

Cuvânt înainte

Acest curs se adresează studenților Facultății de Informatică din Universitatea “Alexandru Ioan Cuza” din Iași. El grupează materiale pe care primul autor le-a predat în cadrul cursului de Inteligență Artificială de-a lungul mai multor ani, dar și capitole noi, contribuite de ceilalți doi autori foarte recent. Un curs cu acest nume a fost introdus în facultatea noastră în anul 1989, deși anumite module care țin de această disciplină fuseseră predate cu mult timp înainte: astfel prof. Costică Cazacu predă de mult timp logica rezolutivă a lui Robinson iar prof. Călin Ignat predă la acea dată limbajul Prolog.

De la an la an acest curs s-a modificat (poate nu întotdeauna către mai bine), pe de o parte datorită nestatorniciei domeniului, iar pe de o altă parte, pentru că unui profesor îi e imposibil să repete aceleași lucruri în doi ani succesivi. Idei vehiculate la cursuri și notate grăbit, însemnări făcute la seminar, copii după cursuri luate de mână de studenți și cerute lor după ore, toate acestea s-au adunat și au format, în timp, un material semnificativ. Doar de câțiva ani am hotărât să culeg aceste materiale în format electronic, întârziere care a fost datorată în mare parte descurajării mele în fața imensității materialelor publicate în acest domeniu. Dar dacă cedezi în fața gândului că orice ai scrie, cineva trebuie să mai fi spus acel lucru altundeva, anii trec, studenții vin și pleacă și nimic nu prinde contur și nu rămâne... La un moment dat, totuși, trebuie să lași în urma ta o carte. Chiar și atunci când domeniul predat nu coincide cu cel în care îți duci activitatea de cercetare, adică cel în care inventezi, încă poți fi scuzat să scrii o carte asupra unui domeniu pe care îl cunoști (desigur, parțial) dacă maniera în care îl descrii îți aparține și dacă, adoptând-o în predare, studenții participă, se implică și se simt bine. Predarea nu-și are rost decât atunci când este înțeleasă ca un act de creație. Deși un violonist nu scrie partitura pe care o execută, felul în care interpretează textul muzical îl transformă într-un creator sau, dimpotrivă, îl arată ca fiind un reproducător de duzină.

Cartea de față, poate la fel ca multe altele, cu siguranță n-ar fi existat sau ar fi fost mult mai subțire dacă aș fi ținut ca toate ideile redată în ea să fi fost inventate de noi, cei trei autori. Cele mai frumoase dintre ideile pe care le-am descris aici au fost inventate de alții, noi nefiind decât interpreții lor, e drept, în propriul nostru stil. Mai toate exemplele însă, cu câteva excepții (cazuri în care am menționat sursele), sunt originale. Cele mai multe dintre ele nu au anvergura unor aplicații reale ci sunt circumscrise tipului “probleme jucărie”, atât de mult agreate în didactica acestui domeniu. Câteva sunt decupate însă din aplicații reale și diminuate până la “jucărioare” ușor de explicat și formalizat.

În toți acești ani de când predau acest curs la Facultate, am avut bucuria să lucrez cu mai mulți tineri care m-au ajutat la catedră ca asistenți. Fiecare dintre ei a lăsat urme asupra generațiilor de studenți și, în diverse moduri, probabil, și asupra acestei cărți. Ei sunt: Andrei Adrian, Amalia Todirașcu, Adrian Iftene, Petrică Obreja, precum și colaboratorii mei actuali, co-autorii, Mădălina Ioniță și Ionuț Pistol. Le mulțumesc tuturor, după cum le mulțumesc acelor studenți ai mei care mi-au făcut bucuria să-i reîntâlnesc, la fiecare curs, atenți și deschiși să primească noi provocări...

Capitolele din carte au fost contribuite după cum urmează: capitolele 1-4 – primul autor, capitolul 5 – al doilea autor, capitolul 6 – al treilea autor.

Dan Cristea

Cuprins

1. Introducere.....	5
1.1 Definiția Inteligenței Artificiale și un pic de istorie	5
1.2 Testul Turing.....	5
1.3 Probleme de natură filosofică și pragmatică	6
1.4 Subdomenii ale inteligenței artificiale.....	8
1.5 Probleme și soluții IA față de probleme și soluții de programare clasică	11
1.5.1 Prin ce diferă o problemă clasică de una de IA ?.....	11
Bibliografie	12
2. LISP	13
2.1 Introducere	13
2.2 Funcții, maniere de definire și apel	15
2.2.1 Transparența referențială	15
2.2.2 Efectul lateral	15
2.2.3 De la notația lambda la Lisp pur	16
2.2.4 Evaluarea numerică față de evaluarea simbolică.....	17
2.2.5 Beneficiile unui Lisp impur: Common Lisp	17
2.3 Tipuri de date în Lisp.....	17
2.3.1 Construcțiile limbajului	18
2.3.2 Un pic de sintaxă	18
2.3.2 Liste și reprezentarea lor prin celule cons.....	21
2.3.3 Perechi și liste cu punct	22
2.3.4 Evaluarea expresiilor	22
2.4 Funcții și forme Lisp.....	24
2.4.1 Notatii	24
2.4.2 Asignarea unei valori unui simbol	26
2.4.3 Funcții pentru controlul evaluării.....	27
2.4.4 Operații asupra listelor	27
2.4.5 Operații cu numere	29
2.4.6 Predicate.....	33
2.4.7 Liste privite ca mulțimi	36
2.4.8 Operații logice și evaluări controlate	37
2.4.9 Forme pentru controlul evaluării.....	38
2.4.10 Liste și tabele de asociație	39
2.4.11 Locații și accese la locații.....	41
2.4.12 Lista de proprietăți a unui simbol.....	41
2.4.13 Funcții chirurgicale.....	42
2.4.14 Forme de apelare a altor funcții.....	44
2.4.15 Lambda expresii	44
2.4.16 Lambda-funcții recursive	45
2.4.17 Funcții de corespondență.....	45
2.5 Tehnici de programare în Lisp	48
2.5.1 Definiții de funcții	48
2.5.2 Recursivitate	48
Recursivitate coadă.....	50
Scrierea funcțiilor cu recursivitate coadă prin parametru acumulator.....	50
2.5.3 Forme de secvențiere	51
2.5.4 Formele speciale <code>let</code> și <code>let*</code>	51
2.5.5 Variabile și domeniile lor	53
2.5.6 Legări versus asignări.....	55
2.5.7 Forme de iterare	56
2.5.8 Valori multiple și exploatarea lor.....	58
2.5.9 Închideri	59
2.5.10 Transferul argumentelor în funcții	60
2.6 Macro-uri	62
2.6.1 Definiție, macroexpandare și evaluare	62
2.6.2 Despre apostrof-stânga (<i>backquote</i>)	63

2.6.3 Asupra manierei de construcție a macro-urilor.....	64
2.6.4 Cum se testează macro-expandările?	66
2.6.5 Asupra destructurizării	66
2.6.6 Când să folosim macro-uri?	67
2.6.7 Argumente pro și contra utilizării macro-urilor.....	67
2.7 Un exemplu de program care se modifică în timpul rulării.....	68
Bibliografie	70
3. Căutare. Rezolvarea problemelor. Sisteme de producție.....	72
3.1 Formalizarea problemelor de IA	72
3.1.1 Problemă, instanță de problemă, spațiul stărilor.....	72
3.1.2 Recunoașterea stărilor	74
3.1.3 Reprezentarea stărilor.....	76
3.1.4 Reprezentarea tranzițiilor între stări.....	76
3.1.5 Căutarea soluției	78
3.2 Exemple de formalizări	80
3.2.1 Maimuța și banana.....	80
3.2.2 Lumea cuburilor (blocurilor).....	84
3.3 Controlul sistemelor de producție	89
3.3.1 Strategii irevocabile.....	89
3.3.2 Strategii tentative.....	94
3.4 Sisteme Expert	108
3.4.1 Tipuri de căutări.....	108
3.4.2 Sistemele Expert (SE).....	109
3.5 Concluzie	109
Bibliografie	113
4. Reprezentarea cunoașterii și raționament	115
4.1 Ontologii	116
4.2 Taxonomii	117
4.3 Paternitate <i>versus</i> monotonie în sisteme ierarhice.....	118
4.3.1 Soluții în cazul moștenirilor contradictorii	120
4.4 Rețelele semantice descriptive – sisteme ierarhice nemonotone.....	120
4.4.1 Un exemplu de reprezentare.....	120
4.4.2 Interogări în rețele semantice	123
4.4.3 Demonii.....	127
4.4.4 Ambiguități în reprezentarea prin rețele semantice	130
4.5 Rețele semantice evenimențiale. Inferențe.....	131
4.5.1 Evenimente aflate în corelație logică.....	133
4.5.2 Un model inferențial.....	134
Bibliografie	144
5. Probleme de satisfacere a constrângerilor	146
5.1 Introducere	146
5.2 <i>Backtracking</i> în satisfacerea constrângerilor	149
5.2.1 <i>Backjumping</i>	150
5.2.2 <i>Forward-checking</i>	152
5.3 Inferența	154
5.3.1 Arc-consistență	155
5.3.2 <i>Bucket-Elimination</i>	157
5.4 Euristici pentru selectarea variabilelor și a valorilor	160
5.4.1 Ordonarea variabilelor.....	160
5.4.2 Selectarea valorilor.....	161
5.5 Metode stochastice.....	162
5.6 Concluzii	162
Probleme	163
Bibliografie	164
6. Învățarea automată	166
6.1 Descriere generală	166
6.2 Caracteristici ale unui sistem capabil de învățare	167
6.2.1 Scopul metodei și baza de cunoștințe necesară.....	167

6.2.2 Formalismul de reprezentare a datelor utilizate și a celor învățate	167
6.2.3 Setul de operații	168
6.2.4 Spațiul general al problemei.....	169
6.2.5 Reguli euristice pentru căutare.....	169
6.3. Învățarea supervizată	169
6.3.1 Concepte învățabile PAC	169
6.3.2 Arbori de decizie	170
6.3.3 Calculul câștigului de informație	172
6.3.4 Probleme în utilizarea arborilor de decizie	174
6.3.5 Inducerea șabloanelor logice.....	175
6.3.6 Învățarea Bayesiană.....	177
6.3.7 Învățarea prin încurajare.....	177
6.3.8 Învățarea pasivă într-un domeniu cunoscut de stări.....	178
6.3.9 Învățarea pasivă într-un domeniu necunoscut de stări.....	179
6.3.10 Învățarea activă.....	179
6.3.11 Algoritmii genetici.....	180
6.3.12 Învățarea bazată pe cunoștințe.....	180
6.3.13 Învățarea bazată pe explicații	181
6.3.14 Învățarea bazată pe relevanță	182
6.3.15 Învățarea inductivă bazată pe cunoștințe	182
6.3.16 Dezavantajele învățării supervizate.....	183
6.4. Învățarea nesupervizată	184
6.4.1 Identificarea claselor de instanțe	184
6.4.2 Structura cunoștințelor taxonomice.....	186
6.5. Concluzii	188
Bibliografie	191

Capitolul 1

Introducere

Când un venerabil și respectat om de știință spune că un lucru este posibil, sigur are dreptate. Când un venerabil și respectat om de știință spune că un lucru este imposibil, sigur se înșeală.

Arthur C. Clarke

1.1 Definiția Inteligenței Artificiale și un pic de istorie

Luată din Dicționarul Explicativ al Limbii Române (DEX), definiția inteligenței pune în evidență capacitatea individului de a se adapta și de a rezolva situații noi **pe baza experienței acumulate anterior**. Intuiția lexicografului a devansat cercetările de IA – care doar recent au pus pe primul plan cunoașterea, deplasând centrul de greutate al definiției IA către **cunoașterea aplicată**.

Definiția Inteligenței Artificiale este dată de Barr&Feigenbaum (Barr, Feigenbaum, 1981): *IA este știința, parte a informaticii, care proiectează sisteme artificiale cu comportament inteligent – adică sisteme ce manifestă proprietăți pe care în mod obișnuit le asociem cu existența inteligenței în comportamentul uman - înțelegerea limbajului, învățare, raționament, rezolvarea problemelor ș.a.m.d.* Astfel de sisteme pot răspunde flexibil în situații ce nu au fost anticipate de programator.

Precursor al domeniului este considerat Norbert Wiener, care, bazat pe tripla formație universitară în matematică, zoologie și filosofie, reușește să găsească în cartea sa *Cibernetica sau control și comunicație la om și mașină*. Sintagma *Inteligență Artificială* este pronunțată însă pentru prima oară în 1956 la conferința de la Dartmouth, Canada.

1.2 Testul Turing

Introducerea unui arbitru uman într-o dispută având ca scop definirea inteligenței mașinii pare inevitabilă. Dificultăți de natură filosofică fac imposibilă găsirea unei definiții riguroase a inteligenței. Odată ce acceptăm un amestec uman în această dispută înseamnă că cedăm empiricului, partizanului, subiectivului, adică în fapt recunoaștem imposibilitatea de standardizare într-o chestiune care n-ar fi existat dacă noi n-am fi existat. Suntem capabili a marca inteligența umană pe o scară Q, dar cum putem face distincția dintre inteligența umană și cea a mașinii?

În testul Turing un subiect uman și o mașină sunt așezate în camere distincte de aceea a unui alt personaj uman, numit arbitru. Arbitrul comunică cu subiectul uman și cu mașina prin intermediul unei tastaturi și trebuie, prin întrebări adresate celor doi, să decidă care din ei e omul și care e mașina.

În dilema identificării inteligenței într-o mașină, vom constata că manifestăm întotdeauna o rețineră în a considera că o mașină e înzestrată cu inteligență, chiar dacă răspunsurile ei sunt cele pe care le așteptăm de la un interlocutor uman, pentru că vom fi nesiguri dacă interfața spectaculoasă este susținută de o trăire psihică adecvată ori este doar o simulare. Un obstacol serios în definirea gradului în

care o mașină poate fi considerată inteligentă, va rămâne diferența dintre comportament, ce poate fi obiectivat, și acel ceva adânc, personal și indefinibil care provoacă comportamentul.

Plasată în condiții de test diverse, până la urmă inteligența poate fi decantată de fals. Apoi IA este un domeniu foarte adânc ancorat într-o realitate pragmatică. Ceea ce interesează în fapt este ca mașina să fie capabilă să înlocuiască niște sarcini care în mod normal presupuneau prezența omului. Aceste sarcini sunt adesea foarte bine precizate de un cadru al aplicației și, deocamdată, nu se cere mașinii un comportament universal inteligent.

1.3 Probleme de natură filosofică și pragmatică

Testul Turing – pentru prima oară publicat în revista *Mind*, în 1950. Există o competiție anuală (<http://www.cogsci.ucsd.edu/~asaygin/tt/test.html>). Dificultatea principală constă în încorporarea cunoașterii de bun-simț (*common-sense knowledge*) în mașină.

Dar testul Turing este contestat de anumiți teoreticieni ai domeniului ca fiind relevant pentru a defini inteligența mașinii. Ei argumentează că o mașină care ar trece acest test nu ar manifesta inteligență în aceeași manieră în care un om ar manifesta-o. Testul Turing nu dă decât o privire superficială asupra inteligenței, utilizând un criteriu distinctiv pur pragmatic, pe când adevărata inteligență e de natură adânc semantică. O mașină care ar trece testul Turing se spune că ar manifesta inteligență artificială slabă (*weak AI*).

Penrose (Penrose, 1998) nu crede că inteligență adevărată poate fi cu adevărat prezentă fără conștiință și de aici că inteligența nu va putea fi niciodată produsă de nici un algoritm ce e executat pe o mașină. El enunță patru puncte de vedere ce pot fi identificate relativ la problema realizării inteligenței pe o mașină.

Punctul de vedere **mecanicist**, sau al **inteligenței artificiale tari**, confundă gândirea cu calculul. Mai mult decât gândirea, ca manifestare logică a unui creier, toate trăirile care implică conștiința reprezintă rezultate ale unor calcule. La una din extremele acestui punct de vedere întregul univers e privit ca un imens calculator, mintea omenească reprezentând doar una din componentele acestui proces de calcul. Faptul că însăși substanța pare a se raporta din punct de vedere computațional la energie (conform formulei lui Einstein $E=mc^2$) întărește acest punct de vedere după care fenomenele ce se desfășoară în celulele creierului pot fi gândite matematic, deci simulate pe un alt mediu computațional decât cel original. Cum toate manifestările interne ale unui creier ar putea fi reproduse pe o mașină care efectuează calcule, atunci ar trebui să acceptăm că până și manifestările interne – conștiința însăși – ar putea fi redată pe o mașină. În esență dispare diferența dintre conștiință și simularea ei, tot ceea ce se poate petrece pe o mașină performantă este adevăr în aceeași măsură în care adevăr reprezintă trăirile noastre adânci, sentimentele, bucuriile și durerile noastre, ideile pe care le avem ori înțelegerea pe care credem că o avem față de o operă de artă sau o bucată muzicală. În conformitate cu acest punct de vedere testul Turing este absolut relevant în depistarea inteligenței artificiale, dacă o mașină răspunde adecvat la întrebări înseamnă că și înțelege acele întrebări și răspunsurile ei sunt în rezonanță cu procese mentale de profunzime pe care aceste întrebări le inspiră.

Punctul de vedere al **inteligenței artificiale slabe** acceptă din nou conștiința ca rezultat al acțiunilor fizice ale creierului și faptul că orice acțiune fizică poate fi reprodusă computațional, dar simularea computațională nu trezește sau nu presupune și conștiință. Spre deosebire de punctul de vedere mecanicist un robot care trece testul Turing în mare măsură simulează înțelegerea fără ca aceasta să aibă cu adevărat loc, la nivelul reacțiilor o persoană și o mașină se manifestă la fel, dar mașina simulează conștiința, o raportează doar, pe când o ființă o experimentează. Simularea computațională a unui proces este un lucru complet diferit de procesul însuși (Searle, 1992). Searle (Searle, 1980) imaginează un experiment capabil să demonstreze că un calculator nu are aceeași înțelegere ca un om: conceptul de *cameră chinezească*. În acest experiment se presupune existența unui calculator supersofisticat capabil să „înțeleagă” (în orice accepțiune posibilă a acestui termen) un text în chineză și să răspundă la întrebări legate de acesta. Textul și apoi întrebările sunt prezentate mașinii ca șiruri de caractere chinezești pe cartonașe, iar răspunsurile acesteia sunt tipărite și prezentate în aceeași manieră. În faza a doua a imaginarului experiment, intervine un individ superdotat, programator, care nu știe o iotă chinezește. Lui i se prezintă programul mașinii și i se cere să-l aplice pentru a obține, el însuși, același rezultat. După mai mult timp, în care acesta deslușește programul mașinii, el va fi capabil să obțină aceleași rezultate ca și mașina. Problema este așadar următoarea: faptul că individul uman este acum capabil să răspundă corect unor întrebări puse într-o limbă pe care el nu o vorbește, relative la un text prezentat în aceeași limbă, ne poate face să spunem că el „a înțeles” acel text și acele întrebări? E posibil să credem că atât mașina cât și individul din experimentul lui Searle ar fi trecut testul Turing (în sensul că, plasat în locul calculatorului și deci întrecându-se cu un individ care știe chinezește, arbitru nu l-ar fi putut demasca). Și totuși accepțiunea lui „a înțelege chinezește” este evident una particulară aici. În enunțul experimentului său Searle reduce problematica la prelucrări dibace de șiruri de caractere chinezești. Dintr-un anumit punct de vedere, care se concentrează strict asupra unei perechi de cartonașe intrare-ieșire, într-adevăr avem de a face cu prelucrări de caractere. Conștient sau nu, în formularea experimentului *camerei chinezești* Searle nu insistă asupra faptului că pentru a obține performanțele de prelucrare descrise programul trebuie să se ridice de la date prezentate sub formă de șiruri de caractere la concepte. Doar transformarea șirului de caractere chinezești din intrare în concepte (care sunt independente de o limbă sau alta), urmat de capacitatea de a opera cu acestea și de a transpune din nou conceptele în șiruri de caractere prezentate în ieșire, ar fi putut face posibilă obținerea unui rezultat de acest fel. Numai că această ridicare de la nivelul de prelucrare al șirurilor de caractere la cel al unei prelucrări de tip conceptual nu schimbă cu nimic concluzia. Pentru că în definitiv și conceptele pot fi etichetate cu șiruri de caractere.

Punctul de vedere **realist** (îmbrățișat de Penrose, ca și de mine) nu acceptă nici măcar posibilitatea unei reproduceri realiste a conștiinței. Mai devreme sau mai târziu, dacă dialogul continuă, mașina trebuie să se dea de gol. Efectele externe ale conștiinței nu pot fi simulate corespunzător. Există manifestări ale conștiinței care nu pot fi explicate prin legile actualmente cunoscute ale fizicii și, ca urmare, nu pot fi reproduse computațional. Nu este nici un fel de fetișism în acest punct de vedere. Pentru mai multe detalii cititorul este îndemnat să consulte cartea lui Penrose.

În sfârșit, punctul de vedere **mistic** este cel conform căruia conștiința nu poate fi explicată în termeni fizici, știința e neputincioasă în tentativa de a explica conștiința. Cu toată aparenta identitate dintre aceste două ultime puncte de vedere, există totuși o mare diferență între ele: ultimul neagă a-priori capacitatea științei de a cunoaște conștiința, pe când cel realist afirmă că problema conștiinței conștiente este una științifică și doar incapacitatea noastră momentană nu ne permite să o pătrundem deplin.

În ciuda dificultăților de a identifica inteligența artificială cu cea naturală, există puternice motivații de natură pragmatică pentru realizarea de sisteme cu comportament inteligent sau apropiat de cel inteligent. Așadar, mai puțin aplecat asupra detaliilor demonstrațiilor filosofice, domeniul IA este complet absorbit de practică. Acest lucru se observă cu predilecție în apelul la proiecte de cercetare în domeniul tehnologiei informației (*Information and Science Technology - IST*) din cadrul programului cadru 6 (*framework program 6 – FP6*) al Comisiei Europene, lansat în decembrie 2002 (v. situl web <http://www.cordis.lu>).

1.4 Subdomenii ale inteligenței artificiale

Prelucrările simbolice (*symbolic processing*) reprezintă componenta cu preocupări de programare a domeniului. Ea a dus la dezvoltarea unor limbaje specializate în prelucrarea simbolurilor, cel mai cunoscut dintre ele fiind Lisp. Fără a neglija aspectele clasice ale programării, cum ar fi utilizarea numerelor în calcule, în toate aceste limbaje se pune un accent deosebit pe manipularea simbolurilor. Un limbaj adecvat prelucrărilor simbolice găzduiește cu ușurință reprezentarea și manipularea simbolurilor de orice natură, nu numai numerice. Pentru că inteligența artificială se bazează în mare măsură pe manipularea simbolurilor, Lisp este considerat limbajul de casă (sau nativ) al inteligenței artificiale. Un tip particular de prelucrări simbolice sunt **confruntările de șabloane** (*pattern matching*). Șabloanele sunt obiecte abstracte formate din părți definite riguros și altele doar schițate. Șabloanele sunt utilizate pentru regăsirea de obiecte ce satisfac anumite constrângeri în mulțimi de obiecte asemănătoare.

Procesare simbolică și limbaje de procesare simbolică. În general se acceptă că sistemele inteligente din natură prelucrează două tipuri de informații: de natură simbolică sau concentrată și de natură difuză, distribuită. Calculatoarele noastre, prin proiectare, sunt destinate a manipula numere și caractere, având compartimente special proiectate pentru realizarea performantă de calcule numerice. Lucrul cu simboluri (șiruri de caractere, imagini sau sunete) deși realizabil pe calculatoarele clasice, necesită tehnici speciale mulate pe natura digitală a aparatului de calcul. Mașina uzuală pe care o avem la dispoziție nu are cablate disponibilități de a opera cu simboluri, de a realiza regăsiri simbolice pe bază de asociații, ori de a forma alți simboluri prin compunere. Manipularea simbolurilor necesită un aparat logic și un suport adecvat de calcul (o mașinărie). Aparatul logic este dat de logica simbolică care reprezintă atât un model de formalizare a cunoștințelor cât și o metodă de raționament. Mașinăria este asigurată de limbajele de procesare simbolică capabile a simula un comportament adecvat prelucrărilor simbolice mulat pe natura slab simbolică a mașinii. Dintre acestea, putem identifica:

Lisp – în paradigma funcțională, Prolog – în paradigma logică și Clips – în paradigma bazată pe reguli¹.

Reprezentarea cunoașterii (*knowledge representation*). Putem interacționa cu mediul înconjurător pentru că, pe de o parte, datorită simțurilor, intrăm în legătură cu ea, iar pe de altă parte o "înțelegem" așa cum este. Ca să înțelegem realitatea avem însă nevoie de o "proiecție" în mintea noastră. Această proiecție este o reprezentare a realității. Fără a avea o reprezentare asupra unei entități nu putem emite judecăți asupra ei. Reprezentarea implică, în principiu, trei componente: o notație, o denotație și un calcul. Notația este un desen, sau o structură de date, care respectă anumite convenții (date de obicei de o sintaxă). Conotația, sau semantica, este interpretarea pe care o dăm notației, din nou pe baza unui sistem de convenții. Fără aceste reguli de interpretare notația este superfluă. În sfârșit o reprezentare trebuie să includă un model computațional care să facă posibilă operarea cu obiectele și relațiile dintre obiecte ce compun reprezentarea. Datorită modelului de calcul entitățile modelate prind viață, putându-se studia astfel comportamentul lor în condiții ce simulează de aproape realitatea.

Raționament automat și demonstrarea teoremelor (*theorem proving, problem solving*). Cea mai veche ramură a IA și cea mai teoretică dintre ele, utilizează logica ca sistem formal de găsim a demonstrațiilor și de generare a inferențelor automate. Foarte multe probleme pot fi formalizate ca probleme de matematică sau logică, rezolvarea lor reducându-se la o demonstrare a unei teoreme plecând de la un sistem de axiome și utilizând un mecanism de deducție logică. Ca și în alte cazuri, dificultățile rezidă în numărul extrem de mare de posibilități ce pot apare. Pentru controlul exploziei de posibilități în căutarea unei soluții se utilizează de multe ori soluții euristice, care deși nu garantează soluția optimă, pot în general duce la găsirea uneia, cel puțin.

Înțelegerea limbajului natural (*natural language processing*). Este de mult acceptat că utilizarea limbajului este o caracteristică definitorie a inteligenței. Deși nouă ne vine atât de ușor să comunicăm prin limbaj, o încercare de explicare a mecanismelor care stau la baza înțelegerii textelor ori a limbajului vorbit se dovedește deosebit de dificilă. În domeniul înțelegerii limbajului natural trebuie întii stabilită o distincție între a înțelege un mesaj comunicat prin voce și a înțelege un mesaj scris. De prima chestiune se ocupă domeniul de cercetare al interpretării vorbirii (*speech processing*). Problema înțelegerii limbajului – în varianta scrisă – face obiectul a două tipuri de preocupări. *Lingvistica computațională* pe de o parte, ca domeniu pur academic, pune în discuție modele computaționale ale limbajului natural cu scopul de a investiga și descoperi natura însuși a limbajului și a abilităților cognitive umane. *Ingineria Lingvistică* (sau în varianta predominant americană, *Tehnologia Limbajului Uman*), pe de altă parte, este preocupată de a dezvolta

¹ Paradigmele de programare sunt stiluri ori precepte generale de programare transpuse în caracteristicile unor limbaje. Astfel în cadrul paradigmei imperative se clasifică limbajele bazate pe comenzi, paradigma funcțională pune la baza programării noțiunea matematică de funcție ideea de bază fiind că un program înseamnă un apel de funcție, paradigma orientată obiect aduce în prim plan noțiunea de obiect înzestrat cu proprietăți și metode, paradigma logică realizează un program ca o demonstrație de teoremă, în paradigma bazată pe reguli un program reprezintă o colecție de reguli și un motor aplică aceste reguli într-o anumită ordine.

aplicații care se bazează pe utilizarea limbajului natural, orientate spre industrie, comerț, sfera socialului sau cea educațională².

Vederea artificială (*vision*). Cinci simțuri³ (vederea, auzul, simțul tactil, mirosul și gustul) ne stau la dispoziție pentru a interacționa cu mediul, pentru a lua cunoștință de ceea ce se află în afara corpului nostru, în imediata noastră vecinătate. Pentru a ne prelungi "bătăia" acestor simțuri am inventat aparate care ne ajută să vedem ori să auzim la distanță. Modelarea acestor simțuri pe sisteme automate reprezintă cu certitudine, o preocupare de mare însemnătate a domeniului IA. Dintre cele cinci, vederea ocupă un loc privilegiat. Sistemele de vedere artificială încearcă să descifreze mecanismele vederii și ale interpretării imaginilor statice și în mișcare.

Planificare și robotică (*robotics*). Sistemele inteligente vii au abilitatea de a găsi soluții de deplasare în spațiu în vederea atingerii unor obiective sau de mutare a unor obiecte folosind brațele. Căutarea unui drum pe un teren accidentat (lunar, de exemplu, sau pe fundul oceanelor) sau într-un spațiu presărat cu obstacole (cum este o cameră) nu este o problemă simplă, nu în ultimul rând datorită numărului mare de posibilități de alegere la fiecare pas, complexitate ce face imposibilă tentativa căutării exhaustive. Încercând a modela procesele cognitive care se desfășoară în sistemele vii, planificarea roboților este un câmp de cercetare care urmărește a îmbunătăți comportamentul roboților atunci când ei sunt programați a executa diferite sarcini.

Învățare automată (*learning*). Un sistem înzestrat cu inteligență este capabil să învețe pentru a-și îmbunătăți interacțiunea cu mediul. El poate învăța fie din greșeli, prin auto-perfecționare, fie ghidat de un profesor, fie generalizând, fie prin analogie, sau din exemple pozitive și negative. Domeniul învățării automate formalizează aceste metode și caută aplicarea lor la sistemele automate.

Sisteme expert (*expert systems*). Puterea stă în cunoaștere. Aplicarea acestui truism în sisteme artificiale înseamnă dotarea lor cu abilitatea de a se servi de cunoaștere specifică (cunoaștere expert). Un medic este bun în găsirea unui diagnostic pentru că are un bagaj de cunoștințe generale dar și specifice despre boli și bolnavi. Parțial această cunoaștere a deprins-o din cărți, parțial în cursul anilor de experiență clinică, prin atâtea cazuri în care s-a implicat. Achiziționarea, formalizarea și includerea cunoașterii expert în sistemele artificiale reprezintă scopul domeniului Sistemelor Expert. Cursul nostru va dezbate cu precădere probleme de această natură.

² Definițiile sunt preluate dintr-un recent mesaj difuzat pe lista corpora@hd.uib.no de Prof. Geoffrey Sampson, School of Cognitive & Computing Sciences, University of Sussex, care la rândul său citează un articol publicat într-un număr recent al revistei *Natural Language Engineering*.

³ Cercetări noi lasă speranța că vom învăța să ne servim de încă alte simțuri din sfera para-normalului, ce la majoritatea dintre noi sunt de utilitate subliminală, dar pentru care alții manifestă o dotare aparte.

1.5 Probleme și soluții IA față de probleme și soluții de programare clasică

1.5.1 Prin ce diferă o problemă clasică de una de IA ?

Există o seamă de trăsături care diferențiază câmpul IA de alte domenii ale informaticii (*computer science*). Cursul de față încearcă să convingă cititorul că măcar următoarele trăsături pot fi considerate definitorii:

- Ce face ca o problemă să fie considerată de IA, iar o alta nu?
 - problemele de IA necesită în general un raționament predominant simbolic;
 - probleme care îmbie la investigații IA sunt și cele care manipulează informație incompletă ori nesigură;
- Ce face ca o soluție să fie considerată ca fiind caracteristică metodelor IA?
 - de cele mai multe ori nu este de natură algoritmică. Adesea, ea este obținută în urma unei căutări într-un spațiu al soluțiilor posibile;
 - soluțiile ce se cer nu trebuie cu necesitate să fie cele mai bune sau exacte. Uneori e suficient dacă o soluție este găsită, sau dacă o formulare aproximativă a ei este obținută;
 - în rezolvarea problemelor de IA adesea intervin cantități foarte mari de informații specifice. Găsirea unui diagnostic medical nu poate fi algoritmatizat pentru că diferențele de date asupra pacientului incumbă tipuri însuși de soluții diferite. De aceea doctorii spun că nu există boli și doar pacienți;
 - natura cunoașterii ce se manipulează în problemele de IA poate fi ușor departajată în procedurală și declarativă: este diferența dintre cunoașterea pe care o posedă cu păianjen și un inginer constructor de poduri, sau diferența dintre cunoașterea pe care o posedă un jucător de tenis și un bun antrenor. Cunoașterea unuia se află 'în vârful degetelor', a celuilalt într-un sistem de reguli.
- Pot probleme de IA fi rezolvate prin algoritmi clasici ?
 - Cu siguranță că da, dar soluțiile vor fi probabil greoaie, greu generalizabile, nefirești și neelegante.
- Invers: pot probleme clasice fi rezolvate prin metode ale IA ?
 - Iarăși lucrul este posibil, dar e ca și cum am folosi un strung ca să ascuțim un creion.

De ce un program de calcul al salariilor nu este un program de IA pe când a face un robot să măture prin casă fără a distruge mobila, sau a face un program să recunoască figuri umane este o aplicație de IA?

Distincția dintre cunoașterea procedurală și cea declarativă. Trăsături ale cunoașterii declarative:

- poate fi schimbată mai rapid și mai ușor pentru că este de natură discretă;
- poate fi folosită la mai multe scopuri decât la cel imaginat inițial;

- poate fi extinsă prin procese de raționament care adaugă cunoaștere nouă;
- poate fi accesată de programe de introspecție care să ofere explicații la cerere.

Nivelul actual al cercetărilor și realizărilor în IA

Exemple de direcții de cercetare particulare în IA:

- sisteme inteligente pentru biologie moleculară (până în 1997 – s-au ținut 5 conferințe internaționale pe această temă, începând din 93);
- roboți inteligenți – competiții de roboți inteligenți (primul ținut la San Jose, California, în 1992, în 1996 – a 5-a ediție, în Portland, Oregon).
- sisteme expert (medicină, chimie, geologie, configurarea sistemelor de calcul, descoperirea legilor fizicii);
- raționament automat, demonstrarea de teoreme;
- verificarea programelor;
- matematică simbolică (calcul diferențial, integral, manipularea expresiilor);
- jocuri (șah, table, *checkers*)
- interfețe în limbaj natural la baze de date și de cunoștințe;
- sisteme de întrebare-răspuns pe domenii limitate sau largi;
- sisteme de traducere automată;
- senzori inteligenți;
- controlul roboților staționari și mobili (linii de asamblare, roboți subacvatici sau spațiali).

Cerințe pentru studenți

- Să cunoască semnificația și detaliile testului Turing.
- Să cunoască definiția domeniului IA și cu ce se ocupă subdomeniile sale.
- Să poată distinge o problemă de IA de una clasică.

Bibliografie

Barr, A., Feigenbaum, E. 1981. *The handbook of Artificial Intelligence*. William Kaufmann, Inc. Se poate consulta la Biblioteca Seminarului Matematic de la Facultatea de Matematică a Universității "Al.I.Cuza".

Penrose, R. 1989. *The Emperor's New Mind: Concerning Computers, Minds and the Laws of Physics*, Oxford University Press, New York. Tradusă în limba română de Cornelia C. Rusu și Mircea V. Rusu ca "Mintea noastră... cea de toate zilele: Despre gândire, fizică și calculatoare", Ed. Tehnică, București 1996.

Penrose, R. 1994. *Shadows of the Mind*, Oxford University Press. Tradusă în limba română de Dana Jalobeanu ca "Incertitudinile rațiunii (Umbrele minții). În căutarea unei teorii științifice a conștiinței", Ed. Tehnică, București, 1999.

Searle, J. R. 1980. *Minds, brains and programs*. În „The behaviour and brain sciences”, vol. 3, Cambridge University Press.

Searle, J. R. 1992. *The rediscovery of the mind*. MIT Press, Cambridge University Press, Massachusetts.

Capitolul 2

LISP

Mintea funcționează ca o parașută: e utilă doar când e deschisă.

Lord Thomas Dewar

2.1 Introducere

Intenția acestui capitol nu este aceea de a da o descriere exhaustivă a limbajului LISP ci de a introduce noțiunile esențiale ale lui, care să facă posibilă scrierea de aplicații simple la capitolele de inteligență artificială ce urmează. Nu vrem ca prezentarea să urmărească cu necesitate un anumit dialect de Lisp. Toate exemplele noastre vor putea însă fi rulate direct în Allegro Common Lisp, versiunea de Common Lisp creată de Franz Inc. (www.franz.com). Common Lisp (Steele, 1990) este dialectul pe care l-a dezvoltat Comisia de standardizare a Lisp-ului X3J13 din cadrul ANSI (*American National Standard Institute*). Datorită flexibilității deosebite a limbajului, care parcă invită la implementări aventuroase, procesul de standardizare a fost unul de durată, dar după anul 1990 a fost finalizat în varianta care este acceptată astăzi de cele mai multe implementări Common Lisp.

Lisp – limbajul proiectat special pentru problema pe care o am de rezolvat

Dacă ar fi să inventăm o reclamă pentru promovarea limbajului Lisp, ar trebui, probabil, să ne exprimăm astfel:

- Lisp este limbajul care găzduiește, cu simplitate și eleganță, conceptul de prelucrare a datelor cunoscut sub numele de **prelucrare simbolică**. Fără a neglija calculul numeric, limbajele simbolice au facilități speciale de a lucra cu simboluri nenumere.
- Lisp este unul dintre limbajele care exemplifică paradigma de **programare funcțională**⁴. O funcție este un obiect matematic care întoarce o unică valoare pentru un set de valori date în intrare. Toate construcțiile unui limbaj funcțional pur sunt funcții care întorc valori atunci când sunt evaluate. Lisp, ca și multe dintre suratele lui funcționale, a acceptat însă și efecte laterale în evaluarea funcțiilor, care deși l-au impurificat i-au mărit expresivitatea.
- Lisp este **limbajul care se mulează pe problema pe care o aveți de rezolvat**. Această afirmație trebuie înțeleasă în sensul ergonomiei excepționale pe care o oferă limbajul actului de programare. Programarea în Lisp este, pentru un cunoscător, o plăcere, un spectacol și o invitație la creație. Programarea în Lisp este ușoară iar productivitatea limbajului este remarcabilă. Lispul este adesea mai concis decât alte limbaje.

⁴ Alături de Erlang – pentru aplicații concurente, R – pentru prelucrări statistice, Mathematica – pentru matematică simbolică, APL – cunoscut mai ales pentru facilitățile de a opera cu matrici, extensia ulterioară a acestuia J, K – pentru analiză financiară, XSLT – pentru transformarea documentelor XML, cât și limbajele noi ML (al lui Robin Milner), cât și Miranda (David Turner) și urmașul acestuia Haskell (dezvoltat de Haskell Curry).

- Lisp este **limbajul care se dezvoltă pe măsură ce rezolvați problema**. Această trăsătură provine din utilizarea macrourilor – secvențe de cod ce suportă execuții specială. Prin macrouri se poate da nu numai o nouă interpretare noțiunii de evaluare a formelor limbajului, ci se poate schimba însăși sintaxa. Se pot crea în acest fel linii de cod care nu mai seamănă deloc cu sintaxa obișnuită a limbajului.
- Lisp este un **limbaj specializat pentru prelucrarea listelor**, ceea ce se reflectă în chiar numele lui (LISt Processing). Motivul pentru care lista, o structură de date relativ nespectaculoasă, poate sta la baza unui limbaj dedicat dezvoltării de aplicații într-un domeniu atât de pretențios precum inteligența artificială este că această structură este extrem de generală, o listă putând înlocui nu numai simboluri precum numere și cuvinte ci și alte liste, oferind posibilitatea reprezentării de o manieră uniformă a unor structuri oricât de complicate.
- Lisp este **limbajul „de casă” al inteligenței artificiale**. Caracteristicile ce-i conferă această calitate sunt: facilitatea de a lucra cu simboluri, lista – ca structură fundamentală de date, mecanismele complexe de evaluare și utilizare a macrourilor, care pot duce la construcții procesuale sofisticate și care se comportă diferit în funcție de context.

Lisp facilitează o abordare multi-strat în programare. Implementările orientate-obiect ale acestui limbaj îi conferă toate trăsăturile cunoscute ale paradigmei. În mod particular, programarea multi-strat înseamnă construcția de straturi de definiții funcționale, ceea ce invită la o abordare *bottom-up* în rezolvarea problemelor, în așa fel încât apelurile de la un nivel superior să încorporeze definiții de funcții și construcții de pe un nivel inferior.

Programarea multi-strat este cu deosebire o abordare care se pretează dezvoltării proiectelor mari, care necesită lucru în echipă, uneori conlucrarea mai multor echipe. Așa cum se sugerează în Figura 2.1, într-o abordare de jos în sus, se pleacă de la un nivel de bază pentru a se construi deasupra lui un număr de niveluri derivate. Pe nivelul de bază se află diferite componente ale problemei care trebuie rezolvată într-un anumit limbaj de programare. Prin utilizarea posibilităților oferite de limbaj se creează deasupra acestuia un prim nivel de instrumente ori funcții, capabile să facă față cerințelor problemei și, eventual, chiar să le depășească. Acest nivel poate fi văzut ca un alt limbaj, superior celui aflat la bază. În același mod, nivelurile superioare se creează deasupra celor imediat inferioare, până la atingerea specificațiilor proiectate.

În particular, Lisp-ul invită nu numai la crearea unor limbaje de nivel intermediar bazate pe definiții de funcții și în care construcțiile să rezulte prin manipularea apelurilor, ci inclusiv la inventarea unor sintaxe speciale care să se integreze cerințelor nivelurile de proiectare respective.

Invers, într-o abordare de sus în jos, se pleacă de la o proiectare inițială a soluției. Această soluție, definită în termeni generali, face apoi obiectul unor rafinamente succesive, până la rezolvarea efectivă a problemei. Dificultatea într-o astfel de abordare constă în centrarea proiectului inițial pe soluție fără a se scăpa din vedere aspecte ce ar putea să o facă neaplicabilă pe unele cazuri speciale – „capetele problemei”. Într-o astfel de deplasare nefericită nu deranjează acoperirea

unor aspecte neavute în vedere inițial ci neacoperirea altora care sunt esențiale în rezolvarea problemei.

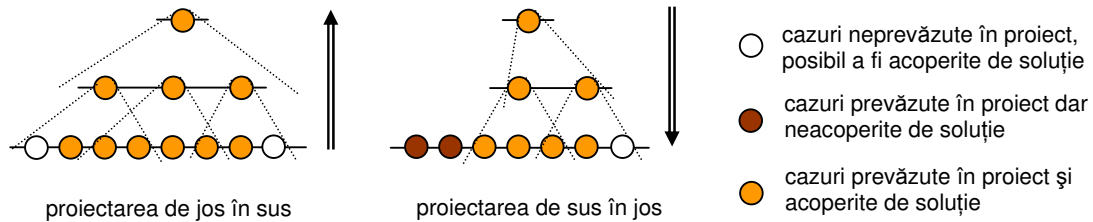


Figura 2.1: Crearea programelor bottom-up față de top-down

2.2 Funcții, maniere de definire și apel

2.2.1 Transparența referențială

Spunem că o funcție este transparentă referențial dacă valoarea întoarsă de ea depinde numai de parametrii pe care i-a primit în intrare. Astfel, dacă scriem în limbajul C:

```
function fool(x)
{ return(x+2);
}
```

valoarea întoarsă de funcția `fool` depinde, la orice apel al ei, numai de valoarea intrării `x`. Dacă însă o transformăm în:

```
int a=2;

int function foo2(int x)
{ return(x+a);
}
```

atunci valoarea întoarsă de `foo2` la un apel al acesteia de forma `foo2(3)`, de exemplu, depinde nu numai de valoarea intrării `x`, care este 3 în cazul de față, ci și de a variabilei globale `a` (2 în cazul de față). Spunem deci că `fool` este transparentă referențial în timp ce `foo2` – nu.

2.2.2 Efectul lateral

Se spune că o funcție are un efect lateral dacă evaluarea ei adaugă contextului și alte modificări decât strict valoarea întoarsă de funcție. Astfel, funcția `fool` definită mai sus nu are efecte laterale, în timp ce funcția `foo3` de mai jos, are:

```
int a;

function foo3(int x)
{ a := x+2;
```

```
    return(a);
}
```

pentru că, în afara valorii întoarse, ea mai provoacă asignarea unei valori variabilei globale *a*.

2.2.3 De la notația *lambda* la Lisp pur

Lisp este unul dintre cele mai vechi limbaje de programare aflate încă în largă utilizare (numai FORTRAN este mai vechi decât el). Primele specificații ale limbajului au fost elaborate de John McCarthy, în 1958 și publicate în 1960 (McCarthy, 1960), pe baza calculului- λ dezvoltat de Alonso Church (Church, 1941) și a unor dezvoltări precedente ce au aparținut în principal lui Herbert Simon și Allen Newell (*Logic Theory Machine* și *General Problem Solver*, programe dezvoltate în limbajul IPL inventat de Newell).

În notația *lambda*, într-o definiție de funcție se pun în evidență parametrii formali ai funcției și corpul ei:

$$\lambda(x) = x+2;$$

sau:

$$\lambda(x, y) = x+y;$$

Într-o astfel de notație funcțiile nu au nume. Asocierea unui nume funcțiilor, în vederea creării posibilității de apel al lor, trebuie făcută explicit:

```
function f:  $\lambda(x) = x+2;$ 
function g:  $\lambda(x, y) = x+y;$ 
```

Într-un apel de funcție trebuie să apară numele ei urmat de o listă de parametri actuali:

```
f(3)
g(5, 1)
```

Un apel poate, în anumite condiții, să amorseze un proces de evaluare care să ducă la generarea unui rezultat. În evaluare se disting două faze: prima în care parametrii formali ai funcției sunt legați la valorile corespunzătoare ale parametrilor actuali din corpul definiției funcției și a doua în care are loc evaluarea expresiei astfel obținute a corpului funcției. În evaluarea corpului pot să intervină alte apeluri de funcție, care se derulează în aceeași manieră. Astfel se pot apela funcții în funcții.

Spunem că un apel poate, în anumite condiții, porni un proces de evaluare, pentru că o scriere precum *f(3)* nu garantează efectuarea evaluării. Ea nu este decât o simplă notație până ce se comandă explicit unui evaluator (uman sau mașină) începerea unui astfel de proces:


```
g (f (3), 1) >>> g (5, 1) >>> 6
```

unde >>> trebuie citit „se evaluează la”.

Între caracteristicile esențiale care definesc așa-numitul **Lisp pur** se numără faptul că funcțiile sunt transparente referențial, că funcțiile nu au efecte laterale și că din limbaj lipsesc asignările.

2.2.4 Evaluarea numerică față de evaluarea simbolică

Fie funcția f definită mai sus. Apelată asupra parametrului 3 avem o evaluare numerică:

```
f (3) >>> 3 + 2 >>> 5
```

Dacă însă parametrul actual care ia locul parametrului formal x din corpul definiției funcției este unul nenumeric, să zicem a , atunci avem de a face cu o evaluare simbolică:

```
f (a) >>> a + 2
```

pentru că forma rezultată conține un simbol nenumeric.

2.2.5 Beneficiile unui Lisp impur: Common Lisp

Pe considerente de creștere a puterii de expresie a limbajului și de mărire a productivității activității de programare, Lisp-ul pur a fost ulterior impurificat în absolut toate direcțiile posibile, fără însă a pierde, prin aceasta, sclipirea de geniu originală. Astfel, după cum vom constata în cele ce urmează, s-au acceptat definiții de funcții care să nu mai respecte trăsătura de transparentă referențială (fără a le face însă astfel opace referențial...), s-au încurajat efectele laterale, s-au adoptat câteva operații de asignare, ba chiar s-a permis ca o funcție să întoarcă mai mult decât o unică valoare, această din urmă „trădare” făcându-se însă, aparent paradoxal, fără să se încalce câtuși de puțin definiția matematică a funcției (o corespondență de la un domeniu de intrare la un domeniu de ieșire în care oricărei valori de intrare îi corespunde o singură valoare de ieșire).

2.3 Tipuri de date în Lisp

Schema următoare face o clasificare a celor mai uzuale tipuri de date în Lisp:

s-expresie (expresie simbolică – *symbolic expression*)

atom

numeric

întreg

rațional (fracție)

real

complex

nenumeric (în fapt *atom literal*, noi îi vom numi *atom simbolic* sau, simplu, *simbol*)

listă

listă simplă

listă cu punct

șir (*array*)

șir de caractere

tablou (cu oricâte dimensiuni)

tabelă de asociație (*hash table*) – structură de date care permite asocierea și regăsirea cu ușurință a valorilor atașate simbolurilor;

pachet (sau spațiu de nume) – colecții încapsulate de simboluri. Un parser Lisp recunoaște un nume prin căutarea lui în pachetul curent;

flux de date – sursă de date, tipic șir de caractere, utilizată pentru canalizarea operațiilor de intrare/ieșire.

Numai datele au tipuri. O variabilă în Lisp nu are definit un tip. Ea poate primi ca valoare orice tip de dată.

2.3.1 Construcțiile limbajului

În Lisp operăm cu următoarele categorii de **obiecte**: variabile, constante, date, funcții, macro-uri și forme speciale (la acestea mai trebuie adăugate clasele și metodele în implementările orientate-obiect).

Variabilele sunt asociații între simboluri utilizate în anumite spații lexicale ale limbajului și valori asociate acestora. După cum vom vedea mai departe, există trei moduri prin care o variabilă poate să primească valori, uneori chiar mai multe odată: prin asignare, prin legare și prin intermediul listelor de proprietăți. Două dintre aceste moduri de a primi valori (asignarea și listele de proprietăți) sunt caracteristice oricărui simbol Lisp. Ceea ce diferențiază un simbol lexical de o variabilă sunt numai situațiile în care variabilele pot fi legate la valori.

Constantele sunt simboluri (în general ale sistemului) care au atașate valori ce nu pot fi modificate.

În Lisp nu există proceduri ci numai funcții, în sensul că orice *rutină* întoarce obligatoriu și o valoare. Macro-urile sunt funcții care au un mecanism de evaluare special, în doi pași: expandarea și evaluarea propriu-zisă (prezentate în secțiunea 2.6 Macro-uri). Formele speciale sunt funcții de sistem care au, în general, o sintaxă și un comportament aparte.

2.3.2 Un pic de sintaxă

Următoarele caractere au o semnificație specială în Lisp:

(– o paranteză stângă marchează începutul unei liste;
) – o paranteză dreaptă marchează sfârșitul unei liste;
' – un apostrof, urmat de o expresie *e*, *'e*, reprezintă o scriere condensată pentru un apel (*quote e*);

; – punct-virgula marchează începutul unui comentariu. El însuși, împreună cu toate caracterele care urmează până la sfârșitul rândului, sunt ignorate;

" – între o pereche de ghilimele se include un șir de caractere;

\ – o bară oblică stânga prefixează un caracter pe care dorim să-l utilizăm în contextul respectiv ca o literă și nu cu semnificația lui uzuală. De exemplu, șirul de caractere "a\"B" este format din trei caractere: a, " și B pentru că cel de al doilea rând de ghilimele nu trebuie considerate ca închizând șirul ci ca un simplu caracter din interiorul lui;

| – între o pereche de bare verticale poate fi dat un șir de caractere speciale pe care dorim să le utilizăm într-un nume de simbol. Inclusiv caracterul bară oblică stânga (\) poate apare între o pereche de bare verticale, dar ea își păstrează semnificația de prefix. În particular, ea poate prefixa o bară verticală între o pereche de bare verticale. Astfel, șirurile:

```
|&.+\\:;|
a|&.+\\:;|b
a|&.+\\| 2)<n1>|b
```

unde prin <n1> am notat caracterul rând-nou (*new line*), pot constitui nume de simboluri;

– un diez semnalează că următorul caracter definește modul în care trebuie interpretată construcția care urmează. Cea mai importantă utilizare a diezului este de a semnală o formă funcțională, într-o secvență de genul: #'fn, unde fn este un nume sau o lambda expresie (definiție de funcție fără nume);

` – un accent invers semnalează că ceea ce urmează este un *template* care conține virgule (mai multe despre *template*-uri, în secțiunea 2.6.2 Despre apostrof-stânga). Un *template* funcționează ca un program care modifică forma unui șir de obiecte Lisp;

, – virgulele sunt utilizate în interiorul *template*-urilor pentru a semnală cazuri speciale de înlocuiri;

: – două puncte semnalează, în general, că următorul simbol trebuie privit ca un simbol constant care se evaluează la el însuși. În alte cazuri, două puncte despart numele unui pachet de numele unei variabile definite în acel pachet (de exemplu, în user1:alpha, user1 este numele unui pachet, iar alpha este numele unei variabile).

Implementările uzuale de Common Lisp sunt insensibile la forma caracterelor (minuscule sau majuscule). Intern însă, formele Lisp sunt reprezentate cu majuscule, de aceea formele rezultate în urma evaluărilor sunt redată în astfel de caractere.

Numerele întregi zecimale pot fi precedate opțional de un semn și urmate opțional de un punct zecimal (adăugarea unui zero după punct le transformă însă în numere reale). Numere întregi în alte baze au sintaxa #nnRdddddd sau #nnrdddddd, în care nn exprimă baza iar ddddddd – numărul. Bazele 2, 8 și 16 permit și o altă scriere, respectiv #b pentru #2r, #o pentru #8r, și #x pentru #16r.

Fracțiile, sau numerele raționale, sunt reprezentate ca rapoarte dintre un numărător și un numitor, adică o secvență: semn (opțional), întreg (numărător), caracterul /, întreg (numitor). Reprezentarea internă și cea tipărită este

întotdeauna a unei fracții simplificate. Dacă numărătorul e notat în altă bază decât cea zecimală, numitorul va fi interpretat în aceeași bază. O fracție simplificabilă la un întreg este convertită automat la întreg. Numitorul trebuie să fie diferit de zero. Exemple de fracții:

```
1/2
-2/3
4/6 (echivalent cu fracția 2/3)
#o243/13 (echivalent cu 163/11)
```

Numerele reale au notația obișnuită ce cuprinde punctul zecimal, semnul (opțional) și puterea (opțional). Formatul simplu sau dublu poate, de asemenea, fi specificat:

```
3.1415926535897932384d0 ; o aproximare în format dublu pentru  $\pi$ 
6.02E+23 ; numărul lui Avogadro
```

Numerele complexe sunt notate ca o secvență: `#c(<real>, <imaginar>)`, în care `<real>` și `<imaginar>` sunt numere în același format. Exemple de numere complexe:

```
#c(1 2) ; numărul 1 + 2*i
#c(0 1) ; numărul i
#c(2/3 1.3) ; convertit intern la #c(0.6666667 1.3)
```

Orice combinație de litere și cifre poate constitui un nume de **simbol**. În plus, oricare dintre următoarele caractere poate interveni într-un nume de simbol: + - * / @ \$ % ^ & _ = < > ~.⁵ Printre altele, așadar, următoarele șiruri pot fi nume de simboluri:

```
alpha
a21
1a2
*alpha* (variabilele globale sau de sistem au, în general, nume care încep și se încheie cu asterisc)
a+b
a-b+c
a/1
$13
1%2
a^b
cel_mare
a=b
a<b~c&
dudu@info.uaic.ro
```

⁵ Ultimul caracter din acest rând, punctul (.), este caracter de sfârșit de propoziție și, deci, nu trebuie considerat printre caracterele enunțate.

2.3.2 Liste și reprezentarea lor prin celule cons

În Lisp o listă se notează ca un șir de s-expresii separate prin blankuri și închise între paranteze. De exemplu:

`()` – lista vidă, notată și `nil` sau `NIL`;

`(a alpha 3)` – listă formată din 3 atomi, dintre care primii doi – nenumeriți, iar al treilea numeric;

`(alpha (a) beta (b 1) gamma (c 1 2))` – listă formată din 6 elemente: primul, al treilea și al cincilea fiind atomi nenumeriți, iar al doilea, al patrulea și al șaselea – fiind la rândul lor liste cu unu, două, respectiv trei elemente.

