoare la un anumit cadru sunt colectate laolaltă de către relaţia a cărei denumire este chiar numele cadrului. Două exemple de cadre reprezentate în acest format sunt următoarele:

```
%Cadrul kiwi:
kiwi(un_fel_de,pasare).
kiwi(metoda_de_miscare,mers).
kiwi(activ,noaptea).
kiwi(marime,40).
kiwi(culoare,maro).

%Cadrul mihai:
mihai(instantiere_pentru,albatros).
mihai(marime,120).
```

Pentru a putea folosi o astfel de mulţime de cadre este necesară existenţa unei proceduri pentru regăsirea faptelor referitoare la valorile corespunzătoare slot-urilor. O procedură Prolog cu acest rol este propusă în [3] ca fiind de forma

```
valoare(Cadru,Slot,Valoare)
```

unde `Valoare` este valoarea corespunzătoare slot-ului `Slot` în cadrul `Cadru`. Dacă slot-ul este plin - valoarea sa este dată în mod explicit în cadru - atunci aceasta este valoarea; altfel, această valoare poate fi obţinută prin inferenţă, de pildă ca o consecinţă a moştenirii proprietăţilor. Pentru a găsi o valoare prin moştenire, trebuie să ne deplasăm de la cadrul curent la un cadru mai general, în conformitate cu relaţia `un_fel_de` sau cu relaţia `instantiere_pentru` dintre cadre. În acest fel se ajunge la un "cadru-părinte", iar valoarea poate fi găsită în mod explicit în acest cadru sau prin continuarea procesului de moştenire.

Acest proces de regăsire a informaţiei din cadre, fie că este o regăsire directă, fie că este o regăsire prin moştenire, poate fi exprimat în Prolog după cum urmează:

```
valoare(Cadru,Slot,Valoare):-
%valoare regasita in mod direct
      Interogare=..[Cadru,Slot,Valoare],
      call(Interogare),!.
valoare(Cadru,Slot,Valoare):-
%un cadru mai general
      parinte(Cadru,CadruParinte),
      valoare(CadruParinte,Slot,Valoare).

parinte(Cadru,CadruParinte):-
      (Interogare=..[Cadru,un_fel_de,CadruParinte];
      Interogare=..[Cadru,instantiere_pentru,
                    CadruParinte]),
      call(Interogare).
```


Interogarea Prologului se poate face atunci în felul următor:

```
?-valoare(mihai,activ,TimpMihai).
TimpMihai = ziua

?-valoare(kiwi,activ,TimpKiwi).
TimpKiwi = noaptea
```

Inferenţa între cadre este un proces extrem de subtil, care ridică diverse probleme ce nu vor fi tratate aici, ele neconstituind obiectul acestei lucrări. Dintre acestea amintim, în primul rând, cazul mai complicat de inferenţă în care, corespunzător unui slot, este dată o *procedură de calcul* a valorii, în locul valorii efective. O altă subtilitate a inferenţei între cadre o constituie aşa-numita *moştenire multiplă*. Această problemă se ridică atunci când un cadru are mai multe cadre părinte (conform relaţiei `instantiere_pentru` sau relaţiei `un_fel_de`). În acest caz o valoare a unui slot moştenită poate proveni de la mai mult de un cadru părinte, problema care se pune fiind aceea a adoptării valorii corecte. Soluţia propusă în [3], de pildă, este aceea de a se prelua prima valoare întâlnită, găsită în urma efectuării unei *căutări de tip depth-first* printre cadrele care ar putea furniza această valoare. Alte soluţii există, în egală măsură, principala concluzie care se desprinde în urma acestei succinte prezentări fiind aceea că ***reprezentarea cunoştinţelor şi procesul de căutare se află într-o strânsă interdependenţă***.

4.5. Consideraţii finale

Sistemele de reprezentare a cunoştinţelor amintite sau discutate de noi până în prezent au fost: structurile de tip slot-and-filler, sistemele bazate pe reguli, reţelele semantice şi cadrele. Capitolul 6 al lucrării de faţă va introduce sistemele de raţionament statistic şi se va ocupa de două tipuri de abordări statistice pentru sisteme de raţionament incert (factorii de certitudine în sistemele bazate pe reguli şi, respectiv, reţelele Bayesiene). Indiferent de tipul de structură aleasă pentru reprezentarea cunoştinţelor, a devenit însă evident

faptul că aceste sisteme trebuie nu numai să încorporeze cunoştinţe, ci, în egală măsură, să furnizeze o mulţime de proceduri de inferenţă de bază, cum ar fi cea referitoare la moştenirea proprietăţilor.

Structurile de tip slot-and-filler s-au dovedit foarte valoroase pentru memorarea şi regăsirea eficientă a cunoştinţelor în programele inteligenţei artificiale. Ele au însă o performanţă scăzută atunci când se pune problema reprezentării unor aserţiuni de tipul regulilor "dacă *x*, *y* şi *z*, atunci trage concluzia *w*". Calculul predicatelor este mult mai adecvat pentru reprezentarea unor asemenea aserţiuni, dar prezintă, la rândul său, dezavantajul de a fi ineficient pentru raţionamentul general care le utilizează. Reprezentările de tip slot-and-filler sunt mai orientate către semantică, ceea ce face ca procedurile lor de raţionament să fie mai variate, mai eficiente şi mai strâns legate de tipuri specifice, particulare de cunoştinţe. Pe de altă parte, în cadrul structurilor slot-and-filler este dificil să se exprime aserţiuni mai complexe decât moştenirea.

Reţelele semantice sunt proiectate cu scopul principal de a capta relaţii, legături şi raporturi semantice între entităţi. Ele sunt, de regulă, utilizate împreună cu o mulţime de reguli de inferenţă, proiectate special pentru a putea trata în mod corect tipurile de arce specifice reţelei.

Sistemele bazate pe cadre sunt în general mai structurate decât reţelele semantice şi ele conţin o mulţime chiar mai mare de reguli de inferenţă specializate, inclusiv cele care implementează o întreagă matrice de reguli de moştenire implicite, precum şi alte proceduri, cum ar fi verificarea consistenţei.

CAPITOLUL 5
GENERAREA SEMIAUTOMATĂ A REŢELELOR SEMANTICE DE TIP WORDNET

În acest capitol ne vom referi pe larg la reţeaua semantică **WordNet (WN)**, care are o semnificaţie aparte în domeniul procesării limbajului natural şi care a fost menţionată şi descrisă de noi în §4.4.1. al lucrării de faţă (vezi p.172).

WN reprezintă, aşa cum s-a arătat deja (§4.4.1), o bază de date lexicală a limbii engleze, care a fost adoptată pe scară largă pentru o întreagă varietate de aplicaţii practice din domeniul inteligenţei artificiale, cu precădere din subdomeniul prelucrării limbajului natural. WN reprezintă, în egală măsură, o bază de cunoştinţe lexicale, fiind valorificată ca atare în procesarea cunostinţelor, care a dobândit noi dimensiuni în S.U.A. datorită existenţei sale. Posibilele aplicaţii ale WN în cele mai variate domenii (regăsirea informaţiei, extragerea informaţiei, dezambiguizarea, generarea limbajului natural, învăţarea, dicţionarele electronice, achiziţia de cunoştinţe ş.a.) au determinat un interes crescând al comunităţii ştiinţifice internationale în dezvoltarea unor baze de date lexicale de tip WN pentru cât mai multe limbi. Eforturile informaticienilor se concentrează şi asupra problemei generării automte a unor asemenea baze de date, corespunzător diverselor limbi, generare care să pornească de la reţeaua semantică WN a limbii engleze. Capitolul de faţă se va ocupa de această problemă şi va propune un algoritm pentru generarea semiautomată a synset-urilor şi cluster-elor de tip WN, algoritm care va folosi în mod esenţial structura WN de reţea semantică.

Aşa cum remarcă autorii ei (Miller et. al., 90), WN este o bază de cunostinţe lexicală, dezvoltată mai întâi pentru engleză şi apoi adoptată pentru mai multe limbi vest-europene, care a fost creată sub forma unui dicţionar electronic bazat pe principii psiholingvistice. Studiul de faţă este o tentativă de a discuta generarea semiautomată a WN-urilor pentru alte limbi decât engleza, un subiect de mare interes, întrucât existenţa unor asemenea WN-uri va crea infrastructura adecvată pentru sisteme avansate de tehnologie a informaţiei. Extinzând abordarea algoritmică introdusă de Nikolov şi Petrova [24], propunem o metodă semiautomată, bazată pe euristici, pentru generarea synset-urilor şi cluster-elor de tip WN. Ne vom concentra atenţia asupra synset-urilor de substantive şi de adjective, întrucât substantivele şi adjectivele au în WN organizări complet diferite, dar va fi adresată şi problema generării synset-urilor de verbe şi de adverbe. Limba ţintă pentru efectuarea testelor va fi româna. Abordarea noastră referitoare la generarea WN se bazează pe

aşa-numitele "metode de clasă", cu alte cuvinte foloseşte ca surse de cunostinţe intrări individuale provenind din dicţionare bilingve şi synset-uri ale WN, dar, în acelaşi timp, demonstrează necesitatea de a combina asemenea metode cu unele de natură structurală.

5.1. Consideraţii introductive

Studiul de faţă se concentrează asupra subiectului important şi actual al generării automate a WN-urilor pentru alte limbi decât engleza. Abordarea privind generarea WN constă dintr-o metodă semiautomată bazată pe euristici, care aparţine aşa-numitelor **metode de clasă** [2]. Prin urmare, ea foloseşte ca surse de cunoştinţe intrări individuale provenind din dicţionare bilingve şi synset-uri ale WN. Astfel, succesul metodei noastre depinde în mod direct de existenţa unor dicţionare bilingve comprehensive în format electronic.

Algoritmul de traducere de bază (Algoritmul 5.1 din lucrarea de faţă) va folosi aşa-numite "mulţimi elementare", un concept introdus în [23]. Algoritmul 5.1, descris în [24], va fi, în continuare, completat cu Algoritmul 5.2, propus în [11], care efectuează o acţiune de backtracking (pasul 1) pentru a obţine, ca output final, synset-ul străin corespunzator celui englezesc dat. Ar trebui consemnat faptul că autorii bulgari care descriu primii [24] Algoritmul 5.1, având ca output o listă sortată de mulţimi elementare, nu fac nici un fel de comentariu privitor la felul în care obţin synset-ul străin final, în cazul lor synset-ul de substantive bulgar final. Se poate uşor presupune că acesta este obţinut de către lingvişti în mod manual, folosindu-se output-ul Algoritmului 5.1. Automatizarea procesului creării unui synset străin de tip WN în gradul cel mai înalt cu putinţă a constituit preocuparea lui F. Hristea în [11], iar comentariile noastre privitoare la output-ul obţinut în cazul limbii române vor fi făcute în acest cadru de lucru[19].

Întrucât aceiaşi autori bulgari nu specifică [24] ce funcţie de evaluare a fost utilizată, vor fi făcute aici comentarii adiţionale, luându-se în consideraţie output-ul românesc menţionat, cu privire la tipul de funcţie de evaluare care a fost sau ar trebui folosită în procesul de traducere.

În sfârşit, spre lauda autorilor menţionaţi, care sunt preocupaţi numai de obtinerea "unui miez de synset-uri bulgăreşti de substantive", se dovedeşte ca algoritmul acestora poate fi extins (cu mai mult sau mai puţin succes) la cazul general al oricarei limbi straine (nu numai bulgara). În plus, este credinţa noastră ca Algoritmul 5.1 poate fi utilizat cu succes în cazul tuturor celorlalte (trei) părţi de vorbire de care se ocupă WN, cu condiţia să fie modificat în mod corespunzator. Astfel de modificări ar trebui să ia în consideraţie relaţiile semantice tipice implementate în WN cu privire la fiecare parte de vorbire,

[19] Şi utilizând **versiunea 1.7.1** a bazei de date WN.

combinând, în acest fel, metoda de clasă folosită iniţial în cazul substantivelor cu o abordare structurală a problemei generării WN (a se vedea diversele tehnici de îmbogăţire propuse în cadrul studiului de faţă).

Întrucât în WN adjectivele au o organizare complet diferită de cea a substantivelor (pentru care vezi [22], [13] şi Anexa 1) - hiperspaţiul N-dimensional - studiul nostru privitor la această parte de vorbire este în continuare extins prin luarea în consideraţie a generării semiautomate a cluster-elor de adjective străine. În acest moment abordarea noastră va face, din nou, legăturile necesare între metodele de clasă şi cele **structurale** (adică acelea care profită de structura WN). Algoritmul 5.3, propus în [11], reprezintă o primă abordare a generării semiautomate a cluster-elor străine de adjective care nu foloseşte resurse monolingve, ci numai bilingve, mai exact dicţionare bilingve în format electronic.

5.2. Algoritmul de traducere

Algoritmul de traducere a unui synset englezesc dat în synset-ul corespunzător dintr-o limbă diferită de limba engleză va folosi aşa-numite "mulţimi elementare" sau **e-mulţimi**, concept introdus în [23]. O e-mulţime corespunde sensului unui cuvânt şi poate fi definită după cum urmează:

Definiţia 5.1
O e-mulţime relativă la un cuvânt este mulţimea sinonimelor corespunzând unui anumit sens al acelui cuvânt.

Să notăm prin **CE** orice cuvânt englezesc şi prin **CS** orice cuvânt străin, adică orice cuvânt al unei alte limbi decât limba engleză. Fie *cuvante* din secvenţa (1) un CE, în timp ce *cuvants1, cuvants2 şi cuvants3* ale aceleiaşi secvenţe reprezintă echivalentele lui obţinute prin traducere (conform dicţionarului bilingv adecvat):

cuvante cuvants1; cuvants2, cuvants3 (1)

Pentru a face distincţia între *cuvants1, cuvants2* şi *cuvants3*, în dicţionarele standard se folosesc doi separatori diferiţi. Simbolul "punct şi virgulă" [;] separă sensuri diferite ale unui cuvânt dat. La rândul ei, "virgula" [,] separă acele sinonime care se referă la un acelaşi sens al cuvântului. (În cazul de faţă, *cuvants2* şi *cuvants3* sunt sinonime). Aceasta este forma unui dicţionar bilingv care va fi folosită de către programele ce implementează algoritmul de traducere propus. În exemplul de mai sus, e-mulţimile care intervin sunt următoarele:

{*cuvants1*} şi {*cuvants2, cuvants3*}.

Programele care implementează algoritmul de traducere vor genera lista tuturor e-mulţimilor de cuvinte străine (CS-uri) corespunzătoare sensului tuturor

cuvintelor englezeşti (CE-uri) care intervin într-un synset englezesc dat. Synset-ul străin corespunzător celui englezesc studiat este format din una sau mai multe dintre e-mulţimile generate (care pot fi reunite). "Candidaţii" la includerea în synset-ul străin sunt *e-mulţimi etichetate*, adică acele e-mulţimi care conţin *cuvinte etichetate*.

Pentru a eticheta CS-urile aparţinând e-mulţimilor generate, s-a decis etichetarea, mai întâi, a CE-urilor aparţinând synset-ului englezesc studiat. Aceste CE-uri vor fi etichetate, în ordinea apariţiei lor, cu numere întregi de la 1 la *n* (unde *n* reprezintă dimensiunea synset-ului, adică numărul de cuvinte pe care acesta îl conţine). După etichetarea CE-urilor din synset-ul dat, CS-urile aparţinând e-mulţimilor generate vor fi căutate în dicţionarul bilingv corespunzător. De fiecare dată când un CE din synset-ul dat reprezintă, conform dicţionarului, traducerea unui CS, acest CS primeşte eticheta CE-ului respectiv. Dacă cel puţin un cuvânt al unei e-mulţimi străine poate fi tradus prin unul dintre cuvintele synset-ului englezesc dat, atunci întreaga e-mulţime străină este inclusă în "lista candidaţilor". Aşa cum se remarcă în [24], atunci când devine completă, această listă a candidaţilor reprezintă cel mai important rezultat preliminar. Synset-ul străin adecvat va reprezenta o combinaţie a unor e-mulţimi aparţinând acestei liste. Au fost concepute diverse *funcţii de evaluare* care sortează e-mulţimile extrase, punându-le în evidenţă pe cele mai adecvate. Pentru a defini astfel de funcţii de evaluare, ne vom referi, mai întâi, la următoarele concepte:

Definiţia 5.2
Eticheta unei e-mulţimi reprezintă numărul de etichete atribuite cuvintelor aparţinând acelei e-mulţimi.

Definiţia 5.3
O e-mulţime este *neetichetată* dacă ea nu conţine nici un cuvânt etichetat.

Orice cuvânt poate avea atribuite una sau mai multe etichete, precum şi nici o etichetă. Cea mai simplă funcţie de evaluare care este propusă în literartura de specialitate [24] are ca argument o e-mulţime şi ia valoarea dată de însăşi eticheta acelei e-mulţimi. O variantă a acestei funcţii de evaluare este aceea care împarte numărul reprezentând eticheta e-mulţimii la cardinalul aceleiaşi e-mulţimi.

În ceea ce ne priveşte, am luat în consideraţie funcţia de evaluare care este definită mai jos.

Fiecare CE aparţinând synset-ului englezesc dat va avea o etichetă (reprezentată printr-un număr întreg de la 1 la *n*, unde *n* este dimensiunea synset-ului), iar etichetarea CS-urilor aparţinând e-mulţimilor va fi efectuată în

conformitate cu această etichetă. Etichetele CS-urilor care diferă de eticheta CE-ului corespunzător vor fi considerate ca reprezentând două puncte, în timp ce valoarea celorlalte etichete este de numai un punct. Valoarea funcţiei de evaluare relativ la o anumită e-mulţime este dată de numărul total de puncte corespunzător e-mulţimii, împărţit la cardinalul acesteia.

Având definite toate conceptele necesare, putem, în cele ce urmează, să enunţăm algoritmul de ***generare a e-mulţimilor străine*** corespunzătoare unui synset englezesc dat:

Algoritmul 5.1

Input: Fişierul conţinând synset-urile englezeşti şi fişierele reprezentând cele două dicţionare bilingve (spre exemplu, Dicţionarul englez-român şi respectiv Dicţionarul român-englez).

1. Creează (prin consultarea dicţionarului bilingv adecvat) e-mulţimile corespunzătoare fiecărui cuvânt al synset-ului englezesc dat.
2. Etichetează cuvintele englezeşti aparţinând synset-ului englezesc dat.
3. Etichetează fiecare dintre e-mulţimile generate la pasul 1.
4. Îndepărtează toate mulţimile neetichetate.
5. Evaluează e-mulţimile (folosind etichetele atribuite şi o funcţie de evaluare).
6. Sortează, în ordine crescătoare sau descrescătoare, lista de e-mulţimi evaluate obţinută.

Output: Lista sortată a e-mulţimilor corespunzătoare synset-ului englezesc dat.

□

Traducerile în limba străină ale cuvintelor care intervin în synset-ul englezesc sunt extrase din dicţionarul bilingv după cum urmează:

cuvante1 sens*11*; sens*12*; ... ; sens$1m_1$
.....
cuvanten sens*n1*; sens*n2*; ... ; sensnm_n

Mulţimea e-mulţimilor generată de Algoritmul 5.1 este de următoarea formă:

$$\{\{sens_{ij}\} \mid 1 \le i \le n, 1 \le j \le m_i\}.$$

Synset-ul străin va fi generat folosind această mulţime.

În generarea automată a *synset-ului străin* corespunzător unui synset englezesc dat vom lua, de asemenea, în consideraţie

Observaţia 5.1
Dintre toate sensurile posibile ale unui cuvânt, numai unul se referă la un anumit concept (căruia îi corespunde un synset).

Folosind lista sortată a e-mulţimilor generată de Algoritmul 5.1 (cu alte cuvinte, e-mulţimile evaluate) va fi ales, corespunzător fiecărui cuvânt englezesc, sensul (mulţimea elementară) evaluat la cea mai mare valoare. Fie acest sens, corespunzător lui *cuvantej*, $sens_{ij_j}$. ***Synset-ul străin*** va fi generat folosind e-mulţimile obţinute prin intermediul Algoritmului 5.1, luând în consideraţie Observaţia 5.1 şi conform

Algoritmul 5.2

Input: Lista sortată a e-mulţimilor generată de Algoritmul 5.1 corespunzător synset-ului englezesc dat [*cuvante1*, *cuvante2*,..., *cuvanten*].

1. Determină synset-ul străin ca având următoarea formă:

   $\{sens1_{i_1}\} \cup \{sens2_{i_2}\} \cup \ldots \cup \{sensn_{i_n}\}, \quad 1 \le i_j \le m_j, \ \forall\ j = \overline{1,n}$

2. Şterge, din această reuniune, cuvintele aparţinând mai multor e-mulţimi, astfel încât fiecare cuvânt să apară o singură dată.

Output: Synset-ul străin corespunzător synset-ului englezesc dat.

□

A devenit acum evident faptul că abordarea noastră referitoare la generarea WN aparţine clasei de metode semiautomate bazate pe euristici. După cum se ştie [2], astfel de euristici pot aparţine la două mari categorii: una în care euristicile corespunzatoare se bazează pe informaţia găsită în dicţionarele bilingve şi în structura WN, o alta conţinând euristici care se bazează pe informaţia specifică extrasă din dicţionarul monolingv. In mod evident, euristica folosită aici aparţine primei categorii menţionate, întrucât metoda noastră de generare nu foloseşte resurse monolingve (cu excepţia a însaşi WN) ci se bazează exclusiv pe dicţionare bilingve (în format electronic).

Un exemplu

Synset englezesc: {personification, incarnation}

Glosa: the act of attributing human characteristics to abstract ideas etc.

---

e-multimi:

cuvante	e-multime	scor
incarnation	{personificare}	2.0
incarnation	{incarnare, intruchipare, intrupare}	1.3333334
personification	{personificare, intruchipare}	1.0

Synset romanesc propus:

- {personificare, intruchipare}

5.3. Synset-uri de substantive

Algoritmii 5.1 si 5.2 au fost implementaţi în Prolog şi testaţi de către noi, cu rezultate suficient de bune, în cazul *substantivelor româneşti*. Pentru a testa algoritmii, am folosit fragmente de dicţionare bilingve în format electronic. Atunci când se lucrează cu o reţea semantică de tipul WN, bogăţia dicţionarelor bilingve care se folosesc este esenţială. Datoriă imperfecţiunii dicţionarelor român-englez şi englez-român în format electronic existente (vezi, de pildă, www.castingsnet.com/dictionaries) şi pentru a asigura cea mai fidelă traducere cu putinţă, am generat propriile noastre fragmente de dicţionare electronice bilingve, folosind unele dintre cele mai complete dicţionare neelectronice existente [16], [17]. Dicţionarele român-englez şi englez-român compilate folosite în testările noastre pot fi văzute la adresele

http://phobos.cs.unibuc.ro/roric/wn/r_e.dict

şi respectiv

http://phobos.cs.unibuc.ro/roric/wn/e_r.dict

A fost ales în mod aleator un numar de 200 synset-uri englezeşti de substantive pentru care am generat în mod automat pe cele româneşti corespunzatoare. Întrucât majoritatea synset-urilor englezeşti conţin două cuvinte, eşantionul nostru de date a fost ales în conformitate cu acelaşi model. Astfel, dintre cele 200 de synset-uri englezeşti luate în consideraţie, 179 conţineau câte două

substantive englezeşti, patru synset-uri conţineau trei substantive englezeşti şi 17 synset-uri conţineau mai mult de trei substantive englezeşti (între patru şi şapte cuvinte). Numarul de e-mulţimi implicate în experiment a fost de 616. Mai multe synset-uri englezeşti conţinând un unic substantiv au fost ulterior luate în consideraţie. Toate testele efectuate au folosit versiunea 1.7.1 a WN în format Prolog.

Synset-urile româneşti generate au fost validate de către lingvişti români, folosindu-se cele mai recente dicţionare bilingve şi glosa corespunzatoare, indicată în WN-ul american.

Atunci când s-a testat algoritmul de traducere referitor la substantivele româneşti, am observat că, în diverse cazuri, Algoritmul 5.2 a generat mai mult de un synset românesc corespunzător celui englezesc dat. Aceasta a fost situaţia atunci când Algoritmul 5.1 a avut ca output o listă de e-mulţimi (corespunzând mai multor sensuri ale aceluiaşi cuvânt) care fuseseră evaluate cu aceeaşi valoare. Fiecare e-mulţime reprezenta atunci un candidat şi conducea la un synset românesc sau, în general, străin diferit. În astfel de cazuri, synset-ul străin corect va fi ales din lista de synset-uri generată de Algoritmul 5.2 în conformitate cu glosa synset-ului englezesc dat. Programul de calculator care implementează Algoritmul 5.2 trebuie, prin urmare, să furnizeze ca output şi glosa, întrucât aceasta este necesară în validarea efectuată de către lingvişti.

Atunci când s-au efectuat teste referitoare la substantivele româneşti a reieşit că, în afara cazurilor în care rezultatul a fost corect, în majoritatea celorlalte cazuri algoritmul generase mai multe synset-uri româneşti, printre care putea fi găsit şi cel corect. În acele cazuri în care synset-urile englezeşti nu aveau corespondente româneşti corecte, aceasta se datora cel mai mult datelor greşite sau care lipseau din dicţionarele bilingve. Probleme speciale au survenit în cazul synset-urilor englezeşti conţinând un singur substantiv polisemantic. Pentru comentarii mai detaliate cu privire la output-ul obţinut, vezi §5.5.

Pentru a uşura experimentul, atunci când ne-am ales eşantionul de synset-uri englezeşti, un pas necesar a fost acela al înlaturării synset-urilor cu nume proprii, cuvinte compuse şi colocaţii. Acestea ar trebui tratate separat şi cu o contribuţie mai semnificativă din partea lingviştilor. Cu toate acestea, considerăm algoritmii prezentaţi ca fiind suficienţi pentru construirea unui ***miez*** de synset-uri corespunzător tuturor celor patru părţi de vorbire, în mai mult sau mai puţin orice limbă diferită de limba engleză, cu condiţia să existe dicţionare bilingve în format electronic complete, pentru limba străină concretă implicată.

Aşa cum se menţionează în [24], cel mai mare avantaj al Algoritmului 5.1 este abilitatea sa de a crea synset-uri care pot include cuvinte străine ce nu ar fi extrase din resursa constituind input-ul la primul pas al traducerii. Astfel, chiar dacă un cuvânt străin apare în Dicţionarul englez-român, spre exemplu, dar lipseşte din cel român-englez, există încă o mare şansă ca acest cuvânt să fie inclus în synset-ul rezultat. (Singura condiţie necesară pentru aceasta este

prezenţa în lista candidaţilor a unei e-mulţimi care include cuvântul respectiv). Acest fapt este extrem de important, având în vedere cât de incomplete sunt, de regulă, dicţionarele bilingve. Algoritmul propus nu reprezintă, prin urmare, o simplă "traducere în oglindă".

În mod evident, atunci când se utilizează Algoritmii 5.1 şi 5.2 pentru diverse limbi particulare, felurite dificultăţi vor interveni, în funcţie de ceea ce este specific fiecărei limbi la nivel morfologic şi derivativ. O dificultate generală, de natură diferită, întâlnită indiferent ce limbă este luată în consideraţie, constă în ceea ce am putea numi "sensul larg de-a lungul limbilor" al unui cuvant dat. Cu alte cuvinte, într-o limbă dată, un anumit cuvânt se poate uneori referi la un concept relativ general şi este pus în legătură cu mai multe cuvinte dintr-o altă limbă, în care fiecare dintre cuvintele de care el este legat descrie un concept mai specializat. Aceasta este o chestiune extrem de importantă din punctul de vedere al abordării bazate pe WN, întrucât synset-urile din WN există în funcţie de conceptele aflate la baza lor.

În cazul limbii române, s-a ajuns la concluzia că, în acele situaţii, mai interesante, în care dicţionarele bilingve nu sunt de vină, principalele probleme care intervin atunci când synset-urile englezeşti sunt traduse automat în unele româneşti au fost generate de calcul lingvistic[20] şi de faptul că polisemia multor cuvinte englezeşti este în mare măsură superioară celei a cuvintelor româneşti corespunzătoare (pentru mai multe detalii, vezi §5.5).

Un caz special îl constituie cel al synset-urilor englezeşti conţinând un singur cuvânt polisemantic, o situaţie în care Algoritmul 5.1, sau oricare alt algoritm de acelasi tip, nu va fi capabil să distingă între diversele sensuri. Acest tip de dificultate ne-a sugerat tehnica de îmbogăţire care este propusă în §5.4.1 cu privire la adjective, precum şi alte posibile îmbogăţiri, care vor fi prezentate în cele ce urmează. Astfel de îmbogăţiri de synset-uri vor lua în consideraţie structura WN, dovedind necesitatea combinării metodelor de clasă cu cele structurale.

În ciuda unor asemenea dificultăţi, considerăm totuşi algoritmii de traducere prezentaţi ca fiind adecvaţi pentru realizarea extragerii semiautomate a miezului unei WN străine, pornind de la WN originală. Cea mai importantă problemă aici este faptul că Algoritmii 5.1 si 5.2 nu depind de tipul părţii de vorbire implicate în traducere. Este, prin urmare, natural să aşteptăm rezultate similare sau chiar mai bune decât cele obţinute pentru substantive atunci când se testează cu privire la alte părţi de vorbire, cum ar fi adjectivul. Mai ales întrucât adjectivele sunt mai puţin polisemantice decât adjectivele.

---

[20] Într-o accepţie foarte largă, prin *calc* se denumeşte procedeul de transpunere literală, exactă, a unui cuvânt semantic analizabil, a unei construcţii sau numai a unui sens, dintr-o limbă **A** într-o limbă **B**, cu materialul limbii **B**.

În cele ce urmează, vom stabili modul în care acest algoritm general trebuie îmbogăţit pentru ca el să realizeze generarea semiautomată a **synset-urilor** şi **cluster-elor de adjective** în alte limbi decât engleza.

5.4. Synset-uri şi cluster-e de adjective

WN împarte adjectivele în două mari clase: adjective *descriptive* şi *relaţionale*. Adjectivele *cromatice*, care desemnează culori, sunt privite ca reprezentând un caz special.

În cele ce urmează, ne vom ocupa de generarea semiautomată a ***cluster-elor de adjective*** în alte limbi decât engleza şi, prin urmare, ne vom referi exclusiv la ***adjectivele descriptive***, care pot fi organizate sub forma unei structuri de acest tip (vezi Anexa 1).

Traducerea cluster-elor de adjective englezeşti este complet asigurată de către traducerea synset-urilor englezeşti de adjective şi cea a relaţiei **ant** (care desemnează antonime).

5.4.1. Generarea semiautomată a synset-urilor de adjective

Pentru a traduce într-o limbă străină synset-urile de adjective englezeşti, au fost folosiţi Algoritmii 5.1 şi 5.2. Atunci când se traduce din engleză în orice altă limbă, ***id***[21]-ul care este asociat unui synset nu se modifică. Aceasta

[21] Baza de date WN în format Prolog este conţinută în fişierele **wn_*.pl** şi foloseşte ***synset_id*** - uri ca identificatori unici de synset-uri. Un *synset_id* este un câmp de 9 bytes în care primul byte defineşte categoria sintactică a synset-ului, iar cei 8 bytes rămaşi reprezintă un *synset_offset*.

A fost creat câte un fişier separat pentru fiecare relaţie din WN, ceea ce îi oferă utilizatorului posibilitatea de a încărca numai acele părţi ale acestei imense baze de date care îi sunt necesare la un moment dat. Fiecare fişier al bazei de date Prolog conţine informaţii corespunzătoare synset-urilor şi sensurilor cuvintelor incluse în baza de date WN. Fiecare linie a unui fişier conţine un operator care va corespunde unei relaţii din WN. Toate liniile cu aceeaşi valoare a operatorului sunt incluse în fişierul wn_*operator.pl*. Forma generală a unei linii dintr-un fişier al bazei de date Prolog este

operator(camp1,...,campn).

Ea nu conţine nici un spaţiu şi se termină cu un caracter de tip "newline".

Un operator **s** este prezent pentru fiecare sens al unui cuvânt din WN, în timp ce operatorul **g** specifică glosa corespunzătoare unui synset. Fişierul **wn_s.pl** conţine toate synset-urile din WN, în timp ce fişierul **wn_g.pl** conţine toate glosele.

Fişierul **wn_s.pl** ce conţine toate synset-urile din WN este format din fapte Prolog având forma generală

s(*synset_id,nr_cuv,'cuvant',tip_ss, nr_sens, stare_etic*).

unde:

înseamnă că relaţia de similaritate existentă între două synset-uri englezeşti se va menţine după efectuarea traducerii şi va interveni, în egală măsură, între synset-urile de adjective ale limbii străine respective.

O problemă specială o ridică synset-urile conţinând un singur cuvânt polisemantic. În acest caz este imposibil de ştiut care dintre sensurile cuvântului a intervenit în crearea acelui synset dacă nu avem acces la nici o informaţie suplimentară. Sensul poate fi dedus numai pe baza glosei. Totuşi, în astfel de cazuri, s-a folosit o strategie constând în îmbogăţirea synset-ului dat cu noi adjective sugerând sensul celui care intervine în synset-ul iniţial. Noile adjective sunt obţinute folosindu-se relaţia de similaritate, care există în mod tipic în WN între synset-urile de adjective. Astfel, pentru a îmbogăţi synset-ul dat cu noi cuvinte, au fost alese adjectivele care intervin pe prima poziţie în cadrul synset-urilor legate semantic de cel iniţial prin intermediul relaţiei de similaritate. Aceste cuvinte au fost adăugate synset-ului iniţial începând cu cea de-a doua poziţie. Această idee a fost inspirată de modul în care sunt organizate şi structurate în WN cluster-ele de adjective (vezi Anexa 1). În acest punct se simte necesitatea de a combina metoda de clasă prezentată cu una structurală (adică una care profită de structura WN).

Lista necesară a e-mulţimilor legate de synset-ul englezesc dat va fi generată folosind Algoritmul 5.1. Atunci când se creează synset-ul de adjective străin reprezentând traducerea celui englezesc dat, Algoritmul 5.2 va combina toate e-mulţimile evaluate maximal corespunzător fiecăruia dintre cuvintele care intervin în synset-ul englezesc. În acele cazuri în care mai multe e-mulţimi corespunzătoare aceluiaşi cuvânt englezesc vor fi evaluate maximal, Algoritmul 5.2 va genera mai multe synset-uri străine. Decizia finală cu privire la traducerea corectă se va lua din nou în conformitate cu glosa.

- *synset_id* este identificatorul synset-ului;
- *nr_cuv* este numărul cuvântului în cadrul synset-ului (comunică al câtelea cuvânt este 'cuvânt' în cadrul synset-ului);
- *'cuvant'* este textul ASCII al cuvântului la care se referă fapta Prolog;
- *tip_ss* este un cod de un caracter ce indică tipul synset-ului, cod care specifică partea de vorbire a lui 'cuvânt' (spre exemplu, **n** pentru substantiv, de la engl. "noun");
- *nr_sens* este un număr natural care desemnează numărul sensului;
- *stare_etic* poate fi 1 sau 0, coduri având următoarea semnificaţie:
  1 - sensurile cuvântului sunt ordonate după frecvenţa utilizării;
  0 - sensurile nu sunt ordonate dupa frecvenţa utilizării, ci conform altor criterii.

Un operator de tip **s** este prezent pentru fiecare sens din WN.

Un exemplu de o asemenea faptă Prolog, corespunzătoare adjectivului englezesc *abridged*, este următorul: s(300004481,1,'abridged',a,1,0).

Pentru a ilustra modul în care lucrează Algoritmii 5.1 şi 5.2 în cazul synset-urilor de adjective, vom lua în consideraţie synset-ul englezesc având *id*-ul 302428719 şi conţinând unicul adjectiv *sticky*. Vom realiza traducerea în limba română a acestui synset. Să observăm, mai întâi, că limba ţintă aleasă nu este esenţială pentru ceea ce dorim să demonstrăm aici. Rezultatele comentate reprezintă output-ul diverselor programe Prolog care implementează algoritmii amintiţi.

Întrucât synset-ul englezesc dat conţine un singur cuvânt, el va fi îmbogăţit, aşa cum s-a menţionat, în conformitate cu relaţia de similaritate. După căutarea în baza de date se ajunge la concluzia că unica relaţie de similaritate (desemnată prin operatorul **sim**) este

sim(302428719, 302425348).

precum şi simetrica ei. Synset-ul având *id* = 302425348, care este considerat similar ca sens cu cel aflat sub investigaţie, conţine unicul adjectiv *wet*. Synset-ul englezesc dat este, prin urmare, îmbogăţit cu acest adjectiv. E-mulţimile evaluate obţinute corespunzător synset-ului îmbogăţit, atunci când se foloseşte pentru Algoritmul 5.1 funcţia de evaluare menţionată în §5.2, sunt următoarele:

```
evset (302428719, sticky, 1.0, [lipicios, cleios, vascos]).
evset (302428719, sticky, 1.0, [umed, cetos]).
evset (302428719, wet, 1.0, [umed, jilav, ud]).
evset (302428719, wet, 0.6666666666666666, [ploios, umed,
igrasios]).
```

Aici *evset* este un operator care desemnează e-mulţimi evaluate. Primul câmp reprezintă *id*-ul synset-ului, cel de-al doilea este textul ASCII al cuvântului, aşa cum este el introdus de către lexicograf, cel de-al treilea furnizează valoarea funcţiei de evaluare, iar ultimul desemnează mulţimea evaluată străină.

În acest caz, programul care implementează Algoritmul 5.2 are următorul output:

```
Synset englezesc: [sticky]
Glosa:(moist as with undried perspiration and with clothing
sticking to the body; "felt sticky and chilly at the same
time")
Synset romanesc: [[lipicios,cleios,vascos,umed,jilav,ud],
[umed,cetos,jilav,ud]]
```

Se observă că au fost generate două synset-uri româneşti posibile. Cu toate acestea, doar unul dintre ele corespunde sensului lui *sticky* care se referă la conceptul aflat la baza synset-ului având *id* = 302428719. Synset-ul străin (în

acest caz românesc) corect poate fi ales cu uşurinţă în conformitate cu glosa corespunzătoare.

O astfel de îmbogăţire cu cuvinte suplimentare, provenite din synset-uri înrudite prin relaţia de similaritate cu cel iniţial, nu este întotdeauna necesară. Atunci când ea este totuşi efectuată, posibilitatea obţinerii unor synset-uri străine vide (datorată generării în exclusivitate a e-mulţimilor neetichetate) este considerabil redusă. Operaţia de îmbogăţire poate produce o uşoară deplasare a sensului, în ceea ce priveşte conceptul aflat la baza synset-ului englezesc iniţial. Această relaţie, tipică pentru adjectivele descriptive, se referă însă numai la concepte similare, fapt care ne determină să recomandăm utilizarea strategiei descrise. Se pot realiza atât traducerea cu îmbogăţire, cât şi cea fără îmbogăţire, relativ la un acelaşi synset (a se vedea exemplele care urmează), oferind lingviştilor posibilitatea de a compara şi de a alege între synset-urile străine propuse, atunci când se ia în cosideraţie, în egală măsură, şi glosa corespunzătoare.

Alte exemple

Eşantion cu synset-uri de adjective româneşti generate automat (fără îmbogăţire)

Synset englezesc: {clear}

Glosa: (meteorology) free from clouds or mist or haze; "on a clear day"

e-multimi:

cuvante	e-multime	scor
clear	{clar, curat, luminos, limpede, senin}	1.0
clear	{limpede, lamurit, inteligibil, clar, deslusit}	1.0
clear	{clar, perceptibil, lamurit, limpede, deslusit}	0.8
clear	{liber, deschis}	0.5
clear	{clar, patrunzator}	0.5
clear	{curat, neincarcat, negrevat, intreg}	0.25

Synset romanesc propus:

- {clar, curat, luminos, limpede, senin}
- {limpede, lamurit, inteligibil, clar, deslusit}

Synset englezesc: {fair}

Glosa: free of clouds or rain; "today will be fair and warm"

e-multimi:

cuvante	e-multime	scor
fair	{bun, frumos}	1.0
fair	{ieftin}	1.0
fair	{balan, balai, blond, deschis}	0.75
fair	{frumos, curat, ingrijit}	0.6666667
fair	{drept, nepartinitor, impartial}	0.6666667
fair	{bun, frumos, placut, prielnic, favorabil}	0.6
fair	{cinstit, onest}	0.5
fair	{cinstit, deschis}	0.5
fair	{convenabil, acceptabil, accesibil, rezonabil}	0.5
fair	{bun, natural, firesc}	0.33333334
fair	{frumos, minunat, atragator, dragut}	0.25

Synset romanesc propus:

- {bun, frumos}
- {ieftin}

Eşantion cu synset-uri de adjective româneşti generate automat (cu îmbogăţire)

Synset englezesc: {clear}

Glosa: (meteorology) free from clouds or mist or haze; "on a clear day"

e-multimi:

cuvante	e-multime	scor
fair	{limpede}	8.0
bright	{senin}	6.0
fair	{senin}	6.0
serene	{clar, senin, limpede}	5.6666665
cloudless	{senin, clar}	5.5
clear	{clar, curat, luminos, limpede, senin}	5.0
bright	{luminos}	5.0
bright	{clar, limpede, transparent}	4.0
clear	{limpede, lamurit, inteligibil, clar, deslusit}	3.0
fair	{citet, clar}	3.0
clear	{clar, perceptibil, lamurit, limpede, deslusit}	2.8
clear	{clar, patrunzator}	2.5
bright	{clar, sonor, cristalin}	1.6666666
serene	{calm, senin, linistit, potolit, netulburat}	1.4
fair	{frumos, curat, ingrijit}	1.3333334
bright	{spiritual}	1.0
fair	{bun, frumos}	1.0
fair	{ieftin}	1.0
serene	{linistit, calm}	1.0
clear	{curat, neincarcat, negrevat, intreg}	0.75
fair	{balan, balai, blond, deschis}	0.75
fair	{drept, nepartinitor, impartial}	0.6666667
fair	{bun, frumos, placut, prielnic, favorabil}	0.6
clear	{liber, deschis}	0.5
fair	{cinstit, onest}	0.5
fair	{cinstit, deschis}	0.5
fair	{convenabil, acceptabil, accesibil, rezonabil}	0.5
bright	{inteligent, ager, destept, sclipitor, scanteietor}	0.4
bright	{stralucitor, scanteietor, sclipitor}	0.33333334
fair	{bun, natural, firesc}	0.33333334
fair	{frumos, minunat, atragator, dragut}	0.25

Synset romanesc propus:

- {clar, curat, luminos, limpede, senin}

Synset englezesc: {fair}

Glosa: free of clouds or rain; "today will be fair and warm"

e-multimi:

cuvante	e-multime	scor
fair	{senin}	2.0
fair	{limpede}	2.0
clear	{clar, curat, luminos, limpede, senin}	1.4
fair	{frumos, curat, ingrijit}	1.3333334
fair	{citet, clar}	1.0
fair	{bun, frumos}	1.0
fair	{ieftin}	1.0
clear	{limpede, lamurit, inteligibil, clar, deslusit}	1.0
clear	{clar, perceptibil, lamurit, limpede, deslusit}	0.8
fair	{balan, balai, blond, deschis}	0.75
clear	{curat, neincarcat, negrevat, intreg}	0.75
fair	{drept, nepartinitor, impartial}	0.6666667
fair	{bun, frumos, placut, prielnic, favorabil}	0.6
fair	{cinstit, onest}	0.5
fair	{cinstit, deschis}	0.5
fair	{convenabil, acceptabil, accesibil, rezonabil}	0.5
clear	{liber, deschis}	0.5
clear	{clar, patrunzator}	0.5
fair	{bun, natural, firesc}	0.33333334
fair	{frumos, minunat, atragator, dragut}	0.25

Synset romanesc propus:

- {senin, clar, curat, luminos, limpede}

Atunci când se studiază exemplele de mai sus, se observă urmatoarele: în cazul synset-ului **{clear}**, care se referă la sensul provenind din meteorologie, dat de glosa menţionată, traducerea fără îmbogăţire conduce la două synset-uri româneşti, corespunzând unor sensuri complet diferite, dintre care primul reprezintă rezultatul corect (conform glosei). Efectuarea aceleiaşi traduceri cu îmbogăţire conduce direct la rezultatul corect unic. În cazul synset-ului **{fair}**, referindu-se, de asemenea, la sensul provenind din meteorologie, traducerea fără îmbogăţire produce două synset-uri româneşti, care, din nou, corespund unor sensuri complet diferite, nici unul dintre ele reprezentând un rezultat corect. Atunci când se foloseşte tehnica îmbogăţirii însă, este generat un unic synset românesc corect.

Efectuarea traducerii automate cu îmbogăţire nu va rezolva totuşi, mereu, toate problemele, aşa cum ar putea sugera exemplele anterioare, iar validarea lingvistică a output-ului obţinut va fi întotdeauna necesară, aşa cum se poate vedea în exemplul următor, în care primul synset românesc generat este cel corect:

Synset englezesc: {serene}

Glosa: completely clear and fine; "serene skies and a bright blue sea"

e-multimi:

cuvante	e-multime	scor
serene	{clar, senin, limpede}	1.0
serene	{liniştit, calm}	1.0
serene	{calm, senin, liniştit, potolit, netulburat}	0.6

Synset romanesc propus:

- {clar, senin, limpede}
- {liniştit, calm}

(În acest caz particular, prin folosirea dicționarelor bilingve existente, a reieșit că traducerea cu și fără îmbogățire conduc la același rezultat).

5.4.2. Generarea semiautomată a cluster-elor de adjective

Traducerea cluster-elor de adjective englezești este complet asigurată de către traducerea synset-urilor englezești de adjective și cea a relației **ant** (care desemnează antonime). Întrucât traducerea synset-urilor de adjective a fost deja discutată, ne vom referi, în cele ce urmează, la traducerea relației **ant**. Aceasta devine o problemă foarte importantă, atunci când se ia în considerație faptul că, pentru un mare număr de limbi, dicționarele de antonime în format electronic sunt inexistente.

În versiunea Prolog a bazei de date WN, care s-a folosit aici, relațiile semantice sunt reprezentate printr-o pereche de *synset_id*-uri, în care primul *id* este, în general, sursa relației, iar cel de-al doilea reprezintă ținta, ca în cazul operatorului deja menționat **sim**. Dacă două perechi *synset_id*, *w_num* sunt prezente, atunci operatorul reprezintă o relație lexicală între forme lexicale, în care *w_num* specifică numărul cuvântului, pentru un anumit cuvânt dintr-un anumit synset. Dacă este prezent, *w_num* indică la care dintre cuvintele synset-ului se face referire. Operatorul **ant**, de pildă, specifică cuvinte antonime în următoarea formă:

ant(*synset_id,w_num,synset_id,w_num*).

Astfel, semnificația faptei Prolog

ant(302425348, 1, 302429323, 1).

este aceea că primul cuvânt al synset-ului având *id*-ul 302425348 și primul cuvânt al synset-ului având *id*-ul 302429323 sunt *antonime directe*. Aceasta este o relație lexicală care este valabilă pentru aproape toate părțile de vorbire, dar care este esențială în formarea cluster-elor de adjective (vezi Anexa 1). În cazul fiecărei perechi de antonime sunt date ambele relații (i.e. fiecare pereche *synset_id,w_num* desemnează atât un cuvânt sursă, cât și unul țintă).

Atunci când este studiat conținutul fișierului **wn_ant.pl** al bazei de date WN în format Prolog, fișier care conține toate faptele Prolog referitoare la cuvinte antonime, se observă cu ușurință că marea majoritate a acestor fapte stabilesc relații de antonimie directă între cuvinte aflate pe prima poziție în cadrul synset-urilor cărora le aparțin. Există mai puțin de 15 excepții de la această regulă. Aceste excepții pot fi cu ușurință procesate de către un operator uman care poate reține noile poziții ale adjectivelor având antonime directe. În aceste condiții, considerăm justificat să formulăm

Observația 5.2
Primul cuvânt al unui synset englezesc de adjective este cel care poate avea un antonim direct.

Să presupunem acum că toate synset-urile de adjective străine (traduse) există, corespunzător unei limbi date și că ele aparțin unui fișier numit **wn_strans.pl**. Folosind Observația 5.2 și în urma generării acestui fișier prin aplicarea algoritmului de traducere, putem formula algoritmul de generare a cluster-elor de adjective străine corespunzătoare celor englezești:

Algoritmul 5.3

Input: Fișierele **wn_ant.pl**, **wn_sim.pl** și **wn_strans.pl**

Pentru fiecare *pereche de synset-uri* desemnată de către fiecare faptă Prolog a fișierului **wn_ant.pl**, execută pașii 1.- 6.:

1. Caută în fișierul **wn_strans.pl** și găsește synset-urile străine reprezentând traducerile celor englezești date.

2. Corespunzător fiecărui synset străin găsit în fișierul **wn_strans.pl** la pasul 1, reține primul cuvânt al acelui synset. (Această pereche de cuvinte va constitui capul cluster-ului străin obținut prin traducere).

3. Corespunzător aceleiași perechi de cuvinte, caută în fișierul **wn_sim.pl** și ia în considerație clauzele **sim** corespunzătoare fiecăruia dintre cele două synset-uri cărora le aparțin cele două cuvinte ale capului de cluster.

4. Ia în considerație toate synset-urile desemnate de clauzele **sim** alese la pasul 3, synset-uri având cel de-al doilea *id* care intervine în clauză. Găsește synset-urile străine reprezentând traducerile acestora în fișierul **wn_strans.pl**.

5. Adaugă fiecare prim cuvânt al acestor synset-uri străine în capul de cluster, împreună cu pointerul &.

6. Adaugă fiecare synset străin "similar", plasând la sfârșitul acestuia pointerul reciproc de similaritate.

Output: Un fișier conținând toate cluster-ele de adjective străine.

□

Algoritmul 5.3 va genera clustere-e străine de adjective cu o structură bipolară ca cea descrisă în [22], un exemplu putând fi văzut în Anexa 1.

Cluster-ul românesc pe care l-am obţinut nu ilustrează decât procedeele de codificare *de bază* folosite în WN-ul american, care au fost preluate de către noi. În acest stadiu de început al studiului nostru, am fost preocupaţi numai de crearea structurii de cluster *de tip WordNet* şi nu am încercat să facem distincţia între diferitele sensuri secundare sau privilegii de apariţie. Nu am încercat, de asemenea, să indicăm limitările anumitor adjective relativ la poziţiile sintactice pe care acestea pot să le ocupe, un tip de limitare care în WN este codificat relativ la adjective individuale. Aceste aspecte de natură lingvistică, ca şi altele, de aceeaşi natură, vor fi tratate după stabilirea algoritmului de bază şi vor face obiectul unor studii viitoare.

Diverse dificultăţi de natură lingvistică vor fi întâmpinate, în mod evident, în funcţie de limba ţintă aleasă. Synset-uri străine identice ar putea fi generate, spre exemplu, de către Algoritmul 5.3, corespunzător unor synset-uri englezeşti diferite şi, prin urmare, corespunzător unor sensuri şi concepte diferite. Aceasta este situaţia atunci când un adjectiv polisemantic englezesc va avea în engleză unul sau mai multe sensuri care în limba ţintă nu există, fenomen lingvistic care este numit *împrumut semantic* şi care este un aspect al *calcului* în general. Validarea lingvistică a output-ului programelor care implementează Algoritmul 5.3, sau orice alt algoritm de aceeaşi natură, va fi, prin urmare, întotdeauna necesară. Cu toate acestea, considerăm că Algoritmul 5.3 acoperă marea majoritate a cazurilor existente, atunci când se lucrează cu cluster-e de adjective de tip WN.

5.4.3. Consideraţii finale privitoare la adjective

Abordarea propusă referitoare la generarea semiautomată a reţelelor semantice de tip WN reprezintă o combinaţie de metode automate şi manuale. Metoda manuală se bazează pe experţi umani, în timp ce metoda de clasă automată se sprijină în mod unic pe dicţionare bilingve.

Folosirea metodei de clasă propuse (care este independentă de limbă şi care nu ţine seama de partea de vorbire supusă analizei) este suficientă pentru a genera în mod automat synset-urile limbii ţintă (care vor fi validate în mod manual). În cazul adjectivelor însă, trebuie să fim preocupaţi nu numai de traducerea automată a synset-urilor de adjective englezeşti, ci şi de crearea structurii tipice de cluster de adjective, corespunzător limbii ţintă. Pentru realizarea acestui deziderat, trebuie luată în consideraţie structura WN, fapt care denotă necesitatea combinării metodelor de clasă cu cele structurale. Generarea cluster-elor de adjective poate fi realizată în întregime automat (folosind Algoritmul 5.3), în condiţiile în care traducerea synset-urilor de adjective care intervin, efectuată prin intermediul metodei de clasă propuse, a fost validată de către experţi umani.

Semnificaţia efortului manual necesar în asigurarea calităţii traducerii este dată, în primul rând, de existenţa instrumentelor adecvate. Efortul uman implicat poate fi mult redus în cazul acelor limbi pentru care există dicţionare bilingve corecte şi complete, în format electronic.

5.5. Alte comentarii lingvistice privitoare la rezultatele obţinute

Dorim să menţionăm, încă de la început, faptul că, atunci cand rezultatele obţinute de către programele de calculator care implementează algoritmii referitori la WN nu sunt cele mai bune cu putinţă, acest lucru se datorează, în primul rând, imperfecţiunii dicţionarelor bilingve existente. În cele ce urmează, ne vom referi, cu precădere, la tipurile de erori şi la dificultăţile întâmpinate în cazul limbii române, şi anume în generarea automată a *synset-urilor românesti de substantive*.

În [15], lucrarea la care ne vom raporta, în continuare, sunt considerate ca fiind interesante trei tipuri de situaţii: "cele în care programul a generat mai multe synset-uri româneşti dintre care unul este corect, cele în care nu a generat nici un synset românesc şi cele, mult mai rare, în care a generat unul sau mai multe synset-uri, dar care sunt greşite".

În primul caz, alegerea traducerii corecte poate fi uşor făcută de către lingvişti (în conformitate cu glosa corespunzătoare) în timp ce în al doilea sunt, în general, de vină dicţionarele bilingve. Aşa cum se arată în [15], "considerăm că mai interesante au fost situaţiile în care, prin program, nu a fost generat nici un synset românesc. Cel mai adesea, cauza o constituie imperfecţiunea dicţionarelor bilingve, care nu includ cuvintele respective. Alteori doar unul dintre dicţionare este de vină, cel mai adesea fiind vorba de dicţionarul român-englez, relativ sărac în privinţa numărului cuvintelor-titlu, dar şi în privinţa celor englezeşti luate în consideraţie. În multe dintre situaţiile datorate acestui fapt, prin algoritmul propus, se vor obţine numai mulţimi elementare neetichetate. În acest caz, nu se generează, conform algoritmului, nici un synset românesc". Dintre situaţiile în care un cuvânt românesc care există în dicţionarul englez-român nu se regăseşte în cel român-englez, este subliniată, în mod special, urmatoarea: "este, în special, cazul substantivelor provenite din verbe şi care au semnificaţia <<acţiunea de a...>>. Cuvinte importante şi uzuale în limba română, cum ar fi **organizare** (de la <<a organiza>> - <<to organize>>) sau **respingere** (de la <<a respinge>> - <<to reject>>), apar ca traduceri ale unor cuvinte englezeşti, dar nu se regăsesc în dicţionarul român-englez. Acest lucru poate duce la eşecul algoritmului de evaluare a e-mulţimilor, întrucât neregăsirea unui cuvânt în dicţionarul român-englez aduce cu sine o valoare mai mică a acelei e-mulţimi".

Aşa cum remarcă acelaşi autor [15], "uneori synset-ul românesc generat de program va fi incorect format datorită funcţiei de evaluare implementate. Noi funcţii de evaluare ar trebui implementate şi testate în viitor".

Cele mai interesante cazuri rămân însă acelea în care dicţionarele bilingve, în general şi cel român-englez, în particular, nu sunt de vină: "În acele situaţii în care dicţionarul român-englez nu este de vină, cauza erorilor pe care le face programul este de cu totul altă natură şi trebuie căutată în sfera conceptelor. Trebuie avut în vedere faptul că limba engleză şi, în mod special, engleza americană, la care se referă WN, este o limbă incomparabil mai bogată decât limba română. Statistic vorbind, în timp ce româna are maximum 150.000 de cuvinte, engleza americană are aproximativ 450.000 cuvinte (judecând după informaţiile furnizate de către lexicograful St. Berg Flexner). Dar, comparativ cu limba română, engleza este o limbă mult mai evoluată nu numai din punct de vedere gramatical şi lexical (adică sub raport strict cantitativ), ceea ce însemnează mai multe cuvinte sau unităţi lexicale. Ea este totodată mult mai evoluată şi sub raport strict semantic, acelaşi cuvânt englezesc având adeseori un conţinut semantic mult mai bogat decât cuvântul românesc corespunzător. Numeroase cuvinte existente atât în română, cât şi în engleză, sunt mai polisemantice în engleză decât în română. Cu alte cuvinte, polisemia cuvintelor englezeşti este superioară polisemiei cuvintelor româneşti. Spre exemplu, cuvântului englezesc **feature** cu sensul de <<an article of merchandise that is displayed or advertised more than other articles>> nu îi corespunde în limba română un cuvânt românesc cu acelaşi sens. Suntem deci obligaţi să recurgem la traducere printr-un grup de cuvinte (o glosă), iar synset-ului englezesc format din singurul cuvânt **feature** care se referă la acest concept nu îi corespunde un synset românesc. În acest caz, programul a lucrat greşit. Este, din nou, o situaţie care afectează în special synset-urile englezeşti alcătuite dintr-un singur cuvânt. Un alt exemplu de cuvânt polisemantic englezesc este **foundation**, care ne atrage atenţia prin unul dintre sensurile sale: <<a woman's undergarment worn to give shape to the contours of the body>>. Acest sens al lui **foundation** nu există în română, iar conceptul la care se referă synset-ul conţinând unicul cuvânt **foundation** cu acest sens trebuie explicat în limba română prin intermediul unei glose. Lui nu trebuie să îi corespundă nici un synset românesc. Programul a lucrat din nou greşit în această situaţie, ca şi în cazul englezescului **quiver** cu sensul <<a case for holding arrows>>".

O altă situaţie în care programele nu au lucrat corect este, conform [15], aceea în care anumite substantive englezeşti sunt folosite cu negaţie, "cum ar fi **matter** cu negaţie, ca în exemplul <<they were friends and it was no matter who won the game>>. Astfel de substantive se traduc în limba română printr-o colocaţie al cărei centru îl constituie un substantiv care nu figurează în dicţionarul englez-român printre posibilele traduceri ale lui **matter** sau care figurează în dicţionar printr-un echivalent de tip locuţional, care nu va fi folosit

de algoritmul pe care îl implementează programul. În acest caz programul nu poate găsi synset-ul românesc (sau, în general, străin) corect. În particular, în cazul lui <<matter>> folosit cu negaţie, au fost găsite câteva synset-uri româneşti posibile, dar nici unul nu este corect deoarece nu conţine substantivul **importanţă** (importance), care intră în colocaţia românească corespunzătoare acestui sens. Expresia românească corespunzătoare este un calc după franţuzescul <<avoir de l'importance>>. Cazurile de calc după franceză sunt foarte frecvente în limba română. Iată de ce simţim nevoia ca programele ulterioare să ia în consideraţie colocaţiile, atât în limba engleză, cât şi în limba ţintă (în particular, româna)".

Aşa cum notează acelaşi lingvist [15], "o altă sursă de dificultăţi o constituie substantivele la plural. Astfel, există în synset-urile englezeşti substantive la singular care se traduc prin substantive româneşti la plural. Exemple din această categorie sunt **foundation** tradus prin **fonduri** sau **knowledge** tradus prin **cunoştinţe**". Acest comentariu ne-a determinat să includem formele de plural ale unor asemenea substantive în varianta electronică a dicţionarului român-englez care a constituit input al programului implementând Algoritmul 5.1.

Relaţia importantă dintre omonimie şi polisemie este, de asemenea, luată în consideraţie [15]: "În limba română, ca şi în alte limbi, cum ar fi de exemplu franceza, relaţia dintre omonimie şi polisemie este o problemă extrem de complicată, asupra căreia specialiştii nu au căzut încă de acord. Ea nu constituie o problemă rezolvată. În foarte multe cazuri, după unii cercetători avem de-a face cu două, trei sau chiar mai multe cuvinte omonime, iar după alţii cu un singur cuvânt polisemic, care are două, trei sau mai multe sensuri fundamentale, mai mult sau mai puţin înrudite. Un exemplu ar fi cuvântul **bun** (**good**), care în limba română este în primul rând adjectiv, având şapte sensuri fundamentale. În al doilea rând este substantiv cu două pluraluri diferite, dar specializate semantic. Substantivul **bun** cu pluralul **bunuri** are patru sensuri, iar substantivul **bun** cu pluralul **buni** are un singur sens, acela de bunic. Astfel de situaţii sunt suficient de frecvente în limba română". Pentru ca programele de calculator care implementează algoritmii descrişi să producă rezultate mai bune, autorul citat recomandă folosirea dicţionarelor "care tratează posibilele omonime, în special omonimele aşa-zise semantice, ca pe un singur cuvânt polisemantic".

Deşi autorul menţionat [15] este preocupat, în primul rând, de analiza greşelilor care intervin în traducerea automată, el ajunge la următoarea concluzie: "atunci când dicţionarele bilingve utilizate au fost corecte şi complete, algoritmul implementat s-a dovedit a fi extrem de performant. Astfel, în cazul unor concepte foarte apropiate în limba engleză, diferenţa fină a fost sesizată de program, care o menţine în mod corect în traducerea românească. Este, de pildă, cazul synset-urilor englezeşti **[banishment, proscription]** cu

sensul <<the act of banishing someone>> şi respectiv **[ostracism]** cu sensul <<the act of excluding someone from society by general consent>>. Primul a fost tradus în limba română prin **[exilare, surghiunire, exil, surghiun, expulzare, ostracizare]**, în timp ce al doilea a fost tradus prin unicul **[ostracism]**. Românescul **ostracism** este singurul care se referă la consens în luarea deciziei de a ostraciza şi, prin folosirea lui în cazul celui de-al doilea synset, este pusă în evidenţă diferenţa dintre cele două concepte diferite, dar înrudite, la care se referă synset-urile englezeşti iniţiale".

Acelaşi lingvist [15] încheie observând că principalele dificultăţi care apar atunci când se efectuează traducerea automată a synset-urilor englezeşti în limba română sunt generate "de colocaţii, de calcul lingvistic şi de superioritatea polisemiei unor cuvinte englezeşti în raport cu corespondentele lor româneşti". În plus, se face observaţia că "cele mai multe probleme au apărut acolo unde synset-ul englezesc era compus dintr-un singur cuvânt, algoritmul neputând decide între sensuri. În viitor se impune, probabil, o tratare diferită a acestor synset-uri. În acelaşi timp, trebuie să accentuăm faptul că lipsa unor instrumente de lucru foarte performante (cu referire la dicţionare) nu poate decât să îngreuneze o reală evaluare a programelor folosite în traducere".

5.6. Revizitarea synset-urilor de substantive

Având ca punct de plecare remarca lingvistului [15] care arată că "cele mai multe probleme au apărut acolo unde synset-ul englezesc era compus dintr-un singur cuvânt, algoritmul neputând decide între sensuri", revizităm, în cele ce urmează, synset-urile englezeşti conţinând **un unic substantiv polisemantic**.

Devine din nou evident faptul că, pentru a face distincţia corectă între sensurile existente, atunci când se efectuează traducerea automată a acestui tip de synset-uri englezeşti, algoritmul propus ar trebui să ţină cont, în egală măsură, de structura reţelei WN. În cazul synset-urilor de substantive însă, relaţia de similaritate, care a fost utilizată relativ la synset-urile de adjective, nu există. Aceasta ne determină să luăm în consideraţie un tip de relaţie complet diferit, şi anume relaţia de **hiperonimie**, aşa cum este sugerat în [29]. Hiperonimia este una dintre relaţiile semantice de bază implementate în WN, care corespunde relaţiei **isa** şi prin intermediul căreia substantivele sunt structurate sub formă de ierarhii. Atunci când am tradus automat synset-uri conţinând un unic substantiv polisemantic ne-am îndreptat, prin urmare, atenţia către hiperonimul synset-ului dat, hiperonim care indică "conceptul părinte" al celui la care se referă synset-ul aflat sub investigaţie.

În versiunea Prolog a bazei de date WN, pe care am folosit-o aici, relaţia de hiperonimie este exprimată sub urmatoarea formă:

hyp(*synset_id, synset_id*).

Operatorul **hyp** specifică faptul că cel de-al doilea synset este un hiperonim al primului synset (sau că acesta din urmă reprezintă un hiponim al celui de-al doilea). Această relaţie este valabilă în cazul substantivelor şi al verbelor.

S-a încercat să se estimeze [29] în ce masură îmbogăţirea synset-urilor, realizată prin intermediul relaţiei de hiperonimie, măreşte şansele de a fi traduse corect synset-urile englezeşti conţinând un unic substantiv polisemantic. Numărul total de asemenea synset-uri existente în versiunea 1.7.1 a reţelei WN, care a fost folosită în toate experimentele, este 13448. Luând în consideraţie acest număr şi aceste synset-uri ca input, a fost generată o selecţie aleatoare, după cum urmează[29]:

1. A fost ales în mod aleator (utilizând funcţia Rand( ) care este implementată în Perl 5.8.0) un număr de 135 synset-uri conţinând un singur substantiv polisemantic. Synset-urile selectate reprezintă 1/100 din input-ul total.

2. Au fost adăugate acestui eşantion de date toate celelalte synset-uri având acelaşi conţinut ca cele generate la pasul anterior. (***Exemplu***: Dacă, la pasul 1, a fost selectat synset-ul **[bearing]** având *id*-ul 102450394, atunci, la pasul 2, vor fi incluse în eşantionul de date toate synset-urile conţinând unicul substantiv polisemantic *bearing*. În cazul de faţă există un singur asemenea synset, şi anume cel având *id*-ul 111640712). După execuţia pasului al doilea al acestei simulări, a fost obţinut un număr de 257 synset-uri de substantive, care vor fi utilizate în estimare şi care reprezintă 1.91% din input-ul iniţial.

3. Fiecare dintre cele 257 synset-uri de substantive obţinute la pasul 2 a fost îmbogăţit prin adăugarea tuturor substantivelor primului synset hiperonim[22].

Observaţia 5.3

Alegerea "primului hiperonim" ca fiind cel mai semnificativ spre a fi folosit pentru efectuarea îmbogăţirii a fost facută în conformitate cu următoarea simulare: există un număr de 199 synset-uri conţinând un unic substantiv polisemantic şi având hiperonime multiple. (Acestea reprezintă numai 1.4% din numărul total al synset-urilor aflate sub investigaţie). Au fost alese în mod aleator 20 de astfel de synset-uri (reprezentând aproximativ 10% dintre synset-urile care au hiperonime multiple) şi s-a ajuns la concluzia că, în 17 cazuri din 20 (cu alte cuvinte

[22] Aici "primul synset hiperonim" se referă la ordonarea existentă între *synset_id* - uri.

în 85% din totalul cazurilor) se poate reţine numai primul hiperonim, care poate fi considerat ca fiind cel mai reprezentativ.

Algoritmul 5.1 a fost utilizat pentru efectuarea traducerii automate, în limba română, a tuturor synset-urilor conţinând un unic substantiv polisemantic, precum şi a celor 257 de synset-uri îmbogăţite, obţinute aşa cum a fost descris anterior. În primul caz (synset-uri neîmbogăţite) algoritmul nu a putut furniza o traducere corectă în 60.75% dintre situaţii. Atunci când s-a operat cu synset-uri îmbogăţite, acelaşi algoritm a eşuat în numai 31% dintre cazuri (conform [29]). Aceasta ne arată că tehnica de îmbogăţire propusă micşorează eşecul în generarea automată a synset-urilor conţinând un unic substantiv polisemantic cu aproximativ 50%, rezultat care ne încurajează să reformulăm Algoritmul 5.1, corespunzător acestei părţi de vorbire, după cum urmează [29]:

Algoritmul 5.4

Input: Synset-ul englezesc care trebuie tradus, fişierul **wn_s.pl** (unde un operator **s** este prezent corespunzător fiecărui sens din WN), fişierul corespunzând operatorului **hyp** şi cele două fişiere reprezentând cele două dicţionare bilingve.

1. Atunci când synset-ul englezesc dat conţine un singur substantiv, află (prin consultarea fişierului **wn_s.pl**) dacă acest substantiv este unul polisemantic. Dacă nu, STOP şi foloseşte Algoritmul 5.1 pentru efectuarea traducerii.

2. Utilizează fişierul corespunzător operatorului **hyp** pentru a găsi primul hiperonim al synset-ului dat. Dacă un asemenea hiperonim există[23], execută:

   2.1. Îmbogăţeşte synset-ul englezesc dat şi care conţine un singur substantiv polisemantic cu toate substantivele synset-ului reprezentând primul său hiperonim.

   2.2. Şterge una dintre apariţiile substantivului din synset-ul iniţial dacă acest cuvânt intervine şi în synset-ul folosit pentru realizarea îmbogăţirii. Noul synset (îmbogăţit) rezultat este cel care va fi tradus.

3. Creează (prin consultarea dicţionarului bilingv adecvat) e-mulţimile corespunzătoare fiecărui cuvânt al synset-ului englezesc care trebuie tradus.

4. Etichetează cuvintele englezeşti aparţinând synset-ului englezesc de tradus.

5. Etichetează fiecare dintre e-mulţimile generate la pasul 2.

6. Înlătură toate e-mulţimile neetichetate.

7. Evaluează e-mulţimile (folosind etichetele atribuite şi o funcţie de evaluare).

8. Sortează (conform scorurilor obţinute şi cuvintelor englezeşti cărora le corespund), în ordine crescătoare sau descrescătoare, lista de e-mulţimi evaluate obţinută.

9. Alege e-mulţimea corespunzătoare unicului substantiv din synset-ul englezesc iniţial care are scorul cel mai mare şi prezint-o ca output. STOP.

Output: Synset-ul străin corespunzător synset-ului englezesc dat.

□

Observaţia 5.4
Întrucât synset-ul hiperonim a fost folosit numai pentru *a preciza sensul* substantivului polisemantic care intervine în synset-ul englezesc de tradus, nu va fi selectată nici o e-mulţime corespunzătoare substantivelor synset-ului hiperonim, atunci când se formează synset-ul străin reprezentând traducerea căutată. În acest caz, nu este, prin urmare, necesară folosirea, în continuare, a Algoritmului 5.2. Algoritmul 5.4 are ca output rezultatul final, cu alte cuvinte synset-ul străin reprezentând traducerea, în limba ţintă, a celui englezesc dat (a se vedea şi exemplul care urmează).

[23] În WN toate synset-urile de substantive au hiperonime. O excepţie o reprezintă doar synset-urile aflate la nivelul cel mai de sus. Printre acestea, există unul singur conţinând un unic substantiv polisemantic, şi anume synset-ul **[state]**, care are glosa (the way something is with respect to its main attributes; "the current state of knowledge"; "his state of health"; "in a weak financial state").

Un exemplu

Pentru a ilustra felul în care lucrează Algoritmul 5.4, ne vom propune să traducem în limba română cele trei synset-uri englezeşti conţinând unicul substantiv polisemantic **tiller**, care au *synset_id*-urile 103867107, 108769239 şi respectiv 111093313. În cele ce urmează, rezultatele menţionate constituie output-ul unui program Prolog, aşa cum sunt ele prezentate în [29].

În cazul synset-ului englezesc `[tiller]` având *id*-ul 103867107 şi glosa corespunzătoare *lever used to turn the rudder on a boat*, synset-ul îmbogăţit, rezultat la pasul 2.2 al Algoritmului 5.4, este `[tiller,lever]`, iar programul de calculator care implementează Algoritmul 5.4 are următorul output:

```
Synset englezesc: [tiller]
(lever used to turn the rudder on a boat)

sin(103867107,[tiller,lever]).
evset(103867107,tiller,1,[biela]).
evset(103867107,lever,0.75,[parghie,levier,maner,brat]).
evset(103867107,lever,1,[maner]).

evset(103867107,tiller,1,[biela]).

Synset romanesc: [[biela]]
```

E-mulţimea aleasă la pasul 9 al Algoritmului 5.4 în acest caz este, prin urmare, cea dată de clauza Prolog

`evset(103867107,tiller,1,[biela]).`

iar rezultatul final îl constituie synset-ul românesc `[biela]`, care reprezintă o traducere corectă.

În cadrul output-ului prezentat anterior, *evset* este un operator care desemnează e-mulţimi evaluate. Primul câmp reprezintă *id*-ul synset-ului, iar cel de-al doilea este textul ASCII al cuvântului, aşa cum a fost el introdus de către lexicograf. Numărul corespunzător celui de-al treilea câmp reprezintă valoarea funcţiei de evaluare în cazul fiecărei e-mulţimi obţinute la pasul 3 al Algoritmului 5.4. Ultimul câmp indică e-mulţimea evaluată străină (în acest caz românească). Predicatul Prolog *sin* are două argumente, primul reprezentând *id*-ul synset-ului care trebuie tradus, în timp ce al doilea constituie synset-ul îmbogăţit pe care îl traducem de fapt. Intrări de tipul *sin* există numai relativ la acele synset-uri care trebuie traduse şi care sunt considerate "complete" (în sensul că există în dicţionarul englez-român intrări corespunzătoare tuturor cuvintelor englezeşti care intervin).

În cazul synset-ului englezesc `[tiller]` având *id*-ul 108769239 şi glosa corespunzătoare *someone who tills land (prepares the soil for the planting of crops)*, synset-ul îmbogăţit rezultat la pasul 2.2 al Algoritmului 5.4 este

`[tiller, sodbuster, granger, husbandman, farmer]`

iar programul de calculator care implementează Algoritmul 5.4 produce acum următorul output:

```
Synset englezesc: [tiller]
(someone who tills land (prepares the soil for the planting
of crops))

sin(108769239,[tiller,sodbuster,granger,husbandman,
farmer]).
evset(108769239,tiller,2,[agricultor,plugar]).
evset(108769239,tiller,1,[biela]).
evset(108769239,sodbuster,4,[cultivator]).
evset(108769239,granger,1,[agricultor,gospodar]).
evset(108769239,husbandman,2,[cultivator,agricultor,
gospodar]).
evset(108769239,farmer,0.5,[fermier,arendas]).

evset(108769239,tiller,2,[agricultor,plugar]).
evset(108769239,tiller,1,[biela]).

Synset romanesc: [[agricultor,plugar]]
```

E-mulţimile care sunt acum generate corespunzător substantivului *tiller* aparţinând synset-ului englezesc iniţial sunt `[agricultor,plugar]` şi `[biela]`. E-mulţimea aleasă la pasul 9 al Algoritmului 5.4 este `[agricultor,plugar]`, care desemnează synset-ul românesc căutat şi care, din nou, reprezintă o traducere corectă a celui englezesc dat.

În fine, în cazul synset-ului englezesc `[tiller]` având *id*-ul 111093313 şi glosa corespunzătoare *a shoot that sprouts from the base of a grass*, synset-ul îmbogăţit rezultat la pasul 2.2 al Algoritmului 5.4 este `[tiller,shoot]` şi programul de calculator implementând Algoritmul 5.4 are următorul output:

```
Synset englezesc:[tiller]
(a shoot that sprouts from the base of a grass)

sin(111093313,[tiller,shoot]).
evset(111093313,tiller,1.33333,[vlastar,mladita,puiet]).
evset(111093313,tiller,1,[biela]).
evset(111093313,shoot,0.666667,[mladita,vlastar,mugurel]).
evset(111093313,shoot,0.5,[scoc,jgheab]).
```

```
evset(111093313,tiller,1.33333,[vlastar,mladita,puiet]).
evset(111093313,tiller,1,[biela]).

Synset romanesc: [[vlastar,mladita,puiet]]
```

E-mulțimile generate corespunzător substantivului *tiller* ale synset-ului englezesc inițial sunt `[vlastar, mladita, puiet]` și `[biela]`. La pasul 9 al Algoritmului 5.4 e-mulțimea `[vlastar,mladita,puiet]`, evaluată cu un scor mai mare, este aleasă și prezentată ca output. Aceasta reprezintă, din nou, o traducere corectă a synset-ului englezesc dat.

În [29] se prezintă și rezultatul traducerii fără îmbogățire a celor trei synset-uri englezești conținând unicul substantiv polisemantic *tiller* (adică al traducerii efectuate prin intermediul Algoritmilor 5.1 si 5.2). În toate cele trei cazuri rezultatul obținut a fost unicul synset românesc `[biela]`[24]. Algoritmul 5.1 nu a fost, prin urmare, capabil să distingă între sensurile diferite existente, iar rezultatul obținut pentru traducere nu poate fi corect decât în unul dintre cele trei cazuri studiate. Acesta este tipul de rezultate care pot fi obținute în cadrul de lucru descris aici și astfel de rezultate demonstrează, încă o dată, necesitatea de a combina metodele de clasă cu cele structurale, atunci când rețeaua semantică WN este utilizată pentru efectuarea traducerii automate.

5.6.1. Considerații finale privitoare la substantive

Ca și în cazul unor studii anterioare [24], [11], [12], pentru facilitarea experimentului, atunci când am efectuat teste în cazul limbii române, nu am luat în considerație synset-uri conținând nume proprii, substantive compuse și colocații. Așa cum se remarcă în [15], acestea ar trebui tratate în mod separat și cu o contribuție mai semnificativă din partea lingvistului.

Pentru a se minimiza eșecul în traducerea automată a synset-urilor englezești constând dintr-un singur substantiv polisemantic, precum și pentru a se micșora efortul uman implicat, a fost inclus [29] un "pas de îmbogățire" în Algoritmul 2.1 de traducere existent. Aceasta îmbunătățește traducerea automată a unor asemenea synset-uri de substantive cu aproximativ 50%.

Metoda îmbunătățită prezentată aici întărește, din nou, ideea necesității combinării metodelor de clasă cu cele structurale, pentru realizarea unei sarcini de acest tip. În același timp, ea se bazează, în mod nemijlocit, pe existența unor dicționare bilingve comprehensive în format electronic.

---

[24] Corespunzător tuturor synset-urilor studiate, `[biela]` a reprezentat *singura e-mulțime etichetată* și, prin urmare, a constituit singura alegere posibilă.

5.7. Synset-uri de verbe

Discuția privind generarea semiautomată a synset-urilor de verbe în alte limbi decât engleza va extinde aceeași abordare algoritmică propusă în [24] și în [11], reiterând, totodată, necesitatea de a combina metodele de clasă cu cele structurale în construcția automată a unor asemenea baze de date lexicale.

Ori de câte ori synset-ul care trebuie tradus automat conține mai multe verbe, traducerea va fi efectuată prin intermediul Algoritmilor 5.1 si 5.2. O problemă specială se ridică, din nou, în cazul synset-urilor conținând *un unic verb polisemantic* întrucât, în astfel de situații, nu există nici o modalitate de a ști la care dintre sensurile cuvântului în cauză se referă synset-ul dat.

Algoritmul de traducere propus [12] va folosi aceleași concepte, precum și ***similaritatea*** în materie de sens a diferitelor synset-uri de verbe. In versiunea Prolog a bazei de date WN această similaritate este exprimată sub forma următoare:

vgp (*synset_id*, *synset_id*).

Operatorul **vgp** specifică synset-uri de verbe care sunt similare ca sens și care ar trebui asociate în cazul afișării ca eventual răspuns la o căutare de synset-uri grupate. Această relație este valabilă numai în cazul synset-urilor de verbe.

Algoritmul propus pentru generarea *e-mulțimilor străine* corespunzătoare unui synset englezesc dat care conține un singur verb este următorul:

Algoritmul 5.5

Input: Synset-ul englezesc care trebuie tradus, fișierul **wn_s.pl** (unde un operator **s** este prezent corespunzător fiecărui sens din WN), fișierul corespunzând operatorului **vgp** și cele două fișiere reprezentând cele două dicționare bilingve.

1. Află (prin consultarea fișierului **wn_s.pl**) dacă unicul verb al synset-ului de tradus este unul polisemantic. Dacă nu, STOP și folosește Algoritmul 5.1 pentru efectuarea traducerii.

2. Utilizează fișierul corespunzător operatorului **vgp** pentru a găsi un synset similar ca sens. Dacă un asemenea synset există, execută:

   2.1. Îmbogățește synset-ul englezesc dat, care include un unic verb polisemantic, cu un întreg synset care este similar ca sens.

   2.2. Șterge una dintre aparițiile verbului din synset-ul inițial dacă acest cuvânt intervine și în synset-ul folosit pentru

realizarea îmbogăţirii. Noul synset (îmbogăţit) rezultat este cel care va fi tradus.

3. Creează (prin consultarea dicţionarului bilingv adecvat) e-mulţimile corespunzătoare fiecărui cuvânt al synset-ului englezesc care trebuie tradus.

4. Etichetează cuvintele englezeşti aparţinând synset-ului englezesc de tradus.

5. Etichetează fiecare dintre e-mulţimile generate la pasul 2.

6. Înlătură toate e-mulţimile neetichetate.

7. Evaluează e-mulţimile (folosind etichetele atribuite şi o funcţie de evaluare).

8. Dacă synset-ul englezesc dat a fost îmbogăţit la pasul 2.1, atunci prelucrează lista de e-mulţimi evaluate obţinută la pasl 7 după cum urmează: dacă o aceeaşi e-mulţime intervine în mod identic (i.e. conţine aceleaşi cuvinte şi în aceeaşi ordine) corespunzător atât verbului din synset-ul englezesc iniţial, cât şi tuturor celorlalte verbe care au fost utilizate pentru îmbogăţire, atunci adună toate scorurile obţinute de această e-mulţime; evaluează aceasta e-mulţime, corespunzător verbului din synset-ul englezesc iniţial, cu scorul total.

9. Sortează (conform scorurilor obţinute şi cuvintelor englezeşti cărora le corespund), în ordine crescătoare sau descrescătoare, lista de e-mulţimi evaluate obţinută.

Output: Lista sortată a e-mulţimilor corespunzătoare synset-ului englezesc dat. ❑

Se observă faptul că pasul 8 al acestui algoritm reprezintă principalul mod în care tehnica de îmbogăţire propusă contribuie la specificarea sensului unicului verb polisemantic care intervine în synset-ul englezesc iniţial (a se vedea şi exemplul din §5.7.1).

Lista sortată de e-mulţimi generată de Algoritmul 5.5 trebuie folosită, în continuare, ca input pentru Algoritmul 5.2, care va avea ca output synset-ul de verbe străin corespunzător celui englezesc dat.

5.7.1. Exemple şi consideraţii finale privitoare la verbe

Pentru a ilustra felul în care lucrează algoritmul de traducere, ne vom propune, în continuare, să traducem două synset-uri de verbe englezeşti în limba română. În cele ce urmează, toate rezultatele prezentate constituie output-ul unor programe Prolog.

Vom considera, mai întâi, synset-ul englezesc `[unite, unify, merge]`, care are *id*-ul 200193167 şi se referă la conceptul definit de următoarea glosă: *join or combine; "We merged our resources"*. E-mulţimile evaluate obţinute corespunzător acestui synset, la pasul 5 al Algoritmului 5.1, atunci când se foloseşte aceeaşi funcţie de evaluare propusă anterior, sunt următoarele:

```
evset(200193167,unite,3,[a_uni, a_reuni, a_unifica]).
evset(200193167,unite,1,[a_lega]).
evset(200193167,unify,3,[a_unifica, a_uni, a_reuni]).
evset(200193167,merge,4,[a_contopi, a_fuziona, a_uni]).
```

Aici *evset* este acelaşi operator care desemnează e-muţimi evaluate folosit şi în secţiunile anterioare. În acest caz, programul implementând Algoritmul 5.2 are următorul output:

```
Traducerea synset-ului 200193167 = [unite, unify, merge]
avand g=(join or combine; "We merged our resources"):
trad(200193167, [a_uni, a_reuni, a_unifica, a_contopi,
a_fuziona]).
```

Traducerea synset-ului englezesc dat, oferită de către algoritmii implementaţi prin intermediul operatorului *trad*, este deci synset-ul românesc de verbe `[a_uni, a_reuni, a_unifica, a_contopi, a_fuziona]`, care reprezintă, în opinia lingviştilor români, o traducere corectă.

Vom lua acum în consideraţie un synset englezesc conţinând un unic verb polisemantic, pentru a testa tehnica de îmbogăţire propusă la pasul 2 al Algoritmului 5.5. În acest scop am ales, în mod aleator, synset-ul englezesc `[change]`, având *id*-ul 200134290 şi care se referă la conceptul definit de următoarea glosă: *change clothes; put on different clothes; "Change before you go to the opera"*.

Vom face, mai întâi, observaţia că verbul polisemantic *change* este înregistrat în versiunea 1.7.1 a WN ca având **zece** sensuri diferite. El există, prin urmare, în zece synset-uri diferite (fără a lua în consideraţie colocaţiile). În **trei** dintre aceste zece synset-uri *change* este singurul cuvânt care intervine. Se mai observă că nu toate cele trei synset-uri pot fi îmbogăţite aşa cum se recomandă

la pasul 2 al Algoritmului 5.5, întrucât nu există, în toate cazurile, synset-uri similare ca sens.

Revenind la synset-ul [change] având *id*-ul 200134290, după căutarea efectuată în fişierul corespunzător operatorului **vgp** la cel de-al doilea pas al Algoritmului 5.5, observăm că acest synset este similar ca sens cu synset-ul de verbe având *id*-ul 200441708:

vgp(200134290, 200441708).

Ultimul este synset-ul [switch, shift, change], care se referă la conceptul desemnat de următoarea glosă: *lay aside, abandon, or leave for another; "switch to a different brand of beer"; "She switched psychiatrists"; "The car changed lanes".*

Synset-ul dat [change] va fi, prin urmare, îmbogăţit cu verbele aparţinând acestui nou synset, obţinându-se, la pasul 2.2 al Algoritmului 5.5, synset-ul [change, switch, shift], care este, de fapt, cel care va fi tradus.

E-mulţimile evaluate obţinute corespunzător acestui ultim synset, la pasul 7 al Algoritmului 5.5, atunci când este folosită aceeaşi funcţie de evaluare, sunt următoarele:

```
evset(200134290, change, 2, [a_schimba, a_modifica]).
evset(200134290, change, 1, [a_altera]).
evset(200134290, change, 2, [a_preface, a_transforma]).
evset(200134290, change, 1, [a_converti]).
evset(200134290, change, 1, [a_schimba]).
evset(200134290, switch, 2, [a_schimba]).
evset(200134290, shift, 2, [a_schimba]).
```

Se observă (la pasul 8 al Algoritmului 5.5) că aceeaşi e-mulţime, [a_schimba], intervine corespunzător atât verbului *change* al synset-ului initial, cât şi tuturor celorlalte verbe care au fost utilizate pentru îmbogăţire. După calcularea scorului total al acestei e-mulţimi relativ la verbul *change*, la pasul 8 al algoritmului şi după sortarea noii liste de e-mulţimi evaluate la pasul 9, se obţine următoarea *listă finală a e-mulţimilor evaluate*, care va constitui input pentru Algoritmul 5.2:

```
evset(200134290, change, 5, [a_schimba]).
evset(200134290, change, 2, [a_schimba, a_modifica]).
evset(200134290, change, 2, [a_preface, a_transforma]).
evset(200134290, change, 1, [a_altera]).
evset(200134290, change, 1, [a_converti]).
evset(200134290, switch, 2, [a_schimba]).
evset(200134290, shift, 2, [a_schimba]).
```


Programul implementând Algoritmul 5.2 a ales synset-ul românesc corect [a_schimba] ca reprezentând traducerea synset-ului englezesc dat [change], care are *id*-ul 200134290. Synset-ului românesc obţinut i se va atribui acelaşi *id* de synset.

Ne propunem acum să vedem ce se întâmplă atunci când se încearcă traducerea în limba română a aceluiaşi synset [change] având *id*-ul 200134290, dar fără a se efectua îmbogăţirea presupusă de pasul 2 al Algoritmului 5.5.

În acest caz se obţin următoarele e-mulţimi evaluate, la pasul 5 al Algoritmului 5.1:

```
evset(200134290, change, 2, [a_schimba, a_modifica]).
evset(200134290, change, 1, [a_altera]).
evset(200134290, change, 2, [a_preface, a_transforma]).
evset(200134290, change, 1, [a_converti]).
evset(200134290, change, 1, [a_schimba]).
```

Traducerea synset-ului ales [change], efectuată fără îmbogăţire, indică (Algoritmul 5.2) două synset-uri româneşti posibile:

[a_schimba, a_modifica] şi [a_preface, a_transforma].

Alegerea cu privire la synset-ul care reprezintă traducerea corectă trebuie făcută în conformitate cu glosa şi implică efort suplimentar de natură umană. În acest caz particular se observă însă că *ambele synset-uri propuse constituie traduceri greşite* ale celui englezesc dat, întrucât aici sensul lui *change* nu este nici acela de a modifica şi nici acela de a transforma.

Atunci când sunt studiate astfel de exemple, necesitatea de a folosi operatorul **vgp** şi relaţia corespunzătoare, ori de câte ori acest lucru este posibil, devine evidentă. Acesta este motivul pentru care, spre deosebire de alţi autori care utilizează conceptul de e-mulţime, restrângându-şi abordarea exclusiv la aplicarea metodelor de clasă [24], credem cu tărie în necesitatea combinării unor astfel de metode cu cele de natură structurală.

Ca şi în cazul substantivelor şi al adjectivelor, o problemă specială o ridică synset-urile conţinând un unic verb polisemantic. În astfel de cazuri, nu se poate şti la care dintre sensurile cuvântului dat se referă un anumit synset. Singura modalitate de a afla este aceea de a consulta glosa asociată. Pentru a minimiza efortul uman şi a automatiza, cât mai mult cu putinţă, procesul de traducere, a fost inclus un "pas de îmbogăţire" în Algoritmul 5.1. Pasul al doilea al algoritmului rezultat foloseşte o relaţie semantică care este tipică synset-urilor de verbe şi care este de o importanţă crucială în specificarea conceptului la care se referă synset-ul englezesc aflat sub investigaţie, aşa cum s-a putut observa în exemplul anterior.

5.8. Synset-uri de adverbe

Întrucât pentru synset-urile de adverbe nu este disponibilă nici o relaţie semantică de tip "similaritate a sensului", am efectuat traducerea unor astfel de synset-uri în limba română prin intermediul Algoritmilor 5.1 si 5.2, restricţionând, în acest caz, abordarea la folosirea unei metode de clasă. Studiul nostru s-a concentrat aici asupra unui alt aspect discutat în [15]. Astfel, cu scopul comparării rezultatelor, am implementat primul algoritm menţionat folosind două funcţii de evaluare diferite, şi anume cea descrisă în §5.2 şi o alta, obţinută în acelaşi fel, dar fără efectuarea împărţirii la cardinalul e-mulţimii.

În cele ce urmează, prezentăm două exemple de traduceri, unde, în fiecare dintre cazuri, primul tabel şi rezultatul corespunzător au fost obţinute prin folosirea funcţiei de evaluare din §5.2, în timp ce al doilea rezultat corespunde celei de a doua funcţii de evaluare menţionate.

Exemplul nr. 1

Synset englezesc: {happily, merrily, mirthfully, gayly, blithely, jubilantly, with_happiness}

Glosa: in a joyous manner ; "they shouted happily"

e-multimi:

cuvante	e-multime	scor
jubilantly	{vesel}	7.0
blithely	{vesel, voios, lipsit_de_griji}	3.6666666666666665
happily	{fericit, vesel, entuziast}	3.0
merrily	{voios, cu_veselie}	3.0
gayly	{vesel, jovial, bine_dispus}	2.9999999999999996
mirthfully	{cu_veselie, cu_voiosie, cu_bucurie}	2.6666666666666665
with_happiness	{cu_veselie, cu_jovialitate}	2.5
happily	{bucuros, cu_bucurie}	2.0
with_happiness	{cu_noroc, cu_bucurie, cu_fericire}	1.6666666666666665
happily	{din_fericire}	1.0
gayly	{tipator}	1.0
gayly	{destrebalat, desucheat}	1.0
gayly	{fara_cumpatare}	1.0

jubilantly	{triumfator, jubiland}	1.0
with_happiness	{cu_fericire, cu_multumire, cu_satisfactie}	1.0
gayly	{zgomotos, tipator}	0.5

Synset romanesc propus:

- {vesel, voios, lipsit_de_griji, jovial, bine_dispus, fericit, entuziast, cu_veselie, cu_voioşie, cu_bucurie, cu_jovialitate}

Synset englezesc: {happily, merrily, mirthfully, gayly, blithely, jubilantly, with_happiness}

Glosa: in a joyous manner ; "they shouted happily"

e-multimi:

cuvante	e-multime	scor
blithely	{vesel, voios, lipsit_de_griji}	11.0
happily	{fericit, vesel, entuziast}	9.0
gayly	{vesel, jovial, bine_dispus}	9.0
mirthfully	{cu_veselie, cu_voiosie, cu_bucurie}	8.0
jubilantly	{vesel}	7.0
merrily	{voios, cu_veselie}	6.0
with_happiness	{cu_noroc, cu_bucurie, cu_fericire}	5.0
with_happiness	{cu_veselie, cu_jovialitate}	5.0
happily	{bucuros, cu_bucurie}	4.0
with_happiness	{cu_fericire, cu_multumire, cu_satisfactie}	3.0
gayly	{destrebalat, desucheat}	2.0
jubilantly	{triumfator, jubiland}	2.0
happily	{din_fericire}	1.0
gayly	{fara_cumpatare}	1.0
gayly	{tipator}	1.0
gayly	{zgomotos, tipator}	1.0

Synset romanesc propus:

- {vesel, voios, lipsit_de_griji, jovial, bine_dispus, fericit, entuziast, cu_veselie, cu_voiosie, cu_bucurie, cu_noroc, cu_fericire}
- {vesel, voios, lipsit_de_griji, jovial, bine_dispus, fericit, entuziast, cu_veselie, cu_voiosie, cu_bucurie, cu_jovialitate}

Se observă faptul că, în acest caz, traducerea obţinută prin folosirea funcţiei de evaluare definite în §5.2 este cea mai adecvată.

Exemplul nr. 2

Synset englezesc: {intentionally, deliberately, designedly, on_purpose, purposely, advisedly, by_choice, by_design}

Glosa: with intention; in an intentional manner; "he used that word intentionally"; "I did this by choice"

e-multimi:

cuvante	e-multime	scor
on_purpose	{dinadins}	7.0
purposely	{dinadins, intentionat, cu_premeditare}	7.0
by_design	{cu_intentie, intentionat}	7.0
deliberately	{dinadins, intentionat, voit, cu_premeditare}	6.0
designedly	{cu_intentie, intentionat, cu_buna_stiinta}	5.0
advisedly	{planuit, intentionat, cu_buna_stiinta}	4.666666666666666
intentionally	{intentionat, voit, dinadins, anume, expré}	4.6000000000000005
advisedly	{dupa_matura_chibzuinta}	1.0
advisedly	{judicious}	1.0
by_choice	{de_preferinta}	1.0
by_choice	{cu_precadere, indeosebi}	1.0
deliberately	{gandit, cu_grija, atent}	0.6666666666666666
deliberately	{incet, fara_graba}	0.5

Synset romanesc propus:

- {planuit, intentionat, cu_buna_stiinta, de_preferinta, cu_intentie, dinadins, voi cu_premeditare, anume, expré}
- {planuit, intentionat, cu_buna_stiinta, cu_precadere, indeosebi, cu_intentie dinadins, voit, cu_premeditare, anume, expré}

Synset englezesc: {intentionally, deliberately, designedly, on_purpose, purposely advisedly, by_choice, by_design}

Glosa: with intention; in an intentional manner; "he used that word intentionally" "I did this by choice"

e-multimi:

cuvante	e-multime	scor
deliberately	{dinadins, intentionat, voit, cu_premeditare}	24.0
intentionally	{intentionat, voit, dinadins, anume, expré}	23.0
purposely	{dinadins, intentionat, cu_premeditare}	21.0
designedly	{cu_intentie, intentionat, cu_buna_stiinta}	15.0
advisedly	{planuit, intentionat, cu_buna_stiinta}	14.0
by_design	{cu_intentie, intentionat}	14.0
on_purpose	{dinadins}	7.0
deliberately	{gandit, cu_grija, atent}	2.0
by_choice	{cu_precadere, indeosebi}	2.0
deliberately	{incet, fara_graba}	1.0
advisedly	{judicious}	1.0
advisedly	{dupa_matura_chibzuinta}	1.0
by_choice	{de_preferinta}	1.0

Synset romanesc propus:

- {planuit, intentionat, cu_buna_stiinta, cu_precadere, indeosebi, cu_intentie, dinadins, voit, cu_premeditare, anume, expré}

Se observă faptul că, în cazul acestui synset englezesc, traducerea obţinută prin folosirea celeilalte funcţii de evaluare pare mult mai adecvată.

Nici una dintre traducerile generate nu este în întregime corectă, ceea ce ne conduce la presupunerea că rezultate mai bune ar putea fi obţinute prin folosirea unei funcţii de evaluare complet diferite.

Acest tip de rezultat reafirmă, o dată în plus, necesitatea combinării metodelor de clasă cu cele structurale, ori de câte ori acest lucru este posibil, punând, totodată, în evidenţă importanţa funcţiei de evaluare folosite de Algoritmul 5.1 şi de variantele sale. Devine evident faptul că, în viitor, sunt necesare conceperea şi testarea unor noi funcţii de evaluare.

5.9. Consideraţii finale

Abordarea propusă referitoare la generarea unor reţele semantice de tip WN reprezintă o combinaţie de metode automate şi manuale. Metoda manuală se bazează pe experţi umani, în timp ce cea automată depinde în mod nemijlocit de existenţa dicţionarelor bilingve în format electronic şi reprezintă, la rândul ei, o combinaţie de metode de clasă şi structurale.

Semnificaţia efortului manual presupus de asigurarea calităţii depinde, în primul rând, de existenţa unor instrumente adecvate. Efortul uman implicat poate fi substanţial redus în cazul acelor limbi pentru care există dicţionare bilingve corecte şi complete, în format electronic. Astfel, s-a constatat că, în majoritatea cazurilor în care rezultatele obţinute nu au fost cele mai bune cu putinţă, acest lucru se datorează, în principal, imperfecţiunii dicţionarelor bilingve existente.

Principalele dificultăţi care au fost întâmpinate la traducerea automată a synset-urilor englezeşti în limba română au fost generate de colocaţii, de calcul lingvistic şi de superioritatea polisemiei multor cuvinte englezeşti în raport cu corespondentele lor româneşti. Indiferent ce limbă este luată în consideraţie, vor exista întotdeauna dificultăţi de natură lingvistică, care nu pot fi evitate. În plus, trebuie să remarcăm faptul că majoritatea problemelor apar atunci când sunt traduse synset-uri englezeşti care includ un singur cuvânt polisemantic, caz în care Algoritmul 5.1 nu poate decide între sensurile existente. Astfel de synset-uri au constituit obiectul unor investigaţii ulterioare, în urma cărora a fost adăugat Algoritmului 5.1 pasul de îmbogăţire, care ia în consideraţie structura WN, ori de câte ori acest lucru a fost posibil. Metoda noastră îmbunătăţită reiterează necesitatea combinării metodelor de clasă cu cele structurale, în realizarea unei sarcini de acest tip. În egală măsură, a devenit evidentă importanţa funcţiei de evaluare, precum şi faptul că noi astfel de funcţii ar trebui concepute şi testate în viitor.

CAPITOLUL 6
RAŢIONAMENT STATISTIC

Diversele tehnici de reprezentare a cunoştinţelor pot fi augmentate cu ajutorul unor măsuri statistice care descriu niveluri ale dovezilor existente, precum şi ale convingerilor unui agent. În cele ce urmează, vom prezenta şi folosi unele instrumente furnizate de statistica Bayesiană pentru a realiza această augmentare, atât de necesară în descrierea părerilor care nu reprezintă certitudini, dar în favoarea cărora există suficiente dovezi.

Se poate spune că principalul avantaj al raţionamentului de tip probabilist, faţă de raţionamentul de tip logic, rezidă în faptul că cel dintâi permite agentului să ajungă la decizii raţionale chiar şi atunci când nu există suficientă informaţie care să dovedească că o acţiune dată va avea efectul dorit.

6.1. Abordări statistice pentru sisteme de raţionament incert

Un important ţel al multor sisteme de rezolvare a problemelor este acela de a colecţiona probe, pe măsură ce sistemul avansează şi de a-şi modifica comportamentul pe baza acestor dovezi. Statistica Bayesiană este o teorie statistică care poate modela acest tip de comportament.

Dacă notăm cu H evenimentul ca ipoteza (notată H) să fie adevărată, iar cu E evenimentul obţinut prin observarea "dovezii" E, atunci conceptul fundamental al statisticii Bayesiene poate fi considerat ca fiind acela de *probabilitate condiţionată*,

$$P(H \mid E),$$

reprezentând probabilitatea ca ipoteza H să fie adevărată, atunci când se observă dovada E. Pentru a calcula această probabilitate trebuie luate în consideraţie probabilitatea prealabilă sau *a priori* a lui H (i.e. probabilitatea pe care am atribui-o lui H în lipsa oricăror probe), precum şi gradul până la care E furnizează dovezi în favoarea lui H. Acest lucru se realizează prin definirea unui univers conţinând o mulţime exhaustivă de ipoteze H_i, care se exclud reciproc şi între care încercăm să discernem. Fie

E = dovezile obţinute printr-un experiment auxiliar.

$P(H_i)$ = probabilitatea a priori ca ipoteza H_i să fie adevărată.

$P(E \mid H_i)$ = probabilitatea de a observa dovezile E (probabilitatea să se realizeze E), atunci când ipoteza H_i este adevărată.

$P(H_i \mid E)$ = probabilitatea a posteriori ca ipoteza H_i să fie adevărată, fiind date dovezile E (ştiind că s-a realizat E).

Ca modalitate de calcul, probabilitatea a posteriori $P(H_i \mid E)$ se obţine în mod Bayesian. Dacă notăm prin k numărul ipotezelor posibile, atunci formula lui Bayes, în varianta necondiţionată, calculează această probabilitate a posteriori în felul următor (vezi Anexa 2):

$$P(H_i \mid E) = \frac{P(E \mid H_i) \cdot P(H_i)}{\sum_{n=1}^{k} P(E \mid H_n) \cdot P(H_n)}$$

Teorema lui Bayes poate sta la baza raţionamentului incert. În general, probabilitatea $P(A \mid B)$ descrie probabilitatea condiţionată a lui A atunci când singurele dovezi de care dispunem sunt reprezentate de B. Dacă însă există şi alte dovezi relevante, atunci acestea trebuie şi ele luate în consideraţie. Atunci când sunt date un corp prealabil de dovezi e şi o nouă observaţie E, trebuie calculată probabilitatea condiţionată a posteriori $P(H \mid E, e)$, prin aplicarea formulei lui Bayes în versiune condiţionată (vezi Anexa 2). Avem:

$$P(H \mid E, e) = P(H \mid E) \cdot \frac{P(e \mid E, H)}{P(e \mid E)}$$

Într-o lume arbitrar de complexă, dimensiunea mulţimii repartiţiilor de probabilitate multidimensionale, care este necesară pentru a calcula această funcţie, creşte la valoarea 2^n dacă sunt luate în consideraţie n propoziţii diferite. Aşa cum se arată în [8], utilizarea teoremei lui Bayes face ca problema să devină insolubilă din mai multe motive:

- Problema achiziţionării de cunoştinţe devine insurmontabilă, întrucât trebuie furnizate prea multe probabilităţi.
- Spaţiul necesar pentru memorarea tuturor probabilităţilor este prea mare.
- Timpul necesar pentru calcularea tuturor probabilităţilor este prea mare.

În ciuda acestor neajunsuri, statistica Bayesiană constituie o bază atractivă pentru un *sistem de raţionament incert*. Din această cauză, au fost dezvoltate câteva mecanisme care îi exploatează puterea, făcând, în acelaşi timp, problema rezolvabilă, din punct de vedere computaţional. Două dintre aceste mecanisme, asupra cărora ne vom opri în continuare, constau în ataşarea unor *factori de certitudine* regulilor de producţie şi respectiv în folosirea *reţelelor Bayesiene*.