Tradițional, listelor li se pot asocia notații grafice care își au obârșia în prima implementare a Lisp-ului, realizată pe o mașină IBM 704 la M.I.T. Astfel, o listă se notează ca o celulă (numită celulă *cons*) ce conține două jumătăți, prima conținând un pointer către primul element al listei, iar a doua – unul către restul elementelor listei. Notația grafică a listei ca celulă *cons* (v. Figura 2.2) mimează definiția recursivă a structurii de listă: o listă este formată dintr-un prim element și lista formată din restul elementelor. Să observăm că această definiție se poate aplica pentru liste nevide, adică liste care au cel puțin un prim element. Tot tradițional, prima jumătate a unei celule *cons*, se numește **car**, iar cea de a doua **cdr**:

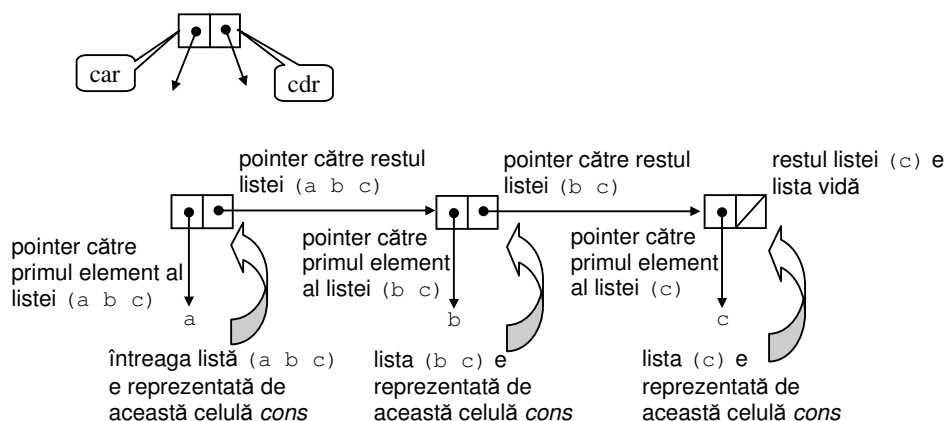
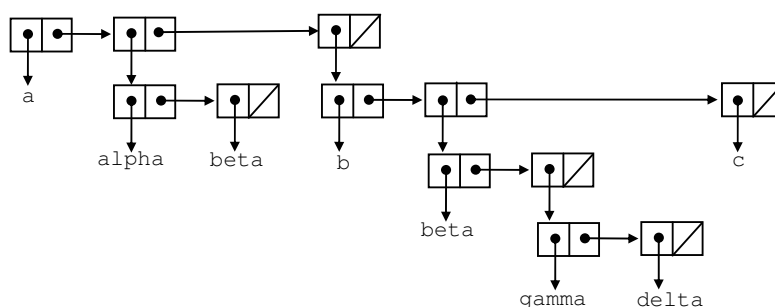


Figura 2.2: Notația grafică de listă

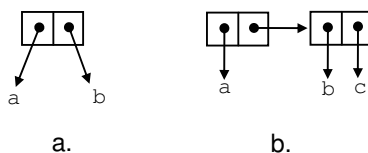
Evident, prin construcții de acest fel pot fi reprezentate liste în liste, pe oricâte niveluri. Figura 2.3 exemplifică lista:

`(a (alpha beta) (b (beta (gamma delta)) c))`

**Figura 2.3:** Un exemplu de listă

2.3.3 Perechi și liste cu punct

Celula *cons* în care atât *car*-ul cât și *cdr*-ul indică s-expresii atomice poartă numele de pereche cu punct, pentru că în notația liniară cele două elemente sunt reprezentate între paranteze, separate prin punct. Astfel celula *cons* din Figura 2.4a se notează: $(a.b)$. Dacă o listă conține perechi cu punct o vom numi **listă cu punct**. Reprezentarea din Figura 2.4b trebuie notată $(a\ b.c)$, pentru că restul listei reprezentată de prima celulă *cons* este lista cu punct $(b.c)$. Să mai observăm că orice listă simplă poate fi notată și ca o listă cu punct. Astfel, lista $(a\ b\ c)$ poate fi notată $(a.(b.(c.nil)))$. Nu este adevărat însă că orice listă cu punct poate fi notată ca o listă simplă.

**Figura 2.4:** Liste cu perechi cu punct

2.3.4 Evaluarea expresiilor

Un program Lisp este format numai din apeluri de funcții. Practic, însăși o definiție de funcție este, de fapt, tot un apel al unei funcții care creează o asociere între un nume al funcției, o listă de parametri formali și un corp al ei.

Vom presupune în cele ce urmează că expresiile Lisp sunt interpretate într-o buclă READ-EVAL-PRINT în care are loc o fază de citire a expresiei de pe fluxul de intrare, urmată de o fază de evaluare a ei și de una de tipărire a rezultatului. Vom presupune că *prompter*-ul Lisp este $>$. Orice expresie ce urmează a fi evaluată se prezintă pe un rând imediat după *prompter*, pentru ca pe rândul următor să apară rezultatul tipărit al evaluării.

Expresiile se evaluează după câteva reguli simple:

- un atom numeric se evaluează la el însuși:

$$\begin{array}{r} > 3 \\ 3 \\ > 3.14 \\ 3.14 \end{array}$$

În Lisp pot fi reprezentate numere oricât de mari fără riscul de a provoca vreo depășire:

[illegible]

- un atom simbolic care are o valoare predefinită (în sensul asignării sau al legării) se evaluează la această valoare. Astfel, presupunând că simbolului `a` i se asociase anterior valoarea `alpha`:

```
> a
alpha
```

Excepție fac simbolurile: `nil` care, fiind notația pentru lista vidă cât și pentru valoarea logică fals, se evaluează la el însuși și `t` care se evaluează la valoare logică adevărat (`true` sau `TRUE`). Orice s-expresie diferită de `nil` este considerată a fi echivalentă logic cu `true`:

```
> nil
NIL
> t
TRUE
```

- un atom simbolic care nu are o valoare predefinită dacă este evaluat va provoca un mesaj de eroare de genul `UNBOUND-VARIABLE`;
- o expresie simbolică prefixată cu apostrof (*quote*) se evaluează la ea însăși:

```
> 'alpha
alpha
> '3
3
> '(a 3 ())
(a 3 NIL)
```

- o listă care are pe prima poziție un atom recunoscut ca nume de formă Lisp predefinită sau de funcție definită de utilizator se evaluează astfel: forma Lisp sau funcția dictează maniera în care se face evaluarea următoarelor elemente ale listei: care dintre acestea se evaluează și care sunt lăsate neevaluate. Evaluate ori nu, argumentele apelului sunt trecute apoi corpului formei sau al funcției, drept parametri actuali. Valoarea rezultată în urma evaluării corpului formei sau funcției cu acești parametri este cea care se întoarce:

$$\begin{matrix} & + & 2 & 1 \\ & & & \\ 3 \end{matrix}$$

Primul element al listei este recunoscut drept numele funcției de adunare, +. Parametrii adunării rezultă prin evaluarea următoarelor două elemente ale listei, respectiv atomii numerici 2 și 1. Cum evaluarea îi lasă neschimbați, ei sunt trecuți ca atare drept parametri actuali ai adunării. Aplicarea funcției + asupra lor produce rezultatul 3, care este tipărit. În Lisp, operatorii de calcul aritmetic, ca și orice alt nume de funcție, se notează înaintea argumentelor (notație prefixată).

```
> (setq a 'alpha)
alpha
```

Primul element al listei este recunoscut drept forma Lisp `setq`. Forma `setq` asignează argumentelor din apel de pe pozițiile impare valorile rezultate din evaluarea argumentelor din apel de pe pozițiile pare. Astfel simbolului `a` i se asignează rezultatul evaluării lui `'alpha` adică `alpha`. Valoarea ultimului element al listei este și valoarea întoarsă de funcție. Forma `setq` constituie un exemplu de funcție cu efect lateral, efect ce se manifestă prin lăsarea în context a unor valori asiguate unor simboluri nenumere. Forma `setq` este prima dintr-o serie de forme cu care ne vom întâlni mai departe și care sunt utilizate cu precădere pentru efectele lor laterale mai degrabă decât pentru valorile întoarse de ele.

- evaluarea unei liste care are pe prima poziție o lambda-expresie va fi prezentată în secțiunea 2.4.15 Lambda expresii;
- o listă care are pe prima poziție un nume de macrodefiniție va fi prezentată în secțiunea 2.6 Macro-uri;
- orice altceva (ca de exemplu, o listă care are pe prima poziție o listă diferită de o lambda-expresie, sau un atom ce nu este cunoscut ca nume de formă Lisp sau macro, așadar nu face parte din biblioteca de funcții ale Lisp-ului și nici nu este o funcție definită de utilizator) provoacă, în momentul lansării în evaluare, un mesaj de eroare de genul `UNDEFINED-FUNCTION`.

2.4 Funcții și forme Lisp

2.4.1 Notății

Am vorbit până acum despre funcții și forme Lisp fără să precizăm distincția dintre ele. Funcțiile Lisp-ului își evaluează toate argumentele înainte de a le transfera corpului funcției. În afara funcțiilor, prin utilizarea macro-urilor (v. secțiunea 2.6 Macro-uri), s-au creat forme speciale care aplică reguli specifice de evaluare a argumentelor. Limbajul Lisp numără o clasă semnificativă de funcții și de forme predefinite.

În prezentarea funcțiilor și a formelor vom adopta o convenție de notație, care să ne permită să identificăm argumentele ce se evaluează față de cele care nu se evaluează, tipul acestora, rezultatul evaluării, precum și manifestarea, atunci când va fi cazul, a efectelor laterale. Aceste convenții de notare sunt următoarele:

- `e` – s-expresie
- `n` – atom numeric
- `i` – număr întreg
- `c` – număr complex

- s – atom nenumeric (simbolic)
- l – listă
- f – funcțională (nume sau definiție de funcție sau formă).

Dacă un astfel de simbol, aflat pe poziția unui argument al unui apel de formă Lisp, este prefixat cu un apostrof ($'e$, spre exemplu), vom înțelege că argumentul aflat pe acea poziție se evaluează la un obiect Lisp de tipul indicat de simbol (s-expresie, în cazul exemplului de mai sus). Dimpotrivă, dacă el este notat fără apostrof, atunci argumentul respectiv, ce trebuie să respecte tipul indicat de simbol, nu se evaluează (să observăm că această convenție este una mnemonică pentru că, într-adevăr, prin evaluarea unui obiect Lisp prefixat cu un apostrof, rămânem cu argumentul fără prefix). În plus, pentru simplificare, vom nota întotdeauna argumentele ce rezultă în urma evaluării obiectelor prefixate cu apostrof, prin aceleași litere. Astfel, dacă $'n_1, \dots, 'n_k$ sunt argumentele evaluabile și de tip numeric ale unui apel, vom considera, fără a mai menționa acest lucru, că numerele ce rezultă în urma evaluării acestora sunt notate cu n_1, \dots, n_k . De asemenea, vom considera de la sine înțeles că orice apel realizat cu argumente de tipuri diferite celor indicate în definiții duce la generarea unor mesaje de eroare.

Ca și mai sus, vom utiliza simbolul $>>>$ cu semnificația „se evaluează la”. Astfel, în definițiile de funcții care urmează, printr-o notație $e_1 >>> e_2$ vom înțelege că forma Lisp e_1 (o s-expresie, în acest caz) se evaluează la forma e_2 (de asemenea o s-expresie). Vom renunța însă la descrierea formală a valorii întoarse în favoarea unei descrieri libere, atunci când, datorită complexității operațiilor efectuate de formă sau funcție, o descriere formală ar începe să semene mult prea mult cu însăși o implementare a ei.

Renunțarea la restricțiile unui Lisp pur a dus inclusiv la acceptarea unui mecanism de evaluare care să întoarcă mai mult decât o singură valoare. Astfel, construcția `values` din Common Lisp pune într-o structură specială valori multiple (vom vedea în secțiunea 2.5.8 Valori multiple și exploatarea lor) cum pot exploatate acestea). În notația noastră vom separa prin virgule valorile multiple rezultate dintr-o evaluare: $e_1 >>> e_2, \dots, e_n$. În plus, vom nota prin $\sim nil$ o valoare despre care ne interesează să știm că e diferită de `nil`.

În cazul în care evaluarea produce și efecte laterale, le vom nota după o bară verticală: $e_1 >>> e_2 \mid \langle \text{efecte} \rangle$. Un anumit tip de efect lateral, atribuirea unei valori unui simbol, va fi notat printr-o săgeată orientată stânga: $s \leftarrow e$, cu semnificația: simbolul s se leagă la valoarea e , sau simbolului s i se asignează valoarea e . De asemenea, în notarea efectelor laterale, vom atribui virgulei (,) semnificația unui operator de execuție serială și dublei linii verticale (\parallel), cea de operator de execuție paralelă. Cu alte cuvinte, o secvență de efecte laterale E_1, \dots, E_k va semnifica consumarea secvențială, în ordinea stânga-dreapta, a acestora, pe când printr-o notație $E_1 \parallel \dots \parallel E_k$ vom înțelege consumarea acelorași efecte în paralel.

Atunci când este important momentul evaluării formei care produce valoarea întoarsă de apelul de funcție relativ la momentele producerii efectelor laterale, vom nota evaluarea formei care produce valoarea întoarsă printre efectele laterale într-o pereche de paranteze pătrate, de exemplu: $[e]$.

Dacă anumite efecte laterale, produse pe parcursul evaluării unei forme, nu sunt persistente, în sensul că nu rămân ca observabile în contextul de evaluare a formei și după terminarea evaluării formei, atunci îndepărtarea lor este notată explicit. Astfel, îndepărtarea efectului asignării unei valori simbolului s va fi notată `unbind(s)`. Pentru a descrie acțiuni repetitive vom folosi structura de iterare `while` `<test> {<corp>}` intercalată în lista efectelor laterale.

În prezentările apelurilor de funcții pot să apară și cuvinte cheie, care se recunosc pentru că sunt întotdeauna prefixate cu caracterul două-puncte (:). Într-un apel real ele trebuie reproduse ca atare. Astfel de cuvinte cheie, pot fi:

`:test` cu semnificația că ceea ce urmează este un test care va acționa prin valoarea `T`; de obicei el prefixează un nume de funcțională, care este dat în sintaxa `#'f`;
`:test-not` – prefixează un test care acționează prin valoarea `NIL`.

2.4.2 Asignarea unei valori unui simbol

Într-un limbaj imperativ, o notație de genul $x := y$, unde x și y sunt variabile, este în general recunoscută ca având interpretarea: se atribuie variabilei x valoarea pe care o are variabila y . Să observăm însă că, în sine, o astfel de notație lasă loc la cel puțin încă două interpretări: variabila x „se leagă” la variabila y , înțelegând prin aceasta că încât ori de câte ori y se va modifica, x se va modifica în același mod și variabilei x i se atribuie ca valoare însuși simbolul y . După cum vom vedea, Lisp-ul face posibile toate aceste interpretări. Cea mai apropiată de accepțiunea din limbajele imperative, „atribuirea” (sau „asignarea”) unei valori este prezentată în continuare.

Despre forma `setq` s-a vorbit deja informal în secțiunea 2.3.4 (Evaluarea expresiilor Lisp). Evaluarea argumentelor de pe pozițiile pare și, respectiv, asignările se fac în ordine, de la stânga la dreapta:

```
(setq s1 'e1... sk 'ek) >>> ek | s1 ← e1, ..., sk ← [ek]
```

```
> (setq x '(a b c) y (cdr x))
(B C)
```

Forma **set** este ca și `setq` numai că ea își evaluează toate argumentele: dacă `'e1 >>> s1,... 'e2*k-1 >>> s2*k-1` atunci:

```
(set 'e1 'e2... 'e2*k-1 'e2*k) >>> e2*k | s1 ← e2, ... s2*k-1 ← [e2*k]
```

```
> (set (car '(x y)) '(a b c) (car x) (cdr x))
(B C)
> a
(B C)
```

Forma **psetq** este analogă lui `setq` cu deosebirea că asignările se fac în paralel: mai întâi toate formele de rang par sunt evaluate serial și apoi tuturor variabilelor (simbolurile de pe pozițiile impare) le sunt asignate valorile corespunzătoare: dacă `'e1 >>> s1,... 'e2*k-1 >>> s2*k-1` atunci:

```
(psetq 'e1 'e2... 'e2*k-1 'e2*k) >>> e2*k | e2, ..., [e2*k], s1 ← e2 || ... || s2*k-1 ← e2*k
```

În exemplul următor valorile lui x și y sunt schimbate între ele:

```
> (setq x 'a)
A
> (setq y 'b)
B
> (psetq x y y x)
NIL
> x
B
> y
A
```

2.4.3 Funcții pentru controlul evaluării

Am văzut deja că prefixarea unei s-expresii cu un apostrof este echivalentă apelului unei funcții `quote`. Așadar `quote` împiedică evaluarea:

```
(quote e) >>> e
```

Funcția `eval` forțază încă un nivel de evaluare. Astfel, dacă: `'e1 >>> e2` și `'e2 >>> e3`, atunci: `(eval 'e1) >>> e3`

```
> (setq x 'a a 'alpha)
ALPHA
> (eval x)
ALPHA
> (eval '(+ 1 2))
3
```

2.4.4 Operații asupra listelor

Construcția listelor

Funcția `cons` construiește o celulă din două s-expresii, rezultate din evaluarea celor două argumente, punându-l pe primul în jumătatea *car* și pe cel de al doilea în jumătatea *cdr*:

```
(cons 'e1 'e2) >>> (e1.e2)
```

```
> (cons 'a nil)
(A)
> (cons 'a 'b)
(A . B)
> (cons 'a '(a b))
(A A B)
> (cons 'alpha (cons 'beta nil))
(ALPHA BETA)
```

Funcția `list` creează o listă din argumentele sale evaluate:

```
(list 'e1 ... 'ek) >>> (e1.(...(ek.nil)...) )
```

```
> (list 'a 'b 'c)
(A B C)
> (list 'a '(b c))
```

```
(A (B C))
```

Funcția **append** creează o listă prin copierea și punerea cap la cap a listelor obținute din evaluarea argumentelor sale. Astfel, dacă:

```
'l1 >>> (e11. (... (e1k1.nil) ...)), ..., 'ln >>> (en1. (... (enkn.nil) ...))
```

atunci:

```
(append 'l1 ... 'ln) >>> (e11. (... (e1k1. (... (en1. (... (enkn.nil) ...)) ...)) ...))
```

```
> (append '(a b c) '(alpha beta) '(1 gamma 2))
(A B C ALPHA BETA 1 GAMMA 2)
```

De notat că ultimul argument poate fi orice s-expresie, iar nu o listă cu necesitate.

```
> (setq l1 (list 'a 'b) l2 (list 'c 'd 'e) l3 'f)
F
> (append l1 l2 l3)
(A B C D E . F)
```

Funcția creează celule *cons* noi corespunzătoare tuturor elementelor listelor componente, cu excepția celor aparținând ultimei liste, pe care le utilizează ca atare.

Accesul la elementele unei liste

Funcțiile **car** și **cdr** extrag s-expresiile aflate în pozițiile *car*, respectiv *cdr*, ale unei celule *cons*. Dacă lista e vidă **car** întoarce încă *nil*. Dacă '*l* >>>

(*e*₁.*e*₂), atunci:

```
(car 'l) >>> e1
```

```
(cdr 'l) >>> e2
```

```
(car nil) >>> nil
```

```
(cdr nil) >>> nil
```

```
> (car '(a b c))
A
> (car '((alpha beta) b c))
(ALPHA BETA)
> (cdr '(a b c))
(B C)
> (cdr '(a))
NIL
> (cdr '(a . b))
B
```

Prin combinații *car-cdr* poate fi accesat orice element de pe orice nivel al unei liste. Pentru simplificarea scrierii, implementările de Lisp pun la dispoziție notații condensate, ca de exemplu *caddar* în loc de (*car* (*cdr* (*cdr* (*car* ...)))).

Funcția **nth** selectează un element de pe o poziție precizată a unei liste. Primul argument trebuie să fie un întreg nenegativ și desemnează numărul de ordine, iar al doilea – o listă, din ea urmând a se face selecția. Dacă '*l* >>>

(*e*₀. (... (*e*_{*m*}.*nil*) ...)) și 0 ≤ *n* ≤ *m*, atunci:

```
(nth 'n 'l) >>> en
```

Dacă *n* > *m*, atunci:

```
(nth 'n 'l) >>> nil
```

```
> (nth 1 '(a b c))
B
> (nth 3 '(a b c))
NIL
```

Similar, funcția **nthcdr** efectuează de un număr întreg și pozitiv de ori **cdr** asupra unei liste. Dacă $l = (e_0. (... (e_m.nil) ...))$ și $0 \leq n \leq m$, atunci:

```
(nth 'n 'l) >>> (e_n. (... (e_m.nil) ...))
```

Dacă $n > m$, atunci:

```
(nth 'n 'l) >>> nil
```

```
> (nthcdr 0 '(a b c))
(A B C)
> (nthcdr 1 '(a b c))
(B C)
> (nthcdr 4 '(a b c))
NIL
```

Funcția **last** întoarce ultima celulă **cons** a unei liste. Dacă $l = (e_1. (... (e_k.nil) ...))$, atunci:

```
(last 'l) >>> (e_k.nil)
```

```
(last nil) >>> nil
```

```
> (last '(a b c))
(c)
> (last (cons 'a 'b))
(A . B)
> (last (cons 'a nil))
(A)
```

2.4.5 Operații cu numere

Lisp-ul are o bibliotecă foarte bogată de funcții aritmetice. Dintre ele, prezentăm doar câteva în rândurile următoare.

Operațiile de adunare, scădere, înmulțire și împărțire ($k \geq 1$):

```
(+ 'n1 ...'n_k) >>> n1 + ... + n_k
```

```
(- 'n) >>> -n și dacă k > 1, atunci: (- 'n1 ...'n_k) >>> n1 - ... - n_k
```

```
(* 'n1 ...'n_k) >>> n1 * ... * n_k
```

```
(/ 'n1 ...'n_k) >>> n1 / ... / n_k
```

```
> (+ 1 2 3)
6
> (- 5 (+ 3 3) 2)
-3
> (* 1 2 3)
6
> (/ 6 3 2)
1
```

Funcția / întoarce o fracție (număr rațional) dacă argumentele ei sunt numere întregi și împărțirea nu e exactă.

$$\begin{matrix} > (/ & 2 & 3) \\ 2/3 \end{matrix}$$

Această funcție, ca și tipul de dată număr rațional, oferă, așadar, o cale foarte elegantă de a opera cu fracții zecimale:

```
> (+ 1 (/ 2 3))
5/3
> (* (/ 2 3) (/ 3 2))
1
> (+ (/ 1 3) (/ 2 5))
11/15
```

Așa cum am arătat și mai sus, reprezentarea numerelor ca liste face să nu existe limite ce pot fi atinse ușor în operațiile cu acestea, de aceea apeluri ca cele de mai jos sunt posibile:

[illegible]

Pentru toate funcțiile aritmetice acționează următoarea **regulă de contaminare**: în cazul în care argumentele unei funcții numerice nu sunt de același tip, toate sunt aduse la tipul celui mai „puternic”, conform următoarei ierarhii: complex > real > rațional > întreg.

Reguli de canonizare

Un rațional cu numitor 1 e considerat întreg, un real cu 0 după virgulă e considerat întreg, un complex cu partea imaginară 0 e considerat real.

```
> (+ (/ 1 3) (/ 2.0 5))
0.73333335
```

Toate operațiile de mai sus se aplică și asupra argumentelor numere complexe:

```
> (+ #c(1 2) #c(3 2))
#C(4 4)
> (- #c(1 2) #c(3 2))
-2
> (* #c(1 2) #c(3 2))
#C(-1 8)
> (/ #c(1 2) #c(3 2))
#C(7/13 4/13)
```

Există mai multe funcții de rotunjire a unei valori. Funcțiile **floor** și **ceiling** rotunjesc o valoare în jos, respectiv în sus. Dacă n este întreg, real sau zecimal și $m \leq n < m+1$, cu m un întreg, atunci:

```
(floor 'n) >>> m, n - m
(ceiling 'n) >>> m + 1, n - (m + 1)
```

```
> (floor 2)
2
0
> (floor 2.9)
2
0.9000001
> (floor 2/3)
0
2/3
> (ceiling 2)
2
0
> (ceiling 2.9)
3
-0.099999905
> (ceiling 2/3)
1
-1/3
```

Dacă apelurile se efectuează cu două argumente întregi se obține calculul câtului și al restului (comportamentul unei funcții care calculează modulo). Astfel, dacă c este cel mai mic întreg pozitiv și r este un întreg pozitiv astfel încât $n_1 = c * n_2 + r$, atunci:

```
(floor 'n1 'n2) >>> c, r
(ceiling 'n1 'n2) >>> c + 1, -r
```

```
> (floor 7 3)
2
1
> (ceiling 7 3)
3
-2
```

Există două forme Lisp de incrementare, respectiv, decrementare, fără efecte laterale: funcțiile **1+**, respectiv **1-**:

```
(1+ 'n) >>> n + 1
(1- 'n) >>> n - 1
```

```
> (setq x 1)
1
> (1+ x)
2
> x
1
> (1- x)
0
> x
1
```

După cum se observă, funcțiile nu afectează argumentele. Macro-urile **incf** și **decf** fac același lucru, dar afectează argumentele:

```
(incf 'n) >>> n + 1 | n ← n + 1
(decf 'n) >>> n - 1 | n ← n - 1
```

```
> (setq x 1)
1
> (incf x)
2
> x
2
> (decf x)
1
> x
1
```

Dacă în apel apare și un al doilea argument, el e considerat incrementul, respectiv decrementul:

```
(incf 'n1 'n2) >>> n1 + n2 | n1 ← n1 + n2
(decf 'n1 'n2) >>> n1 - n2 | n1 ← n1 - n2
```

```
> (setq x 1)
1
> (incf x 2)
3
> x
3
> (decf x 4)
-1
> x
-1
```

De notat că argumentele funcțiilor **1+**, **1-**, **incf** și **decf** pot fi orice fel de număr, inclusiv complex. În acest din urmă caz incrementarea, respectiv, decrementarea se aplică numai părții reale.

Calculul valorii absolute: dacă $n \geq 0$, atunci:

```
(abs 'n) >>> n
```

Dacă $n < 0$, atunci:

```
(abs 'n) >>> -n
```

```
> (abs (- 3 5))
2
```

Dacă argumentul este număr complex, rezultatul este modulul, considerat ca număr real.

```
> (abs #c(3 -4))
5.0
```

Funcțiile **max** și **min** calculează maximumul și minimumul unor șiruri de numere. Dacă $n = \max(n_1, \dots, n_k)$, atunci:


```
(max 'n1 ... 'nk) >>> n
  Dacă n=min(n1, ... nk), atunci:
(min 'n1 ... 'nk) >>> n
```

Aritmetica numerelor complexe

În afara operațiilor uzuale cu numere, ce se răsfrâng și asupra numerelor complexe (*, -, *, /, 1+, 1-, incf, decf, abs), următoarele funcții acceptă ca argumente numai numere complexe: **conjugate**, **realpart**, **imagpart**.

```
  Dacă 'c >>> #c(n1 n2), atunci:
(conjugate 'c) >>> #c(n1 -n2)
(realpart 'c) >>> n1
(imagpart 'c) >>> n2
```

Funcția **complex** compune un număr complex dintr-o parte reală și una imaginară:

```
(complex 'n1 'n2) >>> #c(n1 n2)

> (conjugate (complex 1 2))
#c(1 -2)
> (realpart (complex 2 3))
2
> (imagpart (complex 2 3))
3
```

2.4.6 Predicate

Predicatele sunt funcții care întorc valori de adevăr. Multe dintre ele verifică tipuri și relații.

Dacă *e* este un atom, atunci:

```
(atom 'e) >>> t
altfel:
(atom 'e) >>> nil
```

Dacă *e* este un atom simbolic, atunci:

```
(symbolp 'e) >>> t
altfel:
(symbolp 'e) >>> nil
```

Dacă *e* este număr, atunci:

```
(numberp 'e) >>> t
altfel:
(numberp 'e) >>> nil
```

Dacă *e* este număr întreg, atunci:

```
(integerp 'e) >>> t
altfel:
(integerp 'e) >>> nil
```

Dacă *e* este număr real, atunci:

```
(floatp 'e) >>> t  
altfel:  
(floatp 'e) >>> nil
```

```
    Dacă  $n$  este număr întreg par, atunci:  
(evenp 'n) >>> t  
altfel:  
(evenp 'n) >>> nil
```

```
    Dacă  $n$  este număr întreg impar, atunci:  
(oddp 'n) >>> t  
altfel:  
(oddp 'n) >>> nil
```

```
    Dacă  $n$  este număr și  $n \geq 0$ , atunci:  
(plusp 'n) >>> t  
altfel:  
(plusp 'n) >>> nil
```

```
    Dacă  $n$  este număr și  $n \leq 0$ , atunci:  
(minusp 'n) >>> t  
altfel:  
(minusp 'n) >>> nil
```

```
    Dacă  $n=0$ , atunci:  
(zerop 'n) >>> t  
altfel:  
(zerop 'n) >>> nil
```

```
    Dacă  $n_1 = \dots = n_k$ , atunci:  
(= 'n1 ... 'nk) >>> t  
altfel:  
(= 'n1 ... 'nk) >>> nil
```

```
    Dacă  $n_1 < \dots < n_k$ , atunci:  
(< 'n1 ... 'nk) >>> t  
altfel:  
(< 'n1 ... 'nk) >>> nil
```

```
    Dacă  $n_1 > \dots > n_k$ , atunci:  
(> 'n1 ... 'nk) >>> t  
altfel:  
(> 'n1 ... 'nk) >>> nil
```

```
    Dacă  $n_1 \leq \dots \leq n_k$ , atunci:  
(<= 'n1 ... 'nk) >>> t  
altfel:  
(<= 'n1 ... 'nk) >>> nil
```

Dacă $n_1 \geq \dots \geq n_k$, **atunci**:

```
(>= 'n1 ... 'nk) >>> t
```

altfel:

```
(>= 'n1 ... 'nk) >>> nil
```

Dacă 'e >>> s și s este un simbol legat, **atunci**:

```
(boundp 'e) >>> t
```

altfel:

```
(boundp 'e) >>> nil
```

```
> (setq x 'y y 'z)
Z
> (boundp 'x)
T
> (boundp x)
T
> (boundp 'y)
T
> (boundp y)
NIL
```

Dacă e_1 și e_2 sunt izomorfe structural (deși pot fi obiecte Lisp distincte)

atunci:

```
(equal 'e1 'e2) >>> t
```

altfel:

```
(equal 'e1 'e2) >>> nil
```

```
> (equal 'alpha 'alpha)
T
> (equal '(a b c) (cons 'a (cons 'b (cons 'c nil))))
T
> (equal '(a b c) (cons 'a (cons 'b 'c)))
NIL
> (setq x '(a b c) y (cdr x))
(B C)
> (equal y '(b c))
T
> (equal (cdr x) y)
T
```

Dacă e_1 și e_2 reprezintă aceeași structură (obiect) Lisp, cu alte cuvinte, dacă ele adresează elemente aflate la aceeași adresă în memorie, **atunci**:

```
(eq 'e1 'e2) >>> t
```

altfel:

```
(eq 'e1 'e2) >>> nil
```

```
> (eq 'alpha 'alpha)
T
```

În Common Lisp simbolurile sunt reprezentate cu unicitate. Acest lucru face ca apelul de mai sus se evalueze la T. Standardul nu obligă însă reprezentarea cu

unicitate a numerelor și nici a șirurilor de caractere. Din acest motiv nu poate fi garantat un rezultat pozitiv în cazul testării cu `eq` a două numere egale sau a două șiruri de caractere identice și, ca urmare, astfel de operații trebuie evitate:

```
> (eq 1 1)
???
> (eq "alpha" "alpha")
???
> (eq '(a b c) (cons 'a (cons 'b (cons 'c nil))))
NIL
> (setq x '(a b c) y (cdr x))
(B C)
> (eq (cdr x) '(b c))
NIL
> (eq (cdr x) y)
T
```

Predicatul `eq1` funcționează la fel ca și `eq`, cu excepția faptului că el întoarce `T` inclusiv pentru numere egale, deci obiecte care sunt și „conceptual”, iar nu numai strict implementațional, identice. În privința listelor și a șirurilor de caractere `eq` și `eq1` se comportă la fel.

```
> (eq1 1 1)
T
```

Dacă `e=nil` atunci:

```
(null 'e) >>> t
```

altfel:

```
(null 'e) >>> nil
```

2.4.7 Liste privite ca mulțimi

Funcția **member** verifică existența unei s-expresii într-o listă. Dacă `'l >>> (e1. (... (ek. (... (en.nil)...)...))`, unde `ek` este prima apariție a acestei s-expresii în lista `l` care satisface `(f e ek) >>> t`, cu `f` o funcțională ce necesită exact doi parametri, atunci:

```
(member 'e 'l :test #'f) >>> (ek. (... (en.nil)...) )
```

Implicit, Common Lisp consideră funcționala de test ca fiind `eq1`. Când se dorește utilizarea acesteia în test, ea poate fi ignorată:

```
(member 'ek 'l) >>> (ek. (... (en.nil)...) )
```

```
> (member 'alpha '(a alpha b alpha))
(ALPHA B ALPHA)
> (member 3 '(1 2 3 4 5) :test #'<)
(4 5)
> (member 3 '(1 2 3 4 5) :test #'evenp)
Error: EVENP got 2 args, wanted 1 arg.
> (member '(3 4) '(1 2 (3 4) 5))
NIL
> (member '(3 4) '(1 2 (3 4) 5) :test #'equal)
((3 4) 5)
```

```
> (setq x '(1 2 (3 4) 5))
(1 2 (3 4) 5)
> (member (caddr x) x)
((3 4) 5)
```

Funcțiile **union** și **intersection** realizează reuniunea, respectiv intersecția, a două liste, ca operații cu mulțimi, în sensul excluderii elementelor identice. Testarea identității, ca și în cazul funcției **member**, este lăsată la latitudinea programatorului. Testarea trebuie realizată cu o funcție binară anunțată de cuvântul cheie `:test`. Dacă lipsește, testul de identitate se face implicit cu `eq1`. Ordinea elementelor în lista finală este neprecizată. Numai forma apelului este dată mai jos:

```
(union 'l1 'l2 :test #'f)
(intersect 'l1 'l2 :test #'f)

> (union '(a b c d) '(1 a 2 b 3))
(D C 1 A 2 B 3)
> (intersection '(a b c d) '(1 a 2 b 3))
(B A)
```

Testul excluderii poate lăsa incertă proveniența elementelor ce se copiază în lista finală.

```
> (union '((a 1) (b 2) (c 3) (d 4)) '((a alpha) (c gamma) (e
epsilon))) :test #'(lambda(x y) (eq1 (car x) (car y)))
((D 4) (B 2) (A ALPHA) (C GAMMA) (E EPSILON))
> (intersection '((a 1) (b 2) (c 3) (d 4)) '((a alpha) (c gamma) (e
epsilon))) :test #'(lambda(x y) (eq1 (car x) (car y)))
((C 3) (A 1))
```

În aceste exemple testul excluderii verifică identitatea elementelor de pe poziția car a listelor-perechi-de-elemente ce intră în componența argumentelor originale. Perechile de elemente (a 1) și (a alpha), respectiv (c 3) și (c gamma) răspund pozitiv la test. Proveniența elementelor selectate în lista-mulțime finală, în cazul lor, e nedecidabilă. Funcționala lambda, utilizată în aceste exemple, este tratată în secțiunea 2.4.15 Lambda expresii.

2.4.8 Operații logice și evaluări controlate

Operatorii logici în Lisp sunt **and**, **or** și **not**. Dintre aceștia **and** și **or**, pentru că își evaluează argumentele în mod condiționat, sunt considerați și structuri de control. Funcția **not** are același comportament ca și **null**. Dacă e `>>> nil`, atunci:

```
(not 'e) >>> t
altfel:
(not 'e) >>> nil
```

Macro-ul **and** primește un număr oarecare $n \geq 1$ de argumente pe care le evaluează de la stânga la dreapta până când unul din ele întoarce `nil`, caz în care evaluarea se oprește și valoarea întoarsă este `nil`. Dacă, nici unul dintre primele $n-1$ de argumente nu se evaluează la `nil`, atunci **and** întoarce valoarea ultimului argument. Dacă `'e1 >>> nil`, atunci:

```
(and 'e1 e2 ... en) >>> nil
altfel, dacă 'e2 >>> nil, atunci:
(and 'e1 'e2 ... en) >>> nil
ș.a.m.d., altfel:
(and 'e1 'e2 ... 'en) >>> en
```

```
> (setq n 7)
7
> (and (not (zerop n)) (/ 1 n))
1/7
> (setq n 0)
0
> (and (not (zerop n)) (/ 1 n))
NIL
```

Macro-ul **or** primește un număr oarecare $n \geq 1$ de argumente pe care le evaluează de la stânga la dreapta până când unul din ele întoarce o valoare diferită de `nil`, caz în care evaluarea se oprește și valoarea acelui argument este și cea întoarsă de **or**. Dacă, toate primele $n-1$ de argumente se evaluează la `nil`, atunci **or** întoarce valoarea ultimului argument. Dacă `'e1 >>> ~nil`, atunci:

```
(or 'e1 e2 ... en) >>> e1
altfel, dacă 'e2 >>> ~nil, atunci:
(or 'e1 'e2 ... en) >>> e2
ș.a.m.d., altfel:
(or 'e1 'e2 ... 'en) >>> en
```

```
> (setq n 7)
7
> (or (not (zerop n)) (/ 1 n))
T
> (setq n 0)
0
> (or (not (zerop n)) (/ 1 n))
Error: Attempt to divide 1 by zero.
```

2.4.9 Forme pentru controlul evaluării

Cele mai utilizate forme Lisp pentru controlul explicit al evaluării sunt **if** și **cond**. Forma **if** poate fi chemată cu doi sau trei parametri. În varianta cu trei parametri, evaluarea ei se face astfel: dacă `'e1 >>> ~nil`, atunci:

```
(if 'e1 'e2 e3) >>> e2
altfel:
```

```
(if 'e1 e2 'e3) >>> e3
```

În varianta cu doi parametri, dacă: `'e1 >>> ~nil`, atunci:

```
(if 'e1 'e2) >>> e2
```

altfel:

```
(if 'e1 e2) >>> nil
```

```
> (if (zerop (setq x 0)) "eroare" (/ 1 x))
"eroare"
> (if (zerop (setq x 2)) "eroare" (/ 1 x))
```

cond este cea mai utilizată formă de control a evaluării. Sintactic, ea este formată dintr-un număr de clauze. Elementele de pe prima poziție a clauzelor se evaluează în secvență până la primul care e diferit de `nil`. În acest moment celelalte elemente ale clauzei de evaluează și `cond` întoarce valoarea ultimului element din clauză. Dacă toate elementele de pe prima poziție din clauze se evaluează la `nil`, atunci `cond` însuși întoarce `nil`. Dacă `'e11 >>> ~nil`, atunci:

```
(cond ('e11 ... 'e1n1) ... (ek1 ... eknk)) >>> e1n1
ș.a.m.d., altfel dacă 'ek1 >>> ~nil, atunci:
(cond ('e11 ... e1n1) ... ('ek1 ... 'eknk)) >>> eknk
altfel:
(cond ('e11 ... e1n1) ... ('ek1 ... eknk)) >>> nil
```

```
> (cond ((setq x t) "unu") ((setq x t) "doi") (t "trei"))
"unu"
> (cond ((setq x nil) "unu") ((setq x t) "doi") (t "trei"))
"doi"
> (cond ((setq x nil) "unu") ((setq x nil) "doi") (t "trei"))
"trei"
```

Cu toată simplitatea ei, forma `if` are un dezavantaj, și anume faptul că nu permite evaluarea a mai mult decât o singură expresie înainte de ieșire. Soluția o dau formele `when` și `unless`. Astfel, dacă `'e1 >>> ~nil`, atunci:

```
(when 'e1 'e2 ... 'en) >>> en|e2, ..., [en]
altfel:
(when 'e1 e2 ... en) >>> nil
Similar, dacă 'e1 >>> nil, atunci:
(unless 'e1 'e2 ... 'en) >>> en|e2, ..., [en]
altfel:
(unless 'e1 e2 ... en) >>> nil
```

2.4.10 Liste și tabele de asociație

Listele de asociație sunt structuri frecvent folosite în Lisp pentru accesul rapid la o dată prin intermediul unei chei. Elementele unei liste de asociație sunt celule *cons* în care părțile aflate în *car* se numesc **chei** și cele aflate în *cdr* – **date**. Pentru că introducerea și extragerea noilor elemente, de regulă, se face printr-un capăt al listei, ele pot fi făcute să aibă un comportament analog stivelor. Într-o astfel de structură, introducerea unei noi perechi cheie-dată cu o cheie identică uneia deja existentă are semnificația „umbririi” asociației vechi, după cum eliminarea ei poate să însemne revenirea „în istorie” la asociația anterioară. Pe acest comportament se bazează, de exemplu, „legarea” variabilelor la valori.

Funcția **acons** construiește o nouă listă de asociație copiind o listă veche și adăugând o nouă pereche de asociație pe prima poziție a acesteia. Ea nu modifică lista de asociație.

```
Dacă 'l >>> ((e11.e21).(...((e1n.e2n).nil)...)), atunci:
(acons 'e1 'e2 'l) >>> ((e1.e2).((e11.e21).(...((e1n.e2n).nil)...)))
```

```
> (setq la (acons 'a 1 nil))
((A . 1))
> (acons 'b 2 la)
((B . 2) (A . 1))
> (acons 'a 3 la)
((A . 3) (A . 1))
```

Funcția **pairlis** organizează o listă de asociație din chei aflate într-o listă și date aflate în alta. Nu există o ordine a-priorică de introducere a elementelor în lista de asociație. Evident, cele două liste-argument trebuie să aibă aceeași lungime. Dacă `'l1 >>> (e11. (... (e1n.nil) ...))` și `'l2 >>> (e21. (... (e2n.nil) ...))`, atunci:

```
(pairlis 'l1 'l2) >>> ((e11.e21) . (... ((e1n.e2n) .nil) ...))
```

```
> (pairlis (list 'a 'b 'c) (list 1 2 3))
((C . 3) (B . 2) (A . 1))
```

Dacă apare și un al treilea argument, care trebuie să fie o listă de asociație, introducerea noilor elemente se face în fața celor deja existente în această listă.

Funcțiile **assoc** și **rassoc** sunt funcții de acces într-o listă de asociație – **assoc** folosind cheia pentru identificarea elementului căutat, iar **rassoc** – data.

Dacă: `'l >>> ((e11.e21) . (... ((e1k.e2k) . (... ((e1n.e2n) .nil) ...)) ...))`, și `(e1k.e2k)` este prima pereche cheie-dată din lista de asociație în care apare `e1k` pe poziția cheii, atunci:

```
(assoc 'e1k 'l) >>> (e1k.e2k)
```

Pentru aceeași listă de asociație, dacă `(e1k.e2k)` este prima pereche cheie-dată din lista de asociație în care apare `e2k` pe poziția datei, atunci:

```
(rassoc 'e2k 'l) >>> (e1k.e2k)
```

```
> (setq vals (pairlis (list 'a 'b 'a) (list 1 2 3)))
((A . 3) (B . 2) (A . 1))
> (assoc 'a vals)
(A . 3)
> (rassoc 1 vals)
(A . 1)
```

În mod implicit, testul de identificare a cheii sau a datei, se face cu funcția `eq`. Dacă însă se dorește un alt tip de identificare, atunci o funcțională (care trebuie să aibă exact doi parametri) poate fi anunțată, prin cuvântul cheie `:test`.

Astfel, dacă `'l >>> ((e11.e21) . (... ((e1k.e2k) . (... ((e1n.e2n) .nil) ...)) ...))` și `e1k` este prima apariție pe poziția cheii care satisface `(f 'e e1k) >>> t`, cu `f` o funcțională care necesită exact doi parametri, atunci:

```
(assoc 'e 'l :test #'f) >>> (e1k.e2k)
```

În aceleași condiții pentru argumentul `'l`, dar cu funcționala satisfăcând `(f 'e e2k) >>> t`:

```
(rassoc 'e 'l :test #'f) >>> (e1k.e2k)
```

```
> (rassoc 2 vals :test #'>)
(A . 1)
```


Exemplul probează și ordinea în care se transmit argumentele funcționalei: ca prim argument al funcționalei este considerat primul argument al funcției `assoc` sau `rassoc`, iar ca cel de al doilea argument – pe rând câte o cheie, respectiv, o dată din lista de asociație.

2.4.11 Locații și accese la locații

Forma `setf` accesează o locație, pentru completarea ei prin intermediul unei funcționale care, în mod uzual, solicită o valoare depozitată în acea locație. `setf`, așadar, inversează comportamentul funcționalei pe care o primește ca prim argument, care, în loc de a extrage o valoare deja depozitată într-o locație, va deschide calea pentru a se memora acolo o valoare. Dacă `'f1' >>> ea, ... , 'fn' >>> ez` atunci:

```
(setf 'f1 'e1 ... 'fn 'en) >>> en | 'f1 >>> e1, ... , 'fn >>> en
```

```
> (setq l '(a b c))
(A B C)
> (setf (car l) 1 (cadr l) 'beta)
BETA
> l
(1 BETA C)
```

Oricare dintre următoarele funcționale poate fi utilizată ca prim argument: toate funcțiile `c...r`, `nth`. Alte funcționale vor fi adăugate la această listă pe măsură ce vor fi introduse.

2.4.12 Lista de proprietăți a unui simbol

Unui simbol `i` se poate asocia o listă de perechi proprietate-valoare, în care proprietățile sunt simboluri, iar valorile – date Lisp. În această listă o proprietate poate să apară o singură dată. O listă de proprietăți are asemănări cu o listă de asociație (astfel numele de proprietate corespunde cheii, iar valoarea proprietății – datei) dar există și diferențe între ele (în lista de proprietăți o singură valoare poate fi atribuită unei proprietăți, dar mai multe în lista de asociație, ce pot fi regăsite în ordinea inversă a atribuirilor). Implementațional, o listă de proprietăți a unui simbol este o listă de lungime pară, în care pe pozițiile impare sunt memorate (cu unicitate) numele proprietăților și alături de ele, pe pozițiile pare, – valorile acestora.

Cercetarea valorii unei proprietăți a unui simbol se face prin funcția `get`: dacă lista de proprietăți a simbolului `s` este: `(s1.(e1.(...(sn.(en.nil))...)))`, atunci, dacă `sk ∈ {s1, ..., sn}`:

```
(get 's 'sk) >>> ek
```

altfel:

```
(get 's 'sk) >>> nil
```

În cazul în care un al treilea argument e specificat, atunci el indică valoarea care se dorește să înlocuiască valoarea implicit întoarsă `nil` atunci când proprietatea solicitată nu a fost setată: dacă `sk ∉ {s1, ..., sn}`:

```
(get 's 'sk 'e) >>> e
```

Atribuirea valorii unei proprietăți a unui simbol se face printr-un apel `setf` în care pe poziția funcționalei se folosește `get`:

```
> (setf (get 'a 'p1) 'v1)
V1
> (setf (get 'a 'p2) 'v2)
V2
> (get 'a 'p1)
V1
> (get 'a 'p2)
V2
```

Funcția `remprop` îndepărtează o valoare pe de o proprietate a unui simbol. Astfel, dacă lista de proprietăți a simbolului `s` este:

$l = (s_1. (e_1. (... (s_k. (e_k. (... (s_n. (e_n.nil)) ...)) ...)) ...))$, astfel încât:

```
(get 's 's_k) >>> e_k
```

atunci:

```
(remprop 's 's_k) >>> ~nil | l ← (s_1. (e_1. (... (s_n. (e_n.nil)) ...))
```

Funcția **`symbol-plist`** întoarce lista de proprietăți a unui simbol:

Dacă lista de proprietăți a simbolului `s` este: $(s_1. (e_1. (... (s_n. (e_n.nil)) ...))$, atunci:

```
(symbol-plist 's) >>> (s_1. (e_1. (... (s_n. (e_n.nil)) ...))
```

```
> (setf (get 'a 'p1) 'v1)
V1
> (setf (get 'a 'p2) 'v2)
V2
> (symbol-plist 'a)
(P2 V2 P1 V1)
```

2.4.13 Funcții chirurgicale

Funcțiile chirurgicale își justifică numele prin faptul că realizează modificări asupra argumentele. Ele sunt așadar funcții în care efectul lateral este cel care primează.

Funcția **`nconc`** modifică toate argumentele (fiecare de tip listă) cu excepția ultimului, realizând o listă din toate elementele listelor componente. Astfel, dacă:

$l_1 = (e_{11}. (... (e_{1k}^1.nil) ...))$, ... $l_{n-1} = (e_{n-1,1}. (... (e_{n-1,k}^{n-1}.nil) ...))$,

$l_n = (e_{n1}. (... (e_{nk}^n.nil) ...))$

atunci:

```
(nconc 'l_1 ... 'l_{n-1} 'l_n) >>>
(e_{11}. (... (e_{1k}^1. (... (e_{n-1,1}. (... (e_{n-1,k}^{n-1}. (e_{n1}. (... (e_{nk}^n.nil) ...)) ...)) ...)) ...)) |
l_1 >>> (e_{11}. (... (e_{1k}^1. (... (e_{n1}. (... (e_{nk}^n.nil) ...)) ...)) ...))
', ...
l_{n-1} >>> (e_{n-1,1}. (... (e_{n-1,k}^{n-1}. (e_{n1}. (... (e_{nk}^n.nil) ...)) ...)) ...))
```

Notăția pune în evidență faptul că primele $n-1$ argumente, din cele n ale apelului, vor rămâne modificate în urma apelului.

```

> (setq x '(a b c))
(A B C)
> (setq y '(1 2 3))
(1 2 3)
> (setq z '(u v))
(U V)
> (nconc x y z)
(A B C 1 2 3 U V)
> x
(A B C 1 2 3 U V)
> y
(1 2 3 U V)
> z
(U V)

```

Funcția **rplaca** modifică *car*-ul celulei *cons* obținută din evaluarea primului argument la valoarea celui de al doilea argument și întoarce celula *cons* modificată.

Dacă: $l = (e_1.e_2)$, atunci:

```
(rplaca 'l 'e) >>> (e.e2) | l = (e.e2)
```

```

> (setq x '(a b c))
(A B C)
> (rplaca (cdr x) 'd)
(D C)
> x
(A D C)

```

Funcția **rplacd** modifică *cdr*-ul celulei *cons* obținută din evaluarea primului argument la valoarea celui de al doilea argument și întoarce celula *cons* modificată.

Dacă: $l = (e_1.e_2)$, atunci:

```
(rplacd 'l 'e) >>> (e1.e) | l = (e1.e)
```

```

> (setq x '(a b c))
(A B C)
> (rplacd (cdr x) 'd)
(B . D)
> x
(A B . D)

```

Următorul exemplu probează că un apel `append` copiază toate elementele listelor argument cu excepția ultimului:

```

> (setq x '(a b c) y '(d e))
(D E)
> (setq z (append x y))
(A B C D E)
> (rplaca x 'alpha)
(ALPHA B C)
> z
(A B C D E)
> (rplaca y 'beta)
(BETA E)
> z
(A B C BETA E)

```

2.4.14 Forme de apelare a altor funcții

Funcția **apply** cheamă o funcție asupra unei liste de argumente.

```
> (setq f '+)
+
> (apply f '(1 2 3))
6
> (apply #' + '())
0
> (apply #'min '(2 -6 8))
-6
```

Funcția **funcall** aplică o funcție asupra unor argumente.

```
> (cons 1 2)
(1 . 2)
> (setq cons (symbol-function '+))
#<Function +>
> (funcall cons 1 2)
3
> (funcall #'max 1 2 3 4)
4
> (setq x 2)
2
> (funcall (if (> x 0) #'max #'min) 1 2 3 4)
4
> (setq x -2)
-2
> (funcall (if (> x 0) #'max #'min) 1 2 3 4)
1
```

2.4.15 Lambda expresii

O lambda-expresie atașează unui set de parametri formali un corp de funcție. O lambda-expresie poate fi folosită în locul unui nume de funcție:

```
((lambda (s1... sk) ec1... ecn) 'ep1... 'epk) >>> ecn | s1 ← ep1, ..., sk ← epk,  
ec1, ..., [ecn], unbind(s1, ..., sk)
```

Proceduri uzuale de apel sunt: ca prim argument al unei liste supusă evaluării, sau prin intermediul funcțiilor **apply**, **funcall** ori **map...** (a se vedea secțiunea următoare). La evaluare, mai întâi argumentele formale ale lambda-definiției ($s_1 \dots s_k$) se leagă la valorile actuale evaluate ($e_{p1} \dots e_{pk}$), apoi formele corpului definiției ($e_{c1} \dots e_{cn}$) sunt evaluate una după alta. Valoarea întoarsă este cea rezultată din evaluarea ultimei forme. Lambda-definiția marchează un context (domeniu) lexical (discutat în secțiunea 2.5.5 Variabile și domeniile lor) pentru variabilele locale.

```
> ((lambda (x y) (> x y)) 3 2)
T
```

2.4.16 Lambda-funcții recursive⁶

Nu putem utiliza o lambda definiție pentru o funcție recursivă pentru că lambda-funcția nu are nume. Pentru a asocia nume funcțiilor definite ca lambda-expresii se folosește construcția **labels**. Forma unui apel este:

```
(labels (<specificație-legare>*) <apel>*)
```

În care fiecare dintre specificațiile de legare trebuie să aibă forma:

```
(<nume> <parametri> . <corp>)
```

adică analog unei definiții lambda. În interiorul expresiilor din `labels`, `<nume>` va referi acum o funcție ca după un apel:

```
#'(lambda <parametri> . <corp>)

> (labels ((inc (x) (1+ x))) (inc 3))
4
```

În exemplul următor primul argument al lui `mapcar` este o funcție recursivă:

```
> (defun count-instances (obj lsts)
  (labels ((instances-in (lst)
    (if (consp lst)
        (+ (if (eq (car lst) obj) 1 0)
            (instances-in (cdr lst)))
        0)))
    (mapcar #'instances-in lsts)))
COUNT-INSTANCES
> (count-instances 'a '((a b c) (d a r p a) (d a r) (a a)))
(1 2 1 2)
```

Funcția primește un obiect și o listă și întoarce o listă a numărului de apariții a obiectului în fiecare element al listei.

```
> (labels ((dot-product (a b)
  (if (or (null a) (null b))
      0
      (+ (* (car a) (car b)) (dot-product (cdr a) (cdr b))))))
  (dot-product '(1 2 3) '(10 20 30)))
140
```

Apelul de mai sus cuprinde o definiție recursivă a produsului scalar al doi vectori (dați ca liste).

2.4.17 Funcții de corespondență

Din această categorie fac parte funcții care aplică o funcțională asupra argumentelor construite din listele primite ca parametri. Funcțiile de corespondență,

⁶ Exemplele din această secțiune sînt preluate din (Graham, 1994).

exersate corect, formează deprinderea de a gândi transformări aplicate unei mulțimi de obiecte, de o manieră globală, iar nu ca iterări asupra elementelor mulțimii.

Mapcar aplică o funcție asupra elementelor unor liste: dacă $'l_1 \ggg (e_{11}. (... (e_{1k}^1.nil)...)) , \dots 'l_n \ggg (e_{n1}. (... (e_{nk}^n.nil)...))$ și f este o funcțională de n argumente (așadar aritatea funcționalei este egală cu numărul listelor comunicate ca parametri) și dacă lungimea celei mai mici liste argument este k ($k = \min (k^1, \dots, k^n)$), atunci:

$$(\text{mapcar } 'f \ 'l_1 \dots 'l_n) = (\text{list } (\text{funcall } \#f \ 'e_{11} \dots 'e_{n1}) \\ \dots \\ (\text{funcall } \#f \ 'e_{1k} \dots 'e_{nk}))$$

cu alte cuvinte, valoarea întoarsă de **mapcar** va fi o listă de lungime k , în care fiecare element de rang j ($j \in \{1, \dots, k\}$) reprezintă valoarea întoarsă de evaluarea funcționalei f asupra elementelor de rang j din fiecare din listele din intrare (v. și Figura 2.5).

```
> (mapcar 'equal '(a (b c) d e) '(b (b c) f e 3))
(NIL T NIL T)
```

Următorul apel realizează produsul scalar al doi vectori⁷ dați ca liste de numere:

```
> (apply #'+ (mapcar #'* '(1 2 3) '(10 20 30)))
140
```

Următoarea secvență intenționează să înlocuiască nume de persoane cu diminutivele lor într-un text dat:

```
> (setq l (pairlis '(Mihai Gheorghe Nicolae Ion) '(Misu Ghita Nicu Ionica)))
((ION . IONICA) (NICOLAE . NICU) (GHEORGHE . GHITA) (MIHAI . MISU))
> (mapcar #'(lambda(x) (if (null (assoc x l))
                           x
                           (cdr (assoc x l))))
      '(Mihai s-a intilnit cu Mircea ca sa-l viziteze
        impreuna pe Ion))
(MISU S-A INTILNIT CU MIRCEA CA SA-L VIZITEZE IMPREUNA PE IONICA)
```

⁷ Produsul scalar al doi vectori de numere este numărul egal cu suma produselor elementelor de același rang din cei doi vectori: dacă $v_1 = (a_1, \dots, a_n)$, $v_2 = (b_1, \dots, b_n)$, atunci $v_1 \cdot v_2 = a_1 \cdot b_1 + \dots + a_n \cdot b_n$.

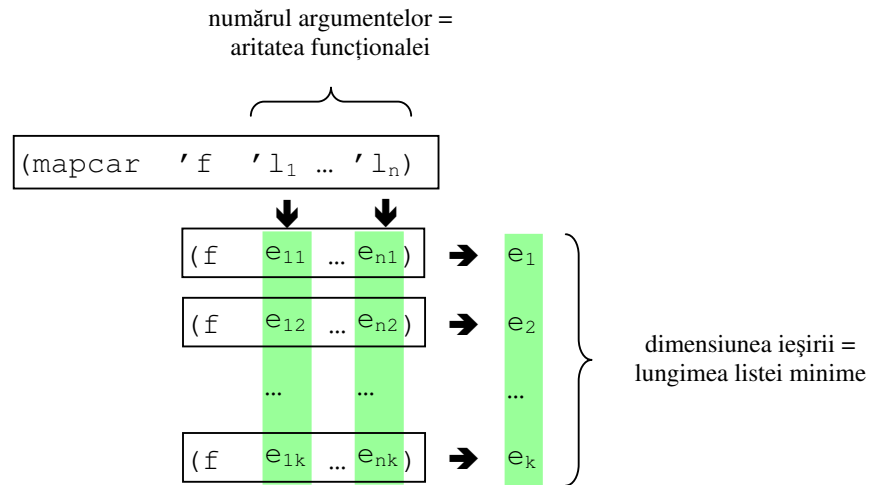


Figura 2.5: Evaluarea în cazul unui apel *mapcar*

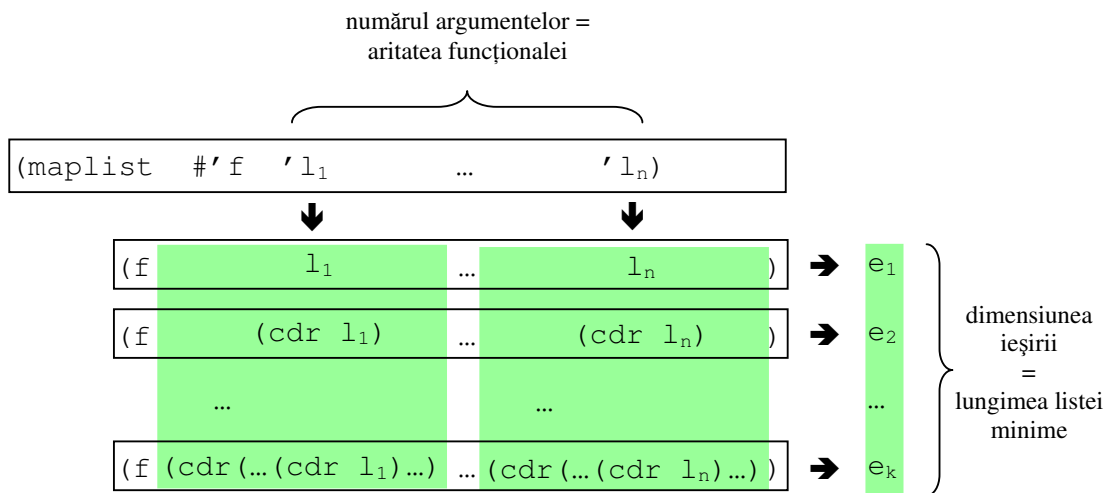


Figura 2.6: Evaluarea în cazul unui apel *maplist*

Această implementare suferă de un defect: execuția este ineficientă pentru că asocierea unui nume într-o listă este făcută de două ori. Vom corecta această deficiență în secțiunea dedicată formei [let](#).

Maplist – funcționează asemănător cu **mapcar**, numai că funcționala este aplicată listelor și *cdr*-urilor succesive ale acestora, în secvență:

```
> (maplist #'(lambda (x) x) '(1 2 3 4))
((1 2 3 4) (2 3 4) (3 4) (4))
> (mapcar #'(lambda (x) (cons 'alpha x))
  (maplist #'(lambda (x) x) '(1 2 3 4)))
((ALPHA 1 2 3 4) (ALPHA 2 3 4) (ALPHA 3 4) (ALPHA 4))
> (mapcar #'(lambda (x) (apply #' + x))
  (maplist #'(lambda (x) x) '(1 2 3 4)))
(10 9 7 4)
```

2.5 Tehnici de programare în Lisp

2.5.1 Definiții de funcții

Definiția unei funcții se face cu construcția **defun**:

```
(defun s (s1... sk) e1... en) >>> s | s ← (lambda (s1... sk) e1... en)
```

Așadar, rezultatul unei definiții de funcții este crearea unei legări între un nume, recunoscut global, – *s*, o listă de variabile, considerate variabile formale ale funcției, – *s₁... s_k* și un corp al funcției – secvența *e₁... e_k*.

Deși neuzuală, este permisă, desigur, definirea de funcții în interiorul altor funcții. O funcție definită în interiorul altei funcții devine cunoscută sistemului însă numai după apelarea cel puțin o dată a funcției în care a fost ea definită:

```
> (defun f1 () (princ "f1"))
F1
> (defun f2() (defun f3() (princ "f3")) (princ "f2"))
F2
> (f1)
f1
"f1"
> (f2)
f2
"f2"
> (f3)
f3
"f3"
> (defun f4() (defun f5() (princ "f5")) (princ "f4"))
F4
> (f5)
Error: attempt to call `F5' which is an undefined function.
[condition type: UNDEFINED-FUNCTION]
> (f4)
f4
"f4"
> (f5)
f5
"f5"
```

2.5.2 Recursivitate

În Lisp recursia este la ea acasă. Iată definiția funcției factorial:

```
> (defun fact (n) (if (zerop n) 1 (* n (fact (1- n)))))
```

O strategie, care nu dă greș, de definire a funcțiilor recursive zice așa:

- începe prin a scrie condiția de oprire;
- scrie apoi apelul recursiv.

```
> (defun fact (n)
  (if (zerop n) 0
      (* n (fact (- n 1)))))
```


FACT

Orice argument, oricât de mare, poate fi dat acestei funcții:

```
>(fact 1000)
402387260077093773543702433923003985719374864210714632543799910429938
512398629020592044208486969404800479988610197196058631666872994808558
901323829669944590997424504087073759918823627727188732519779505950995
276120874975462497043601418278094646496291056393887437886487337119181
045825783647849977012476632889835955735432513185323958463075557409114
262417474349347553428646576611667797396668820291207379143853719588249
808126867838374559731746136085379534524221586593201928090878297308431
392844403281231558611036976801357304216168747609675871348312025478589
320767169132448426236131412508780208000261683151027341827977704784635
868170164365024153691398281264810213092761244896359928705114964975419
909342221566832572080821333186116811553615836546984046708975602900950
537616475847728421889679646244945160765353408198901385442487984959953
31910172335556602139450399736280750137837615307127761926849034352625
200015888535147331611702103968175921510907788019393178114194545257223
865541461062892187960223838971476088506276862967146674697562911234082
439208160153780889893964518263243671616762179168909779911903754031274
62289988005195444414282012187361745992642956581746628302955570299024
324153181617210465832036786906117260158783520751516284225540265170483
304226143974286933061690897968482590125458327168226458066526769958652
682272807075781391858178889652208164348344825993266043367660176999612
831860788386150279465955131156552036093988180612138558600301435694527
224206344631797460594682573103790084024432438465657245014402821885252
470935190620929023136493273497565513958720559654228749774011413346962
715422845862377387538230483865688976461927383814900140767310446640259
899490222221765904339901886018566526485061799702356193897017860040811
889729918311021171229845901641921068884387121855646124960798722908519
296819372388642614839657382291123125024186649353143970137428531926649
875337218940694281434118520158014123344828015051399694290153483077644
569099073152433278288269864602789864321139083506217095002597389863554
277196742822248757586765752344220207573630569498825087968928162753848
863396909959826280956121450994871701244516461260379029309120889086942
028510640182154399457156805941872748998094254742173582401063677404595
741785160829230135358081840096996372524230560855903700624271243416909
00415369010593398383577793941097002775347200000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
>
```

Abia un argument exagerat de mare, de genul:

```
>(fact (fact 1000))
```

ne poate cauza neplăceri (atunci când interpretorul utilizat nu este unul comercial):

```
Error: An allocation request for 1080 bytes caused a need for 2883584
more bytes of heap. This request cannot be satisfied because you have
hit the Allegro CL Trial heap limit. [condition type: STORAGE-
CONDITION]
```

Recursivitate coadă⁸

Se spune că o funcție este coadă-recursivă dacă, după apelul recursiv, nu mai are nimic de făcut. Următoarea funcție e coadă-recursivă:

```
> (defun our-find-if (fn lst)
  (if (funcall fn (car lst)
    (car lst)
    (our-find-if fn (cdr lst)))))
```

pentru că funcția întoarce direct valoarea apelului recursiv. Următoarele funcții nu sunt coadă-recursivă:

```
> (defun our-length (lst)
  (if (null lst)
    0
    (1+ (our-length (cdr lst)))))
> (defun suma (lst)
  (if (null lst) 0
    (+ (car lst) (suma (cdr lst)))))
```

pentru că rezultatul apelului recursiv este transferat, într-un caz lui 1+ și, în celălalt caz, funcției de adunare +. Nici funcția `factorial`, definită mai sus, nu e coadă-recursivă.

Recursivitatea coadă este căutată pentru că multe compilatoare Common Lisp pot transforma funcții coadă-recursive în bucle (iterații). O funcție care nu e coadă-recursivă poate adesea fi transformată într-una care e coadă recursivă prin scufundarea în ea a unei alte funcții ce conține un **argument acumulator** (ce păstrează valoarea calculată până la un moment dat).

Scrierea funcțiilor cu recursivitate coadă prin parametru acumulator

Următoarea funcție întoarce lungimea unei liste:

```
> (defun tail-length (lst acc)
  (if (null lst) acc
    (tail-length (cdr lst) (1+ acc))))
```

Funcția de adunare a elementelor unei liste, de mai sus, scrisă în varianta cu registru acumulator:

```
> (defun sumal (lst ac)
  (if (null lst) ac
    (sumal (cdr lst) (+ ac (car lst)))))
> (sumal '(1 2 3 4 5) 0)
0[1]: (SUMA1 (1 2 3 4 5) 0)
1[1]: (SUMA1 (2 3 4 5) 1)
2[1]: (SUMA1 (3 4 5) 3)
3[1]: (SUMA1 (4 5) 6)
4[1]: (SUMA1 (5) 10)
```

⁸ O parte din exemplele acestei secțiuni sînt preluate din (Graham, 1994).

```

5[1]: (SUMA1 NIL 15)
5[1]: returned 15
4[1]: returned 15
3[1]: returned 15
2[1]: returned 15
1[1]: returned 15
0[1]: returned 15
15

```

Apelul acestei funcții trebuie încapsulat într-un alt apel care să inițializeze acumulatorul:

```

> (defun suma (lst)
  (suma1 lst 0))

```

Același lucru poate fi însă realizat cu ajutorul lui `labels` astfel încât funcția recursivă să fie ascunsă în interiorul alteia care nu afișează argumentul acumulator. În acest fel "bucătăria" apelului recursiv nu se manifestă la suprafață:

```

> (defun our-length (lst)
  (labels ((rec (lst acc)
    (if (null lst)
        acc
        (rec (cdr lst) (1+ acc)))))
    (rec lst 0)))

```

2.5.3 Forme de secvențiere

progn este o formă al cărei scop este pur și simplu acela de a evalua într-o secvență o succesiune de forme Lisp în vederea întoarcerii ultimei valori:

```
(progn 'e1... 'en) >>> en | e1, ..., [en]
```

prog1 se comportă la fel ca `progn`, cu deosebirea că valoarea întoarsă este cea a primei forme din cuprinderea sa:

```
(prog1 'e1... 'en) >>> e1 | [e1], ..., en
```

Utilizând formele `progn` și `prog1` se pot realiza mai multe evaluări în contexte sintactice în care doar o singură formă este permisă (ca de exemplu, toate cele trei poziții ale argumentelor formei `if`). Este clar că rostul utilizării unui `prog1` într-o definiție de funcție este acela de a întoarce o valoare înainte efectuării unor evaluări care sunt importante numai prin efectele lor laterale.

2.5.4 Formele speciale `let` și `let*`

Forma **let** oferă un mijloc de a defini variabile a căror semnificație să fie aceeași într-o anumită întindere de program. Rostul ei este, așadar, de a defini domenii lexicale și de a preciza legări (despre domenii vom vorbi pe larg în secțiunea următoare). Cel mai frecvent tip de apel al formei `let` are forma:

```
(let ((s1 'ei1) ... (sn 'ein)) 'ec1...'eck) >>> eck | ei1, ..., ein, s1 ← ei1 || ... ||
sn ← ein, ec1, ..., [eck], unbind(s1, ..., sn)
```

În acest apel, $s_1 \dots s_n$ sunt variabile locale, $'e_{i1} \dots 'e_{in}$ sunt expresii care, evaluate, configurează valorile inițiale pe care le iau variabilele înainte de evaluarea în secvență a expresiilor $'e_{c1} \dots 'e_{ck}$. Valoarea întoarsă de `let` este cea a ultimei expresii, e_{ck} .

Definiția de mai sus caută să surprindă, în secțiunea de efecte laterale, secvența evaluărilor, cu precădere faptul că evaluarea secvenței $'e_{c1} \dots 'e_{ck}$ se face în contextul inițial al legărilor variabilelor s_1, \dots, s_n la valorile inițiale e_{i1}, \dots, e_{in} , legările fiind făcute în paralel după ce valorile inițiale au fost evaluate serial. Aceste legări sunt însă „uite” la ieșirea din formă. Într-adevăr, dincolo de granița acestei construcții, semnificația variabilelor locale, ca și legările realizate, se pierd. Notăția noastră mai comunică și ideea că valoarea întoarsă este într-adevăr acel e_{ck} , obținut la un anumit moment în secvența evaluărilor, după care contextul legărilor este uitat.

Legarea simultană a variabilelor la valori face posibilă schimbarea între ele a valorilor a două variabile fără intermediul unei a treia variabile care să memoreze temporar valoarea uneia dintre ele:

```
> (let ((x 'a) (y 'b)) (prin1 x) (prin1 y)
    (let ((x y) (y x)) (prin1 x) (prin1 y)))
ABBA
A
```

Folosind forma `let` putem eficientiza soluția exercițiului cu diminutive dat mai sus:

```
> (mapcar #'(lambda(x) (let ((temp (assoc x l)))
                        (if (null temp) x (cdr temp))))
    '(Mihai s-a intilnit cu Mircea ca sa-l viziteze
      impreuna pe Ion))
(MISU S-A INTILNIT CU MIRCEA CA SA-L VIZITEZE IMPREUNA PE IONICA)
```

Forma `let*` este similară formei `let`, cu deosebirea că legarea variabilelor la valori este făcută serial. Cel mai frecvent tip de apel al formei `let*` are forma:

```
(let* ((s1 'ei1) ... (sn 'ein)) 'ec1...'eck) >>> eck | ei1, s1 ← ei1, ...,
ein, sn ← ein, ec1, ..., [eck], unbind(s1), ..., unbind(sn)
```

Desigur, cu `let*` rezultatul permutării de valori de mai sus nu se mai păstrează:

```
> (let ((x 'a) (y 'b)) (prin1 x) (prin1 y)
    (let* ((x y) (y x)) (prin1 x) (prin1 y)))
ABBB
B
```

Exemplele care urmează pun în evidență diverse domenii create cu `let` și `let*`:

```
> (let ((x 1)) (prin1 x)
    (let ((x 2) (y x)) (prin1 x) (prin1 y)))
```

```

                                (prin1 x))
1211
1
> (let ((x 1)) (prin1 x)
    (let* ((x 2) (y x)) (prin1 x) (prin1 y))
    (prin1 x))
1221
1

```

2.5.5 Variabile și domeniile lor

În Lisp noțiunea de **variabilă**, atât de comună în alte limbaje de programare, înțeleasă ca asocierea dintre un nume simbolic, un tip și un spațiu de memorie unde poate fi depozitată o valoare, trebuie privită oarecum diferit. Vom continua să numim variabilă un simbol care apare în codul programului cu intenția ca lui să i se asocieze valori. Mai întâi, așa cum am arătat deja (v. secțiunea 2.3 Tipuri de date în Lisp), variabilele nu au tipuri. Apoi, nu este cazul ca o variabilă să definească un spațiu de memorie care să fie ocupat de o valoare. Aici, simbolului care dă numele variabilei i se poate asocia o valoare.

Un număr de forme ale Lisp-ului (printre ele: `defun`, `lambda`, `let`, `let*`, `do`, `dolist`, `dotimes` etc., adică acele forme ce permit definirea de parametri locali) creează domenii (sau întinderi) ale variabilelor ce apar pe post de parametri locali în aceste forme. Un **domeniu** este un **spațiu lexical** contiguu, mărginit de perechea de paranteze care „îmbracă” o formă, ce evidențiază o listă de parametri locali. La orice moment funcționează următoarea **regulă de legare**: valoarea unei variabile este dictată de ultima legare ce a avut loc în cel mai adânc domeniu care cuprinde variabila în lista de parametri locali ai săi. Dacă nu există nici un domeniu care să numere variabila printre parametrii săi atunci variabila e considerată globală și legarea se face în domeniul de adâncime zero, cel al expresiilor evaluate la *prompter*.

Astfel, în Figura 2.7 sunt schițate trei domenii, în afara celui global (considerat de adâncime zero): cel exterior (sau de adâncime unu), notat cu A, care definește ca locale variabilele x , u și w , și două domenii de adâncime doi, notate B și respectiv C, care au ca variabile locale, B pe x și y , iar C pe x , z și u . În contextul domeniului B valorile variabilelor referite sunt: pentru x și y – cele legate în acest context, pentru u – cea legată în contextul A și pentru z – cea definită în contextul global. În contextul domeniului A, în continuare sunt referite variabilele: x – cu valoarea legată în contextul A și v – cu valoare globală. În sfârșit, în domeniul C sunt referite variabilele: x , z și u – cu valorile date de legările domeniului C, și w – cu legarea din contextul domeniului A. O variabilă care nu este definită ca locală într-un domeniu se spune că este liberă în acel domeniu. Astfel, de exemplu, domeniul A conține variabila liberă v , domeniul B – variabilele u și y , iar domeniul C – variabila w .

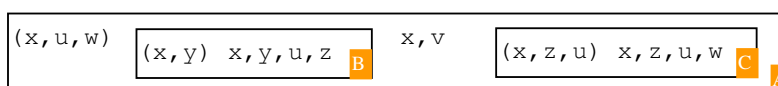


Figura 2.7: Domenii lexicale incluse

Teoretic, există două modalități prin care putem să asociem o valoare unei variabile ce nu este parametru local al unei funcții: prin **legare statică** și prin **legare dinamică**. În legarea statică, valoarea unei astfel de variabile este stabilită de contextul lexical (întinderea de program) în care este ea folosită, pe când în legarea dinamică atribuirea valorii rezultă în urma execuției. O valoare a unei variabile legată lexical poate fi referită numai de forme ce apar textual în interiorul construcției ce a produs legarea. O astfel de legare impune așadar o limitare spațială, iar nu una temporală, a domeniului unde referința se poate realiza în siguranță. Dimpotrivă, o valoare a unei variabile legată dinamic poate fi referită în siguranță în toate momentele ulterioare legării pe parcursul evaluării formei în care a fost efectuată legarea. Ca urmare ea impune o limitare temporală asupra apariției referințelor iar nu una spațială.

Tipul de legare considerat implicit în Common Lisp este cea statică, deși nu întotdeauna lucrurile au stat așa. Într-adevăr în multe implementări anterioare ale Lisp-ului (ca de exemplu, realizările românești DM-LISP (Giumale et al., 1987) și TC-LISP (Tufiș, 1987) (Tufiș, Popescu, 1987)) se optase pentru o legare dinamică. Legarea statică este o trăsătură a majorității limbajelor moderne și ea asigură o mai mare rezistență la erori programelor. O strategie de legare în care valorile atribuite variabilelor să depindă de firul curent al execuției a constituit, la un moment dat, o alternativă de implementare atrăgătoare, din cauza posibilităților mai bogate, aproape exotice, de comunicare a valorilor variabilelor în timpul execuției, pe care această opțiune le oferea. Experiența a arătat însă că un astfel de comportament predispunea la erori și, ca urmare, standardul Common Lisp, fără a-l invalida, nu îl recomandă. El poate fi ales explicit de utilizator printr-o declarație `special` (v. cap. 9 *Declarations* în (Steele, 1990)).

```
> (defun F1()
  (let ((x 'a))
    (defun F2()
      (prin1 x)
    )
  )
)
F1
> (F1)
F2
> (let ((x 'b)) (F2))
A
A
```

Exemplul dorește să evidențieze cele două posibilități de legare ale variabilei `x`. Definiția funcției `F1` cuprinde o legare a simbolului `x` la valoarea a urmată de o definiție a funcției `F2`, în care valoarea lui `x` este tipărită. Corpul definiției funcției `F2` creează un domeniu pentru `x`. În `F2` variabila `x` nu este parametru formal și deci aici nu avem un nou domeniu pentru `x`. Definiția funcției `F2` are loc odată efectuat apelul lui `F1`. Mai departe are loc apelul lui `F2` într-un context în care `x` este legată la valoarea `b` (domeniu precizat de forma `let`, ce va fi prezentată în secțiunea următoare). Un comportament tipic legării dinamice ar trebui să producă tipărirea valorii `B`, pentru că simbolul `x` era legat la această valoare înainte de apelul funcției `F2` în care are loc tipărirea. Faptul că valoarea tipărită este însă `A` semnalează un

comportament tipic legării statice, adică unul în care simbolul x din contextul formei `let` este „altul” decât cel din corpul definiției lui `F2`. Pentru alt exemplu v. și (Graham, 1994), p. 16.

2.5.6 Legări versus asignări

Chestiunile discutate până acum probează că există două maniere în care o variabilă poate căpăta o valoare: prin **legare** (exemplu: `let`) și prin **asignare** (exemplu: `setq`). Orice construcție care leagă o variabilă la o nouă valoare salvează vechea valoare, dacă noul domeniu este inclus într-unul vechi, astfel încât la ieșirea din construcția care a produs noua legare variabila revine la vechea valoare. Exemplul următor lămurește:

```
> (let ((x 'a)) (prin1 x) (let ((x 'b)) (prin1 x)) (prin1 x))
ABA
A
> (let ((x 'a)) (prin1 x) (let () (setq x 'b) (prin1 x)) (prin1 x))
ABB
B
> (let ((x 'a)) (prin1 x) (let ((x 'b)) (setq x 'c) (prin1 x)) (prin1 x))
ACA
A
```

În primul exemplu, avem două domenii incluse unul în altul, pentru variabila x . În domeniul exterior x se leagă la valoarea A , la intrarea în domeniul interior x se leagă la valoarea B , iar la ieșirea din acesta revine la vechea valoare – B .

În exemplul al doilea, x este liber în domeniul interior, dar îi este asignată acolo valoarea B . La ieșirea din acest domeniu, care nu este al său, e normal ca x să păstreze această valoare.

În al treilea exemplu, x este legat din nou în ambele domenii și, ulterior legării interioare la valoarea B , îi este asignată o a treia valoare – C . La revenirea în domeniul exterior, x recapătă valoarea la care era legat acolo – A .

Valoarea pe care o poate avea o variabilă când nu s-a realizat nici o legare explicită a ei se consideră globală. O valoare globală poate fi dată numai prin asignare.

Definiție. Un simbol s apare **liber** într-o expresie când e folosit ca variabilă în acea expresie dar expresia nu creează o legare pentru el. Astfel, în exemplul următor, w , x și z apar libere în `list` iar w și y apar libere în `let`:

```
(let ((x y) (z 10)) (list w x z))
```

Despre o variabilă care este liberă într-un domeniu, precum și în toate domeniile care-l înglobează pe acesta, și care nu are asignată o valoare în domeniul global vom spune că e nelegată (*unbound*).

În secțiunea 2.4.2 Asignarea unei valori unui simbol discutăm modalități diferite de interpretare a unei notații de genul $x := y$. Aparent cea mai simplă, prima interpretare discutată este în realitate cea mai subtilă: ea dorea ca variabila x să primească valoarea pe care o are variabila y , după care ele să aibă „vieți” separate,

în sensul că orice modificări efectuate asupra uneia să nu se resfrângă și asupra celeilalte. Să observăm că nici asignarea simplă, de genul `(setq x y)`, și nici legarea, de genul `(let* ((y ...) (x y)) ...)` nu realizează acest deziderat pentru că există pericolul ca modificările asupra uneia dintre variabile să fie rezultatul intervenției unei funcții chirurgicale (v. secțiunea 2.4.13 Funcții chirurgicale), caz în care ea ar fi resimțită de ambele variabile. Am avea așadar cel de al doilea comportament comentat, care este unul în care cele două variabile devin „surori siameze”. Pentru a realiza primul efect este necesară copierea valorii simbolului `y` ca valoare a lui `x`. Common-Lisp dispune de mai multe funcții de copiere. Astfel dacă `y` este o listă, putem scrie: `(setq x (copy-list y))`. În ultima interpretare variabilei `x` i se atribuie ca valoare însuși simbolul `y`: `(setq x 'y)`.

2.5.7 Forme de iterare

dolist iterează aceleași evaluări asupra tuturor elementelor unei liste. Dacă `l = (e1. (... (ek.nil) ...))`, atunci:

```
(dolist (s 'l 'e) 'ec1... 'ecn) >>> e | s ← e1, ec1, ..., ecn, ..., s ← ek,  
ec1, ..., ecn, [e], unbind(s)
```

```
> (dolist (x '(a b c d) 'exit)
      (prin1 x)
      (princ " "))
)
A B C D
EXIT
```

Într-o altă formă a apelului, valoarea întoarsă poate fi ignorată:

```
(dolist (s 'l) 'ec1... 'ecn) >>> nil | s ← e1, ec1, ..., ecn, ..., s ← ek,  
ec1, ..., ecn, [nil], unbind(s)
```

dotimes iterează aceleași evaluări de un număr anumit de ori: Dacă `n > 0`, atunci:

```
(dotimes (s 'n 'e) 'ec1... 'ecn) >>> e | s ← 0, ec1, ..., ecn, ...,  
s ← n-1, ec1, ..., ecn, s ← n, [e], unbind(s)
```

```
> (dotimes (x 4 x)
      (prin1 x)
      (princ " "))
)
0 1 2 3
4
NIL
```

Într-o altă formă a apelului, valoarea întoarsă poate fi ignorată:


```
(dotimes (s 'n) 'ec1...'ecn) >>> nil | s ← 0, ec1, ..., ecn, ...,
s ← n-1, ec1, ..., ecn, [nil], unbind(s)
```

Forma **do** reprezintă maniera cea mai generală de a organiza o iterație în Lisp. Ea permite utilizarea unui număr oarecare de variabile și controlarea valorilor lor de la un pas al iterației la următorul. Forma cea mai complexă a unui apel **do** este:

```
(do ((s1 'ei1 'es1)... (sn 'ein 'esn)) ('et 'er1... 'erp) 'ec1... 'ecq)
>>> erp | ei1, ..., ein, s1 ← ei1 || ... || sn ← ein, while(not et)
{ec1, ..., ecq, es1, ..., esn, s1 ← es1 || ... || sn ← esn}, er1, ..., [erp], unbind(s1, ..., sn)
```

Primul element al formei este o listă definind variabilele de control ale buclei, valorile lor de inițializare și de incrementare. Astfel, fiecărei variabile îi corespunde o listă formată din numele variabilei, eventual valoarea inițială și, când aceasta apare, eventual o formă de incrementare a pasului. Dacă expresia de inițializare e omisă, ea va fi implicit considerată *nil*. Dacă expresia de incrementare este omisă, variabila nu va fi schimbată între pașii consecutivi ai iterației (deși corpul lui **do** poate modifica valorile variabilei prin *setq*).

Înainte de prima iterație, toate formele de inițializare sunt evaluate și fiecare variabilă este legată la valoarea de inițializare corespunzătoare (acestea sunt legări iar nu asignări, astfel încât după ieșirea din iterație, variabilele revin la valorile la care erau legate înainte de intrarea în iterație).

La începutul fiecărei iterații, după procesarea variabilelor, o expresie de test *e_t* este evaluată. Dacă rezultatul este *nil*, execuția continuă cu evaluarea formelor din corpul **do**-ului: *e_{c1}*, ..., *e_{cq}*. Dacă *e_t* este diferită de *nil* se evaluează în ordine formele *e_{r1}*, ..., *e_{rp}*, ultima valoare fiind și cea întoarsă de **do**.

La începutul oricărei iterații, cu excepția primei, variabilele sunt actualizate astfel: toate formele de incrementare sunt evaluate de la stânga la dreapta și rezultatele reprezintă valorile la care sunt legate variabilele în paralel.

În cazul formei **do***, evaluarea formelor de inițializare urmată de legarea variabilelor la aceste valori, atât la inițializare cât și la fiecare ciclu al iterației, se efectuează serial:

```
(do* ((s1 'ei1 'es1)... (sn 'ein 'esn)) ('et 'er1... 'erp) 'ec1... 'ecq)
>>> erp | ei1, s1 ← ei1, ..., ein, sn ← ein, while(not et) {ec1, ...,
ecq, es1, s1 ← es1, ..., esn, sn ← esn}, er1, ..., [erp], unbind(s1, ..., sn)
```

Exemplele următoare exploatează legările paralele ale variabilelor de index:

```
(defun list-reverse(lst)
  (do ((x lst (cdr x))
      (y '() (cons (car x) y)))
    ((endp x) y)))
```

Funcția `list-reverse`⁹, realizează inversarea unei liste. Variabila de ciclu `x` se leagă la liste din ce în ce mai scurte din cea inițială, `lst`, în timp ce variabila `y` pleacă de la lista vidă și adaugă la fiecare iterație primul element al listei `x`. Când `x` ajunge la lista vidă do se termină întorcând valoarea acumulată în `y`.

```
(defun rev (lst)
  (cond ((null lst) nil)
        ((null (cdr lst)) lst)
        (t (do ((z (cddr lst) (cdr z))
                (y (cdr lst) z)
                (x (progn() (rplacd lst nil) lst) y))
                ((null z) (rplacd y x) y)
                (rplacd y x))))))
```

Funcția `rev` definită aici realizează, de asemenea, inversa unei liste, dar, spre deosebire de exemplul precedent, nu se consumă alte celule de memorie față de cele în care era memorată lista dată în intrare. Cu alte cuvinte, lista se inversează “în ea însăși”. Primele două clauze `cond` tratează lista vidă și lista formată dintr-un singur element. Orice listă care are minimum două elemente este preluată de `do`. Variabilele `x`, `y` și `z` țin minte la fiecare ciclu trei celule `cons` aflate în secvență în lista inițială. La inițializare, `cdr`-ul celulei de pe prima poziție (indicată de `x`) este făcut `nil` pentru a realiza sfârșitul de listă. Operația efectivă a fiecărui pas, realizată pe ultimul rând al definiției de funcție, constă în modificarea părții `cdr` a elementului din mijloc (`y`) pentru a indica elementul precedent (`x`). După realizarea operației din ciclu, variabilele își transferă una alteia valorile în paralel mutându-se toate cu câte un element mai spre sfârșitul listei inițiale. Când ultima variabilă (`z`) devine `nil` înseamnă că `y` indică ultimul element al vechii liste, ca urmare însăși `cdr`-ul acesteia este modificat la elementul precedent și se întoarce valoarea indicată de ea, care reprezintă acum capătul listei inversate.

2.5.8 Valori multiple și exploatarea lor

După cum s-a putut vedea, o seamă de funcții, printre care `floor` și `ceiling`, întorc valori multiple. O generare explicită de valori multiple se poate face cu formele `values-list` și `values`:

```
> (values-list (list 'a 'b 3 'c))
A
B
3
C
> (values 'a 'b 3 'c)
A
B
3
C
```

Diferența dintre ele este că prima solicită o listă pe când a doua primește valorile pe care le “multiplică” ca simple argumente.

Cea mai simplă metodă de exploatare a valorilor multiple constă în transformarea lor în liste. Forma `multiple-value-list` face acest lucru:

⁹ Exemplu preluat din (Steele, 1990).

```
> (multiple-value-list (ceiling 7 3))
(3 -2)
```

Se observă că `values-list` și `multiple-value-list` sunt inverse una alteie:

```
> (multiple-value-list (values-list (list 'a 'b 3 'c)))
(A B 3 C)

> (values-list (multiple-value-list (ceiling 7 3)))
3
-2
```

Forma `multiple-value-call` transferă valori multiple, ca argumente, unui apel de funcție. Funcționala care asamblează valorile multiple trebuie dată ca prim argument al apelului:

```
> (multiple-value-call #'list 1 (ceiling 7 3) (values 5 6) 'alpha)
(1 3 -2 5 6 ALPHA)
```

O formă (macro) care produce legări (generează domenii) prin exploatarea valorilor multiple este `multiple-value-bind`. Sintaxa (simplificată) este următoarea:

```
(multiple-value-bind ({var}*) values-form {form}*) >>> {result}*
```

În care `var` reprezintă un nume de variabilă, `values-form` este o formă care generează valori multiple, iar `form` reprezintă corpul în care legările sunt valorificate. Ultima formă evaluată generează valoarea/valorile înapoi/înapoase de macro.

```
> (multiple-value-bind (x y) (ceiling 7 3) (list x y))
(3 -2)
```

2.5.9 Închideri¹⁰

Combinatia dintre o funcție și un set de legări de variabile libere ale funcției la momentul apelului acelei funcții se numește **încchidere** (*closure*). Închiderile sunt funcții împreună cu stări locale. În exemplul următor, se definește o închidere la nivelul apelului lui `mapcar`, pentru care `n` este o variabilă liberă. Ea dispăre la nivelul definiției funcției `list+`:

```
> (defun list+ (lst n)
  (mapcar #'(lambda (x) (+ x n))
    lst))

> (list+ '(1 2 3) 10)
(11 12 13)
```

¹⁰ Exemplele din această secțiune sînt preluate din (Graham, 1994).

Următoarele funcții folosesc împreună o variabilă comună ce servește de numărător. Închiderea contorului într-un `let` în loc de a-l considera o variabilă globală îl protejează asupra referirilor accidentale.

```
> (let ((counter 0))
    (defun new-id () (incf counter))
    (defun reset-id () (setq counter 0)))
```

În următorul exemplu avem o funcție care la fiecare apel întoarce o funcție împreună cu o stare locală:

```
> (defun make-adder (n)
    #'(lambda (x) (+ x n)))

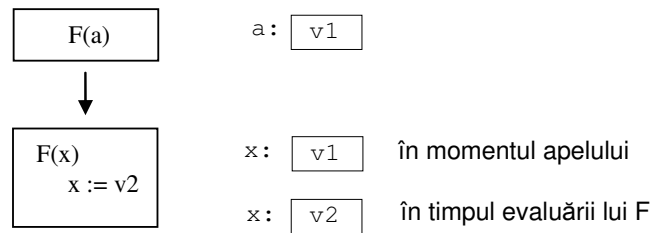
> (setq add2 (make-adder 2)
    add10 (make-adder 10))
#<Interpreted-Function BF162E>
> (funcall add2 5)
7
> (funcall add10 3)
13
```

Funcția `make-adder` primește un număr și întoarce o închidere, care, atunci când e chemată, adună numărul la argument. În această variantă în închiderea întoarsă de `make-adder` starea internă e constantă. Următoarea variantă realizează o închidere a cărei stare poate fi schimbată la anumite apeluri:

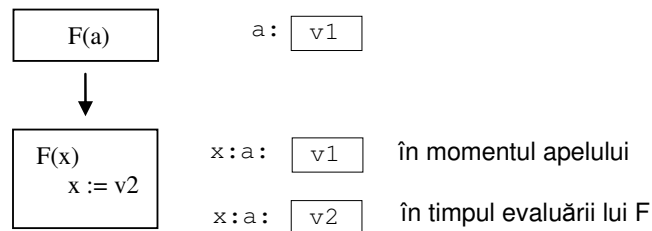
```
> (defun make-adder-b (n)
    #'(lambda (x &optional change)
        (if change (setq n x) (+ x n))))
> (setq addx (make-adder-b 1))
#<Interpreted-Function BF1C66>
> (funcall addx 3)
4
> (funcall addx 100 t)
100
> (funcall addx 3)
103
```

2.5.10 Transferul argumentelor în funcții

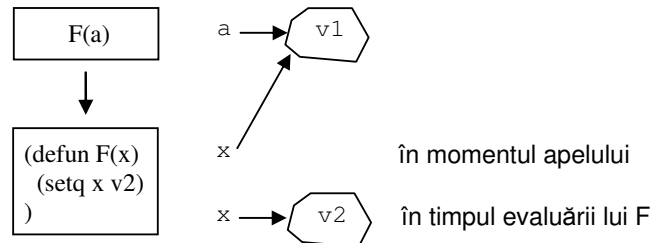
În **transferul prin valoare** (Figura 2.8), înaintea evaluării funcției, valorile actuale ale parametrilor se copiază ca valori ale parametrilor formali ai funcției. Modificări ale valorilor parametrilor formali în timpul evaluării funcției nu afectează valorile parametrilor actuali:

**Figura 2.8:** Transferul prin valoare, în general

În **transferul prin referință** (Figura 2.9) numele parametrilor formali reprezintă sinonime ale numelor parametrilor actuali. Modificarea valorilor parametrilor formali în timpul evaluării funcției provoacă astfel o schimbare a înșăși valorilor parametrilor actuali. Această modificare reprezintă, așadar, un efect lateral al evaluării funcției:

**Figura 2.9:** Transferul prin referință, în general

Transferul în Lisp nu se face nici prin valoare nici prin referință, dar ambele tipuri pot fi simulate. În Lisp un parametru formal se leagă la valoarea comunicată prin parametrul actual. Atunci când, în interiorul funcției, are loc o asignare a unei noi valori variabilei formale, ea este dezlegată de la valoarea veche și legată la una nouă (Figura 2.10).

**Figura 2.10:** Transferul prin valoare în Lisp

Comportamentul este, așadar, acela al unui transfer prin valoare atât timp cât în corpul funcției se realizează numai deasignări ale valorilor parametrilor formali. Îndată însă ce în corpul funcției au loc modificări “chirurgicale” ale valorii parametrilor formali, funcționarea capătă trăsăturile unui transfer prin referință (Figura 2.11) pentru că în acest mod parametrul actual „simte” transformările structurii accesate temporar prin parametrul formal.

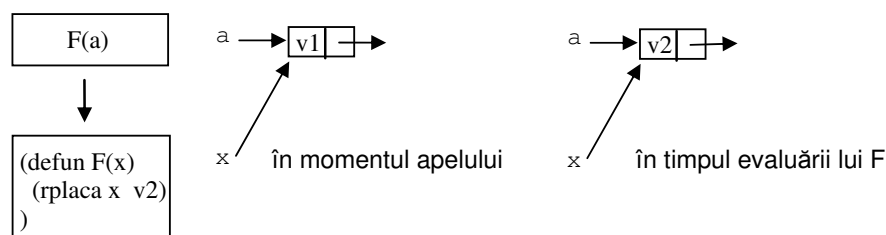


Figura 2.11: Transferul prin referință în Lisp

2.6 Macro-uri

2.6.1 Definiție, macroexpandare și evaluare

Un **macro** este în esență o funcție care definește o funcție. Macro-urile sunt și mijloace prin care sintaxa Lisp-ului poate fi extinsă. Evaluarea apelurilor de macro-uri este un proces în doi pași: întâi o expresie specificată în definiție este construită, apoi această expresie este evaluată. Primul pas – cel al construirii macro-expresiei se numește **macroexpandare**.

O definiție de macro, analog unei definiții de funcție, conține trei elemente: un simbol care dă numele macro-ului, o listă de parametri și corpul.

Într-un apel de funcție definită de utilizator, parametrii actuali sunt evaluați înainte ca parametrii formali din definiție să se lege la aceștia. Rezultă că o evaluare diferențiată a parametrilor nu poate fi realizată printr-o definiție de funcție. Toate formele Lisp-ului în care parametrii se evaluează diferențiat sunt realizate intern ca macro-uri. Dacă ar fi să realizăm o funcție care să aibă comportamentul unui `if`, de exemplu, utilizând însăși forma `if` pentru aceasta, o definiție precum următoarea:

```
> (defun my-if(test expr-da expr-nu)
  (if test expr-da expr-nu))
```

nu satisface, pentru că la intrarea în funcție toți cei trei parametri sunt evaluați. Astfel, într-un apel în care am dori să atribuim variabilei `x` valoarea `DA` sau `NU`, în funcție de un argument, de genul `(my-if t (setq x 'da) (setq x 'nu))`, cu toate că testul se evaluează la `T`, `x` ar fi întâi setat la `DA` și apoi la `NU`, el rămânând în cele din urmă cu această valoare. Apelul de funcție întoarce însă valoarea celui de al doilea parametru:

```

> (my-if t (setq x 'da) (setq x 'nu))
DA
> x
NU

```

Un apel de macro poate include un alt apel de macro. Evaluarea unui apel al acestuia produce, mai întâi, expandarea lui, generând inclusiv un apel al macro-ului interior. Urmează apoi faza de evaluare propriu-zisă a macro-ului exterior în care, la un moment dat, se lansează evaluarea macro-ului interior. Ca urmare, la rândul lui, acesta mai întâi se expandează și, într-o a doua fază abia, se evaluează. Procesul poate continua în același mod pe oricâte niveluri de apeluri de macro-uri în macro-uri, ceea ce presupune inclusiv posibilitatea de a scrie definiții de macro-uri recursive.

Macroexpandările au loc la momente diferite în diverse implementări. Comportamentul descris mai sus este tipic unei implementări de genul interpretorului. Acesta așteaptă momentul evaluării apelului de macro pentru a face macroexpandarea macro-ului. Aceasta înseamnă că orice macro trebuie definit înaintea codului care se referă la el și că dacă un macro este redefinit, orice funcție care referă macro-ul trebuie de asemenea redefinită.

Un compilator va efectua toate macroexpandările la momentul compilării. Un apel de macro ce apare în corpul definiției unei funcții va fi expandat la momentul compilării funcției, dar expresia (sau codul obiect rezultat) nu va fi evaluată până ce funcția nu e chemată. Rezultă că un compilator nu are cum trata apeluri recursive de macro-uri, pentru că el are tendința de a expanda apelul interior înainte de evaluare, ceea ce duce la un alt apel de macro, care trebuie și el expandat ș.a.m.d., făcând astfel imposibilă controlarea acestui proces.

```

> (defmacro nthb (n lst)
  `(if (= ,n 0) (car ,lst) (nthb (- ,n 1) (cdr ,lst))))
NTHB
> (nthb 2 '(a b c d e))
C
> (defmacro fact(n)
  `(if (zerop ,n) 1 (* ,n (fact (- ,n 1)))))
FACT
> (fact 4)
24

```

Este important să facem distincția dintre aceste două momente, respectiv obiectele asupra cărora operează ele, în evaluarea unui macro: **pasul macroexpandării operează cu expresii, cel al evaluării – cu valorile lor.**

2.6.2 Despre apostrof-stânga (*backquote*)

Un apostrof-stânga (```) construiește o formă Lisp conform modelului (*template*) care urmează după el. Sintactic, el prefixează o listă. La evaluare orice formă a listei va fi copiată, cu excepția:

- formelor prefixate de virgulă (`,`), care sunt evaluate;
- unei liste prefixată de o secvență virgulă-*at* (`,@`), care provoacă inserarea elementelor listei.

```

> (setq b 'beta d 'gamma)
GAMMA
> `(a ,b c ,d)
(A BETA C GAMMA)
> (setq x `(a b c))
(A B C)
> `(x 1 ,x 2 ,@x)
(X 1 (A B C) 2 A B C)
> `(a (x 1) (,x 2))
(A (X 1) ((A B C) 2))

```

Apostrof-stânga însuși poate fi copiat într-o expresie prefixată cu apostrof-stânga:

```

> (setq x '(a b c))
(A B C)
> `(a `b ,x)
(A `B (A B C))

```

Putem acum relua definiția unui macro cu comportamentul lui `if`:

```

> (defmacro my-if(test expr-da expr-nu)
  `(if ,test ,expr-da ,expr-nu))
MY-IF
> (my-if t (setq x 'da) (setq x 'nu))
DA
> x
DA

```

Restricții privind folosirea virgulei și a virgulei-*at*:

- virgula poate să apară numai în interiorul unei expresii prefixate cu apostrof-stânga:

```

> `(a ,(cons ,x '(alpha beta)) ,x)
Error: Comma not inside a backquote. [file position = 12]

```

- pentru ca elementele unei liste să fie expandate prin virgulă-*at*, locul acestora trebuie să fie într-o secvență. E o eroare a se scrie la prompter: `,@x`
- obiectul de inserat trebuie să fie o listă, cu excepția cazului în care apare ca ultim element într-o listă. Astfel expresia ``(a ,@1)` se va evalua la `(a . 1)`, dar ``(a ,@1 b)` va genera un mesaj de eroare.

Virgule-*at* se folosesc cu predilecție în definițiile macro-urilor cu un număr nedefinit de argumente.

2.6.3 Asupra manierei de construcție a macro-urilor¹¹

Se începe prin a scrie un apel al macro-ului ce se dorește a fi definit, urmat de expresia în care se dorește ca acesta să fie expandat. Din apelul de macro se construiește lista de parametri alegând câte un nume pentru fiecare argument. Între

¹¹ Exemplele din această secțiune și următoarele sunt reproduse din (Graham, 1994).

rândul apelului și cel al expansiunii se trasează săgeți între argumentele corespunzătoare (Figura 2.12).

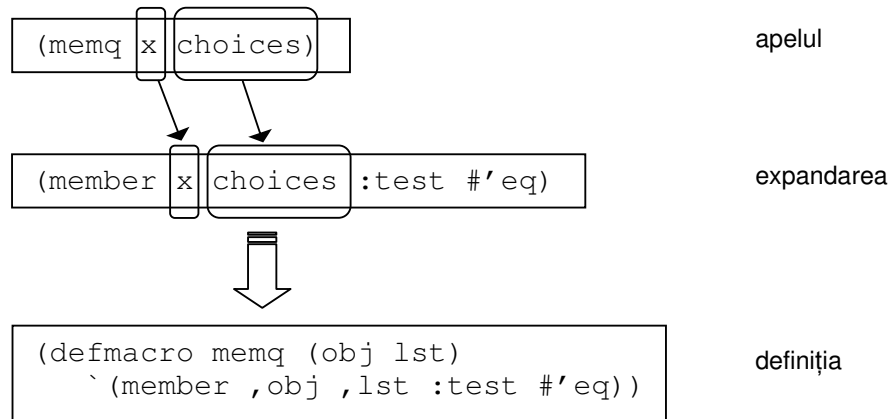


Figura 2.12: Reguli de dezvoltare a macro-urilor

Cazul unui macro cu un număr oarecare de argumente vom începe, la fel, prin a scrie un eșantion de apel de macro. Plecând de la acesta, construim lista de parametri ai macro-ului, dar în care, în afară de parametrul care realizează testul, vom descrie corpul macro-ului printr-un parametru `&rest` sau `&body`.

Să presupunem că vrem să scriem un `while` care primește un test și un corp format dintr-un număr oarecare de expresii. Evaluarea lui va însemna buclarea expresiilor corpului atât timp cât expresia din test întoarce o valoare diferită de `nil`.
Apel:

```
(my-while (not running-engine)
  (look-at-engine)
  (or (ask-advice)
      (think))
  (try-to-fix-the-motor)
  (turn-on-the-key))

(defmacro while (test &rest body)
```

Scriem apoi expansiunea dorită sub apel și unim prin linii argumentele din corpul de apel cu poziția lor din expansiune, dar unde secvența din expansiune ce va corespunde unicului parametru din apel este grupată, ca aici:

```
(do ()
  ((not (not running-engine)))
  (look-at-engine)
  (or (ask-advice)
      (think))
  (try-to-fix-the-motor)
  (turn-on-the-key))
```

În corpul definiției, apoi, parametrul `&rest` (sau `&body`) va fi prefixat cu virgula-*at*:

```
(defmacro my-while (test &body body)
  `(do ()
      ((not ,test))
      ,@body))
```

2.6.4 Cum se testează macro-expandările?

Funcția `macroexpand-1` arată primul nivel al expandării. Funcția `macroexpand` arată toate nivelurile unei expandări.

```
> (do ((w 3)
      (x 1 (1+ x))
      (y 2 (1+ y))
      (z))
    ((> x 10) (princ z) y)
  (princ x)
  (princ y))

> (macroexpand-1 '(do ((w 3)
                      (x 1 (1+ x))
                      (y 2 (1+ y))
                      (z))
                    ((> x 10) (princ z) y)
                    (princ x)
                    (princ y)))
(BLOCK NIL
 (LET ((W 3) (X 1) (Y 2) (Z))
  (TAGBODY
   #:$CFNH
   (WHEN # #)
   (PRINC X)
   (PRINC Y)
   (PSETQ X # Y #)
   (GO #:$CFNH)
   #:$CFNI)
  (PRINC Z)
  Y))
```

T

2.6.5 Asupra destructurizării

Forma `destructuring-bind` primește un șablon, un argument ce se evaluează la o listă și un corp de expresii și evaluează expresiile cu parametrii din șablon legați la elementele corespunzătoare din listă:

```
> (destructuring-bind (x (y) . z) '(a (b) c d)
  (list x y z))
(A B (C D))
```

Destructurizarea e posibilă, de asemenea, în listele parametrilor macro-urilor. Într-o definiție de macro, `defmacro`, se permite ca listele de parametri să fie structuri de liste arbitrare. Când un macro de acest fel e expandat, componentele apelului vor fi asignate parametrilor ca printr-un `destructuring-bind`:

```
> (defmacro our-dolist ((var list &optional result) &body body)
  `(progn
    (mapc #'(lambda (,var) ,@body)
      ,list)
    (let ((,var nil))
      ,result)))
OUR-DOLIST
> (our-dolist (x '(a b c)) (print x))
A
B
C
NIL
> (our-dolist (x '(a b c) 'bravo) (print x))
A
B
C
BRAVO
```

2.6.6 Când să folosim macro-uri?

Există bucăți de cod care pot fi scrise atât ca macro-uri cât și ca funcții. De exemplu:

```
(defun 1+ (x) (+ 1 x))

(defmacro 1+ (x) `(+ 1 ,x))
```

Un `while` însă nu poate fi scris decât ca un macro:

```
(defmacro while (test &body body)
  `(do ()
    ((not ,test))
    ,@body))
```

pentru că el integrează expresiile pe care le primește ca `body` în corpul unui `do` unde ele vor fi evaluate numai dacă `test` întoarce `true`. Printr-un macro se pot controla evaluările argumentelor din apel. **Orice operator care trebuie să acceseze parametrii înainte ca aceștia să fie evaluați trebuie scris ca macro.**

2.6.7 Argumente pro și contra utilizării macro-urilor

Evaluarea la momentul compilării:

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))

(defmacro avg (&rest args)
```

```
` (/ (+ ,@args) , (length args)))
```

În prima definiție `(length args)` este evaluată la momentul rulării pe când în cea de a doua – la momentul compilării:

```
> (macroexpand-1 '(avg 1 3 4 6))
(/ (+ 1 3 4 6) 4)
T
```

2.7 Un exemplu de program care se modifică în timpul rulării

Mai jos, simbolului `animal` îi este atribuită o expresie care, dacă este lansată în evaluare, inițiază un dialog ce duce la recunoașterea unui animal. Atunci când recunoașterea este eronată, programul cere interlocutorului informații pentru rafinarea dialogului. Secvența nouă de dialog astfel generată este inserată în program, astfel încât, la o evaluare ulterioară, arborele de decizie al programului este mai bogat. În acest fel, prin rulări repetate, cunoașterea programului asupra diferitelor animale se perfecționează.