6.1.1. Factori de certitudine şi sisteme bazate pe reguli

Abordarea pe care o propunem aici a fost preluată din [8] şi provine de la sistemul MYCIN [Shortliffe, 1976; Buchanan şi Shortliffe, 1984; Shortliffe şi Buchanan, 1975], care îşi propune să recomande terapii adecvate pacienţilor cu infecţii bacteriene. MYCIN este un exemplu de *sistem expert*, care interacţionează cu medicul pentru a dobândi datele clinice de care are nevoie.

MYCIN reprezintă majoritatea cunoştinţelor sale legate de diagnostic ca pe o mulţime de reguli, fiecare regulă având asociat un factor de certitudine. Sistemul foloseşte aceste reguli pentru a face un raţionament de tip *înlănţuire înapoi* de la scopul său de a detecta organisme semnificative care pot cauza maladii şi până la datele clinice disponibile. De îndată ce identifică asemenea organisme, MYCIN încearcă să selecteze o terapie prin care boala să fie tratată. Pentru a înţelege cum exploatează sistemul informaţia incertă trebuie stabilit *cum combină acesta estimaţiile de certitudine ale fiecărei reguli în parte pentru a produce o estimaţie finală a certitudinii concluziilor sale.* O întrebare naturală care se ridică, în egală măsură, având în vedere observaţiile anterioare privind insolvabilitatea spre care conduce raţionamentul Bayesian pur, este următoarea: ce compromisuri trebuie să facă tehnica MYCIN şi care sunt riscurile asociate acestor compromisuri?

Un ***factor de certitudine*** (notat aici prin $FC[h,e]$ sau, mai simplu, prin FC) este definit în termenii a două componente:

- $MI[h,e]$ - o măsură (între 0 şi 1) a încrederii în ipoteza h fiind dată dovada e. MI măsoară gradul până la care dovezile existente susţin ipoteza. Valoarea este 0 dacă aceste dovezi eşuează în susţinerea ipotezei.
- $MN[h,e]$ - o măsură (între 0 şi 1) a neîncrederii în ipoteza h fiind dată dovada e. MN măsoară gradul până la care dovezile existente susţin *negaţia ipotezei*. Valoarea este 0 dacă aceste dovezi susţin ipoteza.

Fiind date aceste două măsuri, putem defini factorul de certitudine ca fiind

$$FC[h,e] = MI[h,e] - MN[h,e].$$

Întrucât orice dovadă particulară fie susţine, fie neagă o ipoteză şi întrucât fiecare regulă MYCIN corespunde unei anumite dovezi (deşi aceasta ar putea fi o dovadă compusă), este suficient un singur număr corespunzător fiecărei reguli pentru a defini atât MI, cât şi MN şi, prin urmare, factorul de certitudine, FC.

Factorii de certitudine ai regulilor MYCIN sunt furnizaţi de către experţii care scriu aceste reguli. Pe măsură însă ce sistemul raţionează, *aceşti*

factori de certitudine trebuie combinaţi, pentru a reflecta dovezi multiple şi reguli multiple aplicate aceleiaşi probleme. Cele trei tipuri de combinaţii care trebuie luate în consideraţie sunt reflectate în Fig. 6.1. Astfel, în Fig. 6.1 (a) mai multe reguli furnizează dovezi referitoare la o singură ipoteză. În Fig. 6.1 (b) mai multe propoziţii trebuie luate în consideraţie împreună pentru a ne forma o părere. În Fig. 6.1 (c) output-ul corespunzător unei reguli furnizează input pentru o alta.

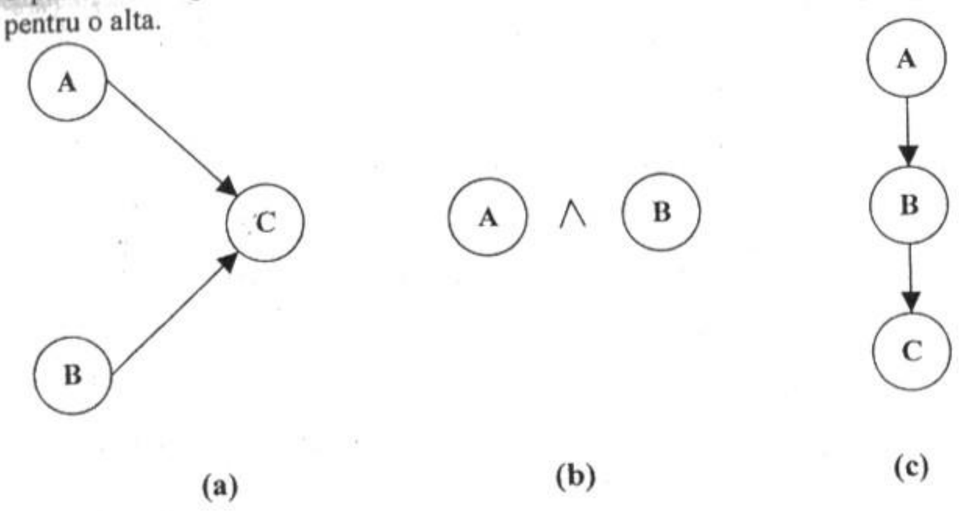


Fig. 6.1

Înainte de a stabili formulele care trebuie folosite pentru a realiza aceste combinaţii, trebuie avute în vedere anumite *proprietăţi pe care funcţia de combinare trebuie să le satisfacă*, şi anume:

- Întrucât ordinea în care dovezile sunt colectate este arbitrară, funcţiile de combinare trebuie să fie comutative şi asociative.
- Până când este atinsă certitudinea, dovezile suplimentare ar trebui să crească valoarea lui MI. (În mod similar, dovezile care infirmă ipoteza ar trebui să crească valoarea lui MN).
- Dacă sunt înlănţuite inferenţe incerte, atunci rezultatul ar trebui să fie mai puţin cert decât fiecare inferenţă în parte.

Acceptând necesitatea acestor proprietăţi[25], vom considera mai întâi situaţia din Fig. 6.1 (a), în care *mai multe probe se combină pentru a se determina factorul de certitudine al unei ipoteze*. Măsura încrederii şi a neîncrederii într-o ipoteză, fiind date două observaţii, s_1 şi s_2, se calculează după cum urmează:

[25] Descrierea sistemului MYCIN este făcută aici conform [26].

$$MI[h, s_1 \wedge s_2] = \begin{cases} 0, & \text{daca } MN[h, s_1 \wedge s_2] = 1 \\ MI[h, s_1] + MI[h, s_2] \cdot (1 - MI[h, s_1]), & \text{altfel} \end{cases}$$

$$MN[h, s_1 \wedge s_2] = \begin{cases} 0, & \text{daca } MI[h, s_1 \wedge s_2] = 1 \\ MN[h, s_1] + MN[h, s_2] \cdot (1 - MN[h, s_1]), & \text{altfel} \end{cases}$$

$FC[h, s_1 \wedge s_2]$ se calculează pe baza lui $MI[h, s_1 \wedge s_2]$ şi $MN[h, s_1 \wedge s_2]$. Se observă că, dacă se coroborează mai multe dovezi ale aceleiaşi ipoteze, atunci valoarea absolută a lui FC va creşte. Dacă sunt introduse dovezi conflictuale, atunci valoarea absolută a lui FC va descreşte.

În situaţia din Fig. 6.1 (b) este necesar *calculul factorului de certitudine al unei conjuncţii de ipoteze*. FC se calculează pe baza valorilor MI şi MN. Formulele folosite de MYCIN pentru calculul lui MI, corespunzător unei conjuncţii şi respectiv unei disjuncţii de ipoteze, sunt următoarele:

$$MI[h_1 \wedge h_2, e] = \min(MI[h_1, e], MI[h_2, e])$$

$$MI[h_1 \vee h_2, e] = \max(MI[h_1, e], MI[h_2, e])$$

MN poate fi calculat în mod analog.

Fig. 6.1 (c) prezintă cazul în care *regulile sunt înlănţuite laolaltă*, ceea ce are ca rezultat faptul că ieşirea incertă a uneia dintre reguli trebuie să constituie intrarea alteia. Soluţia dată acestei probleme se va ocupa şi de cazul în care trebuie atribuită o măsură a incertitudinii intrărilor iniţiale. Aceste situaţii sunt frecvente şi corespund cazurilor în care dovezile provin ca urmare a unui experiment sau al unui test de laborator ale cărui rezultate nu sunt suficient de exacte. În aceste cazuri, factorul de certitudine al unei ipoteze trebuie să ia în consideraţie atât tăria cu care probele sugerează ipoteza, cât şi nivelul de încredere în aceste probe. MYCIN furnizează o regulă de înlănţuire care este definită după cum urmează.

Fie $MI'[h, s]$ măsura încrederii în h atunci când suntem absolut siguri de validitatea lui s. Fie e dovada care ne-a determinat să credem în s (spre exemplu, măsurătorile efectuate cu ajutorul instrumentelor de laborator sau rezultatele aplicării altor reguli). Atunci:

$$MI[h, s] = MI'[h, s] \cdot \max(0, FC[s, e])$$

Întrucât valorile FC *iniţiale* din sistemul MYCIN sunt date de către experţii care scriu regulile, nu este necesar să se formuleze o definiţie mai exactă a factorului de certitudine decât cea dată până acum. Autorii sistemului

inițial au dat însă o astfel de definiție, exprimând pe MI (care poate fi privit ca o descreștere proporțională a neîncrederii în h ca rezultat al observării lui e) după cum urmează:

$$MI[h,e]=\begin{cases}1, & \text{daca } P(h)=1\\ \dfrac{\max[P(h|e),P(h)]-P(h)}{1-P(h)}, & \text{altfel}\end{cases}$$

În mod similar, MN este descreșterea proporțională a încrederii în h ca rezultat al observării lui e:

$$MN[h,e]=\begin{cases}1, & \text{daca } P(h)=0\\ \dfrac{\min[P(h|e),P(h)]-P(h)}{-P(h)}, & \text{altfel}\end{cases}$$

Aceste definiții s-au dovedit a fi incompatibile cu viziunea Bayesiană asupra probabilității condiționate. Modificări minore le fac însă compatibile [Heckerman, 1986]. În particular, putem redefini MI astfel:

$$MI[h,e]=\begin{cases}1, & \text{daca } P(h)=1\\ \dfrac{\max[P(h|e),P(h)]-P(h)}{(1-P(h))\cdot P(h|e)}, & \text{altfel}\end{cases}$$

Definiția lui MN trebuie schimbată în mod similar.

Cu aceste reinterpretări nu mai există nici un conflict fundamental între tehnicile MYCIN și cele sugerate de statistica Bayesiană, MYCIN nefiind un sistem care conduce la insolubilitate.

Fiecare FC dintr-o regulă MYCIN reprezintă contribuția unei reguli individuale la încrederea pe care MYCIN o acordă unei anumite ipoteze. Într-un anume sens, acest factor reprezintă o probabilitate condiționată, $P(H|E)$. Dar, într-un sistem Bayesian pur, $P(H|E)$ descrie probabilitatea condiționată a lui H atunci când singurele dovezi relevante sunt date de E. Dacă există și alte probe, atunci trebuie luate în considerație repartiții de probabilitate multidimensionale. Acesta este momentul în care MYCIN se distanțează de un sistem Bayesian pur, devenind mai eficient în execuție, dar cu riscul de a avea un comportament mai puțin intuitiv. În particular, *formulele MYCIN pentru cele trei situații din Fig. 6.1 fac presupunerea că toate regulile sunt independente.* Sarcina garantării independenței revine expertului care scrie regulile. Fiecare dintre scenariile combinate ale Fig. 6.1 devine vulnerabil atunci când această presupunere de independență este încălcată.

Abordarea care utilizează factori de certitudine și sisteme bazate pe reguli face, prin urmare, *presupuneri puternice privitoare la independență*, datorită cărora este ușor de folosit în practică. *În același timp, regulile trebuie scrise în așa fel încât ele să reflecte dependențele importante*. Abordarea stă la baza multor programe, precum MYCIN, ca și a unei game largi de sisteme diferite, care au fost construite pe platforma EMYCIN [van Melle et al., 1981]. Platforma EMYCIN constituie o generalizare (numită *shell*) a lui MYCIN, în cadrul căreia au fost înlăturate toate regulile specifice domeniului. Unul dintre motivele pentru care acest cadru de lucru este util, în ciuda limitărilor sale, este acela că, într-un sistem robust, valorile exacte care sunt folosite nu au o importanță prea mare. Această abordare pare, de asemenea, a imita suficient de bine [Shultz et al., 1989] felul în care oamenii manipulează certitudinile.

6.1.2. Rețele Bayesiene

Factorii de certitudine au fost descrişi ca un mecanism de reducere a complexității unui sistem de raționament Bayesian prin intermediul unor aproximări ale formalismului. O *abordare alternativă* este cea bazată pe ***rețele Bayesiene*** [Pearl, 1988], abordare care conservă formalismul bazându-se, în schimb, pe modularitatea universului a cărui modelare se încearcă.

Ideea de bază a acestei abordări este aceea că, pentru a descrie lumea reală, nu este necesară folosirea unei tabele imense de repartiții de probabilitate multidimensionale în care să fie listate probabilitățile tuturor combinațiilor posibile de evenimente. Majoritatea evenimentelor sunt independente de celelalte, astfel încât interacțiunile dintre ele nu trebuie luate în considerație. În schimb, se poate folosi *o reprezentare locală* având rolul de a descrie grupuri de evenimente care interacționează. Astfel, în contextul folosirii regulii lui Bayes, relațiile de independență condiționată dintre variabile pot reduce substanțial numărul probabilităților condiționate care trebuie specificate. Structura de date folosită cel mai frecvent pentru a reprezenta dependența dintre variabile și pentru a da o specificație concisă a repartiției de probabilitate multidimensionale este ***rețeaua Bayesiană***.

Definiția 6.1

O *rețea Bayesiană* este un graf cu următoarele proprietăți:

(1) Graful este direcționat și aciclic.

(2) O mulțime de variabile aleatoare constituie nodurile rețelei.

(3) O mulțime de arce direcționate conectează perechi de noduri. În mod intuitiv, semnificația unui arc direcționat de la nodul X la nodul Y este aceea că X are *o influență directă* asupra lui Y.

(4) Fiecărui nod îi corespunde un tabel de probabilități condiționate care cuantifică efectele pe care părinții le au asupra nodului respectiv.

(Părinţii unui nod sunt toate acele noduri din care pleacă arce direcţionate înspre acesta).

Un expert în domeniu poate, de obicei, să decidă relativ uşor relaţiile directe de dependenţă condiţionată care sunt valabile în domeniu, acest lucru fiind mult mai uşor decât specificarea probabilităţilor propriu-zise. După stabilirea topologiei reţelei Bayesiene, nu mai este necesară decât specificarea probabilităţilor condiţionate ale acelor noduri care participă în dependenţe directe, urmând ca acestea să fie apoi folosite în calculul oricăror alte probabilităţi.

În mod concret, se construieşte un graf direcţionat aciclic care reprezintă *relaţiile de cauzalitate* dintre variabile. Variabilele dintr-un astfel de graf pot fi propoziţionale (caz în care pot lua valorile TRUE sau FALSE) sau pot fi variabile care primesc valori de un alt tip (spre exemplu o temperatură a corpului sau măsurători făcute de către un dispozitiv de diagnosticare).

În Fig. 6.2 este prezentat un astfel de graf al cauzalităţii (reţea). Un asemenea graf ilustrează relaţiile de cauzalitate care intervin între nodurile pe care le conţine. Pentru a-l putea folosi ca bază a unui raţionament de tip probabilist sunt necesare însă mai multe informaţii. În particular este necesar să cunoaştem, pentru o valoare a unui nod părinte, ce dovezi sunt furnizate referitor la valorile pe care le poate lua nodul fiu.

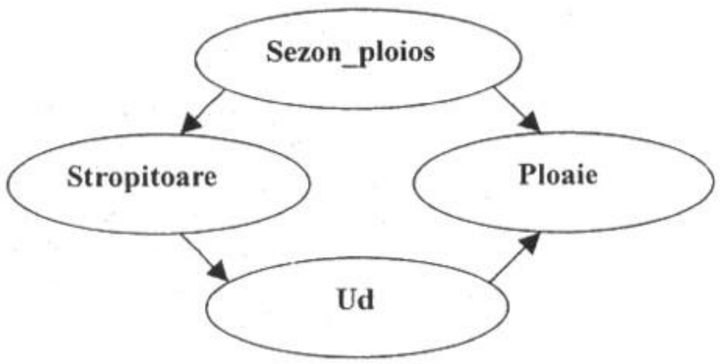


Fig. 6.2

Probabilităţile condiţionate corespunzătoare pot fi date sub forma unui tabel de tipul celui care urmează. Din acest tabel, preluat de noi, spre exemplificare, din [8], se poate citi, de pildă, probabilitatea a priori de a avea un sezon ploios ca fiind 0.5. Dacă suntem în timpul unui sezon ploios, probabilitatea de a avea ploaie într-o noapte dată este 0.9. Dacă sezonul nu este ploios, această probabilitate este numai 0.1.

Atribut	Probabilitate
p(Ud|Stropitoare, Ploaie)	0.95
p(Ud|Stropitoare, ¬ Ploaie)	0.9
p(Ud| ¬ Stropitoare, Ploaie)	0.8
p(Ud| ¬ Stropitoare, ¬ Ploaie)	0.1
p(Stropitoare| Sezon ploios)	0.0
p(Stropitoare| ¬ Sezon ploios)	1.0
p(Ploaie|Sezon ploios)	0.9
p(Ploaie| ¬ Sezon ploios)	0.1
p(Sezon ploios)	0.5

Pentru a putea folosi o asemenea reprezentare în rezolvarea problemelor este necesar un mecanism de calcul al influenţei oricărui nod arbitrar asupra oricărui alt nod. Pentru a obţine acest mecanism de calcul este necesar ca graful iniţial să fie convertit la un graf nedirecţionat, în care arcele să poată fi folosite pentru a se transmite probabilităţi în oricare dintre direcţii, în funcţie de locul din care provin dovezile. Este, de asemenea, necesar un mecanism de folosire a grafului care să garanteze transmiterea corectă a probabilităţilor. Spre exemplu, deşi este adevărat că iarba udă poate constitui o dovadă de ploaie şi că existenţa ploii constituie o dovadă în favoarea ierbii ude, trebuie să garantăm faptul că nici un ciclu nu este vreodată traversat astfel încât iarba udă să constituie o dovadă a ploii, care să fie considerată apoi o dovadă a existenţei ierbii ude ş.a.m.d..

Există trei mari clase de algoritmi [8] care efectuează aceste calcule, ideea care stă la baza tuturor acestor metode fiind aceea că nodurile au domenii limitate de influenţă. Metoda cea mai folosită este probabil cea a transmiterii mesajelor [Pearl, 1988], abordare care se bazează pe observaţia că, pentru a calcula probabilitatea unui nod A condiţionat de ceea ce se ştie despre celelalte noduri din reţea, este necesară cunoaşterea a trei tipuri de informaţii:

- suportul total care soseşte în A de la nodurile sale părinte (reprezentând cauzele sale);
- suportul total care soseşte în A de la fiii acestuia (reprezentând simptomele sale);
- intrarea în matricea fixată de probabilităţi condiţionate care face legătura dintre nodul A şi cauzele sale.

Au fost dezvoltate câteva metode de propagare a mesajelor de primele două tipuri, precum şi de actualizare a probabilităţilor corespunzătoare nodurilor. Structura reţelei determină abordarea care va fi folosită. Spre exemplu, în reţelele unic conectate (acelea în care există un singur drum între

fiecare pereche de noduri), se poate folosi un algoritm mai simplu (a se vedea §6.1.2.2.1 al lucrării de faţă) decât în cazul celor conectate multiplu.

6.1.2.1. Relaţii de independenţă condiţionată în reţele Bayesiene

O reţea Bayesiană exprimă independenţa condiţionată a unui nod şi a predecesorilor săi, fiind daţi părinţii acestuia şi foloseşte această independenţă pentru a proiecta o metodă de construcţie a reţelelor. Pentru a stabili *algoritmi de inferenţă* trebuie însă să ştim dacă se verifică independenţe condiţionate mai generale. Fiind dată o reţea dorim să putem "citi" dacă o mulţime de noduri X este independentă de o altă mulţime Y, fiind dată o mulţime de "noduri dovezi" E. Metoda de stabilire a acestui fapt se bazează pe noţiunea de *separare dependentă de direcţie* sau **d-separare**, definită după cum urmează:

Definiţia 6.2
O mulţime de noduri E *d-separă* două mulţimi de noduri X şi Y dacă orice drum nedirecţionat de la un nod din X la un nod din Y este **blocat** condiţionat de E.

Definiţia 6.3
Fiind dată o mulţime de noduri E, spunem că un drum este *blocat* condiţionat de E dacă există un nod Z aparţinând drumului, pentru care una dintre următoarele trei condiţii se verifică:
(1) Z aparţine lui E şi Z are o săgeată a drumului intrând în E şi o săgeată a drumului ieşind din E.
(2) Z aparţine lui E şi Z are ambele săgeţi ale drumului ieşind din E.
(3) Nici Z şi nici vreunul dintre descendenţii săi nu aparţin lui E, iar ambele săgeţi ale drumului ţintesc înspre Z.

Fig. 6.3 ilustrează cele trei cazuri posibile. S-a demonstrat că *dacă orice drum nedirecţionat de la un nod din X la un nod din Y este d-separat de E, atunci X şi Y sunt independente condiţionat de E*. Aceasta este teorema fundamentală a reţelelor Bayesiene, demonstrată de Verma şi Pearl. Demonstraţia acestei teoreme este suficient de complicată şi nu ne vom opri asupra ei în cadrul restrâns al lucrării de faţă. Reţinem faptul că noţiunea de d-separare este fundamentală în construcţia algoritmilor de inferenţă. Procesul construirii şi folosirii reţelelor Bayesiene nu utilizează d-separarea.

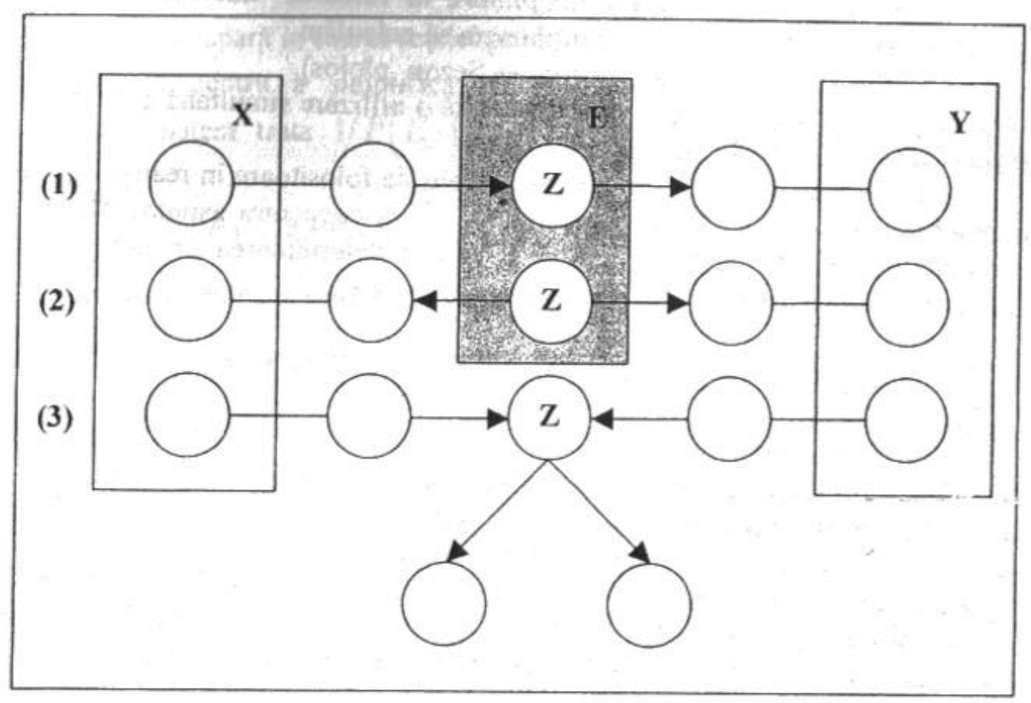


Fig. 6.3

6.1.2.2. Inferenţa în reţele Bayesiene

Sarcina principală a oricărui sistem probabilist de inferenţă este aceea de a calcula probabilităţi a posteriori de tipul

P(Interogare|Dovezi)

corespunzător unei mulţimi de *variabile de interogare* condiţionat de valori exacte ale unor *variabile dovezi*. (În exemplul considerat, *Ud* este o variabilă de interogare, iar *Stropitoare* şi *Sezon_ploios* ar putea fi variabile dovezi).

Reţelele Bayesiene sunt suficient de flexibile pentru ca orice nod să poată servi fie ca o variabilă de interogare, fie ca o variabilă dovadă. În general, un agent primeşte valori ale variabilelor dovezi de la senzorii săi (sau în urma altor raţionamente) şi întreabă despre posibilele valori ale altor variabile astfel încât să poată decide ce acţiune trebuie întreprinsă.

Literatura de specialitate discută patru tipuri distincte de inferenţă, care poate fi realizată de reţelele Bayesiene:

- inferenţă de tip diagnostic (de la efecte la cauze);
- inferenţă cauzală (de la cauze la efecte);
- inferenţă intercauzală (între cauze ale unui efect comun);
- inferenţe mixte (reprezentând combinaţii a două sau mai multe dintre inferenţele anterioare).

Exemplu: Setând efectul *Stropitoare* la valoarea "adevărat" şi cauza *Sezon_ploios* la valoarea "fals", ne propunem să calculăm

P(Ud|Stropitoare, ¬ Sezon_ploios).

Aceasta este o inferenţă mixtă, care reprezintă o utilizare simultană a inferenţei de tip diagnostic şi a celei cauzale.

Reţelele Bayesiene s-au dovedit extrem de folositoare în realizarea mai multor tipuri de sarcini. Paragraful următor se va concentra asupra celei mai frecvente şi mai utile dintre acestea, şi anume determinarea probabilităţilor condiţionate a posteriori ale variabilelor de interogare.

6.1.2.2.1. Un algoritm pentru răspunsul la interogări

În cele ce urmează, ne propunem să stabilim un algoritm de calcul al probabilităţilor condiţionate a posteriori ale variabilelor de interogare. Algoritmul găsit va fi de tip *înlănţuire înapoi* prin faptul că pleacă de la variabila de interogare şi urmează drumurile de la acel nod până la nodurile dovezi. Datorită complicaţiilor care se pot naşte atunci când două drumuri diferite converg la acelaşi nod, algoritmul pe care îl vom prezenta şi care este preluat din [27] se referă numai la *reţele unic conectate*, cunoscute şi sub denumirea de **polyarbori**. Reamintim faptul că, în astfel de reţele, există cel mult un drum nedirecţionat între oricare două noduri ale reţelei. Algoritmii pentru reţele generale, asupra cărora nu ne vom opri în cadrul restrâns al acestei lucrări, vor folosi algoritmii referitori la polyarbori ca principală subrutină.

Fig. 6.4 prezintă o reţea generică unic conectată. În această reţea nodul X are părinţii $\mathbf{U} = U_1 \ldots U_m$ şi fiii $\mathbf{Y} = Y_1 \ldots Y_n$. Corespunzător fiecărui fiu şi fiecărui părinte a fost desenat un dreptunghi care include toţi descendenţii nodului şi toţi strămoşii lui (cu excepţia lui X). Proprietatea de *unică conectare* înseamnă că toate dreptunghiurile sunt disjuncte şi că nu există legături care să le conecteze între ele. Se presupune că X este variabila de interogare şi că există o mulţime E de variabile dovezi[26]. Se urmăreşte calcularea probabilităţii condiţionate

$$P(X \mid E)$$

În mod evident, dacă însuşi X este o variabilă dovadă din E, atunci calcularea lui $P(X \mid E)$ este banală. Vom presupune, de aceea, că X nu aparţine lui E.

Reţeaua din Fig. 6.4 este partiţionată în conformitate cu părinţii şi cu fiii variabilei de interogare X. Pentru a concepe un algoritm, va fi util să ne

[26] $E = \{E_X^-, E_X^+\} = \{U_1, \ldots, U_m, Y_1, \ldots, Y_n\}$, mulţime de variabile aleatoare discrete.

putem referi la diferite porţiuni ale dovezilor. Prima distincţie pe care o vom face este următoarea:

- E_X^+ reprezintă **suportul cauzal** pentru X - variabilele dovezi aflate "deasupra" lui X, care sunt conectate la X prin intermediul părinţilor săi.
- E_X^- reprezintă **suportul probatoriu** pentru X - variabilele dovezi aflate "dedesubtul" lui X şi care sunt conectate la X prin intermediul fiilor săi.

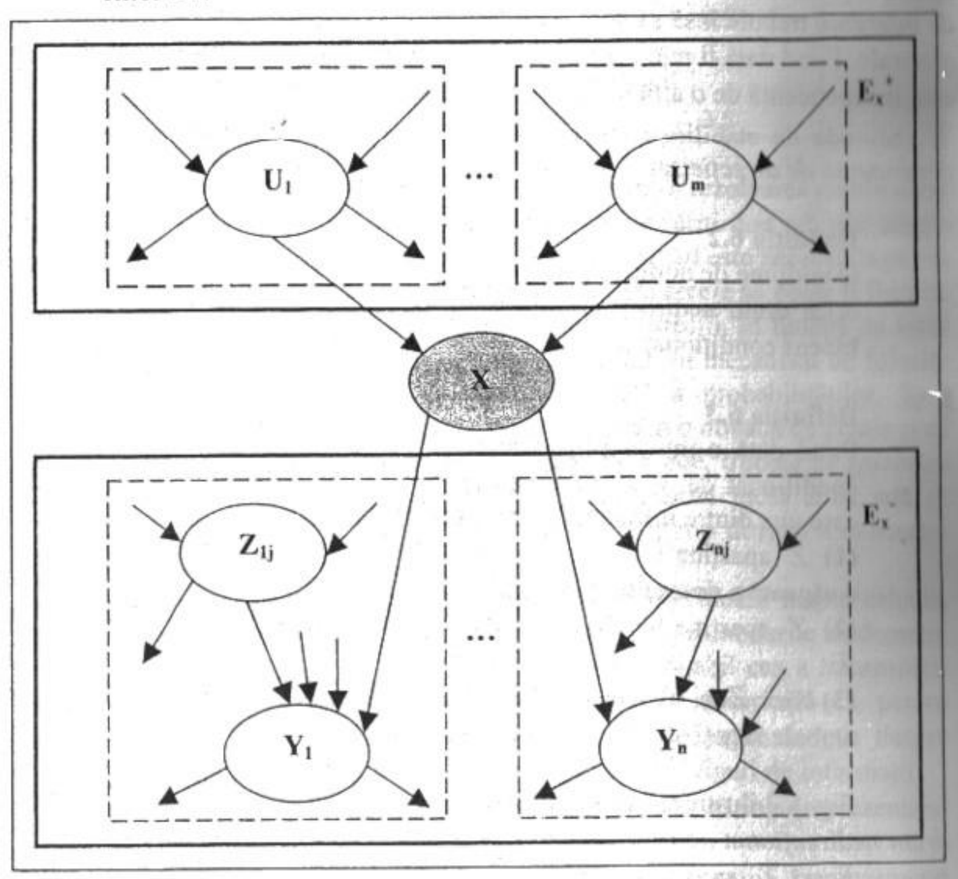


Fig. 6.4

Notaţia $E_{U_i \setminus X}$ va fi folosită pentru a se face referire la toate dovezile conectate cu nodul U_i, mai puţin cele prin drumul de la X. Cu această ultimă

notație, putem partiționa pe E_X^+ în $E_{U_1\backslash X},\ldots,E_{U_m\backslash X}$ și deci putem exprima pe E_X^+ sub forma următoare:

$$E_X^+ = \bigcup_{i=1}^{m} E_{U_i\backslash X} \qquad (1)$$

În mod similar, $E_{Y_i\backslash X}$ semnifică mulțimea tuturor dovezilor conectate la Y_i prin intermediul părinților săi, *cu excepția* lui X. Mulțimea E_X^- poate fi deci partiționată în mulțimile $E_{Y_1\backslash X},\ldots,E_{Y_n\backslash X}$ și este de forma

$$E_X^- = \bigcup_{i=1}^{n} E_{Y_i\backslash X} \qquad (2)$$

Se observă că mulțimea tuturor dovezilor E poate fi scrisă ca E_X (toate dovezile conectate la X) și ca $E_{X\backslash}$ (toate dovezile conectate la X, fără excepție).

Strategia generală pentru calculul lui $P(X \mid E)$ este atunci următoarea:

- Exprimă $P(X \mid E)$ în termenii contribuțiilor lui E_X^+ și E_X^-.
- Calculează contribuția mulțimii E_X^+ calculând efectul ei asupra părinților lui X și apoi transmițând acest efect lui X. (Calcularea efectului asupra fiecărui părinte al lui X este o secvență recursivă a problemei calculării efectului asupra lui X).
- Calculează contribuția mulțimii E_X^- calculând efectul ei asupra fiilor lui X și apoi transmițând acest efect lui X. (Calcularea efectului asupra fiecărui fiu al lui X reprezintă o secvență recursivă a problemei calculării efectului asupra lui X).

Totalitatea dovezilor, E, constă din dovezile aflate "deasupra" lui X și din cele aflate "dedesubtul" lui X, întrucât s-a făcut presupunerea că X însuși nu se află în E. Prin urmare, avem

$$P(X \mid E) = P(X \mid E_X^-, E_X^+)$$

Pentru a separa contribuțiile lui E_X^+ și E_X^-, vom aplica versiunea condiționată a regulii lui Bayes, păstrând pe E_X^- ca dovadă fixată în fundal:

$$P(X \mid E_X^-, E_X^+) = \frac{P(X \mid E_X^+)P(E_X^- \mid X, E_X^+)}{P(E_X^- \mid E_X^+)}$$

Întrucât X d-separă în cadrul rețelei pe E_X^+ de E_X^-, putem folosi independența condiționată pentru a simplifica al doilea termen al numărătorului. De asemenea, putem trata $1/P(E_X^- \mid E_X^+)$ ca pe o constantă de normalizare, obținând:

$$P(X \mid E) = \alpha\, P(X \mid E_X^+)P(E_X^- \mid X) \qquad (3)$$

Prin urmare, este necesar să calculăm cei doi termeni $P(X \mid E_X^+)$ și $P(E_X^- \mid X)$. Vom începe prin tratarea primului, care se calculează relativ ușor.

Vom calcula $P(X \mid E_X^+)$ luând în considerație toate configurațiile posibile ale părinților lui X, precum și cât de probabile sunt acestea fiind dată mulțimea E_X^+. În cazul fiecărei configurații date, probabilitatea lui X se cunoaște direct din tabelul probabilităților condiționate.

Fie $\mathbf{U}$ vectorul părinților $U_1,\ldots,U_m$ și fie $\mathbf{u}$ o atribuire de valori pentru aceștia[27]. În calculele care urmează vom folosi faptul că $E_{U_i\backslash X}$ d-separă pe U_i de toate celelalte dovezi din E_X^+. Ținând cont de acest fapt și de formula (1), obținem egalitatea:

$$P(\{U_i = u_i\} \mid E_X^+) = P(\{U_i = u_i\} \mid E_{U_i\backslash X}) \quad i = \overline{1,m} \qquad (4)$$

Luând în considerație evenimentul sigur, putem exprima probabilitatea $P(X \mid E_X^+)$ sub următoarea formă:

$$P(X \mid E_X^+) = P\left(X \bigcap \bigcup_{(u_1,\ldots,u_m)} \{U_1 = u_1,\ldots,U_m = u_m\} \mid E_X^+\right) \qquad (5)$$

Întrucât membrul drept din (5) reprezintă probabilitatea unei reuniuni de mulțimi disjuncte, avem:

$$P(X \mid E_X^+) = \sum_{(u_1,\ldots,u_m)} P(X \bigcap \{U_1 = u_1,\ldots,U_m = u_m\} \mid E_X^+) \qquad (6)$$

Aplicând, în continuare, versiunea condiționată a formulei de înmulțire a probabilităților membrului drept din (6), obținem:

---

[27] Spre exemplu, dacă există doi părinți Booleeni, U_1 și U_2, atunci $\mathbf{u}$ trece peste patru atribuiri posibile, dintre care una este [true, false].

$$P\left(X \mid E_X^+\right) = \sum_{(u_1,\ldots,u_m)} P\left(\{U_1 = u_1,\ldots,U_m = u_m\} \mid E_X^+\right) \cdot$$

$$\cdot P\left(X \mid \{U_1 = u_1,\ldots,U_m = u_m\}, E_X^+\right) \qquad (7)$$

Probabilitatea $P\left(\{U_1 = u_1,\ldots,U_m = u_m\} \mid E_X^+\right)$, care intervine în membrul drept al formulei (7), poate fi explicitată dacă se ţine cont de independenţa variabilelor aleatoare $U_1,\ldots,U_m$, precum şi de faptul că $E_{U_i \setminus X}$ d-separă pe U_i de toate celelalte dovezi din E_X^+, care a fost partiţionat în $E_{U_1 \setminus X},\ldots,E_{U_m \setminus X}$. Întrucât $U_1,\ldots,U_m$ sunt independente,

$$P\left(\{U_1 = u_1,\ldots,U_m = u_m\} \mid E_X^+\right) = \prod_{i=1}^{m} P\left(\{U_i = u_i\} \mid E_X^+\right) \qquad (8)$$

şi, ţinând cont de (1), avem:

$$\prod_{i=1}^{m} P\left(\{U_i = u_i\} \mid E_X^+\right) = \prod_{i=1}^{m} P\left(\{U_i = u_i\} \mid E_{U_i \setminus X}\right) \qquad (9)$$

Cea de-a doua probabilitate condiţionată care intervine în membrul drept al formulei (7) poate fi explicitată ţinându-se cont de faptul că U d-separă pe X de E_X^+:

$$P\left(X \mid \{U_1 = u_1,\ldots,U_m = u_m\}, E_X^+\right) = P\left(X \mid \{U_1 = u_1,\ldots,U_m = u_m\}\right) \qquad (10)$$

Ţinând cont de (7), (8), (9) şi (10), obţinem:

$$P\left(X \mid E_X^+\right) = \sum_{(u_1,\ldots,u_m)} P\left(X \mid \{U_1 = u_1,\ldots,U_m = u_m\}\right) \prod_{i=1}^{m} P\left(\{U_i = u_i\} \mid E_{U_i \setminus X}\right) \qquad (11)$$

Introducând expresia lui $P\left(X \mid E_X^+\right)$ dată de (11) în formula (3), rezultă:

$$P(X \mid E) = \alpha\, P\left(E_X^- \mid X\right) \sum_{\mathbf{u}} P(X \mid \mathbf{u}) \prod_{i=1}^{m} P\left(\{U_i = u_i\} \mid E_{U_i \setminus X}\right) \qquad (12)$$

Ecuaţia (12) sugerează deja un algoritm. Astfel, $P(X \mid \mathbf{u})$ este repartiţia lui X condiţionată de realizarea $(U_1 = u_1,\ldots,U_m = u_m)$. Valoarea acestei probabilităţi poate fi luată din tabelul de probabilităţi condiţionate asociat lui X. Calculul fiecărei probabilităţi $P\left(\{U_i = u_i\} \mid E_{U_i \setminus X}\right)$ reprezintă o secvenţă recursivă a problemei iniţiale, aceea de a calcula $P(X \mid E)$, adică $P\left(X \mid E_{X \setminus}\right)$. Vom mai nota aici faptul că mulţimea variabilelor dovezi care

intervin în apelarea recursivă reprezintă o submulţime a celor din apelarea iniţială, ceea ce constituie un indiciu că procedura de calcul se va termina într-un număr finit de paşi, ea reprezentând într-adevăr un algoritm.

În continuare ne propunem să calculăm probabilitatea $P\left(E_X^- \mid X\right)$, care intervine în formula (12), urmărind; în egală măsură, obţinerea unei soluţii recursive.

Întrucât $E_{Y_i \setminus X}$ d-separă pe Y_i de toate celelalte dovezi din E_X^-, rezultă independenţa dovezilor din fiecare "casetă Y_i" faţă de celelalte, condiţionat de X. Forma lui E_X^- este cea dată de formula (2). Datorită independenţei dovezilor de sub X avem:

$$P\left(E_X^- \mid X\right) = \prod_{i=1}^{n} P\left(E_{Y_i \setminus X} \mid X\right) \qquad (13)$$

Fie $\mathbf{Y}$ vectorul fiilor $Y_1,\ldots,Y_n$ şi fie $\mathbf{y} = (y_1,\ldots,y_n)$ o realizare a acestuia. În cele ce urmează, vom ţine cont de valorile fiilor lui $X, Y_i \left(i = \overline{1,n}\right)$, dar va trebui să includem şi părinţii fiecărui Y_i. Fie $\mathbf{Z}_i$ părinţii lui Y_i, alţii decât X şi fie $\mathbf{z}_i$ o atribuire de valori ale părinţilor. Luând în consideraţie evenimentul sigur $\bigcup_{y_i} \{Y_i = y_i\}$ şi respectiv $\bigcup_{\mathbf{z}_i} \{\mathbf{Z}_i = \mathbf{z}_i\}$, avem:

$$P\left(E_{Y_i \setminus X} \mid X\right) = P\left(E_{Y_i \setminus X} \bigcap \bigcup_{y_i} \{Y_i = y_i\} \bigcap \bigcup_{\mathbf{z}_i} \{\mathbf{Z}_i = \mathbf{z}_i\} \mid X\right) =$$

$$= \sum_{y_i} \sum_{\mathbf{z}_i} P\left(E_{Y_i \setminus X}, \{Y_i = y_i\}, \{\mathbf{Z}_i = \mathbf{z}_i\} \mid X\right)$$

Aplicând, în formula anterioară, versiunea condiţionată a formulei de înmulţire a probabilităţilor, obţinem:

$$P\left(E_{Y_i \setminus X} \mid X\right) = \sum_{y_i} \sum_{\mathbf{z}_i} P\left(\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\} \mid X\right) P\left(E_{Y_i \setminus X} \mid X, \{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}\right) \qquad (14)$$

Mulţimea dovezilor $E_{Y_i \setminus X}$ reprezintă reuniunea a două submulţimi disjuncte şi independente de variabile aleatoare:

$$E_{Y_i \setminus X} = E_{Y_i \setminus X}^+ \bigcup E_{Y_i}^-$$

Prin urmare, avem:

$$P\left(E_{Y_i\setminus X} \mid X,\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}\right) = P\left(E^+_{Y_i\setminus X} \mid X,\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}\right)\cdot$$
$$\cdot P\left(E^-_{Y_i} \mid X,\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}\right) \tag{15}$$

Dar $E^-_{Y_i}$ este o mulțime de variabile aleatoare independente de X și $\mathbf{Z}_i$, iar $E^+_{Y_i\setminus X}$ constituie o mulțime de variabile aleatoare independente de X și Y_i. În aceste condiții, repartițiile condiționate din (15) devin:

$$P\left(E_{Y_i\setminus X} \mid X,\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}\right) = P\left(E^-_{Y_i} \mid \{Y_i = y_i\}\right)\cdot P\left(E^+_{Y_i\setminus X} \mid \{\mathbf{Z}_i = \mathbf{z}_i\}\right) \tag{16}$$

Cea de-a doua repartiție condiționată din membrul drept al lui (16) apare ca o repartiție a posteriori și poate fi exprimată cu formula lui Bayes (în versiune necondiționată):

$$P\left(E^+_{Y_i\setminus X} \mid \{\mathbf{Z}_i = \mathbf{z}_i\}\right) = \frac{P\left(E^+_{Y_i\setminus X}\right)\cdot P\left(\{\mathbf{Z}_i = \mathbf{z}_i\} \mid E^+_{Y_i\setminus X}\right)}{P\left(\{\mathbf{Z}_i = \mathbf{z}_i\}\right)} \tag{17}$$

Introducând (17) în (16) obținem:

$$P\left(E_{Y_i\setminus X} \mid X,\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}\right) =$$
$$= P\left(E^-_{Y_i} \mid \{Y_i = y_i\}\right)\cdot \frac{P\left(E^+_{Y_i\setminus X}\right)P\left(\{\mathbf{Z}_i = \mathbf{z}_i\} \mid E^+_{Y_i\setminus X}\right)}{P\left(\{\mathbf{Z}_i = \mathbf{z}_i\}\right)} \tag{18}$$

Introducând acum expresia lui $P\left(E_{Y_i\setminus X} \mid X,\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}\right)$ dată de (18) în formula (14), rezultă:

$$P\left(E_{Y_i\setminus X} \mid X\right) = \sum_{y_i}\sum_{\mathbf{z}_i} P\left(\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\} \mid X\right)\cdot P\left(E^-_{Y_i} \mid \{Y_i = y_i\}\right)\cdot$$
$$\cdot \frac{P\left(E^+_{Y_i\setminus X}\right)P\left(\{\mathbf{Z}_i = \mathbf{z}_i\} \mid E^+_{Y_i\setminus X}\right)}{P\left(\{\mathbf{Z}_i = \mathbf{z}_i\}\right)} \tag{19}$$

Revenind acum la repartiția lui E^-_X condiționată de X, dată de formula (13) și luând în considerație (19), obținem:

$$P\left(E^-_X \mid X\right) = \prod_{i=1}^{n}\sum_{y_i} P\left(E^-_{Y_i} \mid \{Y_i = y_i\}\right)\sum_{\mathbf{z}_i} P\left(\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\} \mid X\right)\cdot$$
$$\cdot \frac{P\left(E^+_{Y_i\setminus X}\right)P\left(\{\mathbf{Z}_i = \mathbf{z}_i\} \mid E^+_{Y_i\setminus X}\right)}{P\left(\{\mathbf{Z}_i = \mathbf{z}_i\}\right)} \tag{20}$$

Aplicând în (20) versiunea condiționată a formulei de înmulțire a probabilităților, rezultă:

$$P\left(E^-_X \mid X\right) = \prod_{i=1}^{n}\sum_{y_i} P\left(E^-_{Y_i} \mid \{Y_i = y_i\}\right)\cdot \sum_{\mathbf{z}_i} \frac{P\left(E^+_{Y_i\setminus X}\right)P\left(\{\mathbf{Z}_i = \mathbf{z}_i\} \mid E^+_{Y_i\setminus X}\right)}{P\left(\{\mathbf{Z}_i = \mathbf{z}_i\}\right)}\cdot$$
$$\cdot P\left(\{\mathbf{Z}_i = \mathbf{z}_i\} \mid X\right)\cdot P\left(\{Y_i = y_i\} \mid X,\{\mathbf{Z}_i = \mathbf{z}_i\}\right) \tag{21}$$

Dar $\mathbf{Z}$ și X sunt d-separate, ceea ce înseamnă că $P\left(\{\mathbf{Z}_i = \mathbf{z}_i\} \mid X\right) = P\left(\{\mathbf{Z}_i = \mathbf{z}_i\}\right)$. Putem, de asemenea, înlocui pe $P\left(E^+_{Y_i\setminus X}\right)$ cu o constantă de normalizare β_i. În aceste condiții, egalitatea (21) devine:

$$P\left(E^-_X \mid X\right) = \prod_{i=1}^{n}\sum_{y_i} P\left(E^-_{Y_i} \mid \{Y_i = y_i\}\right)\sum_{\mathbf{z}_i} \beta_i P\left(\{\mathbf{Z}_i = \mathbf{z}_i\} \mid E^+_{Y_i\setminus X}\right)\cdot$$
$$\cdot P\left(\{Y_i = y_i\} \mid X,\{\mathbf{Z}_i = \mathbf{z}_i\}\right)$$

Combinând toți β_i într-o unică constantă de normalizare β și ținând cont de semnificația lui $E^+_{Y_i\setminus X}$, obținem:

$$P\left(E^-_X \mid X\right) = \beta\prod_{i}\sum_{y_i} P\left(E^-_{Y_i} \mid y_i\right)\sum_{\mathbf{z}_i} P\left(y_i \mid X,\mathbf{z}_i\right)\prod_{j} P\left(z_{ij} \mid E_{Z_{ij}\setminus Y_i}\right) \tag{22}$$

Se observă că fiecare dintre termenii expresiei finale date de (22) este ușor de evaluat:

- $P\left(E^-_{Y_i} \mid y_i\right)$ este o secvență recursivă a calculării lui $P\left(E^-_X \mid X\right)$;
- $P\left(y_i \mid X,\mathbf{z}_i\right)$ se citesc din tabelul de probabilități condiționate asociat lui Y_i;
- $P\left(z_{ij} \mid E_{Z_{ij}\setminus Y_i}\right)$ reprezintă o secvență recursivă a problemei inițiale, adică a calculării lui $P\left(X \mid E\right)$, mai precis a lui $P\left(X \mid E_{X\setminus}\right)$.

Problema transformării acestor calcule într-un *algoritm* este relativ simplă. Așa cum se arată în [27], vor fi necesare două subrutine de bază, care vor calcula $P\left(X \mid E_{X\setminus V}\right)$ și respectiv $P\left(E^-_{X\setminus V} \mid X\right)$, unde prin V am notat variabila care desemnează dovezile exceptate atunci când sunt luate în considerație cele conectate la nodul X.

În Algoritmul 6.1 subrutina SUPORT-EXCEPT(X,V) calculează $P\left(X \mid E_{X\setminus V}\right)$ folosind o generalizare a ecuației (12), în timp ce subrutina DOVEZI-EXCEPT(X,V) calculează $P\left(E^-_{X\setminus V} \mid X\right)$ folosind o generalizare a ecuației (22).

Pentru a simplifica prezentarea, se va presupune că reţeaua este fixată şi deja "aprovizionată" cu dovezi, precum şi că variabilele dovezi satisfac predicatul DOVEZI?. Probabilităţile $P(X \mid \mathbf{U})$, unde $\mathbf{U}$ reprezintă vectorul părinţilor lui X, sunt disponibile în tabelul de probabilităţi condiţionate asociat lui X. Calculul expresiilor $\alpha\ldots$ şi $\beta\ldots$ se face prin normalizare.

În general, un agent primeşte valori ale variabilelor dovezi prin intermediul senzorilor săi sau în urma unor raţionamente şi se interesează (întreabă) despre posibilele valori ale altor variabile, astfel încât să poată decide ce acţiune trebuie întreprinsă în continuare. Algoritmul 6.1, preluat de noi din [27], calculează distribuţia de probabilitate condiţionată pentru o variabilă de interogare dată. El reprezintă un algoritm de tip înlănţuire-înapoi pentru rezolvarea interogărilor probabiliste asupra unui polyarbore.