```
(setq animal
  '(let ((int) (aniF))
    (print "Animalul are sange cald? ")
    (setq ras (if (read) (print "caine") (print "lacusta")))
    (print "ok? ")
    (if (not (read))
      (progn
        (my-print (list "Puneti o intrebare la care " ras " sa fie raspunsul
pozitiv: "))
        (setq int (read-line))
        (print "Animalul pentru NIL: ") (setq aniF (string (read)))
        (modify animal (list 'print ras)
          (list 'progn
            (list 'print int)
            (list 'if (list 'read) (list 'print ras) (list 'print aniF)))
          )
        ))
    ))

(defun modify(str old new)
  (cond ((or (atom str) (null str)) nil)
        ((equal (car str) old) (rplaca str new) T)
        ((modify (car str) old new) t)
        (t (modify (cdr str) old new))))

(defun my-print (l)
  (cond ((null l) (terpri))
        (t (princ (car l)) (my-print (cdr l)))))
```

Ceea ce urmează este un dialog în care utilizatorul are în minte animalul șopârlă:

```
> (eval animal)
"Animalul are sange cald? "nil
"lacusta"
"ok? "nil
```

```
Puneti o intrebare la care lacusta sa fie raspunsul pozitiv: Animalul
zboara?
"Animalul pentru NIL: "soparla
T
> (eval animal)
"Animalul are sange cald? "nil
"Animalul zboara?"nil
"SOPARLA"
"ok? "t
NIL
```

Cerințe pentru studenți

- Să fie capabili să implementeze algoritmi în LISP prin funcții și macro-uri.
- Să poată citi o expresie LISP.

Probleme

- P2.1** a). Să se scrie o funcție LISP care să creeze structura din Figura 2.13a.
 b). Să se scrie o funcție care modifică această structură în cea din Figura 2.13b.
 c). Ce va întoarce `(eq (car (cdr (caddr X))) (car (cddddr X)))`?

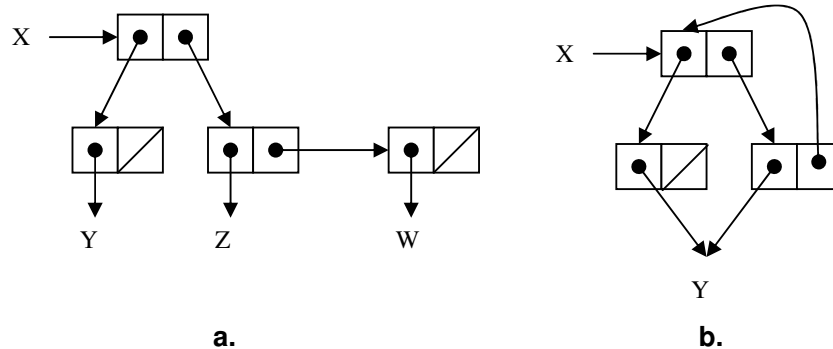


Figura 2.13: a. Structura inițială. b. Structura transformată

- P2.2** a). Să se construiască structură de celule `cons` din Figura 2.14a.
 b). Să se modifice apoi la forma din Figura 2.14b.
 c). La ce se evaluează `(eq (car x) (cadr x))`
- în primul caz;
 - și în cel de al doilea caz.

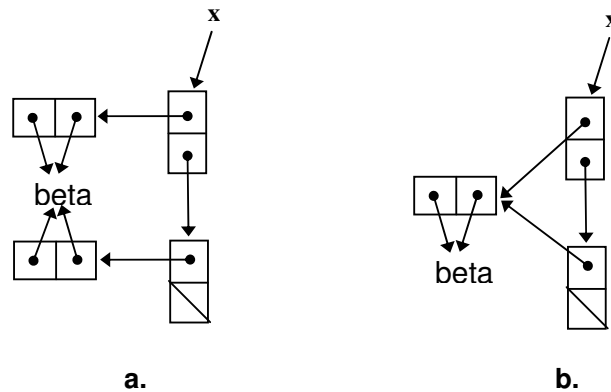


Figura 2.14: a. Structura inițială. b. Structura transformată

P2.3 Să se explice ce întoarce următorul program Lisp:

```
(defun boo(foo)
  (cond ((null foo) 0)
        (t (+ 1 (boo (cdr foo))))))

(boo `(alpha beta gamma))
```

P2.4 Să se transforme următoarea funcție recursivă într-una iterativă:

```
(defun boo-iter(foo)
  (let ((temp 0))
    (while foo
      (setq temp (+ 1 temp)
              foo (cdr foo)))))
```

P2.5 Fie X o mulțime de elemente ordonate crescător, $X=(x_1, x_2 \dots x_n)$, reprezentând puncte pe o axă, și fie $f(x)$ o funcție. Scrieți diverse variante de funcții LISP care să calculeze mulțimea $Y=(f(x_1), f(x_2) \dots f(x_n))$.

Bibliografie

Church, A., 1941. *The Calculi of Lambda-Conversion*, Princeton University Press, Princeton, N. J.

Giumale, Cr., Preoteșcu, D., Șerbănați, L.D. 1987. *LISP*, vol. 1, Editura Tehnică, București.

Graham, P., 1994. *On Lisp. Advanced Techniques for Common Lisp*. Prentice Hall, Englewood Cliffs, New Jersey.

McCarthy, J. 1960. *Recursive Functions of Symbolic Expressions and Their Machine*, în „Communications of the ACM” (poate fi accesată la <http://www-formal.stanford.edu/jmc/recursive.pdf>).

- Steele, G. L., 1990. *Common Lisp the Language*, 2nd edition, Digital Press (versiune on-line la <http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html>).
- Trăușan-Matu, Ș. 2004. *Programare în LISP. Inteligență Artificială și web semantic*. Editura Polirom, Iași, 2004
- Tufiș D. 1987. *TC-LISP-Funcțiile primitive ale interpretorului*. Manual de programare, ITCI, 98 p.
- Tufiș, D., Cristea, D., Tecuci, D. 1987. *LISP*, vol. 2, Editura Tehnică, București.
- Tufiș D., O. Popescu. 1987. *TC-LISP-Biblioteca de funcții*. Manual de programare, ITCI, 101 p.

Situri unde găsiți documentație, cursuri și produse:

<http://www.franz.com/> – situl companiei Franz Inc. care produce și comercializează Allegro Common Lisp. În secțiunea Free Downloads puteți găsi ultimele versiuni de Common Lisp (CL 8.0 în februarie 2007). Allegro CL Certification Program la <http://www.franz.com/services/classes/> oferă gratis cursuri on-line în vederea obținerii certificatelor de dezvoltator CL.

Capitolul 3

Căutare. Rezolvarea problemelor. Sisteme de producție.

Diametrul interior al unei țevi nu trebuie să-l depășească pe cel exterior, altfel gaura va fi pe dinafară.

Michael Stillwell

3.1 Formalizarea problemelor de IA

În acest capitol vom vedea cum pot fi formalizate, în vederea rezolvării, problemele specifice domeniului IA. În multe privințe formalizarea unei probleme de IA este diferită de cea a unei probleme de altă natură. Deși reguli generale sunt dificil de formulat, pentru că varietatea acestor probleme este extrem de mare, vom enunța câteva precepte care pot ghida acest proces.

3.1.1 Problemă, instanță de problemă, spațiul stărilor

De foarte multe ori, când vorbim de o soluție în domeniul IA ne interesează pașii în rezolvarea unei probleme, mai mult decât niște date care să fie obținute în ieșire ca rezultat al unor procese de calcul aplicate unor date prezentate în intrare. Să luăm ca exemplu jocul de șah. O poziție de pe tablă reprezintă complet *starea* jocului într-un anumit moment. Evoluția jocului poate fi privită ca o secvență de tranziții dintr-o stare în alta, plecând de la o stare inițială și ajungând până într-o stare finală. Există o stare inițială descrisă de o poziție standard, întotdeauna aceeași în orice joc de șah. Nu există o anume stare finală, dar poate fi considerată stare finală orice configurație a tablei în care un rege se află într-o poziție atacată și nu poate ieși din acea poziție prin nici o mutare legală. Între aceste două stări există un număr finit, dar extrem de mare, de posibilități de a ajunge prin mutări legale. Numărul total de poziții ale jocului de șah formează ceea ce se numește *spațiul stărilor*.

Vom exemplifica formalizarea problemelor de IA pe câteva exemple de probleme considerate clasice în domeniu. Astfel de probleme, datorită dimensiunii lor mici, sunt cunoscute sub numele de probleme jucărie (*toy problems*). În multe cazuri studiul lor ajută la identificarea metodologiei de rezolvare a problemelor de dimensiune reală. În alte cazuri însă diferența de dimensiune între problemele jucărie și cele reale este atât de mare încât soluțiile probate pe primele nu pot fi aplicate pe cele din urmă. Și într-un caz și în celălalt, studiul problemelor jucărie constituie un antrenament util pentru dezvoltarea abilităților de a lucra cu concepte ale domeniului IA.

Problema 8-puzzle: *Există o tablă 3x3 pe care se găsesc și se pot muta 8 piese pătrate. La un moment dat, o singură piesă poate fi mișcată cu o poziție, pe orizontală sau verticală, în limitele cadrului tablei, în singurul loc liber. Plecând de la o configurație inițială a tablei trebuie să se ajungă într-alta, ce este de asemenea dată.*

Problema misionarilor și canibalilor: *3 misionari și 3 canibali se află la marginea unui râu și doresc să treacă pe celălalt mal. Ei au la dispoziție o barcă de două persoane. Dacă la un moment dat, pe un mal sau pe celălalt numărul canibalilor întrece pe cel al misionarilor, misionarii sunt în pericol de a fi mâncați de canibali. Problema constă în a afla cum pot trece râul cele 6 persoane în deplină siguranță.*

Problema generării frazelor în limbaj natural: *Se dispune de un lexic (prin care fiecărui cuvânt i se atașează o parte de vorbire) și de o gramatică (care este o colecție de reguli, fiecare spunând cum poate fi expandat o categorie compusă în subcompusi). Se dorește generarea unei exprimări corecte gramatical.*

Prima preocupare într-o problemă de IA este de a recunoaște în problema generală instanțele acesteia

Nu poate fi considerat jucător de șah un personaj capabil să rezolve doar o anumită situație de pe tabla de joc, ci cel capabil să abordeze orice situație. Analog, nu poate fi considerat conducător auto o persoană capabilă să miște mașina numai de acasă până la serviciu. Ca să merite acest titlu, teoretic cel puțin, acea persoană trebuie să poată conduce mașina din orice punct în orice alt punct. Aceste exemple vor să pună în evidență diferența dintre probleme și instanțe ale lor. Jocul de șah definește o problemă. Conducerea mașinii – o alta.

În cazul jocului de șah, de obicei se precizează o poziție inițială și o indicație asupra terminării (de exemplu, care jucător trebuie să câștige, eventual și în câte mutări). În cazul conducerii mașinii, se precizează unde se află mașina, unde trebuie ea adusă și se cunoaște o hartă pe care sunt marcate drumurile pe care poate ea circula. Astfel de descrieri definesc *instanțe* de problemă.

În problema 8-puzzle o instanță este dată de o anumită pereche formată din configurația inițială și finală a tablei.

În problema misionarilor și canibalilor, așa cum este ea formulată, nu există decât o unică instanță. Dar ea suportă un enunț general care să admită și alte instanțe de problemă, spre exemplu cazul a n canibali și a n misionari. O instanță poate fi atunci problema în care apar 3 canibali + 3 misionari, alta cea în care apar 4 canibali + 4 misionari etc.

În problema generării limbajului o instanță de problemă este dată de o gramatică. Astfel, de exemplu, am putea avea gramatica $G_1 = \{N_1, T_1, S_1, P_1\}$, în care:

$N_1 = \{\text{PROP}, \text{GN}, \text{GV}, \text{S}, \text{V}\}$ – o mulțime de simboluri neterminale cu semnificațiile: propoziție, grup nominal, grup verbal, substantiv și verb;

$T_1 = \{\text{pisica}, \text{șoarecele}, \text{prinde}\}$ – o mulțime de cuvinte;

$S_1 = \{\text{PROP}\}$ – un simbol de start al gramaticii, alegerea lui semnificând că ceea ce se dorește să se obțină reprezintă propoziții ale acestui mini-limbaj;

$P_1 = \{\text{PROP} := \text{GN GV},$
 $\text{GN} := \text{S},$
 $\text{GV} := \text{V GN},$
 $\text{S} := \text{pisica},$
 $\text{S} := \text{șoarecele},$
 $\text{V} := \text{prinde}\}$ – o mulțime de reguli de producție, cu semnificația evidentă.

O instanță de problema, în acest caz, ar cere obținerea unei propoziții oarecare în limbajul descris de această gramatică.

Toate instanțele aceleiași probleme au ceva în comun: maniera de rezolvare. Același algoritm ar trebui să lucreze pentru orice instanță a unui probleme date. Având o soluție pentru cazul general, rezolvarea unei anumite instanțe reprezintă doar o chestiune de alimentare a programului cu alte date de intrare. Din acest punct de vedere o problemă de IA nu se deosebește cu nimic de o problemă clasică.

3.1.2 Recunoașterea stărilor

O stare reprezintă a configurație anumită a entităților care populează universul problemei în drumul spre soluție. Nu este însă întotdeauna elementar să se decidă care dintre momentele intermediare ale drumului spre soluție trebuie considerată stare și care nu. Dacă am imagina un film care redă drumul spre soluție, întrebarea la care trebuie să răspundem este ce cadre reprezintă imagini de stări și care nu. De asemenea, este foarte util să putem aprecia cât de mare este spațiul stărilor.

A doua preocupare într-o problemă de IA este de a recunoaște o stare și de a aprecia dimensiunea spațiului stărilor

În cazul jocului 8-puzzle recunoaștem imediat că o stare trebuie să reprezinte o configurație anumită a tablei la un moment dat. Ar fi greu de imaginat de ce o alegere a stării ca, de exemplu, aceea în care piesa mobilă să fie într-o poziție intermediară între un cadru și următorul, ar avea vreo semnificație anumită în rezolvarea problemei. Dimensiunea problemei este dată de numărul stărilor teoretic posibile în rezolvarea problemei. La 8-puzzle numărul acestora este $9! = 362.880$ pentru că avem 9 posibilități de a așeza o piesă pe tabla goală, pentru fiecare poziție a acesteia – avem 8 posibilități de a o așeza pe a doua, ș.a.m.d.

În problema canibalilor și misionarilor o stare este o ipostază anumită în traversarea râului. Filmul, prin însuși esența lui de narațiune vizuală, abundă în detalii. Oare ce cadre ale filmului care ar nara vizual traversarea care convine ca

rezolvare a problemei pot fi luate ca reprezentative pentru a descrie stări? Cea mai adecvată ipostază de stare aici este o situație în care barca se află pe unul din maluri și în care avem un număr de misionari și unul de canibali pe malul cu barca în timp ce restul lor se află pe celălalt mal. Nu avem nici un motiv să credem că o poziție intermediară, de exemplu aceea în care barca s-ar afla la mijlocul râului în cursul unei traversări, ar putea să ne intereseze. Considerarea acestor cadre intermediare drept stări ar mări nejustificat spațiul stărilor. Ca să înțelegem de ce este inutilă considerarea unei poziții intermediare în traversarea râului, drept stare, să observăm că din poziția în care barca se află pe unul din maluri, cu un număr anumit de misionari și canibali pe fiecare mal, și până în situația în care barca este pe malul opus, cu o altă combinație de misionari și canibali pe ambele maluri, am putea să deducem – modulo un număr de detalii fără importanță – toate pozițiile intermediare în care barca s-ar fi aflat între un mal și celălalt. Considerarea drept stare a oricărei poziții intermediare nu aduce deci nici un spor de informație față de cazul în care am ignora-o.

Prin aceasta deducem că spațiul stărilor problemei misionarilor și canibalilor este dat de numărul posibil de combinații $c+m+b$ (cu c – numărul de canibali, m – numărul de misionari, b – existența ori nu a bărcii) pe cele două maluri, fiecare în parte respectând restricția, să-i zicem de "ne-ingerare":

$b - ccc+mmm$ (invalidă: barca e pe malul pe care nu se află nici un călător)
 $m+b - ccc+mm$ (invalidă: situație periculoasă pe malul drept)
 $mm+b - ccc+m$ (invalidă: situație periculoasă pe malul drept)
 $mmm+b - ccc$
 $c+b - cc+mmm$
 $c+m+b - cc+mm$
 $c+mm+b - cc+m$ (invalidă: situație periculoasă pe malul drept)
 $c+mmm+b - cc$
 $cc+b - c+mmm$
 $cc+m+b - c+mm$ (invalidă: situație periculoasă pe malul stâng)
 $cc+mm+b - c+m$
 $cc+mmm+b - c$
 $ccc+b - mmm$
 $ccc+m+b - mm$ (invalidă: situație periculoasă pe malul stâng)
 $ccc+mm+b - m$ (invalidă: situație periculoasă pe malul stâng)
 $ccc+mmm+b -$

adică 9 stări valide cu barca pe malul stâng, la care se adaugă tot atâtea cazuri în care barca se află pe malul drept, deci un total de 18 stări.

În problema generării frazelor, o stare poate fi o configurație de simboluri terminale și neterminale, la un anumit moment dat în derivare. Spațiul stărilor este corelat cu dimensiunea limbajului generat de gramatică, mai mare decât acesta, posibil infinită dacă gramatica este recursivă. În cazul particular al gramaticii date mai sus în această secțiune, spațiul total cuprinde aproximativ 37 de stări.

Comparând spațiile acestor probleme cu cel al stărilor jocului de șah, care se știe că este de ordinul a 10^{120} putem înțelege de ce jocul de șah este mult mai greu de rezolvat decât oricare dintre problemele noastre jucărie.

3.1.3 Reprezentarea stărilor

În problema 8-puzzle, o stare poate fi reprezentată de o matrice 3x3 în care elementele sunt numere cuprinse între 0 (reprezentând spațiul) și 9. O altă posibilitate ar fi să reprezentăm tabla ca trei vectori, fiecare cu câte trei poziții. Alegerea uneia sau a alteia dintre posibilități depinde de ușurința cu care se descriu condițiile asupra tablei și acțiunile capabile să transforme o stare în alta, după cum vom vedea mai departe.

Să observăm că în problema canibalilor și misionarilor este suficient să ținem minte situația de pe un mal și poziția bărcii, pentru că situația de pe celălalt mal se obține prin diferență. Deci o reprezentare adecvată poate fi aici un vector format din doi întregi (fiecare având valori cuprinse între 0 și 3) ce dau numărul canibalilor, respectiv al misionarilor, de pe malul stâng, și o variabilă logică care definește existența ori nu a bărcii pe malul stâng: (c, m, b) .

În problema generării limbajului, cum o posibilă stare este o combinație de simboluri terminale și neterminale, reprezentarea unei stări poate fi un șir (listă) de simboluri. Astfel, o derivare posibilă plecând de la gramatica aleasă ca exemplu este: $PROP \rightarrow GN \ GV \rightarrow S \ GV \rightarrow pisica \ GV \rightarrow pisica \ V \ GN \rightarrow pisica \ prinde \ GN \rightarrow pisica \ prinde \ S \rightarrow pisica \ prinde \ pisica$. Stările atinse în această rezolvare au fost în ordine: (PROP), (GN GV), (S GV), (pisica GV), (pisica V GN), (pisica prinde GN), (pisica prinde S), (pisica prinde pisica).

A treia preocupare într-o problemă de IA este găsirea celei mai adecvate reprezentări a stărilor

3.1.4 Reprezentarea tranzițiilor între stări

O soluție într-o problemă de IA înseamnă găsirea unei căi între o stare inițială și o stare finală. Putem să ne imaginăm un proces de rezolvare în două moduri: o manieră în care stările există deja, iar rezolvarea presupune o tranzitare, de către un automat, a acestui spațiu al lor deja generat și o alta în care stările se creează doar în momentul atingerii lor, ele neexistând anterior. În primul caz o mișcare legală provoacă părăsirea unei stări și trecerea în alta, în timp ce în al doilea caz o mișcare legală provoacă generarea unei noi stări plecând de la una dată. Deosebirea este de natură implementațională și are, evident, o mare importanță în eficiența procesului de rezoluție însă nu influențează maniera generală de rezolvare a problemei.

A patra preocupare în descrierea unei probleme de IA este reprezentarea tranzițiilor posibile între stări (reguli)

Pentru efectuarea acestei "deplasări" în spațiul stărilor trebuie definit un set de *operatori*, sau *reguli de producție*, sau simplu *reguli*, care precizează mișcările legale. Se întâmplă adesea ca aceleași tipuri de tranziții să poată fi aplicate între perechi de stări diferite. Spre exemplu, în **Figura 3.1** sunt sugerate prin săgeți de forme diferite patru tipuri de tranziții.

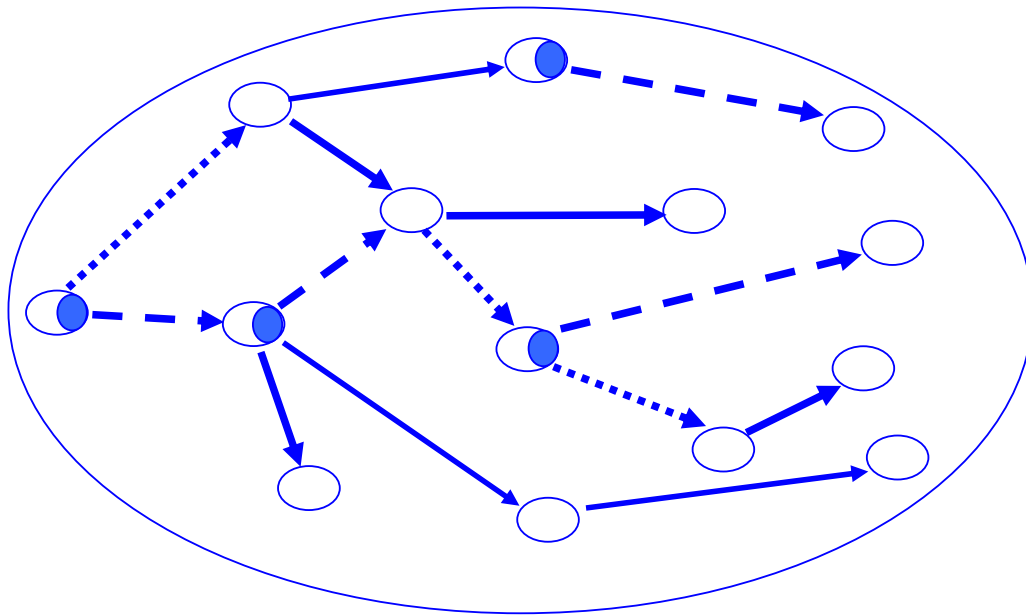


Figura 3.1: Tranzițiile între stări pot fi clasificate după tipuri

Dacă este posibilă clasificarea tranzițiilor pe tipuri, atunci este probabil ca aceleași tranziții să se aplice atunci când stările de plecare au ceva în comun, cu alte cuvinte îndeplinesc niște condiții similare.

Există mai multe modalități de a preciza tranzițiile legale dintr-o stare în alta. Una este de a le descrie "cu cărțile foarte aproape de ochi", cu alte cuvinte într-o manieră care să ia în considerare detalii legate anumite instanțe de problemă. În general obținem astfel un număr mare de reguli. O descriere satisfăcătoare trebuie să se preocupe să fie suficient de generală pentru a o putea utiliza în cât mai multe instanțe date de problemă.

De exemplu, o mișcare uzuală de deschidere la un joc de șah este avansarea pionului cu două poziții. Această mișcare a pionului nu este însă permisă decât atunci când pionul se află în poziția lui inițială, în orice alt moment al jocului pionul putând avansa doar cu o poziție. Am avea așadar o serie de reguli de forma:

```
regula mut•-pion-din-a
DAC.
    pion în pozi•ia (a,2) •i
    pozi•ia (a,3) e liber• •i
    pozi•ia (a,4) e liber•
ATUNCI
    mut• pionul din pozi•ia (a,2) în pozi•ia (a,4).
```

Vom avea un număr de 8 reguli de acest fel, câte una pentru fiecare pion. O reprezentare mai economică ar utiliza o singură regulă capabilă să exprime poziția oricărui pion aflat pe al doilea rând care avansează pe rândul patru.

```

regula mut•-pion(x)
DAC•
    pion în pozi•ia (x,2) •i
    pozi•ia (x,3) e liber• •i
    pozi•ia (x,4) e liber•
ATUNCI
    mut• pionul din pozi•ia (x,2) în pozi•ia (x,4).

```

În general o regulă este de forma:

```

<nume regulă>
DACĂ <condiții> ATUNCI <acțiuni>

```

Activitatea pe care o poate desfășura o regulă poate fi exprimată astfel: dacă partea de condiții este satisfăcută de starea curentă, atunci se realizează acțiunile. Uneori, partea de acțiuni înseamnă generarea unei noi stări, alteori doar tranzitarea într-o stare care era anterior generată.

Din cele de mai sus rezultă că preocuparea a patra constă în inventarierea tuturor tranzițiilor posibile între stări și descrierea acestor tranziții ca reguli formate din condiții și acțiuni. Un set de reguli este specific unei probleme iar nu unei instanțe de problemă. Cu alte cuvinte, același set de reguli trebuie să poată fi aplicat pentru rezolvarea oricărei instanțe a unei probleme date.

În 8-puzzle, reguli posibile ar putea fi: mută-1-sus, mută-1-jos, mută-1-dreapta, mută-1-stânga, mută-2-sus, ș.a.m.d., fiecare dintre ele condiționând, bineînțeles, executarea mișcării de existența unei poziții în direcția respectivă cuprinsă în cadrul tablei. Am avea astfel câte patru reguli pentru fiecare număr (plăcuță), adică 32 de reguli. Un salt calitativ în reprezentarea regulilor îl obținem însă dacă în loc să ne intereseze mutarea unei piese ne-am concentra asupra mutării blancului. Putem astfel să restrângem numărul de reguli la 4: mută-blanc-sus, mută-blanc-jos, mută-blanc-dreapta, mută-blanc-stânga.

```

Regula mut•-blanc-sus
DAC•
    blancul nu e lipit de marginea de sus a tablei
ATUNCI
    schimb• pozi•ia blancului cu a c•su•ei aflat• deasupra
    acestuia

```

În problema generării frazelor, o regulă ar trebui să mimeze destul de aproape o regulă de producție a gramaticii, pentru că, între o stare și următoarea, un simbol aflat în partea stângă a unei reguli dispare, locul lui fiind luat de simbolurile aflate în partea dreaptă a regulii. Ca urmare vom avea tot atâtea reguli ale sistemului de IA câte reguli de producție are gramatica.

3.1.5 Căutarea soluției

Am discutat până acum despre **spațiul stărilor** unei probleme și am învățat să alegem o **reprezentare** pentru stări. Știm că tranzițiile între stări sunt descrise de un set de operatori pe care le-am numit **reguli de producție** și am văzut că a rezolva o problemă înseamnă a găsi un drum între o **stare inițială** și una sau mai multe **stări finale** ce sunt cunoscute precis ori numai descrise printr-un set de

restricții. Maniera în care organizăm căutarea în spațiul stărilor, deci modul în care înălțăm operatorii pentru găsirea soluției constituie următoarea preocupare.

Problemele de IA se aseamănă prin aceea că în toate rezolvarea presupune nu găsirea unei anume stări, ci găsirea unui drum care să unească o stare inițială de una finală. Atunci când starea finală nu este precizată cu exactitate, găsirea unui drum către o stare, care este recunoscută ca fiind finală prin satisfacerea unor constrângeri specifice, coincide, totodată, cu darea unui răspuns așteptat.

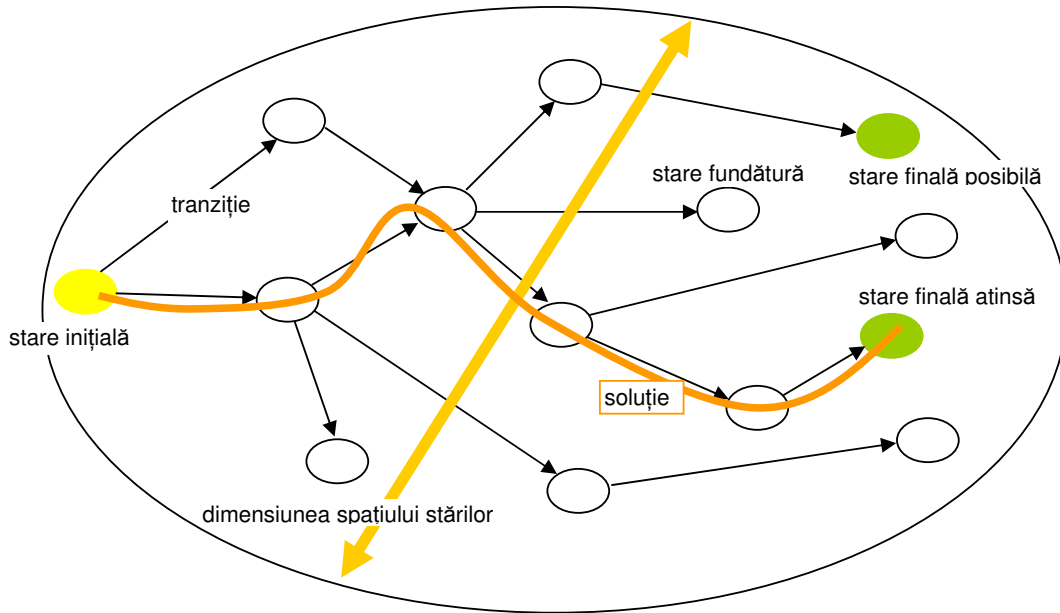


Figura 3.2: Rezolvarea unei probleme de IA prin navigare în spațiul stărilor

Acest lucru face ca o problemă de IA să fie una de căutare într-un spațiu al stărilor. Din cauză că, adesea, spațiul stărilor este foarte mare se utilizează diferite *strategii* pentru a eficientiza căutarea. Există mai multe tipuri de strategii, dar ele sunt clasificate în două clase mari: **irevocabile** și **tentative**. Aplicarea unei strategii este necesară pentru a decide calea de urmat în situațiile în care dintr-o stare anumită există mai multe căi de a tranzita într-o altă stare, ca urmare mai mult decât o singură regulă este posibil de a fi aplicată.

Alegerea unei strategii reprezintă cea de a 5-a preocupare în rezolvarea unei probleme de IA

3.2 Exemple de formalizări

Vom continua exemplificarea primelor patru precepte pentru rezolvarea unei probleme de IA, prezentate în secțiunea precedentă, cu încă două probleme, de asemenea bine-cunoscute.

3.2.1 Maimuța și banana

O maimuță este închisă într-o cușcă în care se mai află o banană atârnată de tavan la o înălțime la care maimuța nu poate ajunge și, într-un colț, o cutie. După un număr de încercări nereușite de a apuca banana, maimuța merge la cutie, o deplasează sub banană, se urcă pe cutie și apucă banana. Se cere să se formalizeze maniera de raționament a maimuței ca un sistem de reguli de producție.

Cerința I. Problemă – instanță de problemă

Distincția dintre problemă și instanță a ei este neesențială în acest caz. Sunt posibile slabe variații ale enunțului în care de exemplu se precizează pozițiile inițiale relative ale obiectelor din cușcă etc.

Cerința a II-a. Aprecierea a ce reprezintă o stare și a spațiului stărilor

Dacă ar fi să eliminăm dintr-un film care ar derula acțiunile maimuței între situația în care ea se află într-un colț, cutia într-alt colț și banana e legată de tavan și situația în care maimuța e urcată pe cutie, la verticala bananei și ține în mână banana, acele secvențe pe care nu le considerăm importante pentru a rămâne numai cu cadrele semnificative, am rămâne probabil cu următoarea secvență de “benzi desenate”:

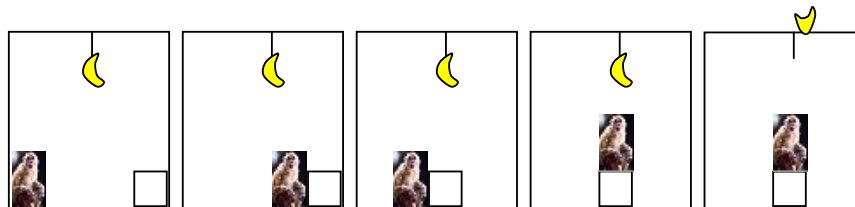


Figura 3.3: O secvență de stări care rezolvă problema maimuței și a bananei

În afara acestor stări, care sunt evidente și absolut necesare în derularea filmului, dacă am investiga alte posibile ipostaze ale maimuții în tentativa ei de a ajunge la banană, am găsi, probabil, printre altele, și pe acestea:

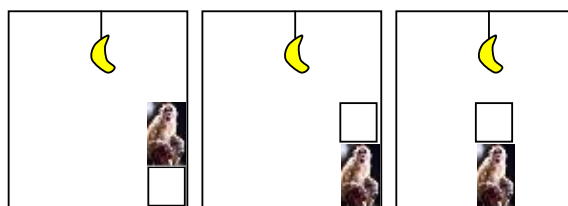


Figura 3.4: Stări posibile dar neinteresante

Mulțimea tuturor acestor stări, fie că sunt ori nu pe drumul spre soluție, formează mulțimea stărilor problemei. Simplitatea acestei probleme ne-ar putea determina să lăsăm la o parte acele stări care nu pot face parte dintr-o eventuală cale spre soluție. Acesta este unul din pericolele care pot face o soluție inacceptabilă. Nu trebuie să uităm că ceea ce facem noi este formalizarea unei probleme ce trebuie rezolvată de “cineva” care nu cunoaște soluția sau cel puțin, nu o cunoaște în maniera în care o cunoaștem noi. Un inventar atent al tuturor situațiilor în care ar putea să ajungă un automat ce se mișcă în spațiul dat trebuie întotdeauna făcut. Să nu amestecăm cunoașterea noastră, ca ființe raționale, cu cea a mașinii care nu “știe” nimic în momentul în care începe să “descopere” soluția.

Cerința a III-a. Reprezentarea stărilor

În reprezentarea stărilor va trebui să ne gândim care sunt aspectele care contează în scenele pe care le-am decelat ca semnificative pentru a reprezenta stări. Uneori ceea ce contează sunt relațiile dintre obiectele ori personajele conținute în scene. Este ceea ce se întâmplă în problema noastră: aici ne interesează relațiile reciproce care există între maimuță, cutie și banană. Astfel

- maimuța față de cutie se poate afla: “la distanță”, “lângă”, “pe” sau “sub”;
- cutia față de banană se poate afla: “lateral” sau “sub”;
- maimuța față de banană se poate afla: “la distanță” (adică la o distanță de unde nu o poate apuca), “aproape” (adică la o distanță de unde o poate apuca) sau “ținând-o”.

Desigur că dacă avem codificare aceste relații, cele simetrice lor (respectiv dintre cutie și maimuță, banană și cutie și banană și maimuță), pot fi inferate imediat și deci nu mai trebuie codificate.

Următoarele predicate devin astfel cele de care avem nevoie.

Relația Maimuță – Cutie:

MC-departe = Maimuța se află departe de Cutie

MC-lângă = Maimuța se află lângă Cutie

MC-pe = Maimuța se afla pe Cutie

MC-sub = Maimuța de află sub Cutie

Relația Cutie – Banană:

CB-lateral = Cutia este așezată lateral față de Banană

CB-sub = Cutia este așezată sub Banană

Relația Maimuța – Banană:

MB-departe = Maimuța se află departe de Banană

MB-aproape = Maimuța se află aproape de Banană

MB-ține = Maimuța ține Banana

Cu aceste predicate, descrierea stărilor problemei devine:

Starea inițială: **MC-departe**, **CB-lateral**, **MB-departe**.

Starea finală: **MC-pe**, **CB-sub**, **MB-ține**.

Cerința a IV-a. Reprezentarea regulilor

Regulile reprezintă descrierea tranzițiilor între stări. Când proiectăm regulile, ca și în cazul reprezentării stărilor, ne punem problema să descriem toate tranzițiile posibile, chiar și cele care nouă, cu mintea noastră de ființe umane, ni se par absurde, ca de exemplu, urcarea maimuței pe cutie într-un punct ce nu se află pe verticala bananei, împingerea ei până într-un punct ce nu se află pe verticala bananei, urcarea cutiei deasupra capului ori coborârea maimuței de pe cutia aflată la verticala bananei fără banană. Trebuie să ne plasăm în postura mașinii care nu știe soluția, ci trebuie s-o găsească singură. Ca urmare trebuie să-i îngăduim toate mișcările posibile în acest univers pe care l-am descris, urmând ca ea singură să evite acele mișcări ce sunt absurde sau inutile.

Următoarele tranziții pot fi inventariate:

- aflată departe de cutie, maimuța de aproprie de cutie: **apropie-MC**;
- aflată lângă cutie, maimuța se depărtează de cutie: **depărtează-MC**;
- aflată lângă cutie, maimuța trage cutia sub banană: **trage-sub-MCB**;
- aflată lângă cutie și sub banană, maimuța trage cutia de sub banană: **trage-lateral-MCB**;
- aflată lângă cutie, maimuța se urcă pe ea: **urcă-MC**;
- aflată pe cutie, maimuța coboară de pe ea: **coboară-MC**;
- aflată pe cutie, maimuța se ridică pentru a se apropia de banană: **apropie-MCB**;

- aflată pe cutie și ridicată pentru a apuca banana, maimuța se ghemuiește, depărtându-se de banană: **depărtează-MCB**;
- aflată lângă cutie, maimuța își urcă cutia deasupra capului: **urcă-pe-cap-MC**;
- din postura în care maimuța ține cutia deasupra capului, maimuța își dă jos cutia de pe cap: **coboară-de-pe-cap-MC**;
- aflată aproape de banană, maimuța apucă banana: **apucă-MB**.

Vom conveni să exprimăm regulile prin forma generală:

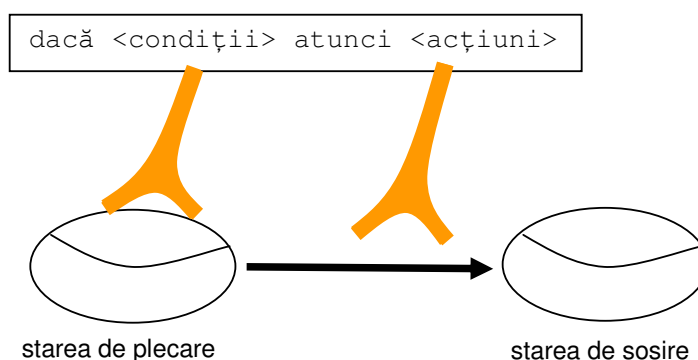


Figura 3.5: O regulă

În partea de condiții caracterizăm minimum de cunoștințe necesare pentru a descrie starea în care dorim să se aplice regula. Trebuie să ne imaginăm o regulă ca și când n-ar fi aplicată de noi ci s-ar aplica singură atunci când găsește condiții favorabile, adică când starea curentă a sistemului satisface condițiile ei. Partea de condiții nu conține nicidecum o descriere exhaustivă a stării, ci numai a unei părți a ei, ceea ce este semnificativ pentru depistarea stării. De exemplu, dacă dorim ca o regulă să se aplice numai când maimuța se află lângă cutie, în partea de condiții a regulii vom include numai predicatul **MC-lângă**, deși starea respectivă, de exemplu starea a doua din secvența de mai sus, mai are în descrierea ei și predicatele **MB-departe** și **CB-lateral**.

Odată găsită starea în care se verifică condițiile regulii, regula se aplică, ceea ce are ca rezultat efectuarea anumitor schimbări în stare. Schimbările din stare sunt descrise în partea ei de acțiuni. Din nou, partea de acțiuni a regulii nu descrie în întregime starea de sosire ci numai schimbările care se produc în stare pentru a o face diferită de starea de plecare. Să mai observăm că starea de plecare din Figura 5 nu trebuie confundată cu starea inițială din formularea problemei, după cum starea de sosire e altceva decât starea finală. Orice pereche de două stări, între care poate avea loc o tranziție, cuprinde o stare de plecare, respectiv una de sosire.

lată exprimările câtorva reguli:

apropie-MC:

dacă {MC-departe} atunci ȘTERGE{MC-departe}, ADAUGĂ{MC-lângă}

depărtează-MC:

dacă {MC-lângă} atunci ȘTERGE{MC-lângă}, ADAUGĂ{MC-departe}

trage-sub-MCB:

dacă {MC-lângă, CB-lateral} atunci ȘTERGE {CB-lateral},
ADAUGĂ{CB-sub}

trage-lateral-MCB:

dacă {MC-lângă, CB-sub} atunci ȘTERGE {CB-sub}, ADAUGĂ{CB-lateral}

urcă-MC:

dacă {MC-lângă} atunci ȘTERGE {MC-lângă}, ADAUGĂ{MC-pe}

coboară-MC:

dacă {MC-pe} atunci ȘTERGE {MC-pe}, ADAUGĂ{MC-lângă}

apropie-MCB:

dacă {MC-pe, MB-distanță, CB-sub} atunci ȘTERGE {MB-distanță},
ADAUGĂ{MB-aproape}

urcă-pe-cap-MC:

dacă {MC-lângă} atunci ȘTERGE {MC-lângă}, ADAUGĂ{MC-sub}

apucă-MB:

dacă {MB-aproape} atunci ȘTERGE {MB-aproape}, ADAUGĂ{MB-ține}

3.2.2 Lumea cuburilor (blocurilor)

Se consideră o stivă de cuburi, ca cea din Figura 6a. Un braț de robot poate efectua următoarele mișcări: prinde un cub aflat deasupra unei stive sau deplasează un cub aflat în mână pentru a-l așeza fie pe masă, fie pe un alt cub care e liber deasupra. Se cere să se formalizeze problema controlului brațului pentru a aduce blocurile dintr-o configurație inițială într-alta considerată finală, de exemplu cea din Figura 6b. Nu este importantă distanța dintre stive, ci doar ordinea blocurilor în stive. Ca să ținem lucrurile simple, vom considera că pentru controlul brațului, din situația în care acesta se află într-o anumită poziție, este suficient să se indice poziția următoare a lui fără a ne preocupa de traiectorie, distanțe, accelerări ori decelerări – adică amănunte ce apar în mod normal în probleme de control a roboților.

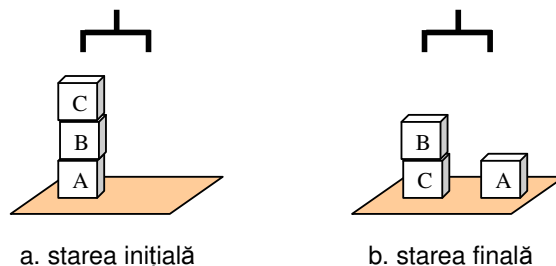


Figura 3.6: Problema cuburilor

Cerința I. Problemă – instanță de problemă

Dintre toate problemele posibile de mutare a unui număr oarecare de cuburi dintr-o configurație inițială într-una finală, problema, așa cum a fost ea definită mai sus, prezintă instanța caracterizată prin exact trei cuburi, o anumită configurație a lor ce reprezintă starea inițială și o anumită configurație ce definește starea finală (precizate în Figura 3.6). Este evident că în cadrul aceleiași probleme o mulțime de alte instanțe de probleme pot fi definite. Conturarea soluțiilor pentru toate acestea ar trebui însă să nu necesite eforturi suplimentare de programare celor necesare rezolvării instanței de față, de exemplu.

Cerința a II-a. Aprecierea a ce reprezintă o stare și a spațiului stărilor

Reflectând la ce anume trebuie să fie o stare în problema de față, vom ajunge inevitabil la dilema: situațiile în care brațul ține un bloc trebuiesc considerate ca stări ori nu. A le ignora înseamnă a lăsa în inventarul stărilor numai poze ale stivelor, ignorând situațiile intermediare în care brațul ține un bloc. Să încercăm să analizăm cele două posibilități.

Există două situații care sunt semnificative înainte de ridicarea unui bloc: blocul se află pe masă sau blocul se află pe alt bloc. Aceleași două situații sunt semnificative după lăsarea unui bloc. Teoretic avem deci $2 \times 2 = 4$ combinații posibile de mișcări: de pe masă pe masă, de pe masă pe un bloc, de pe un bloc pe masă și de pe un bloc pe un alt bloc. Așadar dacă ignorăm posturile intermediare ale brațului ținând blocul ce se transportă, această alegere a stărilor ne va duce la scrierea a patru reguli. Dacă dimpotrivă luăm în considerare ca stări și situațiile în care blocul de transportat este ținut în mâna robotului, vom avea mișcărilor: de pe masă în braț, de pe bloc în braț, din braț pe masă și din braț pe alt bloc, adică tot patru reguli. Rezultă că în acest caz, din punctul de vedere al numărului de reguli ce vor trebui elaborate, nu e importantă alegerea pe care o facem.

Într-un caz general însă, în care numărul stărilor de plecare și respectiv de destinație (v. Figura 5 pentru o reamintire a termenilor) ale mișcărilor ar fi m și n , am avea un număr de $m \times n$ reguli când ignorăm posturile intermediare, dar numai $m+n$ când nu le ignorăm. Este deci mai economic, în număr de reguli ce trebuie proiectate, să încercăm să facem o spargere mai fină a mișcărilor în componente prin considerarea unor stări intermediare, atunci când aceste componente se dovedesc a fi părți constitutive ale mai multor mișcări complexe. O astfel de rafinare nu-și are însă rostul atunci când stările intermediare, astfel puse în evidență, nu se regăsesc în mai multe mișcări pentru că în loc de a micșora numărul total de reguli l-am mări inutil.

Doar din considerentul de a adopta soluția care duce în general la o reprezentare mai compactă, în rezolvarea dată mai jos opțiunea va fi de a considera ca stări intermediare și situațiile în care blocuri se află în mâna robotului. Invităm cititorul să construiască o soluție în care aceste stări intermediare sunt ascunse.

Cerința a III-a. Reprezentarea stărilor

Să privim starea inițială (v. Figura 3.6a) și să încercăm să decidem care sunt elementele esențiale în reprezentarea ei. Așa cum am convenit deja, ceea ce interesează într-o stare este ordinea blocurilor în stive (la un moment dat ar putea să apară mai multe) cât și starea brațului, adică dacă e liber sau dacă dimpotrivă ține un cub. Avem mai multe posibilități de a defini ordinea blocurilor într-o stivă:

- utilizând un vector de nume de blocuri și o convenție asupra capătului stivei: (A, B, C) – cu semnificația, de exemplu, că blocul de pe masă este A, iar cel liber deasupra este C. Reprezentarea aceasta (v. Figura 3.7a) este economică și are avantajul că poate fi ușor modificată pentru a descrie situația în care există mai mult decât o singură stivă. Fiecare stivă poate fi definită printr-un predicat **stiva**. Starea inițială devine **{stiva(A, B, C)}**, iar cea finală **{stiva(C, B), stiva(A)}**;
- precizând prin niște predicate, să zicem **peste** și **sub**, vecinii pe care fiecare bloc îi are dedesubt și respectiv deasupra. Cu această convenție, starea inițială ar fi reprezentată prin setul de predicate: **{peste(A, masă), sub(A, B), peste(B, A), sub(B, C), peste(C, B), sub(C, nil)}**, unde **masă** și **nil** ar semnifica masa și respectiv lipsa unui bloc (v. Figura 3.7b). Această soluție are dezavantajul unei anumite redundanțe în reprezentare. Într-adevăr faptul că blocul B se află peste blocul A este precizat atât de predicatul **sub(A, B)** cât și de **peste(B, A)**;
- redundanța din notația de mai sus sugerează notarea numai a relațiilor dintre blocurile vecine și dintre acestea și masă, precum și a blocurilor care sunt libere deasupra. Cu predicatele **peste** și **liber**, starea inițială poate fi notată: **{peste(A, masă), peste(B, A), peste(C, B), liber(C)}**, iar cea finală: **{peste(C, masă), peste(B, C), liber(B), peste(A, masă), liber(A)}** (v. Figura 7c).

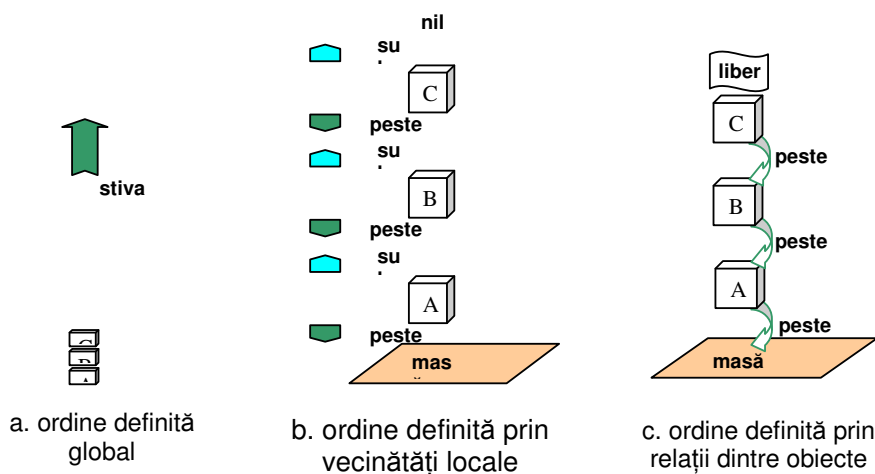


Figura 3.7: Posibilități de reprezentare a așezării relative a blocurilor

În toate aceste cazuri mai trebuie adăugată cunoașterea asupra brațului. Două sunt stările care caracterizează brațul: brațul este gol și brațul ține blocul X (v. Figura 3.8), adică predicatele: **mâna-liberă** și respectiv **mâna-ține(X)**.



Figura 3.8: Reprezentarea configurațiilor mâinii

Vom continua să investigăm în paralel prima și ultima variantă. Astfel pentru prima variantă vom avea:

Starea inițială: {**stiva**(A, B, C), **mâna-liberă**}
 Starea finală: {**stiva**(C, B), **stiva**(A), **mâna-liberă**}

iar pentru ultima variantă:

Starea inițială: {**peste**(A, **masă**), **peste**(B, A), **peste**(C, B), **liber**(C), **mâna-liberă**}
 Starea finală: {**peste**(C, **masă**), **peste**(B, C), **liber**(B), **peste**(A, **masă**), **liber**(A), **mâna-liberă**}

Cerința a IV-a. Reprezentarea regulilor

Cele patru tipuri de mișcări descoperite mai sus, în convenția în care reprezentăm ca stări și situațiile în care blocurile sunt ținute în mâna robotului, duc în mod direct la scrierea a patru reguli, în forma pe care am convenit-o deja în secțiunea 3.1.4, adică: **dacă...atunci...**, unde în partea **dacă** vom descrie condiții pentru recunoașterea stărilor de plecare ale regulii, iar în partea **atunci**, vom descrie acțiuni. Aceste acțiuni trebuie să specifice modificările de realizat în starea de plecare pentru ca ea să se transforme în starea de sosire. Ele vor consta din eliminarea unor predicate din descrierea stării de plecare și includerea altora, deci vor avea forma **șterge...adaugă...**

În varianta din Figura 3.7a avem următoarele reprezentări pentru reguli (v. și Figura 3.9):

ia-de-pe-bloc(X,Y):

dacă {**stiva**(*, Y, X), **mâna-liberă**} **atunci** ȘTERGE{**stiva**(*, Y, X), **mâna-liberă**} ADAUGĂ{**stiva**(*, Y), **mâna-ține**(X)}

unde **stiva**(*, X, Y) semnifică o stivă a cărei parte de la bază este oarecare.

ia-de-pe-masă(X):

dacă {stiva(X), mâna-liberă} atunci ȘTERGE{stiva(X), mâna-liberă} ADAUGĂ{mâna-ține(X)}

pune-pe-bloc(X,Y):

dacă {mâna-ține(X), stiva(*, Y)} atunci ȘTERGE{mâna-ține(X), stiva(*, Y)} ADAUGĂ{stiva(*, Y, X), mâna-liberă}

pune-pe-masă(X):

dacă {mâna-ține(X)} atunci ȘTERGE{mâna-ține(X)} ADAUGĂ{stiva(X), mâna-liberă}

iar pentru varianta de reprezentare din Figura 7c, următoarele reguli:

ia-de-pe-bloc(X,Y):

dacă {peste(X, Y), liber(X), mâna-liberă} atunci ȘTERGE{peste(X, Y), liber(X), mâna-liberă} ADAUGĂ{liber(Y), mâna-ține(X)}

ia-de-pe-masă(X):

dacă {peste(X, masă), liber(X), mâna-liberă} atunci ȘTERGE{peste(X, masă), liber(X), mâna-liberă} ADAUGĂ{mâna-ține(X)}

pune-pe-bloc(X,Y):

dacă {mâna-ține(X), liber(Y)} atunci ȘTERGE{mâna-ține(X), liber(Y)} ADAUGĂ{peste(X, Y), liber(X), mâna-liberă}

pune-pe-masă(X):

dacă {mâna-ține(X)} atunci ȘTERGE{mâna-ține(X)} ADAUGĂ{peste(X, masă), liber(X), mâna-liberă}

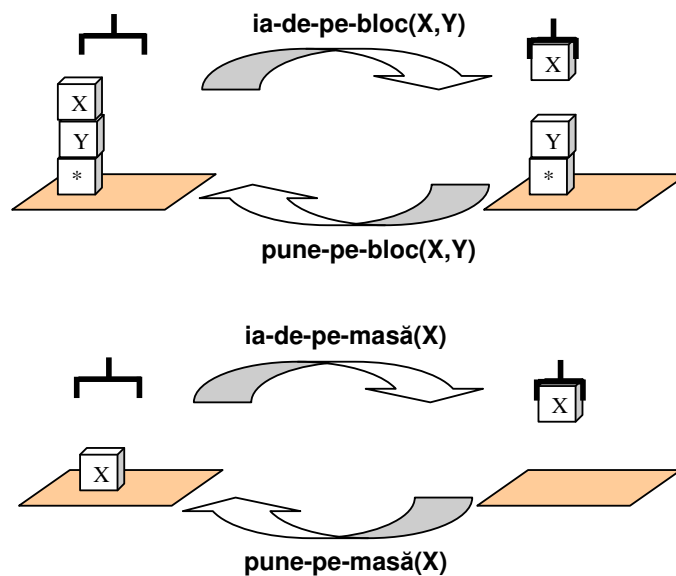


Figura 3.9: Regulile

3.3 Controlul sistemelor de producție

3.3.1 Strategii irevocabile

O strategie irevocabilă este una în care orice mișcare în spațiul stărilor este ireversibilă. Cale de întoarcere nu există. O strategie irevocabilă aparent merge la sigur. Dar imposibilitatea de a mai continua la un moment dat duce, inevitabil, la oprirea căutării și deci eșec. Este ca și cum aș cunoaște deja soluția și atunci mă îndrept direct spre ea. Dar ce sens mai are căutarea dacă cunosc deja soluția? Distincția provine din gradul de cunoaștere a soluției problemei: cunoaștere globală (imaginați-vă găsirea drumului într-un labirint privit dintr-un balon) față de cunoaștere locală (labirintul privit de un om aflat în el). Un sistem de IA care lucrează cu o strategie irevocabilă are o cunoaștere locală a problemei, al aplicând o manieră generală de comportament în toate situațiile similare.

- **Metoda ascensiunii (*hill climbing*)**

Numele metodei este dat de analogia cu maniera de cățărare pe un deal a unui orb. El nu e capabil să vadă vârful dealului unde trebuie să ajungă dar poate găsi în orice moment, încercând la fiecare pas în jurul lui cu bastonul, direcția care-l face să urce (Figura 3.10). Metoda ascensională caută să atingă starea caracterizată de valoarea maximă a funcției euristice. Ea nu garantează găsirea unei soluții din cauză că anumite probleme pot avea maxime locale sau platouri, după cum se sugerează în Figura 3.11.

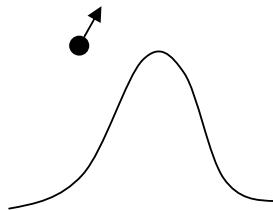


Figura 3.10: Metoda Hill-climbing

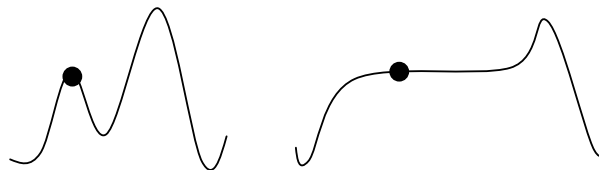


Figura 3.11: Maximele locale, iar uneori și platourile, opresc înaintarea spre soluție

```

procedure hill-climbing(initial-state)
begin
  current-state <- initial-state;
  if (current-state e stare finală) return current-state;
  while(all-new-neighbour-states <- setul stărilor ce pot fi
obținute din current-state prin operatorii aplicabili ei)
  { elimină din setul all-new-neighbour-states toate stările
    deja vizitate;
    sortează all-new-neighbour-states în ordinea descrescătoare
    a valorilor funcției de cost;
    elimină din all-new-neighbour-states toate stările de cost
    mai mic decât current-state;
    if (all-new-neighbour-states ≠ ∅)
    { current-state <- prima clasată în
      all-new-neighbour-states);
      if (current-state e stare finală) return current-state;
    }
    else return FAIL;
  }
  return fail;
end

```

Iterația datorată lui **while** semnifică aici tranzitarea pe rând în stări consecutive către soluție. În această formă se permite deplasarea pe un platou, în sensul că o stare oarecare de aceeași valoare ca cea curentă poate fi selectată. Dacă însă la pasul al treilea din iterație se modifică “mai mic” cu “mai mic sau egal”, atunci înaintarea pe platouri este invalidată. O variantă recursivă a acestui algoritm este următoarea:

```

procedure rec-hill-climbing(current-state)
begin
  if (current-state e stare finală) return current-state;
  all-new-neighbour-states <- setul stărilor ce pot fi obținute
    din current-state prin operatorii aplicabili ei;
  elimină din setul all-new-neighbour-states toate stările deja
    vizitate;
  sortează all-new-neighbour-states în ordinea descrescătoare a
    valorilor funcției de cost;
  elimină din all-new-neighbour-states toate stările de cost mai
    mic decât current-state;
  if (all-new-neighbour-states ≠ ∅)
  { current-state <- prima clasată în all-new-neighbour-states);
    rec-hill-climbing(current-state);
  }
  else return FAIL;
end

```

Amorsarea rulării, în acest caz, se face prin apelul `rec-hill-climbing(initial-state)`.

Să observăm în aceste proceduri:

- valorile întoarse sunt fie o stare finală fie `FAIL`, după cum se găsește ori nu o soluție;
- deși generale, procedurile utilizează în câteva locuri informație specifică problemei: recunoașterea unei stări ca fiind finală și funcția de evaluare a stării;
- în ambele variante, prima stare „mai bună” decât cea curentă este aleasă. Introducerea sortării stărilor posibil de atins din starea curentă face ca dintre toate stările vecine să fie aleasă „cea mai bună”. Din acest motiv metoda se mai numește și **a gradientului maxim** (*stiffest gradient*).

A decide când o stare este „mai bună” decât alta presupune existența unui criteriu de comparare a stărilor între ele. Desigur afirmația dacă o stare este mai promițătoare decât alta poate fi hazardantă, pentru că adesea o stare care prezintă un potențial mic se poate dovedi să conducă căutarea spre zone foarte interesante ale spațiului stărilor. De aceea, de obicei această funcție se bazează pe o **euristică**¹². Primul lucru ce trebuie avut în vedere atunci când se apelează la metoda *hill-climbing* este construirea unei astfel de funcții, lucru nu întotdeauna ușor.

Vom exemplifica o alegere a funcției euristice de calcul a costului stărilor pe o instanță a problemei *8-Puzzle* (Figura 3.12).

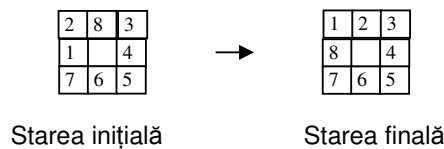


Figura 3.12: O instanță a problemei 8-puzzle

Vom presupune că lucrăm cu **operatorii** cunoscuți: **mut•-blanc-sus**, **mut•-blanc-jos**, **mut•-blanc-dreapta** și **mut•-blanc-stânga**. Vom considera că funcția euristică **f** atribuie stării curente ca scor numărul de piese aflate în poziții finale. Cu acest criteriu, avem:

$$f(\text{stare inițială}) = 5; f(\text{stare finală}) = 8.$$

¹² O euristică este o metodă care, deși nu întotdeauna riguroasă, poate fi de ajutor pentru găsirea unei soluții.

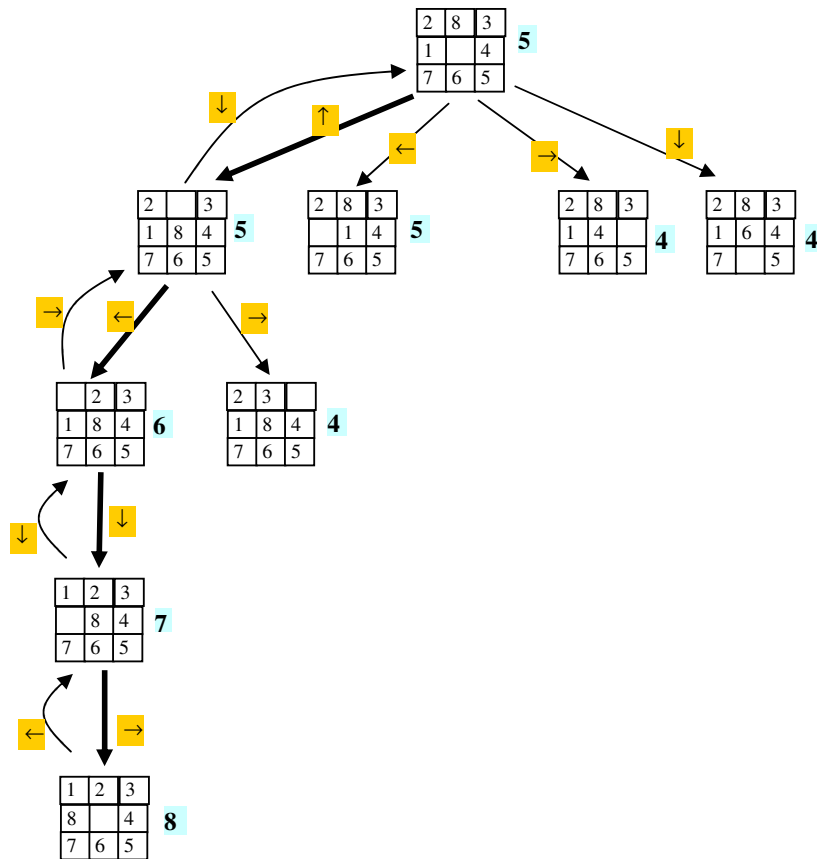


Figura 3.13: O parte dintr-un arbore de căutare pentru o instanță a problemei 8-puzzle

Figura 3.13 arată o parte a arborelui de căutare pentru acest exemplu. Valorile funcției **f** sunt notate lângă desenele stărilor. Arcele semnifică tranziții. Săgețile din micile pătrate ce însoțesc arcele sugerează numele operatorilor aplicați pentru realizarea respectivelor tranziții. Figura 3.14 exemplifică un caz de oprire pe un maxim local. Deși există un drum până la soluție, ea nu poate fi atinsă pentru că stările învecinate stării de scor 7 (în elipsă) sunt toate mai slabe. Pe această figură s-a renunțat la notarea tranzițiilor inverse către stări prin care s-a mai trecut. Figura 3.15 marchează stările poziționate ascendent în funcție de valorile lor. Se observă că nu există un drum în figură care să ajungă în starea de valoare maximă 8. Drumul punctat către starea finală sugerează că soluția parcurge alte zone ale spațiului stărilor decât cea detaliată în exemplu.

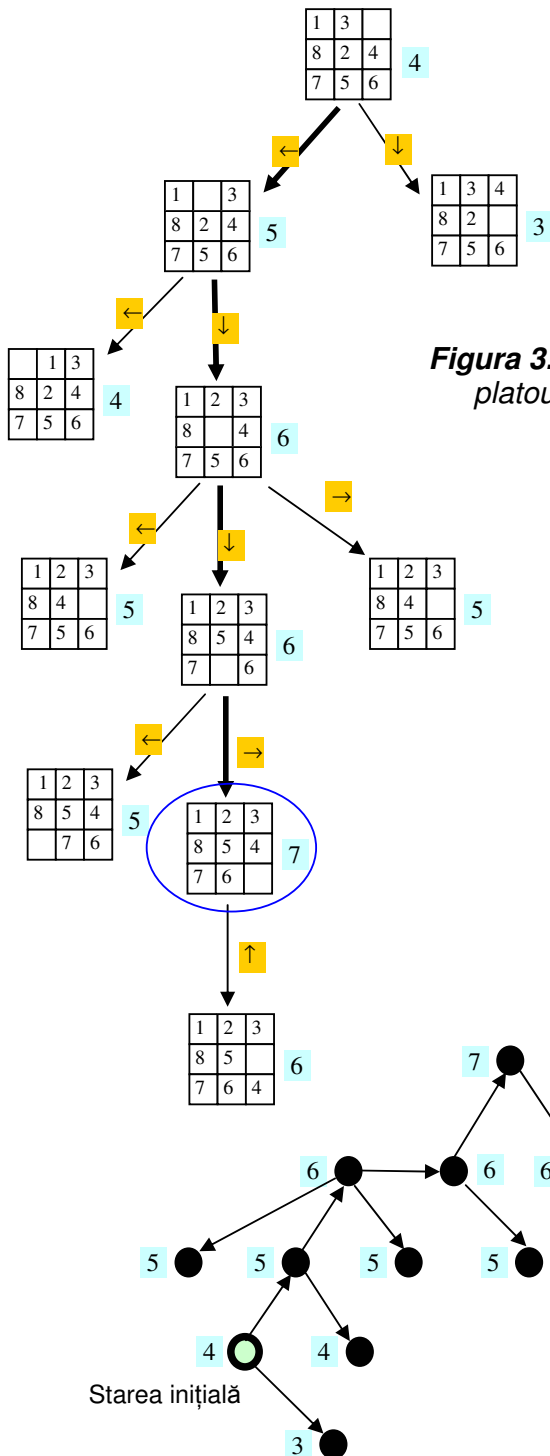


Figura 3.14: Un exemplu de evoluție spre un platou (6) urmat de un maxim local (7)

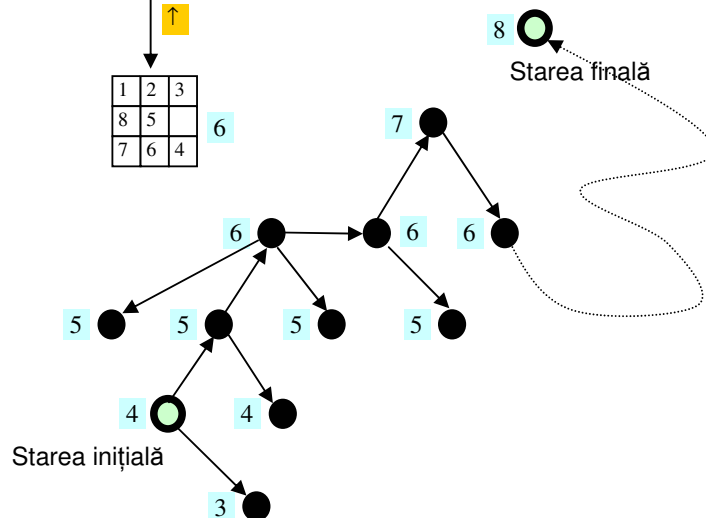


Figura 3.15: Maximul local față de starea finală

3.3.2 Strategii tentative

O strategie tentativă este una șovăitoare: o mișcare ce se dovedește greșită poate fi îndreptată (Cristea, 2002). Deplasările în spațiul stărilor nu sunt fără întoarcere. Dacă am ajuns într-un punct mort, de exemplu într-o stare care nu mai are succesori, pot să "iau urma îndărăt" pentru a face o altă mișcare dintr-o poziție anterioară celei curente și în care am mai fost. În felul acesta se pot releva noi căi către soluție.

- **Metoda ascensională cu revenire (*backtracking hill climbing*)**

Caracteristic pentru această metodă este faptul că utilizează o stivă în care memorează la fiecare pas stările vecine diferite de cea în care efectuează tranziția. Stiva este apoi folosită pentru a recupera din ea alte stări în cazurile în care căutarea se "agață" în "fundături" de genul minimelor locale sau a stărilor fără succesori. Caracteristica de "întoarcere" sau "luare a urmei înapoi", care este sugerată de numele metodei, nu trebuie înțeleasă strict în sensul revenirii, în situațiile de blocare, în stări prin care s-a mai trecut, ci de memorizare a unor căi care se ignoraseră la momentul deciziilor, pentru a putea fi reluate la momente ulterioare. Figura 3.16 lămurește.

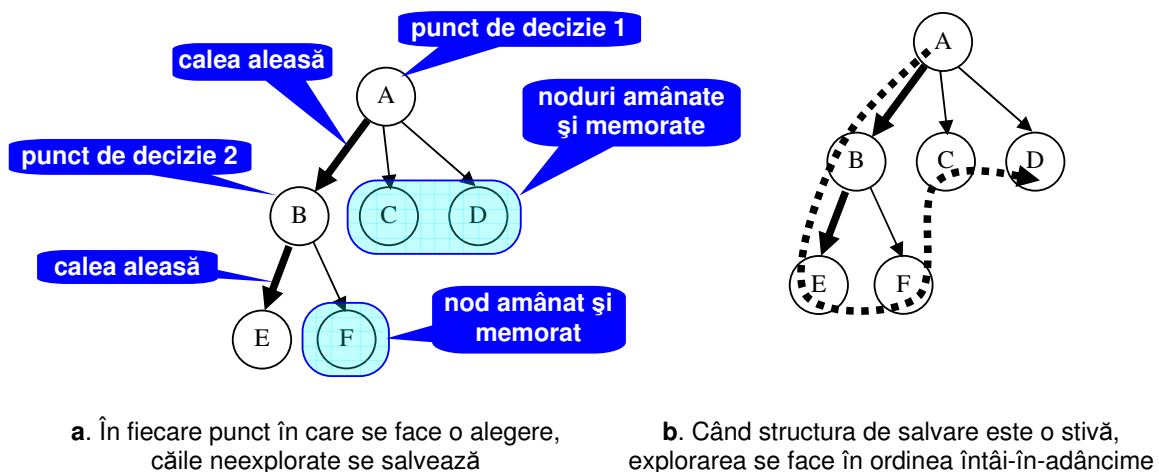


Figura 3.16: "Întoarcerea" în cazuri de blocare înseamnă reconsiderarea unor căi salvate anterior

```

procedure backtracking-hill-climbing(initial-state)
begin
  stack = initial-state;
  while (stack)
  { current-state = pop(stack);
    if (current-state e stare finală) return current-state;
    all-new-neighbour-states <- setul stărilor ce pot fi
      obținute din current-state prin operatorii aplicabili ei);

```

```

elimină din setul all-new-neighbour-states toate stările
deja vizitate;
if (all-new-neighbour-states  $\neq \emptyset$ )
{ sortează all-new-neighbour-states în ordinea
  descrescătoare a valorilor funcției de cost;
  current-state  $\leftarrow$  prima stare clasată în
    all-new-neighbour-states);
if (current-state e stare finală) return current-state;
else
  { all-new-neighbour-states  $\leftarrow$  all-new-neighbour-states -
    {current-state};
    stack  $\leftarrow$  push(all-new-neighbour-states, stack);
  }
}
return FAIL;
end

```

Se observă că ordonarea noilor stări găsite se face înainte de introducerea lor în stivă. Înseamnă că prima stare ce va fi considerată în continuarea stării curente este cea din care se poate ajunge cel mai sus din starea curentă, cu toate că e posibil ca în alte părți ale spațiului de căutare să se afle stări (memorate în stivă) care să fie mai atrăgătoare (scoruri mai bune). De asemenea, să observăm că din *all-new-neighbour-states* n-au mai fost eliminate stările de cost mai mic decât *current-state*, ca în varianta fără revenire. Acest lucru înseamnă că atunci când se ajunge într-un punct de maxim local sau de platou, dacă alte stări noi pot încă fi atinse din starea curentă, ele se introduc în stivă chiar dacă se află „mai jos” decât starea curentă în drumul spre soluție. Acest lucru permite abordarea altor căi neexplorate și „părăsirea” vârfurilor locale sau a platourilor. Interzicerea introducerii în stivă a stărilor prin care s-a mai trecut împiedică intrarea în bucle infinite.

Varianta de mai jos este cea recursivă. Parametrul de intrare al funcției este chiar stiva de stări. Intrarea se face cu un apel în care pe poziția stivei este plasată starea inițială, adică: `rec-backtracking-hill-climbing(push(initial-state, \emptyset))`:

```

procedure rec-backtracking-hill-climbing(stack)
begin
  if null(stack) then return FAIL;
  current-state = pop(stack);
  if (current-state e stare finală) return current-state;
  all-new-neighbour-states  $\leftarrow$  setul stărilor ce pot fi obținute
    din current-state prin operatorii aplicabili ei;
  elimină din setul all-new-neighbour-states toate stările deja
    vizitate;
  sortează all-new-neighbour-states în ordinea descrescătoare a
    valorilor funcției de cost;
  elimină din all-new-neighbour-states toate stările de cost mai
    mic decât current-state;
  rec-backtracking-hill-climbing(push(all-new-neighbour-states,
    stack));
end

```

Funcțiile `pop` și `push` au efect lateral asupra argumentului: în urma apelului, parametrul este modificat, respectiv stiva este decrementată sau incrementată. Valorile întoarse de ele sunt tot stive.

- **Metode de căutare sistematică (*brute-force*)**

Metodele de căutare sistematică (neinformate) investighează în totalitate spațiul stărilor, plecând din starea inițială, pentru găsirea unei căi către starea (stările) finală (finale). Aceste metode pot fi aplicate când spațiul stărilor este rezonabil de mic, în așa fel încât să ne putem permite, în extremis, parcurgerea lui exhaustivă.

- **Căutare întâi-în-adâncime (*depth-first search*)**

În căutarea întâi-în-adâncime se utilizează o stivă pentru memorarea stărilor posibil de atins din starea curentă. Prin aceasta, înainte de a căuta în frații unui nod (stare), întâi fiii acestuia sunt căutați. Această ordine este dictată de faptul că, după vizitarea unui nod, întâi acesta este eliminat din stivă și apoi fiii acestuia sunt introduși în stivă. Datorită caracteristicii *last-in-first-out* ei vor fi parcurși înaintea altor noduri aflate deja acolo. Când spațiul stărilor este un arbore, căutarea întâi-în-adâncime parcurge arborele cu predilecție în adâncime, de unde și numele metodei.

```

procedure depthFirstSearch(root)
begin
    stack <- push(root,  $\emptyset$ );
    while (stack nu e goală)
    { node <- pop(stack);
      if goal(node) then return node;
      else push(succesorii lui node, stack);
    }
    return FAIL;
end

```

Această procedură testează soluția în fiecare nod al arborelui. Dacă doar nodurile frunză sunt soluții posibile, procedura se schimbă astfel:

```

procedure leafDepthFirstSearch(root)
begin
    stack <- push(root,  $\emptyset$ );
    while (stack nu e goală)
    { node = pop(stack);
      if not(leaf(node)) push(succesorii lui node, stack);
      else if goal(node) then return node;
    }
    return FAIL;
end

```

Un dezavantaj al căutării în adâncime îl constituie posibilitatea ca algoritmul să nu se termine în cazul arborilor infiniți. Desigur în practică nu se lucrează cu arbori infiniți, dar problema rămâne pentru că e încă posibil ca algoritmul să

genereze un număr foarte mare de noduri chiar dacă soluția se află pe un nivel superior, dar pe o ramură care nu e atinsă (de exemplu, în partea dreaptă a arborelui dacă parcurgerea fiilor se face de la stânga la dreapta).

- Căutare întâi în lărgime (*breadth-first search*)

La căutarea-în-lărgime parcurgerea avansează de-a lungul conturilor de adâncime egală. În felul acesta, dacă o soluție există, algoritmul garantează găsirea căii celei mai scurte până la un nod final.

```

procedure breadthFirstSearch(root)
begin
    queue <- in(root,  $\emptyset$ );
    while (queue nu e goală)
    { node <- out(queue);
      if goal(node) then return node;
      else in(succesorii lui node, queue);
    }
    return FAIL;
end

```

După cum se constată algoritmul de căutare în lărgime nu diferă de cel de căutare în adâncime decât prin structura de date utilizată, coadă în loc de stivă. Prin *in* și *out* s-au notat aici operațiile de introducere și extragere din coadă (similarele lui *push* și *pop* de la structura de stivă). O căutare în lărgime are mai mari șanse să găsească repede nodul soluție aflat pe cel mai înalt nivel, decât o căutare în adâncime.

• Metode de căutare euristică

Metodele de căutare euristică se bazează pe aprecierea valorii unei stări ori a distanței la care se află ea față de o stare finală. Anumite euristici (generate de existența unor cunoștințe asupra problemei) pot, în anumite cazuri, reduce efortul de căutare, dar, în același timp, metodele de acest tip nu mai garantează găsirea căii minime care era garantată la căutarea întâi-în-lărgime. Interesul este de a reduce atât dimensiunea soluției (ca lungime a ei sau cost), cât și efortul de căutare pentru găsirea unei soluții. Vom spune că o metodă *P* are o **putere euristică** mai mare decât o alta *Q* dacă combinația dintre costul soluției și al găsirii ei în cazul lui *P* este mai mic decât în cazul lui *Q*. Pe de altă parte, dacă cel puțin o soluție există, atunci o soluție trebuie găsită.

În cele ce urmează, pentru că spațiul stărilor este organizat ca arbore sau graf, stările vor fi asociate nodurilor acestor structuri.

- Căutarea cel-mai-bun-întâi (*best-first*)

Dintr-un anumit punct de vedere, metoda de căutare **cel-mai-bun-întâi** (*best-first*) poate fi considerată o combinație a metodelor *depth-first* și *breadth-first*. Ca și primele două și aici avem o structură care memorează nodurile ce urmează a fi vizitate. Această structură este aici o listă, fără organizarea particulară de stivă sau coadă. Spre deosebire însă de metodele menționate, în cazul metodei cel-mai-bun-întâi trebuie făcută o apreciere a unui nod, printr-un scor atașat lui, ce caracterizează “distanța” până la un nod final, sau dificultatea de a atinge soluția. Astfel văzut, un scor mic caracterizează o stare apropiată de final, iar un scor mare – una depărtată. La fiecare pas lista se sortează în ordinea crescătoare a scorurilor nodurilor ei, ceea ce permite să se aleagă pentru explorarea ce urmează cel mai promițător (mai aproape de soluție, sau mai ușor de atins) nod dintre cele ce se învecinează cu nodurile deja vizitate. Ca și în cazul primelor două metode, metoda poate fi aplicată în egală măsură parcurgerii arborilor ori grafurilor. Metoda este descrisă de procedura `bestFirstSearch()` iar Figura 16 prezintă un exemplu de aplicare a ei pe un arbore.

```

procedure bestFirstSearch(root)
begin
  list <- include(root,  $\emptyset$ );
  while (list nu e goală)
    { node <- get-first(list);
      if goal(node) then return node;
      else
        { include(succesorii lui node, list);
          sortează crescător list;
        }
      }
  return FAIL;
end

```

În această procedură: `include(x, y)` este o funcție care introduce un nod sau o listă de noduri *x* în lista *y*, `get-first(x)` – întoarce primul element al listei *x*, ștergând în același timp din lista *x* primul ei element.

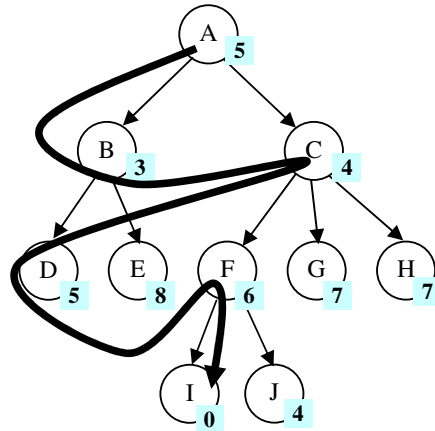


Figura 3.17: Vizitarea unui arbore în ordinea cel-mai-bun-întâi

Cifrele de lângă nodurile din Figura 3.17 și cele ce însoțesc ca puteri numele de noduri în Tabelul 3.1 reprezintă aprecieri ale costurilor până la atingerea soluției.

Tabelul 3.1: Simularea parcurgerii arborelui din Figura 3.17 în ordinea cel-mai-bun-întâi

Pasul	Nodul vizitat	Descendenți	Lista sortată
0			(A)
1	A	B ³ , C ⁴	(B ³ , C ⁴)
2	B	D ⁵ , E ⁸	(C ⁴ , D ⁵ , E ⁸)
3	C	F ⁶ , G ⁷ , H ⁷	(D ⁵ , F ⁶ , G ⁷ , H ⁷ , E ⁸)
4	D	-	(F ⁶ , G ⁷ , H ⁷ , E ⁸)
5	F	I ⁰ , J ⁴	(I ⁰ , J ⁴ , G ⁷ , H ⁷ , E ⁸)
6	I	nod soluție	

Ordinea nodurilor parcurse nu corespunde cu calea. Dar calea poate fi reconstituită din noduri parcurse. Expl.

Următoarea procedură reprezintă a adaptare a precedentei pentru cazul în care căutarea se face într-un spațiu al soluțiilor descris de un graf, iar nu de un arbore. Se utilizează două liste de noduri:

- *open*: fosta *list* din procedura aplicată pe arbori, care memorează noduri care au fost generate și pentru care s-a calculat funcția euristică aplicată asupra lor, dar care nu au fost încă vizitate. Elementele în această listă sunt întotdeauna sortate în ordinea crescătoare a valorilor funcției euristice;

- *closed*: lista nodurilor care au fost deja examinate. Ea e necesară în cazul structurii de graf, întrucât, la fiecare generare a unui nou nod, lista este căutată și nodul este generat numai dacă el nu există deja în *closed*.

```

procedure bestFirstSearchOnGraph(start)
begin
  open <- include(start,  $\emptyset$ );
  closed <-  $\emptyset$ ;
  while (open nu e goală)
  { node <- get-first(open);
    if goal(node) then return node;
    else
      { closed <- include(node, closed);
        succ <- lista succesorilor lui node;
        new-succ <- succ \ closed;
        new-succ <- new-succ \ open;
        /* new-succ reprezintă lista succesorilor lui node care
           nu sunt în closed și nici în open, deci nu au fost încă
           vizitați și nici puși în așteptare. */
        open <- include(new-succ, open);
        sortează crescător open;
      }
    }
  return FAIL;
end

```

Diferența față de alg A.

- Recuperarea soluției în metodele de căutare sistematică și euristică pe arbori

În toate procedurile prezentate până acum am considerat că ne interesează în ieșire doar nodul final și nu calea până la el. Această presupunere este însă în contradicție cu cele afirmate deja relativ la ce înseamnă o soluție la o problemă de IA spre deosebire de o problemă din afara spațiului IA (v. secțiunea 3.1.1). Atunci când spațiul de căutare are structura unui **arbore**, generarea căii este simplă, dacă fiecare nod în structură are memorată o legătură către părinte. Practic, parcurgând înapoi drumul din nodul soluție până în nodul rădăcină se obține inversa unei căi către soluție. Atunci însă când arborele nu este explicitat ca o structură, el fiind generat ad-hoc la parcurgerea spațiului, situația se schimbă. Calea spre soluție trebuie generată o dată cu efectuarea căutării. În cele ce urmează vom relua algoritmi de căutare exhaustivă pe arbori, prezentați până acum, cu această grijă în minte. Vom prefera să lucrăm pe variantele recursive ale acestor algoritmi. Mai întâi, varianta recursivă a căutării întâi în adâncime:

```

procedure recDepthFirstSearch(stack)
begin
  if empty(stack) then return FAIL;
  node <- pop(stack);
  if goal(node) then return node;

```

```

    recDepthFirstSearch(push(succesorii lui node, stack));
end

```

Primul apel trebuie să fie: `recDepthFirstSearch(push(root, \emptyset))`. Similar, varianta recursivă a procedurii de căutare întâi în lărgime este:

```

procedure recBreadthFirstSearch(queue)
begin
    if empty(queue) then return FAIL;
    node <- out(queue);
    if goal(node) then return node;
    recBreadthFirstSearch(in(succesorii lui node, queue));
end

```

cu apelul inițial `recBreadthFirstSearch(in(root, \emptyset))`. Varianta recursivă a funcției de căutare cel-mai-bun-întâi este:

```

procedure recBestFirstSearch(list)
begin
    if empty(list) then return FAIL;
    node <- get-first(list);
    if goal(node) then return node;
    recBestFirstSearch(sort(include(succesorii lui node, list)));
end

```

cu apelul inițial `recBestFirstSearch(include(root, \emptyset))`. Prin funcția `sort`, aplicată aici unei liste, înțelegem sortarea crescătoare a scorurilor nodurilor care sunt elementele listei. În toate aceste proceduri `empty` este o funcție care întoarce TRUE dacă argumentul (stivă, coadă sau simplă listă) este vid.

Și acum să construim variantele acestor algoritmi care se preocupă totodată de recuperarea căilor către soluție. Iată-le, în aceeași ordine:

```

procedure recGetPathDepthFirstSearch(stack)
begin
    if empty(stack) then return FAIL;
    (node, path) <- pop(stack);
    if goal(node) then return path;
    succ <- succesorii lui node;
    while (n <- one of succ) push(<n, path  $\circ$  n>, stack);
    recGetPathDepthFirstSearch(stack);
end

```

Diferențele acestei variante de căutare întâi-în-adâncime față de precedentă sunt următoarele:

- elementele stivei sunt acum formate din pereche de genul `<nod, cale>`, unde `nod` este un nod al arborelui, iar `cale` este o secvență de noduri, configurând o soluție parțială, între rădăcină și `nod`;
- singurul apel recursiv al funcției se detaliază acum în două apeluri pentru că trebuie făcută diferența între cazurile în care nodul curent are ori nu descendenți. Dacă el nu are descendenți (`succ == \emptyset`), atunci calea nu se

modifică, pentru că practic stiva reculează cu nodul curent, ce fusese scos prin operația `pop`. Dacă are descendenți, atunci în stivă se includ perechi formate din fiecare nod descendent și calea corespunzătoare la care se adaugă nodul descendent (prelungirea căii vechi cu nodul nou este notată $path \circ n$, pentru a sugera o concatenare);

- valoarea întoarsă este, de data asta, o cale iar nu un simplu nod.
Apelul inițial trebuie acum să fie de forma:

`recGetPathDepthFirstSearch(push<root, root>, \emptyset)`. În aceeași manieră, funcția de căutare întâi-în-lărgime devine:

```
procedure recGetPathBreadthFirstSearch(queue)
begin
  if empty(queue) then return FAIL;
  (node, path) <- out(queue);
  if goal(node) then return path;
  succ <- succesorii lui node;
  if (succ ==  $\emptyset$ ) recGetPathBreadthFirstSearch(queue);
  else while (n <- one of succ)
    recGetPathBreadthFirstSearch(in(<n, path  $\circ$  n>, queue));
end
```

cu apelul inițial `recGetPathBreadthFirstSearch(in<root, root>, \emptyset)`. În fine, varianta ce se preocupă de generarea soluției pentru funcția de căutare cel-mai-bun-întâi devine:

```
procedure recGetPathBestFirstSearch(list)
begin
  if empty(list) then return FAIL;
  (node, path) <- get-first(list);
  if goal(node) then return path;
  succ <- succesorii lui node;
  while (n <- one of succ) include(<n, path  $\circ$  n>, list);
  recGetPathBestFirstSearch(sort(list));
end
```

al cărei apel inițial trebuie să fie exprimat ca: `recGetPathBestFirstSearch(include<root, root>, \emptyset)`. În această variantă a procedurii de căutare cel-mai-bun-întâi, se presupune că funcția `sort` sortează perechi $\langle nod, cale \rangle$ pe baza strict a scorurilor nodurilor, deci fără a se preocupa de căile atașate.

- **Recuperarea soluției în metodele de căutare euristică pe grafuri. Algoritmul A^***

Mai avem de rezolvat o problemă, și anume, cum se face recuperarea căii atunci când structura e un graf iar nu un arbore. Problema revine la calculul drumurilor de cost minim în grafuri, considerată o problemă clasică de căutare în grafuri, dar, în același timp, și o problemă de IA. În general, în problemele de acest

gen, aprecierea costurilor este făcută de o funcție euristică f care exprimă costul ca o sumă a două valori:

g – un cost al traversării grafului din nodul start până în nodul curent, și

h – un cost al estimării traversării grafului din nodul curent până într-un nod final:

$$f = g + h$$

Toate valorile funcțiilor g și h trebuie să fie pozitive. Evident că precizia funcției euristice reflectă cantitatea de informație pe care ea o încorporează relativ la domeniul problemei. O funcție h identic nulă reflectă lipsă totală de informație asupra problemei (cazul în care nu se poate face nici o estimare asupra apropierii de soluție a unei stări).

Următoarea procedură este o variantă a căutării *best-first*, cunoscută sub numele de **algoritmul A** (Hart, et al., 1968). Procedura generează atât o parte a grafului stărilor problemei cât și un subset al său numit **arbore de căutare** (așadar, în care fiecare nod are un unic părinte). La fiecare pas al iterației exterioare, nodurile din *open* reprezintă noduri frunză ale arborelui de căutare. În derularea algoritmului, nu numai că arborele de căutare se extinde în graful generat dar își poate schimba și forma în așa fel încât unele noduri își modifică părinții. Când o frunză a arborelui de căutare atinge un nod final, atunci căutarea se termină și pentru că fiecare nod al acestui arbore își cunoaște părintele, soluția este dată de calea inversă de la nodul final la nodul de start.

procedure A(*start-node*)

begin

start-node.back-score = 0;

start-node.forth-score = euristics(*start-node*);

 /* Presupunem că atașăm fiecărui nod trei fațete:

 /* back-score – costul căii de la rădăcină la nod;

 /* forth-score – aprecierea costului de la nod la destinație;

 /* father – nodul lui părinte în drumul spre soluție.

 open <- {*start-node*};

 closed <- Ø;

 while (open)

 { open <- sort(open);

 /* Sortarea se face în ordinea crescătoare a valorilor lui

 /* f (respectiv *node*.back-score + *node*.forth-score).

current-node <- first(open);

 open = open \ {*current-node*};

 closed <- closed ∪ {*current-node*};

if goal(*current-node*)

return generatePath(*current-node*, *start-node*);

 /* Funcția generatePath() generează soluția ca inversul

 /* drumului de la *current-node* la *start-node*, cale dată

 /* de fațetele father ale nodurilor din arborele parcurs din

graf.

pairs <- generateSuccessors(*current-node*);

 /* Vom considera că funcția generateSuccessors generează

 /* perechi (*node*,*score*) formate dintr-un nod și scorul

 /* tranziției din *current-node* în respectivul nod.

while (*pairs*)

 { (*node*,*score*) <- first(*pairs*);

```

if (node  $\in$  open  $\cup$  closed)
{ /* Înseamnă că node a mai fost deja generat sau a
  /* fost chiar procesat.
  if (current-node.back-score + score <
                                node.back-score)
  { /* Dacă scorul calculat până la acest nod deja
    /* vizitat este mai mic pe această cale decât pe
    /* cea din care a fost el generat anterior,
    /* atunci i se actualizează scorul și i se
    /* schimbă părintele.
    node.back-score  $\leftarrow$  current-node.back-score +
                                score;
    node.father  $\leftarrow$  current-node;
    updateDescScores(node);
    /* Actualizarea scorurilor descendenților lui
    /* node din graf care au mai fost vizitate se face
    printr-o parcurgere
    /* depth-first a nodurilor posibil de atins
    /* plecând de la node. Algoritmul se stinge în
    /* nodurile care nu au succesori cât și în cele
    /* care nu au fost atinse pe o cale care l-a inclus
    pe node.
    }
  }
else
{ node.father  $\leftarrow$  current-node;
  node.back-score  $\leftarrow$  current-node.back-score +
                                score;
  node.forth-score  $\leftarrow$  heuristics(node);
  open  $\leftarrow$  open  $\cup$  {node};
  /* Funcția heuristics() apreciază o valoare a căii
  /* de la nodul argument până la scop.
  }
}
return FAIL;
end

```

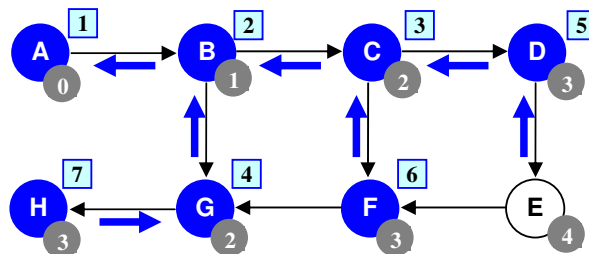


Figura 3.18: Un spațiu al stărilor reprezentat ca graf în care arborele de căutare evoluează fără reveniri

Tabelul 3.2, redă pașii aplicării algoritmului pe spațiul stărilor din Figura 3.18. Graful stărilor este indicat prin noduri, etichetate cu litere, și arce care unesc nodurile și care semnifică tranziții. Nodul start este aici A și nodul final – H. Vom considera că o tranziție costă o unitate. Asta înseamnă că o soluție (o cale care unește nodul start de nodul destinație) costă exact cât lungimea ei (în număr de arce). Numerele în chenare pătrate de deasupra și în dreapta nodurilor grafului în Figura 3.18 indică pașii în care respectivele noduri au fost prelucrate (drept *current-node*). Numerele din cercurile cu fond întunecat plasate în dreapta și jos față de noduri indică costurile căilor până la ele din nodul de start. Legăturile către nodurile părinți în arborele de căutare rezultat sunt indicată de săgețile plasate lângă arce (întotdeauna în sens invers arcului). Nodurile întunecate sunt cele care au ajuns în lista *closed*.

Pentru exemplul de față vom considera o funcție euristică constantă (evident, o astfel de funcție este complet neinteresantă într-un caz concret, pentru că nu diferențează între noduri și ca urmare nu influențează sortarea din algoritm; o astfel de funcție transformă algoritmul într-unul neinformațional). În exemplul nostru toate nodurile vor avea valoarea *forth-score*, dată de această funcție, egală cu 7.

Tabelul 3.2: Trasarea unui exemplu în care nu se revine asupra deciziilor (pe spațiul stărilor din Figura 3.18)

Pas	<i>current-node</i>	<i>pairs: { (node, score) }</i>	<i>open</i>	<i>closed</i>
0			(A)	∅
1	A (A.f=nil, A.bs=0, A.fs=7)	((B,1)) B.f=A, B.bs=1, B.fs=0	(B)	(A)
2	B	((C,1),(G,1)) C.f=B, C.bs=2, C.fs=0 G.f=B, G.bs=2, G.fs=0	(C,G) sortată: (C,G)	(A,B)
3	C	((D,1),(F,1)) D.f=C, D.bs=3, D.fs=0 F.f=C, F.bs=3, F.fs=0	(G,D,F) sortată: (G,D,F)	(A,B,C)
4	G	((H,1)) H.f=G, H.bs=3, H.fs=0	(D,F,H) sortată: (D,F,H)	(A,B,C,G)
5	D	((E,1)) E.f=D, E.bs=4, E.fs=0	(F,H,E) sortată: (F,H,E)	(A,B,C,G,D)
6	F	((G,1))	(H,E)	(A,B,C,G,D,F)
7	H – goal(H)=TRUE		(E)	(A,B,C,G,D,F,H)

Simularea rulării algoritmului din Tabelul 3.2 relevă că un singur nod satisface testul $node \in open \cup closed$, anume G, generat a doua oară ca descendent al nodului F la pasul 6 ($current-node = F$ și $node = G$). Pentru acest nod testul $current-node.back-score + score < node.back-score$ este evaluat la FALSE (într-adevăr $3+1 > 2$). Asta înseamnă că drumul până la G ca A-B-C-F-G este mai lung decât cel găsit anterior ca A-B-G. Din acest motiv părintele nodului G nu se schimbă.

Să luăm însă exemplul grafului din Figura 3.19 și să considerăm că, dintr-un motiv oarecare, funcția euristică pe care o avem la dispoziție apreciază ca *forth-score* pentru nodurile F și J valoarea 20, valorile tuturor celorlalte noduri fiind 10. Atunci evoluția algoritmului ar fi dată de Tabelul 3.3.

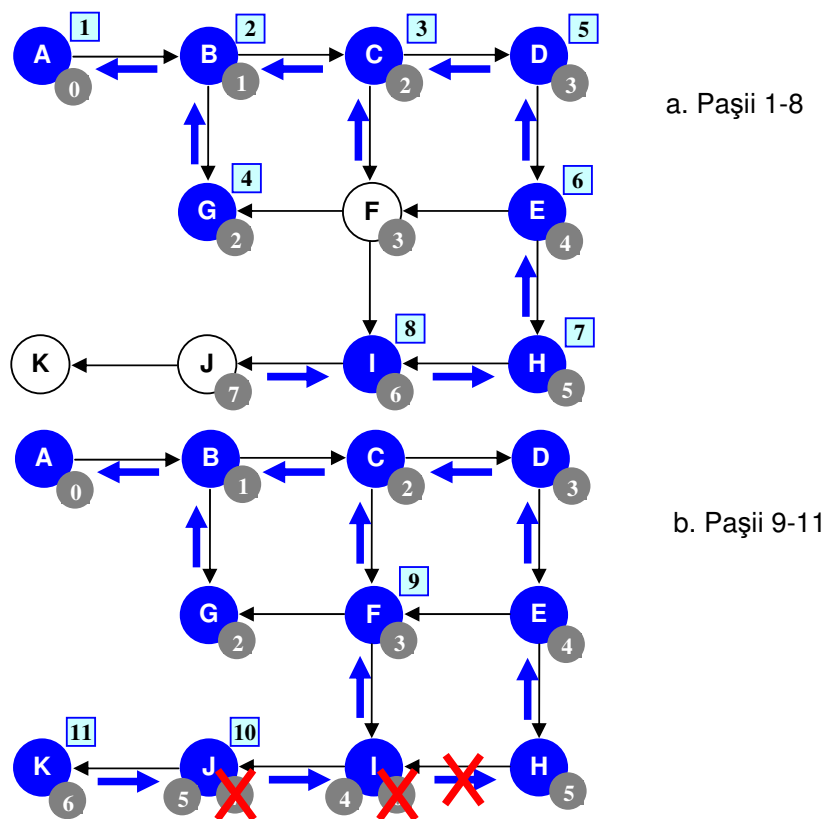


Figura 3.19: Un spațiu al stărilor reprezentat ca graf în care arborele de căutare evoluează cu reveniri

Tabelul 3.3: Derularea algoritmului în caz de revenire asupra deciziilor (pe spațiul stărilor din Figura 3.19)

Pas	current-node	pairs: { (node, score) }	open	closed
0			(A)	∅
1	A (A.f=nil, A.bs=0, A.fs=7)	((B,1)) B.f=A, B.bs=1, B.fs=10	(B)	(A)
2	B	((C,1),(G,1)) C.f=B, C.bs=2, C.fs=10 G.f=B, G.bs=2, G.fs=10	(C,G) sortată: (C,G)	(A,B)
3	C	((D,1),(F,1)) D.f=C, D.bs=3, D.fs=10 F.f=C, F.bs=3, F.fs=20	(G,D,F) sortată: (G,D,F)	(A,B,C)
4	G	()	(D,F) sortată: (D,F)	(A,B,C,G)
5	D	((E,1)) E.f=D, E.bs=4, E.fs=10	(F,E) sortată: (E,F)	(A,B,C,G,D)
6	E	((F,1),(H,1)) H.f=E, H.bs=5, H.fs=10	(F,H) sortată: (H, F)	(A,B,C,G,D,E)
7	H	((I,1)) I.f=H, I.bs=6, I.fs=10	(F,I) sortată: (I,F)	(A,B,C,G,D,E,H)
8	I	((J,1)) J.f=I, J.bs=7, J.fs=20	(F,J) sortată: (F,J)	(A,B,C,G,D,E,H,I)
9	F	((G,1)(I,1)) I.f=F, I.bs=4, I.fs=10 updateDescScores (I) → J.f=I, J.bs=5	(J)	(A,B,C,G,D,E,H,I, F)
10	J	((K,1)) K.f=J, K.bs=6, K.fs=10	(K)	(A,B,C,G,D,E,H,I, F,J)
11	K – goal(K)=TRUE	()	()	(A,B,C,G,D,E,H,I, F,J,K)

În Figura 3.19a se prezintă pașii 1-8. Datorită valorii mare a funcției euristice nodul F a fost ocolit în parcurgere, fiind trecut la fiecare sortare a listei *open* în coada ei (pașii 3-7). În Figura 3.19b sunt dați pașii 9-11. În momentul în care nodul F devine nod curent are loc o rearanjare a valorii de *back-score* a nodului I, descendent al lui F și valoarea acestuia este corectată de la 6 la 4. Această reșezare poate influența toate nodurile ce pot fi atinse din F și care fuseseră deja calculate. Apelul funcției `updateDescScores(I)` realizează acest lucru. Ca urmare și valoarea nodului J se modifică de la 7 la 5.

Să observăm că dacă ceea ce interesează e numai găsirea unei căi spre soluție, atunci putem face $g(n)=0$ în fiecare nod n . După cum am văzut în cele două exemple de mai sus, dacă dorim cea mai scurtă cale spre soluție, atunci $g(n)=nr.$ de pași până în n (costul unei căi = 1) și rezultatul e similar unei căutări *breadth-first*. Dacă fiecare operator are un alt cost, și se dorește cea mai ieftină cale, atunci g poate oglindi costul căii.

Dacă algoritmul A folosește o funcție euristică h care este o limită inferioară a costului căii optime de la un nod n la țintă, atunci el se numește A^* . Următoarele rezultate se cunosc despre A^* (v. Nilsson, 1980):

- A^* se termină întotdeauna în grafuri finite;
- dacă o cale există de la nodul start la o soluție, atunci A^* se termină (chiar dacă graful e infinit);
- A^* e admisibil (adică, dacă există o cale de la nodul de start la un nod soluție, A^* găsește o cale optimă). Spunem că un algoritm A_2 e mai informat decât A_1 , dacă, pentru toate nodurile n diferite de soluție, $h_2(n) > h_1(n)$.
- Dacă A_1 și A_2 sunt două versiuni ale lui A^* , astfel încât A_2 e mai informat decât A_1 , atunci la terminarea căutărilor lor pe orice graf în care există o cale de la nodul start la un nod soluție, orice nod expandat de A_2 este de asemenea expandat și de A_1 . Urmează că A_1 expandează cel puțin tot atâtea noduri ca și A_2 .

3.4 Sisteme Expert

3.4.1 Tipuri de căutări

Am văzut în acest capitol câteva metode de a organiza o căutare în spațiul stărilor pentru găsirea soluției unei probleme de IA. Maniera de căutare a fost întotdeauna dinspre o stare considerată inițială către o stare considerată finală, fie că știm cu exactitate acea stare, fie că o cunoaștem doar prin anumite condiții care sunt verificate de ea. Parcurgerea spațiului stărilor este deci întotdeauna “către înainte” (*forward-looking*), sau plecând de la premise pentru a ajunge la o concluzie. Dar aceasta nu e singura posibilitate: la fel de bine putem adopta o manieră de căutare care pleacă dinspre concluzie pentru a verifica dacă avem premise care s-o verifice. Maniera se numește “căutare înapoi” (*backward-looking*). Limbajul Prolog implementează o astfel de strategie. În sfârșit se poate vorbi și de o căutare bidirecțională, în care se pleacă simultan dinspre premise și concluzie în încercarea de a găsi o cale care să le unească.

3.4.2 Sistemele Expert (SE)

Un Sistem Expert este un program capabil de a încorpora și folosi cunoștințe echivalente cu cele pe care le are un expert uman într-un anumit domeniu. Un **shell** (sau o “carapace”) de SE este ceea ce ar rămâne dintr-un SE dacă l-am goli de cunoașterea expert. El este, așadar, un sistem de control capabil de a efectua o căutare într-un spațiu al stărilor, de aceea se mai numește și **motor de inferențe**.

Premiza principală în construcția unui SE este aceea că un expert își construiește soluția la o problemă din piese elementare de cunoaștere pe care le selectează și le aplică într-o anumită secvență. Pentru a furniza o soluție coerentă la o problemă dată cunoașterea cuprinsă într-un anumit domeniu trebuie așadar formalizată, apoi reprezentată corespunzător și în final manipulată în conformitate cu o anumită metodă de rezolvare de probleme. Se pune astfel în evidență diviziunea dintre secțiunea care păstrează reprezentarea cunoașterii asupra domeniului – **baza de cunoștințe**, formată dintr-o colecție (bază) de fapte împreună cu o colecție (bază) de reguli și diviziunea responsabilă cu organizarea proceselor inferențiale care să implice aceste cunoștințe – **sistemul de control**.

Un *shell* foarte cunoscut de SE este CLIPS (Giarratano, Riley, 1998).

3.5 Concluzie

Rezolvarea unei probleme de IA are două aspecte:

- formalizarea problemei = găsirea unei reprezentări pentru stările problemei și găsirea unei reprezentări pentru reguli;
- lansarea unei căutări în spațiul stărilor în scopul determinării unei căi între starea considerată inițială și o stare finală.

Când un astfel de drum nu poate fi găsit spunem că problema nu are soluție.

Aceste concluzii pun în evidență și preocupările majore ale domeniului: găsirea celor mai adecvate procedee de reprezentarea a problemelor, respectiv găsirea celor mai eficiente procedee de navigare în spații ale stărilor.

Cerințe pentru studenți:

- Să recunoască încadrarea unei probleme în clasa de probleme IA.
- Să deosebească o problemă de o ipostază de problemă.
- Să identifice spațiul stărilor problemei și să-i aprecieze mărimea.
- Să construiască o reprezentare adecvată a stărilor.
- Să identifice regulile și să construiască o reprezentare neformală a lor.
- Să hotărască asupra unui tip de căutare a soluției.
- Să poată adapta algoritmul la cerințele particulare ale problemei.

Probleme

P3.1 Recuperarea soluției: să se modifice algoritmi `hill-climbing()` și `rec-hill-climbing()` din secțiunea 3.3.1 pentru a-i face să întoarcă: a). o secvență de stări prin care se trece în drumul de la starea inițială până la cea finală; b). o secvență de operatori care aplicați stării inițiale să ducă la obținerea stării finale.

P3.2 Aceleași cerințe pentru algoritmi `backtracking-hill-climbing()` și `rec-backtracking-hill-climbing()` din secțiunea 3.3.2.

P3.3 Să se figureze un arbore a soluției pentru instanța problemei cuburilor dată în Figura 3.20, în care funcția euristică adaugă, pentru fiecare bloc, 1 – dacă stă pe blocul pe care ar trebui să ajungă și scade 1 – dacă nu stă pe blocul pe care ar trebui să ajungă. Operatorii sunt cei cunoscuți din secțiunea 3.2.2.

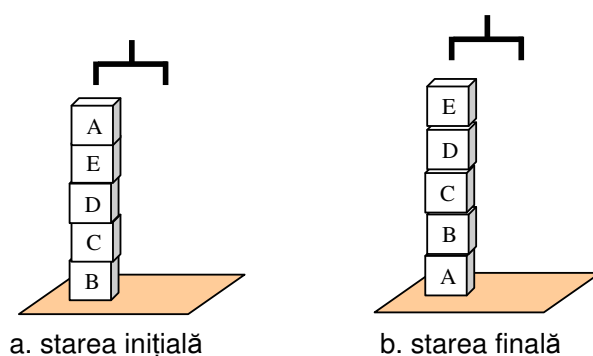


Figura 3.20: O instanță a problemei cuburilor cu cinci piese

P3.4 Să se descrie funcția `updateDescScores(node)` din componența algoritmului A (v. secțiunea 3.3.2).

P3.5 Să se descrie funcția `generatePath(current-node, start-node)` din componența algoritmului A (v. secțiunea 3.3.2).

P3.6 Se dă o gramatică independentă de context și o frază.

- Să se precizeze diferența dintre o problemă și o instanță de problemă în acest caz.
- Să se dea exemple de alte instanțe ale aceleiași probleme.
- Să se explice ce înseamnă o stare în rezolvarea acestei probleme și care e reprezentarea unei stări.
- Să se formalizeze ca o problemă de căutare în spațiul stărilor problema recunoașterii dacă fraza face parte din limbajul generat de gramatică.
- Ce strategie considerați adecvată pentru rezolvare?
- Să se dea exemple de două soluții posibile (derivări).

Exemplu pentru: $G = \{NT, T, S, P\}$, unde:

$NT = \{S, NP, VP, N, V\}$

$T = \{\text{calul}, \text{lacrimă}, \text{duce}\}$, iar

$P = \{S \rightarrow NP VP, NP \rightarrow NP NP, NP \rightarrow N, VP \rightarrow V NP, VP \rightarrow V, N \rightarrow \text{calul}, N \rightarrow \text{lacrimă}, N \rightarrow \text{duce}, V \rightarrow \text{duce}, V \rightarrow \text{lacrimă}\}$

și propoziția: "calul duce lacrimă".

P3.7 În testele de inteligență apar probleme în care este dată o secvență de numere și se cere să se găsească formula prin care se determină succesorul. Vi se cere să rezolvați prin mijloace specifice IA o astfel de problemă. Descrieți un algoritm care să găsească o soluție a unei astfel de probleme, presupunând existența unei proceduri `stripsSearch($n1, n2$)`, unde $n1$ și $n2$ sînt două numere din secvență. Procedura `stripsSearch()` realizează o căutare într-un spațiu al stărilor în care $n1$ este o stare inițială iar $n2$ – una finală, căutând o cale de aplicații de reguli. O regulă în această formalizare transformă un număr într-un alt număr. Procedura `stripsSearch()` întoarce o listă de secvențe de reguli. Numărul de aplicații ale procedurii `stripsSearch()` este dat de numărul de secvențe consecutive de două numere ce se cuprind în secvența originală. După fiecare apel al procedurii `stripsSearch()` numărul de secvențe de reguli rămas trebuie să fie mai mic sau cel mult egal cu cel anterior. În mod ideal, la terminarea acestei faze, o singură cale trebuie să rămână. Soluția problemei va trebui dată de aplicarea căii asupra ultimului număr din secvența dată.

P3.8 Pentru a controla explozia exponențială care poate să apară în anumite probleme de investigare a spațiului stărilor, se recurge la o îngrădire a listei nodurilor ce vor fi exploatate, reținând la fiecare pas primii N cei mai promițători candidați. Varianta se numește *beam-search*. Descrieți un astfel de algoritm de căutare pe arbori.

P3.9 Să se găsească o reprezentare adecvată pentru o stare în jocul de șah și apoi să se modeleze mutările calului și ale nebunului.

P3.10 a). Descrieți în Lisp regula **M-right** prin care un singur misionar se deplasează pe malul din dreapta. Forma generală a regulii trebuie să fie: **if CONDIȚII then ACȚIUNI**;

b). Descrieți o funcție de scor pentru problema canibalilor și misionarilor;

c). Poate problema misionarilor și canibalilor să aibă o rezolvare printr-un algoritm de tip *hill-climbing*? Explicați.

P3.11 Să se formalizeze ca o problemă de căutare în spațiul stărilor următoarea problemă de aliniere: se dau două semnale, fiecare fiind format dintr-o secvență de segmente de linie dreaptă aflate unul în prelungirea celuilalt (ca în Figura 3.21). Fiecare segment e caracterizat de o lungime (pe axa orizontală) și o pantă. Se dorește să se obțină cea mai bună aliniere între cele două semnale. În evaluarea unei alinieri contează lungimea segmentelor și panta acestora.

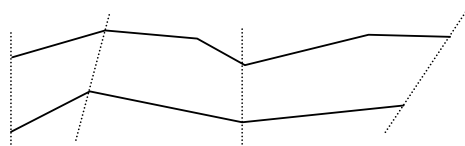


Figura 3.21: Alinierea a două semnale codificate ca secvențe de linii frânte

P3.12 Se consideră un joc ce se desfășoară pe o tablă 3×3 pe care sunt plasate piese albe și negre (Figura 3.22). Fiecare dintre cei doi jucători are asignată câte o culoare. La o mutare, un jucător poate selecta un dreptunghi, piesele din interiorul dreptunghiului urmând să-și schimbe culoarea. Câștigă primul jucător care obține pe tablă toate piesele în culoarea lui.

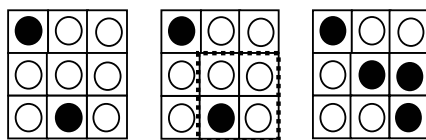


Figura 3.22: O secvență de trei mutări în evoluția jocului

- Să se propună un set de predicate care să facă posibilă descrierea stărilor jocului. Cu ele să se definească starea inițială (figura din stânga) și starea finală.
- Să se propună reguli pentru modelarea mișcărilor jucătorilor.
- Să se propună o funcție de evaluare a unei stări și un algoritm de control.








P3.13 Se dă o tablă de șah pe care se găsesc inițial plasați cai. Ei se deplasează, utilizând salturile caracteristice calului din jocul de șah, întotdeauna spre hambar (colțul din stânga sus al tablei). Mișcările se termina când unul din cai reușește să ajungă în hambar.

- Să se propună un set de predicate care să fie utilizate în reprezentarea unei stări și, cu ajutorul lor, să se descrie o stare inițială.
- Să se descrie un set de reguli care să realizeze mutările cailor.
- Să se descrie o secvență de mutări care să conducă spre soluție.

P3.14 Doi parteneri, A și B, stau față în față: B și-a pus în minte un număr între 1 și 10 pe care A trebuie să-l ghicească. A propune numere iar B răspunde prin “da”, “mai mic” sau “mai mare”.

- Formulați acest dialog ca o problemă IA.
- Descrieți secvența de reguli aplicate și de stări generate când B alege numărul 7.

P3.15 Într-un joc pe calculator intervin următoarele elemente:

-  un omuleț, care este un obiect ușor dar dur, se poate mișca în toate direcțiile, poate împinge obiecte și mânca inimioare, când se află lângă acestea;
-  două bile, care sînt obiecte dure și grele;
-  bombă, care poate exploda dacă un obiect dur cade peste ea sau dacă ea cade peste un obiect dur, caz în care se autodistruge și distruge orice obiect aflat în imediata apropiere;
-  inimioară, care este un obiect dur și greu;
-  cinci parcele de pajiște care sînt obiecte moi ce se distrug la trecerea omulețului;
-  un balon, care poate înălța un singur obiect greu dar care e susținut de pajiște;
-  ușă, care se deschide când omulețul a mâncat inimioara.

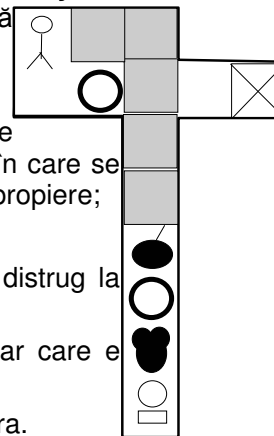


Figura 3.23: Starea inițială a jocului

Un obiect greu poate cădea atunci când nu e susținut. Un obiect ușor nu cade.

- Să se imagineze un set de predicate cu ajutorul cărora să se reprezinte starea din Figura 3.23.
- Să se descrie acțiunile posibile din acest mic univers ca un set de reguli de producție.
- Să se imagineze secvența de aplicări de reguli prin care omulețul poate ieși prin ușă.

P3.16 Descrieți într-un eseu elementele de inteligență artificială pe care vi le imaginați încorporate în casa viitorului (max 2 pagini). Dați cât mai multe indicații de modelare.

P3.17 Ceea ce urmează reprezintă formularea unui joc numit *Lives* și atribuit matematicianului John Conway de la Univ. Cambridge:

Există a tablă pătrată (presupus infinită) pe care se pot dezvolta celule. Fiecare poziție de pe tablă poate evolua în funcție de starea ei și a celor opt vecini ai săi conform a trei reguli:

1. *Supraviețuire. Orice celulă vie care se învecinează cu două sau trei alte celule vii supraviețuiește.*
2. *Moarte. Orice celulă vie care se învecinează cu patru sau mai multe celule vii moare de suprapopulare. Orice celulă vie care se învecinează cu una sau nici o celulă vie moare de izolare.*
3. *Naștere. În orice locație goală care se învecinează cu exact trei celule vii va apare o celulă vie.*

Toate regulile se calculează întâi pentru fiecare celulă și se aplică odată pentru a da naștere la următoarea generație a tablei.

Într-o anumită implementare a jocului prin reguli de producție, s-a preferat o reprezentare în care pentru fiecare locație de coordonate (x, y) a tablei se cunosc, la orice moment, următoarele predicate:

- $\text{current-value}(x, y, v)$ – cu semnificația că valoarea curentă a locației este $v \in \{\text{empty}, \text{cell}\}$;
- $\text{new-value}(x, y, v)$ – cu semnificația că valoarea la generația următoare a locației trebuie să fie $v \in \{\text{empty}, \text{cell}\}$;
- $\text{neighbour}(x, y, n)$ – cu semnificația că numărul de celule vii învecinate locației este n ($0 \leq n \leq 8$);
- $\text{tested}(x, y, t)$ – cu semnificația că locația este deja calculată pentru noua generație ($t = \text{true}$), ori nu ($t = \text{false}$).
 - Descrieți un set de operatori care guvernează evoluția simulată a jocului. Precizați mecanismele suplimentare de control și reprezentare de care aveți nevoie pentru secvențierea fazelor;
 - Descrieți prin setul de predicate precizat anterior următoarea stare inițială:

•	•	•

Figura 3.24: Starea inițială a jocului LIFE

- Descrieți o secvență de reguli care produc două generații de celule plecând de la starea inițială precizată și arătați care va fi configurația ultimei generații din cele două.

Bibliografie

Cristea, D. 2002. *Programarea bazată pe reguli*. Ed. Academiei, București.

Giarratano, J. Riley, G. 1998. *Expert Systems: Principles and Programming*, 3rd Edition, PWS Publishing.

Hart, P.E., Nilsson, N.J., Raphael, B. 1968. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Transactions on Systems Science and Cybernetics, Vol 4, No. 2, 1968, pp. 100-107.

Nilson, N. 1980. *Principles of Artificial Intelligence*. Morgan Kaufmann.

Capitolul 4

Reprezentarea cunoașterii și raționament

Bine, atunci: care e viteza întinericului?
Wright Stevens

Realitatea este tot ceea ce ne înconjoară. La baza interacțiunii noastre, ca sisteme vii, cu realitatea stau necesitățile și dorințele noastre. Credem că sediul acestor imbolduri este creierul nostru și tot acolo realitatea trebuie să fie reflectată într-un anume fel. La masă fiind, întindem mâna după solniță, o apucăm și o scuturăm deasupra grătarului de vită din farfurie, după care o așezăm la loc. Facem toate acestea mai întâi pentru că „ne place” mâncarea mai sărată și apoi pentru că „vrem” să ne satisfacem acest gust. Apoi „știm” că sarea se află în solniță, că o putem face să iasă din solniță și presăra peste mâncare prin scuturare deasupra ei, că pentru a aduce solnița deasupra farfuriei trebuie să o apucăm cu mâna, să o răsturnăm deasupra farfuriei și să o scuturăm, după care, în mod normal, trebuie să o repunem pe masă de unde am luat-o. Suntem apoi capabili să „recunoaștem” solnița undeva pe masă. În fine putem să “controlăm” mișcările mâinii, întâi îndreptând-o spre locul unde se află solnița, apoi deschizând palma ca pentru a apuca între degete, apoi apucând solnița cu mâna, retrăgând-o până solnița ajunge deasupra farfuriei, întorcând mâna și scuturând-o atât cât “credem” că ne trebuie sare și făcând apoi toate mișcările în sens invers pentru a depune înapoi solnița pe masă de unde am luat-o. Toate aceste acțiuni sunt atât de banale încât le facem purtând un dialog cu partenerul de masă despre un subiect care nu are nimic în comun cu condimentarea fripturii din fața noastră, reflex, aproape fără să ne concentrăm la ceea ce facem.

Dar toate ghilimelele din paragraful de mai sus, comportă acte și stări cognitive, care compun un tablou extrem de spectaculos. Dacă ar trebui să-l detaliem, am recunoaște următoarele componente:

- realitatea, adică noi, masa, solnița, sarea, farfuria, friptura etc., este formată din obiecte care sunt independente unele de altele. Acestea se supun legilor fizicii, de exemplu ocupă locuri în spațiu, nu e posibil ca două dintre ele să se afle în același loc în același moment și sunt grele, adică cad dacă nu sunt sprijinite de un suport. Datorită unui complex de împrejurări oarecare, acestea se găsesc toate într-o vecinătate uman sesizabilă, la momentul desfășurării scenei;
- noi avem plăceri, dorințe, nevoi. Aceste stări de spirit sunt motoarele acțiunilor pe care le inițiem. Fără ele am adăsta, inerți ca niște legume, ca vântul să ne mute și ploaia să ne ude;
- noi avem cunoștințe despre foarte multe lucruri, printre altele și despre ceea ce este normal să existe și să se întâmple în realitate. Aceste cunoștințe reprezintă o memorie a lucrurilor văzute, povestite ori citite. De exemplu putem găzdui în această memorie relații între obiecte care se

află în câmpul nostru vizual și altele care nu se află acolo. Astfel, "știm" că, în mod normal, sarea se află în solniță, mai "știm" cum arată o solniță, adică care e forma ei, și "știm" că sarea e sărată, adică are acel gust care dorim să-l completeze pe cel al fripturii. În același timp cunoaștem acțiunea de "a pune sare în mâncare la masă" pentru că am repetat-o de atâtea ori până acum. Dacă nu am avea această cunoaștere, încă ar fi posibil să ne satisfacem nevoia de sare (deși autorul nu o recomandă cu prea aprinsă căldură cititorilor săi), pentru că, înzestrați cu inteligență cum ne aflăm, am putea corela anume cunoștințe pentru a rezolva pentru prima oară această problemă, eventual din câteva încercări. Cu alte cuvinte o experiență de acest tip, trăită, primită ori descoperită, este evocată în memoria noastră ca soluția la nevoia de a avea friptura sărată;

- rezultatul va fi construirea unui plan care să particularizeze cunoașterea de natură generală, la detaliile mesei și aranjării obiectelor aflate în fața noastră. În particularizarea planului, crucială este "recunoașterea" solniței pe masă, deși este pentru prima oară când vedem această solniță anume. Reușim să identificăm solnița pe masă pentru că ea seamănă atât de mult cu forma pe care știm că ar trebui să o aibă o solniță. Dacă, din obscure motive, solnița de pe masa noastră ar avea forma și dimensiunile unei mingi de baschet, e greu de crezut că am reuși să o identificăm;
- în fine, planul este pus în aplicare, creierul comandă mâinii secvența tuturor acelor mișcări și urmărește realizarea lui, controlând permanent poziția mâinii cu imaginea primită în ochi. Dacă, după ce am localizat solnița pe masă, am încerca să ne ducem la îndeplinire planul cu ochii închiși, am avea mari șanse să apucăm în locul solniței piperul pe care să-l scuturăm în paharul cu vin...

Esențială în acest scenariu este reprezentarea realității în memoria noastră. Fără această reprezentare, nu am recunoaște situația drept una deja întâlnită, nu am putea particulariza secvența de acțiuni la realitățile locului pentru a construi un plan și prin urmare am rămâne cu friptura nesărată.

În acest capitol vom trece în revistă câteva tipuri de reprezentări ce se aplică sistemelor de IA.

4.1 Ontologii

În filosofie, cuvântul „ontologie” desemnează un punct de vedere sistematic asupra Existenței. El este împrumutat de IA unde este utilizat cu semnificația de **specificare a unei conceptualizări** (Gruber, 1993, Gruber, 2003). Cu alte cuvinte o ontologie reprezintă o descriere a conceptelor și a relațiilor ce sunt stabilite între acestea pentru a servi unui agent sau unei comunități de agenți. Scopul ontologiei este definirea regulilor formale care să permită unei comunități de agenți să interpreteze în același mod un segment al realității (deci al existenței).

4. 2 Taxonomii

Taxonomia este un tip particular de ontologie. Într-o taxonomie, concepte mai puțin generale sunt definite în funcție de concepte mai generale, acestea – în funcție de altele încă mai generale decât ele ș.a.m.d.

Mulțimea conceptelor este structurată în ierarhii (taxonomii). Relația taxonomică este una de moștenire de proprietăți și se notează ISA (de la *is a* din engleză), AKO (*a kind of*), SUBSET, MEMBER etc.¹³ Dacă x ISA y spunem că y este **părinte** pentru x , iar x este **fiu** al lui y . De asemenea vom spune că orice părinte al lui x ca și orice părinte al unui părinte al lui x este un **predecesor** pentru x , și recursiv – orice părinte al unui predecesor al lui x este predecesor al lui x . Reciproc, un fiu este un **descendent**, la fel ca și un fiu al unui fiu și un fiu al unui descendent.

Iată un exemplu de ierarhie, notată grafic ca în Figura 4.1:

```
câine ISA mamifer
pisică ISA mamifer
mamifer ISA animal
pește ISA animal
animal ISA entitate_vie
copac ISA entitate_vie
entitate_vie ISA obiect_fizic
obiect_fizic ISA entitate_fizică
```

Datorită tranzitivității relațiilor ISA, din aceste relații se pot deduce cel puțin următoarele:

```
câine ISA animal
câine ISA entitate_vie
câine ISA obiect_fizic
câine ISA entitate_fizică
pisică ISA animal
...
pisică ISA entitate_fizică
mamifer ISA entitate_vie
...
mamifer ISA entitate_fizică
pește ISA entitate_vie
...
pește ISA entitate_fizică
animal ISA obiect_fizic
animal ISA entitate_fizică
copac ISA obiect_fizic
copac ISA entitate_fizică
entitate_vie ISA entitate_fizică
```

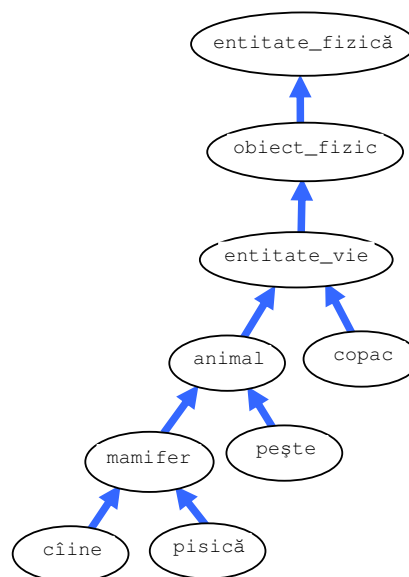


Figura 4.1: Un exemplu de taxonomie

Noțiunea de concept din rețelele semantice este asimilabilă celei de clasă din reprezentările orientate obiect. Numele atașate conceptelor sunt convenții de notație și nu sunt suficiente pentru a individualiza conceptele între ele. O reprezentare semantică satisfăcătoare trebuie să individualizeze conceptele prin descrieri

¹³ Unele sisteme fac deosebire între relațiile ierarhice aplicate la nivelul conceptelor (numite acolo SUBSET) și cele dintre instanțe și concepte (numite ISA) (v. de exemplu (Tufiș, Cristea, 1985)).

specifice, proprietăți caracteristice. Descrierea unui concept este dată de relații de natură semantică atașate acestuia. În reprezentările noastre grafice, relațiile semantice vor fi figurate prin săgeți simple etichetate cu numele lor, în timp de relațiile de natură taxonomică (ISA) vor fi reprezentate prin săgeți îngroșate (ca în Figura 1) sau duble, unind conceptele (sau instanțele lor) de ascendenții imediați. Într-o astfel de ierarhie, conceptele mai particulare se spune că **moștenesc** proprietățile celor mai generale. Pentru că reprezentările noastre sunt, în esență, grafuri, adesea ne vom referi la conceptele sau instanțele ce le populează cu numele generic de noduri iar relațiile dintre ele vor fi arcele care unesc nodurile.

Noțiunea de concept nu poate fi identificată cu mulțimea instanțelor sale pentru că un concept este format nu numai din mulțimea instanțelor cunoscute (înregistrate la un moment dat) ale acestuia dar și de oricâte altele (necunoscute dar posibile) ce ar putea satisface proprietățile atașate lui. De exemplu conceptul de *persoană fizică* este constant indiferent de câte instanțe ale acestuia sunt cunoscute la un moment dat. Deci un concept ar putea fi văzut ca reuniunea unei mulțimi de instanțe fizice cu una de instanțe virtuale, aceasta din urmă – de cardinal neprecizat, potențial infinit. Mulțimile (concrete ori virtuale) ale obiectelor asociate conceptelor nu sunt neapărat disjuncte. De exemplu conceptul de *programator* și cel de *bărbat* se intersectează și ambele sunt conținute în cel de *om*. Într-o reprezentare ca mulțimi, situația poate fi redată ca în Figura 4.2a, în timp ce, într-o reprezentare comună rețelelor semantice, aceleași relații sunt redade ca în Figura 4.2b.

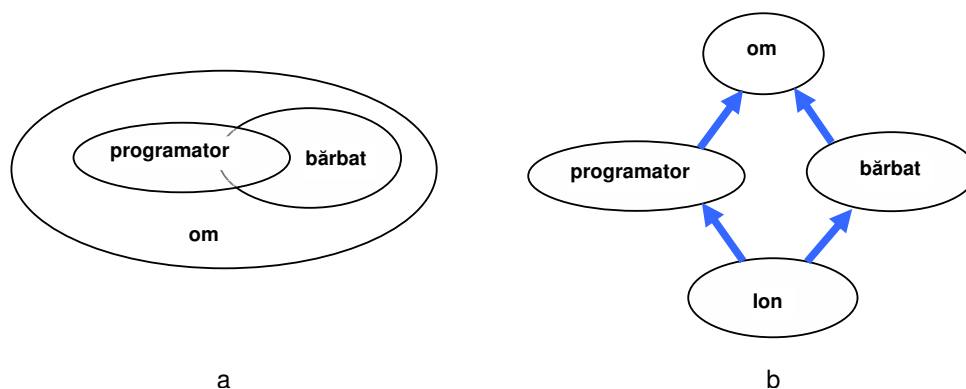


Figura 4.2: De la mulțimi de instanțe la concepte

4.3 Paternitate versus monotonie în sisteme ierarhice

Într-un sistem ierarhic moștenirea proprietăților este caracterizată de două dimensiuni:

- **paternitatea** – cu simplă ori multiplă moștenire. Într-un sistem cu simplă moștenire nici un nod nu poate avea mai mult decât un singur părinte, pe când într-un sistem cu moștenire multiplă există cel puțin un nod care are cel puțin doi părinți;

- **monotonia** – moștenirea proprietăților este monotonă dacă fiecare nod moștenește toate proprietățile părinților și nici o nouă proprietate adăugată pe un nivel inferior nu poate contrazice una omonimă ei declarată în ierarhie pe un nivel superior. Dimpotrivă, într-un sistem nemonoton o proprietate, dacă apare de mai multe ori, este culeasă din predecesorul cel mai apropiat. Astfel, în rețeaua din Figura 4.3, presupunând că relațiile taxonomice sunt reprezentate prin săgeți îngroșate, proprietățile – prin arce orizontale subțiri etichetate și că valoarea unei proprietăți este dat de nodul în care ajunge arcul respectiv, vom găsi că proprietatea R_1 a nodurilor C_0 și C_1 este C_4 , nu C_5 și nici C_9 , și aceeași proprietate R_1 a nodului C_2 este C_5 , iar nu C_9 . La fel, proprietatea R_2 a nodului C_4 și C_5 este C_7 , iar nu C_9 . Dacă am cere valoarea proprietății dată de o cale vidă de relații a nodului x ar trebui să-l obținem pe x .

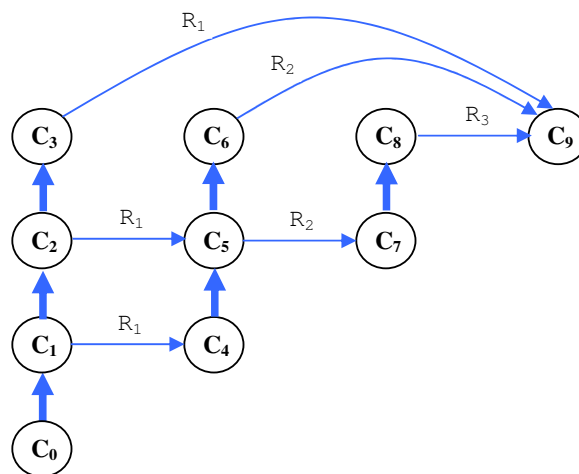


Figura 4.3: O ierarhie nemonotonă

Cele două coordonate, fiecare având două valori posibile ne dau patru tipuri de sisteme. Acestea sunt explicitate în cele ce urmează:

- **sistem monoton cu o singură moștenire:** fiecare entitate are un singur părinte și orice entitate în ierarhie moștenește toate proprietățile părinților;
- **sistem nemonoton cu o singură moștenire:** fiecare entitate are un singur părinte și dacă o proprietate este repetată în mai multe locuri în ierarhie, valoarea cea mai de jos este cea care primează;
- **sistem monoton cu moștenire multiplă:** o entitate poate avea mai mulți părinți și orice entitate în ierarhie moștenește toate proprietățile părinților;
- **sistem nemonoton cu moștenire multiplă:** o entitate poate avea mai mulți părinți și, dacă o proprietate este repetată în mai multe locuri în ierarhie, aceeași relație poate fi comunicată de mai mulți părinți.

4.3.1 Soluții în cazul moștenirilor contradictorii

Pentru că o reprezentare de tip nemonoton în care se acceptă moștenirea multiplă poate genera contradicții sau ambiguități pe motivul că aceeași proprietate poate fi moștenită cu valori diferite din doi sau mai mulți părinți distincți, una din următoarele soluții poate fi adoptată:

- să se adopte o regulă de **moștenire ortogonală** în proiectarea rețelei: informația este partiționată între nodurile părinți astfel încât nici o proprietate să nu poate fi moștenită de la mai mult decât un singur nod părinte;
- să se stabilească o **regulă de prioritate** între nodurile ascendenți, astfel încât, dacă aceeași proprietate apare din mai multe locuri, ordinea dată regula de prioritate să fie cea care să indice informația moștenită. Astfel, în ierarhia de clase a CLOS cât și a sistemului ELU (Estival, 1990), (Russel et al., 1992) este adoptată regula întâi-în-adâncime și de-la-stânga-la-dreapta. În Figura 4-4 “<” trebuie citit ca “este mai prioritar”.

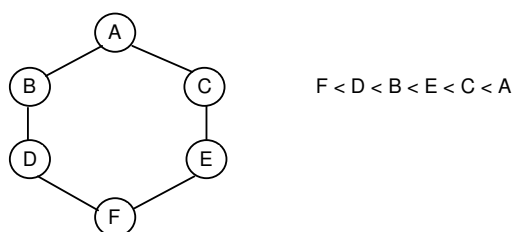


Figura 4.4: Regula întâi-în-adâncime, stânga-dreapta în moștenirea proprietăților

4.4 Rețelele semantice descriptive – sisteme ierarhice nemonotone

Vom studia în cele ce urmează un sistem ierarhic nemonoton numit **rețea semantică**. Vom lăsa înadins nespecificată coordonata paternitate, ea putând fi, după caz, cu simplă ori multiplă moștenire. În acest din urmă caz, vom presupune că rezolvarea ambiguităților datorate posibilelor moșteniri contradictorii este făcută fie printr-o proiectare ortogonală a sistemului astfel încât niciodată aceeași proprietate să nu poată fi moștenită pe două căi, fie prin încorporarea unui mecanism de moștenire cu priorități.

Din punct de vedere matematic, o rețea semantică este un graf – format din noduri (cu semnificația de concepte sau instanțe ale acestora) și arce (cu semnificația de relații). Atributul de semantică este dat din motivul că aceste rețele descriu cu ușurință proprietăți semantice ale unui univers limitat (numit și univers al discursului). Aproximarea (măsurată în lungime de căi) dintre concepte și a instanțe în rețea simulează “apropierea semantică” dintre acestea în lumea pe care o reflectă reprezentarea.

4.4.1 Un exemplu de reprezentare

Din punctul de vedere al tipurilor de lumi reprezentate, rețelele semantice pot fi: **descriptive**, dacă descriu lumi obiectuale, statice, și **evenimentiale**, dacă

descrierile se referă la lumi dinamice, așadar în care accentul este pus pe evenimente aflate în desfășurare.

Vom studia în cele ce urmează un tip de rețea semantică descriptivă în care nodurile sunt structurate pe două niveluri – nivelul **conceptual** (sau intensional) și nivelul **referențial** (sau extensional) (Tufiș, Cristea, 1985). Rețelele semantice evenimențiale vor face obiectul secțiunii 4.5.

Pentru a da un făgaș aplicativ discuției, să considerăm exemplul unui univers al discursului format din corpuri geometrice, în care avem corpuri geometrice precum cuburi și cilindri (v. Figura 4.5).

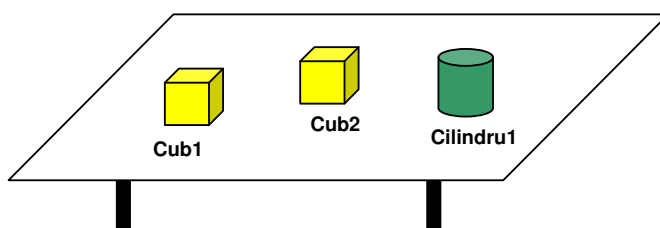


Figura 4.5: Un mini-univers al discursului

O posibilă reprezentare a acestei lumi sub forma unei taxonomii semantice descriptive arată ca în Figura 4.6.

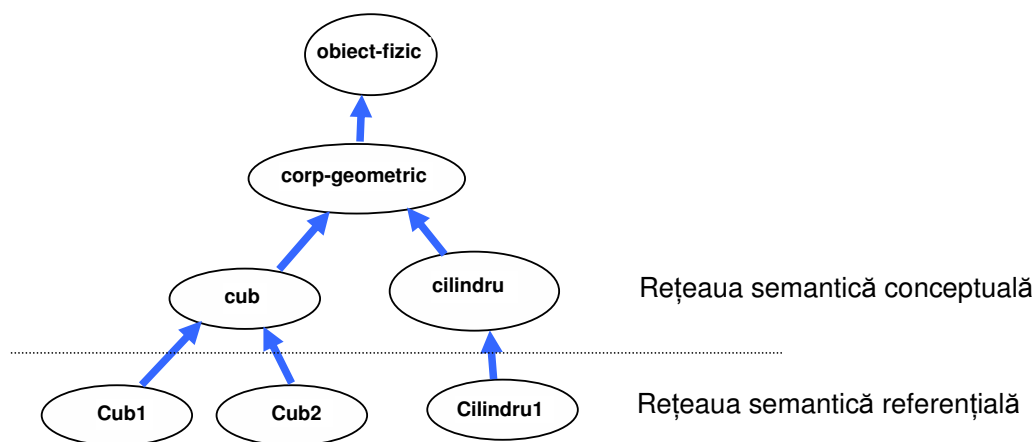


Figura 4.6: Taxonomia conceptelor care reprezintă lumea din Figura 4.5

Desigur această reprezentare include unele elemente ce nu aparțin figurii și anume cunoașterea de ordin general că orice cub este un corp-geometric, că orice cilindru este de asemenea un corp geometric și că un corp geometric este un obiect-fizic. Dintr-un alt punct de vedere, să convenim că etichetele cu care notăm conceptele sunt convenționale și, ca atare, vom presupune că ele, în sine, nu adăugă semnificații suplimentare reprezentării, care ar corespunde, de exemplu, unui pragmatism atașat de noi termenilor lingvistici respectivi.

Pentru lumea obiectuală de mai sus putem pune în evidență proprietăți atașate conceptelor, precum materialele din care sunt realizate corpurile și culorile acestora (v. Figura 4.7).

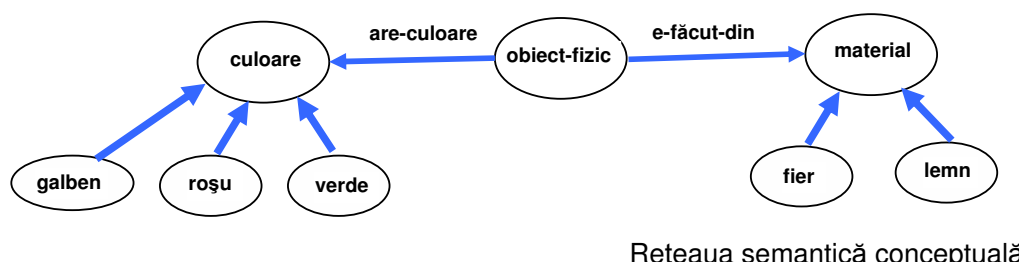


Figura 4.7: *Proprietăți atașate conceptului obiect-fizic*

Relațiile semantice ale unui nod aparținând rețelei sunt moștenite de toate nodurile (aflate în taxonomie pe niveluri inferioare lui). Din acest motiv, reprezentarea din Figura 4.7 pune în evidență faptul că orice obiect fizic are o culoare și este făcut dintr-un material. Dacă singurele instanțe ale conceptului **culoare** reprezentate în rețea sunt **galben**, **roșu** și **verde**, și singurele instanțe ale conceptului **material** sunt **fier** și **lemn**, reprezentarea arată că orice obiect fizic din mini-lumea noastră va fi ori galben, ori roșu, ori verde și va fi făcut din lemn sau fier.

La nivelul extensional, cel al instanțelor, obiectele pot avea, de asemenea, proprietăți atașate. Proprietățile sunt implicite, adică moștenite, sau explicite – deci notate în rețea prin relații. Așa cum am văzut în secțiunea 4.3.1, atunci când aceste relații sunt omonime celor de pe nivelurile superioare, semnificația este de restrângere a valorilor, ori de reprezentare a excepțiilor. Această semantică este dacă de o ordine de prioritate a moștenirii relațiilor semantice în rețea. În conformitate cu regula de moștenire a sistemelor nemonotone, relațiile de jos primează asupra celor aflate pe niveluri superioare.

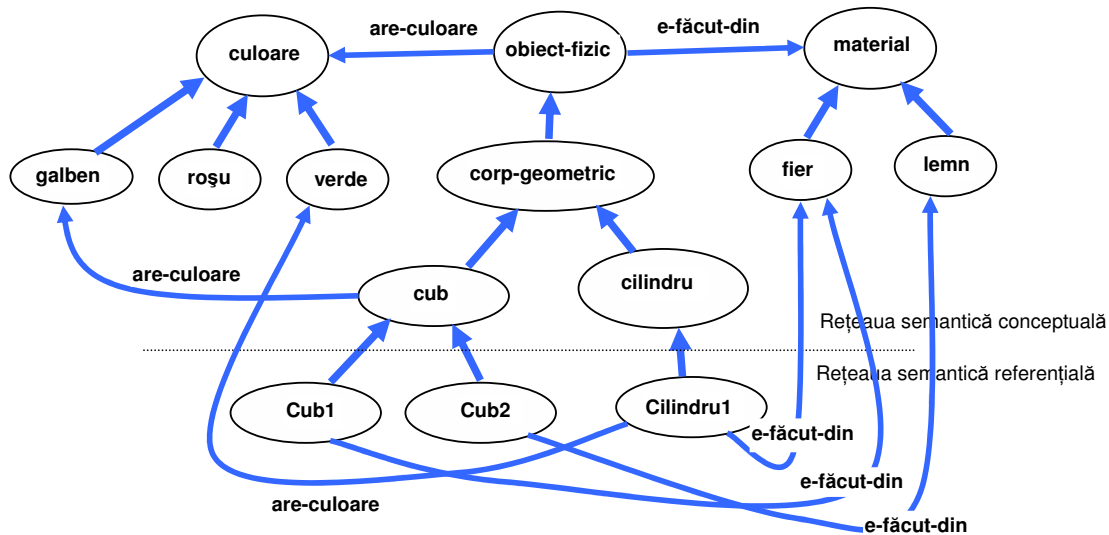


Figura 4.8: Particularizarea proprietăților într-o reprezentare nemonotonă

Astfel, reprezentarea din Figura 4.8 arată nu numai că orice obiect-fizic este înzestrat de proprietățile *are-culoare* și *e-făcut-din* dar că orice cub este de culoarea galbenă, cilindrul1 este verde, cubul1 și cilindrul1 sunt făcute din fier, iar cubul2 - din lemn. Acest înțeles se datorează faptului că relația semantică *are-culoare* ce părăsește nodul *cub* pentru a înțepa nodul *galben* este plasată mai jos în ierarhie decât relația omonimă plasată asupra unui nod aflat mai sus decât *cub* în ierarhie – respectiv conceptul *obiect-fizic*.

4.4.2 Interogări în rețele semantice

Rețele semantice trebuie să permită nu numai statuarea unor cunoștințe din universul de discurs supus reprezentării, dar și recuperarea acestora, indiferent dacă ele sunt reprezentate explicit sau implicit în rețea. Următoarele tipuri de întrebări pot fi adresate rețelelor semantice:

- care este închiderea tranzitivă a relațiilor taxonomice ISA ale unui nod din rețea?
- ce valoare este atașată prin relația semantică R nodului n ?
- care este calea de relații semantice ce se poate stabili între două noduri $n1$ și $n2$?
- care este valoarea regăsită prin navigare în rețea în lungul lanțului de relații $R1 \dots Rn$, plecând din nodul n ?

- Închiderea tranzitivă a relațiilor ISA

Întrebarea este formulată astfel: care sunt toți y astfel încât $x \text{ ISA}^+ y$ (prescurtat, $?y^*: x \text{ ISA}^+ y$). Funcția `findAncestors()` apelează funcția recursivă `findAncestorsRec()`:

```

procedure findAncestors( $x$ )
begin
    findAncestorsRec( $x$ ,  $\emptyset$ );
end

procedure findAncestorsRec( $x$ ,  $ancList$ )
begin
    ;  $ancList$  e un parametru acumulator care memorează toți
    ; strămoșii găsiți până în momentul apelului curent.
    if  $x$  ISA nil then return  $ancList$ ;
     $listFirstAnc \leftarrow$  toți părinții lui  $x$ ;
    while( $listFirstAnc$ )
        {  $someAnc \leftarrow$  first( $listFirstAnc$ );
           $ancList \leftarrow$  findAncestorsRec( $someAnc$ ,  $ancList \cup \{someAnc\}$ );
           $listFirstAnc \leftarrow$  rest( $listFirstAnc$ );
        }
    return  $ancList$ ;
end

```

- Valoarea moștenită a unei relații

Întrebarea este formulată astfel: care este y astfel încât $x R y$ (prescurtat, $?y: x R y$). Următoarea procedură oferă un răspuns printr-o căutare întâi-în-adâncime și de-la-stânga-la-dreapta, ceea ce este în conformitate cu criteriul de precedentă enunțat în secțiunea 4.3.1:

```

procedure getRelation( $x$ ,  $R$ )
begin
    if ( $x R y$ ) then return  $y$ ;
    if ( $x$  ISA nil) then return nil;
     $listFirstAnc \leftarrow$  toți părinții direcți ai lui  $x$  ordonați conform
    convenției de prioritate;
    while ( $listFirstAnc$ )
        {  $someAnc \leftarrow$  first( $listFirstAnc$ );
           $someVal \leftarrow$  getRelation( $someAnc$ ,  $R$ );
          if  $someVal \neq$  nil return  $someVal$ ;
           $listFirstAnc \leftarrow$  rest( $listFirstAnc$ );
        }
    return nil;
end

```

- Găsirea căii de relații semantice distribuite în ierarhia ascendentă a unui nod care îl unește cu un alt nod

Întrebarea este formulată astfel: fiind date nodurile x și y aparținând rețelei, care este calea de relații R_1, \dots, R_k plasată cel mai jos în ierarhie, astfel încât x ISA $\bullet R_1$ •ISA $\bullet \dots$ •ISA $\bullet R_k$ y (pe scurt, $?R^*: x$ ISA $\bullet R_1$ •ISA $\bullet \dots$ •ISA $\bullet R_k$ y)? Întrebarea urmărește găsirea celei mai scurte căi de relații semantice, plasate cel mai jos în ierarhie, care, eventual împreună cu relații taxonomice, să formeze o cale

de la un nod sursă la un nod destinație. Această cale precizează o anumită proprietate a obiectului sursă, inferabilă în virtutea comportamentului specific rețelelor semantice de moștenire a proprietăților. De exemplu, pentru situația schițată în **Figura 4.3** o interogare de genul: care este calea de relații R^* astfel încât $C_0 \ R^* \ C_9$, trebuie să producă soluția: $R_1 \bullet R_2 \bullet R_3$, pentru că, în sensul definiției moștenirii (din secțiunea 4.3.1), nici o altă combinație de relații nu „duce” din C_0 în C_9 . Într-adevăr, să presupunem că soluția ar fi fost calea formată din singura relație R_1 (ce poate fi culeasă printr-o navigare în rețea plecând din C_0 în lungul a trei relații ISA și a unei relații R_1). Valoarea recuperată a acestei relații/proprietate a nodului C_0 este nodul C_4 . Dacă soluția ar fi fost $R_1 \bullet R_2$ (de exemplu, pe motivul culegerii ei printr-o navigare în rețea plecând din C_0 în lungul a două relații ISA, unei relații R_1 , unei relații ISA și unei relații R_2) nodul regăsit ar fi de fapt C_7 ¹⁴.

Există mai multe posibilități de regăsire a acestei căi: căutare înainte, căutare înapoi ori bidirecțională. Următoarea procedură efectuează o căutare înainte:

```
procedure findPath(x, y)
begin
  findPathRec ({<x, Ø>}, y);
; Procedura recursivă findPathRec() primește doi parametri:
; - o structură de coadă în care fiecare intrare este o pereche
; formată dintr-un nod (în care s-a ajuns cu căutarea) și o cale
; de relații de la nodul de start până la el: <x, pathRel>;
; - nodul destinație.
; Structura de coadă este necesară pentru o abordare a căutării
; întâi-în-lărgime, ceea ce asigură calea de adâncime minimă
; în termeni de relații taxonomice accesate.
end

procedure findPathRec(queue, y)
begin
  if (null(queue)) then return FAIL;
  <x, pathRel> <- out(queue);
  if x = y then return pathRel;
  listRel <- mulțimea relațiilor semantice definite pe x;
  newEntries <- Ø;
; newEntries va fi lista de intrări contribuită de nodul curent x
  while (listRel)
  { R <- first(listRel);
    listRel <- cdr(listRel);
    z <- ?z, x R z;
    newEntries <- newEntries ∪ {<z, pathRel • R>;}
  }
  listParents <- ?z*: x ISA+ z;
```

¹⁴ Să observăm că în virtutea proprietății de moștenire, o cale de relații de lungime zero trebuie să ne lase strict în nodul de plecare, iar nu și în oricare din nodurile plasate în ierarhie deasupra nodului de plecare. Într-adevăr, dacă ar fi altfel, atunci mulțimea destinație a căii de lungime zero ar trebui să fie {C0, C1, C2, C3} ceea ce ar face ca destinația unei căi de lungime 1, să zicem R1, aplicată nodului de plecare C3 să fie mulțimea {C4, C5, C9}, iar nu strict C4.

```

; căutarea trebuie să continue însă și în nodurile părinți ale
; lui x;
while (listParents)
  { n <- first(listParents);
    listParents <- cdr(listParents);
    newEntries <- append(newEntries, {<n, pathRel>});
; calea de la nodul start până la oricare din părinții lui x este
; aceeași cu calea de la nodul start până la x
  }
return findPathRec(in(queue, newEntries), y);
; în apelul recursiv coada apare incrementată cu noile intrări.
end

```

- Găsirea unei valori prin navigare pe o cale de relații

Întrebarea este formulată astfel: plecând de la nod x să se găsească nodul y la care se poate ajunge (prin moștenire) în lungul căii de relații $ISA \bullet R_1 \bullet ISA \bullet \dots \bullet ISA \bullet R_k$ (pe scurt, $?y: x \text{ } ISA \bullet R_1 \bullet ISA \bullet \dots \bullet ISA \bullet R_k \text{ } y$)?

```

procedure getValueOnPath(x, listRel)
begin
  getValueOnPathRec({<x, listRel>});
; Procedura recursivă getValueOnPathRec() primește o structură de
; coadă în care fiecare intrare din este o pereche formată
; dintr-un nod (în care s-a ajuns cu căutarea) și calea
; de relații care au mai rămas de parcurs;
; Structura de coadă este necesară pentru o abordare a căutării
; întâi-în-lărgime, ceea ce asigură respectarea condiției
; de moștenire a proprietăților.
end

procedure getValueOnPathRec(queue)
begin
  if null(queue) then return FAIL;
  <x, pathRel> <- out(queue);
  if null(pathRel) then return x;
  R <- car(pathRel);
  listRel <- mulțimea relațiilor semantice definite pe x;
  newEntries <-  $\emptyset$ ;
  if  $R \in listRel$  then
    { y <- ?y: x R y;
      newEntries <- newEntries  $\cup$  {<y, cdr(pathRel)>};
    }
  listParents <- ?y*: x ISA+ y;
; căutarea trebuie să continue însă și din nodurile părinți ale
; lui x;
while (listParents)
  { n <- first(listParents);
    listParents <- cdr(listParents);
    newEntries <- append(newEntries, {<n, pathRel>});
; calea de la nodul start până la oricare din părinții lui x este
; aceeași cu calea de la nodul start până la x
  }

```

```

getValueOnPathRec(in(queue, newEntries));
; În apelul recursiv coada apare incrementată cu noile intrări.
end

```

- Căutare combinată în rețeaua conceptuală și referențială

Acest tip de interogare urmărește regăsirea instanței unui concept plecând dintr-o altă instanță. Pentru aceasta se urcă întâi în rețeaua conceptuală din instanța dată, se determină calea care unește părintele conceptual găsit cu conceptul destinație și se coboară din nou în rețeaua referențială pentru a se determina instanța aflată în capătul căii de relații care au același nume cu cele găsite în rețeaua conceptuală, plecând din instanța dată. Astfel dacă x este instanța dată și C conceptul al cărei instanță se dorește a se afla, răspunsul este dat de următorul lanț de trei interogări:

```

?Cx: x ISA Cx (se cere părintele conceptual al instanței x)
?R*: Cx R* C (se cere calea de relații în rețeaua conceptuală
care unește părintele lui x de conceptul destinație C)
?y: x R* y (răspunsul la întrebare este dat de instanța care se
află în capătul căii omonime de relații, plecând din x)

```

Să urmărim pe Figura 4.9 răspunsul rețelei la întrebarea: *Care este densitatea corpului Cub2?*

```

?CCub2: Cub2 ISA CCub2 → CCub2 = cub
?R*: cub R* densitate → R* = e-făcut-din • are-dens
?y: Cub2 e-făcut-din • are-dens y → y = 0.8

```

4.4.3 Demoni

Demonii sunt proceduri care nu se apelează ci se activează singure când anumite condiții pe care ei sunt pregătiți să le sesizeze sunt îndeplinite. Cu alte cuvinte, un demon este întotdeauna gata să-și ofere serviciile, atunci când cineva are nevoie de ele. Un demon poate fi într-una din stările: **adormit**, **disponibil** (*idle*) sau **activ**.

Când e adormit, demonul nu este la curent asupra modificărilor ce se petrec în lume. Când e disponibil el supraveghează schimbările ce apar și este sensibil la ele, putând să decidă dacă lumea s-a schimbat în așa fel încât serviciile pe care poate el să le pună la dispoziție devin utile. Când este activ demonul lucrează, depune rezultatul în lume, modificând-o, după care revine iar în starea disponibil. Doar un alt proces poate să decidă când un demon disponibil trebuie să adoarmă și când un demon adormit trebuie trezit, pentru a deveni disponibil.

Ca exemplu de utilizare a demonilor în rețele semantice, să ne imaginăm că în rețeaua pe care am proiectat-o deja ne interesează să știm și masa corpurilor. Putem înscrie în rețea aceste informații în două moduri: cântărind corpurile și atașând valorile maselor în nodurile instanță respective, ori construind un demon care devine activ numai când această informație lipsește și care e capabil să deducă masa prin calcul utilizând formula: $m = \rho \cdot V$. Această din urmă variantă presupune interogarea rețelei pentru aflarea densității și a volumului corpurilor. În exemplul

nostru, procedura de calcul a masei, realizată ca un demon, este atașată nodului conceptual *obiect-fizic*, prin intermediul fațetei *demon a proprietății are-masă*. În aceeași manieră se pot atașa proceduri-demoni de calcul a volumelor diferitelor corpuri geometrice. În

Figura 4.9 acest lucru s-a realizat prin atașarea fațetelor *demon proprietăților are-vol* ale nodurile *cilindru* și *cub*.

Pentru a înțelege felul în care un demon preia singur o sarcină spre rezolvare când e disponibil, să vedem mai întâi care e maniera uzuală în care are loc interogarea rețelei, urmărind reprezentarea din Figura 4.9. Astfel, instanței *Cub1* îi este atașată masa în mod explicit, prin relația *are-masă* indicând valoarea 2.500 g. Pentru instanța *Cub2* nu se cunoaște masa direct, dar ea poate fi calculată din faptul că se știe că materialul din care e făcut este lemnul și se cunoaște volumul acestui corp (1.000 cm^3). În sfârșit, pentru *Cilindru1* nu se cunoaște nici masa și nici volumul, dar se știe că e un cilindru făcut din fier, că raza bazei este de 3 cm, iar înălțimea de 10 cm.

Procedurile de tip demon amintite mai sus sunt:

```
procedure ComputeMass(x)
begin
; află densitatea lui x:
  ?Cx: x ISA Cx
  ?R1*: Cx R1* densitate
  ?y1: x R1* y1
; află volumul lui x:
  ?R2*: Cx R2* volum
  ?y2: x R2* y2
; calculează masa ca densitate * volum:
  return y1 * y2;
end
```

```
procedure ComputeVolCube(x)
begin
; află latura lui x:
  ?y: x are-latură y
; calculează volumul:
  return y * y * y;
end
```

```
procedure ComputeVolCylinder(x)
begin
; află raza bazei lui x:
  ?r: x are-rază r
; află înălțimea lui x:
  ?h: x are-înălțime h
; calculează volumul:
  return 3.14 * r * r * h;
end
```

Pentru a exemplifica intervenția demonilor, să încercăm să răspundem la întrebarea: *Ce masă are cilindrul 1?*:


```

?y: Cilindrul ISA y → y = cilindru
?R*, cilindru R* masă → R* = are-masă
?y, Cilindrul are-masă y → nil
  → demon: ComputeMass(Cilindrul)
    → ?Cx: Cilindrul ISA Cx → Cx = cilindru
    → ?R1*: cilindru R1* densitate → e-făcut-din • are-dens
    → ?y1: Cilindrul e-făcut-din • are-dens y1 → 2.4
    → ?R2*: cilindru R2* volum → R2* = are-vol
    → ?y2: Cilindrul are-vol y2 → nil
      → demon: ComputeVolCylinder(Cilindrul)
        → ?r: Cilindrul are-rază r → r = 3
        → ?h: Cilindrul are-înălțime h → h = 10
          ← 282.6
        ← 678.24

```

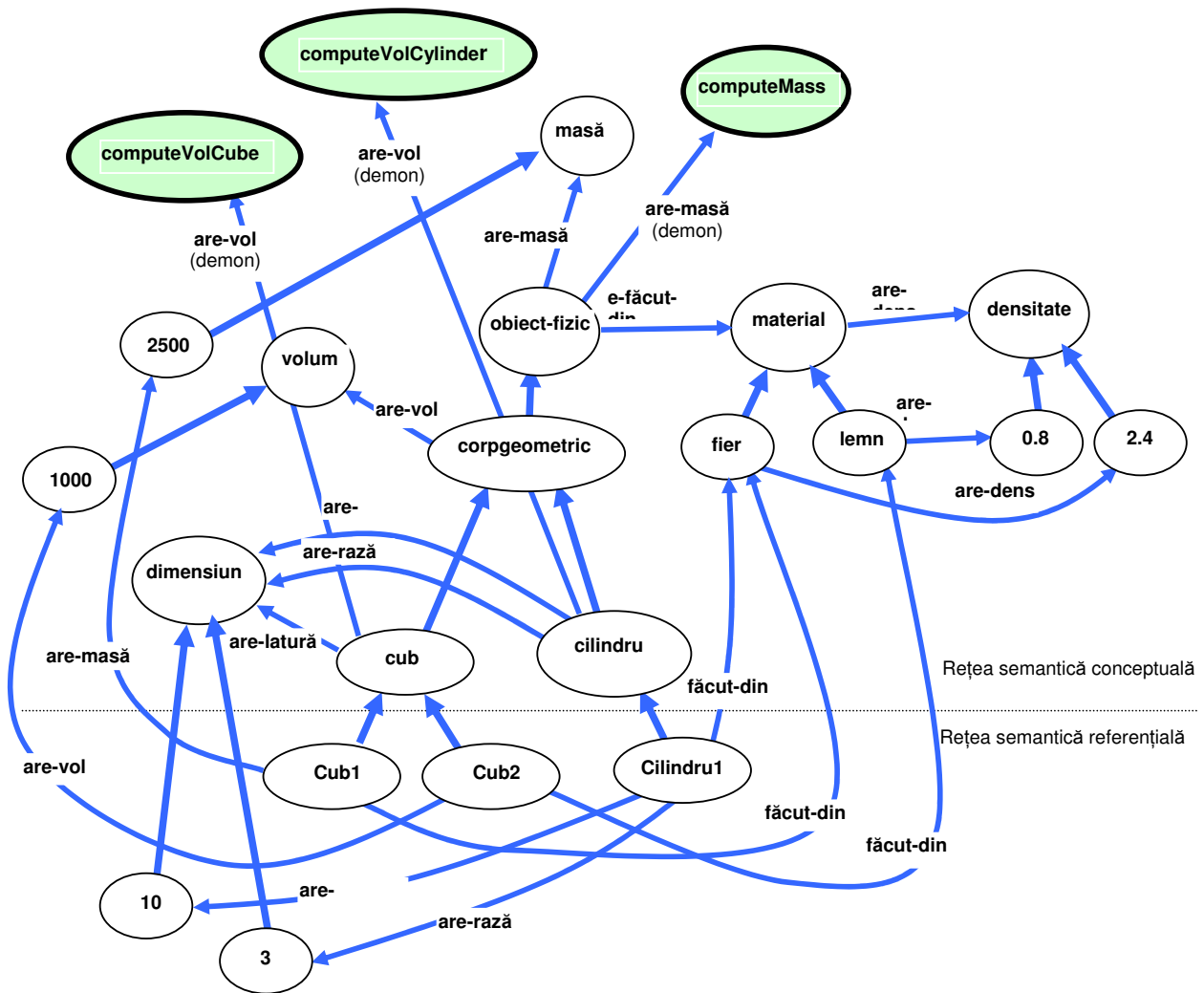


Figura 4.9: Demonii reprezentați ca noduri procedurale în rețele semantice

Având la bază mecanismul descris în această secțiune, în anii '80 a fost construit sistemul IURES¹⁵ (Tufis&Cristea, 1985), (Tufiş et al., 1989), care era capabil să răspundă la întrebări adresate de un utilizator în limba română. Aplicațiile dezvoltate cu IURES au vizat baza de date națională de programe, o bază de date geografice etc.

4.4.4 Ambiguități în reprezentarea prin rețele semantice

Se cunoaște că rețelele semantice prezintă anumite slăbiciuni în modelarea lumilor statice de genul celor prezentate mai sus (Sowa, 1999). Astfel, dacă, ignorând anumite detalii, o relație generală precum "tatăl unei persoane este și ea o persoană" se reprezintă ca în Figura 4.10a, în care, conform convențiilor de mai sus,

¹⁵ Înțeleg Ușor Românește Eliminând Sintaxa (*I Understand and Reply Eliminating Syntax*)

săgeata plină semnifică o relație ISA iar cea subțire – o relație semantică. În aceste condiții să vedem cum putem reprezenta o afirmație precum "tatăl unei persoane o iubește pe mama acelei persoane"?

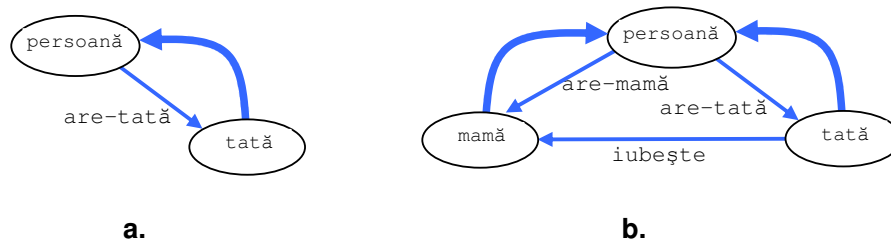


Figura 4.10: Reprezentări pentru a. "tatăl unei persoane este persoană"
b. "tatăl o iubește pe mama"

O reprezentare precum cea din Figura 4.10b poate fi defectuoasă într-o interpretare în care relațiile de la acest nivel sunt înțelese ca fiind moștenite între conceptele ori instanțele aflate pe niveluri ierarhic inferioare. Într-adevăr, presupunând că Ion este un tată iar Maria o mamă, adică o situație ca cea din Figura 4.11, nu am vrea să tragem concluzia că Ion o iubește pe Maria, decât în cazul în care ei ar fi tatăl, respectiv mama, unei aceleiași persoane. În (Thomason, Touretzky, 1991) se prezintă un sistem formal de reprezentare prin rețele semantice care evită ambiguități de genul celor puse în evidență în exemplul care urmează.

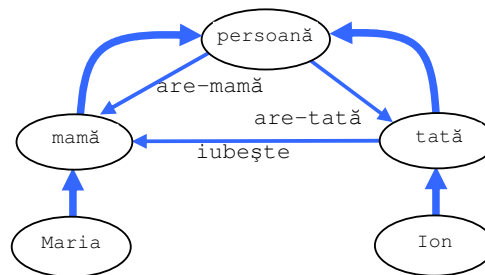


Figura 4.11: Necazuri cu reprezentarea din Figura 4.10b

O altă posibilitate ar fi însoțirea relațiilor de condiții ce ar trebui să fie verificate asupra oricărei perechi de instanțe care particularizează conceptele unite de relație. De exemplu, în cazul acestui exemplu, condiția atașată relației *iubește* ar trebui să fie:

$$\text{iubește}(x, y) \Leftarrow \exists a, \text{ISA}(a, \text{persoană}) \wedge \text{are-tată}(a, x) \wedge \text{are-mamă}(a, y)$$

4.5 Rețele semantice evenimentțiale. Inferențe

Când utilizăm limbajul pentru comunicare, realizăm continuu inferențe. Le facem fără nici un efort aparent; ele rezultă direct din înlănțuirea logică a evenimentelor comunicate, din experiența de viață pe care am acumulat-o și din

cunoașterea sensurilor cuvintelor. Să încercăm să găsim o explicație pentru care următoarele două propoziții citite ori auzite în secvență au sens:

1. *Maria a scăpat oul din mână.*
2. *Ea a trebuit să curețe apoi pardoseala.*

În efortul de a înțelege acest text efectuăm un dublu lanț de inferențe: unul pleacă dinspre prima frază:

Maria scapă oul din mână → când nu mai este susținut de mână oul cade → în cădere oul atinge la un moment dat pardoseala → la atingerea pardoselii, oul se sparge → prin spargere, oul își varsă conținutul pe pardoseală → conținutul lichid al oului se află pe pardoseală → conținutul oului pe pardoseală e perceput de Maria ca o murdărie

iar al doilea lanț de inferențe are ca origine cea de a doua frază:

Ea = Maria trebuie să curețe pardoseala → Maria gândește că se impune curățarea pardoselii → Maria percepe pardoseala ca fiind murdară

În felul acesta găsim un fapt folosit în comun de cele două lanțuri inferențiale: faptul că Maria gândește că pardoseala este murdară, ceea ce explică textul și prin aceasta îl face coerent.

Alte inferențe însoțesc de asemenea acest text: înainte de a fi scăpat, oul era ținut în mână de Maria; Maria nu a avut intenția să scape oul; Maria este necăjită că a pierdut un ou și că a făcut un efort ca să facă iar pardoseala curată. Gândim toate aceste lucruri fără a le folosi în vreun fel, cu excepția cazului în care suntem obligați de apariția altor exprimări ce ni le solicită pentru a fi, la rândul lor, înțelese.

De exemplu, dacă prima fraza a exemplului dat mai sus, ar fi urmată de:

2a. *Ea punea ouăle pe care tocmai le cumpărase din sacoșă în frigider.*

atunci fraza 2a, ca să fie înțeleasă, trebuie să fie însoțită de următoarele inferențe:

1: Maria scapă oul din mână → dacă scap din mână un ou, anterior trebuie să-l fi ținut în mână

2a: *Ea* = Maria transferă ouă din sacoșă în frigider → Maria ia unul sau mai multe ouă în mână și îl (le) depune în frigider → dacă iau în mână un ou înseamnă că țin în mână oul un timp

Din nou două lanțuri inferențiale se unesc într-un fapt comun: Maria ține un ou în mână. Acest fapt contribuie la înțelegerea lui 2a în contextul în care 1 fusese pronunțat.

Cum putem simula aceste procese de inferență?

4.5.1 Evenimente aflate în corelație logică

Implicația logică (*entailment*, în engleză) reprezintă o componentă inferențială esențială. Evenimentele aflate în relație de implicație logică sunt fie simultane, fie în secvență unul față de altul (cel obținut, inferat, este anterior celui presupus) (v. Figura 4.12).



Figura 4.12: Poziționarea în timp a evenimentelor aflate în implicație logică

Exemple de implicații logice:

- dacă scap un obiect din mână înseamnă că anterior țineam acel obiect în mână;
- dacă sforăi înseamnă că dorm;
- dacă iese fum înseamnă că undeva arde ceva;
- dacă îmi cer scuze pentru un lucru înseamnă că admit că am făcut acel lucru;
- dacă sunt așezat înseamnă că anterior m-am așezat.

Relația de implicație logică nu trebuie confundată cu **relația de cauzalitate** între evenimente. În general evenimentele cauză preced temporal pe cele efect: (Figura 4.13).

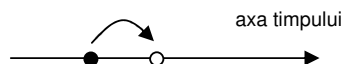


Figura 4.13: Poziționarea în timp a evenimentelor implicate în relație cauzală

- dacă plouă rezultă că pământul va fi ud;
- dacă mă așez înseamnă că în foarte scurt timp voi fi așezat;
- dacă vopsesc un obiect înseamnă că obiectul va avea o altă culoare;
- dacă scap un obiect din mână atunci acel obiect cade.

Există o diferență dintre cauzalitatea din natură și reflectarea ei cognitivă. Întrucât abordarea noastră este una legată de interpretarea textelor, în acest studiu nu ne interesează să modelăm realitatea ci doar maniera în care reflectăm realitatea în procesul lecturii. Ce am dat mai sus sunt câteva reguli cauzale “de bun simț” (*common-sense*) și ele pot neglija unele aspecte legate de fizică. Așa bunăoară, căderea corpurilor presupune un câmp gravitațional de care ținem seama implicit în reprezentările noastre. Dacă însă ne-am plasa într-un mediu în care acesta ar lipsi, am avea nevoie de un timp în care reprezentările noastre mintale să sufere corecții prin confruntare cu experiențele ce contrazic reprezentările. După un timp vom fi înlocuit aceste reprezentări cu altele. De exemplu:

- dacă scap un obiect din mână atunci acel obiect rămâne lângă mână

Al treilea tip de relație logică între evenimente este aceea de **plauzibilitate** (*plausibility*):

- dacă un obiect cade este plauzibil ca un obstacol să oprească căderea la un moment dat;

- dacă un ou este lovit de un anumit obiect dur este plauzibil ca oul să se spargă;
- dacă vorbesc este plauzibil că cineva mă ascultă;
- dacă cineva doarme este plauzibil ca el să sforăie.

Relația de plauzibilitate este una abductivă: ea reprezintă o inversare a relației de implicație logică. Pentru a o explica, cineva a formulat următoarea analogie: imaginați-vă că sunteți în casă și priviți pe fereastră ramurile unui pom aflat în grădină. Dacă cineva scutură pomul veți vedea ramurile lui tremurând. Avem de-a face cu o implicație logică. Acum să presupunem că vedem ramurile pomului tremurând. Dacă presupunem că acest lucru are loc pentru că cineva scutură pomul spunem că am făcut o *abducție*. Concluzia nu este sigură, ci doar probabilă, pentru că ramurile pomului pot să tremure și din cauză că le scutură vântul, de exemplu. Relația de plauzibilitate este mai mult una de analogie. De multe ori la aflarea unui eveniment ne trec prin minte cauze probabile ale lui. Nu toate inferențele de acest tip sunt la fel de probabile. De exemplu, între cele de mai sus, ultima este cel mai puțin probabilă.

Din punct de vedere al dispunerii în timp, evenimentul plauzibil inferat poate fi plasat simultan, anterior sau posterior celui presupus (Figura 4.14).



Figura 4.14: Poziționarea în timp a evenimentelor implicate în relație de plauzibilitate

Revenind asupra exemplului de mai sus detaliem următoarele inferențe:

Maria scapă oul din mână → **CAUSE** → oul cade → **PLAUSIBLE** → oul se lovește de un obiect → **PLAUSIBLE** → oul se sparge → **CAUSE** → conținutul oului se varsă → **PLAUSIBLE** → conținutul oului ajunge pe obiect → **PLAUSIBLE** → Maria percepe conținutul oului pe obiect → **PLAUSIBLE** → Maria gândește conținutul oului pe obiect ca o murdărire a obiectului → **PLAUSIBLE** → Maria este afectată negativ de acest gând...

Alte inferențe sunt de asemenea posibile, dar nu le detaliem aici.

4.5.2 Un model inferențial

În cele ce urmează propunem un model de dezvoltare a inferențelor în rețele semantice, model capabil să dea o explicație înțelegerii textelor. Ne interesează procesele semantice, deci vom ignora orice tratament sintactic aplicat textului. Modelul presupune existența unei **memorii de lungă durată** ce conține reprezentări ale cadrelor evenimentțiale ale diferitelor sensuri ale verbelor. Sensurile verbelor, substantivelor, adjectivelor și adverbilor sunt totodată individualizate în ierarhii conceptuale în maniera WordNet (Fellbaum, 1998).

Modelul simulează conștientizarea “citirii printre rânduri”, a “subînțelegerii” lanțului de evenimente care leagă două situații ori evenimente redade de text, utilizând o **memorie de scurtă durată**. Această memorie este organizată ca o rețea semantică, fiind populată cu noduri evenimentțiale, instanțe ale verbelor-concepte

predefinite în memoria de lungă durată. Evenimentele rămân în această memorie atât cât e nevoie pentru “prinderea” înțelesului, adică realizarea unui lanț inferențial capabil să găsească legătura dintre două fraze ce apar consecutiv în discurs. Odată realizat acest lucru, memoria de scurtă durată este eliberată de nodurile ce au contribuit la acest proces, însă anumite rezultate ce sunt considerate “importante” pot fi eventual transferate în memoria de lungă durată.

Un pas inferențial poate să aibă loc fie direct, prin parcurgerea unei relații implicaționale, cauzale ori plauzibile plecând dintr-un eveniment realizat, fie indirect, parcurgând mai întâi prin moștenire unul sau mai multe niveluri ierarhice din reprezentarea statică a unui tip de eveniment și efectuând apoi o implicație logică, cauzală ori plauzibilă.

În cele ce urmează vom detalia acest proces pentru simularea “înțelegerii” exprimării din exemplul prezentat mai sus. Întâi câteva reguli inferențiale:

RC_scăpa (regulă cauzală cu *a scăpa*): ori de câte ori o persoană scapă un obiect, obiectul va cădea (v. Figura 4.15). Deși, definirea rolurilor de **AG** (agent – cel mare face acțiunea), **REC** (receptor – cel asupra căruia se răsfrânge acțiunea), **OB** (obiectul angrenat în acțiune), este până la urmă o chestiune de convenție, atâta vreme cât nu există reguli specifice general aplicabile unor anumite roluri semantice, vom conveni să considerăm ca participând în postură de **AG** entitățile care sunt activ implicat într-o acțiune sau o situație, în postură de **REC** – pe cele implicate pasiv, iar în postură de **OB** – pe cele care intermediază realizarea evenimentului. Din acest punct de vedere, persoana care scapă obiectul este un activ și deci joacă rolul de **AG**, obiectul scăpat este un pasiv, asupra lui manifestându-se acțiunea de scăpare, deci este un **REC**.

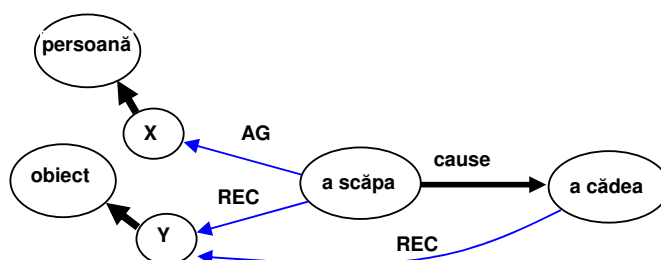


Figura 4.15: O regulă cauzală cu *a scăpa*

RP_cădea (regulă plauzibilă cu *a cădea*): dacă un obiect cade este plauzibil ca după un timp el să atingă repede un obstacol – care este un obiect (v. Figura 4.16).

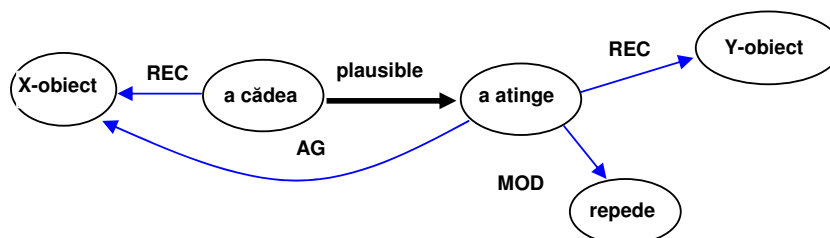


Figura 4.16: O regulă plauzibilă cu *a cădea*

RP_lovi1 (regulă plauzibilă cu *a lovi*): dacă un obiect fragil lovește un alt obiect dur este plauzibil ca obiectul fragil să se spargă (Figura 4.17).

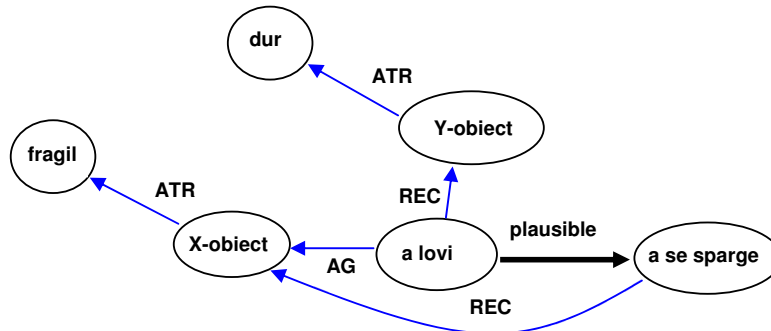


Figura 4.17: O regulă plauzibilă cu *a lovi*

Urmând a ne sluji de ea mai târziu, să mai adăugăm în acest punct o regulă plauzibilă cu *a lovi*, de data acesta în care agentul este un lichid.

RP_lovi2: dacă un lichid lovește un obiect este plauzibil să credem că lichidul va fi întins pe acel obiect (Figura 4.18).

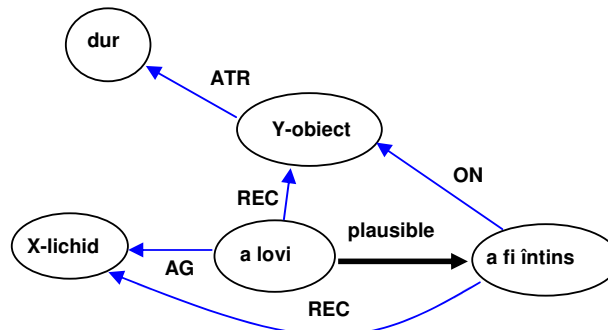


Figura 4.18: Dacă un lichid lovește un obiect, e plauzibil să inferăm că lichidul va fi întins pe obiect

RC_sparge (regulă cauzală cu *a se sparge*): dacă un recipient se sparge și el conține un lichid, este cauzal adevărat că lichidul se va revărsa din recipient (Figura 4.19). Să urmărim marcarea recipientului și a lichidului pe roluri **REC** în evenimentele de spargere și revărsare, ca fiind pasiv implicate în ele.

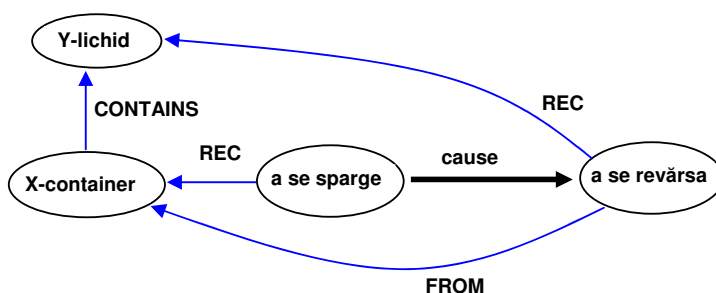


Figura 4.19: O regulă cauzală cu *a se sparge*

Vom vedea imediat că aceste reguli de “recunoaștere a realității din jurul nostru” încă nu sunt suficiente pentru a modela procesul inferențial din exemplul nostru. Deocamdată să mai adăugăm cunoașterea că un ou este (și) un recipient, în sensul de ținător de substanță¹⁶. De asemenea, putem vedea **a se revărsa** ca un fel particular de **a cădea** în care receptorul acțiunii este un lichid. Dacă adoptăm această plasare ierarhică¹⁷, atunci din vărsarea conținutului rezultă prin regula de plauzibilitate **RP_cădea** că acesta va atinge un obstacol. Inferența este una indirectă și rezultă din compunerea a doi pași: unul de moștenire și celălalt cauzal.

În continuare vom detalia maniera în care aceste reguli se pot combina pentru a forma un lanț inferențial.

1. *Maria a scăpat oul din mână.*

Prima propoziție provoacă apariția unui eveniment cu verbul *a scăpa* (Figura 4.20). Apariția unui eveniment cu **a scăpa** va amorsa un proces care verifică *pattern*-urile inferențiale (implicaționale, cauzale și plauzibile) atașate acestui verb.

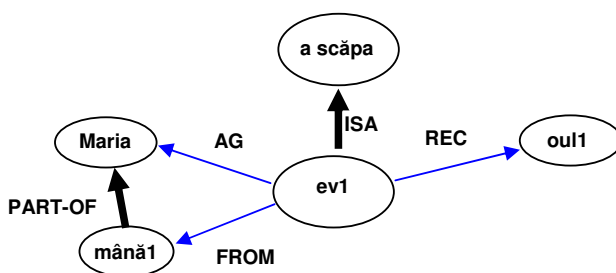


Figura 4.20: Evenimentul nou apărut *ev1*

¹⁶ În WordNet 1.6 o astfel de informație lipsește. Un ou e privit numai din punctul de vedere biologic, ca celulă ce poate da viață. Informația morfologică, ce pune în evidență constituția sa de “ținător” de substanță vâscoasă, este ignorată.

¹⁷ Din nou WordNet 1.6 nu clasifică **a vărsa** (*leak, flow, pour*) ca fiind în relație de hiponimie cu **a cădea** (*to fall*).

Procesul general este acela în care apariția unui eveniment al unui verb (concept) provoacă apariția unor evenimente legate prin relații inferențiale de acel verb. Funcționarea în acest mod poate fi pusă pe seama unei meta-reguli de forma:

MRCause