Algoritmul 6.1

function INTEROGARE-REŢEA(X) **return**

o distribuţie de probabilitate a valorilor lui X

input: X, o variabilă aleatoare

SUPORT-EXCEPT$(X,$ nul$)$

function SUPORT-EXCEPT(X,V) **return** $P(X \mid E_{X\setminus V})$

if DOVEZI ? (X) **then return** distribuţia observată pentru X

else

calculează $P(E^-_{X\setminus V} \mid X)$ = DOVEZI-EXCEPT (X,V)

$\mathbf{U} \leftarrow$ PĂRINŢI $[X]$

if $\mathbf{U}$ este vid

then return $\alpha\, P(E^-_{X\setminus V} \mid X)P(X)$

else

pentru fiecare U_i **in** U **do**

calculează şi memorează $P(U_i \mid E_{U_i\setminus X})=$

$=$SUPORT-EXCEPT (U_i, X)

return $\alpha\, P(E^-_{X\setminus V} \mid X)\sum_{\mathbf{u}} P(X \mid \mathbf{u})\prod_i P(U_i \mid E_{U_i\setminus X})$

function DOVEZI-EXCEPT (X,V) **return** $P(E^-_{X\setminus V} \mid X)$

$\mathbf{Y} \leftarrow$ FII $[X]-V$

if $\mathbf{Y}$ este vid

then return o repartiţie uniformă

else

pentru fiecare Y_i **in** $\mathbf{Y}$ **do**

calculează $P(E^-_{Y_i} \mid y_i)=$DOVEZI-EXCEPT$(Y_i,$ nul$)$

$\mathbf{Z}_i \leftarrow$ PĂRINŢI $[Y_i]-X$

pentru fiecare Z_{ij} **in** $\mathbf{Z}_i$

calculează $P(Z_{ij} \mid E_{Z_{ij}\setminus Y_i})=$

$=$SUPORT-EXCEPT(Z_{ij}, Y_i)

return $\beta \prod_i \sum_{y_i} P(E^-_{Y_i} \mid y_i)\sum_{\mathbf{z}_i} P(y_i \mid X, \mathbf{z}_i)\prod_j P(z_{ij} \mid E_{Z_{ij}\setminus Y_i})$

□

Calculele efectuate de Algoritmul 6.1 presupun apeluri recursive care se extind de la X de-a lungul tuturor drumurilor din reţea. Recursivitatea se încheie atunci când sunt atinse noduri-dovezi, noduri-rădăcină (care nu au părinţi) şi noduri-frunză (care nu au fii). Fiecare apel recursiv exclude nodul de la care s-a produs apelul, astfel încât fiecare nod din arbore este tratat o singură dată. Prin urmare, algoritmul este liniar raportat la numărul de noduri ale reţelei. Algoritmul se comportă în acest fel deoarece reţeaua dată reprezintă un *polyarbore*. Dacă ar exista mai mult de un singur drum între o pereche de noduri, atunci fie procedurile recursive ar lua în calcul aceleaşi dovezi de mai multe ori, fie execuţia nu s-ar încheia într-un număr finit de paşi.

Algoritmul prezentat, de tip înlănţuire înapoi, este cel mai simplu algoritm pentru polyarbori. Principalul neajuns al unui asemenea algoritm este acela că el calculează repartiţia de probabilitate pentru o unică variabilă. Dacă se doreşte determinarea repartiţiilor a posteriori pentru toate variabilele care nu constituie variabile dovezi, atunci programul implementând Algoritmul 6.1 ar trebui executat pentru fiecare dintre acestea, ceea ce ar mări timpul de execuţie. În acest caz este preferabilă o abordare de tip înlănţuire înainte, care pleacă de la variabilele dovezi. În cazul unei contabilizări adecvate, calculele pot fi efectuate într-un timp liniar. Versiunea algoritmului care foloseşte înlănţuirea înainte poate fi privită ca reprezentând o "propagare a mesajelor" prin reţea. Din această cauză implementarea pe calculatoare paralele este relativ simplă şi pot fi făcute analogii interesante cu propagarea mesajelor între neuronii din creier.

CAPITOLUL 7
CĂUTARE ŞI REPREZENTAREA CUNOŞTINŢELOR ÎN PROCESAREA LIMBAJULUI NATURAL. ANALIZA SINTACTICĂ

Procesarea limbajului natural reprezintă o ***tehnologie*** (adică un ansamblu de procese, metode, operaţii) care creează şi implementează modalităţi de a executa diferite sarcini referitoare la limbajul natural (cum ar fi construcţia unor interfeţe - bazate pe limbaj natural - cu baze de date, traducerea automată ş.a.). Procesarea limbajului natural reprezintă şi astăzi o problemă dificilă şi în cea mai mare parte nerezolvată. Găsirea unei tehnologii adecvate este extrem de grea datorită naturii multidisciplinare a problemei, fiind implicate diferite ştiinţe şi domenii: lingvistică, psiholingvistică, lingvistică computaţională, filozofie, informatică, în general, şi inteligenţă artificială, în mod special, etc. *Ingineria limbajului natural* se ocupă de implementarea unor sisteme de mare anvergură.

Aplicaţiile procesării limbajului natural se înscriu în trei mari categorii:

- aplicaţiile bazate pe text, dintre care amintim:
  - clasificarea documentelor (şi respectiv găsirea documentelor legate de anumite subiecte);
  - regăsirea informaţiei (căutarea unor cuvinte-cheie sau concepte);
  - extragerea informaţiei (legate de un anumit subiect, deci de un anumit cuvânt-cheie);
  - înţelegerea textelor (care presupune o analiză profundă a structurii acestora);
  - traducerea automată şi traducerea asistată de calculator dintr-o limbă în alta;
  - alcătuirea de sinteze;
  - achiziţia de cunoştinţe.
- aplicaţiile bazate pe dialog, care implică comunicarea între om şi maşină, aplicaţii cum ar fi sistemele de învăţare, sistemele de interogare şi răspuns la întrebări, rezolvarea problemelor, controlul (bazat pe limba vorbită) al unui calculator ş.a.
- procesarea vorbirii
  Este important să facem distincţia între problemele de *recunoaştere a vorbirii* şi cele de *înţelegere* a limbajului. Astfel, trebuie să remarcăm încă de la început faptul că un sistem de recunoaştere a vorbirii nu

folosește nici un element de înțelegere a limbajului. Recunoașterea vorbirii se ocupă numai de identificarea cuvintelor vorbite provenind de la un semnal dat, nu și de înțelegerea mesajului, adică a modului în care aceste cuvinte sunt folosite în procesul de comunicare. Pentru a deveni un sistem de înțelegere a limbajului, un dispozitiv de recunoaștere a vorbirii trebuie să furnizeze intrarea sa unui sistem de înțelegere a limbajului natural, operație care produce un așa-numit "sistem de înțelegere a limbajului vorbit". Caracteristica de bază a oricărui sistem de înțelegere este aceea că el realizează o reprezentare a înțelesului propozițiilor într-un limbaj de reprezentare, care poate fi utilizat în vederea unei procesări ulterioare.

Structura oricărei limbi naturale (umane) se împarte în cinci niveluri diferite: *fonologie, morfologie, sintaxă, semantică* și *pragmatică*. Aceste niveluri coincid cu formele de cunoștințe relevante și, în același timp, necesare pentru înțelegerea limbajului natural.

Sintaxa reprezintă, într-o concepție asupra organizării stratificate, pe niveluri, a limbii, acel nivel a cărui organizare se desfășoară între cuvânt, ca unitate minimală și combinațiile acestuia: propoziții/fraze, ca unități maximale. Cunoștințele sintactice se referă la modul în care pot fi alăturate cuvintele pentru a forma propoziții corecte și determină care este rolul structural al fiecărui cuvânt în cadrul propoziției, precum și ce grupuri sintactice intră în componența altori grupuri sintactice (vezi §7.1). Prin urmare, nivelul sintactic determină rolul structural al fiecărui cuvânt în interiorul unei propoziții, precum și relațiile dintre propoziții în cadrul frazei.

Sintaxa (construcția propozițiilor) reprezintă nivelul cel mai de jos la care limbajul natural (uman) este, în mod constant, creator. Vorbitorii unei limbi creează mult mai rar unități fonice și lexicale. În schimb, sunt concepute în mod constant noi propoziții și fraze. Atâta timp cât nu este impusă o limită asupra lungimii propozițiilor, se poate demonstra că numărul structurilor de propoziție admisibile este infinit.

Noam Chomsky (1957) este primul care evidențiază această idee. El introduce *gramatica generativă*, care descrie propozițiile furnizând reguli de construcție a lor (vezi §7.2). Astfel de reguli vor deveni standard nu numai în lingvistică, ci și în informatică, cu precădere în *proiectarea compilatoarelor*. Ceea ce este interesant și, în egală măsură, crucial, în cadrul acestei teorii, este faptul că o mulțime finită de reguli poate descrie un număr infinit de propoziții.

Procesul de recunoaștere a structurii unei propoziții de către un calculator se numește ***parsing***. Aceasta este *analiza sintactică computațională*, la care ne vom referi în capitolul de față al acestei lucrări. În esență, vom spune că, pentru a analiza sintactic o propoziție, un calculator trebuie să o împerecheze pe aceasta cu regulile care o generează. Acest proces de împerechere poate fi

realizat în manieră *top-down* (de sus în jos) sau *bottom-up* (de jos în sus). Capitolul de față se va referi în detaliu la ambele tipuri de tehnici, precum și la unele combinații ale lor care și-au dovedit eficiența.

Așa cum se remarcă în [5], analiza sintactică de acest tip a propozițiilor englezești a fost studiată pe larg și este practic considerată o problemă încheiată. Astăzi se poartă discuții exclusiv asupra modului *cel mai eficient* în care se poate realiza procesul de parsing în cazul limbii engleze și nu referitor la faptul că acest lucru este sau nu posibil. Tocmai de aceea, capitolul de față realizează o prezentare amănunțită a principalelor tehnici de parsing existente. Analiza sintactică computațională referitoare la alte limbi nu a fost investigată la fel de riguros și ea rămâne, în multe cazuri, o problemă deschisă. Cu atât mai mult cu cât majoritatea tehnicilor de parsing existente se bazează pe ordinea fixată a cuvintelor și nu lucrează la fel de eficient pentru limbile în care ordinea cuvintelor este extrem de variabilă, cum ar fi latina, rusa sau finlandeza.

O operație esențială în domeniul procesării limbajului natural este, prin urmare, aceea de a atribui structuri sintactice acelor propoziții care sunt considerate admisibile sau corecte în conformitate cu regulile unei gramatici date. Este un fapt unanim acceptat acela că obiectele lingvistice sunt niște obiecte structurate, care însă nu își dezvăluie structura în mod evident. *Înțelegerea sensului* unei propoziții depinde în mod crucial de abilitatea avută în regăsirea *structurii* acesteia. În cazul vorbitorului nativ al unei limbi acest lucru se produce cel mai adesea în mod inconștient. Un *dispozitiv computațional* care deduce structura pornind de la șiruri de cuvinte corecte din punct de vedere gramatical se numește ***analizor sintactic***.

Pentru *a formaliza studiul structurii sintactice a unei propoziții*, sunt necesare două elemente: **gramatica** - o construcție formală specifică unei structuri lingvistice - și **tehnicile de analiză sintactică**, care determină corectitudinea unei propoziții în conformitate cu regulile gramaticii.

Tehnicile de analiză sintactică sunt deosebit de variate și diferă în funcție de tipul de gramatică utilizat. Majoritatea algoritmilor existenți utilizează însă gramaticile PS independente de context (vezi §7.2) și, prin urmare, efectuează o **analiză sintactică bazată pe constituenți**.

7.1. Noțiunea de grup sintactic

După cum am arătat deja, cunoștințele sintactice se referă la modul în care pot fi alăturate cuvintele pentru a forma propoziții corecte și determină care este rolul structural al fiecărui cuvânt în cadrul propoziției, precum și ce grupuri sintactice intră în componența altor grupuri sintactice.

În sintaxa modernă, prin *grup* în general se înțelege un component al structurii propoziției constituit în jurul unui centru (sau cap) de grup: verb, nume (substantiv), adjectiv, adverb, prepoziție, grup a cărui coeziune sintactico-

semantică este asigurată prin constrângeri de formă gramaticală (de caz, de prepoziție, de topică, de acord), dar și prin rolurile semantico-tematice impuse de centru determinanților.

În cele ce urmează, vom opera cu noțiunea de **grup sintactic** în general și, în particular, cu noțiuni ca grup nominal, grup verbal, grup prepozițional ș.a. și vom folosi abrevierile englezești consacrate. Astfel, în funcție de *clasa morfologică a centrului*, se disting:

- *grupul verbal*, care va fi notat VP (după engl. *verb phrase*);
- *grupul nominal*, care va fi notat NP (după engl. *noun phrase*);
- *grupul adjectival*, care va fi notat ADJP (după engl. *adjectival phrase*);
- *grupul adverbial*, care va fi notat ADVP (după engl. *adverbial phrase*);
- *grupul prepozițional*, care va fi notat PP (după engl. *prepositional phrase*).

Spre exemplu, în limba română, în jurul verbului *a plăcea*, se poate constitui un grup verbal (VP) de forma

îmi place cartea

în structura căruia centrul atrage doi actanți (sau două argumente), iar coeziunea se asigură prin restricțiile de caz și de topică impuse acestor argumente (dativul antepus și nominativul postpus), dar și prin atribuirea funcțiilor de obiect indirect și de subiect.

În terminologia românească, termenul *grup* este preferat celui de frază (engl. *phrase*) pentru a evita confuzia cu accepția tradițională a termenului *frază*. În gramatica engleză termenul *phrase* desemnează o îmbinare de cuvinte, fără subiect sau predicat, care funcționează ca o singură parte de vorbire. Unii autori utilizează și termenul de *categorie sintactică* pentru a desemna aceste grupuri de cuvinte. Vom reveni asupra acestor noțiuni în §7.2, atunci când vom discuta despre gramatici și respectiv despre categorii (gramaticale și lexicale).

Grupurile sintactice sunt folosite în *reprezentarea structurii sintactice* a unei propoziții.

Structura sintactică a propoziției indică modul în care cuvintele din propoziție sunt legate unele de altele. Ea arată cum sunt grupate cuvintele, cum anumite cuvinte modifică pe altele și care dintre cuvinte sunt de importanță centrală în cadrul propoziției. În plus, sintaxa poate identifica tipurile de relații care există între grupurile sintactice și poate reține diverse informații despre structura propoziției, care vor fi necesare în prelucrarea ulterioară.

Majoritatea *reprezentărilor sintactice* ale limbajului sunt bazate pe noțiunea de *gramatici independente de context*. Acestea reprezintă structura propoziției în termeni de *grupuri sintactice*, care, la rândul lor, sunt părți componente ale altor grupuri sintactice. Informația de acest tip este adesea

prezentată sub formă de *arbore*. Iată, spre exemplu, reprezentarea sintactică a propoziției

O minge este roșie.

în care, pe lângă abrevierile introduse deja corespunzător grupurilor sintactice, mai intervin următoarele notații: S pentru grupul sintactic care reprezintă întreaga propoziție (de la engl. *sentence*), ART pentru articol, N pentru substantiv (de la engl. *noun*) și V pentru verb:

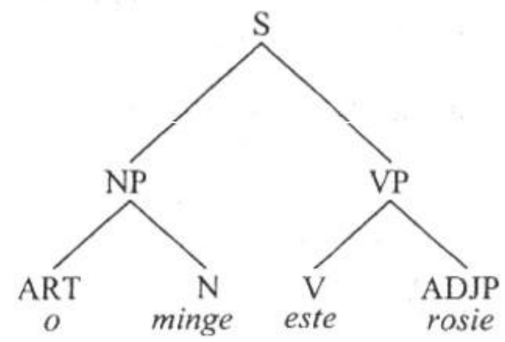


Fig. 7.1

7.2. Gramatici

O **gramatică** reprezintă o specificare formală a regulilor ce definesc structurile "legale" (admise) ale unui limbaj.

Întrucât nici un limbaj natural nu este finit, se simte nevoia existenței unor sisteme formale (matematice) care să definească, în mod riguros, apartenența mulțimilor infinite de expresii lingvistice și care să atribuie o structură fiecărui element al acestor mulțimi. Astfel de sisteme formale sunt gramaticile. O *gramatică* poate fi, prin urmare, privită ca reprezentând *o definiție abstractă a unei mulțimi de obiecte structurate corect formate*.

Gramaticile vor fi gândite ca gestionând două procese total opuse: acela de **generare** și acela de **recunoaștere**. Astfel, o gramatică poate genera propoziții și fraze. Dar ea poate fi utilizată și pentru a recunoaște o propoziție dată. Un dispozitiv de recunoaștere decide dacă o propoziție dată aparține unui limbaj. Cu alte cuvinte, recunoaște dacă propoziția poate fi generată de gramatica corespunzătoare și, prin urmare, decide dacă propoziția este corectă în conformitate cu gramatica dată.

Gramaticile pe care le vom prezenta, în cele ce urmează, vor utiliza *recursivitatea* pentru a gestiona complexitatea structurală. Gramaticile discutate

vor fi *declarative* şi se vor baza, în majoritatea cazurilor, pe o descompunere a grupurilor sintactice în componente denumite *caracteristici*.

Din punctul de vedere al procesării limbajului natural, studiul gramaticilor este în general privit ca fiind ***o ramură a reprezentării cunoştinţelor***. Astfel, o gramatică poate fi pur şi simplu privită ca o modalitate de a reprezenta anumite aspecte legate de ceea ce se cunoaşte referitor la un limbaj, care este suficient de formal şi de explicit pentru a fi înţeles de către o maşină.

Gramaticile utilizate în domeniul procesării limbajului natural trebuie să satisfacă atât criterii lingvistice, cât şi computaţionale. În esenţă, ele definesc mulţimi de şiruri de caractere şi asociază acestora anumite *structuri*. Pentru reprezentarea structurii sintactice sunt, în general, utilizate diagramele care folosesc structuri arborescente.

Numeroase formalisme au fost alese, de-a lungul anilor, pentru a construi gramatici care să poată fi utilizate cu succes de către diverşi analizori sintactici (parsers). Dintre numeroasele criterii generale care stau la baza proiectării unei gramatici, reamintim:

- naturaleţea lingvistică;
- puterea matematică;
- eficienţa computaţională.

Toate tipurile de gramatici care au fost utilizate de lingvistica computaţională au folosit, într-o formă sau alta, următoarele elemente:

- o reprezentare pentru grupurile sintactice (respectiv pentru părţile de vorbire care alcătuiesc aceste grupuri);
- un tip de dată pentru cuvinte (şi, prin urmare, un lexicon asociat, un dicţionar asociat sau o listă de cuvinte asociată);
- un tip de dată pentru reguli sintactice;
- un tip de dată pentru structuri sintactice.

Prin urmare, un formalism complet de specificare a unei gramatici trebuie să furnizeze cel puţin:

- un limbaj pentru specificarea grupurilor sintactice;
- un limbaj pentru reprezentarea intrărilor lexicale;
- un limbaj în care să poată fi scrise reguli (în general de mai multe tipuri);
- un limbaj de prezentare a structurilor sintactice.

Aceste limbaje pot fi distincte, dar este posibil şi ca două sau mai multe dintre ele să se suprapună.

Formalismele DCG[28] şi PATR[29] de reprezentare a gramaticilor, care vor fi utilizate, cu precădere, în cele ce urmează, au rolul de a reprezenta în special aşa-numitele **"phrase structure grammars"** (gramatici ale structurii grupului sintactic sau gramatici de structură sintagmatică) independente de context.

7.2.1. Noţiunea de constituent

Structura unei propoziţii, dată de constituenţii acesteia, reprezintă conceptul central al sintaxei.

Definiţia 7.1
i) Un **şir** este orice secvenţă de două sau mai multe elemente adiacente.
ii) Un **constituent** este un şir care are o *coeziune internă*.

Este sarcina gramaticii să atribuie o analiză sintactică oricărei propoziţii, adică să reprezinte structura sintactică a acesteia, prin care înţelegem structura propoziţiei dată de constituenţii săi.

În ceea ce priveşte analiza sintactică, trebuie să remarcăm, încă de la început, că a şti cum să segmentăm o propoziţie dată şi a construi un sistem de reguli (generative) care să atribuie în mod explicit propoziţiilor o structură de constituenţi corectă sunt două lucruri complet diferite. Domeniul procesării limbajului natural utilizează, cu precădere, această a doua accepţie a termenului de **analiză sintactică**.

---

[28] DCG este un formalism în care gramaticile sunt definite prin clauze. Caracteristica principală a acestui formalism, introdus de către Pereira şi Warren (1980), este aceea că el îmbogăţeşte regulile gramaticilor PS independente de context cu argumente şi variabile. Multe implementări Prolog oferă extensia notaţiei numită DCG (de la englezescul "definite clause grammars"). Clauzele care definesc gramatica sunt considerate definite deoarece nu reprezintă întrebări, ci clauze cu cap. Ele reprezintă, pur şi simplu, reguli şi fapte Prolog. O gramatică enunţată în DCG este executabilă în mod direct de către Prolog ca reprezentând un analizor sintactic. (a se vedea şi § 7.2.4.1).

[29] PATR este un formalism în care gramatica include caracteristici. Acest formalism se datorează lui Stuart Shieber şi asociaţilor săi de la SRI International. El reprezintă un formalism tipic aşa-numitelor gramatici de unificare. Este suficient de expresiv pentru ca tot ceea ce se reprezintă prin intermediul unui alt formalism să poată fi tradus într-un formalism echivalent de tip PATR, ceea ce a dus la o utilizare a lui pe scară largă în domeniul procesării limbajului natural. PATR este un acronim pentru "parse and translate". Caracteristica principală a formalismului PATR este aceea că el combină regulile gramaticilor PS independente de context cu un puternic sistem de caracteristici. (A se vedea şi § 7.2.4.2).

Deşi se pot construi multe sisteme diferite de reguli gramaticale care să determine o anumită analiză a structurii de constituenţi a propoziţiilor pe care le generează, ne vom limita la considerarea sistemelor de rescriere concatenative, introduse de Chomsky. Aceste sisteme vor fi numite **gramatici de structură sintagmatică**.

Pentru a determina structura unei propoziţii prin constituenţii acesteia, trebuie să se indice: care dintre şirurile propoziţiei analizate reprezintă constituenţi şi ce fel de constituent este fiecare, deci cărei *categorii* îi aparţine. Astfel, **vocabularul unei gramatici** este alcătuit din *categorii, formative* şi *caracteristici*, aşa cum vom explicita în cele ce urmează.

Categoriile se împart în *categorii gramaticale* şi *categorii lexicale*.

Categoriile gramaticale corespund **grupurilor sintactice** sau unor unităţi cu caracter mai larg decât acestea. Exemple de categorii gramaticale sunt:

- **S** (de la engl. "sentence") - reprezintă simbolul iniţial al unei gramatici;
- **NP** (de la engl. "noun phrase") - reprezintă un grup sintactic al cărui singur element obligatoriu este un substantiv (grup substantival);
- **VP** (de la engl. "verb phrase") - reprezintă un grup sintactic al cărui element principal obligatoriu este un verb (grup verbal);
- **AP** sau **ADJP** (de la engl. "adjectival phrase") - reprezintă un grup sintactic al cărui singur element obligatoriu este un adjectiv (grup adjectival);
- **PP** (de la engl. "prepositional phrase") - reprezintă un grup sintactic al cărui singur element obligatoriu este o prepoziţie (grup prepoziţional);
- **AvP** sau **ADVP** (de la engl. "adverb phrase") - reprezintă un grup sintactic al cărui singur element obligatoriu este un adverb.

Categoriile lexicale corespund **părţilor de vorbire.** Ele poartă această denumire deoarece membrii acestor categorii figurează în *lexicon*: N (de la engl. "noun") -substantiv, V (de la engl. "verb") - verb, A (de la engl. "adjective") - adjectiv, Av (de la engl. "adverb") - adverb, P (de la engl. "preposition") - prepoziţie, Det (de la engl. "determiner") - determinant (prin care se desemnează, în general, cuvinte precum articolele ori numeralele, care apar în faţa substantivelor comune).

Categoriile sunt definite strict formal, adică numai din punctul de vedere al felului în care funcţionează în cadrul regulilor.

Formativele sunt elemente minimale care au funcţie sintactică. Ca şi categoriile, ele pot fi *lexicale* şi *gramaticale*.

Formativele lexicale includ elemente lexicale care figurează în lexicon ca reprezentând diferite părţi de vorbire: $fată_N$, $băiat_N$, $pentru_P$, $frumos_A$ etc.

(Precizăm faptul că, în accepţia de aici, termenul de *lexicon* este sinonim cu cel de *vocabular*).

Formativele gramaticale sunt elemente menţionate în mod individual în anumite reguli ale gramaticii. Ele se mai numesc şi *cuvinte gramaticale* ("grammatical words"). În cazul limbii engleze se pot da ca exemple: *by*, care introduce agentul unei construcţii pasive (*It was brought by him)* sau *there* ca subiect formal (*There is no one here*). În propoziţia românească

Cartea este scrisă de către Ion.

de către reprezintă o prepoziţie compusă cu ajutorul căreia se exprimă complementul de agent (persoana care realizează o acţiune). Fiecare dintre prepoziţiile *de* şi *către* luate în consideraţie separat reprezintă formative lexicale, în timp ce prepoziţia compusă *de către* are funcţiune de *formativ gramatical*.

Caracteristicile, asupra cărora vom reveni, exprimă proprietăţi ale categoriilor lexicale şi pot fi fonologice, semantice sau sintactice. Fiecare *specificaţie a unei caracteristici* constă dintr-o caracteristică, spre exemplu *caz* şi din o valoare a acelei caracteristici, spre exemplu *acuzativ*. Distincţia care se face între categorii şi caracteristici se bazează pe opoziţia dintre funcţia formativă a categoriilor care figurează în regulile gramaticii şi funcţia de caracterizare pe care o au caracteristicile cu privire la elemente date ale limbajului.

Operaţia care construieşte obiecte complexe pornind de la cele elementare este **concatenarea**. Dându-se două obiecte, x şi y, prin concatenare se poate construi fie obiectul xy, fie obiectul yx. **Regulile** prin care se formează aceste obiecte sunt numite, în literatura anglo-saxonă de specialitate, "**phrase structure rules**" (în accepţia prezentă reguli ale structurii grupului sintactic sau, într-un sens mai larg, **reguli de structură sintagmatică**). Denumirea atribuită acestor reguli se bazează pe **relaţiile sintagmatice** (sau **de succesiune**) care se stabilesc între cuvinte. În contextul de faţă, aceste reguli indică structura grupurilor sintactice dată de constituenţii lor. O regulă de tipul $S \rightarrow NP\ VP$ afirmă că orice propoziţie conţine un grup nominal (NP) şi un grup verbal (VP), în această ordine. În cele ce urmează, vom numi regulile de structură sintagmatică sau de tip "phrase-structure" **reguli PS**.

7.2.2. Reguli PS şi gramatici PS. Gramatici generative

Utilizând reguli PS o gramatică PS poate genera propoziţii cărora le atribuie o anumită structură dată prin constituenţi şi, prin urmare, o anumită

analiză sintactică. Determinarea structurii propoziţiei facilitează înţelegerea sensului ei (*interpretarea semantică*).

Iată câteva dintre cele mai importante **proprietăţi ale regulilor PS** ca **reguli de structură**:

1) Regulile PS prezintă **structura** prin specificarea constituenţilor obligatorii şi a celor opţionali. Spre exemplu, în cazul limbii engleze, s-a constatat că cele mai frecvente reguli PS referitoare la grupul verbal sunt:

$VP \rightarrow V\ NP$

$VP \rightarrow V$

$VP \rightarrow V\ PP$

$VP \rightarrow V\ NP\ PP$

Acestea reprezintă instanţieri ale unei reguli PS generale, care spune că verbul reprezintă un constituent obligatoriu, în timp ce grupul nominal şi grupul prepoziţional reprezintă constituenţi opţionali, regulă PS de forma:

$VP \rightarrow V\ (NP)\ (PP)$

(În această notaţie parantezele rotunde desemnează prezenţa facultativă a constituenţilor incluşi între paranteze).

2) Regulile PS reprezintă aşa-numite **reguli de rescriere**, întrucât ele înlocuiesc categoria pe care o analizează prin constituenţii săi, care sunt concatenaţi. Astfel, spre exemplu, regula PS

$S \rightarrow NP\ VP$

reprezintă instrucţiunea de **a rescrie** (sau de a înlocui) simbolul categorial S prin succesiunea de categorii NP VP.

3) Regulile PS sunt **reguli independente de context**. Aceasta înseamnă că, în cazul existenţei unei reguli PS de forma $A \rightarrow Z$, rescrierea lui A ca fiind Z nu depinde de contextul în care intervine A.

4) Regulile PS **nu sunt ordonate**. Categoriile pot fi rescrise în orice moment al analizei unei propoziţii în care ele intervin. Totuşi, la un anumit moment, poate fi analizată o singură categorie.

5) Regulile PS furnizează **proceduri de decizie** referitoare la şirurile de cuvinte în sensul că ele selectează, din totalitatea şirurilor construite cu elemente ale lexiconului, pe acelea care reprezintă constituenţi de un anumit tip.

O **derivare** este o secvenţă de şiruri de simboluri, fiecare dintre acestea fiind obţinut din anteriorul, prin aplicarea unei reguli a gramaticii. Derivările încep cu simbolul iniţial. Ordinea în care se aplică regulile nu joacă

nici un rol. Prin urmare, pot exista mai multe derivări echivalente ale aceleiaşi propoziţii.

Fiecare dintre şirurile aparţinând unei derivări reprezintă propoziţia analizată într-un mod care este relevant pentru proprietăţile formale şi *sensul* propoziţiei. Analiza derivării poate arăta, spre exemplu, dacă un şir dat reprezintă un constituent şi, în caz afirmativ, care este tipul acestuia.

Definiţia 7.2
O gramatică care se bazează pe reguli PS va fi numită **gramatică PS**. Ea este o gramatică dată de o mulţime *finită* de reguli care operează asupra *categoriilor* şi *lexiconului* şi generează un *limbaj*, adică o mulţime *infinită* de propoziţii.

Preluând terminologia existentă în studiul gramaticilor formale vom nota că simbolurile gramaticii care nu mai pot fi descompuse în continuare se numesc *simboluri terminale*. În cazul limbilor naturale simbolurile terminale sunt chiar cuvintele unei limbi. Celelalte simboluri, cum ar fi NP, VP, S etc. se numesc *simboluri neterminale*. Simbolurile gramaticale cum ar fi N sau V, care se referă la părţi de vorbire, se numesc *simboluri lexicale*. Se spune că o astfel de gramatică *derivă* o propoziţie dacă există o secvenţă de reguli care permit rescrierea simbolului iniţial sub forma întregii propoziţii. Două procese extrem de importante se bazează pe derivare: unul dintre acestea este *generarea* propoziţiilor corecte şi deci a limbajului, celălalt constă în identificarea structurii propoziţiilor (*parsing*).

Întrucât regulile PS sunt independente de context, gramaticile PS reprezintă gramatici independente de context şi ele generează **limbaje independente de context**.

Vom mai nota faptul că gramaticile PS pot conţine *reguli recursive*.

Definiţia 7.3
Un simbol neterminal (S, NP etc.) este recursiv dacă poate constitui rădăcina unui arbore sau subarbore care îl conţine.

O astfel de situaţie se întâlneşte în următorul exemplu:

[Profesorul, [care cunoaşte limba engleză]$_S$]$_{NP}$, a tradus textul.

Arborele de derivare corespunzător este cel din Fig. 7.2. Întrucât însuşi S este un simbol recursiv, aşa cum se vede şi în acest exemplu, o gramatică PS are capacitatea de a genera propoziţii şi fraze de o mare complexitate.

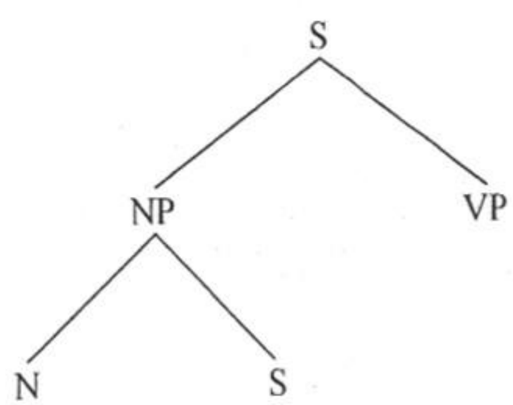


Fig. 7.2

Există şi *neajunsuri* ale analizei sintactice bazate pe gramatici PS. Sunt cunoscute, în aproape toate limbile, diferite tipuri de construcţii a căror structură dată prin constituenţi nu poate fi în mod corect şi riguros pusă în evidenţă cu ajutorul acestui tip de gamatici. Cazul *constituenţilor discontinui*[30], la care ne-am referit anterior, reprezintă o astfel de situaţie. Există, de asemenea, cazuri de *ambiguitate* sau de *propoziţii eliptice*, care nu pot fi tratate în mod satisfăcător de riguros de către gramaticile PS. Astfel, în cazul propoziţiei eliptice

Profesorul s-a bazat întotdeauna pe student şi studentul pe el.

este dificil de atribuit o analiză sintactică de tip PS şirului "*studentul pe el*". Pentru a justifica utilizarea prepoziţiei *pe* în acest context, este necesar să ne raportăm la un nivel mai abstract al analizei, în care există formularea completă *studentul s-a bazat întotdeauna pe el*. Tocmai de aceea s-au dezvoltat, de-a lungul anilor, o multitudine de teorii lingvistice diferite asupra analizei

[30] Grupurile verbale sunt adesea şi în foarte multe limbi discontinue, iar numeroase teorii gramaticale consideră discontinuitatea grupurilor verbale ca reprezentând un obiect de studiu major (Emonds, 1976, 1979; Hepple, 1990; Hoekstra, 1990; McCawley, 1982; Reape, 1990; Ross, 1973). *Constituenţii discontinui* în general sunt suficient de frecvenţi pentru a suscita interesul crescând al cercetătorilor. Problema descrierii structurilor alcătuite din elemente neadiacente prezintă numeroase dificultăţi şi încercările de rezolvare a ei au dat naştere unor tehnici de analiză sintactică specifice "Gramaticilor PS discontinue" şi diferite de cele clasice (a căror prezentare se va face în lucrarea de faţă). Ideea centrală, în acest caz, este aceea de a privi constituenţii nu ca pe nişte succesiuni de cuvinte ci ca pe *mulţimi* de cuvinte care au anumite relaţii de precedenţă între ele, dar şi faţă de alte cuvinte. În elaborarea acestei idei este fundamentală noţiunea de *arbore discontinuu*, care este o structură de tip arbore ce descrie discontinuitatea constituenţilor.

sintactice şi a tipurilor de gramatici. Unele dintre acestea concep un model de gramatică care nu se bazează pe noţiunea de constituent, ci pe relaţiile directe existente între cuvinte, indiferent dacă prin acestea înţelegem funcţii gramaticale, ori legături care organizează cuvintele în unităţi mai largi (cum ar fi grupurile sintactice).

Revenind la gramaticile PS vom observa că ele reprezintă cadrul mai larg în care ne-am propus să ne referim la **gramaticile generative**.

Este cunoscut faptul că obiectivul unei sintaxe tradiţionale îl constituie descompunerea frazelor în propoziţii componente şi descompunerea propoziţiilor în părţi de propoziţie. Pentru acest motiv, sintaxa tradiţională poate fi considerată o **sintaxă analitică**.

Gramaticile generative sunt modele lingvistice propuse de N. Chomsky în lucrările din perioada 1957-1965, ca reacţie în primul rând la aspectul analitic excesiv existent până la acea oră. Gramaticile generative reprezintă modele formalizate de tip **sintetic**, concepute ca mecanisme formale capabile să capteze şi să explice procesele creative ale limbii. În procesul de adecvare progresivă a modelului generativ la descrierea lingvistică, N. Chomsky a propus, succesiv, trei tipuri de modele generative: modelul gramaticii cu un număr finit de stări; al gramaticii de constituenţi imediaţi; al gramaticii generativ-transformaţionale.

Automatele cu număr finit de stări reprezintă cele mai simple sintaxe generative. Maşina memorează un număr de stări şi este programată să treacă de la o stare la alta, într-o ordine dată, fiecare tranziţie având ca efect producerea unui cuvânt. Orice traseu parcurs de maşină reprezintă, de fapt, *procesul de construcţie a unei propoziţii*. O propoziţie este o secvenţă de tranziţii în care ultima tranziţie are ca urmare atingerea unei stări finale.

Ultimul model prezentat de N. Chomsky în 1965 este considerat ca având puterea generativă, descriptivă şi explicativă cea mai mare şi este desemnat astăzi ca reprezentând **varianta standard a gramaticii generative sau varianta generativ-transformaţională**. Obiectivul fundamental al acestui model a fost prezentarea organizării şi a funcţionării aşa-numitei **competenţe lingvistice**. Astfel, cunoaşterea unui număr limitat de simboluri şi de reguli de combinare a acestora reprezintă tocmai abilitatea (sau mulţimea abilităţilor) de a folosi o limbă şi se numeşte **competenţă.** Actele concrete de producere a propoziţiilor constituie domeniul teoretic nelimitat al **performanţei**.

Obiectivul modelului din 1965 a fost, prin urmare, prezentarea organizării şi a funcţionării competenţei lingvistice, precum şi încercarea de a oferi explicaţii pentru fiecare dintre trăsăturile acesteia: pentru "creativitatea" competenţei; pentru capacitatea de a distinge echivalenţele a două sau mai multe fraze dintr-o limbă, precum şi de a surprinde ambiguităţile; pentru capacitatea de a distinge frazele *bine-formate* de cele *rău-formate*. În esenţă, modelul este conceput ca un mecanism formal de tip generativ, care, prin *regulile recursive*

de structură (reguli PS), produce infinitatea propozițiilor și a frazelor corecte dintr-o limbă.

O *variantă a gramaticii generative* formulată de N. Chomsky după 1980 este cea dată de așa-numita teorie "Government and Binding" (guvernare și legare), care aduce modificări numeroase și substanțiale în raport cu varianta standard. Aceasta din urmă rămâne însă cadrul general în care ne vom plasa, în cele ce urmează.

7.2.3. Clase de gramatici și limbaje

Formalismele de reprezentare a gramaticilor bazate pe reguli de rescriere pot fi comparate în funcție de **capacitatea** lor **generativă**, reprezentată prin gama de limbaje pe care o poate descrie fiecare formalism în parte. O gramatică formală este un sistem pentru rescrierea șirurilor peste un alfabet V. În contextul actual suntem interesați exclusiv de studiul limbilor naturale, dar se pare că nici un limbaj natural nu poate fi caracterizat în mod suficient de exact pentru a se defini capacitatea lui generativă. Limbajele formale, spre deosebire de limbile naturale, permit o descriere matematică exactă. Cele mai importante rezultate în studiul limbajelor formale sunt cele datorate lui N. Chomsky, începând din anul 1956.

Chomsky identifică patru clase de gramatici:

Clasa 0: gramatici PS nerestricționate

În cadrul acestei clase nu există restricții asupra tipurilor de reguli și, prin urmare, sunt permise reguli de rescriere arbitrare.

Clasa 1: gramatici dependente de context (sau senzitive de context)

Regulile de rescriere sunt de forma

$$x \rightarrow y$$

unde x și y sunt secvențe de simboluri terminale și/sau neterminale. Lungimea lui y este mai mare sau egală cu lungimea lui x. Precizăm că, în cazul limbilor naturale, *simbolurile terminale* sunt chiar cuvintele unei anumite limbi sau - dacă se dorește și integrarea morfologiei - morfeme, iar *simbolurile neterminale* sunt cele de tipul S, NP, VP etc.

Clasa 2: gramatici independente de context

Regulile de rescriere sunt de forma

$$A \rightarrow x$$

unde: A este un simbol neterminal;
x este o secvență de simboluri terminale și/sau neterminale.

Clasa 3: gramatici regulate

Regulile de rescriere sunt de forma

$$A \rightarrow tB$$

sau

$$A \rightarrow t$$

unde: A, B sunt simboluri neterminale;
t este simbol terminal.

Observația 7.1
Cu cât clasa are un număr de ordine mai mare, cu atât ea este mai restrictivă. Întrucât limbajele generate de gramaticile regulate reprezintă o submulțime a celor generate de gramaticile independente de context, care constituie o submulțime a celor generate de gramaticile dependente de context, care, la rândul lor, formează o submulțime a celor generate de gramaticile de tipul 0, ele constituie o ierarhie de limbaje, numită **ierarhia Chomsky**.

Dintre limbajele ce constituie ierarhia Chomsky, un interes deosebit suscită cele *independente de context*, întrucât s-a arătat că *aproape toate construcțiile din aproape toate limbile naturale pot fi analizate folosind tehnici care limitează sistemul la analiza limbajelor de acest tip*. Limbajele independente de context sunt cele care pot fi descrise prin intermediul unei gramatici PS independente de context. Formalismul oferit de gramaticile independente de context este suficient de puternic pentru a descrie majoritatea structurilor care apar în limbile naturale, dar, în același timp, este suficient de restricționat pentru a permite construcția unor analizori sintactici eficienți.

Gramaticile independente de context sunt relativ simplu de scris și de modificat. Există deja o tradiție în ceea ce privește tehnicile de traducere, compilare și analiză sintactică (parsing) legate de aceste gramatici. Ele sunt considerate deosebit de importante datorită existenței unor algoritmi considerabil de eficienți pentru recunoașterea și analiza sintactică a limbajelor independente de context. Tehnici de parsing suficient de evoluate au fost deja puse la punct relativ la aceste limbaje, iar parsingul independent de context rămâne unul dintre instrumentele de bază ale procesării limbajului natural. În prezent se cunoaște faptul că majoritatea covârșitoare a structurilor oricărui limbaj natural pot fi analizate sintactic în mod eficient utilizând tehnici de parsing independente de context. Aceste tehnici vor fi descrise pe larg în §7.3 al lucrării de față.

7.2.4. Reprezentarea gramaticilor în Prolog

Caracterul regulilor utilizate de o gramatică variază considerabil în funcţie de tipul acesteia, adică de teoria referitoare la gramatici care se implementează. Tipul de reguli cel mai frecvent folosit în procesarea limbajului natural şi în lingvistica computaţională rămâne însă cel al regulilor PS. În cele ce urmează, ne vom concentra atenţia asupra reprezentării în Prolog a gramaticilor PS independente de context, care utilizează o mulţime finită de categorii gramaticale şi o mulţime finită de reguli pentru a specifica modul în care categoriile din membrul stâng pot fi realizate ca o succesiune de elemente ale membrului drept.

În scrierea unei gramatici, pe lângă specificarea *categoriilor gramaticale* şi a *regulilor sau producţiilor*, este necesară şi furnizarea unui *lexicon*, care să arate *modul de realizare a unor categorii gramaticale şi, mai ales, lexicale*. Conform unei definiţii minimale, un **lexicon** reprezintă o listă de cuvinte care asociază fiecărui cuvânt o serie de proprietăţi sintactice şi morfologice. Cea mai importantă dintre acestea este dată de categoria căreia îi aparţine cuvântul. În plus, un lexicon mai poate conţine informaţii referitoare la subcategoria unui cuvânt (dacă un verb, spre exemplu, este tranzitiv sau intranzitiv), ori alte informaţii, de natură morfologică (cum ar fi genul substantivelor în cazul limbilor care fac distincţia de gen) sau de natură semantică. În cea mai simplă gramatică pe care o putem lua în consideraţie, utilizând notaţia de tip PATR, o *intrare lexicală* ar putea avea următoarea formă:

Cuvânt iubeşte:

<cat> = V.

Aceasta ne spune, pur şi simplu, că "iubeşte" este un cuvânt a cărui categorie (lexicală) este aceea de verb. O intrare lexicală puţin mai complexă, corespunzătoare aceluiaşi cuvânt, ar putea fi:

Cuvânt iubeşte:

<cat>=V

<timp>=prezent

<arg1>=NP.

Aceasta ne spune, în plus, că verbul este la timpul prezent şi că "iubeşte" reprezintă un verb tranzitiv (deoarece admite un complement direct, întrucât este urmat de un grup nominal).

Reprezentarea gramaticilor PS independente de context în Prolog se poate face într-un mod extrem de natural şi de direct. Astfel, fiecărei reguli PS de tipul

$$S \rightarrow NP\ VP$$

îi va corespunde, în programul Prolog de implementare a gramaticii, clauza

```
s(L1,L):-np(L1,L2),vp(L2,L).
```

în care: L1 reprezintă şirul de intrare iniţial (spre exemplu, şirul ['Colentina', angajează, surori]);

L2 reprezintă şirul de intrare fără grupul nominal iniţial (în exemplul dat, [angajează, surori]);

L reprezintă şirul de intrare fără grupul nominal şi grupul verbal (în acest caz, [ ]).

Această clauză Prolog ar mai putea fi descrisă astfel:

Înlăturarea (separarea, izolarea) unui S din L1 produce L

dacă

înlăturarea unui NP din L1 produce L2

şi

înlăturarea unui VP din L2 produce L.

Regulile care introduc simboluri terminale (cuvinte) vor fi de forma:

```
n([pacient|L],L).
```

O astfel de clauză Prolog s-ar putea traduce prin: "pentru a izola un substantiv, înlătură cuvântul *pacient* de la începutul şirului de intrare". Ea corespunde unei intrări lexicale de cea mai simplă formă cu putinţă, adică una care specifică numai categoria lexicală a cuvântului dat.

Menţionăm faptul că, o asemenea formă a clauzelor Prolog presupune ca propoziţiile să fie reprezentate sub forma *diferenţei de liste* sau, mai corect spus, a *listei diferenţă*. Aceasta este o reprezentare a listelor care doreşte să pună în evidenţă *sfârşitul* unei liste. O soluţie pentru a realiza acest lucru este reprezentarea unei liste printr-o *pereche de liste*. Spre exemplu, lista

```
[a,b,c]
```

poate fi reprezentată prin intermediul următoarelor două liste:

```
L1=[a,b,c,d,e]
L2=[d,e]
```

O asemenea pereche de liste, pe care o vom *nota* L1-L2, reprezintă "diferenţa" dintre listele L1 şi L2. Condiţia care trebuie satisfăcută pentru a utiliza o asemenea reprezentare şi notaţia aferentă este ca lista L2 să reprezinte un sufix al listei L1. Se observă că o aceeaşi listă poate fi reprezentată prin mai multe perechi de liste. Spre exemplu, aceeaşi listă `[a,b,c]` mai poate fi reprezentată în modurile următoare:

```
[a,b,c]-[]
```

sau

```
[a,b,c,d,e|T]-[d,e|T]
```

sau

```
[a,b,c|T]-T
```

unde T este orice listă.

Întrucât cel de-al doilea membru al perechii indică sfârşitul listei pe care o reprezentăm, acest sfârşit de listă poate fi accesat în mod direct. Această reprezentare poate fi, prin urmare, utilizată pentru o implementare eficientă a *concatenării*. Lista reprezentată prin perechea A1-Z1, concatenată cu lista reprezentată prin perechea Z1-Z2, produce o listă reprezentată prin perechea A1-Z2. Relaţia de concatenare corespunzătoare poate fi tradusă prin următoarea faptă Prolog:

```
concat(A1-Z1,Z1-Z2,A1-Z2).
```

În acest caz, pentru concatenarea listei `[a,b,c]` reprezentată prin perechea `[a,b,c|T1]-T1` cu lista `[d,e]` reprezentată prin perechea `[d,e|T2]-T2` se specifică scopul

```
?-concat([a,b,c|T1]-T1,[d,e|T2]-T2,L).
```

Concatenarea se face prin împerecherea acestui scop cu clauza **concat**, definită anterior, iar programul răspunde:

```
L=[a,b,c,d,e|T2]-T2
T1=[d,e|T2]
```

Aşa cum se arată în [3], datorită eficienţei sale, această tehnică este mult mai populară decât utilizarea lui **append** pentru realizarea concatenării.

Revenind la o regulă PS de tipul

$S \rightarrow NP\ VP$

căreia îi va corespunde, în implementarea Prolog a gramaticii, o clauză de forma

```
s(L1,L):-np(L1,L2),vp(L2,L).
```


vom observa că operaţia care stă la baza acestei clauze este aceea a *concatenării* unui şir de cuvinte ce formează un grup nominal şi pe care îl privim ca pe o listă reprezentată prin perechea de liste L1-L2 cu un alt şir de cuvinte ce formează un grup verbal, privit ca o listă reprezentată prin perechea de liste L2-L. Rezultatul concatenării îl constituie propoziţia dată de lista reprezentată prin perechea de liste L1-L.

Utilizând o reprezentare Prolog de tipul descris anterior (şi care, la rândul ei, foloseşte reprezentarea propoziţiilor sub forma listei diferenţă) vom indica, în continuare, un prim exemplu de gramatică reprezentată în Prolog:

```
s(X,Z):-np(X,Y),vp(Y,Z).
vp(X,Z):-v(X,Z).
vp(X,Z):-v(X,Y),np(Y,Z).
np(['Dr. Popescu'|X],X).
np(['Colentina'|X],X).
np([surori|X],X).
np([pacienti|X],X).
v([omoara|X],X).
v([angajeaza|X],X).
```

Gramatica nr.1

O astfel de gramatică **generează** propoziţii. Ea poate fi însă utilizată şi "în sens invers", adică pentru a **recunoaşte** o propoziţie dată. Un dispozitiv de recunoaştere ca cel programat mai sus decide dacă o propoziţie dată aparţine unui anumit limbaj. Cu alte cuvinte, recunoaşte dacă propoziţia poate fi generată de gramatica corespunzătoare.

Se spune că o gramatică *supragenerează* dacă ea legitimează şiruri de cuvinte care nu pot fi construite ca expresii gramaticale (corecte) ale limbajului studiat. Într-un context computaţional, supragenerarea de către o gramatică nu reprezintă o problemă reală atunci când demersul este acela de a *recunoaşte* sau înţelege limbajul corect format. Pe de altă parte, fenomenul de supragenerare s-ar putea dovedi fatal dacă scopul este acela de a *produce* şiruri corect formate.

Se spune că o gramatică *subgenerează* atunci când există şiruri de cuvinte ale limbajului studiat corect formate din punct de vedere sintactic şi cărora gramatica nu le atribuie nici o structură. În context computaţional, subgenerarea de către o gramatică nu reprezintă o problemă reală dacă scopul este acela de a *produce* limbaj adecvat. Pe de altă parte, fenomenul se poate dovedi fatal dacă ţelul este acela de a *recunoaşte* sau de a înţelege.

În cazul exemplului de gramatică prezentat (Gramatica nr.1), atunci când dorim să testăm, de pildă, dacă propoziţia

Colentina angajează surori.

este una corect formată, adică dacă ea este *recunoscută* de către gramatică, vom interoga Prologul astfel:

```
?-s(['Colentina',angajeaza,surori],[]).
```

Programul va răspunde afirmativ. Pentru a utiliza aceeaşi gramatică în *generare*, interogarea se va face astfel:

```
?-s(Propozitie,[]).
```

Printre şirurile generate de gramatică se numără

```
Propozitie=['Dr. Popescu',omoara,pacienti]
Propozitie=['Dr. Popescu',angajeaza,surori]
```

dar şi

```
Propozitie=[surori,omoara,'Colentina']
```

Dacă utilizăm Gramatica nr.1 în generare şi cerem generarea tuturor şirurilor corect formate, vom obţine 40 de propoziţii, care sunt fie corecte, fie aproape corecte, atât din punct de vedere sintactic, cât şi semantic, mai ales dacă se au în vedere diverse posibile sensuri figurate ale cuvintelor.

Un alt exemplu de gramatică, care utilizează regulile de rescriere

$$S \rightarrow NP\ VP$$
$$VP \rightarrow V\ NP$$
$$NP \rightarrow Det\ N$$

şi dispune de intrări lexicale diferite este:

```
s(X,Z):-np(X,Y),vp(Y,Z).
vp(X,Z):-v(X,Y),np(Y,Z).
np(X,Z):-det(X,Y),n(Y,Z).
det([un|X],X).
det([niste|X],X).
det([o|X],X).
n([elev|X],X).
n([elevi|X],X).
n([carte|X],X).
v([iubeste|X],X).
v([iubesc|X],X).
```

Gramatica nr.2

Printre propoziţiile generate de această gramatică se numără propoziţia corectă

Un elev iubeşte o carte.

dar şi

Un elev iubeşte un elevi.
Un elev iubeşte nişte elev.
Un elev iubeşte un carte.
Un elev iubesc un carte.

şi altele (atât corecte, cât şi incorecte din punct de vedere gramatical).

7.2.4.1. Gramatici DC

Conversia regulilor PS în clauze Prolog este atât de simplă, încât ea se poate efectua şi în mod automat. De altfel, Prologul include o facilitate pentru a realiza acest lucru, şi anume o extindere a notaţiei numită *notaţie DCG* (de la englezescul *definite-clause grammar*). Aceasta reprezintă o notaţie specială pentru regulile unei gramatici. Exemple de clauze scrise utilizând **notaţia DCG** sunt:

```
s --> np , vp.
np --> det , n.
n --> [elev].
```

Aceste clauze vor fi automat convertite, în faza de consultare, la:

```
s(X,Z):-np(X,Y),vp(Y,Z).
np(X,Z):-det(X,Y),n(Y,Z).
n([elev|X],X).
```

Sistemul DCG reprezintă un *compilator* pentru regulile gramaticii, pe care le traduce direct în clauze Prolog executabile.

Definiţia 7.4
Prin **gramatici DC** vom înţelege acele gramatici ale căror reguli de rescriere sunt exprimate în notaţia DCG.

Aşa cum s-a mai spus, o gramatică generează propoziţii. Dar ea poate fi, în egală măsură, utilizată pentru a recunoaşte o propoziţie dată. În timpul unui asemenea proces de recunoaştere propoziţia dată este practic descompusă în constituenţii săi. Aceasta reprezintă tocmai procesul de analiză sintactică (sau parsing), la care ne-am mai referit. Pentru a implementa o gramatică este necesar să se scrie, de fapt, un program de analiză sintactică corespunzător acelei gramatici. O gramatică scrisă în DCG reprezintă deja un program de parsing pentru ea însăşi.

Un program Prolog poate conţine atât reguli DCG, cât şi clauze Prolog obişnuite. Translatorul DCG afectează numai clauzele ce conţin functorul '-->'. Toate celelalte clauze sunt presupuse a fi clauze Prolog obişnuite şi sunt lăsate neschimbate. Translatorul transformă regulile DCG în clauze Prolog prin adăugarea a două argumente suplimentare corespunzător fiecărui simbol care nu se află inclus între paranteze sau acolade. Argumentele sunt în mod automat aranjate astfel încât să poată fi corect utilizate în procesul de analiză sintactică.

O **regulă DCG** este de forma

Simbol neterminal --> extindere

unde *extindere* constă în unul dintre următoarele elemente:

- un simbol neterminal (ex.: np);
- o listă de simboluri terminale (ex.: [elev] sau [fiecare,elev]);
- un constituent vid reprezentat prin [ ];
- un scop Prolog inclus între acolade, cum ar fi `{write('Gasit NP')}`;
- o serie de oricare dintre aceste elemente, separate prin virgule.

Ca şi în Prolog, simbolul "punct şi virgulă" are semnificaţia lui "sau", ceea ce permite scrierea unor reguli DCG de forma

```
n-->[elev];[elevi];[carte].
```

În ceea ce priveşte sintaxa DCG, să mai notăm faptul că simbolurile neterminale nu se mai află între paranteze, în timp ce simbolurile terminale sunt incluse între paranteze drepte, ceea ce le transformă în liste Prolog. Simbolurile sunt separate prin virgule şi fiecare regulă se încheie prin punct, ca orice clauză Prolog.

Spre deosebire de notaţia PATR, care nu impune o ordine şi o aritate fixate, notaţia DCG presupune în mod esenţial ca numărul şi ordinea argumentelor să fie fixate. De asemenea, DCG permite imbricarea termenilor şi a variabilelor până la adâncimi arbitrare.

În cazul implementărilor Prolog care acceptă notaţia DCG, gramaticile transformate corespunzătoare pot fi imediat utilizate ca dispozitive de recunoaştere a propoziţiilor. Un asemenea dispozitiv de recunoaştere presupune însă, în mod esenţial, ca propoziţiile să fie reprezentate sub forma listei diferenţă de simboluri terminale. Acest fapt a fost deja justificat atunci când s-a arătat că transcrierea regulilor PS în Prolog se bazează pe reprezentarea operaţiei de concatenare, iar aceasta din urmă se realizează mult mai eficient utilizând lista diferenţă în detrimentul lui **append**. Luând în consideraţie această reprezentare a propoziţiilor, gramatica DC


```
s --> [a],[b].
s --> [a],s,[b].
```

poate fi utilizată în recunoaşterea unor propoziţii prin interogări de tipul:

```
?-s([a,a,b,b],[]).
yes
```

Prin operaţia de conversie pe care o efectuează, Prologul transformă automat regulile DCG în clauze Prolog obişnuite şi, în acest mod, converteşte regulile de rescriere date ale gramaticii într-un *program* de recunoaştere a propoziţiilor generate de gramatica respectivă.

Iată, spre exemplu, transcrierea în notaţia DCG a Gramaticii nr.2:

```
s-->np,vp.
vp-->v,np.
np-->det,n.
det-->[un].
det-->[niste].
det-->[o].
n-->[elev].
n-->[elevi].
n-->[carte].
v-->[iubeste].
v-->[iubesc].
```

Gramatica nr.3 (notaţie DCG)

Gramaticile DC *extind* gramaticile independente de context în următoarele moduri (cu referire la SICStus Prolog):

- Un simbol neterminal poate fi orice termen Prolog (altul decât o variabilă sau un număr).
- Un simbol terminal poate fi orice termen Prolog. Pentru a se face distincţia între terminale şi neterminale, un şir de unul sau mai multe simboluri terminale este scris, în cadrul unei reguli a gramaticii, ca o listă Prolog. Un şir vid este scris sub forma listei vide [ ]. Dacă simbolurile terminale sunt coduri de caractere, astfel de liste pot fi scrise (ca şi în alte situaţii) ca şiruri de caractere (tipul string). Un şir vid este scris sub forma listei vide, [ ] sau " ".
- Se pot include în membrul drept al unei reguli a gramaticii condiţii suplimentare, sub forma unor apelări Prolog de proceduri. Astfel de apelări de proceduri sunt incluse între acolade { }.

- Membrul stâng al unei reguli a gramaticii constă dintr-un neterminal urmat, în mod opţional, de un şir de terminale (scrise ca listă Prolog).
- Disjuncţia, if-then, if-then-else şi nedemonstrabilul pot fi declarate în mod explicit în membrul drept al unei reguli a gramaticii, prin folosirea operatorilor ; (|), -> şi respectiv \+, ca şi în cazul unei clauze Prolog.
- Simbolul **cut** poate fi inclus în membrul drept al unei reguli a gramaticii, ca şi în cadrul unei clauze Prolog. Simbolul **cut** nu trebuie inclus între acolade { }.

Pentru a rezuma, vom formula în mod mai general felul în care se face *traducerea între DCG şi Prologul standard*. Astfel, fiecare *regulă DCG* este tradusă într-o *clauză Prolog* conform următoarei scheme generale:

- Dacă regula DCG conţine în membrul drept numai simboluri neterminale, adică este de forma

```
n-->n1,n2,...,nn.
```

cu n1, n2,...,nn simboluri neterminale, atunci ea este tradusă în clauza Prolog

```
n(X,Y):-n1(X,Y1),n2(Y1,Y2),...,nn(Yn-1,Y).
```

- Dacă regula DCG conţine în membrul drept atât simboluri neterminale, cât şi terminale, adică este de forma

```
n-->n1,[t2],n3,[t4].
```

cu n1, n3 simboluri neterminale şi t2, t4 simboluri terminale, atunci ea este tradusă în clauza Prolog

```
n(X,Y):-n1(X,[t2|Y1]),n3(Y1,[t4|Y]).
```

(Se observă că simbolurile terminale sunt tratate în mod diferit, în sensul că un simbol terminal nu apare ca un scop al clauzei, ci este inserat direct în lista corespunzătoare).

Pentru a vedea una dintre problemele frecvente care se pot ridica *după traducerea regulilor DCG în clauze Prolog*, să luăm, din nou, în consideraţie Gramatica nr.1, uşor modificată şi scrisă utilizând notaţia DCG. Gramatica astfel rezultată este Gramatica nr. 4. În timp ce Gramatica nr.1 genera 40 de propoziţii, mai mult sau mai puţin corecte din punct de vedere sintactic, datorită schimbării timpurilor verbale, Gramatica nr.4 va genera, pe lângă propoziţii corecte, şi propoziţii cum ar fi:

surori angajase

(în loc de *angajaseră*)

surori omoară pacienţi

(în loc de *omorau*)

```
s-->np,vp.
vp-->v.
vp-->v,np.
np-->['Dr. Popescu'].
np-->['Colentina'].
np-->[surori].
np-->[pacienti].
v-->[omoara].
v-->[angajase].
```

Gramatica nr.4

Dezacordul dintre subiect şi predicat se datorează existenţei regulii

```
s-->np,vp.
```

care afirmă că, practic, *orice* grup nominal şi *orice* grup verbal pot fi concatenate pentru a forma o propoziţie. Dar în limba română ca, de altfel, în majoritatea limbilor, grupul verbal şi grupul nominal nu sunt independente, ci trebuie să concorde în număr. Acesta este un exemplu de *dependenţă de context* - i.e. dependenţa grupului sintactic de contextul în care el intervine. Astfel de fenomene pot fi uşor tratate de către gramaticile DC, întrucât acestea admit adăugarea de *argumente* simbolurilor neterminale ale gramaticii. În exemplul anterior, prin adăugarea argumentului `Numar`, prima regulă DCG devine:

```
s(Numar)-->np(Numar),vp(Numar).
```

şi ea asigură concordanţa de număr între grupul nominal şi grupul verbal care se concatenează pentru a forma întreaga propoziţie.

Atunci când regulile DCG sunt citite de către Prolog şi convertite în clauze Prolog, argumentele simbolurilor neterminale sunt, pur şi simplu, adăugate argumentelor (de tip listă) obişnuite, respectându-se convenţia că cele două liste apar ultimele. Astfel, regula DCG

```
s(Numar)-->np(Numar),vp(Numar).
```

este convertită în următoarea clauză Prolog:

```
s(Numar,X,Y):-np(Numar,X,Y1),vp(Numar,Y1,Y).
```

Interogarea Prologului se modifică în mod corespunzător, pentru a fi incluse noile argumente. Spre exemplu, pentru a face ca Gramatica nr.4, la care s-a

adăugat *caracteristica de număr* prin introducerea parametrului `Numar`, să genereze toate propozițiile posibile, interogarea se va face în felul următor:

```
?-s(Numar,Propozitie,[]).
```

Gramatica nr.5 modifică, în acest sens, Gramatica nr.3 prin adăugarea atât a unei caracteristici de număr, cât şi a uneia de gen.

Pentru a vedea toate propozițiile care pot fi generate de Gramatica nr.5 se interoghează Prologul în acelaşi mod

```
?-s(Numar,Propozitie,[]).
```

şi se obține răspunsul:

```
Numar=sing,
Propozitie=[un,elev,iubeste,un,elev]

Numar=sing
Propozitie=[un,elev,iubeste,o,carte]

Numar=plural
Propozitie=[niste,elevi,iubesc,niste,elevi]

Numar=sing
Propozitie=[o,carte,iubeste,un,elev]

Numar=sing
Propozitie=[o,carte,iubeste,o,carte]

no
```

În total există 5 soluții, toate corecte din punct de vedere sintactic (dar nu şi din punct de vedere semantic). Întrucât, pentru moment, nu suntem interesați şi de aspectele semantice, vom spune că gramatica generează numai propoziții corecte.

Gramatica nr.5 este chiar prea restrictivă, întrucât ea nu generează, spre exemplu, propoziția corectă

Nişte elevi iubesc o carte.

Pentru a vedea dacă această propoziție este generată de gramatica dată, interogarea Prologului se face astfel

```
?-s(plural,[niste,elevi,iubesc,o,carte],[]).
```

iar răspunsul este

```
no
```



```
s(Numar)-->np(Numar),vp(Numar).
vp(Numar)-->v(Numar),np(Numar).
np(Numar)-->det(Numar,Gen),n(Numar,Gen).
det(sing,masc)-->[un].
det(plural,masc)-->[niste].
det(sing,fem)-->[o].
n(sing,masc)-->[elev].
n(plural,masc)-->[elevi].
n(sing,fem)-->[carte].
v(sing)-->[iubeste].
v(plural)-->[iubesc].
```

Gramatica nr.5

Gramatica generează, în schimb, aceeaşi propoziție la singular:

```
?-s(sing,[un,elev,iubeste,o,carte],[]).

yes
```

Neajunsul menționat este remediat de Gramatica nr.6. Interogarea

```
?-s(Numar,Propozitie,[]).
```

primeşte ca răspuns 9 soluții ale căror propoziții sunt toate corecte din punct de vedere sintactic (dar mai puțin din punct de vedere semantic). Printre propozițiile generate se numără şi propoziția pe care Gramatica nr.5 nu o recunoştea:

Nişte elevi iubesc o carte.

```
s(Numar)-->np(Numar),vp(Numar).
vp(Numar)-->v(Numar),np(Numar1).
np(Numar)-->det(Numar,Gen),n(Numar,Gen).
det(sing,masc)-->[un].
det(plural,masc)-->[niste].
det(sing,fem)-->[o].
n(sing,masc)-->[elev].
n(plural,masc)-->[elevi].
n(sing,fem)-->[carte].
v(sing)-->[iubeste].
v(plural)-->[iubesc].
```

Gramatica nr.6

Schimbarea efectuată, care determină recunoaşterea acestei propoziții de către Gramatica nr.6, constă în utilizarea parametrului `Numar1` în cea de-a doua regulă DCG, în locul aceluiaşi parametru `Numar`. Această schimbare de parametru nu mai impune concordanța de număr şi în *interiorul* grupului verbal,

rămânând în vigoare doar concordanţa de număr între grupul nominal şi grupul verbal care, prin concatenare, alcătuiesc întreaga propoziţie.

7.2.4.2. Gramatici cu caracteristici şi reprezentarea lor în notaţia DCG

Majoritatea analizorilor sintactici se bazează pe gramatici independente de context. În mod evident însă acestea nu pot cuprinde toată bogăţia unui limbaj natural. De aceea se obişnuieşte să se utilizeze în procesare o extensie a acestor gramatici, realizată prin asocierea unei mulţimi de caracteristici. Astfel de gramatici augmentate sunt deosebit de utile în modelarea cu precădere a fenomenelor specifice limbajului natural.

Gramatica nr.5 şi Gramatica nr.6 reprezintă nişte **gramatici cu caracteristici,** întrucât ele folosesc caracteristicile de număr şi de gen. Sintaxa teoretică modernă se bazează în mod esenţial pe utilizarea caracteristicilor, întregi categorii atomice cum ar fi NP sau V fiind înlocuite prin mulţimi de specificaţii ale unor caracteristici.

Definiţia 7.5
O **specificaţie a unei caracteristici** constă dintr-o **caracteristică** (cum ar fi *cazul*) şi o **valoare** pentru acea caracteristică (cum ar fi *acuzativ*).

Există teorii sintactice bazate pe caracteristici care reintroduc denumirile cunoscute ale categoriilor, cum ar fi NP sau V, ca **valori** ale unei anumite caracteristici pe care o numesc **cat**. (Această caracteristică a fost deja utilizată de noi în definirea intrărilor lexicale). Întrucât, în felul acesta, se poate face trecerea de la categoriile atomice (de tip S, NP, V etc.) la mulţimi de informaţii exprimate prin caracteristici, se pune problema exprimării regulilor de rescriere ale gramaticilor cu caracteristici. *În mod formal*, putem exprima o regulă de tipul

$$S \rightarrow NP\ VP$$

sub forma:

Regulă {formarea propoziţiilor}

$$X0 \rightarrow X1X2:$$

<X0 cat> = S

<X1 cat> = NP

<X2 cat> = VP.

Într-o notaţie de acest tip, regula în sine conţine nişte *marcaje* (X0, X1 şi X2) însoţite de o mulţime de ecuaţii precum <X0 cat>=S, ecuaţie care trebuie interpretată ca stipulând: "valoarea caracteristicii de categorie a lui X0 este S".

Formalismul DCG este extrem de eficient în formularea gramaticilor cu caracteristici. Acest formalism permite atribuirea de termeni şi/sau de variabile logice, în număr nelimitat, ca argumente ale categoriilor, o facilitate inexistentă în contextul formalismului definit de gramaticile PS. Astfel, următoarei *reguli a unei gramatici cu caracteristici* (exprimate într-o *notaţie de tip PATR*)

Regulă

$$S \rightarrow NP\ VP:$$

<NP persoană>=<VP persoană>

<NP număr>=<VP număr>

<NP caz>=nominativ

<VP vforma>=noninfinitival.

îi corespunde *regula DCG*

```
s-->np(Persoana,Numar,nominativ),
    vp(Persoana,Numar,noninfinitival).
```

O *intrare lexicală* într-o astfel de gramatică ar putea fi de forma:

```
np(3,singular,_)-->[anca].
```

În acest caz, gramatica următoare

```
s-->np(Persoana,Numar,nominativ),
    vp(Persoana,Numar,noninfinitival).
np(3,singular,_)-->[anca].
vp(3,singular,noninfinitival)-->[mananca].
```

generează unica propoziţie

anca mănâncă

Să mai notăm faptul că regulile DCG pot avea forme extrem de complexe, întrucât acest formalism permite imbricarea termenilor până la niveluri arbitrare.

Aşa cum am mai remarcat, caracteristica principală a formalismului DCG este aceea că el îmbogăţeşte regulile gramaticilor PS independente de

context cu *argumente şi variabile*. Printre cele mai importante utilizări ale argumentelor introduse se numără *specificarea acordului între subiect şi predicat*. Cu ajutorul caracteristicilor se poate semnala acordul în număr între verb (cu valoare predicativă) şi subiectul său:

***Copiii** îşi fac lecţiile.*

***Copilul** îşi face lecţiile.*

Se observă că, atât în limba engleză, cât şi în limba română, obiectul verbului (complementul direct) nu se supune aceleiaşi reguli de acord în număr. Astfel, este corectă atât propoziţia

Copiii îşi fac **lecţiile**.

cât şi propoziţia

Copiii îşi fac ***lecţia***.

Pentru a face ca regulile DCG să ia în consideraţie şi să transcrie acordul, se va introduce caracteristica `Numar` având valorile `singular` şi `plural` şi se vor adăuga argumente atât categoriei NP, cât şi categoriei VP, pentru a indica numărul. Un grup nominal (NP) are acelaşi număr (singular sau plural) ca şi substantivul în jurul căruia se organizează, în timp ce un grup verbal (VP) are numărul verbului corespunzător. Iată transcrierea unora dintre regulile DCG deja utilizate, care respectă regula acordului, prin intermediul caracteristicii de număr:

```
np(Numar)-->det,n(Numar).
vp(Numar)-->v(Numar),np(_).
```

Grupul nominal şi grupul verbal prin concatenarea cărora se formează propoziţia trebuie să aibă acelaşi număr:

```
s-->np(Numar),vp(Numar).
```

Un analizor sintactic (parser) care utilizează aceste reguli va accepta numai acele propoziţii în care verbul (cu valoare predicativă) şi subiectul său respectă acordul în număr. La rândul lor, intrările lexicale care vor fi utilizate trebuie să specifice numărul fiecărui substantiv şi respectiv verb, ca în exemplele:

```
n(singular)-->[copil];[elev];[profesor];[lectia].
n(plural)-->[copii];[elevi];[profesori];[lectiile].
v(singular)-->[face];[asista];[citeste];[scrie].
v(plural)-->[fac];[asista];[citesc];[scriu].
```

Structura grupului verbal, atât în limba română, cât şi în limba engleză, depinde de verbul în jurul căruia se organizează acest grup sintactic. În

cazul verbelor *tranzitive*, spre exemplu, adică a acelor verbe care (în sintaxa tradiţională, analitică) acceptă un complement direct, în funcţie de forma (lărgită) a complementului se pot determina structuri diferite ale grupului verbal, care vor genera reguli PS diferite (a se vedea [10]).

Reprezentarea gramaticilor cu caracteristici în DCG se bazează pe o proprietate fundamentală a acestor gramatici: aceea că există o caracteristică (pe care am numit-o **cat**) a cărei valoare este atomică şi întotdeauna furnizată în cadrul oricărei descrieri a unei categorii date a gramaticii. Această proprietate este necesară deoarece am utilizat, în permanenţă, valoarea lui **cat** în rolul predicatului Prolog. Iar în Prolog este imposibilă specificarea unui scop al cărui predicat este necunoscut (o variabilă).

În concluzie, putem spune că **formalismul DCG**, care îmbogăţeşte regulile gramaticilor PS independente de context cu argumente şi variabile, este extrem de util şi că gramaticile DC reprezintă o extensie relativ naturală a gramaticilor PS independente de context, în cadrul oferit de limbajul Prolog. Aşa cum s-a văzut, caracteristica de bază a unei reguli independente de context este faptul că ea nu impune nici un fel de restricţii asupra contextului în care membrul său stâng se poate realiza sub forma membrului său drept. Ne-am concentrat, cu precădere, asupra formalismului oferit de gramaticile independente de context, întrucât, aşa cum s-a menţionat deja, acesta este suficient de puternic pentru a descrie majoritatea structurilor care apar în limbile naturale. Gramaticile DC pot lua însă în consideraţie şi *contextul*, prin înregistrarea de informaţii în argumente suplimentare ale predicatelor. În acest mod, ele sunt capabile să descrie limbaje care nu pot fi descrise prin intermediul gramaticilor PS independente de context. Pentru un comentariu asupra modului în care se face scrierea gramaticilor DC corespunzător limbajelor care nu sunt independente de context, vezi [9].

Reprezentarea gramaticilor în Prolog s-a referit cu precădere la gramaticile PS independente de context, întrucât majoritatea algoritmilor de analiză sintactică existenţi utilizează acest tip de gramatici (a se vedea §7.3).

7.3. Analiza sintactică bazată pe constituenţi

Tehnicile de analiză sintactică sunt deosebit de variate şi diferă în funcţie de tipul de gramatică utilizat. Majoritatea algoritmilor existenţi utilizează însă gramaticile PS independente de context şi, prin urmare, efectuează o **analiză sintactică bazată pe constituenţi**.

Procesul de analiză sintactică este numit, în literatura de specialitate, **parsing**, iar un analizor sintactic este cunoscut sub denumirea de **parser**[31]. (Întrucât studiul gramaticilor PS, care sunt cel mai frecvent utilizate, s-a dezvoltat în special relativ la limba engleză, aceşti termeni englezeşti au fost preluaţi de cea mai mare parte a literaturii de specialitate scrise în diverse limbi). Termenul de **parsing** a desemnat iniţial *analiza gramaticală în sens clasic* (analiza sintactică şi cea morfologică tradiţionale), după care sensul acestui concept s-a extins, ajungând să se refere la o serie de operaţii relativ variate, dar care sunt, într-un fel sau altul, analoage celei tradiţionale). Această extindere a sensului este rezultatul apariţiei unor noi conceptualizări în diverse domenii, cum ar fi lingvistica teoretică, teoria limbajelor formale, inteligenţa artificială şi psiholingvistica.

În lingvistică s-a făcut trecerea de la descrierile neformale, lipsite de norme riguroase, oferite de gramatica tradiţională, spre o nouă viziune, care tratează aceste descrieri ca pe nişte obiecte formale, în particular ca pe nişte structuri alcătuite din constituenţi. *Aceste descrieri gramaticale formalizate (în particular structuri alcătuite din constituenţi) reprezintă ieşirea (output) operaţiei de parsing*. Aceasta din urmă poate fi acum privită ca fiind mai degrabă algoritmică decât euristică. În această viziune, operaţia de parsing constă în aplicarea unui şir de principii caracteristice fiecărei limbi, adică a unui şir de reguli PS, care se aplică (reprezentării) unei propoziţii astfel încât să se obţină toate descrierile gramaticale adecvate ale respectivei propoziţii. Nici o altă descriere a propoziţiei, cu excepţia celor adecvate, nu poate rezulta în urma acestui proces. În acest mod se naşte posibilitatea de a lua în consideraţie principii de tipul $NP + VP = S$ fie din punct de vedere analitic, ca instrucţiuni de atribuire a unor structuri unor propoziţii date dintr-o limbă naturală, fie din punct de vedere sintetic, ca instrucţiuni de compunere a propoziţiilor unei limbi. Mulţimea tuturor acestor principii constituie o **gramatică formală** pentru limba respectivă, gramatică care poate fi privită ca având o *funcţie analitică sau funcţie de parsing*, ori o *funcţie sintetică sau funcţie generativă*. Prin **analiză sintactică bazată pe constituenţi** vom înţelege o analiză gramaticală de acest tip, adică utilizând o gramatică formală de tipul descris. Trebuie remarcat faptul că, atât analiza sintactică, cât şi cea semantică, încep aproape întotdeauna cu descompunerea propoziţiei în grupuri mai mici de cuvinte, cum ar fi grupurile nominale, grupurile verbale, grupurile adjectivale etc. Din punct de vedere sintactic, acest lucru este motivat de presupunerea că, analizând în termeni de grupuri sintactice, se poate obţine un grad mai mare de generalizare în descrierea structurilor aparţinând limbajului natural, decât analizând în termeni de cuvinte individuale.

---

[31] De la verbul englezesc *to parse*, care înseamnă "a separa o propoziţie în părţile sale componente".

În cele ce urmează, ne vom referi la analiza sintactică bazată pe constituenţi[32] şi vom studia, în principal, două mari tipuri de parsing. Atunci când se caută structura propoziţiilor care pot fi construite de către regulile unei gramatici PS, există două tipuri principale de strategii care pot fi aplicate:

- strategia **top-down** (adică "de sus în jos"), care pleacă de la simbolul S şi caută, pe diverse căi, să rescrie simbolurile, până când se ajunge la propoziţia căutată (sau până când se epuizează toate variantele de construcţie, în cazul în care se caută structura unei propoziţii incorecte din punctul de vedere al gramaticii enunţate);
- strategia **bottom-up** (adică "de jos în sus"), care pleacă de la cuvintele propoziţiei şi, prin procedee de deplasare - reducere (căutând membrii drepţi ai regulilor şi înlocuindu-i cu membrii stângi corespunzători), urmăreşte construirea arborelui de derivare de jos în sus, până când se ajunge la simbolul rădăcină S.

Ne vom referi, pe rând, la algoritmii determinaţi de aceste strategii, variante optimizate ale acestora, precum şi implementările lor în Prolog. Aceşti algoritmi de analiză sintactică vor încerca diverse modalităţi de combinare a unor reguli şi deci a constituenţilor, cu scopul de a ajunge la o combinaţie pe baza căreia să se poată construi un arbore de derivare corespunzător structurii secvenţei de intrare, adică structurii propoziţiei analizate. Pentru început, nu vom fi interesaţi de construcţia efectivă a arborelui, ci doar de răspunsul afirmativ sau negativ la întrebarea dacă este posibilă construcţia unui astfel de arbore (cu alte cuvinte dacă secvenţa de intrare poate fi generată sau nu de către gramatica dată).

Atât analizorul sintactic de tip top-down, cât şi cel de tip bottom-up trebuie să aibă capacitatea de a efectua un proces de backtracking, adică de a încerca aplicarea unei reguli şi, în cazul în care aceasta nu reuşeşte, de a reveni şi a încerca o regulă diferită. Un analizor sintactic care are capacitatea de a efectua backtracking este numit un **parser nedeterminist**. După cum se ştie, în Prolog procesul de backtracking se efectuează în mod automat. Aceasta face

---

[32] Constituenţii vor fi presupuşi ca reprezentând structuri alcătuite din elemente adiacente, acesta fiind cadrul în care au fost concepute tehnicile clasice de parsing. Problema tehnicilor de analiză sintactică specifice *constituenţilor discontinui*, care nu constituie obiectul lucrării de faţă, a suscitat, la rândul ei, interesul cercetătorilor şi a fost studiată în special de către McCawley (1982), Tomita (1991, 1995) şi Bunt (1991, 1995). Harry Bunt (1995) propune un algoritm de analiză sintactică cu hartă pentru Gramatici PS discontinue bazat pe ideea centrală că un constituent nu trebuie privit ca o secvenţă de cuvinte, ci ca o *mulţime de cuvinte* înzestrată cu anumite relaţii de precedenţă, relaţii care există atât între cuvintele mulţimii, precum şi între acestea şi cuvinte din afara ei. În elaborarea acestei idei noţiunea centrală este aceea de *arbore discontinuu* ca structură de date de tip arbore care descrie constituentul discontinuu.

ca Prologul să reprezinte un limbaj de programare ideal pentru implementarea analizorilor sintactici de acest tip.

Specificăm faptul că intrarea pentru un parser va fi numită, în cele ce urmează, "şirul de intrare", chiar dacă în Prolog aceasta reprezintă, de obicei, o listă de atomi şi nu un şir de caractere. În fine, pentru a implementa un parser, va fi necesară rescrierea gramaticii ca o mulţime de clauze Prolog (a se vedea §7.2.4).

7.3.1. Analiza sintactică top-down

O modalitate de a efectua analiză sintactică bazată pe constituenţi de tip top-down este aceea de a conferi fiecărei reguli PS o interpretare procedurală. Spre exemplu, regula

$$S \rightarrow NP\ VP$$

poate fi interpretată ca spunând: "pentru a analiza (izola) un S, analizează (izolează) mai întâi un NP şi apoi un VP". În acelaşi timp, o regulă de tipul

$$N \rightarrow mama$$

poate fi interpretată ca spunând: "pentru a analiza (izola) un N (substantiv), acceptă cuvântul *mama* din şirul de intrare". Procesul care ia naştere în virtutea acestor interpretări este adesea numit *coborâre recursivă*, deoarece fiecare regulă PS este transformată într-o procedură, iar aceste proceduri se pot apela una pe cealaltă, ori se pot autoapela în mod recursiv.

Procesul de analiză sintactică (parsing) devine, în acest context, un proces similar cu cel al satisfacerii interogărilor Prolog. Scopul iniţial este S; unele dintre reguli transformă acest scop în alte scopuri, iar alte reguli satisfac scopuri acceptând cuvinte. La fiecare pas al algoritmului analizorul sintactic fie transformă scopul curent într-un nou scop, fie satisface scopul curent prin acceptarea unui cuvânt. În anumite situaţii, analizorul sintactic revine asupra unei acţiuni anterioare, efectuând un proces de backtracking, aşa cum se va vedea în exemplul care urmează. În toate aceste privinţe, analiza sintactică top-down seamănă foarte mult cu modul în care Prologul rezolvă interogările. Se pare că aceasta nu reprezintă o coincidenţă, întrucât analiza sintactică bazată pe constituenţi a fost una dintre aplicaţiile pe care Alain Colmerauer le-a avut în vedere atunci când a conceput Prologul (Colmerauer 1978). Mai mult, formalismul DCG (în traducerea standard) implementează un dispozitiv de recunoaştere sau analizor sintactic de tip top-down şi care lucrează de la stânga la dreapta. Vom reveni asupra implementării Prolog după discutarea algoritmului propriu-zis.

Aşa cum s-a menţionat deja, strategia top-down pleacă de la simbolul S şi încearcă, pe diverse căi, să rescrie simbolurile, până când se ajunge la propoziţia căutată (dacă aceasta este acceptată de gramatica dată). Cu alte cuvinte, un parser top-down începe cu simbolul S şi încearcă să îl rescrie pe acesta sub forma unui şir de simboluri terminale care sunt în concordanţă cu acele cuvinte ce alcătuiesc şirul de intrare din punctul de vedere al categoriei fiecărui cuvânt. *Categoriile cuvintelor* pot fi indicate fie prin intermediul regulilor lexicale ale gramaticii, fie prin utilizarea unui lexicon (caz în care nu mai este necesar ca gramatica să conţină reguli lexicale). *Starea analizei* la un anumit moment este caracterizată şi poate fi reprezentată printr-o listă de simboluri constituind rezultatul operaţiilor efectuate până la acel moment, numită *lista simbolurilor*. Spre exemplu, analizorul începe din starea (S), iar după aplicarea regulii $S \rightarrow NP\ VP$ lista simbolurilor devine (NP VP). Dacă imediat după aceea se aplică regula $NP \rightarrow ART\ N$, atunci lista simbolurilor va fi (ART N VP) ş.a.m.d. Analizorul ar putea continua în acest mod până când starea constă în întregime din simboluri terminale, după care ar putea verifica dacă şirul de intrare coincide cu ceea ce s-a obţinut. Dar, un algoritm construit în acest mod ar fi extrem de neperformant, întrucât preia eventuale erori comise în utilizarea regulilor de rescriere, erori pe care nu le depistează decât mult mai târziu. Un analizor sintactic performant trebuie să facă verificarea şirului de intrare de îndată ce acest lucru este posibil.

Presupunând că este dată Gramatica nr.1 la care ne-am referit anterior, adică gramatica următoare (în notaţia PATR)

Regulă

$S \rightarrow NP\ VP$.

Regulă

$VP \rightarrow V\ NP$.

Regulă

$VP \rightarrow V$.

Cuvânt Dr. Popescu:

<cat> = NP.

Cuvânt surori:

<cat> = NP.

Cuvânt Colentina:

<cat> = NP.

Cuvânt pacienţi:

<cat> = NP.

Cuvânt omoară:

<cat> = V.

Cuvânt angajează:

<cat> = V.

şi că şirul de intrare este

Colentina angajează surori.

un *analizor sintactic top-down care procesează de la stânga la dreapta şi efectuează o căutare de tip depth-first* va lucra, în mare, în felul următor:

Se caută un S.
Din ce poate să constea un S?
Un S poate să constea dintr-un NP urmat de un VP.
Se caută un NP.
Din ce poate să constea un NP?
Gramatica nu conţine reguli pentru rescrierea lui NP.
Există o intrare lexicală care desemnează "surori" ca pe un membru al categoriei NP.
Este primul cuvânt al şirului de intrare "surori"?
Nu.
Există o intrare lexicală care desemnează "Colentina" ca pe un membru al categoriei NP.
Este primul cuvânt al şirului de intrare "Colentina"?
Da.
S-a găsit un NP constând din cuvântul "Colentina".
Se caută un VP.
*** Din ce poate să constea un VP?
Un VP poate să constea dintr-un V.
Se caută un V.
Din ce poate să constea un V?
Nu există reguli de rescriere referitoare la V.
Există o intrare lexicală care desemnează "omoară" ca pe un membru al categoriei V.
Este următorul cuvânt al şirului de intrare "omoară"?
Nu.
Există o intrare lexicală care desemnează "angajează" ca pe un membru al categoriei V.
Este următorul cuvânt al şirului de intrare "angajează"?
Da.
S-a găsit un V constând din cuvântul "angajează".
S-a găsit un VP constând dintr-un V care constă din cuvântul "angajează".

S-a găsit un S constând din:
un NP constând din cuvântul "Colentina" şi un VP constând dintr-un V constând din cuvântul "angajează".
S-a ajuns la sfârşitul şirului de intrare?
Nu.
Se revine la *** şi se ia o altă decizie (*backtracking*).
Se caută din nou un VP.
Din ce poate să constea un VP?
Un VP mai poate să constea dintr-un V urmat de un NP.
Se caută un V.
Din ce poate să constea un V?

ş.a.m.d.

Analizorul sintactic va procesa şirul de intrare acceptând cuvintele unul câte unul. Dacă **C** reprezintă tipul de constituent pe care analizorul sintactic îl caută la un anumit moment (S, NP, V etc.), atunci algoritmul pentru analizarea unui constituent de tipul **C** este următorul:

Algoritmul 7.1

- Dacă **C** reprezintă un cuvânt individual, el este căutat în regulile lexicale ale gramaticii şi acceptat din şirul de intrare.
- Altfel, se caută în regulile PS ale gramaticii pentru a-l rescrie pe **C** sub forma unei liste de constituenţi şi se analizează acei constituenţi unul câte unul.

□

7.3.1.1. Algoritm de analiză sintactică top-down

O *stare a analizei* poate fi definită şi printr-o pereche constând din lista simbolurilor (aşa cum a fost ea descrisă anterior) şi un număr ce indică poziţia curentă în cadrul propoziţiei. Poziţiile se marchează între cuvinte, 1 fiind poziţia din faţa primului cuvânt. Spre exemplu, în cazul propoziţiei

$_1$ *Colentina* $_2$ *angajează* $_3$ *surori* $_4$

o posibilă stare este ((N VP)1), care indică faptul că analizorul sintactic trebuie să găsească un N urmat de un VP începând de la poziţia 1. Noile stări sunt generate din cele vechi în funcţie de cum primul simbol este sau nu simbol lexical. Dacă acesta reprezintă un simbol lexical (cum ar fi N din exemplul anterior) şi dacă următorul cuvânt din şirul de intrare poate aparţine respectivei

categorii lexicale, atunci starea poate fi actualizată prin eliminarea primului simbol şi actualizarea contorului de poziţie.

Presupunând că sunt date gramatica PS independentă de context

$S \rightarrow NP\ VP$

$NP \rightarrow ART\ N$

$NP \rightarrow ART\ ADJ\ N$

$VP \rightarrow V$

$VP \rightarrow V\ NP$

şi lexiconul (în notaţie PATR)

Cuvânt latră:
<cat> = V.
Cuvânt câine:
<cat> = N.
Cuvânt un:
<cat> = ART.

sau, mai simplu,

latră: V
câine: N
un: ART

şi că şirul de intrare este reprezentat de propoziţia

$_1$ *Un* $_2$ *câine* $_3$ *latră* $_4$

o stare tipică a analizei poate fi ((N VP)2). Această stare indică faptul că analizorul caută un N urmat de un VP pornind de pe poziţia 2. Următoarea stare a analizorului va fi generată ţinându-se cont de faptul că primul simbol este unul lexical (N). Deoarece cuvântul "câine" de la poziţia 2 apare în lexicon ca fiind un N (substantiv), generarea stării următoare se va face prin eliminarea primului simbol şi repoziţionarea contorului. Aceasta va fi starea ((VP)3), care indică faptul că analizorul trebuie să găsească un VP pornind de la poziţia 3. Dacă primul simbol este un neterminal (cum ar fi acest VP), atunci el este rescris utilizând o regulă a gramaticii. Spre exemplu, utilizând a patra regulă a gramaticii amintite noua stare ar fi

((V)3),

în timp ce prin utilizarea celei de a cincea reguli s-ar genera noua stare

((V NP)3).

Un algoritm de analiză sintactică complet trebuie să exploreze în mod sistematic toate noile stări posibile, tehnica care asigură cel mai bine acest lucru fiind aceea de backtracking. În această abordare, se vor genera toate noile stări posibile. Una dintre acestea va fi aleasă pentru a reprezenta starea următoare, iar celelalte vor fi salvate ca *stări de revenire* sau *stări backup*. Dacă se ajunge în situaţia în care starea curentă nu poate conduce la o soluţie, se alege o nouă stare curentă din lista stărilor backup.

Un **algoritm de analiză sintactică top-down** lucrează, prin urmare, cu o listă de stări posibile, numită **lista posibilităţilor**. Primul element al acestei liste este **starea curentă**, care constă din o listă a simbolurilor şi o poziţie în cadrul propoziţiei, iar elementele rămase reprezintă **stările backup**. Spre exemplu, lista posibilităţilor

(((N)2) ((NUME)1) ((ADJ N)1))

indică pe post de stare curentă acea stare care constă din lista de simboluri (N) de la poziţia 2 şi stabileşte faptul că există două posibile stări backup: una constând din lista de simboluri (NUME) la poziţia 1 şi cealaltă constând din lista de simboluri (ADJ N) la poziţia 1. Algoritmul de analiză sintactică top-down este

Algoritmul 7.2

Se pleacă din starea iniţială ((S)1), fără stări backup. Paşii algoritmului sunt:

1. *Selectează* starea curentă: se scoate prima stare din lista de posibilităţi; fie ea starea **C**. Dacă lista de posibilităţi este vidă, atunci algoritmul eşuează (i.e. nu se poate face cu succes analiza sintactică, în sensul că secvenţa de intrare este respinsă ca nefiind conformă cu gramatica limbajului) şi **STOP**.

2. Dacă **C** are o listă de simboluri vidă, iar contorul indică poziţia de la sfârşitul propoziţiei, atunci algoritmul are succes (i.e. propoziţia este acceptată ca fiind corectă) şi **STOP**.

3. Altfel, *generează* următoarele stări posibile. (Adăugarea acestor stări în lista posibilităţilor se va face în funcţie de modul de organizare al listei, adică de modul în care se efectuează căutarea).

3.1. Dacă primul simbol din lista de simboluri a lui **C** este unul lexical şi următorul cuvânt al propoziţiei aparţine acelei categorii lexicale, atunci se creează o nouă stare prin înlăturarea primului simbol din lista de simboluri şi repoziţionarea

contorului de poziție. Starea nou creată se adaugă listei posibilităților.

3.2. Altfel (i.e. primul simbol din lista de simboluri a lui C este un neterminal), se generează o nouă stare corespunzător fiecărei reguli a gramaticii care poate rescrie acest neterminal. Se adaugă listei posibilităților noile stări generate.

❑

Acest tip de analiză sintactică bazată pe constituenți poate fi privită ca un caz particular de **problemă de căutare**, așa cum este definită aceasta în inteligența artificială. În particular, analizorul sintactic top-down a fost descris de către noi în termenii următoarei ***proceduri de căutare generalizate***: lista posibilităților este la început setată ca reprezentând starea inițială a analizei; se repetă pașii care urmează până când se înregistrează succes sau eșec. Pașii sunt:

1. Este selectată prima stare din lista posibilităților (și înlăturată din această listă).

2. Se generează noile stări prin încercarea fiecărei opțiuni posibile corespunzător stării selectate. (Se poate ca aceste noi stări să nu existe dacă ne aflăm pe un drum greșit).

3. Se adaugă stările generate la pasul **2** listei posibilităților.

În cazul organizării unei strategii de tip **depth-first**, lista posibilităților este o **stivă**. Cu alte cuvinte, pasul 1 preia întotdeauna primul element al listei, iar la pasul 3 noile stări generate sunt întotdeauna plasate la începutul listei, conducând la o strategie de tip LIFO. Prin contrast, în cazul organizării unei strategii de tip **breadth-first**, lista posibilităților este manipulată ca o **coadă**. Pasul 3 adaugă noile stări generate la sfârșitul listei, conducând la o strategie de tip FIFO.

Diferența dintre cele două abordări este esențială. Astfel, în cadrul strategiei depth-first, o anumită interpretare este luată în considerație și extinsă până când eșuează; numai după aceea este studiată o a doua. În cazul strategiei breadth-first ambele interpretări sunt luate în considerație alternativ. Cu alte cuvinte, căutarea de tip depth-first studiază, pe rând, câte o singură ipoteză. Căutarea de tip breadth-first studiază ipotezele în paralel. O strategie de tip depth-first adesea avansează cu mare viteză către o soluție, dar în alte ocazii poate consuma suficient de mult timp urmând niște piste false. Strategia breadth-first explorează fiecare soluție posibilă până la o anumită adâncime înainte de a avansa. Mulți analizori sintactici folosiți în prezent utilizează strategia depth-first, care tinde să minimizeze numărul stărilor backup necesare. Prin aceasta, ea utilizează mai puțină memorie și necesită ținerea unei evidențe de mai mică amploare (spre deosebire de căutarea de tip breadth-first, care

necesită scrierea unor programe ce pot manipula în mod explicit liste de stări alternative). Implementarea în Prolog a analizorilor sintactici ce utilizează strategia depth-first este, de asemenea, mult mai simplă, întrucât aceasta se bazează în mod esențial pe facilitatea Prologului de a efectua backtracking automat și deci căutare de tip depth-first.

Vom exemplifica, în continuare, analiza sintactică top-down efectuată cu Algoritmul 7.2 și implementând o strategie de căutare de tip depth-first, în cazul propoziției

$_1$ *Un* $_2$ *câine* $_3$ *latră* $_4$

Se utilizează gramatica și lexiconul definite anterior în legătură cu această propoziție. Pașii analizei sunt următorii:

Pas	Stare curentă	Stări backup	Comentarii
1.	((S)1)	-	Poziția inițială
2.	((NP VP)1)	-	Se rescrie S cu regula 1
3.	((ART N VP)1)	((ART ADJ N VP)1)	Se rescrie NP cu regulile 2 și 3
4.	((N VP)2)	((ART ADJ N VP)1)	Se reduce ART cu "Un"
5.	((VP)3)	((ART ADJ N VP)1)	Se reduce N cu "câine"
6.	((V)3)	((V NP)3) ((ART ADJ N VP)1)	Se rescrie VP cu regulile 4 și 5
7.	()4)	((V NP)3) ((ART ADJ N VP)1)	Se reduce V cu "latră"
8.	**STOP**	-	Propoziție corectă

Exemplul a fost preluat de noi din [1] și adaptat limbii române. Pentru alte exemple, referitoare la limba engleză, vezi, de asemenea, [1].

Analiza sintactică top-down, atunci când este efectuată de la stânga la dreapta, ca în exemplele discutate, va avea dificultăți cu acele reguli care prezintă *recursivitate la stânga*. O astfel de regulă este, de pildă,

$$VP \rightarrow VP\ NP\ .$$

Dacă o gramatică admite reguli recursive la stânga, un parser top-down, care lucrează de la stânga la dreapta, va intra într-un ciclu infinit atunci când face alegeri greșite. Iar dacă se cer toate structurile posibile ale propoziției, astfel de decizii sunt inerente, întrucât analizorul trebuie să facă toate alegerile posibile.

În cele ce urmează, pentru a facilita înțelegerea implementării în limbajul Prolog a analizei sintactice top-down vom privi o stare ca fiind

caracterizată exclusiv de o secvență de scopuri și de o altă secvență de cuvinte rămase.

7.3.1.2. Implementare Prolog

Implementarea Prolog, la care ne-am referit și la începutul acestei prezentări, este extrem de firească în ceea ce privește analizorii sintactici sau dispozitivele de recunoaștere top-down, care lucrează de la stânga la dreapta și utilizează o strategie de căutare de tip depth-first. Un parser care folosește, din nou, lista diferență este următorul:

```
recunoaste(Categorie,S1,S2):-
        regula(Categorie,Fii),imperecheaza(Fii,S1,S2).
```

Prin urmare, un șir poate fi *recunoscut* ca o realizare a lui **Categorie** dacă există o regulă care permite lui **Categorie** să domine **Fii** și dacă **Fii** se împerechează cu șirul respectiv. **Prin definiție**, vom considera că o listă de categorii se împerechează cu un șir dacă:

- ambele sunt vide;
- ambele constau dintr-un singur cuvânt;
- o porțiune inițială a șirului poate fi recunoscută ca reprezentând prima categorie din listă, iar restul listei de categorii se împerechează cu restul șirului.

Conform acestei descrieri, predicatul **imperecheaza** se definește în Prolog în felul următor:

```
imperecheaza([],S,S).
imperecheaza([Cuvant],[Cuvant|S],S).
imperecheaza([Categorie|Categorii],S1,S3):-
        recunoaste(Categorie,S1,S2),
        imperecheaza(Categorii,S2,S3).
```

În acest caz, regulile gramaticii trebuie date sub forma

```
regula(s,[np,vp]).
```

corespunzător regulilor PS și respectiv sub forma

```
regula(np,[surori]).
regula(v,[angajeaza]).
```

în cazul regulilor lexicale.

Interogarea se face în felul următor:

```
?- recunoaste(s,['Colentina',angajeaza,surori],[]).
```

Răspunsul sistemului corespunzător interogării de mai sus va fi

```
yes.
```

Dacă gramatica nu conține reguli lexicale, ci este atașat un **lexicon**, atunci regulile PS ale gramaticii vor fi reprezentate în același mod, iar intrările lexicale vor fi reprezentate sub forma

```
cuvant(np,surori).
cuvant(v,angajeaza).
```

În acest caz, utilizând Algoritmul 7.1, putem programa analizorul sintactic folosind predicatele **parse** și **parse_lista**. Predicatul **parse** analizează un constituent având categoria C, începând cu șirul de intrare S1 și sfârșind cu șirul de intrare S. În definirea acestui predicat vom ține seama de cele două situații prevăzute de Algoritmul 7.1. Predicatul **parse_lista** este identic cu predicatul **parse**, cu excepția faptului că se referă la o listă de constituenți și îi analizează pe fiecare în parte. Interogarea Prologului se realizează în același mod. Analizorul sintactic corespunzător Gramaticii nr.1, însoțite de **lexiconul** aferent (și în care corespondența dintre predicatele **parse** și **recunoaste** și respectiv **parse_lista** și **imperecheaza** este evidentă) va fi dat de următorul program scris în SICStus Prolog:

Programul 7.1

```
parse(C,[Cuvant|S],S):-cuvant(C,Cuvant).
parse(C,S1,S):-regula(C,Cs),parse_lista(Cs,S1,S).
parse_lista([C|Cs],S1,S):-parse(C,S1,S2),
                          parse_lista(Cs,S2,S).
parse_lista([],S,S).

regula(s,[np,vp]).
regula(vp,[v]).
regula(vp,[v,np]).

cuvant(np,'Colentina').
cuvant(np,'Dr.Popescu').
```

```
cuvant(np,pacienti).
cuvant(np,surori).
cuvant(v,omoara).
cuvant(v,angajeaza).
```

7.3.2. Analiza sintactică bottom-up

Deşi traducerea standard în programe Prolog a gramaticilor DC dă naştere unor analizori sintactici top-down, aceasta nu reprezintă, în nici un caz, unica traducere posibilă. Pe de altă parte, o importantă limitare a unui parser top-down este aceea că se ajunge la ciclare atunci când acesta tratează reguli recursive la stânga de tipul $A \rightarrow AB$, reguli care, de altfel, apar în limbajul natural. Un exemplu de astfel de regulă este

$$NP \rightarrow NP\ Conj\ NP$$

unde *Conj* este o conjuncţie de tipul şi/sau.

O modalitate de a trata regulile recursive la stânga o reprezintă utilizarea unui parser de tip *bottom-up*, care acceptă cuvinte şi încearcă să le combine în constituenţi, utilizând regulile unei gramatici date. Aşa cum s-a mai specificat, analiza sintactică care utilizează o *strategie bottom-up* (adică "de jos în sus") pleacă de la cuvintele propoziţiei şi, prin procedee de deplasare-reducere (căutând membrii drepţi ai regulilor şi înlocuindu-i cu membrii stângi corespunzători), urmăreşte construirea arborelui de derivare de jos în sus, până când se ajunge la simbolul rădăcină S.

În cazul aceleiaşi propoziţii oferite spre exemplificare

Un câine latră.

analizorul sintactic bottom-up lucrează în felul următor:

Se acceptă un cuvânt al propoziţiei - *un*.
Cuvântul *un* este un ART.
Se acceptă un alt cuvânt - *câine*.
Cuvântul *câine* este un N.
Un ART şi un N alcătuiesc un NP.
Se acceptă un nou cuvânt - *latră*.
Cuvântul *latră* este un V.
Un V singur constituie un VP.
Un NP şi un VP alcătuiesc un S.

O trăsătură importantă a acestui parser este aceea că, întrucât acţiunile sale sunt determinate numai de cuvinte efectiv găsite în cadrul propoziţiei, el nu prezintă limitarea de a cicla atunci când intervin reguli recursive la stânga.

Acest analizor sintactic suferă, în schimb, de o altă limitare, aceea de a nu putea trata reguli de tipul

$$A \rightarrow \phi$$

deoarece nu dispune de nici o modalitate de a reacţiona în faţa unui constituent nul (care lipseşte).

7.3.2.1. Algoritm de deplasare-reducere

Analiza sintactică de tip bottom-up este adesea numită **parsing cu deplasare-reducere** [5]. Ea constă din două operaţii de bază: *deplasarea* cuvintelor în interiorul unei stive şi respectiv *reducerea* conţinutului stivei. Dăm, în continuare, analiza, privită în aceşti termeni, a aceluiaşi exemplu, adică a propoziţiei

Un câine latră.

Pas	Acţiune	Stivă	Şir de intrare
	START		*un câine latră*
1	Deplasare	*un*	*câine latră*
2	Reducere	*ART*	*câine latră*
3	Deplasare	*ART câine*	*latră*
4	Reducere	*ART N*	*latră*
5	Reducere	*NP*	*latră*
6	Deplasare	*NP latră*	-
7	Reducere	*NP V*	-
8	Reducere	*NP VP*	-
9	Reducere	*S*	-

Algoritmul de deplasare-reducere poate fi, prin urmare, formulat în felul următor:

Algoritmul 7.3

1. Se deplasează un cuvânt în stivă.
2. Se reduce stiva în mod repetat, utilizând intrările lexicale şi regulile PS, până când această operaţie nu mai este posibilă.

3. Dacă mai există cuvinte în şirul de intrare se merge la Pasul 1; altfel **STOP**.

❑

Este de remarcat faptul că, un parser bazat pe acest algoritm, spre deosebire de cel top-down, nu lucrează în manieră predictivă, în sensul că nu se aşteaptă, la nici un pas al algoritmului, să găsească un constituent de un anumit tip.