```
if isEvent(ev1, conc1) and isCauseRule(RC, conc1, conc2) and
verifies(ev1, RC) then born(ev2, conc2)
```

Această meta-regulă (*meta* – pentru că ea se aplică unei clase întregi de reguli cauzale) spune că dacă apare un eveniment ev_1 , instanță a unui concept $conc_1$ (*isEvent* – este un predicat cu comportament de demon, el verificând existența unui eveniment dintr-o clasă dată) și dacă există o regulă cauzală RC în care conceptul $conc_1$ apare ca inițiator și conceptul $conc_2$ ca rezultat și dacă evenimentul ev_1 verifică restricțiile semantice ale părții stângi a regulii RC , atunci se va crea un eveniment ev_2 , instanță a conceptului $conc_2$, împreună cu toate legăturile și obiectele noi implicate de instanțierea părții drepte a regulii RC .

Meta-regula **MRCause** poate fi aplicată la apariția evenimentului **ev1** al verbului (concept) **a scăpa** (*isEvent*(ev_1 , $conc_1$) este instanțiat de $ev_1 = ev1$ și, $conc_1 = a scăpa$) pentru că există o regulă cauzală a lui **a scăpa** (*isCauseRule*(RC , $conc_1$, $conc_2$) este verificată de $RC = RC_scăpa$, $conc_1 = a scăpa$, $conc_2 = a cădea$), și restricțiile semantice atașate rolurilor **AG** și **REC** sunt verificate de, respectiv, **Maria**, care este o **persoană**, și **oul1**, care este un **obiect** (adică *verifies*($ev1$, $RC_scăpa$) se evaluează la **true**). Activarea meta-regulii **MRCause** va determina apariția unui nou eveniment, **ev2**, care trebuie să fie unul al verbului **a cădea** și în care rolul **REC** trebuie să fie satisfăcut de aceeași entitate care satisface rolul **REC** al evenimentului precedent cu **a scăpa**, adică **oul1** (v. Figura 4.21).

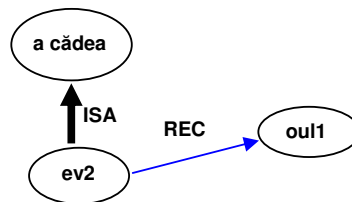


Figura 4.21: Evenimentul nou apărut $ev2$

În mod analog pot fi puse în evidență alte două meta-reguli, care descriu respectiv inferențele implicaționale și pe cele plauzibile:

MRImplies

```
if isEvent(ev1, conc1) and isImPLYRule(RI, conc1, conc2) and
verifies(ev1, R) then born(ev2, conc2)
```

MRPlausible

```
if isEvent(ev1, conc1) and isPlausibleRule(RP, conc1, conc2) and
verifies(ev1, R) then born(ev2, conc2)
```

Mai departe, apariția unui eveniment al verbului **a cădea**, prin efectul meta-regulii **MRPlausible**, atrage declanșarea regulii **RP_cădea** (v. Figura 4.16). Aceasta duce la crearea (cu un anumit grad de incertitudine, datorat naturii plauzibile iar nu sigure a ei) a unui eveniment al verbului **a atinge** (v. Figura 4.22). În acest fel un obiect (necunoscut deocamdată) este creat. El este notat aici cu **X-obiect**.

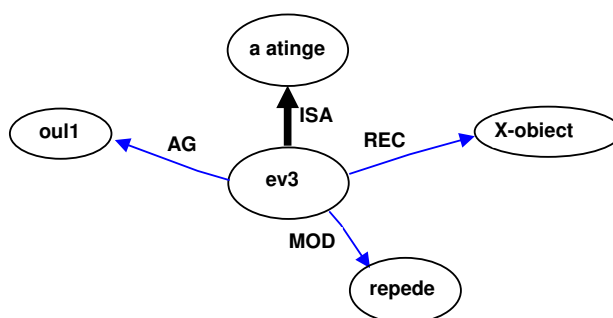


Figura 4.22: Evenimentul nou apărut ev3 al conceptului a atinge

Până acum trei evenimente noi populează spațiul inferențial al memoriei de scurtă durată. Apariția lor simulează conștientizarea situațiilor de scăpare din mână a oului, de cădere a acestuia cât și a faptului că, în cădere, acesta atinge un obiect. Să observăm că prima propoziție nu pune în evidență obiectul de care se va lovi oul: la fel de bine el ar fi putut fi un scaun, colțul mesei sau pălăria unui trecător aflat sub balconul pe care s-ar fi putut petrece acest episod...

Cât de departe trebuie să mergem cu lanțul inferențial? Cât de multe inferențe facem? Oricâte, sau numai câte sunt necesare pentru înțelegerea textului. Există o limită rezonabilă unde am putea opri acest lanț? Preocupați, în textul de față, de găsirea unui model al înțelegerii în profunzime a textelor, vom ignora deocamdată întrebări de acest gen, ce țin de "gestionarea" procesului inferențial¹⁸. Preocuparea noastră, deocamdată, este de a arăta că un astfel de lanț se poate închide firesc pe un rezultat care include una dintre accepțiunile posibile ale textului.

Un mod particular de atingere este lovirea. Astfel WordNet 2.1¹⁹ raportează un sens al lui *to hit* ca troponim (sens particular) al verbului *to touch* (invers, *to touch* este considerat un hiperonim al lui *to hit*):

```

touch -- (make physical contact with, come in contact with;
"Touch the stone for good luck"; "She never touched her
husband"; "The two buildings almost touch")
=> strike, hit -- (produce by manipulating keys or strings of
musical instruments, also metaphorically; "The pianist strikes

```

¹⁸ Un tip de control poate fi acela în care inferențele se desfășoară tot timpul, în paralel cu citirea textului. Fiecare frază amorsează noi lanțuri inferențiale iar unele lanțuri aflate în desfășurare se pot închide atunci când se regăsesc evenimente introduse deja de alte lanțuri. Aceste evenimente, apărute redundant, constituie semnale că un înțeles a fost construit. Scoruri de "satisfacție" pot fi imaginate care să dea o imagine a gradului de înțelegere a textului.

¹⁹ Tezaurul lexical Wordnet, construit la Universitatea din Philadelphia, poate fi consultat și *download*-at la adresa <http://www.upenn.edu/~wn>.

```
a middle C"; "strike `z' on the keyboard"; "her comments struck
a sour note")
```

O manieră generală în care dintr-un concept mai general se poate obține, prin particularizare, unul mai specific, este dată de următoarea meta-regulă:

MRHypernim

```
if isEvent(ev1, conc1) and isHypernimDueToRoles(conc1, conc2,
Roles) and verifiesRoles(ev1, Roles) then born(ev2, conc2)
```

care spune că dacă există un eveniment ev_1 aparținând conceptului $conc_1$ și dacă $conc_1$ este un hiperonim cunoscut al conceptului $conc_2$ și dacă particularizarea lui $conc_1$ în $conc_2$ se face prin rolurile $Roles$ cu anumite valori atașate și dacă evenimentul ev_1 verifică rolurile $Roles$, atunci se crează un eveniment ev_2 al conceptului mai particular $conc_2$ în care apar de asemenea valorile rolurilor $Roles$. Figura 4.23 arată că rolul care particularizează **a atinge** în **a lovi** este **MOD** cu valoarea **repede**.

Ca urmare a aplicării metaregulii de hiperonimie, apariția evenimentului **ev3** duce de asemenea la apariția evenimentului **ev4**, al conceptului **a lovi** (v. Figura 4.24). În acest eveniment, un obiect de care oul s-a lovit în cădere este presupus (notat în Figura 4.24 cu **X-obiect**). Încă nu știm ce poate fi acest obiect. Identitatea lui va fi relevată mult mai târziu, în virtutea menționării podelei în fraza 2 din exemplul nostru.

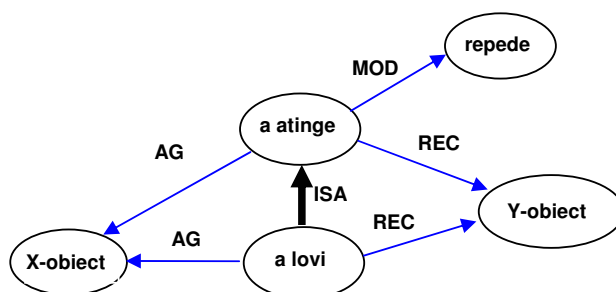


Figura 4.23: A lovi înseamnă a atinge repede

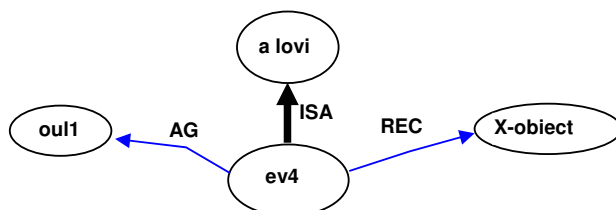


Figura 4.24: Avem acum un eveniment cu a lovi

În continuare, în virtutea regulilor **RP_lovi** și **RC_sparge** și a metaregulilor **MR-Plausible** și **MR_Cause** cât și a cunoașterii faptului că un ou este un obiect fragil de tip container care conține un lichid, două noi evenimente sunt construite (v. Figura 4.25).

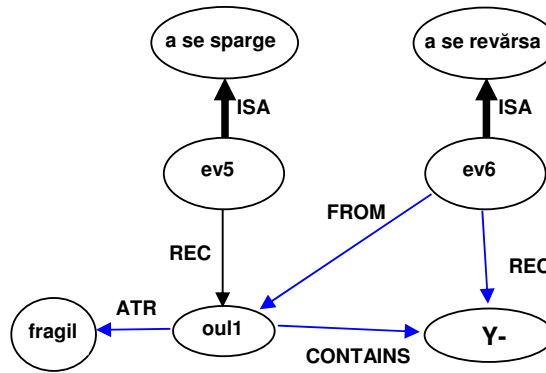


Figura 4.25: Conștientizăm spargerea oului (ev5) și revărsarea conținutului său (ev6)

Pentru a exemplifica maniera în care o regulă de bun simț se poate aplica de mai multe ori, vom face presupunerea că dispunem de cunoașterea de semantică lexicală care clasează verbul **a se revărsa** ca o formă particulară de **cădere**, ca în Figura 4.26.

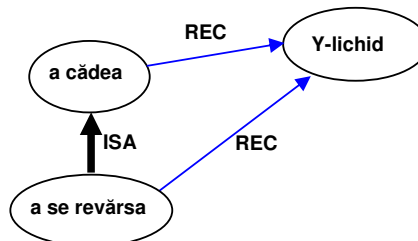


Figura 4.26: Dacă un lichid cade înseamnă că el se revărsa

În virtutea aplicării meta-reguli **MRHypernim** din nou, existența evenimentului **ev6** va duce la apariția unui eveniment al verbului **a cădea**, iar aplicarea recursivă a regulii de bun simț **RP_cădea**, va da naștere, în aceeași manieră ca și mai înainte, la un eveniment cu **a atinge** (v. Figura 4.27).

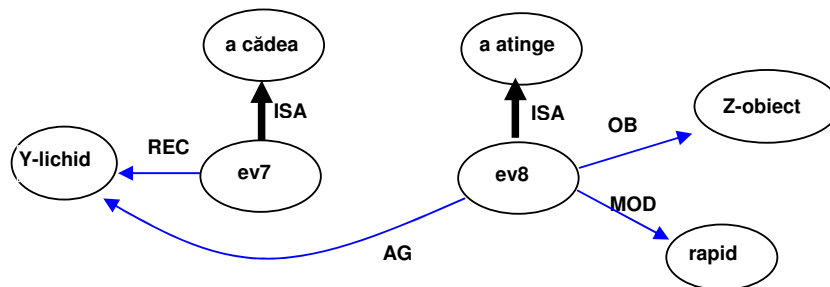


Figura 4.27: Conținutul oului cade și atinge (rapid) un obiect

Să remarcăm că **X-obiectul** de care se lovește oul nu este necesar să fie același cu **Z-obiectul** de care se lovește lichidul. Mai departe însă, pentru că **AG**-ul evenimentului cu **a atinge** nu mai este un obiect ci un lichid, regula **RP_lovi** nu poate fi aplicată încă o dată. O altă regulă pragmatică trebuie să funcționeze, capabilă să infereze că dacă un lichid lovește un obiect, atunci lichidul se întinde pe obiect (v. **RP_lovi2** și Figura 4.28). Ca urmare o situație în care conținutul oului este întins pe acest obiect pe care l-a atins în cădere va apare în memoria de scurtă durată:

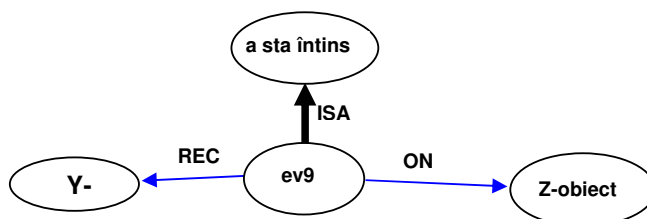


Figura 4.28: Conținutul oului este întins pe un obiect

Exemplul poate fi continuat până la atingerea unei stări în care Maria gândește conținutul oului pe obiect ca o murdărire a obiectului și ea este afectată negativ de acest gând. În mod analog, un proces va fi inițiat din fraza a doua a exemplului considerat. Cum un proces de curățare este datorat perceperii unei murdării, iar necesitatea de a o curăța, unei stări negative insuflată de ea, va exista un moment în care cele două lanțuri inferențiale se ating pe o stare comună. Acest fapt comun, indus de o frază și presupus de cealaltă, este liantul care face ca exprimarea să fie coerentă. Când un astfel de proces se derulează într-o mașină, atunci suntem îndreptățiți a spune că mașina a „înțeles” exprimarea.

Cerințe pentru studenți:

- Să fie capabili să producă o reprezentare a cunoașterii dintr-un univers dat.
- Să recunoască necesitatea utilizării unei rețele semantice descriptive sau a uneia evenimentiale, în funcție de problemă.
- Să poată adapta ori produce un algoritm care să ofere posibilitatea obținerii de inferențe în rețele semantice.

Probleme și proiecte de casă

P4.1 Care dintre următoarele propoziții vi se par importante în definirea rețelor semantice (RS), care sunt adevărate și care false:

- RS descriu relații dintre obiecte reprezentate prin noduri;
- nodurile în RS sunt cercuri cu nume;
- cea mai importantă trăsătură a RS este moștenirea proprietăților;
- cea mai importantă trăsătură a RS este moștenirea multiplă;
- ceea ce deosebește RS de logica predicatelor de ordinul I (LPOI) este că în RS pot exprima Toate păsările zboară cu excepția pinguinului, pe când în LPOI nu;
- atât în LPOI cât și în RS pot deduce din „Orice om e muritor. Socrates este om.” că „Socrates este muritor.”

P4.2 Să se reprezinte prin RS:

- *Un Boeing 747 este un avion.*
- *Avioanele și păsările zboară.*
- *Un vultur este o pasăre.*
- *Avioanele au motoare iar păsările pene.*
- *Avioanele au pilot și păsările cioc.*
- *Atât avioanele cât și păsările folosesc principii aerodinamice.*

P4.3 Știți următoarele fapte despre lume:

- *cele mai multe ființe nu zboară;*
- *cele mai multe păsări, ființe fiind, zboară;*
- *pinguinii și struții, păsări fiind, nu zboară;*
- *struții magici însă zboară;*
- *Birco este o pasăre;*
- *Zuicu este fie un pinguin, fie un struț;*
- *Croț este un struț magic.*

a). Dați o reprezentare a acestor cunoștințe prin rețele semantice descriptive.

b). Aplicând reguli de moștenire nemonotonă, răspundeți apoi la întrebările:

- Birco zboară?
- dar Zuicu?
- dar Croț?
- dar Moțu?

P4.4 Știți următoarele fapte despre lume și numai pe acestea:

- *bicicletele cântă, sunt prevăzute cu spițe și împletesc ciorapi;*
- *caii, biciclete fiind, văd dar nu împletesc ciorapi pentru că sunt prevăzuți cu șei;*
- *ochelarii, biciclete fiind, văd dar nu cântă pentru că nu sunt prevăzute cu spițe;*
- *ochelarii de soare însă nu văd;*
- *și oricine știe că lucrurile nu văd și nu pot cânta;*
- *bicicletele, ca orice din lumea asta, sunt lucruri.*

Să se enunțe proprietățile calului lui Nero, ale bicicletei roșii din holul Facultății, ale ochelarilor de soare ai domnului decan și ale Dunării albastre.

P4.5 Să se reprezinte cu ajutorul rețelor semantice cunoașterea din următorul text:

Aorta este un tip particular de arteră care are un diametru de 2.5 cm. O arteră este un vas de sânge, cu perete muscular și diametrul de 0.4 cm. O venă e un vas de sânge cu pereți fibroși. Vasele de sânge au formă tubulară și conțin sânge.

P4.6 Informații asupra arborelui genealogic al unei familii sunt descrise ca o rețea semantică în care nodurile sunt persoane și legăturile sunt către părinți și (eventual) către fii. Pentru orice persoană se cunoaște sexul. Desenați rețeaua pentru arborele genealogic:

Lili si Axinte sunt copiii lui Leana si Costel. Florin si Ion sunt fiii lui Lili si Vasile. Ioana e fata lui Maria si Axinte, iar Cornel e fratele său. Pe unchiul lui Cornel îl cheamă Gigel.

- a). Descrieți un mod de reprezentare a rețelei semantice prin liste LISP.
- b). Scrieți funcții LISP care să permită aflarea bunicilor și a verilor unei persoane.

P4.7 Să se completeze rețeaua semantică relativă la corpuri geometrice dată la curs cu o piramidă `PIRAM1` făcută din aluminiu (greutate specifică 2.7 g/cm^3) și de volum 50 cm^3 . Să se arate care sunt interogările lansate asupra rețelei de un apel al demonului `computeMass (PIRAM1)`.

P4.8 Să se reprezinte cunoștințele din fraza următoare folosind rețele semantice evenimentțiale:

Maria crede că lui Ion nu îi place să înoate, de aceea nu merge în vacanță la mare.

P4.9 Citiți următorul text:

Maria a văzut un om intrând în casa ei.

Ea a chemat Poliția.

- a. Dați o reprezentare prin rețele semantice evenimentțiale a acestor două fraze.
- b. Precizați un set de reguli inferențiale care, aplicate în secvență, să dea sens acestui text.

P4.10 Analizați-vă reacțiile vis-à-vis de cele două secvențe de mai jos. Explicați motivele acestor reacții. Construiți un sistem capabil să reacționeze la fel ca dv.

*A. Mihai s-a dus la stație să facă plinul
dar benzinarul i-a spus că nu mai are benzină.*

*B. Mihai s-a dus la stație să facă plinul
dar benzinarul i-a spus că nu mai are benzină în mașină.*

Bibliografie

Barr, A. and Feigenbaum, E.A. 1981. *Handbook of Artificial Intelligence*, (Eds.), William Kaufman, Inc., Los Altos, California, 409 pp, ISBN 0-86576-005-5, vol 1.

Cristea, D. 2002. *Să ne jucăm cu cuvintele. Proiecte de prelucrare a limbajului natural* - 1, Revista de Informatică, nr. 1, Iași.

Fellbaum, C. (ed.) 1998. *WordNet. An Electronic Lexical Database*, The MIT Press.

Gruber, T.R. 1993. *Toward principles for the design of ontologies used for knowledge sharing*. În „Formal Ontology in Conceptual Analysis and Knowledge Representation”, edited by Nicola Guarino and Roberto Poli, Kluwer Academic Publishers, Accesibilă la adresa: <http://www.cise.ufl.edu/~jhammer/classes/6930/XML-FA02/papers/gruber93ontology.pdf>

Gruber, T.R. 2003. *What is an Ontology*. V. <http://www-ksl.stanford.edu/kst/what-is-an-ontology.html>

Sowa, J. F., 1999. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks Cole Publishing Co.

Thomason, R.H., Touretzky, D.S. 1991. *Inheritance Theory and Networks with Roles*, în John Sowa (ed.): „Principles of Semantic Networks. Explorations in the Representation of Knowledge”, Morgan Kaufmann Publishers, Inc., San Mateo.

Tufiș, D.; Cristea, D. 1985. *IURES: A Human Engineering Approach To Natural Language Question-Answering Systems*. În Bibel, W.; Petkoff, B. (Eds.) „Artificial Intelligence. Methodology, Systems, Applications”. North-Holland, Amsterdam, pag.177-184.

Tufiș, D., Giumale, C., Cristea, D. 1989. *Lisp*. Ed. Tehnică, București, vol 2.

Capitolul 5

Probleme de satisfacere a constrângerilor

Atunci când nu faci greșeli, nu rezolvi probleme îndeajuns de complexe. Și asta e o mare greșeală.

Frank Wilczek

5.1 Introducere

Faptul că satisfacerea constrângerilor s-a configurat actualmente ca un domeniu important și de sine stătător în informatica teoretică și aplicată este datorat multitudinii de situații din lumea reală care pun pe tapet probleme ce impun respectarea simultană a mai multor cerințe. O listă a domeniilor care abundă în astfel de probleme trebuie să includă: planificarea automată, configurarea resurselor, design, diagnosticare, raționare temporală și spațială etc.

Satisfacerea constrângerilor este acel subdomeniu al inteligenței artificiale care încearcă determinarea unei soluții practice cât mai bune dată fiind o listă de constrângeri și priorități.

Formal, o problemă de satisfacere a constrângerilor (*eng.*: *constraint satisfaction problem – CSP*) este definită printr-o **rețea de constrângeri**. O rețea de constrângeri constă dintr-o mulțime de variabile $X=\{X_1, \dots, X_n\}$ și o mulțime de constrângeri $C=\{C_1, \dots, C_l\}$. Fiecare variabilă X_i poate lua valori dintr-un domeniu D_i . O constrângere C_i este o relație R_i definită pe o submulțime de variabile și care determină asignări legale de valori. Rețeaua este referită printr-un triplet $R=(X, D, C)$. O soluție pentru o astfel de problemă este o asignare de valori variabilelor astfel încât toate constrângerile să fie satisfăcute (Dechter, 2003; Tsang, 1993).

Ca o observație, în definiție nu se impune nici o condiție asupra tipului variabilelor. Acestea pot fi întregi, logice, mulțimi, sau de orice alt tip. De asemenea, nici modul de definire a constrângerilor nu este limitat. Constrângerile pot fi date atât explicit, prin specificarea tuplelor de valori permise, cât și implicit, prin relații (ex: $X_i > 2$).

O problemă pentru care există o soluție se numește *satisfiabilă* sau *consistentă*. În caz contrar, ea se numește *nesatisfiabilă* sau *inconsistentă*. Există situații în care se dorește determinarea tuturor soluțiilor, sau doar a uneia sau, în caz de inconsistență, specificarea acestui lucru. În practică, deseori este greu de determinat o asignare care să satisfacă toate constrângerile. O extensie a problemei CSP este problema Max-CSP în care scopul este găsirea unei asignări cu număr minim de constrângeri violate. În cazul în care constrângerile implică cel mult două variabile, numim problema CSP-binară.

O problemă de satisfacere a constrângerilor poate fi reprezentată printr-un graf, numit *graf constrâns*. Pentru fiecare variabilă este asociat un nod, iar un arc este trasat între fiecare pereche de variabile conținute într-o constrângere.

Exemplul 5.1: problema reginelor

Dată fiind o tablă de șah $n \times n$ dorim să plasăm n regine pe tablă astfel încât nici o regină să nu fie atacată de nici o altă regină.

O posibilă formulare a problemei reginelor ca o problemă CSP ar fi următoarea: pentru fiecare coloană a tablei de șah asociem o variabilă X_i , iar domeniul variabilei sunt liniile, adică $D_i = \{1, \dots, n\}$. Constrângerile sunt asociate fiecărei perechi de coloane și precizează următorul fapt: două regine nu se pot afla pe aceeași linie sau pe aceeași diagonală, ceea ce se exprimă prin relațiile:

$$X_i \neq X_j$$

$$|X_i - X_j| \neq |i - j|, \text{ pentru oricare } i, j \text{ din intervalul } 1, \dots, n.$$

Particularizând la $n = 4$, se obține problema celor 4-regine, în care variabilele sunt $\{X_1, X_2, X_3, X_4\}$, fiecare având domeniul $\{1, 2, 3, 4\}$. Constrângerile sunt $C_1 = R_{12}$, $C_2 = R_{13}$, $C_3 = R_{14}$, $C_4 = R_{23}$, $C_5 = R_{24}$, și $C_6 = R_{34}$, date, în formă extensională, ca în Figura 5.1.

	X_1	X_2	X_3	X_4	
1		Q			$R_{12} = \{(1,3), (1,4), (2,4), (3,1), (4,1), (4,2)\}$
2				Q	$R_{13} = \{(1,2), (1,4), (2,1), (2,3), (3,2), (3,4), (4,1), (4,3)\}$
3	Q				$R_{14} = \{(1,2), (1,3), (2,1), (2,3), (2,4), (3,1), (3,2), (3,4), (4,2), (4,3)\}$
4			Q		$R_{23} = \{(1,3), (1,4), (2,4), (3,1), (4,1), (4,2)\}$
					$R_{24} = \{(1,2), (1,4), (2,1), (2,3), (3,2), (3,4), (4,1), (4,3)\}$
					$R_{34} = \{(1,3), (1,4), (2,4), (3,1), (4,1), (4,2)\}$

Figura 5.1: Problema celor 4-regine ca o problemă de satisfacere a constrângerilor

Graful constrâns este complet deoarece poziția unei regine pe o coloană influențează pozițiile valide ale reginelor pe restul coloanelor.

Exemplul 5.2: Problema colorării hărții

Dată o hartă cu n țări, să se asigneze câte o culoare dintr-o mulțime dată fiecărei țări, astfel încât țările vecine să aibă culori diferite.

Asignăm fiecărei țări de pe hartă o variabilă care va avea ca domeniu mulțimea de culori. Între două țări vecine în graful constrâns vom adăuga un arc.

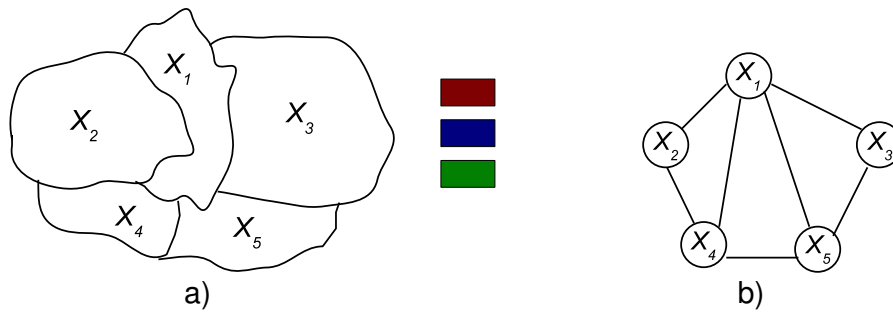


Figura 5.2: a) Problema colorării unei hărți cu 5 țări, culorile posibile fiind roșu, albastru și verde b) Graful constrâns

Exemplul 5.3: Sudoku

Cele mai răspândite puzzle-uri Sudoku sunt cele de ordin 3. Un astfel de puzzle constă dintr-o tablă de 9×9 , în care fiecare pătrat poate avea un număr de la 1 la 9. Dată o asignare parțială a tablei, scopul este de a completa pozițiile rămase libere astfel încât fiecare număr să apară o singură dată pe o linie, pe o coloană și într-o regiune 3×3 .

O posibilă modelare ca o problemă CSP: fiecare poziție a tablei este reprezentată de o variabilă, iar domeniul unei variabile este un număr de la 1 la 9. Pentru fiecare linie, coloană și bloc 3×3 vom asocia o constrângere de tip *alldifferent*, care specifică că toate variabilele din constrângere trebuie să aibă valori diferite de restul variabilelor.

2								
	8	9						
		7						
1	4			7		2	5	
	9	2				8	6	
	3	5		6			9	7
6				1		3		
	1	8				6	2	
	2		6	4				5

Figura 5.3: O tablă Sudoku

Când există preferințe între soluții, acestea pot fi exprimate cu ajutorul unei funcții de cost, numită și funcție obiectiv. Scopul problemei este de a determina o soluție cu costul cel mai bun sau o aproximare a acesteia. Astfel de probleme se numesc probleme de optimizare a constrângerilor (*constraint optimization*).

În general problemele de satisfacere a constrângerilor sunt din punct de vedere computațional intractabile (*NP-hard*). Tehnicile utilizate în rezolvarea unor astfel de probleme se împart în două mari categorii: căutare și inferență. Algoritmii de căutare traversează spațiul soluțiilor parțiale construind o instanțiere completă care satisface toate constrângerile, sau determină inconsistența problemei. Din cadrul acestor tehnici fac parte schemele de backtracking. Algoritmii de inferență a

consistenței modifică la fiecare pas problema pentru a o face mai explicită, prin urmare mai ușor de rezolvat. Există algoritmi care combină cele două metode și care dau rezultate mai precise.

5.2 *Backtracking* în satisfacerea constrângerilor

Putem aplica un algoritm de *backtracking* pentru parcurgerea în adâncime (DFS) a arborelui de căutare. Ordinea variabilelor poate fi fixată înainte sau poate fi determinată la execuție. Algoritmul menține de-a lungul execuției o mulțime de variabile instanțiate corect, adică o soluție parțială pe care o extinde pas cu pas. Inițial, mulțimea este vidă. La fiecare pas se selectează următoarea variabilă din ordonare și se încearcă asignarea variabilei cu o valoare consistentă cu instanțierea parțială. Dacă este găsită o astfel de valoare, algoritmul continuă procedeul cu următoarea variabilă. În caz contrar, algoritmul se întoarce la variabila anterioară și îi asignează o altă valoare consistentă. Dacă domeniul unei variabile devine vid, atunci nu există o soluție care să satisfacă toate constrângerile.

Algoritmul este descris mai jos. Intrarea algoritmului este rețeaua de constrângeri $R=(X, D, C)$. Mulțimile D_i' păstrează valori din domeniul D_i care nu au fost încă examinate pentru instanțierea parțială curentă. În cazul în care domeniile sunt mulțimi ordonate de întregi nu mai sunt necesare aceste mulțimi, ci doar un indicator care să specifice poziția până la care au fost considerate valorile.

```

procedure BACKTRACKING ( $R$ )
begin
     $i \leftarrow 1$ ;
     $D_i' \leftarrow D_i$ ;
    while  $1 \leq i \leq n$ 
    {  $X_i \leftarrow \text{SELECTEAZĂ-VALOARE}()$ ;
      if  $X_i = \text{null}$  then  $i \leftarrow i-1$ ; (întoarcere)
      else
        {  $i \leftarrow i+1$ ; (înaintare)
           $D_i' \leftarrow D_i$ ;
        }
      }
    if  $i = 0$  then return "problemă inconsistentă";
    else return instanțierile variabilelor  $X_i$ ;
end

procedure SELECTEAZĂ-VALOARE()
begin
    while  $D_i'$  nu e vid
    { selectează aleator o valoare  $a$  din  $D_i'$ ;
       $D_i' \leftarrow D_i' \setminus \{a\}$ ;
      if asignarea ( $X_i = a$ ) este consistentă cu  $(a_1, \dots, a_i)$  then
        return  $a$ ;
      }
    return null;
end

```

Algoritmul are complexitatea timp exponențială și complexitatea spațiu liniară.

Acțiunile algoritmului de căutare pot fi descrise de un arbore de căutare. În Figura 5.4 este schițat subarborele asociat problemei de colorare a hărții din exemplul 5.2 pentru o asignare parțială.

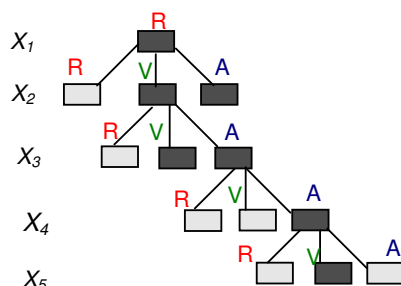


Figura 5.4: Subarborele de căutare pentru cazul în care nodul X_1 este colorat cu roșu (R), X_2 cu verde (V) și X_3 cu albastru (A); nodurile întunecate nu reprezintă asignări valide

Îmbunătățirile cunoscute ale algoritmului s-au focalizat pe cele două faze ale algoritmului: pasul de înaintare (schemele *look-ahead*) și pasul de întoarcere (schemele *look-back*). Dintre aceste extensii amintim:

- *backmarking*: reduce numărul de verificări ale consistenței;
- *backjumping*: îmbunătățește alegerea variabilei pentru pasul de întoarcere;
- *forward-checking*: verifică ca valoarea selectată pentru noua variabilă să fie compatibilă cu valorile variabilelor viitoare.

O altă direcție de îmbunătățire o constituie euristici de ordonare a variabilelor, statice și dinamice pentru selectarea variabilei următoare. Există și câteva abordări care "învață" prin înregistrarea de constrângeri adiționale de-a lungul căutării. Exemplificăm în continuare câteva din aceste tehnici.

5.2.1 Backjumping

Unul din dezavantajele algoritmului *backtracking* este așa zisul mecanism de *trashing*: aceeași situație de blocaj (*dead-end*) poate fi întâlnită de mai ori. O situație de blocaj apare atunci când nu există o valoare consistentă din domeniul noii variabile cu asignarea parțială anterioară. Dacă X_i este variabila la care apare blocajul, algoritmul de *backtracking* se va întoarce la variabila anterioară X_{i-1} . Să presupunem că nu există nici o constrângere între variabilele X_i și X_{i-1} și că există o nouă valoare pentru X_{i-1} . Același blocaj va fi întâlnit până când vor fi epuizate toate valorile lui X_{i-1} .

Pentru a reduce astfel de verificări inutile, a fost propusă o nouă schemă de *backtracking* numită *backjumping* (Gaschnig, 1979; Dechter, 1990). Acest algoritm se întoarce la variabila care cauzează blocajul. Identificarea unei astfel de variabile se bazează pe noțiunea de *multiple conflict*. O instanțiere consistentă (a_1, \dots, a_i) este o mulțime conflict pentru variabila neinstanciată X (sau (a_1, \dots, a_i) este în conflict cu X) dacă nici o valoare din domeniul lui X nu este consistentă cu asignarea (a_1, \dots, a_i) . Când se ajunge într-o situație de blocaj, este recomandat să ne întoarcem cât mai

mult posibil, fără însă a omite posibile soluții. O variabilă este responsabilă pentru situația de blocaj (*culprit variable*) dacă instanțierea (a_1, \dots, a_b) este o mulțime conflict minimală, adică indicele b este cel mai mic indice cu proprietatea $b \leq i$, pentru care a_b este în conflict cu X_{i+1} . Variabila cauză este sigură și optimală; sigură în sensul că nu poate fi extinsă la o soluție și optimală deoarece întorcându-ne la un nod dinaintea acesteia riscăm să pierdem soluții.

Algoritmul *backjumping* al lui Gaschnig este una din metodele care implementează această idee. Pentru a localiza o astfel de variabilă, algoritmul utilizează următoarea tehnică de marcare: pentru fiecare variabilă X_i este reținut un pointer $ultim_i$ la cea mai recentă variabilă pentru care s-a testat consistența cu X_i și care are o valoare în conflict cu o valoare din domeniul lui X_i . Pe măsură ce este completată asignarea parțială, se înregistrează o serie de informații care vor fi utilizate pentru a determina variabila ce cauzează blocajul. Dacă pentru asignarea (a_1, \dots, a_i) există o valoare consistentă atunci $ultim_i$ va fi egal cu $i-1$. Când ajungem într-o situație de blocaj și asignarea (a_1, \dots, a_i) este inconsistentă cu X_{i+1} , algoritmul se va întoarce la variabila cauză, $X_{ultim(i+1)}$.

```

procedure BACKJUMPING (R)
begin
   $i \leftarrow 1$ ;
   $D_i' \leftarrow D_i$ ;
   $ultim_i \leftarrow 0$ ;
  while  $1 \leq i \leq n$ 
  {  $X_i \leftarrow \text{SELECTEAZĂ-VALOARE}()$ ;
    if  $X_i = \text{null}$  then  $i \leftarrow ultim_i$ ; (întoarcere)
    else
      {  $i \leftarrow i+1$ ;
         $D_i' \leftarrow D_i$ ;
         $ultim_i \leftarrow 0$ ;
      }
    }
  if  $i = 0$  then return "problemă inconsistentă";
  else return instanțierile variabilelor  $X_i$ ;
end

```

```

procedure SELECTEAZĂ-VALOARE()
begin
  while  $D_i'$  nu e vid
  { selectează aleator o valoare  $a$  din  $D_i'$ ;
     $D_i' \leftarrow D_i' \setminus \{a\}$ ;
     $consistent \leftarrow \text{true}$ ;
     $k \leftarrow 1$ ;
    while  $k < i$  &  $consistent = \text{true}$ 
    { if  $k > ultim_i$  then  $ultim_i \leftarrow k$ ;
      if asignarea parțială  $(a_1, \dots, a_k, X_i=a)$  nu este consistentă
      then
         $consistent \leftarrow \text{false}$ ;
      else  $k \leftarrow k + 1$ ;
    }
  }

```

```

    if consistent = true then return a;
  }
  return null;
end

```

Ca și la *backtracking*, parametrul de intrare R este rețeaua de constrângeri.

Exemplul 5.4: considerăm problema de 3-colorare al cărei graf este dat în figura de mai jos. Mulțimea de culori este: roșu, verde și albastru.

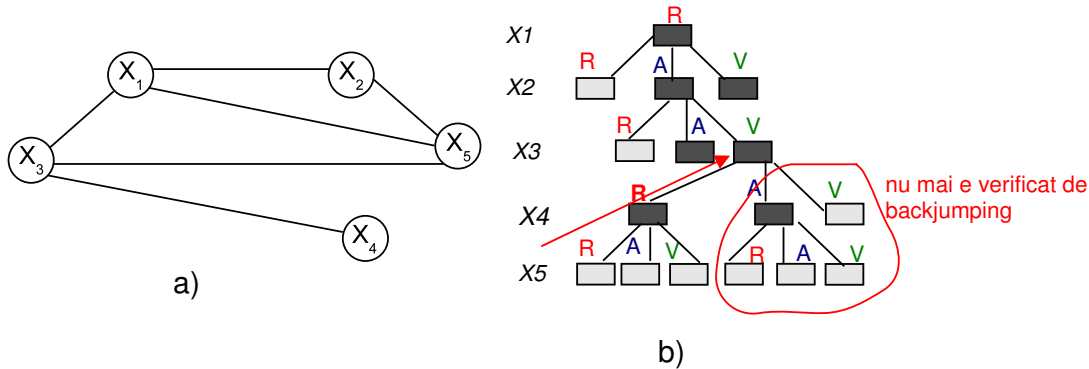


Figura 5.5: a) graful b) subarborile de căutare

Mai sus avem subarborile de căutare pentru cazul în care nodul X_1 este colorat cu roșu (R), X_2 este colorat cu albastru (A) și X_3 cu verde (V). În cazul în care nodul X_4 este colorat cu roșu, pentru nodul X_5 nu mai există nici o valoare consistentă cu asignarea parțială; suntem deci într-o situație de blocaj. Dacă am fi aplicat algoritmul *backtracking*, ne-am fi întors la variabila X_4 și am fi încercat a-i asigna o nouă valoare (albastru), ajungând însă în aceeași situație de blocaj. Între X_4 și X_5 nu există nici o constrângere. Algoritmul *backjumping* se întoarce la variabila care este cauză a blocajului, în cazul nostru variabila X_3 . Astfel reducem numărul de verificări inutile deoarece nu vom mai testa o porțiune a acestui subgraf.

O altă variantă de *backjumping* este algoritmul *graph-based backjumping* care extrage informații despre posibile mulțimi de conflicte din graful constrâns. Când apare o situație de blocaj algoritmul se întoarce la cea mai recentă variabilă care este conectată cu variabila curentă în graful constrâns.

5.2.2 Forward-checking

În algoritmul *backtracking* după selectarea variabilei următoare, valoarea acesteia va fi aleasă astfel încât să fie consistentă cu instanțierea parțială. *Forward-checking* (Haralick, 1980) verifică ca această valoare să fie compatibilă cu cel puțin o valoare din domeniul fiecărei variabile viitoare. Astfel algoritmul instanțiază variabila cu o valoare și apoi elimină valori din domeniul variabilelor viitoare care sunt în conflict cu instanțierea curentă. Dacă domeniul unei variabile viitoare devine vid, algoritmul consideră următoarea valoare posibilă pentru variabila curentă.

Algoritmul este descris mai jos. Fie X_i variabila curentă. Ca și la *backjumping*, mulțimile D' conțin domeniile reduse. Inițial acestea sunt egale cu domeniile

originale, și vor fi modificate în funcția SELECTEAZĂ-VALOARE. FORWARD-CHECKING() (referit ca FC, mai departe) alege variabilei curente o valoare consistentă cu variabilele viitoare.

```

procedure FORWARD-CHECKING(R)
begin
   $D_i' \leftarrow D_i$  ,  $1 \leq i \leq n$ ;
   $i \leftarrow 1$ ;
  while  $1 \leq i \leq n$ 
  { selectează pentru  $X_i$  o valoare din domeniu consistentă cu
    cel puțin o valoare pentru fiecare din variabilele
    următoare:
       $X_i \leftarrow \text{SELECTEAZĂ-VALOARE}()$ ;
      if  $X_i = \text{null}$  then
      {  $i \leftarrow i-1$ 
        resetează  $D_k'$  la valoarea dinaintea ultimei instanțieri
        a lui  $X_i$ ,  $k > i$ ;
      }
      else  $i \leftarrow i+1$ ;
    }
  if  $i = 0$  then return "problemă inconsistentă"
  else return instanțierile variabilelor  $X_i$ ;
end

procedure SELECTEAZĂ-VALOARE()
begin
  while  $D_i'$  nu e vid
  { selectează aleator o valoare  $a$  din  $D_i'$  ;
     $D_i' \leftarrow D_i' \setminus \{a\}$ ;
    domeniu-vid  $\leftarrow$  false;
    for  $1 \leq k \leq n$ 
    { for  $b$  din  $D_k'$ 
      if asignarea  $(a_1, \dots, a_{i-1}, X_i=a, X_k=b)$  nu este consistentă
      then  $D_k' \leftarrow D_k' \setminus \{b\}$ ;
      if  $D_k'$  este vid (bolcaj) then domeniu-vid  $\leftarrow$  true;
    }
    if domeniu-vid = true then
      resetează  $D_k'$  la valoarea dinaintea selectării lui  $a$ ,
       $i \leq k \leq n$ ;
    else return  $a$ ;
  }
  return null;
end

```

Reluarea exemplului 5.1: considerăm problema celor 4 regine, modelată ca în Exemplul 5.1, în cazul în care am asignat variabilei X_1 valoarea 4. La acest pas, algoritmul FC elimină temporar din domeniul variabilei X_2 valorile 3 și 4 deoarece nu

sunt consistente cu asignarea parțială ($X_1=4$) (a). Similar pentru domeniile variabilelor X_3 și X_4 . Cazul în care pentru variabila X_2 este aleasă valoarea 1 este evidențiat la punctul b) – noi valori pentru variabilele X_3 și X_4 sunt restricționate. Singura valoare rămasă pentru variabila X_3 este 3. Alegând această valoare, domeniul variabilei X_4 devine vid (blocaj).

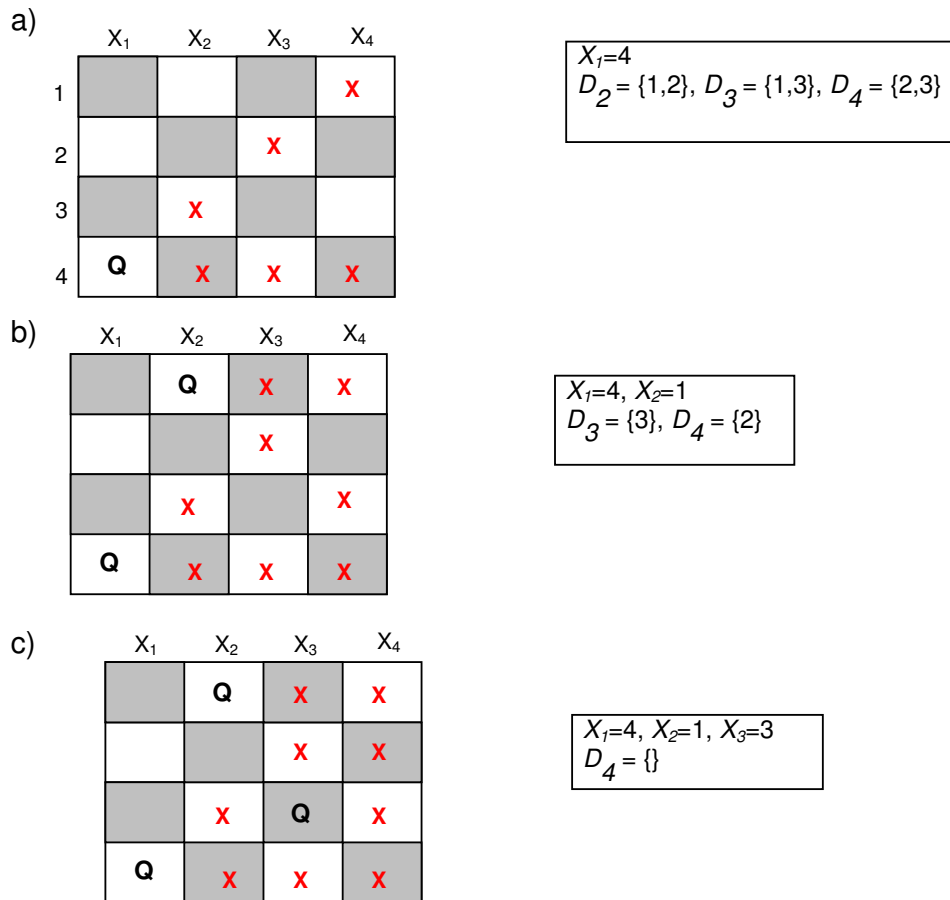


Figura 5.6: execuția pas-cu-pas a algoritmului FC pentru problema celor 4-regine; a) după asignarea variabilei X_1 b) după asignarea lui X_2 c) după asignarea lui X_3

5.3 Inferența

Algoritmii de inferență transformă problema într-una echivalentă, mai explicită prin deducerea unor noi constrângeri care vor fi adăugate mulțimii inițiale. Spre exemplu, din constrângerile $X=Y$ și $Y=Z$ putem deduce o nouă constrângere $X=Z$. Cele două probleme sunt echivalente, adică au același set de soluții, însă utilizând cea de-a doua reprezentare putem evita situații de blocare pe care le-am fi întâlnit în primul caz. Cum problema devine mai restrictivă, spațiul de căutare se va micșora, în consecință căutarea va fi mai eficientă.

Un algoritm de propagare a constrângerilor garantează că orice soluție parțială poate fi extinsă la o nouă variabilă astfel încât noua asignare să rămână consistentă (Mackworth, 1977). Algoritmii de i -consistență în general garantează că orice instanțiere consistentă a $i-1$ variabile poate fi extinsă la o instanțiere consistentă de lungime i . Cel mai cunoscut algoritm de consistență este arc-consistența (sau 2-consistența).

5.3.1 Arc-consistență

O componentă principală a algoritmului de arc-consistență o constituie procedura pentru verificarea consistenței valorilor din domeniu în raport cu o constrângere. Procedura `Revise()` are ca intrare două variabile X_i și X_j (indicii lor) și verifică dacă pentru fiecare valoare y din D_i există cel puțin o valoare compatibilă cu ea din domeniul D_j . Dacă nu există o astfel de valoare, atunci o atribuire de tipul $X_i=y$ nu este validă. Rezultă că valoarea y poate fi ștearsă din domeniu.