7.3.2.2. Implementare Prolog

O implementare eficientă în Prolog a parsing-ului cu deplasare-reducere necesită, în esenţă, utilizarea a două tipuri de tehnici, după cum urmează:

1. *Construcţia stivei se face înapoi*, prin aceasta înţelegându-se că se deplasează cuvintele de la *începutul* şirului de intrare la *începutul* stivei, ca în exemplul:

START:	[ ]	[un,caine,latra]
Deplasare:	[un]	[caine,latra]
Reducere:	[art]	[caine,latra]
Deplasare:	[caine,art]	[latra]
Reducere:	[n,art]	[latra]
Reducere:	[np]	[latra]
Deplasare:	[latra,np]	[ ]
Reducere:	[v,np]	[ ]
Reducere:	[vp,np]	[ ]
Reducere:	[s]	[ ]

Această tehnică este eficientă deoarece face ca toate acţiunile să aibă loc la începutul fiecărei liste. Listele nu trebuie, prin urmare, parcurse, pentru a se ajunge la sfârşitul lor.

2. *Regulile se memorează înapoi.* Spre exemplu, regula

$NP \rightarrow ART\ N$

va fi transpusă în Prolog sub forma

```
iregula([n,art|X],[np|X]).
```

(Denumirea "iregula" sugerează o "regulă inversă" sau "regulă înapoi"). Ca urmare, pasul 2 al algoritmului anterior (pasul de reducere) se va executa mult mai rapid. Primul argument al regulii menţionate, `[n,art|X]` coincide cu vârful stivei căreia i se aplică această regulă, în timp ce `[np|X]` reprezintă ceea ce devine stiva după reducere. Prin urmare, pasul de reducere al analizorului este foarte simplu: dacă există o regulă aplicabilă, aceasta este utilizată, după care se încearcă efectuarea unei noi reduceri; altfel, stiva este lăsată neschimbată. Vom reveni asupra unei variante mai performante a acestui algoritm, variantă care utilizează o structură de date numită "hartă".

Un **parser bottom-up cu deplasare-reducere**, care se bazează pe cele două tehnici amintite, de construcţie a stivei şi respectiv de memorare a regulilor, şi care foloseşte Gramatica nr.1, este prezentat în continuare. Analizorul sintactic utilizează predicatul **parse (S, Rezultat)**, care analizează şirul de intrare S, pe care îl reduce la lista de categorii Rezultat. Programul corespunzător, scris în SICStus Prolog, este următorul:

Programul 7.2

```
parse(S,Rezultat):-depl_red(S,[],Rezultat).

% Predicatul depl_red(S,Stiva,Rezultat) analizeaza sirul de
% intrare S, unde Stiva este lista categoriilor analizate
% pana la momentul curent. Atunci cand sirul de intrare
% devine vid, inseamna ca toate cuvintele au fost deplasate
% in stiva si reduse, deci stiva contine chiar rezultatul.
% Predicatul depl_red se defineste recursiv astfel:

depl_red(S,Stiva,Rezultat):-deplasare(Stiva,S,StivaNoua,S1),
                            reducere(StivaNoua,StivaRedusa),
                            depl_red(S1,StivaRedusa,Rezultat).
depl_red([],Rezultat,Rezultat).

% OBSERVATIE: predicatul deplasare esueaza daca sirul de
% intrare este vid. Sirul de intrare devine vid atunci cand
% toate cuvintele sale au fost deplasate in stiva si reduse,
% deci in cazul tratat de clauza depl_red anterioara.
% Altfel, in cazul general, predicatul deplasare se defineste
% astfel:

deplasare(X,[H|Y],[H|X],Y).
```

```
% Predicatul reducere(Stiva,StivaRedusa) reduce in mod repe-
% tat varful stivei, pentru a forma constituenti mai putini,
% dar mai ampli. Definitia recursiva este:

reducere(Stiva,StivaRedusa):-iregula(Stiva,Stiva1),
                             reducere(Stiva1,StivaRedusa).
reducere(Stiva,Stiva).

% Definitia a tinut cont de cazul cand nu exista nici
% o iregula, deci nu se poate efectua nici o reducere. La
% clauza reducere(Stiva,Stiva). se ajunge atunci cand nu se
% poate aplica nici o iregula si stiva ramane neschimbata.

% Regulile PS ale gramaticii se memoreaza tot inapoi.

iregula([vp,np|X],[s|X]).
iregula([v|X],[vp|X]).
iregula([np,v|X],[vp|X]).

% Pentru ca lexiconul sa poata fi utilizat este nevoie, in
% plus, de specificarea urmatoarei "reguli inapoi":

iregula([Cuvant|X],[Cat|X]):-cuvant(Cat,Cuvant).

% Lexiconul este:

cuvant(np,'Dr.Popescu').
cuvant(np,'Colentina').
cuvant(np,surori).
cuvant(np,pacienti).
cuvant(v,omoara).
cuvant(v,angajeaza).
```

Interogarea Prologului se face astfel:

```
?- parse(['Colentina',angajeaza,surori],[s]).
```

Răspunsul sistemului va fi `yes`, ceea ce arată că `[s]` reprezintă, într-adevăr, lista categoriilor la care se reduce şirul de intrare. Cu alte cuvinte, propoziţia

Colentina angajează surori.

a fost analizată sintactic cu succes, în sensul că ea este acceptată ca fiind corectă (în conformitate cu gramatica dată). O interogare a Prologului de tipul

```
?- parse(['Colentina',angajeaza,surori],[X]).
```

are ca răspuns

```
X=s?
```

Dacă, în continuare, se apasă tasta <Enter>, răspunsul sistemului este `yes`, ceea ce arată că şirul de intrare se poate reduce la categoria `s` şi deci reprezintă o propoziţie corectă în conformitate cu gramatica dată. Dacă, în loc de a se apăsa tasta <Enter>, se tastează "punct şi virgulă" [;], răspunsul sistemului este `no`, cu sensul că şirul de intrare se reduce la categoria `s` şi că aceasta este unica soluţie posibilă.

7.3.3. Analiza sintactică din colţul stâng

Aşa-numita "analiză sintactică din colţul stâng" este adesea descrisă ca fiind de tip bottom-up, dar ea reprezintă, în realitate, o combinaţie a strategiilor bottom-up şi top-down [5]. Această tehnică a fost prezentată de către Rosenkrantz şi Lewis (1970), precum şi de către Aho şi Ullman (1972), toţi aceşti autori atribuindu-i-o lui Irons (1961).

Ideea centrală care stă la baza acestui tip de analiză este aceea de a accepta un cuvânt, a determina tipul de constituent al cărui început îl marchează cuvântul respectiv şi a analiza apoi restul acelui constituent în manieră top-down. Arborele de derivare corespunzător propoziţiei este astfel "descoperit" începând din colţul din stânga jos.

Ca şi un parser top-down, un analizor sintactic "din colţul stâng" se aşteaptă întotdeauna să găsească un anumit tip de constituent şi, prin urmare, ştie că numai câteva dintre regulile gramaticii sunt relevante. Aceasta îl face mult mai eficient decât analizorul sintactic de tip bottom-up descris anterior. În plus, ca şi acesta din urmă, el poate trata regulile recursive la stânga fără a cicla, deoarece începe analiza fiecărui constituent prin acceptarea unui cuvânt din şirul de intrare.

Algoritmul de analiză sintactică corespunzător constă din două părţi distincte, şi anume: acceptarea şi identificarea unui cuvânt, urmate de formarea constituentului corespunzător. Algoritmul pentru analiza *unui constituent de tipul C* este următorul:

Algoritmul 7.4

1. Se acceptă un cuvânt din şirul de intrare şi i se determină categoria. Fie această categorie **W**.
2. Se completează **C** astfel: dacă **W=C**, atunci **STOP**; altfel:
   - cu ajutorul regulilor gramaticii se găseşte un constituent a cărui dezvoltare începe cu **W**; fie **P** acest constituent[33];
   - se analizează în mod recursiv şi utilizând strategia **top-down** toate elementele rămase ale dezvoltării lui **P**;
   - se înlocuieşte **W** cu **P** şi se merge înapoi la începutul pasului 2 (pentru a continua completarea lui **C**).

□

7.3.3.1. Implementare Prolog

Implementarea Prolog va utiliza predicatele **parse**, **parse_lista** şi **completeaza**.

Predicatul **parse(C, S1, S)** analizează un constituent de categorie **C**, începând cu şirul de intrare S1 şi încheind cu şirul de intrare S. Analiza se face conform Algoritmului 7.4, deci presupune acceptarea unui cuvânt de categorie **W** din şirul de intrare, urmată de completarea constituentului de tipul **C**:

```
parse(C,[Cuvant|S2],S):-cuvant(W,Cuvant),
                        completeaza(W,C,S2,S).
```

Predicatul **completeaza (W, C, S1, S)** verifică dacă **W** poate fi primul subconstituent al lui **C**, apoi analizează restul constituentului de tipul **C**. Dacă **C=W**, analiza se încheie

```
completeaza(C,C,S,S).
```

altfel continuă în mod recursiv:

```
completeaza(W,C,S1,S):-regula(P,[W|Rest]),
                       parse_lista(Rest,S1,S2),
                       completeaza(P,C,S2,S).
```

[33] Spre exemplu, dacă categoria **W** este *Det*, iar gramatica conţine regula PS *NP→Det N*, atunci se utilizează această regulă, care precizează pe Det N ca reprezentând o posibilă dezvoltare a lui NP şi se determină **P** ca fiind grup nominal (NP).

Lista de constituenţi **Rest** este analizată urmând strategia clasică top-down şi, prin urmare, se utilizează acelaşi predicat **parse_lista** care a fost definit cu prilejul implementării acestui tip de analiză sintactică (§7.3.1.2.). Regulile PS ale gramaticii şi lexiconul aferent vor fi reprezentate tot ca în cazul analizorului sintactic top-down. În aceste condiţii, interogarea Prologului se face, de asemenea, în acelaşi mod (§ 7.3.1.2.).

În cele ce urmează, prezentăm un parser (imperfect), care efectuează analiză sintactică din colţul stâng, corespunzător Gramaticii nr.2. Programul aferent, scris în SICStus Prolog, este următorul:

Programul 7.3

```
parse(C,[Cuvant|S2],S):-cuvant(W,Cuvant),
                        completeaza(W,C,S2,S).

parse_lista([C|Cs],S1,S):-parse(C,S1,S2),
                          parse_lista(Cs,S2,S).
parse_lista([],S,S).

completeaza(C,C,S,S).
completeaza(W,C,S1,S):-regula(P,[W|Rest]),
                       parse_lista(Rest,S1,S2),
                       completeaza(P,C,S2,S).

regula(s,[np,vp]).
regula(vp,[v,np]).
regula(np,[det,n]).

cuvant(det,un).
cuvant(det,niste).
cuvant(det,o).
cuvant(n,elev).
cuvant(n,elevi).
cuvant(n,carte).
cuvant(v,iubeste).
cuvant(v,iubesc).
```

7.3.3.1.1. Legături

Aşa cum a fost descris până în prezent, conform [5], parser-ul care efectuează analiză sintactică din colţul stâng poate numai parţial să trateze regulile de tipul $A \rightarrow \phi$, adică acele reguli care introduc constituenţi nuli. În esenţă, putem spune că nu va fi nici o problemă atunci când constituentul nul este întâlnit în timp ce se analizează sintactic top-down, întrucât parser-ul descris lucrează, în momentul respectiv, exact ca un analizor sintactic de acest tip. Aceasta va fi situaţia atunci când se aplică reguli de forma

$$A \rightarrow B\ C$$
$$C \rightarrow \phi$$

deoarece constituentul C va fi analizat top-down. Trebuie doar să i se specifice analizorului sintactic că, în situaţia în care se caută un C, se poate merge mai departe fără a analiza nimic. Această specificare poate fi realizată printr-o codificare a regulilor de tipul

```
regula(a,[b,c]).
regula(c,[]).
```

şi prin adăugarea următoarei clauze Prolog referitoare la predicatul **parse**:

```
parse(C,S2,S):-regula(W,[]),completeaza(W,C,S2,S).
```

(Această clauză spune că, dacă W este o categorie care poate fi reprezentată printr-un constituent nul, atunci este permisă oricând completarea lui W practic neîntreprinzând nici o acţiune).

Această completare a analizorului sintactic descris nu este însă suficientă pentru limbi ca engleza sau româna, limbi în care o regulă de tipul

$$NP \rightarrow Det\ N$$

poate fi însoţită de regula

$$Det \rightarrow \phi$$

Cu alte cuvinte constituentul nul poate fi un determinant nul, care apare la începutul unui grup nominal şi care, prin urmare, va fi analizat bottom-up. Dacă, spre exemplu, determinantul este un articol cu care începe un grup nominal ("o fată"), conform regulii

$$NP \rightarrow ART\ N$$

atunci, în limba română, putem avea tripletul de reguli

$S \rightarrow NP\ VP$ (*o fată citeşte; fata citeşte*)

$NP \rightarrow ART\ N$ (*o fată*)

$ART \rightarrow \phi$ (*fata*)

care spune analizorului sintactic să accepte determinantul nul ca reprezentând primul pas al analizei unui NP sau al analizei unui S. În cazul existenţei unor astfel de reguli, analizorul sintactic descris va cicla.

Problema constă în aceea că analizorului de tip "din colţul stâng" i se permite, ca şi celui bottom-up, să accepte constituenţi nuli oricând doreşte acest lucru. Soluţia acestei probleme este aceea de a constrânge parser-ul prin adăugarea unui **tabel de legături** [5]. În afară de rezolvarea problemei constituenţilor nuli, astfel de tabele fac ca analizorul sintactic să devină mult mai eficient, aşa cum se va vedea în cele ce urmează.

Tabelul de legături are rolul de a specifica *ce tipuri de constituenţi pot să apară pe poziţii de început*. Spre exemplu, tripletul de reguli

$$S \rightarrow NP\ VP$$
$$NP \rightarrow Det\ N$$
$$VP \rightarrow V\ NP$$

defineşte următoarele legături:

```
legatura(np,s).
legatura(det,np).
legatura(det,s).
legatura(v,vp).
legatura(X,X).
```

Ultima dintre acestea afirmă că orice constituent poate începe cu el însuşi. Ea se aplică atunci când, spre exemplu, analizorul caută un constituent de tipul N şi, în acelaşi timp, acceptă un substantiv. Introducerea tabelului de legături măreşte eficienţa parser-ului prin aceea că rezolvă problema existenţei intrărilor lexicale multiple corespunzătoare aceluiaşi cuvânt. Acest tabel introduce o *relaţie* numită **legătura**, care spune că: o categorie **C1** este legată de o categorie **C2** dacă şi numai dacă **C1** poate interveni în calitate de *constituent iniţial* al lui **C2**. În mod evident, această relaţie este reflexivă şi tranzitivă, iar luarea în consideraţie a ei rezolvă problema consumului excesiv de timp calculator în cazul acceptării de către parser a unui cuvânt ce aparţine mai multor categorii lexicale. Predicatul **parse** trebuie modificat în consecinţă, adică în aşa fel încât să existe şi să fie exprimată o legătură între constituentul căutat la un anumit

moment şi cuvântul acceptat din şirul de intrare. Această legătură se exprimă în felul următor:

```
parse(C,[Cuvant|S2],S):-cuvant(W,Cuvant),
                        legatura(W,C),
                        completeaza(W,C,S2,S).
parse(C,S2,S):-regula(W,[]),
               % pentru constituenti nuli
               legatura(W,C),
               completeaza(W,C,S2,S).
```

Prin modificarea primei clauze Prolog amintite şi prin adăugarea celei de a doua, problema este rezolvată, în sensul că analizorul nu mai poate accepta constituenţi nuli în locaţii arbitrare. Prezentăm, în continuare, analizorul sintactic astfel modificat (şi utilizând tot Gramatica nr.2). Programul corespunzător, scris în SICStus Prolog, este următorul:

Programul 7.4

```
parse(C,[Cuvant|S2],S):-cuvant(W,Cuvant),
                        legatura(W,C),
                        completeaza(W,C,S2,S).
parse(C,S2,S):-regula(W,[]),
               legatura(W,C),
               completeaza(W,C,S2,S).

parse_lista([C|Cs],S1,S):-parse(C,S1,S2),
                          parse_lista(Cs,S2,S).
parse_lista([],S,S).

completeaza(C,C,S,S).
completeaza(W,C,S1,S):-regula(P,[W|Rest]),
                       parse_lista(Rest,S1,S2),
                       completeaza(P,C,S2,S).

regula(s,[np,vp]).
regula(vp,[]).
```



```
legatura(np,s).
regula(vp,[v,np]).
regula(np,[]).
legatura(v,vp).
regula(np,[det,n]).
regula(n,[]).
legatura(det,np).
legatura(det,s).
legatura(X,X).

cuvant(det,un).
cuvant(det,niste).
cuvant(det,o).
cuvant(n,elev).
cuvant(n,elevi).
cuvant(n,carte).
cuvant(v,iubeste).
cuvant(v,iubesc).
```

Acest parser poate încă să cicleze, şi anume atunci când întâlneşte un set de reguli de forma

$$A \rightarrow B\ A$$

$$B \rightarrow \phi$$

dar, în acest caz, ciclarea se datorează *regulilor în sine* şi nu analizorului sintactic. (Aceste reguli specifică un număr infinit de analize sintactice diferite pentru fiecare şir de intrare, după cum urmează: un A poate fi format dintr-un constituent B nul urmat de un alt A, ori din doi constituenţi nuli de tip B urmaţi de un alt A, ori din trei constituenţi nuli de tip B urmaţi de un alt A ş.a.m.d.. Analizorul sintactic va executa un proces de backtracking infinit în încercarea de a găsi toate analizele posibile). Problema persistă în cazul tuturor gramaticilor care specifică, prin regulile lor, un număr infinit de analize sintactice.

7.3.3.1.2. BUP

Cea mai cunoscută implementare în Prolog a analizei sintactice din colţul stâng o reprezintă, după toate probabilităţile, aşa-numita BUP, concepută de către Matsumoto, Tanaka, Hirakawa, Miyoshi şi Yasukawa (1983).

Implementarea BUP (de la "bottom-up parser") transformă fiecare regulă PS într-o clauză Prolog al cărei cap nu este dat de nodul părinte ci de fiul cel mai din stânga. Astfel, regula

NP → Det N PP

se transformă în clauza Prolog

```
det(C,S1,S):-parse(n,S1,S2),parse(pp,S2,S3),np(C,S3,S).
```

Această clauză Prolog spune că "dacă tocmai s-a completat un constituent de tip Det, atunci se analizează un N şi apoi un PP; după care se apelează procedura care tratează un constituent completat de tip NP". Subliniem faptul că C reprezintă constituentul de *rang cel mai înalt* pe care analizorul sintactic încearcă să îl completeze. În acelaşi timp, S1, S2, S3 şi S reprezintă şirul de intrare în diverse momente ale analizei.

Pe lângă câte o clauză Prolog corespunzând fiecăreia dintre regulile PS, implementarea BUP necesită şi o aşa-numită "clauză de terminare" pentru fiecare tip de constituent. Astfel de clauze sunt:

```
np(np,S,S).
n(n,S,S).
det(det,S,S).
vp(vp,S,S).
v(v,S,S).
s(s,S,S).
```

Clauza

```
np(np,S,S).
```

afirmă, spre exemplu, că "dacă tocmai s-a acceptat un NP şi ceea ce se căuta era un grup nominal, atunci căutarea s-a încheiat". Prezentăm, în continuare, un parser BUP fără legături care efectuează analiză sintactică din colţul stâng şi care utilizează tot Gramatica nr.2. Predicatul **parse(C, S1, S)** analizează un constituent având categoria C, plecând de la şirul de intrare S1 şi ajungând la şirul de intrare S. Analizorul sintactic este cel din programul care urmează, program scris în SICStus Prolog:

Programul 7.5

```
parse(C,S1,S):-cuvant(W,S1,S2),
               P=..[W,C,S2,S],
               Call(P).

% Reguli PS si clauze de terminare

np(C,S1,S):-parse(vp,S1,S2),s(C,S2,S).
np(np,X,X).
det(C,S1,S):-parse(n,S1,S2),np(C,S2,S).
det(det,X,X).
v(C,S1,S):-parse(np,S1,S2),vp(C,S2,S).
v(v,X,X).

s(s,X,X).
vp(vp,X,X).
pp(pp,X,X).
n(n,X,X).

% Lexicon

cuvant(det,[un|X],X).
cuvant(det,[niste|X],X).
cuvant(det,[o|X],X).
cuvant(n,[elev|X],X).
cuvant(n,[elevi|X],X).
cuvant(n,[carte|X],X).
cuvant(v,[iubeste|X],X).
cuvant(v,[iubesc|X],X).
```

Interogarea Prologului se face în acelaşi mod:

```
?- parse(s,[niste,elevi,iubesc,o,carte],[]).
yes
```

sau

```
?- parse(s,[niste,elevi,iubesc],[]).
no
```

Se remarcă faptul că acest parser de tip BUP nu recunoaşte propoziţia

Nişte elevi iubesc.

ca fiind una corect generată în raport cu Gramatica nr.2, în timp ce parser-ul prezentat anterior (§7.3.3.1.1.) o va recunoaşte ca atare. (După acceptarea verbului "iubesc", acesta din urmă îşi propune să analizeze top-down un constituent de tip VP prin admiterea unui constituent NP nul). Pentru a accepta (analiza cu succes) această propoziţie, parser-ul BUP necesită introducerea în mod explicit în gramatică a regulii suplimentare $VP \rightarrow V$, sub forma:

```
v(C,S1,S):-vp(C,S1,S).
```

Implementarea BUP este extrem de eficientă datorită faptului că, aşa cum rezultă şi din programul Prolog prezentat, partea foarte delicată a procesului de căutare, aceea care urmează după completarea unui fiu aflat cel mai în stânga, este rezolvată de către cel mai rapid mecanism de căutare al Prologului, şi anume mecanismul de găsire a unei clauze fiind dat predicatul corespunzător.

7.3.4. Asupra eficienţei strategiilor de tip bottom-up şi top-down; ambiguitate

O primă *concluzie* care a fost pusă, deja, în evidenţă, este aceea că o strategie *pură* de tip bottom-up sau top-down nu va fi suficientă pentru a realiza o analiză sintactică eficientă. (Vom reveni asupra noţiunii de eficienţă în acest context).

În acelaşi timp, *o primă problemă* care se ridică în urma prezentării tehnicilor de analiză sintactică de până acum este aceea că analizorii sintactici discutaţi până în prezent nu memorează **rezultatele intermediare**. În consecinţă, ei verifică din nou aceleaşi lucruri, care nu aveau cum să se schimbe până la momentul curent, consumând timp în mod inutil şi afectând în acest fel execuţia programului. Este motivul pentru care vom studia aşa-numiţii **analizori sintactici cu hartă**. Un analizor sintactic cu hartă foloseşte *o structură de date numită hartă* pentru a memora toate rezultatele intermediare obţinute în cursul unei analize. Această creştere a eficienţei va avea însă un cost, care nu este numai unul de mărire a complexităţii implementării: programul de analiză corespunzător va avea nevoie de mai mult spaţiu.

O *a doua problemă* care se ridică, în mod natural, în acest context, este aceea a **eficienţei** programelor de analiză sintactică. Eficienţa nu înseamnă numai spaţiul utilizat de program sau timpul necesar pentru recunoaşterea unui şir, adică timpul cerut pentru a decide dacă acesta este corect din punct de vedere gramatical. Contează, în plus, timpul utilizat pentru a enumera *toate analizele posibile* ale unui şir de intrare, iar acesta este influenţat de *gradul de ambiguitate* al propoziţiei (şi s-a arătat că este, în cel mai rău caz, exponenţial).

Pentru a vedea un exemplu de ambiguitate, vom adăuga o nouă intrare lexiconului Gramaticii nr.1, după cum urmează:

Cuvânt surori:

<cat> = V.

Aceasta permite Gramaticii nr.1 să admită propoziţii ca

Dr. Popescu surori pacienţi.

deoarece lexiconul său a fost afectat de o *ambiguitate lexicală* cauzată de apartenenţa cuvântului *surori* la două categorii lexicale diferite. Această ambiguitate lexicală va face ca cel puţin unul dintre analizorii sintactici descrişi până acum să lucreze suplimentar: un parser bottom-up care găseşte cuvântul "surori" îl va atribui pe acesta atât categoriei V, cât şi categoriei NP şi va explora fiecare dintre aceste posibilităţi.

Ambiguitatea lexicală de acest tip este omniprezentă în limba engleză. Se ştie că majoritatea cuvintelor englezeşti sunt substantive şi că multe substantive englezeşti pot fi folosite ca verbe. În general, majoritatea cuvintelor englezeşti pot aparţine mai multor părţi de vorbire, deci mai multor categorii lexicale, ceea ce determină existenţa unui mare număr de posibilităţi pentru generarea ambiguităţii. De asemenea, marea majoritate a cuvintelor englezeşti sunt polisemantice. Dacă un parser aparţine unui sistem care are rolul de a realiza corespondenţa dintre propoziţii englezeşti şi înţelesuri ale lor, atunci el se va confrunta cu ambele aspecte ale ambiguităţii lexicale amintite. Prin contrast, un sistem experimental care nu face decât să atribuie structuri sintactice unor şiruri de intrare, poate ignora acele ambiguităţi lexicale de ordin semantic care nu sunt dublate de ambiguitate generată de apartenenţa la diverse categorii lexicale. Aceleaşi probleme referitor la ambiguitatea lexicală se ridică şi în cazul limbii române. Şi în această limbă multe cuvinte sunt polisemantice şi pot aparţine unor categorii lexicale diferite. Numeroase cuvinte, spre exemplu, pot fi atât substantive, cât şi verbe (*cânt, blestem, botez*) ori atât substantive, cât şi adjective, cât şi adverbe (*bun*). De asemenea, în limba română, orice verb la participiul trecut poate funcţiona şi ca adjectiv (ca în *uşa stă deschisă* şi respectiv *uşă deschisă*). Toate acestea, ca şi multe alte fenomene întâlnite în limba română, sunt generatoare de ambiguitate lexicală.

În principiu, trebuie făcută o distincţie clară între ambiguitatea lexicală şi **ambiguitatea structurală** deşi, în practică, prima o generează adesea pe cea din urmă, cele două fiind, prin urmare, legate între ele. Pentru a pune în mod clar în evidenţă diferenţa dintre cele două tipuri de ambiguitate, vom spune că propoziţia

Merg spre bancă.

oferă un exemplu tipic de ambiguitate lexicală pură, generată de cele două sensuri diferite ale substantivului *bancă*, în timp ce propoziţia

Observ băiatul cu un telescop.

conţine o ambiguitate structurală pură, corespunzând celor două situaţii în care fie vorbitorul, fie băiatul, reprezintă posesorul telescopului. Un alt exemplu de ambiguitate structurală este cel din *frumoasa fată bogată*, fragment de propoziţie pe care un parser îl poate analiza în două moduri diferite:

((frumoasa fată) bogată)

(frumoasa (fată bogată))

S-a arătat că în numeroase limbi (şi cu precădere în limba engleză) există trei surse principale ale ambiguităţii structurale pure, şi anume: ataşarea în propoziţie a grupurilor prepoziţionale, coordonarea şi alăturarea substantivelor. Pentru exemple referitoare la limba engleză şi corespunzând tuturor acestor situaţii, vezi [9].

Discuţia de până acum referitoare la ambiguitate s-a ocupat, aproape în exclusivitate, de aşa-numita **ambiguitate globală**. Ambiguitatea globală se manifestă în acele cazuri în care unei expresii i se atribuie în mod corect două sau mai multe structuri, iar acele structuri se transmit, adică persistă în cadrul structurilor mai ample din care expresia face parte. În proiectarea şi implementarea analizorilor sintactici suscită, în egală măsură, interes şi fenomenul de **ambiguitate locală**. Aceasta se manifestă în acele cazuri în care unei expresii i se atribuie în mod corect două sau mai multe structuri, care s-ar putea să persiste sau nu în cadrul structurilor mai ample din care expresia face parte. Trebuie luat în consideraţie faptul că analiza sintactică a unei propoziţii în totalitate neambigue poate presupune explorarea mai multor situaţii de ambiguitate locală în timpul procesului de căutare a unicei descrieri structurale a întregii propoziţii. Activităţile dedicate eliminării sau minimizării ambiguităţii locale consumă mult timp calculator, din care cauză numeroase eforturi ale lingvisticii computaţionale s-au concentrat în această direcţie. La originea acestor eforturi se află lucrările lui M. P. Marcus (1980), care îşi propune să construiască analizori sintactici ce analizează propoziţii neambigue în mod determinist. Aceasta înseamnă că un parser nu este indus în eroare de ambiguităţile locale şi nu efectuează niciodată un proces de backtracking. În

realitate, un astfel de parser are la bază o tehnică (*lookahead*) prin care priveşte înainte în cadrul şirului de intrare, până la o distanţă determinată şi limitată înainte de a decide cum să interpreteze un cuvânt sau o secvenţă de cuvinte. Tehnica de *lookahead* poate fi privită ca o formă de backtracking cu multe constrângeri. Un parser de tip Marcus analizează numai limbaje independente de context, iar analizorii sintactici determinişti, în general, obţin o performanţă adecvată utilizând o combinaţie de *lookahead* şi euristică referitoare la frecvenţele relative ale diferitelor construcţii posibile. Întrucât ei nu formează obiectul acestei lucrări, vom reţine doar faptul că ambiguitatea locală şi cea globală constituie probleme cheie în analiza sintactică şi că un parser determinist încearcă să elimine ambiguitatea locală.

7.3.5. Memorarea rezultatelor intermediare

Aşa cum am mai remarcat, analizorii sintactici discutaţi până în prezent nu memorează rezultatele intermediare obţinute în cursul analizei, ceea ce face ca ei să efectueze aceleaşi verificări în repetate rânduri, consumând timp calculator şi afectând în acest fel execuţia programului. În cele ce urmează, vom studia o serie de tehnici ale analizei sintactice bazate tocmai pe memorarea rezultatelor intermediare (analiza sintactică cu hartă) şi vom urma prezentarea acestor tehnici aşa cum a fost ea realizată în [9].

7.3.5.1. Tabel de subşiruri bine formate

O primă structură de date care a fost introdusă [9] pentru ca un parser să memoreze în mod eficient rezultatele intermediare este aceea de **tabel de subşiruri bine formate (TSBF)**. Un TSBF reprezintă un mecanism prin care i se permite unui parser să păstreze o evidenţă a structurilor care au fost deja găsite, evitându-se astfel căutarea din nou a acestora. Încercăm să sugerăm, în cele ce urmează, cum s-a ajuns la soluţia utilizării unui TSBF.

După cum se ştie, există *redundanţă* în spaţiul de căutare al analizei (repetări ale aceloraşi situaţii). Un parser eficient trebuie, prin urmare, să fie înzestrat cu *memorie*, în sensul că el trebuie să poată înregistra constituenţii găsiţi deja, precum şi structura acestora. Cu alte cuvinte, trebuie memorată informaţie despre analizele parţiale alternative ale şirului de intrare, fără a se lua decizii premature cu privire la corectitudinea fiecăreia. Există şi alte situaţii şi/sau modalităţi în care o asemenea abilitate a parser-ului de a memora rezultate intermediare poate fi utilă. Spre exemplu, în cazul unui şir de intrare pe care gramatica dată nu îl acceptă ca fiind corect, un sistem robust nu se va mulţumi să producă un mesaj de tipul "Nu există analiză", ci va dori din partea componentei sintactice să pună la dispoziţia celorlalte componente (semantică şi pragmatică) cât mai multe informaţii posibil despre structura sintactică. Cu alte

cuvinte, un sistem robust de prelucrare a limbajului natural va fi interesat de valorificarea la maximum a rezultatelor intermediare găsite de componenta sintactică.

Există situații în care, fiind dată o gramatică, aceasta nu acceptă un anumit şir de intrare şi, prin urmare, nu îi poate atribui un singur arbore de derivare. (Reprezentarea structurală a unui şir neambiguu admis de către o gramatică se realizează cel mai adesea prin intermediul arborilor). În schimb, sistemul poate fi interesat de secvența de arbori care analizează diferite părți ale şirului de intrare, adică de reprezentarea *structurilor parțiale*. Gramatica cu caracteristici nr. 5 nu admitea propoziția corectă *Nişte elevi iubesc o carte*, dar majoritatea analizorilor sintactici vor găsi structurile parțiale prezentate in Fig. 7.3:

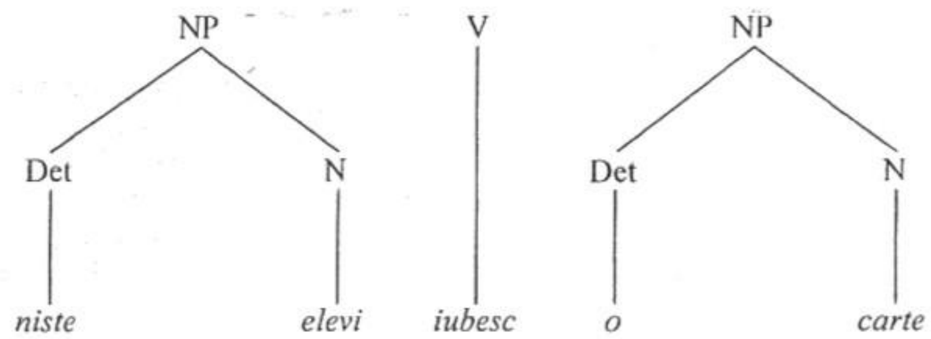


Fig. 7.3

(În general, o problemă importantă care se pune este aceea a reprezentării structurilor parțiale, care s-ar putea să facă parte sau nu din reprezentarea finală, corespunzătoare analizei complete).

În exemplul anterior, în care am avut un şir de intrare incorect din punctul de vedere al gramaticii date, gramatica furnizează mai puține informații decât ar oferi un singur arbore de structură complet. Există însă şiruri de intrare despre care se poate da mai multă informație decât cea furnizată de un unic arbore (complet). *Reprezentarea structurilor alternative* corespunzătoare unui acelaşi şir de intrare este o altă problemă cu care se confruntă orice analizor sintactic. Propoziției

Elevii ascultă un raport despre o călătorie.

îi corespund cele două reprezentări structurale din Fig. 7.4. Acest fapt se datorează *ambiguității structurale* care se manifestă în cazul propoziției enunțate, ambiguitate determinată de posibilitățile diferite de atașare ale grupului prepozițional (PP) existent:

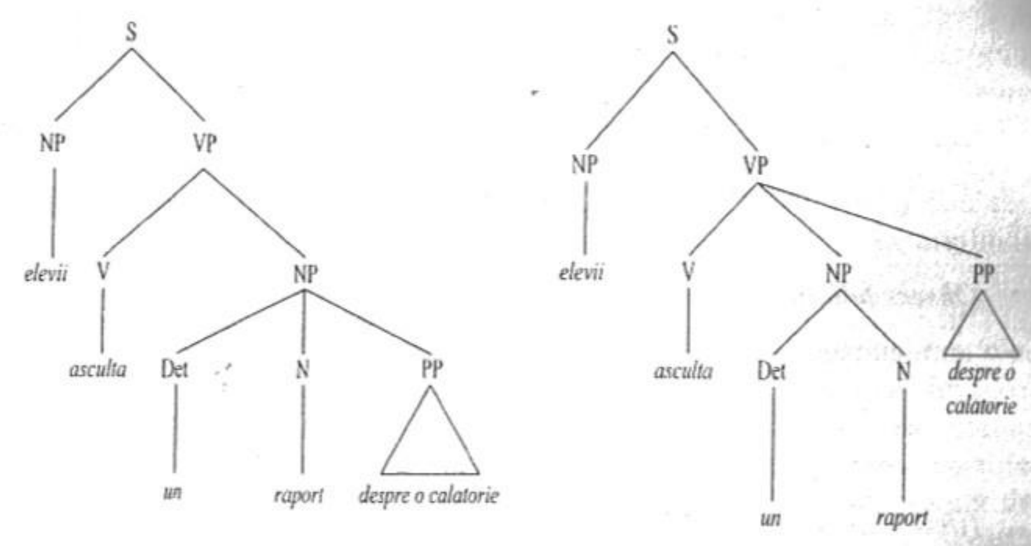


Fig. 7.4

Conform acestei discuții, problema memorării rezultatelor parțiale ale analizei (ca, de altfel, şi a celor finale), problemă pe care ne-am pus-o la începutul acestui paragraf, pare să ne conducă spre introducerea unei structuri de date constând dintr-o secvență de arbori ca cea din figura anterioară. Această structură de date poate însă deveni extrem de complexă, mai ales dacă se ia în considerație fenomenul de ambiguitate prezent în majoritatea limbilor naturale. În [9], a cărei linie de argumentație o urmăm aici, se propune ca primă soluție la aceste probleme de reprezentare structurală o tehnică prin care se reprezintă toată informația care ar fi necesară pentru a enumera secvențele posibile de arbori (i.e. arborii parțiali posibili, precum şi acele porțiuni din şirul de intrare cărora le corespund), fără a face propriu-zis enumerarea. Reprezentarea care rezultă în acest mod constituie un aşa-numit **tabel de subşiruri bine formate (TSBF)**.

Pentru a vedea în ce constă această reprezentare vom considera pozițiile de început şi de sfârşit ale şirului de intrare ca fiind numerotate *0* şi respectiv *n*, iar spațiile dintre cuvinte ca fiind numerotate de la stânga la dreapta, cu numere de la *1* la *n-1*. Un TSBF indică, pentru fiecare pereche (i,j) cu $0 \le i < j \le n$, categoriile care pot acoperi subşirul de cuvinte ce se găseşte între *i* şi *j*.
Modalitatea de a privi TSBF-urile pe care o adoptăm, în cele ce urmează, este aceea de a le considera nişte grafuri orientate, aciclice, având un unic vârf inițial şi un unic vârf final şi ale căror vârfuri sunt etichetate de la *0* (cel inițial) la *n* (cel final), unde *n* reprezintă numărul de cuvinte al şirului de intrare. Arcele

corespunzătoare sunt etichetate cu categorii (gramaticale şi lexicale) şi cu cuvinte, ca în Fig. 7.5:

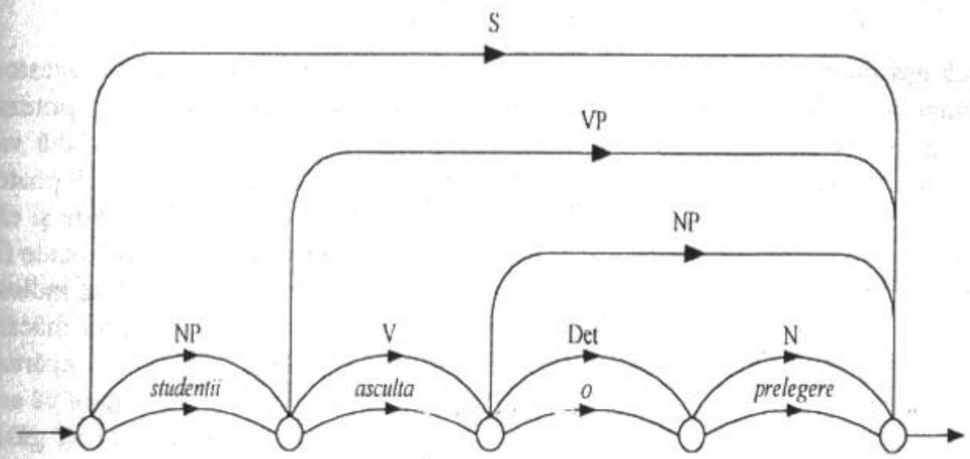


Fig. 7.5

Un TSBF poate fi reprezentat ca o mulţime de arce, unde un arc este o structură cu următoarele atribute:

<START> = ... un întreg ...
<FINAL> = ... un întreg ...
<ETICHETĂ> = ... o categorie ...

În mod evident, un *arbore PS de structură* de tipul convenţional este, de asemenea, un graf orientat aciclic. Acest graf are însă vârfurile etichetate cu categorii, în timp ce muchiile sunt neetichetate. În contextul utilizării unui TSBF arcele (muchiile) sunt purtătoare de etichete reprezentând categorii în timp ce vârfurile au, în esenţă, denumiri arbitrare. Această reprezentare a utilizat numerele întregi şi a exploatat relaţia ' < ' asupra întregilor pentru a codifica în mod implicit capetele muchiilor orientate. Se observă că un TSBF, spre deosebire de un arbore PS, codifică în mod direct *ordinea grupurilor sintactice*: un constituent căruia îi corespunde în tabel un arc de la i la j precede un constituent căruia îi corespunde un arc de la m la n numai în cazul $j<m$. Arborii, în schimb, codifică relaţiile de dominare imediată, indicând care grupuri sintactice sunt părţi constitutive ale altor grupuri sintactice.

Un TSBF, aşa cum a fost prezentat până acum, nu este suficient pentru a regăsi întreaga informaţie codificată într-o structură de date bazată pe arbori PS. Analizând un TSBF nu se poate determina, spre exemplu, regula gramaticii care legitimează un anume arc. În cazul TSBF-ului din Fig. 7.6, de pildă, nu se

poate decide dacă arcul etichetat cu H este legitimat de faptul că H domină D şi E sau de faptul că H domină F şi G:

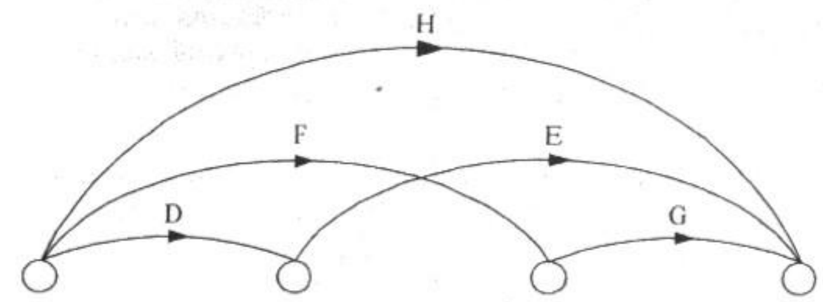


Fig. 7.6

Răspunsul la această întrebare nu poate fi dat fără a cunoaşte regulile gramaticii. Această informaţie ar putea fi disponibilă dacă se transformă eticheta unui arc dintr-o simplă categorie într-un arbore de derivare - o categorie împreună cu secvenţa de arce corespunzând subgrupurilor sintactice imediate.

Utilizarea unui TSBF permite unui parser să se abţină de la a redescoperi fapte care au fost deja stabilite. Aceste tabele pot fi utilizate de către orice tip de parser. Ele reprezintă o structură de date complet neutră, ca şi cea a arborilor PS. Un parser top-down, spre exemplu, poate folosi un TSBF în felul următor: se va impune cerinţa ca grupurile sintactice determinate să fie înregistrate ca intrări în diferite locaţii ale tabelului. Atunci când, la o anumită poziţie din şirul de intrare, se caută un grup sintactic de un anumit tip, tabelul va fi consultat. Dacă în tabel există o intrare de tipul dorit şi care începe la poziţia cerută, atunci se folosesc intrările din tabel ca reprezentând analizele posibile, fără ca această muncă să fie făcută, de fiecare dată, de către parser. Această strategie se bazează pe o anumită convenţie legată de modul de inserare a analizelor parţiale în tabel: o intrare corespunzând unei categorii date şi care începe la o poziţie dată nu este creată decât atunci când toate intrările de acest tip pot fi create. Altfel, cea de-a doua parte a procesului de analiză, care se bazează pe prima, nu va lua în calcul toate analizele posibile. O astfel de convenţie este necesară şi unui dispozitiv de recunoaştere, deoarece *analize posibile diferite ale unei categorii, care încep de la o anumită poziţie dată, pot avea poziţii de sfârşit diferite*.

Prin memorarea rezultatelor intermediare şi prin faptul că întotdeauna testează prezenţa unei categorii în tabel înainte de a încerca formarea acesteia, un parser care utilizează un TSBF poate evita executarea aceleiaşi operaţii de mai multe ori. Preţul care se plăteşte constă în spaţiul consumat, dar, întrucât un TSBF nu conţine redundanţă, cerinţa de spaţiu nu va fi niciodată mai mare decât un multiplu de n^2 (pentru o unică analiză). Evident, este vorba de găsirea *primei*

analize a unui şir de intrare. În practică, putem fi interesaţi de găsirea tuturor analizelor posibile, complexitatea acestei operaţii fiind, atunci când limbajul conţine ambiguitate arbitrară, de ordin exponenţial. Cu toate acestea, începând din anii 70, foarte mulţi analizori sintactici nedeterminişti ai limbajului natural au utilizat diverse variante de TSBF-uri.

7.3..5.2. Harta activă

Un TSBF de tipul celui prezentat este un instrument adecvat pentru a memora fapte referitoare la *structură*. Utilizarea unei asemenea structuri de date face, spre exemplu, ca un grup nominal determinat cu succes să nu trebuiască să fie găsit decât o singură dată. Analizorul sintactic nu va reuşi însă să evite reinvestigarea ipotezelor care anterior au eşuat. Pentru a se evita duplicarea tentativelor anterioare de a analiza într-un anume mod, este necesară existenţa unei reprezentări explicite a *ipotezelor* şi *ţelurilor* diferite pe care parser-ul le ia în consideraţie şi respectiv le are în orice moment al analizei. În cele ce urmează, vom fi preocupaţi de îmbogăţirea structurii de date prezentate anterior (TSBF) în aşa fel încât ea să poată ţine evidenţa ipotezelor structurale şi a scopurilor propuse. Pentru aceasta vom opera două mici *modificări asupra unui TSBF*. Astfel, în loc de a se cere ca graful orientat aflat la baza TSBF-ului să fie aciclic, se va permite existenţa unor arce care revin în vârful de unde au plecat, acesta fiind singurul tip de ciclu admis. Această relaxare a cerinţei de aciclicitate, care introduce aşa-numitele "arce vide", va fi justificată în cele ce urmează şi va conduce la existenţa unor situaţii de tipul celei din Fig. 7.7:

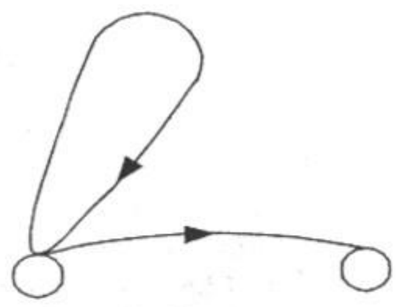


Fig. 7.7

O a doua modificare efectuată constă în schimbarea etichetei arcelor de la o simplă categorie la o regulă a gramaticii, în felul următor: dacă $S \rightarrow NP\ VP$ este o regulă a gramaticii PS date, atunci se vor utiliza ca etichete ale arcelor obiecte de tipul

$$S \rightarrow \bullet\ NP\ VP$$

$$S \rightarrow NP \bullet\ VP$$

$$S \rightarrow NP\ VP \bullet$$

adică aşa-numite "reguli punctate". Punctul care intervine în structura acestor etichete are rolul de a indica până unde a fost verificată de către parser ipoteza că regula respectivă se poate aplica. În particular, primul tip de etichetă va corespunde aşa-numitelor arce vide. Ea denotă ipoteza că un constituent S poate fi găsit ca acoperind un subşir care începe de la nodul luat în consideraţie şi că acest S acoperă subşirul respectiv în virtutea faptului că acesta din urmă poate fi acoperit, în egală măsură, de o secvenţă având structura NP VP. Eticheta indică faptul că o asemenea ipoteză a fost formulată, dar că ea nu a fost nici măcar parţial verificată. Cel de al doilea şi al treilea tip de etichetă, care nu vor apărea niciodată în cazul arcelor vide, denotă aceeaşi ipoteză, dar indică şi faptul că ea a fost parţial sau total confirmată. Cel de-al doilea tip de etichetă se va găsi numai în cazul arcelor care acopăr un NP. Cel de-al treilea tip de etichetă indică o ipoteză pe deplin confirmată şi corespunde numai arcelor care acopăr un şir alcătuit dintr-un subşir de tip NP urmat de un subşir de tip VP. Acest tip de etichetă este cel mai apropiat echivalent al etichetelor constând din categorii care au fost utilizate în cazul TSBF-urilor, dar el furnizează o informaţie suplimentară. Este indicată, în plus, secvenţa care justifică prezenţa respectivei categorii.

Un TSBF care a fost modificat pentru a include ipoteze în maniera descrisă se numeşte **hartă activă**, iar analizorii sintactici care exploatează o asemenea structură de date se numesc **analizori sintactici cu hartă**. În cele ce urmează, harta activă va fi denumită, simplu, *hartă*, iar arcele vor fi denumite *muchii*. Mai precis, vom gândi harta ca fiind o structură de date constând dintr-o mulţime de arce etichetate numite muchii. Arcele care reprezintă ipoteze neconfirmate vor fi numite **muchii active**, iar cele corespunzătoare ipotezelor confirmate vor fi numite **muchii inactive**. Cu alte cuvinte, o muchie inactivă reprezintă un rezultat, în timp ce o muchie activă reprezintă o ipoteză de structură. Hărţile pot reprezenta întreaga informaţie inclusă în TSBF-uri. Fig. 7.8 prezintă acelaşi exemplu pe care l-am utilizat în cazul TSBF-urilor, de data aceasta într-o notaţie cu hărţi.

Ca şi TSBF-urile, hărţile sunt complet neutre în raport cu strategia de parsing folosită.

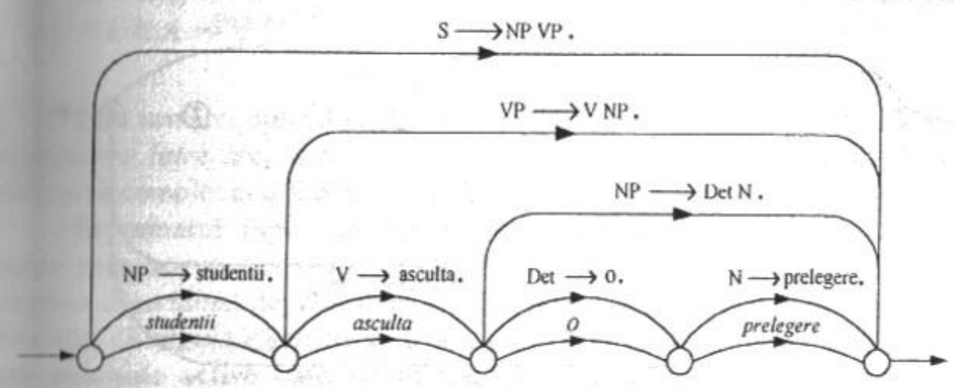


Fig. 7.8

Conform descrierii anterioare, o hartă poate fi reprezentată ca o mulțime de structuri, fiecare dintre acestea având atributele

< START > = ... un întreg ...

< FINAL > = ... un întreg ...

< ETICHETA > = ... o categorie ...

< GĂSIT > = ... o secvență de categorii ...

< DEGĂSIT > = ... o secvență de categorii ...

unde <ETICHETA> reprezintă membrul stâng al regulii punctate corespunzătoare, < GĂSIT > este secvența de categorii din membrul drept aflate la stânga punctului, iar <DEGĂSIT> este secvența de categorii din membrul drept situate la dreapta punctului. În această reprezentare, o muchie având valoarea corespunzătoare a lui <DEGĂSIT> secvența vidă va fi o muchie inactivă, în timp ce toate celelalte muchii vor fi active. Pentru a reprezenta astfel de înregistrări vom folosi adesea notații de tipul < i, j, R >, unde i și j sunt numere întregi, iar R desemnează o regulă punctată. Astfel, structura

< 0, 2, S → NP• VP >

reprezintă următoarea muchie activă:

< START > = 0

< FINAL > = 2

< ETICHETA > = S

< GĂSIT > = < NP >

< DEGĂSIT > = < VP >

7.3.5.2.1. Analiza sintactică cu hartă

În cele ce urmează, ne propunem să utilizăm structura de date descrisă - harta - pentru a face analiză sintactică. Vom avea în vedere faptul că harta reprezintă, în ultimă instanță, o mulțime de muchii și, ca și în prezentările anterioare, nu vom fi preocupați de problema construirii efective a arborilor PS de structură.

Pentru a vedea procesul care reprezintă esența analizei sintactice cu hartă vom considera un exemplu. Să presupunem că, într-un anumit moment al analizei, harta conține, printre altele, următoarele muchii:

{ < 0, 2, S → NP• VP >,

< 2, 3, VP → V• NP PP >,

< 3, 5, NP → Det N• >,

< 5, 8, PP → P NP• >,

... }

Aceste muchii pot fi reprezentate ca în Fig. 7.9, care arată două muchii active și două muchii inactive:

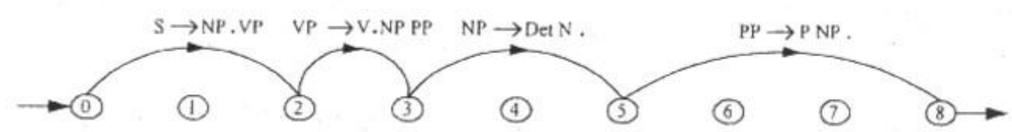


Fig. 7.9

(Pentru claritate am omis celelalte muchii care au fost obținute înainte de a se ajunge la situația din figură).

Prima muchie activă reprezintă o ipoteză despre o propoziție care a găsit un grup nominal (chiar dacă nu este cel reprezentat pe figură), iar acum caută un grup verbal. Pentru a satisface această ipoteză trebuie găsită pe hartă o muchie corespunzătoare *inactivă* care începe la poziția 2. Harta dată nu oferă decât o ipoteză privitoare la *eventuala* prezență a unei astfel de muchii (pentru moment activă). Până când această a doua ipoteză nu va fi confirmată, nu se poate efectua nici o acțiune privitoare la prima muchie activă. Din această cauză, parser-ul își va concentra atenția asupra celei de a doua muchii active din figură.

Cea de-a doua muchie activă reprezintă o ipoteză privitoare la un grup verbal care a găsit deja un verb, iar acum caută un grup nominal urmat de un grup prepozițional. Ipoteza aceasta caută un grup nominal care începe la poziția 3. O astfel de muchie (inactivă) există. Prin urmare, ipoteza referitoare la grupul verbal a fost confirmată în continuare, chiar dacă nu în întregime. Această confirmare parțială se reprezintă prin adăugarea pe hartă a noii muchii active

< 2, 5, VP → V NP. PP >. Aceasta reprezintă o nouă ipoteză, care caută un grup prepozițional începând de la poziția 5. O muchie inactivă corespunzătoare există. Prin urmare, noua ipoteză referitoare la grupul verbal este pe deplin confirmată. În mod corespunzător, se poate adăuga pe hartă muchia inactivă

< 2, 8, VP → V NP PP. >. S-a ajuns la următoarea structură a muchiilor:

{ < 0, 2, S → NP. VP >,

< 2, 3, VP → V. NP PP >,

< 2, 5, VP → V NP. PP >,

< 2, 8, VP → V NP PP. >,

< 3, 5, NP → Det N. >,

< 5, 8, PP → P NP. >,

... }

Întorcându-ne la prima muchie activă vom observa că ipoteza indusă de aceasta este acum pe deplin confirmată, întrucât a fost găsit un grup verbal care începe la poziția 2. Prin urmare, se poate adăuga muchia inactivă

< 0, 8, S → NP VP. >

și harta arată acum ca în Fig. 7.10:

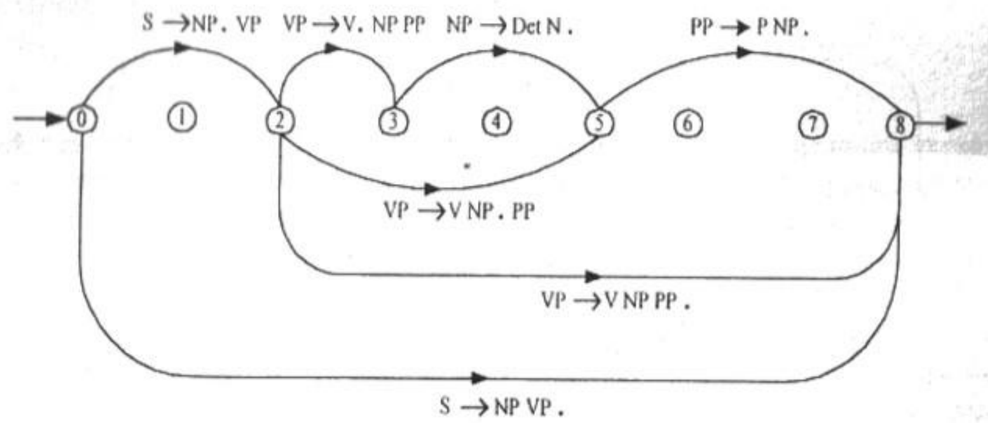


Fig. 7.10

Faptul că există o *muchie de tip S* care acoperă întreaga lățime a grafului înseamnă că s-a înregistrat succes în a analiza șirul de intrare ca pe o propoziție. (Alte analize mai pot exista, dar una dintre ele a fost deja găsită).

Menționăm faptul că, cea mai bună abordare, ca și în cazul TSBF-urilor, este aceea de a folosi harta numai dacă ea este *completă* relativ la un anumit tip de constituent intervenind la o anumită poziție. Spre exemplu, dacă se caută un constituent de tip NP la poziția 3 din șirul de intrare, se utilizează harta numai atunci când se cunoaște faptul că ea conține toate grupurile nominale posibile intervenind la poziția 3. Altfel, se poate ignora harta și efectua analiza sintactică în modul convențional, fără utilizarea acestei structuri de date. Dacă, în timpul acestui tip de analiză, nu se găsește ceea ce se căuta, atunci se afirmă faptul că harta este completă relativ la tipul de constituent căutat la poziția respectivă (și se include acest fapt în baza de date Prolog).

Regula fundamentală

Procesul descris anterior reprezintă esența analizei sintactice cu hartă. Ceea ce s-a făcut, de fapt, a fost aplicarea aceleiași reguli de trei ori: dacă o muchie activă întâlnește o muchie inactivă de categoria dorită, atunci se adaugă hărții o nouă muchie, acoperitoare pentru ambele. M. Kay, ale cărui lucrări le urmăm în această prezentare, numește aceasta "regula fundamentală a analizei sintactice cu hartă". **Regula fundamentală** poate fi formulată [9] după cum urmează:

Regulă fundamentală

Dacă harta conține muchiile < i, j, A → W1.B W2 > și < j, k, B → W3.>, unde A și B sunt categorii, iar W1, W2 și W3 reprezintă secvențe

de categorii sau cuvinte (posibil vide), atunci se adaugă hărții muchia $< i, k, A \rightarrow W1\ B.\ W2 >$.

Prin urmare, operațiile de bază ale unui analizor bazat pe hărți folosesc o *combinare între arce active și constituenți compleți*. Rezultatul este sau un constituent complet nou sau un arc activ nou, extensie a arcului activ originar.

Se remarcă faptul că regula fundamentală nu precizează dacă noua muchie este activă sau inactivă, dar acest fapt nu este necesar, el fiind în întregime determinat de W2, după cum acesta este sau nu vid. Se observă, de asemenea, că *regula efectuează numai adăugări pe hartă*. Ea nu înlătură de pe hartă muchiile active care au înregistrat succes. Acest lucru este necesar deoarece o astfel de muchie poate avea succes grație unei alte muchii inactive, care apare ulterior pe hartă. Îndepărtarea ei poate face, prin urmare, ca parser-ul să nu găsească toate analizele posibile ale unui șir de intrare.

Pentru a descrie în întregime un analizor sintactic cu hartă este necesar să tratăm, în plus, următoarele trei probleme: inițializarea, strategia de invocare a regulilor și strategia de căutare.

Inițializare

Inițializarea este necesară deoarece nu putem aplica regula fundamentală unei hărți care nu conține nici o muchie. (Este necesară existența cel puțin a unei muchii active și a unei muchii inactive pentru a se putea declanșa un algoritm de analiză sintactică).

A inițializa harta înseamnă a asigura existența unor muchii *inactive*. Acest lucru poate fi ușor realizat consultând lexiconul, care înregistrează cărei categorii lexicale îi aparține fiecare cuvânt. Arce multiple vor corespunde acelor cazuri de ambiguitate în care un același cuvânt aparține mai multor categorii lexicale.

Presupunând că, în cazul exemplului de până acum, este dat lexiconul

Cuvânt studenții:

< cat > = NP.

Cuvânt ascultă:

< cat > = V.

Cuvânt o:

< cat > = ART.

Cuvânt prelegere:

< cat > = N, V.

harta inițializată este cea din Fig. 7.11:

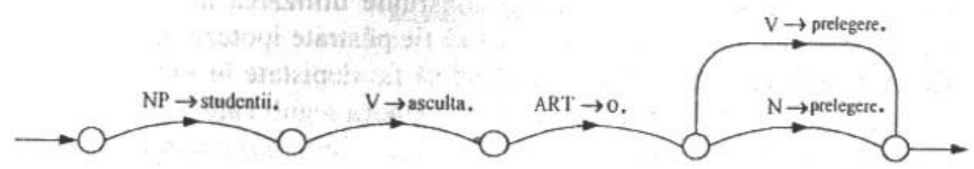


Fig. 7.11

Invocarea regulilor

Muchiile furnizate de pasul de inițializare nu sunt suficiente pentru ca analiza sintactică să poată începe. Ele reprezintă muchiile inactive cu care va debuta un algoritm de parsing cu hartă. Pentru aplicarea regulii fundamentale este însă necesară crearea, în egală măsură, a unor muchii active. Un principiu simplu care asigură crearea unor astfel de muchii este următorul: de fiecare dată când se adaugă hărții o muchie inactivă de categorie C, se adaugă și o nouă muchie activă vidă, începând din același vârf și corespunzând fiecărei reguli a gramaticii care necesită un constituent de categorie C în rolul fiului celui mai din stânga. (Prin muchie activă vidă înțelegem, așa cum era de așteptat, o muchie activă care urmează să își stabilească componentele și care, prin urmare, începe din și revine în același vârf).

Această strategie de invocare a regulilor va da naștere unei analize sintactice de tip bottom-up. Vom lua în considerație și alte strategii de invocare a regulilor în viitor. Pentru moment, ne vom limita la strategia amintită, care ar putea fi rezumată prin intermediul următoarei reguli:

Regulă bottom-up

Dacă se adaugă hărții muchia $< i, j, C \rightarrow W1.\ >$, *atunci, pentru fiecare regulă a gramaticii de forma* $B \rightarrow C\ W2$, *se adaugă hărții muchia* $< i, i, B \rightarrow .C\ W2 >$.

Cu ajutorul acestei reguli se pot adăuga hărții muchii active după pasul de inițializare. Este evident faptul că regula fundamentală și regula bottom-up sunt suficiente pentru a asigura găsirea tuturor analizelor posibile ale șirului de intrare. *Modul de interacțiune* al regulii bottom-up cu regula fundamentală depinde de ordinea în care diferitele muchii sunt adăugate hărții. Această chestiune depinde, la rândul ei, de *strategia de căutare*, problemă asupra căreia vom reveni.

Strategia de analiză sintactică cu hartă pe care tocmai am prezentat-o presupune că o regulă este regăsită de îndată ce este localizat elementul cel mai

din stânga al membrului ei drept. Această indexare după fiul cel mai din stânga sugerează o strategie de tip "din colţul stâng". O altă caracteristică interesantă a acestei tehnici de analiză sintactică o constituie utilizarea muchiilor active. Folosirea acestora în modul prezentat face să fie păstrate ipoteze despre grupuri sintactice care *ar putea fi prezente*, urmând să fie depistate în viitor. Muchiile active furnizează o modalitate eficientă de a memora reguli care s-ar putea să fie aplicate (bottom-up) într-o fază ulterioară. Atunci când vom lua în consideraţie un alt tip de regulă ("regula top-down"), vom vedea modalităţi mai eficiente decât aceasta de a utiliza muchiile active.

Pentru moment, vom reţine faptul că există două mari strategii de invocare a regulilor: cea bottom-up (la care ne-am referit deja) şi cea top-down. Indiferent de strategia aplicată, vom avea în vedere faptul că fiecare muchie activă reprezintă o ipoteză care trebuie explorată. Dacă această ipoteză înregistrează succes, fie şi numai parţial, atunci ea poate genera noi ipoteze (muchii active şi inactive), graţie acţiunii regulii fundamentale, precum şi a strategiei de invocare a regulilor care se adoptă.

Primii analizori sintactici pe care îi vom prezenta au în comun două caracteristici importante, şi anume:

- utilizează baza de date Prolog pentru memorarea hărţii;
- nu lucrează în mod explicit cu o structură de date de tip "agendă", ci se bazează, ca strategie de control, pe procesul de backtracking pe care Prologul îl efectuează.

Elemente de organizare

Implementarea unui analizor sintactic cu hartă va trebui să ţină cont de câteva elemente suplimentare, care au fost numite "elemente de organizare" [9].

Mai întâi, atunci când una dintre regulile discutate propune adăugarea unei noi muchii, aceasta va trebui inclusă în hartă numai dacă ea nu se află deja acolo. Prin urmare, adăugarea oricărei muchii trebuie să fie precedată de o verificare a existenţei acesteia. (Fără această verificare harta nu ar putea înlătura duplicatele din spaţiul de căutare).

Organizarea construirii arborelui PS corespunzător şirului analizat este o altă problemă care trebuie luată în consideraţie. Ca şi în cazul TSBF-urilor, ideea de bază este aceea de a avea ca elemente ale structurii de date pe care o formează muchiile *arbori* (reprezentaţi sub formă de liste), nu simple categorii. Această cerinţă va afecta secvenţa categoriilor găsite corespunzător unei muchii. Astfel, atributul GĂSIT al unei muchii va fi dat de secvenţa arborilor de derivare corespunzători grupurilor sintactice deja găsite, iar nu de simple categorii.

Regula fundamentală trebuie modificată pentru a lua în consideraţie construcţia arborilor în sensul amintit.

7.3.5.3. Implementarea unui parser bottom-up cu hartă

O primă implementare în Prolog a unui **parser bottom-up cu hartă** pe care o vom lua în consideraţie va utiliza însăşi baza de date Prolog ca depozitar al hărţii. Memorarea muchiilor se va face prin declararea lor în următorul format:

muchie(START, FINAL, ETICHETA, DEGĂSIT, GĂSIT).

Aşa cum se arată în [9], aceasta este cea mai simplă implementare posibilă, chiar dacă nu şi cea mai flexibilă.

Primul pas al algoritmului de analiză sintactică constă în *iniţializarea hărţii* prin introducerea muchiilor inactive corespunzător fiecărui cuvânt din şirul de intrare. Predicatul **init_harta**, care realizează iniţializarea, are ca al treilea argument al său un şir de cuvinte, iar ca prim argument un număr reprezentând poziţia de pe hartă de la care se porneşte. Rolul său este acela de a introduce pe hartă cuvintele adecvate şi de a instanţia cel de-al doilea argument la poziţia finală de pe hartă. Dacă şirul de cuvinte este vid, predicatul nu face nimic şi întoarce poziţia din şir. Altfel, el ia primul cuvânt din şir, adaugă pe hartă câte o muchie *inactivă* corespunzător fiecărei apariţii a cuvântului în lexicon şi apoi continuă în acelaşi mod cu celelalte cuvinte ale şirului. Predicatul **init_harta** se defineşte, prin urmare, astfel:

```
init_harta(V0, V0, []).
init_harta(V0, Vn, [Cuvant|Cuvinte]):-
        V1 is V0 + 1,
        pentru_fiecare(cuvant(Categorie, Cuvant),
              adauga_muchie(V0, V1, Categorie, [],
                            [Cuvant, Categorie])),
        init_harta(V1, Vn, Cuvinte).
```

Predicatul **pentru_fiecare** are rolul de a executa un proces de backtracking în căutarea tuturor modalităţilor de satisfacere a primului său argument ca scop Prolog. Corespunzător fiecăreia dintre acestea este invocat cel de-al doilea argument ca scop Prolog. În cazul de faţă, se apelează **adauga_muchie** pentru fiecare combinaţie de tipul cuvânt - categorie găsită de predicatul **cuvant**. Definiţia exactă a predicatului **pentru_fiecare** poate fi văzută în cadrul programului Prolog prezentat la sfârşitul acestui paragraf.

Se observă faptul că în argumentul GĂSIT al fiecărei muchii care trebuie adăugată se include însăşi categoria, întrucât scopul nostru este construirea arborelui de derivare (reprezentat ca o listă de categorii). Prin organizarea argumentului GĂSIT ca o listă incluzând categoria şi arborii de

derivare corespunzători grupurilor sintactice deja găsite se construiesc în mod automat arbori de derivare.

Predicatul **adauga_muchie** face însă mai mult decât să afirme existenţa muchiei, pe care o introduce în baza de date. El determină, în plus, ce alte muchii trebuie incluse pe hartă ca o consecinţă a acestei adăugări. Mai întâi, acest predicat verifică dacă muchia deja există, caz în care nu se întreprinde nimic:

```
adauga_muchie(V0, V1, Categorie, Categorii, Analiza):-
          muchie(V0, V1, Categorie, Categorii, Analiza),!.
```

Altfel, este introdusă muchia în baza de date, după care sunt adăugate eventuale alte muchii *active* a căror prezenţă pe hartă este determinată de includerea muchiei amintite.

În general, acest predicat se poate afla în una dintre următoarele două situaţii:

- adăugarea unei muchii inactive;
- adăugarea unei muchii active.

Dacă se adaugă hărţii, de la **V1** la **V2**, o muchie **inactivă** (argumentul DEGĂSIT este lista vidă) şi având eticheta **Categorie1**, atunci, corespunzător fiecărei reguli a gramaticii care are în membrul drept fiul cel mai din stânga de **Categorie1**, predicatul **adauga_muchie** va adăuga o *muchie activă vidă* la **V1**, de tipul dictat de către regula care se aplică. Aceasta este situaţia în care se utilizează regula bottom-up. Partea a doua a aceleiaşi clauze Prolog va trata cazul în care se aplică regula fundamentală, şi anume: pentru fiecare muchie *activă* care ajunge în **V1** şi care caută pe **Categorie1** se adaugă o nouă muchie, care ajunge în **V2** şi care poate fi activă sau inactivă. Prin urmare, clauza care se ocupă de *tot ceea ce trebuie făcut în cazul adăugării pe hartă a unei* ***muchii inactive*** este următoarea:

```
adauga_muchie(V1,V2,Categorie1,[],Analiza):-
    asserta(muchie(V1,V2,Categorie1,[],Analiza)),
    pentru_fiecare(regula(Categorie2,[Categorie1|
                          Categorii]),
        adauga_muchie(V1,V1,Categorie2,
        [Categorie1|Categorii],[Categorie2])),
    pentru_fiecare(muchie(V0,V1,Categorie2,
```



```
        [Categorie1|Categorii], Analize),
        adauga_muchie(V0, V2, Categorie2,
                Categorii,[Analiza|Analize])).
```

În *cazul adăugării pe hartă a unei* ***muchii active***, se caută fiecare muchie *inactivă* care ar putea permite extinderea muchiei active respective. În această situaţie, se aplică numai regula fundamentală, după cum urmează:

```
adauga_muchie(V0,V1,Categorie1,[Categorie2|Categorii],
            Analize):-
    asserta(muchie(V0,V1,Categorie1,
                [Categorie2|Categorii],Analize)),
    pentru_fiecare(muchie(V1,V2,Categorie2,[],Analiza),
            adauga_muchie(V0,V2,Categorie1,
                Categorii,[Analiza|Analize])).
```

Analizorul sintactic este complet, dar se va adăuga un ultim predicat, **parse**, utilizat în interogarea Prologului. Predicatul **parse(Cat, Sir)** îndeplineşte următoarele atribuţii:

- specifică vârful iniţial;
- afişează rezultatele analizei;
- înlătură toate muchiile create în timpul analizei.

Atunci când în rolul argumentului **Cat** se utilizează **s** (simbolul de start) se va face analiza sintactică a unui şir de intrare reprezentând o întreagă propoziţie. În general, argumentul **Cat** specifică categoria sintactică care trebuie analizată ori recunoscută (după cum se consideră că am implementat un analizor sintactic sau un dispozitiv de recunoaştere). Afişarea rezultatelor analizei se va face cu ajutorul predicatului **mwrite**. Predicatul **mwrite(Analiza)** este definit direct în program. Acest predicat are rolul de a afişa imaginea în oglindă (inversă) a unui arbore codificat sub formă de listă.

Predicatul **parse(Cat, Sir)** specifică, prin urmare, vârful iniţial, după care apelează predicatul **init_harta(V0, Vn, Sir)** care realizează efectiv analiza sintactică. Predicatul **parse(Cat, Sir)** trebuie să ţină cont de cele două situaţii posibile:

- nu există nici o muchie de tipul
  muchie(V0, Vn, Cat, [ ], _),

deci şirul de intrare nu este corect din punctul de vedere al categoriei specificate şi al gramaticii date;

- există măcar o muchie de tipul

  muchie(V0, Vn, Cat, [ ], _),

  adică măcar o analiză a şirului de intrare în conformitate cu gramatica dată şi cu categoria specificată.

În primul dintre cele două cazuri predicatul **parse** eşuează şi nu trebuie decât să fie retrase din baza de date muchiile create pe parcursul analizei. În cel de-al doilea caz se caută toate analizele posibile ale şirului de intrare, se afişează acestea şi se retrag din baza de date toate muchiile create:

```
parse(Cat,Sir):-
        V0 is 1,
        init_harta(V0,Vn,Sir),
        ((\+ muchie(V0,Vn,Cat,[],_),
        retractall(muchie(_,_,_,_,_)),!,fail);
       (pentru_fiecare(muchie(V0,Vn,Cat,[],Analiza),
                    mwrite(Analiza)),
       retractall(muchie(_,_,_,_,_)))).
```

Programul complet, scris în SICStus Prolog şi reprezentând implementarea parser-ului bottom-up cu hartă descris, care utilizează Gramatica nr. 2, este următorul:

Programul 7.6

```
:-dynamic muchie / 5 .

init_harta(V0,V0,[]).
init_harta(V0,Vn,[Cuvant|Cuvinte]):-
        V1 is V0+1,
        pentru_fiecare(cuvant(Categorie,Cuvant),
                    adauga_muchie(V0,V1,Categorie,[],
                                  [Cuvant,Categorie])),
        init_harta(V1,Vn,Cuvinte).
```



```
pentru_fiecare(X,Y):-X,executa(Y),fail.
pentru_fiecare(_,_):-true.

executa (Y):-Y,!.

adauga_muchie(V0,V1,Categorie,Categorii,Analiza):-
            muchie(V0,V1,Categorie,Categorii,Analiza),!.
adauga_muchie(V1,V2,Categorie1,[],Analiza):-
            asserta(muchie(V1,V2,Categorie1,[],Analiza)),
            pentru_fiecare(regula(Categorie2,
                                  [Categorie1|Categorii]),
                        adauga_muchie(V1,V1,Categorie2,
                        [Categorie1|Categorii],[Categorie2])),
            pentru_fiecare(muchie(V0,V1,Categorie2,
                        [Categorie1|Categorii],Analize),
                        adauga_muchie(V0,V2,Categorie2,
                        Categorii,[Analiza|Analize])).
adauga_muchie(V0,V1,Categorie1,[Categorie2|Categorii],
                        Analize):-
            asserta(muchie(V0,V1,Categorie1,
                        [Categorie2|Categorii],Analize)),
            pentru_fiecare(muchie(V1,V2,Categorie2,[],Analiza),
                        adauga_muchie(V0,V2,Categorie1,
                                  Categorii,[Analiza|Analize])).

parse(Cat,Sir):-
                V0 is 1,
                init_harta(V0,Vn,Sir),
                ((\+ muchie(V0,Vn,Cat,[],_),
                retractall(muchie(_,_,_,_,_)),!,fail);
                (pentru_fiecare(muchie(V0,Vn,Cat,[],Analiza),
                            mwrite(Analiza)),
                retractall(muchie(_,_,_,_,_)))).
```

```
mwrite(Arbore):-
                oglinda(Arbore,Imagine),
                write(Imagine),
                nl.

oglinda([],[]):-!.
oglinda(Atom,Atom):-atomic(Atom).
oglinda([X1|X2],Imagine):-
                        oglinda(X1,Y2),
                        oglinda(X2,Y1),
                        append(Y1,[Y2],Imagine).

append([], L, L).
append([H|T], L, [H|T1]) :- append(T, L, T1).

regula(s,[np,vp]).
regula(vp,[v,np]).
regula(np,[det,n]).

cuvant(det,un).
cuvant(det,niste).
cuvant(det,o).
cuvant(n,elev).
cuvant(n,elevi).
cuvant(n,carte).
cuvant(v,iubeste).
cuvant(v,iubesc).
```

Interogarea Prologului se face după cum urmează:

```
?- parse(s,[un,elev,iubeste,o,carte]).
```

sau

```
?- parse(vp,[iubeste,o,carte]).
```

În cazul celui de-al doilea exemplu de interogare, de pildă, răspunsul sistemului va fi:


```
[vp,[v,iubeste],[np,[det,o],[n,carte]]]
yes
```

Şirul de intrare a fost acceptat ca reprezentând un grup verbal, iar analiza sintactică a acestuia este afişată pe ecran.

7.3.5.4. Strategii alternative de invocare a regulilor

Analizorul sintactic implementat în §7.3.5.3 a fost unul de tip bottom-up deoarece invocă o regulă de acest gen. Aşa cum s-a văzut, structura de date numită *hartă* este neutră cu privire la strategia de căutare adoptată (depth-first, breadth-first etc.). Aceeaşi neutralitate se extinde şi asupra strategiei de invocare a regulilor. În cele ce urmează, vom prezenta şi implementa, conform [9], un parser care invocă o regulă top-down, adică o regulă care constă din următorii paşi:

Strategia top-down

(1) În momentul iniţializării, pentru fiecare regulă de forma $A \rightarrow W$, unde A reprezintă o categorie care poate "acoperi" o hartă (în mod tipic A desemnează categoria S), se adaugă hărţii muchia $< 0, 0, A \rightarrow .W >$.

(2) Dacă se adaugă hărţii muchia $< i, j, C \rightarrow W1.BW2 >$, atunci, pentru fiecare regulă de forma $B \rightarrow W$, se inserează pe hartă şi muchia $< j, j, B \rightarrow .W >$.

Aici, prima clauză asigură faptul că procesul de analiză sintactică începe prin adăugarea unei mulţimi de muchii active vide etichetate cu S şi adăugate vârfului iniţial, ca în exemplul din Fig.7.12 (care utilizează Gramatica nr. 2, căreia i s-au adăugat intrările lexicale `cuvant(v,elev)`, `cuvant(v,carte)` şi regula *NP → Det N PP*):

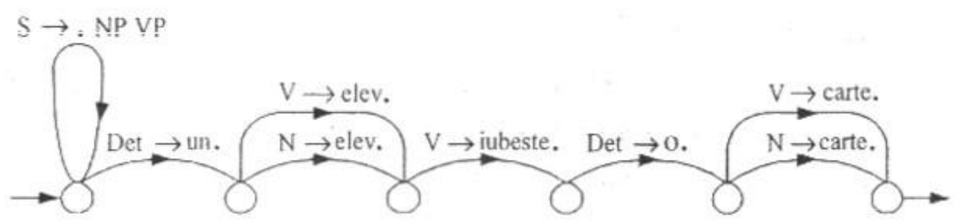


Fig. 7.12

Cea de a doua clauză a strategiei top-down asigură faptul că fiecare muchie activă produce noi muchii active vide, care urmăresc să găsească prima categorie din lista DEGĂSIT corespunzătoare muchiei active adăugate. În cazul exemplului nostru, după aplicarea acestei clauze, harta inițializată devine cea din Fig. 7.13:

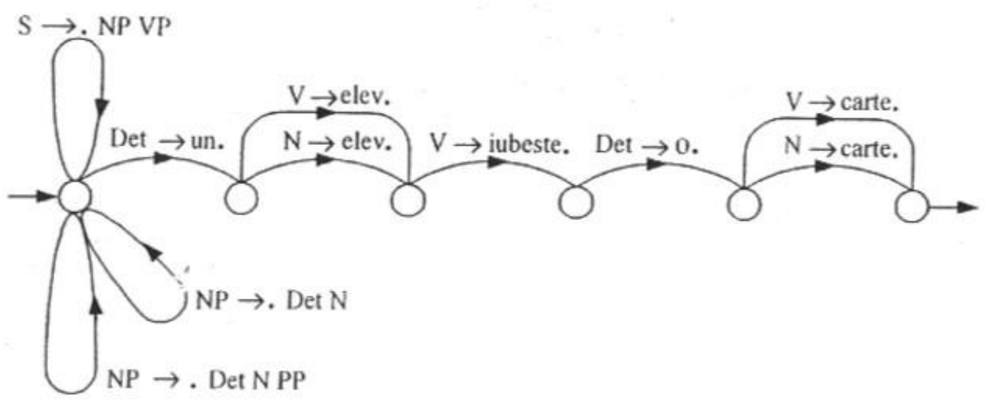


Fig. 7.13

În mod evident, este posibilă şi conceperea altor strategii de invocare a regulilor, eventual a unor strategii care să combine regulile bottom-up şi top-down. În orice astfel de situaţie trebuie avută în vedere problema *completitudinii* analizorului sintactic, adică a capacităţii acestuia de a găsi toate analizele posibile ale unui şir de intrare.

7.3.5.5. Implementarea unui parser top-down cu hartă

Programul de implementare a unui **parser top-down cu hartă** este similar cu cel prezentat în §7.3.5.3, singurele modificări datorându-se faptului că strategia de invocare a regulilor nu mai este una de tip bottom-up, ci este una top-down.

În cazul de faţă, predicatul **parse** realizează iniţializarea hărţii cu muchii lexicale inactive, stabileşte categoria simbolului considerat ca fiind iniţial sau de start (de obicei S) şi adaugă la începutul hărţii muchii active vide etichetate cu această categorie:

```
parse(V0,Vn,Sir):-
        init_harta(V0,Vn,Sir),
        initial(Simbol),
        init_activ(V0,Simbol).
```


Precizăm că, şi de această dată, fiecare adăugare a unei muchii pe hartă poate determina adăugarea unor alte muchii, proces care va fi urmărit în program.

Definiţia predicatului **init_harta** este aceeaşi din programul corespunzător parser-ului bottom-up. (Diferită va fi doar definiţia predicatului **adauga_muchie**, asupra căreia vom reveni).

În ceea ce priveşte definiţia predicatului **init_activ**, aceasta stipulează faptul că, pentru fiecare regulă care extinde **Categorie**, predicatul **init_activ(Varf,Categorie)** creează o muchie activă vidă, în **Varf**, bazată pe acea regulă:

```
init_activ(V0,Categorie):-
    pentru_fiecare(regula(Categorie,Categorii),
        adauga_muchie(V0,V0,Categorie,Categorii,
            [Categorie])).
```

Atât **init_harta** cât şi **init_activ** apelează predicatul **adauga_muchie**, a cărui definiţie va distinge între adăugarea pe hartă a unei muchii active sau a uneia inactive. Înainte de adăugarea oricărei muchii însă acest predicat verifică dacă muchia deja există, caz în care nu face nimic:

```
adauga_muchie(V1,V2,Categorie,Categorii,Analiza):-
    muchie(V1,V2,Categorie,Categorii,Analiza),!.
```

În *cazul adăugării pe hartă a unei* ***muchii inactive*** se are în vedere numai regula fundamentală:

```
adauga_muchie(V1,V2,Categorie1,[],Analiza):-
    asserta(muchie(V1,V2,Categorie1,[],Analiza)),
    pentru_fiecare(muchie(V0,V1,Categorie2,
            [Categorie1|Categorii],Analize),
        adauga_muchie(V0,V2,Categorie2,
            Categorii,[Analiza|Analize])).
```

În *cazul adăugării pe hartă a unei* ***muchii active*** se au în vedere atât regula fundamentală, cât şi regula top_down:

```
adauga_muchie(V1,V2,Categorie1,
```

```
        [Categorie2|Categorii],Analize):-
    asserta(muchie(V1,V2,Categorie1,
        [Categorie2|Categorii],Analize)),
    pentru_fiecare(muchie(V2,V3,Categorie2,[],
                Analiza),
                adauga_muchie(V1,V3,Categorie1,
                    Categorii,[Analiza|Analize])),
    init_activ(V2,Categorie2).
```

Predicatul **test(Sir)** are rolul de a

- specifica vârful inițial
- afișa rezultatele analizei
- înlătura toate muchiile create

și este definit după aceleași principii care au condus la definirea predicatului **parse** din § 7.3.5.3:

```
test(Sir):-
        V0 is 1,
        initial(Simbol),
        parse(V0,Vn,Sir),
        ((\+ muchie(V0,Vn,Simbol,[],_),
        retractall(muchie(_,_,_,_,_)),!,fail);
       (pentru_fiecare(muchie(V0,Vn,Simbol,[],Analiza),
                    mwrite(Analiza)),
        retractall(muchie(_,_,_,_,_)))).
```

În mod evident, programul va trebui să specifice care este simbolul inițial (sau simbolul de start al gramaticii). Execuția programului corespunzătoare unui simbol de start diferit va fi posibilă, dar va avea ca urmare schimbarea bazei de fapte Prolog. Programul complet, scris în SICStus Prolog și reprezentând implementarea parser-ului top-down cu hartă descris, care utilizează Gramatica nr. 2 în forma ei inițială, cu simbolul de start S, este următorul:

Programul 7.7

```
:-dynamic muchie / 5 .

parse(V0,Vn,Sir):-
                    init_harta(V0,Vn,Sir),
                    initial(Simbol),
                    init_activ(V0,Simbol).

init_harta(V0,V0,[]).
init_harta(V0,Vn,[Cuvant|Cuvinte]):-
        V1 is V0+1,
        pentru_fiecare(cuvant(Categorie,Cuvant),
                    adauga_muchie(V0,V1,Categorie,[],
                        [Cuvant,Categorie])),
        init_harta(V1,Vn,Cuvinte).

pentru_fiecare(X,Y):- X, executa(Y), fail.
pentru_fiecare(_,_):- true.

executa(Y):- Y,!.

init_activ(V0,Categorie):-
            pentru_fiecare(regula(Categorie,Categorii),
                    adauga_muchie(V0,V0,Categorie,
                        Categorii,[Categorie])).

adauga_muchie(V1,V2,Categorie,Categorii,Analiza):-
            muchie(V1,V2,Categorie,Categorii,Analiza),!.
adauga_muchie(V1,V2,Categorie1,[],Analiza):-
            asserta(muchie(V1,V2,Categorie1,[],Analiza)),
            pentru_fiecare(muchie(V0,V1,Categorie2,
                    [Categorie1|Categorii],Analize),
            adauga_muchie(V0,V2,Categorie2,Categorii,
```

```
                                [Analiza|Analize])).
adauga_muchie(V1,V2,Categorie1,[Categorie2|Categorii],
                    Analize):-
        asserta(muchie(V1,V2,Categorie1,[Categorie2|Categorii],
                    Analize)),
              pentru_fiecare(muchie(V2,V3,Categorie2,[],Analiza),
                              adauga_muchie(V1,V3,Categorie1,
                                            Categorii,
                                            [Analiza|Analize])),
              init_activ(V2,Categorie2).

test(Sir):- V0 is  1,
        initial(Simbol),
        parse(V0,Vn,Sir),
        ((\+ muchie(V0,Vn,Simbol,[],_),
        retractall(muchie(_,_,_,_,_)),!,fail);
        (pentru_fiecare(muchie(V0,Vn,Simbol,[],Analiza),
                        mwrite(Analiza)),
        retractall(muchie(_,_,_,_,_)))).

mwrite(Arbore):-
              oglinda(Arbore,Imagine),
              write(Imagine),
              nl.

oglinda([],[]):- !.
oglinda(Atom,Atom):-atomic(Atom).
oglinda([X1|X2],Imagine):-
                              oglinda(X1,Y2),
                              oglinda(X2,Y1),
                              append(Y1,[Y2],Imagine).

append([], L, L).
```



```
append([H|T], L, [H|T1]):- append(T, L, T1).

regula(s,[np,vp]).
regula(vp,[v,np]).
regula(np,[det,n]).

cuvant(det,un).
cuvant(det,niste).
cuvant(det,o).
cuvant(n,elev).
cuvant(n,elevi).
cuvant(n,carte).
cuvant(v,iubeste).
cuvant(v,iubesc).
initial(s).
```

Interogarea Prologului se face astfel:

```
?- test([un,elev,iubeste,o,carte]).
```

sau

```
?- test([iubeste,o carte]).
```

dacă se declară în program

```
initial(vp).
```

În acest din urmă caz răspunsul sistemului va fi:

```
[vp,[v,iubeste],[np,[det,o],[n,carte]]]
yes.
```

7.3.5.6. Strategia de căutare: utilizarea agendei

Aşa cum s-a mai arătat, fiecare muchie *activă* reprezintă o *ipoteză* care trebuie explorată. Dacă o astfel de ipoteză înregistrează succes, fie el şi parţial, atunci este foarte probabil ca ea să genereze noi ipoteze (muchii active şi inactive), datorită acţiunii regulii fundamentale, precum şi a strategiei de invocare a regulilor care a fost adoptată. În mod evident, pe parcursul derulării

analizei trebuie luate decizii cu privire la *ordinea* în care aceste ipoteze vor fi analizate. Modul în care sunt luate aceste decizii constituie *strategia de căutare*.

În efectuarea *analizei sintactice cu hartă* s-a constatat că este foarte utilă folosirea unei structuri de date suplimentare, sub forma unei **agende**, care să memoreze ipotezele de analizat sau muchiile care trebuie adăugate hărţii. În acest mod se evită investigarea unor ipoteze identice şi se iau decizii cu privire la ordinea şi momentul în care este analizată fiecare ipoteză.

O posibilitate existentă este aceea de a organiza agenda ca pe o **stivă**. Astfel, pe măsură ce sunt generate noi muchii, acestea se plasează în stivă. Atunci când trebuie selectată o muchie cu care să se lucreze este aleasă cea din vârful stivei. Această tehnică produce un comportament similar cu cel al căutării de tip depth-first. Fiecare posibilitate existentă este urmărită cât de mult posibil înainte de a se încerca o alternativă a ei. O altă strategie posibilă este aceea de a privi agenda ca pe o **coadă**. În acest caz, pe măsură ce se produc noi muchii, ele sunt plasate la sfârşitul cozii. Atunci când trebuie aleasă o muchie cu care să se lucreze este selectată cea din faţa cozii, ceea ce produce un comportament similar cu cel al căutării de tip *breadth_first*. Dacă muchiile diferite generate la fiecare pas reprezintă alternative, atunci putem spune că sistemul petrece numai o mică cantitate de timp analizând fiecare posibilitate înainte de a comuta la următoarea.

Multe alte strategii sunt posibile prin manevrarea agendei în moduri diferite. Spre exemplu, muchiile *active* pot fi poziţionate în agendă în funcţie de cât de aproape se află de stadiul reprezentării unor constituenţi compleţi. Într-o strategie de acest tip muchiile vide ar fi plasate la sfârşitul listei ce reprezintă agenda, în timp ce acelea care mai caută o unică categorie ar fi plasate la începutul ei, ori pe dos. Pot fi favorizate muchiile care rezultă din aplicarea anumitor reguli ale gramaticii faţă de cele care rezultă în urma aplicării altor reguli, exploatându-se, în acest fel, rezultatele şi informaţiile statistice existente referitoare la utilizarea limbajului natural. De asemenea, anumite muchii pot fi favorizate în conformitate cu informaţiile obţinute din alte componente ale sistemului, cum ar fi cea semantică. Dacă se doreşte ca analizorul sintactic implementat să producă *toate* analizele posibile, atunci harta va produce, până la urmă, exact acelaşi rezultat, indiferent de strategia de căutare impusă. Prin urmare, dacă *ordinea* în care analizele posibile ale şirului de intrare sunt generate nu prezintă interes, atunci alegerea strategiei de căutare rămâne, în esenţă, arbitrară. Dacă, în schimb, se caută o unică analiză, ori numai acele analize care respectă anumite condiţii (ce s-ar putea să nu derive exclusiv din regulile gramaticii), ori dacă se doreşte execuţia în paralel a unui alt proces (cum ar fi interpretarea semantică a constituenţilor), atunci strategia de căutare aleasă poate avea implicaţii serioase asupra eficienţei, fenomen asupra căruia vom reveni.

7.3.5.7. Implementarea controlului flexibil

Analizorii sintactici cu hartă descrişi până în prezent au în comun două caracteristici majore:

- utilizează baza de date Prolog pentru memorarea hărţii;
- în ceea ce priveşte *strategia de control*, nu lucrează în mod explicit cu o agendă, ci *se bazează pe procesul de backtracking* efectuat în mod automat de către Prolog.

Această abordare uşurează implementarea şi scurtează codul sursă al programului, dar sacrifică flexibilitatea. În cele ce urmează, vom menţine harta utilizată până în prezent şi vom lucra în mod explicit cu o structură de date suplimentară de tipul agendei descrise anterior. Acestea vor fi manevrate ca liste de muchii care sunt transmise ca argumente de la predicat la predicat. În acest mod devine posibilă comutarea între procesarea de tip depth-first şi cea de tip breadth-first, prin simpla interschimbare a ordinii argumentelor într-o unică apelare de predicat.

Structura de date care va sta la baza implementărilor analizorilor sintactici pe care îi vom prezenta în continuare este tot aceea de *muchie*, având următoarea reprezentare propusă în [9]:

muchie(START, FINAL, ETICHETA, DEGĂSIT).

Se observă eliminarea componentei finale GĂSIT, utilizată în reprezentarea de până acum a muchiilor. Aceasta înseamnă că, pentru a simplifica codul sursă, programele prezentate în cele ce urmează vor implementa mai degrabă un *dispozitiv de recunoaştere* şi nu un analizor sintactic propriu-zis. Astfel, în momentul interogării Prologului, programul va răspunde `yes` sau `no`, după cum şirul de intrare este recunoscut sau nu ca fiind corect din punctul de vedere al gramaticii date şi eventual al categoriei specificate, fără a afişa pe ecran analiza sintactică propriu-zisă a acestuia, ca în cazurile de până acum (§7.3.5.3 şi §7.3.5.5). *Hărţile şi agendele* vor fi privite ca nişte *liste* ale căror elemente sunt muchii având reprezentarea amintită.

7.3.5.7.1. Implementarea unui parser bottom-up cu hartă şi agendă

Pasul de iniţializare al algoritmului de analiză sintactică bottom-up va fi asigurat de către predicatul (de iniţializare) **init_agenda(Sir, Varf, Agenda)**, care are rolul de a crea o *agendă iniţială* ce conţine *muchii lexicale*, fiind dat şirul de intrare **Sir**, care începe de la nodul **Varf**. Dacă şirul de intrare este vid, atunci se creează o agendă iniţială vidă:

```
init_agenda([],_,[]).
```

Altfel, se izolează primul cuvânt, se consultă lexiconul şi se plasează într-o agendă toate muchiile inactive care "acopăr" cuvântul. Această agendă se adaugă (prin intermediul predicatului **append**, definit, în modul uzual, direct în program) în faţa agendei obţinute ca urmare a consultării lexiconului relativ la celelalte cuvinte:

```
init_agenda([Cuvant|Cuvinte],V0,Agenda):-
            V1 is V0+1,
            findall(muchie(V0,V1,Categorie,[]),
                    cuvant(Categorie,Cuvant),
                    Agenda1),
            init_agenda(Cuvinte,V1,Agenda2),
            append(Agenda1,Agenda2,Agenda).
```

Menţionăm că predicatul **findall**, predefinit în SICStus Prolog, are rolul de a forma, în cel de-al treilea argument al său, o listă care conţine, pentru fiecare soluţie a celui de-al doilea argument, instanţierea adecvată a primului argument. Atunci când cel de-al doilea argument al său nu are soluţii, predicatul **findall** produce, în cel de-al treilea argument, lista vidă. În cazul de faţă, **findall** realizează o iteraţie a tuturor modurilor posibile de a găsi **Cuvant** ca pe o instanţiere a lui **Categorie**. Pentru fiecare soluţie existentă, termenul **muchie** adecvat este inclus în lista rezultat (**Agenda 1**).

Predicatul **extinde_muchii(Agenda, InHarta, OutHarta)** este folosit pentru adăugarea muchiilor din **Agenda** unei hărţi iniţiale **InHarta**, adăugare ce trebuie să ia în consideraţie atât regula fundamentală, cât şi regula bottom-up, pentru a genera noi muchii. Aceste acţiuni au ca rezultat crearea unei noi hărţi numite **OutHarta**. Atunci când agenda este vidă, acest predicat reprezintă funcţia identitate:

```
extinde_muchii([],Harta,Harta).
```

Altfel, predicatul ia în consideraţie prima muchie din agendă şi vede dacă aceasta este deja plasată pe hartă. În caz afirmativ, el continuă parcurgerea agendei:

```
extinde_muchii([Muchie|Agenda1],Harta1,Harta2):-
                membru(Muchie,Harta1),!,
                extinde_muchii(Agenda1,Harta1,Harta2).
```


În caz contrar, **extinde_muchii** adaugă hărţii prima muchie, caută muchiile noi care sunt generate ca urmare a acestei adăugări, include aceste noi muchii în agendă şi apoi continuă procesul de analiză sintactică utilizând noua agendă şi harta mărită:

```
extinde_muchii([Muchie|Agenda1],Harta1,Harta3):-
                Harta2 = [Muchie|Harta1],
                muchii_noi(Muchie,Harta2,Muchii),
                adauga_muchii(Muchii,Agenda1,Agenda2),
                extinde_muchii(Agenda2,Harta2,Harta3).
```

O muchie se adaugă unei liste de muchii existente fie constatând că ea face deja parte din listă, fie printr-o includere directă în lista respectivă (pe post de cap al listei):

```
adauga_muchie(Muchie,Muchii,Muchii):-
                membru(Muchie,Muchii),!.
adauga_muchie(Muchie,Muchii,[Muchie|Muchii]).
```

Definiţia aproape evidentă a predicatului **membru** poate fi văzută direct în program.

Două liste de muchii sunt combinate prin apelarea predicatului **adauga_muchie** într-o manieră recursivă, de către predicatul **adauga_muchii**:

```
adauga_muchii([],Muchii,Muchii).
adauga_muchii([Muchie|Muchii],Muchii1,Muchii3):-
                adauga_muchie(Muchie,Muchii1,Muchii2),
                adauga_muchii(Muchii,Muchii2,Muchii3).
```

Revenind la predicatul **extinde_muchii**, care adaugă hărţii prima muchie din agendă în cazul în care agenda este nevidă iar muchia nu se regăseşte deja pa hartă, trebuie analizată posibila apariţie a unor noi muchii ca urmare a acestei adăugări. Cu alte cuvinte, trebuie luate în consideraţie acţiunile regulii fundamentale şi respectiv ale regulii bottom-up. Această analiză va fi realizată de către predicatul **muchii_noi(Muchie, Harta, Muchii)**, care face corespondenţa dintre **Muchie** şi **Harta** şi o mulţime de noi muchii (**Muchii**), apărută în urma adăugării lui **Muchie** la **Harta**. Acelaşi predicat are rolul de a adăuga hărţii muchiile active vide care declanşează procesul de analiză sintactică.

În definirea predicatului **muchii_noi** trebuie luate în considerație cele două cazuri posibile în inserarea pe hartă a unei noi muchii:

- adăugarea unei muchii inactive;
- adăugarea unei muchii active.

Dacă se adaugă, de la **V1** la **V2**, o muchie inactivă etichetată **Categorie1**, atunci, pentru fiecare regulă care are în membrul drept fiul cel mai din stânga de **Categorie1**, se creează în **V1** o muchie activă vidă de tipul impus de către regula care se utilizează. Aceasta este consecința aplicării regulii bottom-up. În continuare, conform regulii fundamentale, pentru fiecare muchie activă existentă care se termină în **V1** și caută **Categorie1**, se adaugă o muchie care ajunge în **V2** și care s-ar putea să fie activă ori inactivă. În final, cele două mulțimi de muchii care rezultă se combină. Acestea sunt acțiunile care derivă din *adăugarea pe hartă a unei **muchii inactive***. Ele sunt exprimate de următoarea clauză Prolog:

```
muchii_noi(muchie(V1,V2,Categorie1,[]),Harta,Muchii):-
      findall(muchie(V1,V1,Categorie2,
              [Categorie1|Categorii1]),
              regula(Categorie2,[Categorie1|Categorii1]),
              Muchii1),
      findall(muchie(V0,V2,Categorie3,Categorii2),
              membru(muchie(V0,V1,Categorie3,
                     [Categorie1|Categorii2]),Harta),
              Muchii2),
      adauga_muchii(Muchii1,Muchii2,Muchii).
```

În *situația adăugării pe hartă a unei **muchii active*** va acționa numai regula fundamentală. În acest caz, se caută fiecare muchie inactivă care ar putea permite extinderea muchiei active și se adaugă muchiile găsite operând extinderile de rigoare:

```
muchii_noi(muchie(V1,V2,Categorie1,[Categorie2|Categorii]),
        Harta,Muchii):-
         findall(muchie(V1,V3,Categorie1,Categorii),
             membru(muchie(V2,V3,Categorie2,[]),Harta),
             Muchii).
```


Întrucât toate cazurile posibile au fost luate în considerație, vom introduce un ultim predicat, **parse**, utilizat în interogarea Prologului și care reunește predicatele **init_agenda** și **extinde_muchii**. Predicatul **parse(Cat, Sir)** îndeplinește atribuțiile de a

- specifica vârful inițial;
- verifica dacă harta conține într-adevăr o muchie care acoperă *întreg* șirul de intrare relativ la categoria specificată:

```
parse(Cat,Sir):-
            init_agenda(Sir,0,Agenda),
            extinde_muchii(Agenda,[],Harta),
            membru(muchie(0,M,Cat,[]),Harta),
            N is M+1,
            \+ (membru(muchie(_,N,_,_),Harta)).
```

Programul complet, scris în SICStus Prolog și reprezentând implementarea parser-ului bottom-up descris, care utilizează Gramatica nr. 2 (în forma ei inițială), este următorul:

Programul 7.8

```
:- dynamic muchie / 4.

init_agenda([],_,[]).
init_agenda([Cuvant|Cuvinte],V0,Agenda):-
                V1 is V0+1,
                findall(muchie(V0,V1,Categorie,[]),
                        cuvant(Categorie,Cuvant),Agenda1),
                init_agenda(Cuvinte,V1,Agenda2),
                append(Agenda1,Agenda2,Agenda).

append([],L,L).
append([H|T],L,[H|T1]):-append(T,L,T1).
```

```
extinde_muchii([],Harta,Harta).
extinde_muchii([Muchie|Agenda1],Harta1,Harta2):-
                membru(Muchie,Harta1),!,
                extinde_muchii(Agenda1,Harta1,Harta2).
extinde_muchii([Muchie|Agenda1],Harta1,Harta3):-
                Harta2 = [Muchie|Harta1],
                muchii_noi(Muchie,Harta2,Muchii),
                adauga_muchii(Muchii,Agenda1,Agenda2),
                extinde_muchii(Agenda2,Harta2,Harta3).

membru(X,[X|_]).
membru(X,[_|Y]):- membru(X,Y).

adauga_muchie(Muchie,Muchii,Muchii):-
            membru(Muchie,Muchii),!.
adauga_muchie(Muchie,Muchii,[Muchie|Muchii]).

adauga_muchii([],Muchii,Muchii).
adauga_muchii([Muchie|Muchii],Muchii1,Muchii3):-
              adauga_muchie(Muchie,Muchii1,Muchii2),
              adauga_muchii(Muchii,Muchii2,Muchii3).

muchii_noi(muchie(V1,V2,Categorie1,[]),Harta,Muchii):-
          findall(muchie(V1,V1,Categorie2,
                         [Categorie1|Categorii1]),
                  regula(Categorie2,[Categorie1|Categorii1]),
                  Muchii1),
          findall(muchie(V0,V2,Categorie3,Categorii2),
                  membru(muchie(V0,V1,Categorie3,
                                [Categorie1|Categorii2]),Harta),
                  Muchii2),
          adauga_muchii(Muchii1,Muchii2,Muchii).
muchii_noi(muchie(V1,V2,Categorie1,[Categorie2|Categorii]),
```



```
                Harta,Muchii):-
                findall(muchie(V1,V3,Categorie1,Categorii),
                        membru(muchie(V2,V3,Categorie2,[]),Harta),
                        Muchii).

parse(Cat,Sir):-
                        init_agenda(Sir,0,Agenda),
                        extinde_muchii(Agenda,[],Harta),
                        membru(muchie(0,M,Cat,[]),Harta),
                        N is M + 1,
                        \+ (membru(muchie(_,N,_,_),Harta)).

regula(s,[np,vp]).
regula(vp,[v,np]).
regula(np,[det,n]).

cuvant(det,un).
cuvant(det,niste).
cuvant(det,o).
cuvant(n,elev).
cuvant(n,elevi).
cuvant(n,carte).
cuvant(v,iubeste).
cuvant(v,iubesc).
```

Interogarea Prologului se face astfel:

```
?- parse(s,[un,elev,iubeste,o,carte]).
yes
```

sau

```
?- parse(vp,[iubeste,o,carte]).
yes
```

Se observă că analizorul sintactic considerat operează în manieră **depth-first** (datorită ordinii în care muchiile sunt adăugate în agendă). Astfel, în prezent, agenda este organizată ca o *stivă*, noile muchii care rezultă în urma adăugării unei muchii fiind inserate în fața agendei, adică în vârful stivei.

Agenda poate fi organizată, în egală măsură, ca o *coadă*, caz în care analizorul sintactic va explora spațiul de căutare în manieră **breadth-first**.

Diferența în organizarea agendei poate fi foarte ușor pusă în evidență în cadrul definiției predicatului **extinde_muchii**. Ordinea primelor două argumente (reprezentând agenda existentă și noile muchii) ale predicatului **adauga_muchii**, apelat de către **extinde_muchii** pentru a construi noua agendă, este cea care determină organizarea acesteia:

```
extinde_muchii([Muchie|Agenda1],Harta1,Harta3):-
        Harta2 = [Muchie|Harta1],
        muchii_noi(Muchie,Harta2,Muchii),
        %adauga_muchii(Muchii,Agenda1,Agenda2),
        %procesare depth-first
        adauga_muchii(Agenda1,Muchii,Agenda2),
        %procesare breadth-first
        extinde_muchii(Agenda2,Harta2,Harta3).
```

7.3.5.7.2. Implementarea unui parser top-down cu hartă și agendă

Programul care implementează un parser top-down în acest context va utiliza o serie dintre predicatele definite anterior (§7.3.5.7.1). Este vorba despre predicatele **init_agenda**, **adauga_muchie**, **adauga_muchii** și **extinde_muchii**, care toate rămân neschimbate.

Predicatul **init_agenda** creează o agendă inițializată cu muchii lexicale. Întrucât, în acest caz, nu acționează regula bottom-up, această agendă poate fi considerată ca reprezentând harta inițială.

Este necesară, de asemenea, introducerea pe harta inițială a unor muchii active, lucru care va fi realizat, ca și în trecut, de predicatul **init_activ**. Definiția acestuia este una directă: pentru fiecare regulă care extinde **Categorie**, predicatul **init_activ(Categorie, Varf, Muchii)** creează, la **Varf**, o muchie activă vidă bazată pe regula respectivă. Muchia este introdusă în lista **Muchii**, după cum urmează:

```
init_activ(Categorie,Varf,Muchii):-
```



```
        findall(muchie(Varf,Varf,Categorie,Categorii),
                regula(Categorie,Categorii),Muchii).
```

Regula fundamentală a analizei sintactice cu hartă, precum și regula top-down vor fi încorporate în predicatul **muchii_noi(Muchie, Harta, Muchii)**, care face corespondența dintre **Muchie** și **Harta** și o mulțime de muchii noi, numită **Muchii**, care ia naștere prin adăugarea lui **Muchie** la **Harta**. În definirea predicatului **muchii-noi** trebuie din nou luate în considerație cele două cazuri posibile în inserarea pe hartă a unei noi muchii:

- adăugarea unei muchii inactive;
- adăugarea unei muchii active.

Dacă se adaugă, de la **V1** la **V2**, o ***muchie inactivă*** etichetată **Categorie1**, atunci, pentru fiecare muchie activă existentă, care se termină în **V1** și care caută **Categorie1**, se adaugă o nouă muchie, care se termină în **V2** și care poate fi atât activă, cât și inactivă:

```
muchii_noi(muchie(V1,V2,Categorie1,[]),Harta,Muchii):-
        findall(muchie(V0,V2,Categorie2,Categorii),
                membru(muchie(V0,V1,Categorie2,
                              [Categorie1|Categorii]),Harta),
                Muchii).
```

Această clauză tratează *cazul adăugării unei **muchii inactive***.

În situația în care *se adaugă hărții o **muchie activă***, trebuie avute în vedere atât acțiunea regulii fundamentale, cât și cea a regulii top-down. Prin urmare, se vor executa următoarele trei operații:

- se creează muchiile active vide relevante care caută următoarea categorie (strategie top-down);
- se caută fiecare muchie inactivă care permite extinderea muchiei active adăugate și se creează, în mod efectiv, noi muchii, prin extinderea acesteia;
- se combină cele două mulțimi de muchii rezultate.

Clauza Prolog corespunzătoare, care încorporează aceste trei acțiuni, este următoarea:

```
muchii_noi(muchie(V1,V2,Categorie1,[Categorie2|Categorii]),
        Harta,Muchii):-
    init_activ(Categorie2,V2,Muchii1),
    findall(muchie(V1,V3,Categorie1,Categorii),
            membru(muchie(V2,V3,Categorie2,[]),Harta),
            Muchii2),
    append(Muchii1,Muchii2,Muchii).
```

Interogarea Prologului se va realiza prin intermediul predicatului **parse(Cat, Sir)**, care

- inițializează harta;
- creează, la începutul hărții, muchia (muchiile) activă vidă corespunzătoare categoriei analizate (recunoscute);
- apelează predicatul **extinde_muchii**, care realizează efectiv analiza sintactică a șirului de intrare **Sir** relativ la categoria specificată **Cat**;
- elaborează harta până la final efectuând o verificare a sfârșitului șirului de intrare (i.e. vede dacă harta conține într-adevăr o muchie care acoperă *întreg* șirul de intrare relativ la categoria specificată;
- verifică dacă s-a înregistrat succes în analiza (recunoașterea) șirului de intrare:

```
parse(Cat,Sir):-
        init_agenda(Sir,0,Harta1),
        init_activ(Cat,0,Agenda),
        extinde_muchii(Agenda,Harta1,Harta2),
        membru(muchie(0,M,Cat,[]),Harta2),
        N is M+1,
        \+ (membru(muchie(_,N,_,_),Harta2)).
```

Programul complet, scris în SICStus Prolog și reprezentând implementarea parser-ului top-down descris, care utilizează Gramatica nr. 2 (în forma ei inițială), este următorul:

Programul 7.9

```
:- dynamic muchie / 4.

init_agenda([],_,[]).
init_agenda([Cuvant|Cuvinte],V0,Agenda):-
            V1 is V0+1,
            findall(muchie(V0,V1,Categorie,[]),
                    cuvant(Categorie,Cuvant),
                    Agenda1),
            init_agenda(Cuvinte,V1,Agenda2),
            append(Agenda1,Agenda2,Agenda).

append([],L,L).
append([H|T],L,[H|T1]):-append(T,L,T1).

extinde_muchii([],Harta,Harta).
extinde_muchii([Muchie|Agenda1],Harta1,Harta2):-
            membru(Muchie,Harta1),!,
            extinde_muchii(Agenda1,Harta1,Harta2).
extinde_muchii([Muchie|Agenda1],Harta1,Harta3):-
            Harta2 = [Muchie|Harta1],
            muchii_noi(Muchie,Harta2,Muchii),
            adauga_muchii(Muchii,Agenda1,Agenda2),
            extinde_muchii(Agenda2,Harta2,Harta3).

membru(X,[X|_]).
membru(X,[_|Y]):- membru(X,Y).

adauga_muchie(Muchie,Muchii,Muchii):-
            membru(Muchie,Muchii),! .
adauga_muchie(Muchie,Muchii,[Muchie|Muchii]).

adauga_muchii([],Muchii,Muchii)
```

```
adauga_muchii([Muchie|Muchii],Muchii1,Muchii3):-
                adauga_muchie(Muchie,Muchii1,Muchii2),
                adauga_muchii(Muchii,Muchii2,Muchii3).

init_activ(Categorie,Varf,Muchii):-
        findall(muchie(Varf,Varf,Categorie,Categorii),
                regula(Categorie,Categorii),Muchii).

muchii_noi(muchie(V1,V2,Categorie1,[]),Harta,Muchii):-
        findall(muchie(V0,V2,Categorie2,Categorii),
                membru(muchie(V0,V1,Categorie2,
                        [Categorie1|Categorii]),Harta),
                Muchii).
muchii_noi(muchie(V1,V2,Categorie1,[Categorie2|Categorii]),
          Harta,Muchii):-
        init_activ(Categorie2,V2,Muchii1),
        findall(muchie(V1,V3,Categorie1,Categorii),
                membru(muchie(V2,V3,Categorie2,[]),Harta),Muchii2),
        append(Muchii1,Muchii2,Muchii).

parse(Cat,Sir):-
                init_agenda(Sir,0,Harta1),
                init_activ(Cat,0,Agenda),
                extinde_muchii(Agenda,Harta1,Harta2),
                membru(muchie(0,M,Cat,[]),Harta2),
                N is M+1,
                \+ (membru(muchie(_,N,_,_),Harta2)).

regula(s,[np,vp]).
regula(vp,[v,np]).
regula(np,[det,n]).

cuvant(det,un).
cuvant(det,niste).
```



```
cuvant(det,o).
cuvant(n,elev).
cuvant(n,elevi).
cuvant(n,carte).
cuvant(v,iubeste).
cuvant((v,iubesc).
```

Interogarea Prologului se face astfel:

```
? - parse(s,[un,elev,iubeste,o,carte]).
yes
```

sau

```
?- parse(vp,[iubeste,o,carte]).
yes
```

7.3.5.7.3. Variante ale analizorilor sintactici bottom-up şi top-down cu hartă şi agendă

Orice variantă a analizorilor sintactici cu hartă prezentaţi deja va lua în consideraţie faptul că operaţia de bază efectuată de un asemenea parser constă în combinarea unei muchii active cu o muchie inactivă, aceasta din urmă reprezentând un constituent gata completat, adică un aşa-numit **constituent complet**. Rezultatul acestei operaţii de bază este fie apariţia unui nou constituent complet, fie a unei noi muchii active, care reprezintă o extindere a muchiei active iniţiale.

Variantele prezentate aici şi preluate din [1] ale analizorilor sintactici implementaţi până în acest moment organizează în mod diferit lista reprezentând *agenda*, în sensul că aceasta *conţine numai constituenţi compleţi*. Astfel, noii constituenţi compleţi, rezultaţi în urma aplicării regulii fundamentale, sunt incluşi în agendă, până în momentul în care vor fi adăugaţi hărţii. Acest proces este descris în mod exact de către următorul *Algoritm de extindere a muchiilor*:

Algoritmul 7.5

Pentru a adăuga un constituent C de la poziţia p_1 la poziţia p_2, execută:

1. Inserează C în hartă de la p_1 la p_2.

2. Pentru orice muchie activă de forma $X \rightarrow X_1 \ldots .C \ldots X_n$ de la p_0 la p_1, adaugă o muchie activă nouă $X \rightarrow X_1 \ldots C. \ldots X_n$ de la p_0 la p_2.

3. Pentru orice muchie activă de forma $X \rightarrow X_1 \ldots X_n .C$ de la p_0 la p_1, adaugă în agendă un nou constituent de tipul X, de la p_0 la p_2.

❑

Fiind dat Algoritmul 7.5 de extindere a muchiilor, **algoritmul de analiză sintactică bottom-up cu hartă şi agendă** este următorul:

Algoritmul 7.6

Execută următorii paşi până când se parcurge întreg şirul de intrare:

1. Dacă agenda este vidă, se caută şi se introduc în agendă toate interpretările pentru următorul cuvânt din şirul de intrare.
2. Se alege un constituent din agendă. Fie acesta constituentul C de la poziţia p_1 la poziţia p_2.
3. Pentru fiecare regulă a gramaticii de forma $X \rightarrow CX_1 \ldots X_n$, se adaugă o muchie activă de forma $X \rightarrow C.X_1 \ldots X_n$ de la poziţia p_1 la poziţia p_2.
4. Se adaugă hărţii constituentul C utilizând Algoritmul 7.5 de extindere a muchiilor.

❑

Ca şi până acum, se poate folosi o strategie de căutare de tip depth-first sau una de tip breadth-first, după cum agenda este implementată sub formă de stivă ori de coadă. Remarcăm faptul că, pentru aplicarea unei strategii veritabile de tip breadth-first, este necesară citirea întregului şir de intrare cu adăugarea în agenda iniţială a tuturor interpretărilor cuvintelor acestuia înainte de declanşarea algoritmului propriu-zis de analiză sintactică.

În cele ce urmează, prezentăm un **algoritm de analiză sintactică top-down cu hartă** care se bazează, de asemenea, pe o *agendă alcătuită numai din constituenţi compleţi* şi pe Algoritmul 7.5 de extindere a muchiilor. Diferenţa fundamentală faţă de algoritmul top-down cu hartă prezentat anterior (§7.3.5.7.2) constă tot în faptul că agenda este formată numai din constituenţi compleţi. De asemenea, în algoritmul care urmează, iniţializarea agendei nu se va face prin introducerea prealabilă a tuturor muchiilor lexicale posibile, cuvintele şirului de intrare fiind pe rând interpretate cu ajutorul lexiconului,

atunci când se constată că agenda este vidă. În abordarea top-down care urmează şi a cărei implementare este lăsată în seama cititorului, sunt generate muchii active noi ori de câte ori o nouă muchie activă este adăugată hărţii, conform **Algoritmului de introducere top-down a muchiilor** [care reproduce pasul general (2) al strategiei top-down enunţate în §7.3.5.4]:

Algoritmul 7.7

Pentru a adăuga o muchie $S \rightarrow C_1 \ldots .C_i \ldots C_n$ care se termină la poziţia j, execută:
Pentru fiecare regulă a gramaticii de forma $C_i \rightarrow X_1 \ldots X_k$ adaugă în mod recursiv noua muchie $C_i \rightarrow .X_1 \ldots X_k$ de la poziţia j la poziţia j.

❑

Algoritmul de analiză sintactică top-down cu hartă este următorul:

Algoritmul 7.8

Iniţializare: Pentru fiecare regulă a gramaticii de forma $S \rightarrow X_1 \ldots X_k$, se adaugă o muchie etichetată $S \rightarrow .X_1 \ldots X_k$ utilizând *Algoritmul de introducere top-down a muchiilor*.

Analiză: Execută până când se parcurge întreg şirul de intrare:

1. Dacă agenda este vidă, caută interpretările posibile ale cuvântului următor al şirului de intrare şi adaugă-le agendei.
2. Selectează un constituent din agendă. Fie acesta constituentul C.
3. Combină C cu fiecare muchie activă de pe hartă folosind Algoritmul de extindere a muchiilor. Adaugă agendei eventualii constituenţi noi care rezultă.
4. Adaugă hărţii orice muchie activă creată la pasul 3 utilizând Algoritmul de introducere top-down a muchiilor.

❑

Algoritmii 7.6 şi 7.8 nu reprezintă un salt calitativ din punctul de vedere al eficienţei, dar uşurează implementarea analizorilor sintactici top-down şi bottom-up în alte limbaje decât Prologul, şi anume în *limbajele care admit accesul direct*. Caracteristica acestor *variante* ale algoritmilor top-down şi

bottom-up cu hartă şi agendă prezentaţi anterior este aceea că în agendă se depun numai constituenţi compleţi. Prin urmare, agenda este menţinută ca având o structură cât mai simplă, restul fiind plasat pe hartă, care se prelucrează în *acces direct*. Tocmai de aceea se recomandă implementarea acestor algoritmi în alte limbaje decât Prologul.

Majoritatea algoritmilor de analiză sintactică au fost dezvoltaţi la mijlocul anilor '60, de către informaticieni, cu scopul principal de a analiza limbajele de programare şi nu limbajul natural. Conceptul de hartă este descris în special de către Kay (1973; 1980) şi a fost preluat şi adaptat de către numeroase sisteme ulterioare bazate pe analizori sintactici. Algoritmul 7.6, prezentat aici, este similar Algoritmului de analiză sintactică din colţul stâng al lui Aho şi Ullman (1972), în timp ce Algoritmul 7.8 este similar celui descris de către Earley (1970) şi, prin urmare, este numit Algoritmul lui Earley.

Analiza sintactică de tip Earley este definită pentru gramatici independente de context, algoritmul corespunzător lucrând în manieră top-down şi de la stânga la dreapta. Algoritmul lui Earley (1970) este printre primii algoritmi bazaţi pe gramatici PS independente de context care explorează în mod sistematic îmbunătăţirile legate de eficienţă realizate prin memorarea rezultatelor intermediare. Printre caracteristicile algoritmului lui Earley se numără următoarele proprietăţi:

- analizează propoziţii alcătuite din n cuvinte într-un interval de timp proporţional cu cel mult n^3, ceea ce se apropie de cea mai bună performanţă posibilă;
- manevrează constituenţii nuli în mod corect;
- nu ciclează atunci când întâlneşte reguli recursive la stânga ($A \rightarrow AB$)[34].

Ideea centrală a acestui algoritm este aceea că harta poate memora atât constituenţi compleţi, cât şi constituenţi incompleţi, la sfârşitul procesului de analiză sintactică toate analizele alternative găsindu-se pe hartă. Pentru o descriere şi o implementare în Prolog a algoritmului lui Earley, a se vedea şi [5].

7.3.5.8. Eficienţă

Aşa cum se arată în [9], eficienţa unui analizor sintactic cu hartă dat, relativ la un şir dat de intrare, se află într-o relaţie invers proporţională cu

[34] S-a arătat că algoritmul lui Earley ciclează, totuşi, în cazul utilizării unei reguli a gramaticii în care unul dintre nodurile fii are un argument care conţine un argument al nodului părinte, adică în cazul unei reguli de tipul: a(X) --> a(f(X)).

cantitatea de "structură în exces" care va fi prezentă pe hartă la sfârşitul procesului de analiză sintactică.

Dacă, spre exemplu, şirul de intrare nu este corect relativ la gramatica specificată, dar harta conţine un număr mare de muchii la sfârşitul analizei, aceasta înseamnă că analizorul sintactic a efectuat un număr semnificativ de operaţii inutile. Într-un astfel de caz ar fi de dorit ca parser-ul să construiască foarte puţine muchii înainte de a da răspunsul conform căruia şirul de intrare nu poate fi analizat din punctul de vedere al gramaticii date.

În situaţia în care şirul de intrare este corect iar harta conţine un număr mare de muchii, atunci parser-ul este unul eficient dacă fiecare muchie inactivă prezentă este folosită în una sau mai multe dintre structurile atribuite şirului şi dacă fiecare dintre muchiile active prezente reprezintă o ipoteză care, până la urmă, va înregistra succes. Muchiile inactive neutilizate, precum şi muchiile active care nu conduc la nimic reflectă, în mod direct, lipsa de eficienţă a parser-ului. Experimentarea cu strategiile alternative de invocare a regulilor are ca scop tocmai reducerea la maximum a acestei ineficienţe.

Eficienţa unui analizor sintactic cu hartă este, de asemenea, mult influenţată de modul în care sunt armonizate structurile de date utilizate pentru a susţine operaţiile de bază care se repetă în decursul procesului de analiză sintactică. Principalele operaţii de bază, în decursul cărora parser-ul regăseşte informaţie utilizând atât harta, cât şi gramatica, sunt următoarele:

- atunci când *se adaugă hărţii o muchie inactivă*, parser-ul caută (pe hartă) muchiile active necesare aplicării regulii fundamentale;
- atunci când *se adaugă hărţii o muchie activă*, parser-ul caută (pe hartă) muchiile inactive necesare aplicării regulii fundamentale;
- atunci când *se adaugă hărţii o muchie inactivă*, parser-ul caută toate regulile gramaticii în care prima categorie a membrului drept coincide cu categoria muchiei, cu scopul de a aplica regula bottom-up;
- atunci când *se adaugă hărţii o muchie activă*, parser-ul caută toate regulile gramaticii al căror membru stâng reprezintă prima categorie cerută de muchie, cu scopul de a aplica regula top-down.

Este evident faptul că performanţa ar scădea în mod dramatic odată cu mărirea hărţii dacă parser-ul ar fi nevoit să parcurgă toate muchiile acesteia ori de câte ori caută o muchie de un anumit tip. Acesta este tipul de performanţă scăzută care se obţine atunci când harta este reprezentată sub forma unei simple liste de structuri de date de tip *muchie*, ca în cazul analizorilor sintactici prezentaţi anterior (§7.3.5.3, §7.3.5.5, §7.3.5.7.1, §7.3.5.7.2). O implementare mai eficientă ar putea grupa, spre exemplu, muchiile hărţii într-o structură de

date indexată după poziţia de început şi de sfârşit a fiecărei muchii, ori în conformitate cu anumite categorii relevante ale muchiei (cum ar fi eticheta unei muchii inactive sau prima categorie de tip <DEGĂSIT> a unei muchii active). În acest mod căutarea ar putea progresa rapid spre o mulţime restrânsă de potenţiale muchii relevante, fără a se lua în consideraţie întreaga hartă. Aceleaşi observaţii sunt valabile relativ la organizarea agendei.

Analizorii sintactici prezentaţi în acest capitol nu au fost proiectaţi în mod special după criteriul vitezei. (Numai DCG şi BUP compilează regulile gramaticii în clauze Prolog executabile. În rest, regulile sunt memorate ca fapte Prolog şi sunt manevrate în momentul execuţiei). O analiză comparativă a vitezei algoritmilor amintiţi conduce la concluzia [5] că, pe măsură ce algoritmul devine mai "bun", procesul de analiză sintactică se realizează din ce în ce mai încet, unul dintre algoritmii cei mai neperformanţi, *din acest punct de vedere*, fiind cel al lui Earley. Rezultă în mod clar că analiza sintactică bazată pe constituenţi este un proces care nu se realizează uşor şi că, atât analizorii sintactici bazaţi pe backtracking, cât şi cei bazaţi pe utilizarea unei structuri de date de tip hartă, care reduce backtracking-ul, prezintă imperfecţiuni şi neajunsuri. Tocmai de aceea proiectarea de analizori sintactici performanţi rămâne o problemă de actualitate şi reprezintă una dintre sferele de preocupări de cea mai mare importanţă în domeniul procesării limbajului natural.

Algoritmul lui Earley este extrem de important din punct de vedere "istoric", deoarece el a demonstrat că analiza sintactică bazată pe reguli PS se poate realiza în *timp polinomial*, i.e. într-un interval de timp proporţional cu n^k, unde n este lungimea şirului de intrare, iar k reprezintă o constantă. În continuare, Earley a demonstrat că întotdeauna $k \leq 3$. Acesta este un rezultat extrem de important, deoarece mulţi alţi analizori sintactici (cum ar fi cel cu deplasare - reducere) necesită, în cazul cel mai nefavorabil, *timp exponenţial* i.e. proporţional cu k^n (care, pentru o valoare ridicată a lui n, este mult mai mare decât n^3). Analizorii sintactici de tipul celui cu deplasare - reducere necesită un interval de timp exponenţial, întrucât ei încearcă determinarea separat a fiecărui arbore de derivare. Prin contrast, Algoritmul lui Earley a demonstrat că analiza sintactică poate fi organizată în jurul unor principii şi nu prin încercarea consecutivă a tuturor combinaţiilor posibile.

Aşa cum se observă însă în [5], aceste rezultate celebre au, din diverse motive, o importanţă mai mică decât ar putea părea la prima vedere. Astfel, realitatea lingvistică arată că un parser pentru limbajul natural nu se va confrunta niciodată cu valori mari ale lui n. Chiar şi analizorii sintactici care lucrează în timp exponenţial pot fi rapizi atunci când n este mic, iar în limbile naturale este puţin probabil să întâlnim propoziţii alcătuite din mai mult de 30 de cuvinte. Atunci când intervin, cu adevărat, propoziţii sau fraze lungi, acestea pot fi despărţite în propoziţii de lungime mai mică, legate între ele prin conjuncţii de tipul lui *şi*, care pot fi analizate separat. Mai mult, rezultatele

referitoare la n^3 şi k^n sunt cele obţinute pentru cazul cel mai nefavorabil, adică în situaţia în care parser-ul ia, pe cât posibil, decizii greşite, utilizând regula indicată numai după ce le-a încercat pe toate cele care nu se aplică. Această situaţie extremă se regăseşte relativ rar în practică. Un al treilea motiv invocat [5] este acela că rezultatul privitor la n^3 este valabil numai în cazul gramaticilor în care nodurile nu au argumente. Barton, Berwick şi Ristad (1987) au demonstrat că procesul de parsing, în cazul general, este NP - COMPLET - adică aparţine unei clase de probleme care necesită timp exponenţial - dacă li se permite gramaticilor să aibă *caracteristici* (i.e. să admită argumente ale nodurilor, cum ar fi *singular* sau *plural*, împreună cu regulile de concordanţă aferente) şi dacă cuvintele pot fi ambigue. Gramaticile care descriu limbajul natural necesită, în mod evident, aceste cerinţe.

Progresul în dezvoltarea unor analizori sintactici mai eficienţi pare, aşadar, să depindă, cu precădere, de următorii doi factori: studiul atent al complexităţii în cazurile tipice şi/sau de frecvenţă medie (şi nu în cazul cel mai nefavorabil), precum şi continua descoperire a constrângerilor insuficient explorate cărora trebuie să li se supună gramaticile referitoare la limbajul natural.

ANEXA 1. Adjectivele în WordNet

WordNet (WN) împarte adjectivele în două mari clase[35]: adjective *descriptive* şi *relaţionale*. Adjectivele cromatice, care desemnează culori, sunt privite ca reprezentând un caz special.

Un *adjectiv descriptiv* este un adjectiv care atribuie unui substantiv o valoare a unui atribut. Cu alte cuvinte, *x este Adj* presupune existenţa unui atribut A astfel încât *A(x) = Adj*. De pildă, *scund* şi *înalt* sunt valori ale atributului ÎNĂLŢIME. WN conţine pointeri între adjectivele descriptive şi synset-urile de substantive care se referă la atributele adecvate.

Organizarea semantică a adjectivelor descriptive este fundamental diferită de cea a substantivelor. Relaţia hiponimică, care generează ierarhiile substantivale, nu este disponibilă şi în cazul adjectivelor. Organizarea semantică a adjectivelor este gândită, în mod mai natural, ca reprezentând un hiperspaţiu abstract cu *N* dimensiuni şi nu un arbore ierarhic. Relaţia semantică de bază dintre adjectivele descriptive este *antonimia*.

Importanţa antonimiei în organizarea adjectivelor descriptive devine evidenta dacă avem în vedere faptul că funcţia acestor adjective este aceea de a exprima valori ale atributelor, iar atributele sunt, în marea lor majoritate, bipolare. Adjectivele antonime exprimă valori opuse ale unui atribut. Spre exemplu, antonimul lui *greu* este *uşor*, care exprimă o valoare de la polul opus al atributului GREUTATE. În WN această opoziţie binară este reprezentată prin intermediul pointerilor reciproci etichetaţi: ***greu!->uşor*** **si** ***uşor!->greu***. În implementarea în Prolog a WN, pe care am folosit-o aici, operatorul **ant** specifică cuvinte antonime şi toate faptele Prolog care utilizează acest operator sunt incluse în fişierul **wn_ant.pl**. Adjectivele descriptive care nu posedă antonime directe sunt privite ca având antonime indirecte, datorită similaritaţii lor semantice cu adjective care au antonime directe. Un pointer de similaritate a fost folosit pentru a indica faptul că toate adjectivele care nu posedă antonime sunt similare ca sens cu adjective care au antonime. În implementarea în Prolog a WN operatorul **sim** specifică faptul că două synset-uri sunt similare ca înţeles şi toate faptele Prolog care utilizează acest operator sunt incluse în fişierul **wn_sim.pl**.

Prin urmare, adjectivele descriptive atribuie "substantivelor cap" corespunzătoare valori ale unor atribute cel mai adesea bipolare şi, în consecinţă, sunt organizate în termeni de *opoziţie binară (antonimie)* şi *similaritate a sensului (sinonimie)*.

---

[35] Prezentarea adjectivelor în WordNet este făcută aici conform [22].

Gross, Fischer şi Miller (1989) propun ca synset-urile de adjective să fie privite ca nişte ***cluster-e de adjective*** asociate prin similaritate semantică unui adjectiv focal, care face legătura dintre cluster-ul respectiv şi un cluster contrastant, aflat la polul opus al atributului. Tot Gross, Fischer şi Miller fac distincţia între antonimele directe, cum ar fi *heavy / light (greu / uşor)*, care sunt opuse conceptual, reprezentând, în acelaşi timp, perechi lexicale şi între antonimele indirecte, cum ar fi *heavy / weightless (greu / lipsit de greutate)*, care sunt opuse conceptual fără a reprezenta perechi lexicale. Sub acestă formulare, toate adjectivele descriptive au antonime; cele cărora le lipsesc antonimele directe au antonime indirecte i.e. sunt sinonime ale unor adjective care au antonime directe.

În WN antonimele directe sunt reprezentate prin pointerul de antonimie '!->'; antonimele indirecte sunt moştenite prin similaritate, care este indicată prin pointerul de similaritate '&->'. Configuraţia rezultată este ilustrată în Fig.1.1, pentru cluster-ul de adjective centrat în jurul antonimelor directe *wet / dry (umed / uscat)*, care definesc atributul WETNESS sau MOISTNESS (UMEZEALĂ), un exemplu oferit în [22]. Atunci când se analizează acest cluster de adjective englezeşti se poate observa, spre exemplu, că *moist* nu are un antonim direct, dar antonimul său indirect poate fi găsit via drumul *moist&->wet!->dry*.

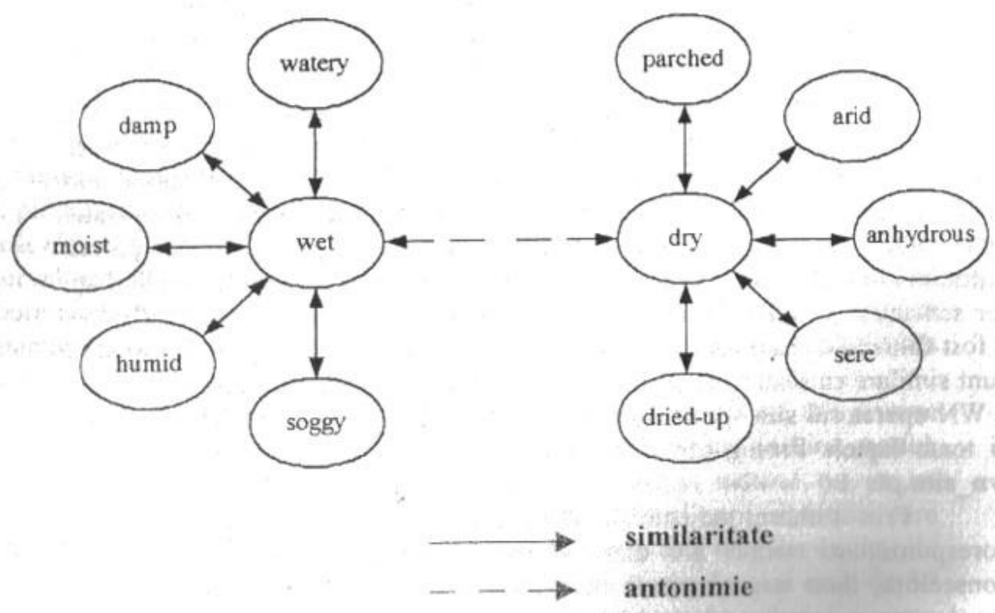


Fig.1.1 Structură bipolară de adjective

Corespunzător Figurii 1.1, care prezintă, în mod intuitiv, structura unui cluster de adjective în WN, se obţine următorul cluster bipolar, având capul reprezentat de perechea de antonime *wet / dry*:

[{ [WET, DRY, !] watery,& damp,& moist,& humid,& soggy,& }

{ watery, tearful, teary, wet, & }

{ damp, wet,& }

{ moist, wet,& }

{ humid, muggy, steamy, sticky, sultry, wet,& }

{ soggy, saturated, sodden, waterlogged, wet,& }

-

{ [DRY, WET,!] parched,& arid,& anhydrous,& sere,& dried-up,& }

{ parched, dehydrated, desiccated, dry,& }

{ arid, waterless, dry,& }

{ anhydrous, dry,& ((chem) with all water removed)}

{ sere, shriveled, withered, wizened, dry,& (used of vegetation)}

{ dried-up, dry,& ("a dry water hole") }].

Se poate observa că un cluster are două părţi distincte. Fiecare jumătate a unui cluster începe cu un synset numit *synset cap*. Primele două elemente ale synset-ului cap reprezintă perechea de antonime care defineşte cluster-ul şi sunt scrise cu majuscule. Această pereche de antonime este urmată de adjective reprezentând pointeri de similaritate (watery,& damp,& etc.), unul către fiecare synset similar ca sens din jumătatea respectivă a cluster-ului. Fiecare astfel de synset conţine un pointer reciproc pentru întoarcerea la synset-ul cap. Se observă, de asemenea, că pointerii de similaritate care intervin în synset-ul cap sunt, de fapt, cuvinte care ocupă prima poziţie în cadrul synset-urilor înrudite prin similaritate cu synset-ul căruia îi aparţine adjectivul ce are un antonim direct. Cele două părţi distincte ale cluster-ului sunt separate printr-o cratimă, iar întreaga structură este inclusă între paranteze drepte. Fiecare cluster bipolar este de sine stătător, iar codificarea este restricţionată la relaţiile din interiorul cluster-ului.

Semnificaţia excepţiilor care, fără îndoială, există nu este majoră şi considerăm, împreună cu autorii WN [Miller et. al., 90], că modelul prezentat acoperă marea majoritate a adjectivelor descriptive englezeşti. Importanţa relaţiei de similaritate este evidentă.

În WN relațiile semantice sunt reprezentate printr-o pereche de id-uri de synset (*synset_id*-uri), pereche în cadrul căreia primul *synset_id* reprezintă, în general, sursa relației, iar cel de-al doilea ținta acesteia.

Un *synset_id* este un câmp de 9 bytes în care primul byte definește categoria sintactică a synset-ului, iar cei 8 bytes rămași reprezintă un *synset_offset*. În versiunea Prolog a bazei de date WN, *synset_id*-urile sunt folosite ca identificatori unici de synset-uri. Așa cum s-a remarcat deja, în implementarea în Prolog a WN operatorul **sim** este folosit pentru a desemna relația de similaritate, ca în exemplul **sim**(302425348, 302425924). În general,

sim(*synset_id, synset_id*).

este o faptă Prolog care specifică faptul că al doilea synset este similar ca sens cu primul. Aceasta înseamnă că cel de-al doilea synset este un satelit al primului synset, care reprezintă capul cluster-ului. Această relație este valabilă numai pentru synset-uri de adjective conținute în cluster-e de adjective.

În WN *adjectivele relaționale*, care au fost discutate, mai întâi, pe larg, de catre Levi (1978), înseamnă ceva de tipul "relativ la, referitor la, ținând de sau asociat cu" un substantiv. Adjectivele relaționale se deosebesc de cele descriptive prin aceea că ele nu pot fi puse în legătură cu un atribut. Prin urmare, ele nu se referă la o proprietate a substantivului cap. Întrucât adjectivele relaționale nu au antonime, ele nu pot fi încorporate în cluster-ele care caracterizează adjectivele descriptive. WN menține un fișier separat de adjective relaționale cu pointeri către substantivele corespunzătoare. Fiecare synset constă din unul sau mai multe adjective relaționale urmate de un pointer către substantivul corespunzător.

Lucrarea de față se ocupă de generarea semiautomată a *cluster-elor de adjective* din alte limbi decât engleza și, prin urmare, se referă exclusiv la adjectivele descriptive, care pot fi organizate sub forma unei structuri de acest tip.

ANEXA 2. Notații și formule matematice

1. Repartiții de probabilitate multidimensionale

Fie $E=(X_1,\ldots,X_n)$ un vector aleator format din n variabile discrete. Repartiția sa, $P_0(X_1,\ldots,X_n)^{-1}$, este complet specificată de valorile

$$p(x_1,\ldots,x_n)=P(X_1=x_1,\ldots,X_n=x_n)\geq 0$$

$$\sum_{(x_1,\ldots,x_n)} p(x_1,\ldots,x_n)=1,$$

unde $(x_1,\ldots,x_n)$ este o "realizare" a lui E.

În general, pentru a desemna $P_0(X_1,\ldots,X_n)^{-1}$, se folosește notația simplificată $P(E)$.

În acest context, are loc următoarea formulă de înmulțire a probabilităților:

$$P(X_1=x_1,\ldots,X_n=x_n)=P(X_1=x_1)\cdot P(X_2=x_2\mid X_1=x_1)\cdot\ldots$$
$$\cdot P(X_n=x_n\mid X_1=x_1,\ldots,X_{n-1}=x_{n-1})$$

2. Repartiții condiționate

Fie Y o variabilă aleatoare discretă și $E=(X_1,\ldots,X_n)$ un vector aleator cu componente discrete. Corpul de evenimente generate de E este finit, având ca generatori pe $\{X_1=x_1,\ldots,X_n=x_n\}$.

Pentru fiecare realizare $(x_1,\ldots,x_n)$ a lui E, notăm probabilitatea condiționată a unui eveniment A cu

$$P(A\mid(x_1,\ldots,x_n))=\frac{P(A\cap\{X_1=x_1,\ldots,X_n=x_n\})}{P(X_1=x_1,\ldots,X_n=x_n)}$$

Corespunzător variabilei Y putem lua în considerație:

- repartiția lui Y, $P(Y)$, dată de valorile

$$p(y)=P(Y=y);$$

- repartiția lui Y condiționată de E, $P(Y \mid E)$, care este dată, pentru fiecare realizare $(x_1,\ldots,x_n)$, de valorile

$$p(y \mid (x_1,\ldots,x_n)) = P(\{Y=y\} \mid (x_1,\ldots,x_n)).$$

În acest context se verifică versiunea condiționată a formulei de înmulțire a probabilităților:

$$P(X_1 = x_1,\ldots,X_n = x_n \mid C) = P(X_1 = x_1 \mid C)\cdot P(X_2 = x_2 \mid C, X_1 = x_1)\cdot\ldots$$
$$\cdot P(X_n = x_n \mid C, X_1 = x_1,\ldots,X_{n-1} = x_{n-1}).$$

3. Formula lui Bayes

Fie E și Y cu aceeași semnificație din §1 și §2.

- Formula lui Bayes în *versiune necondiționată* este utilizată pentru a exprima o repartiție a posteriori

$$P(E \mid Y=y) = \frac{P(E)\cdot P(\{Y=y\} \mid E)}{P(\{Y=y\})}$$

unde:

$P(E)$ este repartiția a priori a lui E ;

$\{Y=y\}$ este evenimentul observat ;

$P(E \mid Y=y)$ este repartiția a posteriori a lui E ;

$P(\{Y=y\} \mid E)$ este complet determinată de valorile

$$P(\{Y=y\} \mid \{X_1 = x_1,\ldots,X_n = x_n\}).$$

- Formula lui Bayes în *versiune condiționată* este utilizată pentru a exprima o repartiție a posteriori condiționată

$$P(Y \mid E,e) = \frac{P(Y \mid E)\cdot P(e \mid E,Y)}{P(e \mid E)}$$

unde:

$P(Y \mid E)$ este o probabilitate condiționată, a priori ;

e reprezintă o nouă dovadă ;

$P(Y \mid E,e)$ este o probabilitate condiționată, a posteriori.

BIBLIOGRAFIE

1. ALLEN, J. F., *Natural Language Understanding*. Menlo Park, California, Benjamin - Cummings, 1987.

2. ATSERIAS, J., CLIMENT, S., FARRERES, X., RIGAU, G., RODRIGUEZ, H., "Combining Multiple Methods for the Automatic Construction of Multi-lingual WordNets", în: *Recent Advances in Natural Language Processing II. Selected papers from RANLP '97* (Editori Nicolas Nicolov și Ruslan Mitkov). John Benjamins Publishing Company, Amsterdam/Philadelphia, 1997, p. 327-338.

3. BRATKO, I., *Prolog Programming for Artificial Intelligence* (ed. a II-a). Workingham, Addison Wesley, 1990.

4. CLOCKSIN, W. F., MELLISH, C. S., *Programming in Prolog*. Springer-Verlag, 1994.

5. COVINGTON, M. A., *Natural Language Processing for Prolog Programmers*. Englewood Cliffs, NJ, Prentice Hall, 1994.

6. DURKIN, J., *Expert Systems Design and Development*. Prentice Hall, 1994.

7. FELLBAUM, C. (Editor), *WordNet: An Electronic Lexical Database*. The MIT Press, Cambridge/London/England, 1998.

8. FLOREA, A. M., *Elemente de inteligență artificială. Vol. I, Principii și modele*. Lito. UPB, Bucuresti, 1993.

9. GAZDAR, G., MELLISH, C., *Natural Language Processing in Prolog: an Introduction to Computational Linguistics*. Wokingham, Addison - Wesley, 1989.

10. HRISTEA, F., *Introducere în procesarea limbajului natural cu aplicații în Prolog*. Editura Universității din București, 2000.

11. HRISTEA, F., "On the Semiautomatic Generation of WordNet Type Synsets and Clusters", *Journal of Universal Computer Science*, vol. 8, no. 12, 2002, p. 1047-1064.

12. HRISTEA, F., "On the Semiautomatic Generation of Verb Synsets in Languages other than English", *Analele Universității din București*, anul LII, 2003, p. 75-86.

13. HRISTEA, F.: "Asupra generării semiautomate a synset-urilor și cluster-elor de tip WordNet cu specială referire la limba română", în: *Building Awareness in Language Technology* (Editori: Florentina Hristea și Marius Popescu). Editura Universității din București, 2003, p. 369-397.

14. HRISTEA, F., BALCAN, M. F., *Aspecte ale căutării și reprezentării cunoștințelor în inteligența artificială*. Editura Universității din București, 2004.

15. HRISTEA, Th.: "Unele comentarii lingvistice privind rezultatele obtinute", în: *Building Awareness in Language Technology* (Editori: Florentina Hristea și Marius Popescu). Editura Universității din București, 2003, p. 409-413.

16. LEVIȚCHI, L., *Dicționar român-englez* (ed. a III-a). Editura Științifică, 1973.

17. LEVIȚCHI, L., BANTAȘ, A., NICOLESCU, A., *Dicționar englez-român*. Editura Academiei Române, 1974.

18. LUGER, G. F., STUBBLEFIELD, W. A., *Artificial Intelligence and the Design of Expert Systems*. Redwood City, California: Benjamin / Cummings, 1989.

19. LUGER, G. F., STUBBLEFIELD, W. A., *Artificial Intelligence. Structures and Strategies for Complex Problem Solving* (ed. a III-a). Addison – Wesley, 1998.

20. MERRITT, D., *Building Expert Systems in Prolog*. Springer-Verlag, 1989.

21. MICROSOFT PRESS. *Dicționar de calculatoare*. (Traducere de Nicolae Pora). Editura Teora, Bucuresti, 1999.

22. MILLER, G.A., BECKWITH, R., FELLBAUM, C., GROSS, D., MILLER, K.J., "Introduction to WordNet: an on-line lexical database", *International Journal of Lexicography*, 3,4, 1990, p. 235-244.

23. NIKOLOV, T., PETROVA, K., "Building and Evaluating a Core of Bulgarian WordNet for Nouns", OntoLex '2000 Report, Sozopol, Bulgaria, 2000.

24. NIKOLOV, T., PETROVA, K.: "Towards Building Bulgarian WordNet", Proc. RANLP '01, INCOMA Ltd., Tzigov Chark, Bulgaria, 2001, p. 199-203.

25. NILLSON, N., *Artificial Intelligence: A New Synthesis*. Morgan Kauffman, 1998.

26. RICH, E., KNIGHT, K., *Artificial Intelligence* (ed. a II-a). Tata McGraw-Hill Publishing Company Limited, New Delhi, 1993.

27. RUSSELL, S. J., NORVIG, P., *Artificial Intelligence. A Modern Approach*. Prentice - Hall International, Inc., 1995.

28. *SICSTUS PROLOG USER'S MANUAL*. Intelligent Systems Laboratory, Sweedish Institute of Computer Science; http://www.sics.se/isl/sicstuswww/site/documentation.html

29. SIMA, C., VAȚĂ, C., "On the Semiautomatic Generation of Romanian Noun Synsets", *Analele Universității din București*, Anul LIII, No.1, 2004, p. 125-136.

30. WALKER, A. (Editor), *Knowledge Systems and Prolog: A Logical Approach to Expert Systems and Natural Language Processing*. Reading, Massachusetts: Addison - Wesley, 1987.

CUPRINS

CUVÂNT ÎNAINTE 3

1. CONSIDERAȚII PRELIMINARE 5

1.1. Domeniul inteligenței artificiale. Definiții și un scurt istoric 6

1.2. Subdomenii ale inteligenței artificiale 9

1.3. Tehnici ale inteligenței artificiale 11

1.4. Un exemplu de problemă tipică în inteligența artificială 13

2. TEHNICI DE CĂUTARE 23

2.1. Rezolvarea problemelor prin intermediul căutării 24

2.1.1. Tipuri de probleme 25

2.1.2. Probleme și soluții corect definite 26

2.1.3. Căutarea soluțiilor și generarea secvențelor de acțiuni 29

2.1.4. Structuri de date pentru arbori de căutare 30

2.1.5. Evaluarea strategiilor de căutare 31

2.2. Căutarea neinformată 31

2.2.1. Căutarea de tip breadth-first 32

2.2.1.1. Algoritmul de căutare breadth-first 32

2.2.1.2. Implementare Prolog 33

2.2.1.3. Timpul și memoria cerute de strategia breadth-first 37

2.2.1.4. Un exemplu. Problema misionarilor și canibalilor 38

2.2.2. Căutarea de tip depth-first 42
2.2.2.1. Prezentare generală şi comparaţie cu strategia breadth-first 42
2.2.2.2. Implementare Prolog 44
2.2.3. Căutarea în adâncime iterativă 49
2.2.3.1. Prezentare generală 49
2.2.3.2. Implementare Prolog 50
2.2.4. Strategii de căutare neinformată. Un exemplu 52
2.3. Căutarea informată 59
2.3.1. Căutarea de tip best-first 59
2.3.2. Algoritm de căutare general bazat pe grafuri 61
2.3.2.1. Algoritmul A* 62
2.3.2.1.1. Admisibilitatea Algoritmului A* 65
2.3.2.1.2. Condiţia de consistenţă 70
2.3.2.1.3. Optimalitatea Algoritmului A* 73
2.3.2.1.4. Completitudinea Algoritmului A* 74
2.3.2.1.5. Complexitatea Algoritmului A* 74
2.3.2.2. Iterative Deepening A* (IDA*) 75
2.3.3. Implementarea Prolog a căutării de tip best-first 76
2.3.3.1. Exemple 82
2.3.4. Concluzii 91

3. JOCURILE CA PROBLEME DE CĂUTARE 95
3.1. O definire formală a jocurilor 96
3.2. Algoritmul Minimax 97
3.2.1. Funcţii de evaluare 100

3.2.2. Implementare Prolog 100
3.3. O implementare eficientă a principiului Minimax: Algoritmul Alpha-Beta 102
3.3.1. Implementare Prolog 106
3.3.2. Consideraţii privitoare la eficienţă 107
3.4. Un exemplu. Jocul X şi 0 108

4. ASPECTE ALE REPREZENTĂRII CUNOŞTINŢELOR 121
4.1. Tipuri de cunoştinţe 122
4.1.1. Cunoştinţe relaţionale simple 122
4.1.2. Cunoştinţe care se moştenesc 122
4.1.3. Cunoştinţe inferenţiale 124
4.1.4. Cunoştinţe procedurale 125
4.2. Clase de metode pentru reprezentarea cunoştinţelor 125
4.3. Reprezentarea cunoştinţelor în sistemele expert 126
4.3.1. Structura de bază a unui sistem expert 127
4.3.2. Reprezentarea cunoştinţelor cu reguli if-then 128
4.3.3. Înlănţuire înainte şi înapoi în sistemele bazate pe reguli 129
4.3.3.1. Înlănţuirea înapoi 130
4.3.3.2. Înlănţuirea înainte 132
4.3.3.3. Concluzii 133
4.3.4. Generarea explicaţiilor 134
4.3.5. Introducerea incertitudinii 136
4.3.6. Un exemplu de sistem expert. Implementare Prolog 138
4.4. Reţele semantice şi cadre 169

4.4.1. Reţele semantice 169
4.4.2. Cadre 174
4.5. Consideraţii finale 177

5. GENERAREA SEMIAUTOMATĂ A REŢELELOR SEMANTICE DE TIP WORDNET 179
5.1. Consideraţii introductive 180
5.2. Algoritmul de traducere 181
5.3. Synset-uri de substantive 185
5.4. Synset-uri şi cluster-e de adjective 188
5.4.1. Generarea semiautomată a synset-urilor de adjective 188
5.4.2. Generarea semiautomată a cluster-elor de adjective 196
5.4.3. Consideraţii finale privitoare la adjective 198
5.5. Alte comentarii lingvistice privitoare la rezultatele obţinute 199
5.6. Revizitarea synset-urilor de substantive 202
5.6.1. Consideraţii finale privitoare la substantive 208
5.7. Synset-uri de verbe 209
5.7.1. Exemple şi consideraţii finale privitoare la verbe 211
5.8. Synset-uri de adverbe 214
5.9. Consideraţii finale 218

6. RAŢIONAMENT STATISTIC 219
6.1. Abordări statistice pentru sisteme de raţionament incert 219
6.1.1. Factori de certitudine şi sisteme bazate pe reguli 221
6.1.2. Reţele Bayesiene 225
6.1.2.1. Relaţii de independenţă condiţionată în reţele Bayesiene 228

6.1.2.2. Inferenţa în reţele Bayesiene 229
6.1.2.2.1. Un algoritm pentru răspunsul la interogări 230

7. CĂUTARE ŞI REPREZENTAREA CUNOŞTINŢELOR ÎN PROCESAREA LIMBAJULUI NATURAL. ANALIZA SINTACTICĂ 241
7.1. Noţiunea de grup sintactic 243
7.2. Gramatici 245
7.2.1. Noţiunea de constituent 247
7.2.2. Reguli PS şi gramatici PS. Gramatici generative 249
7.2.3. Clase de gramatici şi limbaje 254
7.2.4. Reprezentarea gramaticilor în Prolog 256
7.2.4.1. Gramatici DC 261
7.2.4.2. Gramatici cu caracteristici şi reprezentarea lor în notaţia DCG 268
7.3. Analiza sintactică bazată pe constituenţi 271
7.3.1. Analiza sintactică top-down 274
7.3.1.1. Algoritm de analiză sintactică top-down 277
7.3.1.2. Implementare Prolog 282
7.3.2. Analiza sintactică bottom-up 284
7.3.2.1. Algoritm de deplasare-reducere 285
7.3.2.2. Implementare Prolog 286
7.3.3. Analiza sintactică din colţul stâng 289
7.3.3.1. Implementare Prolog 290
7.3.3.1.1. Legături 292
7.3.3.1.2. BUP 296
7.3.4. Asupra eficienţei strategiilor de tip bottom-up

și top-down; ambiguitate 298
7.3.5. Memorarea rezultatelor intermediare 301
7.3.5.1. Tabel de subșiruri bine formate 301
7.3.5.2. Harta activă 306
7.3.5.2.1. Analiza sintactică cu hartă 309
7.3.5.3. Implementarea unui parser bottom-up cu hartă 315
7.3.5.4. Strategii alternative de invocare a regulilor 321
7.3.5.5. Implementarea unui parser top-down cu hartă 322
7.3.5.6. Strategia de căutare: utilizarea agendei 327
7.3.5.7. Implementarea controlului flexibil 329
7.3.5.7.1. Implementarea unui parser bottom-up cu hartă și agendă 329
7.3.5.7.2. Implementarea unui parser top-down cu hartă și agendă 336
7.3.5.7.3. Variante ale analizorilor sintactici bottom-up și top-down cu hartă și agendă ... 341
7.3.5.8. Eficiență 344

ANEXA 1. Adjectivele în WordNet 349

ANEXA 2. Notații și formule matematice 353

BIBLIOGRAFIE 355

Cumpărătoarea
RMF 113/20.09.2010.