```

procedure REVISE( $i, j$ )
begin
  for  $y$  în  $D_i$ 
    if nu există o valoare  $z$  în  $D_j$  a.î. asignarea  $(X_i=y, X_j=z)$ 
      să fie consistentă:  $(y, z) \in R_{ij}$  then
        șterge  $y$  din  $D_i$ ;
end

```

Complexitatea procedurii este $O(k^2)$, unde k este dimensiunea domeniului.

Descriem în continuare cea mai simplă variantă de arc-consistență. Algoritmul Arc-Consistență-1 (AC-1) aplică procedura `Revise` tuturor perechilor de variabile care participă într-o constrângere până când nu se mai modifică nici un domeniu.

```

procedure AC-1( $R$ )
begin
  repeat
    for perechea  $\{X_i, X_j\}$  care participă într-o constrângere
      { Revise( $i, j$ );
        Revise( $j, i$ );
      }
    until nici un domeniu nu se mai modifică
end

```

Complexitatea algoritmului AC-1 este $O(enk^3)$, unde n este numărul de variabile, domeniul este limitat superior de k , iar e numărul de constrângeri binare.

Exemplul 5.5 Considerăm următoarea problemă de colorare a grafului cu nodurile X_1, X_2, X_3 . Domeniile de valori ale variabilelor asociate vârfurilor grafului sunt $D_1=\{R, V, G\}$, $D_2=\{R, V\}$, $D_3=\{V\}$. Graful constrâns este reprezentat mai jos.

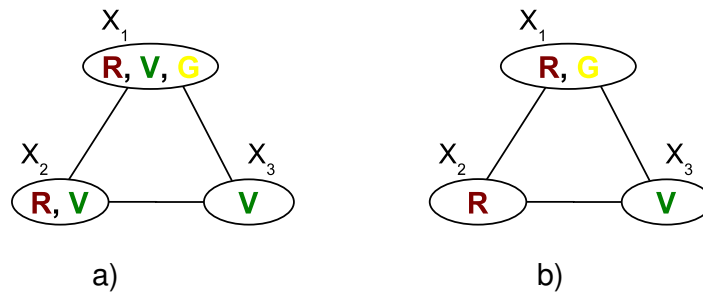


Figura 5.7: Problema colorării grafului: noduri X_1, X_2, X_3 ; domeniile sunt cele din interiorul elipselor a) graful constrâns inițial b) graful după aplicarea arc-consistenței

La examinarea arcului dintre nodurile X_1 și X_2 nu se modifică domeniul nici unei variabile. În cazul arcului (X_1, X_3) pentru culoarea verde (V) a nodului X_1 nu există o culoare corespondentă diferită de aceasta pentru a satisface constrângerea. Astfel valoarea verde va fi eliminată din domeniul variabilei X_1 . Procedeuul continuă similar prin eliminarea valorii verde pentru variabila X_2 la procesarea nodului (X_2, X_3) , rezultând graful din Figura 5.7 b).

Una din îmbunătățirile algoritmului AC-1 se sprijină pe următoarea observație: chiar dacă numai un singur arc este modificat la iterația curentă, AC-1 revizuieste toate arcele la iterația următoare; este mult mai probabil ca un număr mic de arce să fie afectate de această modificare. Algoritmul AC-3 elimină acest dezavantaj revizuiind doar acele constrângeri care pot fi afectate de modificările de la iterația anterioară. Reținem într-o coadă constrângerile care trebuiesc procesate. Inițial fiecare pereche de variabile care participă într-o constrângere este pusă în coadă de două ori (pentru fiecare ordonare a perechii de variabile). La procesarea unei perechi ordonate de variabile, aceasta va fi ștearsă din coadă și va fi adăugată din nou în coadă doar dacă domeniul celei de-a doua variabile este modificat în urma procesării constrângerilor adiacente.

```

procedure AC-3(R)
begin
   $Q \leftarrow \emptyset$ 
  for perechea  $\{X_i, X_j\}$  care participă într-o constrângere
  {  $Q.insearează(X_i, X_j)$ ;
     $Q.insearează(X_j, X_i)$ ;
  }
  while  $Q$  nu e vidă
  {  $Q.șterge(X_i, X_j)$ ;
    Revise( $i, j$ );
    if Revise( $i, j$ ) modifică  $D_i$  then
       $Q.insearează(X_k, X_i)$ , unde  $i \neq k$ , dacă arcul nu există în  $Q$ ;
  }
end

```

Exemplul 5.6: considerăm problema din Exemplul anterior 5.5. La procesarea constrângerii dintre variabilele X_1 și X_2 domeniile variabilelor rămân neschimbate.

Când verificăm arcul (X_1, X_3) domeniul variabilei X_1 se modifică. Adăugăm la coadă arcul (X_2, X_1) (arcul (X_3, X_1) există deja în coadă). Procedeu este repetat până când coada devine vidă. Observăm că dacă am fi utilizat AC-1 am fi verificat în plus încă 4 arce corespunzătoare constrângerilor dintre X_1 și X_3 , respectiv X_2 și X_3 .

Tabela 5.1: execuția algoritmului AC-3 pentru problema de colorare din Exemplul 5.5

<i>arc</i>	<i>Q</i>	<i>domenii</i>
	$\{(X_1, X_2), (X_2, X_1), (X_1, X_3), (X_3, X_1), (X_2, X_3), (X_3, X_2)\}$	$D_1=\{R, V, G\}, D_2=\{R, V\}, D_3=\{V\}$
(X_1, X_2)	$\{(X_2, X_1), (X_1, X_3), (X_3, X_1), (X_2, X_3), (X_3, X_2)\}$	
(X_2, X_1)	$\{(X_1, X_3), (X_3, X_1), (X_2, X_3), (X_3, X_2)\}$	
(X_1, X_3)	$\{(X_3, X_1), (X_2, X_3), (X_3, X_2), (X_2, X_1)\}$	$D_1=\{R, G\}, D_2=\{R, V\}, D_3=\{V\}$
(X_3, X_1)	$\{(X_2, X_3), (X_3, X_2), (X_2, X_1)\}$	
(X_2, X_3)	$\{(X_3, X_2), (X_2, X_1), (X_1, X_2)\}$	$D_1=\{R, G\}, D_2=\{V\}, D_3=\{V\}$
(X_3, X_2)	$\{(X_2, X_1), (X_1, X_2)\}$	
(X_2, X_1)	$\{(X_1, X_2)\}$	
(X_1, X_2)	$\{\}$	

Complexitatea algoritmului AC-3 este $O(ek^3)$. Algoritmul de arc-consistență poate fi îmbunătățit mai mult, ajungându-se la complexitatea minimă de $O(ek^2)$.

Algoritmii care asigură consistență locală sunt folosiți în general ca algoritmi de preprocesare, utilizați înaintea căutării. *Backtracking*-ul va fi mai eficient pe reprezentări mai explicite, deci cu un grad de consistență locală mai mare. Metodele folosite în practică combină de obicei inferența cu căutarea, în ideea îmbunătățirii rezultatelor căutării. Trebuie avută însă grijă la cantitatea de inferență utilizată: este bine de obicei să existe un echilibru între efortul depus în propagarea constrângerilor și cel depus în căutare.

5.3.2 Bucket-Elimination

Deoarece algoritmii de căutare în mod uzual extind o soluție parțială pas cu pas, inferența poate fi restricționată la o ordine a variabilelor. Această idee corespunde noțiunii de consistență direcțională. *Bucket-elimination* (Dechter, 1996) este un algoritm din această categorie, mai puțin costisitor. Algoritmul are ca intrare o mulțime de relații sau constrângeri. Dată o anumită ordonare, algoritmul împarte mulțimea de relații în submulțimi (*bucket-uri*). Fiecare submulțime este asociată unei singure variabile. O relație va aparține submulțimii corespunzătoare argumentului care apare ultimul în ordonare.

În prima fază a algoritmului fiecare *bucket* este procesat începând de la ultima variabilă către prima. Procesarea variabilei X constă în aplicarea unui operator de eliminare de variabile. În urma aplicării unei astfel de proceduri, rezultă o nouă funcție definită peste aproape toate variabilele din *bucket* cu excepția lui X . Această funcție rezumă efectul variabilei X . Ea este introdusă într-un *bucket* inferior.

În cea de-a doua fază, algoritmul construiește o soluție asignând o valoare fiecărei variabile. Ordinea de asignare este cea considerată inițial. În această etapă

sunt consultate relațiile create în cadrul primei faze. Algoritmul este prezentat mai jos.

procedure BE-MaxCSP (R)

begin

1. Inițializare: partiționează relațiile în $bucket_1, \dots, bucket_n$, unde $bucket_i$ conține relațiile care au variabila cea mai îndepărtată X_i . Fie S_1, \dots, S_j scopurile relațiilor din $bucket$ -ul procesat.

2. *Backward*:

for $p \leftarrow n$ până la 1

for h_1, h_2, \dots, h_j din $bucket_p$

if $bucket_p$ conține o instanțiere $X_p = x_p$ **then**

 { asignează variabila X_p cu valoarea x_p pentru fiecare
 relație h_i ;
 adaugă relațiile rezultate în $bucket$ -urile
 corespunzătoare;

 }

else

 { generează funcția $h^p: h^p = \min_{x_p} \sum_{i=1}^j h_i$;

$U_p = \bigcup_{i=1}^j S_i - \{X_p\}$;

 adaugă h_p la $bucket$ -ul corespunzător variabilei cu
 indexul cel mai mare din U_p ;

 }

3. *Forward*: asignează valori variabilelor în ordinea considerată
a.î. combinația de funcții din fiecare $bucket$ să fie optimizată;

return asignarea optimă și funcția calculată în $bucket$ -ul
 corespunzător primei variabile

end

Algoritmii de acest tip au avantajul cunoașterii a-priori a performanței lor, performanță ce poate fi mărginită de un parametru al grafului, *lățimea indusă* w^* . Pentru definirea acestei valori, avem nevoie de următoarele noțiuni. *Lățimea* unui nod într-un graf ordonat (G, d) este egală cu numărul de părinți ai nodului. *Graful indus* al unui graf ordonat (G, d) este graful reprezentat prin perechea (G^*, d) unde G^* este obținut prin procesarea nodurilor grafului inițial G . Procesarea unui nod rezultă în conectarea părinților acestuia. *Lățimea indusă* a grafului ordonat (G, d) , $w^*(d)$ este egală cu lățimea grafului indus (G^*, d) . *Lățimea indusă* a unui graf, w^* este lățimea indusă minimală peste toate ordonările.

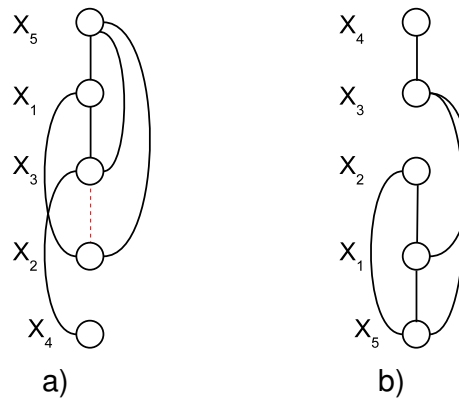


Figura 5.8: graful indus al problemei de 3-colorare din Exemplul 5.4, construit pentru a) ordonarea $d_1=(X_4, X_2, X_3, X_1, X_5)$ și b) ordonarea $d_2=(X_5, X_1, X_2, X_3, X_4)$; noile arce sunt reprezentate prin linii întrerupte (X_3X_4).

În exemplul de mai sus lățimea indusă a grafului indus (G^*, d_1) este 3, iar în cel de-al doilea caz, pentru (G^*, d_2) , este 2.

Complexitatea timp și spațiu a algoritmului *Bucket-Elimination* este exponențială în lățimea indusă a grafului de interacțiune a problemei pentru o ordonare dată. Valoarea lățimii induse poate varia pentru ordonări diferite, ceea ce implică complexități diferite.

Exemplul 5.7: considerăm aceeași problemă de 3-colorare din Exemplul 5.4. Primii doi pași ai algoritmului sunt evidențiați în Figura 5.9, pentru ordonarea $d_1=(X_4, X_2, X_3, X_1, X_5)$.

bucket(X_5): $X_5 \neq X_2, X_5 \neq X_3, X_5 \neq X_1$	bucket(X_5): $X_5 \neq X_2, X_5 \neq X_3, X_5 \neq X_1$
bucket(X_1): $X_1 \neq X_2, X_1 \neq X_3$	bucket(X_1): $X_1 \neq X_2, X_1 \neq X_3, R_{X_1X_2X_3}$
bucket(X_3): $X_3 \neq X_4$	bucket(X_3): $X_3 \neq X_4, R_{X_2X_3}$
bucket(X_2):	bucket(X_2): $R_{X_2X_4}$
bucket(X_4):	bucket(X_4): R_{X_4}

Figura 5.9. pasul de inițializare și pasul Backward ai algoritmului *Bucket-Elimination*

În partea stângă avem partiționarea inițială, iar în dreapta relațiile după procesare. Din cele trei constrângeri din *bucket*-ul 5 rezultă o nouă relație între variabilele rămase, X_1, X_2 și X_3 , relație care este copiată în *bucket*-ul corespunzător variabilei X_1 . Această relație nu mai este binară, ca relațiile inițiale, ci conține trei variabile. Similar, din procesarea fiecărui *bucket* următor rezultă câte o relație nouă.

Principalul dezavantaj al algoritmului *Bucket-Elimination* este complexitatea timp mare dar în special complexitatea spațiu datorată memorării funcțiilor intermediare. Pentru a reduce complexitatea spațiu se pot utiliza algoritmi de aproximare. Puterea acestor algoritmi constă în limitarea dimensiunii și/sau a numărului de funcții înregistrate. Consistența locală este un exemplu de inferență

aproximată. *Mini-bucket elimination* este deasemeni o schemă de aproximare care încearcă să elimine dezavantajul amintit prin partiționarea *bucket*-urilor mari în submulțimi mai mici (*mini-bucket*-uri). Această metodă permite un echilibru controlat între calitatea aproximării și complexitatea calculului.

5.4 Euristici pentru selectarea variabilelor și a valorilor

Când încercăm să extindem o soluție parțială, avem de ales următoarea variabilă și valoarea acesteia. Aceste decizii sunt importante deoarece pot reduce din verificările algoritmului de căutare, evitând situațiile de blocaj.

5.4.1 Ordonarea variabilelor

Metodele pentru ordonarea variabilelor se pot împărți în două categorii: ordonări statice sau dinamice. Aceste euristici îmbunătățesc pasul de alegere a variabilei următoare în algoritmul *backtracking*.

În primul caz, ordinea este specificată înaintea execuției algoritmului de căutare și nu este modificată apoi. Printre cele mai eficiente metode de acest tip amintim euristici *Minimum width (MW)* și *Maximum cardinality*, euristici care utilizează informații din graful constrâns. *MW* construiește o ordine în sens invers, de la ultima către prima variabilă, după următorul criteriu: alege variabila cu cele mai puține conexiuni către variabilele care nu au fost încă alese. Euristica *Maximum cardinality* selectează inițial o variabilă aleatoare și apoi la fiecare pas alege variabila conectată cu o mulțime maximală de variabile deja alese.

În cazul unei ordonări dinamice, alegerea variabilei următoare depinde de starea curentă. Ordonarea dinamică nu se poate aplica pentru toți algoritmi de căutare. Spre exemplu, pentru *backtracking* nu există informație suplimentară care să poată diferenția între variabile. În cazul algoritmilor *Forward-checking* sau *Arc-consistență*, starea curentă include domeniile restrânse ale variabilelor. Subproblema care trebuie rezolvată de acești algoritmi se modifică după fiecare asignare. Astfel o ordonare dinamică, în care următoarea variabilă de asignat este selectată pe baza informațiilor despre starea curentă ar putea conduce la o performanță mai bună a algoritmilor.

O metodă des întâlnită de alegere a variabilei următoare este selectarea celei mai constrânse variabile (euristica *Minimum Remaining Values MRV*). Altfel spus variabila cu domeniul curent cel mai mic urmează a fi asignată la pasul următor. Motivul acestei decizii se bazează pe următoarea idee: este mai eficient a elimina cât mai multe posibilități cât mai devreme. Variabila selectată minimizează factorul de ramificare al nodului curent, adică numărul de direcții posibil de explorat.

Exemplul 5.8: în cazul problemei de colorare din Figura 5.2, după asignările $X_1=R$ și $X_3=V$, domeniul variabilei X_5 va conține o singură valoare, deci ar fi mai avantajos să asignăm lui X_5 culoarea albastru decât să asignăm o valoare variabilelor X_2 sau X_4 care au câte două valori în domeniu.

Există situații în care dimensiunea domeniului variabilelor este aceeași. În multe probleme (de exemplu, problema reginelor) domeniile inițiale au aceeași dimensiune. Putem alege în acest caz variabila care constrânge cel mai mult spațiul

de căutare, adică constrânge cât mai multe variabile neasignate. Justificarea e că încercăm astfel să reducem factorul de ramificare al nodurilor viitoare.

Pentru aceeași problemă de 3-colorare, toate variabilele au domeniile inițiale egale, conțin cele trei culori posibile. Nodurile X_1 , X_3 și X_5 au gradul cel mai mare. La primul pas alegem unul din aceste noduri. Similar, după asignarea culorii roșu pentru variabila X_1 , domeniile variabilelor X_2 , X_3 și X_5 sunt egale. Variabilele X_3 și X_5 au câte două conexiuni către nodurile încă neasignate (X_2 are un singur arc). Se alege unul din aceste două noduri la pasul următor. Tabelul 5.2 exemplifică pașii algoritmului *Forward-checking* combinat cu cele două euristici de selectare a variabilei următoare.

Tabela 5.2: Execuția algoritmului FC pentru problema din Exemplul 5.2

	X_1	X_2	X_3	X_4	X_5
domeniile inițiale	R, A, V	R, A, V	R, A, V	R, A, V	R, A, V
după $X_1=R$	R	A, V	A, V	R, A, V	A, V
după $X_3=V$	R	A, V	V	R, A	A
după $X_5=A$	R	V	V	R, A	A
după $X_2=V$	R	V	V	R, A	A
după $X_4=R$	R	V	V	R	A

Această combinație a fost propusă inițial pentru problema colorării grafurilor (*euristica lui Brelaz*). Se alege drept nod următor vârful cu cele mai puține culori disponibile, iar în caz de egalitate - vârful adiacent cu cel mai mare număr de noduri necolorate. Această euristică furnizează rezultate bune pentru problema colorării. Domeniile au inițial aceeași dimensiune și constrângerile sunt similare. Euristică explorează întâi zonele cele mai dense ale grafului. Astfel sunt șanse mai mari să detectăm, încă de la începutul căutării, o clică cu un număr de noduri mai mare decât numărul de culori disponibile și să oprim căutarea, pe motiv că problema e inconsistentă.

Pentru probleme CSP generale, în care aceste condiții nu au loc, există metode care dau rezultate mai bune. De exemplu, dacă unele constrângeri sunt mai dificil de satisfăcut decât altele, atunci ar fi de preferat să asignăm valori variabilelor implicate în aceste constrângeri mai întâi, indiferent de dimensiunea domeniului.

5.4.2 Selectarea valorilor

Algoritmul de căutare trebuie să aibă definită o ordine în care vor fi asignate valorile variabilelor. O ordonare bună a valorilor poate avantaja procesul de căutare. O direcție care conduce la o soluție este verificată mai devreme decât cele care conduc la situații de blocaj. Această situație este valabilă pentru cazul în care se dorește determinarea unei soluții. În cazul în care dorim să identificăm toate soluțiile problemei, sau în caz de inconsistență, această ordine nu contează atât de mult.

În majoritatea cazurilor, se alege valoarea cea mai puțin constrânsă, care maximizează deci opțiunile viitoare de instanțiere.

Pe baza principiului de mai sus s-au mai propus și alte câteva metode pentru selectarea valorii variabilei curente. Se verifică pentru fiecare valoare în parte domeniile variabilelor viitoare (similar procedurii *Forward-checking*). Se alege valoarea cu costul cel mai mic, unde costul este dat de procentul de valori din domeniile viitoare care nu vor mai putea fi utilizate. O altă variantă propusă constă în selectarea celei mai “promițătoare” valori, adică cu cea mai mare valoare pentru produsul dimensiunii domeniilor variabilelor viitoare. Aceste variante sunt din păcate destul de costisitoare. În general, beneficiul alegerii unei valori care are o probabilitate mai mare de a conduce către o soluție nu este contrabalansat de efortul depus în *Forward-checking* pentru fiecare valoare. În cazuri particulare, există informație suplimentară care poate fi utilizată pentru selectarea unei astfel de valori.

O descriere a tehnicilor generale de rezolvare a problemelor de satisfacere a constrângerilor poate fi consultată și în tutorialul on-line al lui Bartak (vezi Bartak, R. în referințe).

5.5 Metode stochastice

Deoarece majoritatea problemelor din lumea reală sunt supra-constrânse și nu au o soluție exactă, este preferat un algoritm de căutare stohastică pentru o rezolvare mai rapidă a problemei. Abordări bazate pe metaeuristici și paradigme inspirate din natură s-au dovedit a fi eficiente în domeniul inteligenței artificiale aplicate și cel al optimizării combinatorii. Au fost încercate și pentru problemele de satisfacere a constrângerilor metode de căutare locală, ca euristicele *min-conflicts*, *stochastic local search*, sau metaeuristici precum algoritmi genetici (Craenen, 2003).

Algoritmii de căutare locală pornesc cu o asignare completă, și la fiecare pas, încearcă să o îmbunătățească prin modificarea valorii unei variabile. Asignarea inițială este generată aleator sau utilizând o metodă deterministă. Funcția de evaluare a unei astfel de soluții poate fi, în cazul problemelor CSP, numărul de constrângeri violate. Variabila a cărei valoare va fi modificată, respectiv noua valoare în cel mai simplu caz, sunt alese aleator.

De exemplu, pentru problema reginelor, starea inițială este o configurație aleatoare a reginelor pe coloane. Un pas al algoritmului ar consta în alegerea unei regine și mutarea acesteia pe o altă poziție în cadrul coloanei respective.

Euristica *min-conflicts* selectează valoarea care minimizează numărul de conflicte. Metoda s-a dovedit a fi eficientă pentru problema reginelor: găsește soluția după un număr mic de iterații. Rezultate bune s-au obținut și pentru probleme de planificare.

În cazul în care problema nu are soluție, algoritmii stochastici nu pot detecta acest lucru. Avantajul utilizării lor îl constituie faptul că pot determina un optim local destul de bun pentru problemele în care algoritmii determiniști nu sunt fezabili.

5.6 Concluzii

În acest capitol am prezentat o descriere sumară a problemelor de satisfacere a constrângerilor. Multe probleme din lumea reală pot fi modelate ca probleme CSP. Printre tehnicile de rezolvare a acestora se numără scheme de

backtracking inteligent și inferență. Euristicile de ordonare a variabilelor și a valorilor îmbunătățesc căutarea.

Cerințe pentru studenți

- să recunoască care probleme pot fi modelate ca probleme CSP
- să identifice metoda cea mai potrivită pentru o problemă CSP

Probleme

P5.1 Formulați următoarele probleme ca probleme de satisfacere a constrângerilor:

- Problema planificării unei suprafețe dreptunghiulare (*rectilinear floor-planning*): dată o mulțime de dreptunghiuri mici, să se amplaseze aceste dreptunghiuri pe o suprafața dreptunghiulară mai mare astfel încât acestea să nu se suprapună.
- Pătratul magic: să se aranjeze cifre de la 1 la 9 într-un pătrat de 3x3 astfel încât suma celor 3 numere din fiecare linie, coloană și diagonală să fie aceeași.
- Problema planificării examenelor: să se planifice examenele unei mulțimi de studenți. Pentru fiecare student se cunosc obiectele la care acesta are de susținut o probă. Posibilele perioade de timp sunt stabilite apriori. Condiția principală este ca examenele unui student să fie programate în perioade diferite.

P5.2 Modelați problema de mai jos ca o problemă CSP:

Cinci copii sunt în spital și fiecare este vizitat în perioada stabilită pentru vizite de un prieten sau o rudă. Acesta îi aduce două cadouri: ceva de mâncare sau de băut și o jucărie.

Problema este de a identifica pentru fiecare copil care este vizitatorul său și cadourile pe care le-a primit, având date următoarele informații:

Cei cinci copii: Maria, Ana, Petru, Ștefan, Ioana;

Vizitatorii: matusa, tata, prietenul, bunicul, mama;

Cadouri: banane, prajitură, jeleuri, suc de portocale, caramele;

carte de colorat, mingea, păpușă, cărți de joc, carte de povești.

Mai știm în plus:

Copilul care a primit păpușa nu a primit dulciuri.

Bananele și cărțile de joc au fost date aceleiași fetițe.

Persoana care a cumpărat suc de portocale nu a fost o rudă de sex feminin.

Prietenul lui Ștefan i-a adus mingea.

Păpușa a fost dăruită unei fetițe.

Prajitura a fost pentru ziua Mariei.

Caramelele au fost date unui băiat.

Vizitatorul Ioanei nu a fost o rudă de sex feminin.

Bunicul Anei i-a adus dulciurile ei preferate.

Vizitatorul lui Petru a fost mătușa lui; el nu a primit cartea de colorat.

Aplicați algoritmul Arc-consistența pentru această problemă.

P5.3 Fie următoarea problemă de satisfacere a constrângerilor, dată prin graful constrâns din Figura 5.10. Domeniile variabilelor sunt: $D_1=\{P, K, B\}$, $D_2=\{U, L, I\}$, $D_3=\{V, U, N\}$, $D_4=\{K, J, E, B\}$. Constrângerile exprimă ordinea lexicografică între variabilele implicate în relație. Descrieți pașii algoritmului *Backjumping* pentru această problemă.

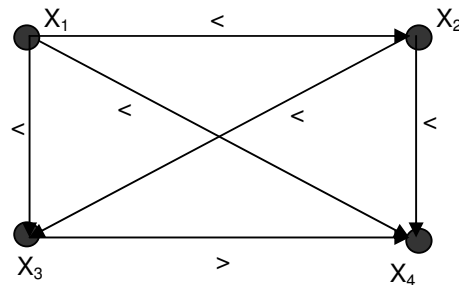


Figura 5.10: graful constrâns

P5.4 Considerați problema 8-regine. Exemplificați printr-o situație utilitatea algoritmului *Backjumping* relativ la *Backtracking*.

P5.5 Modelați problema reprezentată de puzzle-ul de mai jos ca o problemă CSP. Propuneți cel puțin două variante de modelare. Rezolvați problema utilizând metoda *Forward-checking* combinată cu euristica *MRV* de selectare a variabilei și euristica care selectează valoarea cea mai puțin constrânsă. Descrieți pas cu pas execuția algoritmului.

SATURN
+URANUS
=PLANETS

P5.6 Fie următoarele opt pătrate poziționate ca în Figura 5.11:

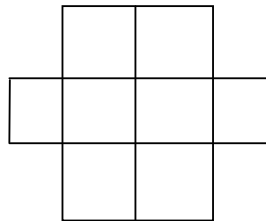


Figura 5.11: pătratele neetichetate inițial

Problema constă în etichetarea pătratelor cu numere de la 1 la 8 astfel încât etichetele oricărei perechi de pătrate adiacente (orizontale, verticale sau diagonale) să difere prin cel puțin 2.

- Scrieți constrângerile sub formă de relații și desenați graful constrâns.
- Este rețeaua arc-consistentă? Dacă nu, aduceți-o la o formă arc-consistentă.
- Este rețeaua consistentă? Dacă da, dați exemplu de o soluție.

P5.7 Utilizați metoda de căutare locală min-conflicts pentru a rezolva problema celor 4 regine. Inițial considerați reginele pe diagonala principală.

Bibliografie

Dechter, R. (2003) Constraint Processing. Morgan Kaufmann Publishers.

Tsang, E.P.K. (1993) *Foundations of Constraint Satisfaction*, Academic Press.

Dechter, R. (1990) Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cut-set Decomposition. *Artificial Intelligence*, 41: 273-312.

Gaschnig, J. (1979) Performance Measurement and Analysis of Certain Search Algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University.

Haralick, M., Elliot, G. L. (1980) Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14: 263–313.

Mackworth, A. K. (1977) Consistency in networks of relations. *Artificial Intelligence*, 8(1): 99 – 118.

Dechter, R. (1996) Bucket elimination: A unifying framework for probabilistic inference algorithms. *Uncertainty in Artificial Intelligence (UAI-96)*, pages 211–219.

Craenen, B.G.W., Eiben, A.E., van Hemert, J.I. (2003) Comparing Evolutionary Algorithms on Binary Constraint Satisfaction Problems. *IEEE Transactions on Evolutionary Computation*, 7(5) : 424-444.

Bartak, R. On-line guide to Constraint Programming
<http://kti.ms.mff.cuni.cz/~bartak/constraints/index.html>

Capitolul 6

Învățarea automată

*Înălțimea unui munte nu este măsurată calculând efortul
necesar pentru a ajunge în vârf.*

Friedrich Nietzsche

6.1 Descriere generală

Învățarea automată, unul din sub-domeniile de bază ale Inteligenței Artificiale, se preocupă cu dezvoltarea de algoritmi și metode ce permit unui sistem informatic să învețe date, reguli, chiar algoritmi. Învățarea automată presupune în primul rând identificarea și implementarea unei modalități cât mai eficiente de a reprezenta informații, în sensul facilitării căutării, re-organizării și modificării lor. Alegerea modului de a reprezenta aceste date ține atât de concepția generală asupra modului de rezolvare a problemei, cât și de caracteristicile datelor cu care se lucrează.

Învățarea nu se poate face pe baza unui set foarte mare de cunoștințe, atât din cauza costurilor mari presupuse de acumularea unor baze de informații mari cât și din cauza complexității memorării și prelucrării unui volum mare de informații. În același timp însă, învățarea trebuie să ducă la formularea de suficiente “reguli” atât cât să permită rezolvarea unor probleme dintr-un spațiu mai larg decât cel pe baza căruia s-a făcut învățarea. Adică învățarea trebuie să îmbunătățească performanța unui sistem nu doar în rezolvarea repetată a unui același set de probleme, ci și în rezolvarea unor probleme noi. Acest lucru presupune o generalizare a unei metode de rezolvare pentru a acoperi un număr cât mai mare de instanțe posibile, dar și păstrarea unei specializări suficiente pentru a fi identificate corect instanțele acceptate. Aceasta se poate face fie inductiv, generalizând o problemă plecând de la un set de exemple, fie deductiv, plecând de la o bază de cunoștințe suficiente asupra universului problemei și extrăgând date și reguli esențiale. Pentru a putea face acest lucru, un algoritm de învățare trebuie să fie capabil să selecteze acele elemente semnificative pentru rezolvarea unei instanțe viitoare a problemei. Aceasta alegere se face pe baza unor criterii de selecție numite diagonale inductive.

O altă componentă esențială al unui algoritm de învățare este metoda de verificare, o metodă capabilă să confirme dacă generalizările făcute sau regulile deduse se apropie mai mult de soluția ideală decât starea anterioară a sistemului. O prezentare mai detaliată a componentelor esențiale ale unui sistem capabil de învățare este făcută în secțiunea a doua a acestui capitol.

Studiul învățării automate a dus la descrierea a numeroase metode, variind după scop, date de antrenament, strategia de învățare și modalitatea de reprezentare a datelor. Secțiunea a treia face o prezentare a principalelor direcții în învățarea supervizată, ce folosește un set de instanțe rezolvate ale problemei pentru a antrena sistemul în vederea rezolvării unor instanțe noi. Secțiunea a patra prezintă

învățarea prin încurajare, ce implementează o metodă de a “răsplăti” sistemul în funcție de progresul făcut în găsirea unei soluții optime. Algoritmii genetici, prezentați în secțiunea a cincea, folosesc metode de reprezentare și căutare similare mecanismelor biologice. Secțiunea a șasea prezintă învățarea bazată pe cunoștințe, ce deduce soluții pe baza unor cunoștințe anterioare asupra domeniului problemei. Secțiunea a șaptea face o prezentare a dezavantajelor conceptului de învățare supervizată, iar apoi în secțiunea a opta se descrie învățarea nesupervizată și problema clasificării.

6.2 Caracteristici ale unui sistem capabil de învățare

Specificarea unei metode de învățare automată presupune definirea următoarelor date:

- scopul metodei și baza de cunoștințe necesară;
- formalismul de reprezentare a datelor utilizate și a celor învățate;
- un set de operații asupra datelor disponibile și învățate;
- un spațiu general al problemei în care se va specifica soluția;
- opțional, reguli euristice pentru căutarea în spațiul problemei.

6.2.1 Scopul metodei și baza de cunoștințe necesară

O metodă de învățare este definită în primul rând de datele de plecare și scopul algoritmului, adică ce se dorește să se obțină pe baza acestor date. Plecând de la aceste caracteristici, avem următoarele variante:

- datele de plecare sunt un set de exemple pozitive și negative de instanțe ale problemei, algoritmul trebuind să găsească o generalizare care să includă instanțele pozitive și să le excludă pe cele negative;
- datele de plecare constau într-un set redus de exemple pozitive, iar algoritmul, plecând cu o bază de cunoștințe generale asupra domeniului problemei, trebuie să facă o generalizare;
- datele de plecare sunt un set de instanțe neclasificate, algoritmul trebuind să identifice clase de instanțe cu proprietăți similare;
- date de plecare sub forma uneia sau a mai multor instanțe ce descriu situații ca fiind analogii ale problemei de rezolvat, algoritmul trebuind să identifice elementele ce formează analogia și să deducă soluția problemei.

Datele din baza de cunoștințe mai sunt caracterizate și de acuratețea și calitatea lor. Cel ce învață trebuie să țină cont de încrederea în sursa de informații și de detalierea și gradul de organizare a informațiilor.

Anumite date ce ar putea contribui semnificativ la acuratețea unei metode de învățare, dacă ar fi prezente în baza de cunoștințe, pot fi prea greu de obținut. Raportul preț / câștig de informație trebuie avut în vedere atât în crearea bazei de cunoștințe cât și în descrierea metodelor de învățare.

6.2.2 Formalismul de reprezentare a datelor utilizate și a celor învățate

Conceptele și instanțele problemei utilizate în algoritmul de învățare trebuie formalizate ca expresii sau obiecte cu proprietăți cuantificabile și clar definite. Reprezentarea acestor proprietăți se face în general într-o formă structurată, cum ar fi un tabel sau un graf.

Un exemplu de descriere a unor instanțe folosind logica propozițională este:

```
mărime(obj1, mică) ∧ culoare(obj1, roșie) ∧ forma(obj1, rotundă)
mărime(obj2, mare) ∧ culoare(obj2, roșie) ∧ forma(obj2, rotundă)
```

iar descrierea unui concept general pentru instanțe de tipul celor de mai sus ar fi:

```
mărime(x, y) ∧ culoare(x, z) ∧ forma(x, rotundă)
```

6.2.3 Setul de operații

Algoritmul trebuie să aibă la dispoziție unelte care să îi permită manipularea datelor în reprezentarea specifică. Operațiile necesare sunt în general cele de generalizare și specializare a unui concept, de modificare și adăugare de instanțe și expresii, de căutare în spațiul problemei.

Operatorii de generalizare și specializare sunt esențiali în orice algoritm de învățare. Tipurile principale de generalizări folosite sunt:

- înlocuirea constantelor cu variabile; Exemplu:

```
culoare(obj1, roșie)                se generalizează
culoare(x, roșie) ∨ culoare(x, y)
```

- renunțarea la condiții dintr-o conjuncție; Exemplu:

```
mărime(obj1, mică) ∧ culoare(obj1, roșie) ∧
forma(obj1, rotundă)                se generalizează
culoare(obj1, roșie) ∧ forma(obj1, rotundă)
```

- adăugarea unei disjuncții la o expresie; Exemplu:

```
mărime(obj1, mică) ∧ culoare(obj1, roșie) ∧
forma(obj1, rotundă)                se generalizează
mărime(obj1, mică) ∧ culoare(obj1, roșie) ∧
forma(obj1, rotundă) ∨ mărime(obj1, mare)
```

- înlocuirea unei proprietăți cu o descriere mai generală; Exemplu:

```
culoare(obj1, roșie)                se generalizează
culoare(obj1, culoare_de_bază)
```

Mărimea setului de operații disponibile și complexitatea lor sunt lucruri care au în general un impact puternic asupra rezultatelor obținute prin învățare, asupra vitezei învățării. De asemenea, operațiile disponibile restrâng soluțiile posibile la cele care pot fi obținute și exprimate prin aceste operații, deci definirea lor trebuie să fie considerată una din cele mai importante etape, atât în descrierea algoritmilor de învățare cât și în formularea problemelor pentru sistemele de învățare automată.

6.2.4 Spațiul general al problemei

Limbajul de reprezentare și setul de operații descriu un spațiu de definiție a conceptului problemei. Învățarea constă în “navigarea” prin acest spațiu în scopul ajungerii la un concept-țintă sau la noi cunoștințe. Dificultatea rezolvării problemei este direct proporțională cu complexitatea acestui spațiu general al problemei.

Evident, dimensiunea sa este strâns legată de formalismul de reprezentare ales și de operațiile descrise pentru aceste reprezentări, lucruri ce sunt făcute odată cu descrierea metodei de învățare și formularea problemei. Aceasta înseamnă că putem descrie o metodă de învățare rapidă, pe un spațiu mic de soluții, dar formalismul de reprezentare și operațiile nu vor avea o putere expresivă mare. Trebuie avut în vedere echilibrul între puterea descriptivă și timpul de găsim a unei soluții. De asemenea, de aici rezultă și faptul că o metodă de învățare concepută pentru o problemă specifică se va comporta de obicei mai bine decât o metodă generală de învățare.

6.2.5 Reguli euristice pentru căutare

Ordinea și modalitatea prin care se face căutarea în spațiul de soluții al problemei sunt, în general, stabilite printr-un set de reguli euristice. Aceste reguli direcționează algoritmul și îl ajută să ia decizii în privința momentului și felului în care se poate generaliza sau specializa un concept, sau se poate introduce un concept nou. Aceste euristici sunt în general descrise odată cu algoritmul propriu-zis, fiind proprii fiecărei metode de învățare.

Aceste reguli pot lipsi, ele nefiind în general esențiale funcționării algoritmului, ci având doar rolul de a optimiza funcționarea sa.

6.3. Învățarea supervizată

Învățarea supervizată este un tip de învățare inductivă ce pleacă de la un set de exemple de instanțe ale problemei și formează o funcție de evaluare (șablon) care să permită clasificarea (rezolvarea) unor instanțe noi. Învățarea este supervizată în sensul că setul de exemple este dat împreună cu clasificarea lor corectă. Aceste instanțe rezolvate se numesc instanțe de antrenament. Formal, setul de instanțe de antrenament este o mulțime de perechi atribut-valoare $(x, f(x))$, unde x este instanța iar $f(x)$ clasa căreia îi aparține instanța respectivă. De exemplu, un set de instanțe de antrenament ar putea fi:

```
I1: (culoare(obj1, roșie) ∧ forma(obj1, rotundă),  
f(obj1) = "sferă")  
I2: (mărime(obj2, mare) ∧ forma(obj2, cubică), f(obj2) = "cub")
```

Scopul învățării este construirea unei funcții-șablon care să clasifice corect instanțele-exemplu, iar pentru un x pentru care nu se cunoaște $f(x)$ să propună o aproximare cât mai corectă a valorii $f(x)$.

6.3.1 Concepte învățabile PAC

Teoria Învățării Computaționale propune o baza teoretică pentru sistemele capabile de învățare. Problema principală care se pune este: cum demonstrăm faptul că un șablon se apropie de perfecțiune în etichetarea oricăror instanțe noi?

Fie f funcția pe care dorim să o aproximăm, S un set de instanțe de antrenament $(x, f(x))$, iar h funcția indusă prin învățare. Este sau nu $h(x)$ aproape de $f(x)$, pentru orice x din spațiul instanțelor posibile?

Argumentul standard este acela că h nu poate fi prea departe de f , căci h clasifică corect instanțele de antrenament, deci probabil și pe celelalte. Deci h este *probabil aproximativ corect* (PAC). Un concept este învățabil PAC dacă există un algoritm eficient care are o probabilitate mare de a găsi o aproximare a conceptului *probabil aproximativ corectă*. Termenul “*învățabil PAC*” a fost introdus de Valiant (1984).

Presupunerea aflată la baza acestei justificări este aceea că instanțele de antrenament și instanțele de testare sunt uniform distribuite în spațiul instanțelor posibile ale problemei. Acest lucru este fundamental în justificarea oricărui rezultat al unei învățării supervizate.

Pentru calculul devierii unei funcții-șablon h față de f , putem defini o funcție de eroare $E(h)$ ca fiind:

$$E(h) = \Pr(h(x) \neq f(x) \mid x \text{ din } D)$$

unde D este o mulțime de instanțe uniform distribuite în spațiul instanțelor posibile. Șablonul h este numit *aproximativ corect* dacă $E(h) \leq e$, unde e este o probabilitate maximă de eroare.

Capacitatea unui concept de a fi învățabil PAC este independentă de algoritmul folosit. Atunci de ce depinde? Cum recunoaștem un concept învățabil? Răspunsul stă în capacitatea aceluși concept de a fi reprezentat printr-o formalizare neambiguă. Concepte ca *minge*, *pătrat*, *profitabil* pot fi definite prin proprietăți ce le identifică în mod unic. Aceste proprietăți pot fi definite pentru că există limbaje de reprezentare pentru ele. Aceste concepte pot fi învățate cu atât mai ușor cu cât numărul de proprietăți ce le definesc în mod unic este mai mic.

6.3.2 Arbori de decizie

Un arbore de decizie este una din cele mai utilizate structuri de reprezentare utilizate în învățarea automată. Pentru o instanță specificată de un set de proprietăți, arborele verifică anumite proprietăți pentru a “naviga” prin arbore și ajunge la o frunză care va fi “eticheta” acelei instanțe. Fiecare nod intern al arborelui reprezintă un test făcut asupra uneia sau mai multor proprietăți ale instanței, iar ramurile descendente din acel nod sunt identificate de posibilele rezultate ale celui test.

Un arbore de decizie construiește pentru o instanță o conjuncție logică ce se verifică pentru proprietățile instanței și formează un fel de demonstrație a clasificării făcute pe baza acelor proprietăți.

Ca exemplu, fie arborele binar de decizie din Figura 6.1.

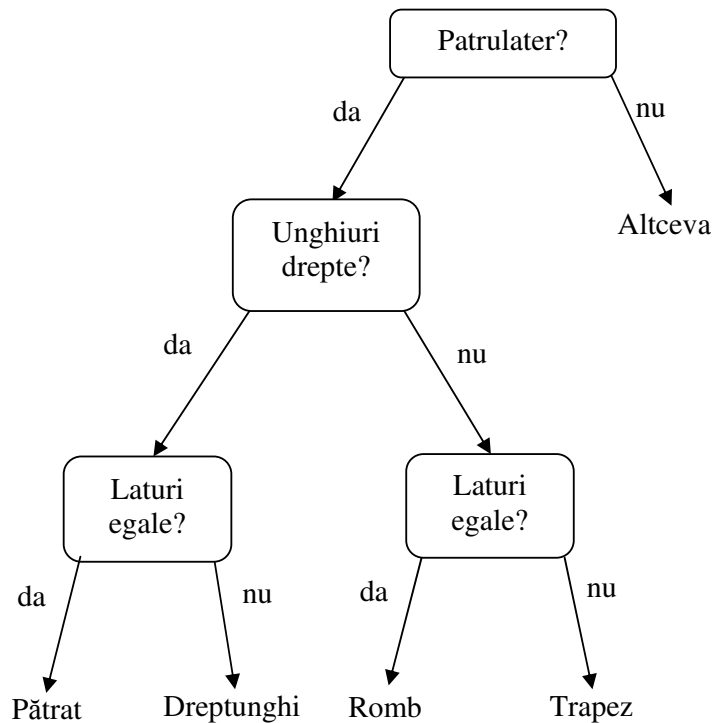
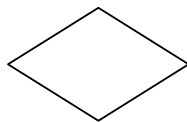


Figura 6.1: Exemplu de arbore binar de decizie

Acest arbore identifică tipurile de patrulaterare ținând cont de anumite proprietăți de bază. O instanță de intrare pentru acest arbore este o figură geometrică definită de proprietățile sale. Arborele face anumite teste și ajunge la un nod terminal care identifică tipul figurii geometrice, dacă este un patrulater, sau răspunde “Altceva” dacă nu este un patrulater. Drumul luat pentru a ajunge la răspuns poate fi reconstituit și formează o “demonstrație” a răspunsului.

De exemplu, pentru instanța:



Drumul parcurs va forma demonstrația:

Figura este romb, deoarece este patrulater și nu are unghiuri drepte și are laturile egale.

Avantajul principal al arborilor de decizie este acela că minimizează numărul de teste la cele suficiente pentru a formula un răspuns. Avantajul este evident chiar în exemplul dat: arborele identifică orice figură geometrică, indiferent de proprietățile sale. De exemplu, pentru instanța:



arborele nu va verifica proprietățile nerelevante pentru problemă, ci va face testul “*Patrulater?*”, și va răspunde imediat “*nu*”.

Învățarea unui arbore de decizie plecând de la un set S de exemple etichetate și mulțimea P de proprietăți se face după algoritmul:

```

ConstruiesteArbore(S)
begin
  if (toate elementele din S au aceeași etichetă) then
    întoarce un nod terminal cu eticheta respectivă
  else
    { do
      {caută proprietatea  $p$  din  $P$  cu cel mai mare câștig de
        informație (vezi 6.3.3);
      crează un arbore de decizie cu rădăcina conținând testul
        acelei proprietăți;
      leagă subarborele rezultat la ramura curentă;
    }while (există rezultat posibil al testului acelei
      proprietăți neinclus în arbore)
    }
  return arborele;
end

```

Elementul esențial este, evident, identificarea proprietății ce aduce cel mai mare câștig de informație prin testarea ei în arborele de decizie.

6.3.3 Calculul câștigului de informație

Teoria informației (Shannon, 1948) este baza matematică ce permite calculul conținutului de informație al unui mesaj. Un mesaj poate fi privit ca un răspuns la o întrebare I . Astfel, dacă răspunsurile posibile sunt v_i iar probabilitățile lor de apariție sunt $P(v_i)$, conținutul de informație al răspunsului la întrebarea I este dat de formula:

$$I(P(v_1), \dots, P(v_n)) = \sum_{i=1}^n -P(v_i) \log_2 P(v_i)$$

Pentru arborii de decizie vrem să aflăm însă câștigul de informație al unui răspuns la o întrebare, comparativ cu celelalte întrebări posibile. Pentru a calcula câștigul de informație al unui test T se utilizează formula:

$$\text{Câștig}(T) = I(1-p(v_1)/|S|, \dots, 1-p(v_n)/|S|) - \text{CostUlterior}(T)$$

unde $|S|$ este cardinalul mulțimii de instanțe de antrenament, $p(v_i)$ numărul de instanțe pentru care răspunsul la întrebarea I este v_i , iar $\text{CostUlterior}(T)$ reprezintă cantitatea de informație necesară pentru a finaliza arborele de decizie în cazul selectării testului T la pasul curent.

$$\text{CostUlterior}(T) = \sum_{i=1}^n (|S_i|/|S|) \times I(1-p_i(v_{i1})/|S_i|, \dots, 1-p_i(v_{in})/|S_i|)$$

unde S_i este mulțimea de instanțe care au valoarea v_i pentru proprietatea testată.

Exemplu de construcție a unui arbore ID3

Să presupunem că dorim să antrenăm un arbore de decizie capabil să hotărască dacă ziua va fi bună pentru ski. Pentru antrenare, construim instanțe pentru ultimele două săptămâni, colectând pentru fiecare zi informații privind prognoza meteo pentru ziua respectivă, temperatura înregistrată, umiditatea atmosferică și vântul. Pentru acești parametri, selectăm din următoarele variante:

```
prognoza:(optimistă, neutră, proastă)
temperatura:(cald, optim, frig)
umiditatea:(mare, normală)
vântul:(puternic, slab)
```

Pentru două săptămâni, respectiv cele 14 zile folosite ca instanțe de antrenare, avem datele din Tabela 6.1.

Tabela 6.1: Instanțe de antrenare

Zi	prognoza	temperatura	umiditatea	vântul	Optim ski?
1	optimistă	cald	mare	slab	nu
2	optimistă	cald	mare	puternic	nu
3	neutră	cald	mare	slab	da
4	proastă	optim	mare	slab	da
5	proastă	frig	normală	slab	da
6	proastă	frig	normală	puternic	nu
7	neutră	frig	normală	puternic	da
8	optimistă	optim	mare	slab	nu
9	optimistă	frig	normală	slab	da
10	proastă	optim	normal	slab	da
11	optimistă	optim	normală	puternic	da
12	neutră	optim	mare	puternic	da
13	neutră	cald	normală	slab	da
14	proastă	optim	mare	puternic	nu

Pentru valorile de mai sus, se calculează câștigul de informație pentru fiecare parametru (după formulele de mai sus):

```
Câștig(prognoza) = 0,246
Câștig(temperatura) = 0,029
```

$\text{Câștig}(\text{umiditatea}) = 0,151$

$\text{Câștig}(\text{vânt}) = 0,048$

Se observă că evaluarea parametrului *prognoza* obține cel mai mare câștig de informație, deci el este folosit ca test în nodul rădăcină al arborelui ID3.

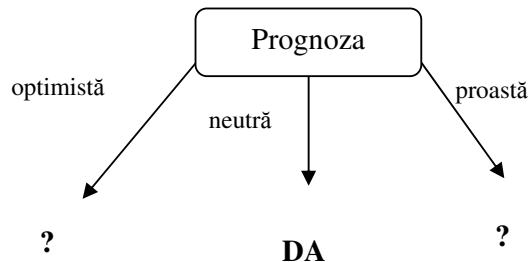


Figura 6.2: Arbore ID3 parțial

Valoarea *neutră* pentru parametrul *Prognoza* corespunde doar răspunsului pozitiv pentru testul „Este ziua optimă pentru ski?”. Cum au rămas încă instanțe neacoperite, se completează arborele adăugând pe rând testul cu cel mai mare câștig de informație, re-calculează după fiecare modificare a arborelui. Se obține arborele din Figura 6.3.

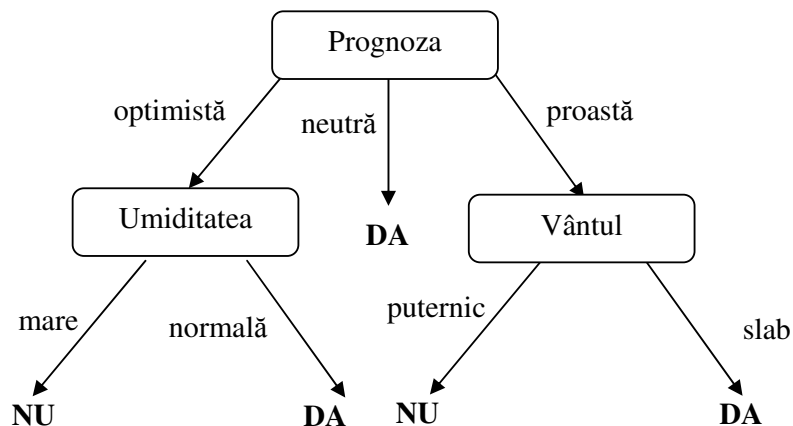


Figura 6.3: Arborele ID3 pentru exemplu

Se observă că testul parametrului temperatura nu este necesar pentru clasificarea corectă a tuturor instanțelor de antrenament.

6.3.4 Probleme în utilizarea arborilor de decizie

Algoritmul de învățare pentru arbori de decizie poate găsi proprietăți nerelevante, dar regulate, ca fiind esențiale în etichetarea unei instanțe. Dacă numărul de proprietăți este mare, iar exemplele sunt insuficient de „împrăștiate” în spațiul problemei, algoritmul poate deduce că, de exemplu, culoarea neagră este

semnificativă pentru a decide dacă o figură geometrică este patrulater. Aceasta problemă se numește *suprapunerea seturilor*.

O alta problemă semnificativă este aceea a luării unei decizii privind adăugarea de noduri în arbore. Este posibil ca adăugarea unui nou test în arbore să nu îmbunătățească precizia sa de clasificare, sau chiar să o scadă.

O soluție la ambele probleme este folosirea unei metode de a direcționa căutarea în mulțimea de proprietăți pentru a asigura identificarea celei mai directe căi spre soluția corectă. În acest scop, unei proprietăți îi asociem și o valoare de relevanță statistică pe baza căreia determinăm șansa ca relevanța proprietății respective să fie independentă de setul de instanțe pe care antrenăm arborele. Astfel eliminăm testarea oricărei proprietăți cu probabilitate mare de a fi irelevantă pentru scopul învățării.

Altă soluție este *validarea încrucișată*, o metodă ce testează submulțimi de instanțe pentru a vedea cât de bine ar fi clasificate folosind testul curent. Se realizează o medie a rezultatelor pentru testele verificate și media se folosește la selectarea testelor viitoare.

O altă problemă este necesitatea existenței unui set complet de proprietăți pentru instanțele problemei. Pentru clasificarea unei instanțe, arborele de decizie poate testa oricare din proprietățile sale, deci nu putem folosi instanțe cu proprietăți ale căror valori sunt necunoscute. Există mai multe variante de a permite algoritmului să lucreze cu seturi incomplete de proprietăți ale instanțelor: putem da unei proprietăți necunoscute o valoare pe baza valorilor luate în instanțe similare, sau putem chiar construi un arbore de decizie pentru a găsi o valoare pentru acea proprietate. Cea mai simplă metodă este însă de a atribui acelei proprietăți valoarea cea mai des întâlnită în setul de instanțe.

Algoritmul de învățare pentru arbori de decizie a fost definit în forma inițială în (Quinlan,1983), unde sunt descriși un tip de arbori de decizie numiți ID3. În acea formă, proprietățile luau doar valori booleene, rezultând deci arbori binari de căutare. O extindere evidentă ar fi folosirea unor proprietăți cu valori multiple, lucru făcut într-o formă ulterioară a algoritmului (ID5 – Utgoff,1988). ID5 mai adaugă și construirea incrementală a arborelui, exemplele de intrare fiind tratate pe rând, în urma fiecărui exemplu rezultând un arbore de decizie ce va fi îmbunătățit de exemplele ulterioare. De asemenea, instanțele propriu-zise sunt păstrate în frunzele arborelui și sunt folosite pentru a determina necesitatea adăugării unui nou nod de testare la acel nivel.

6.3.5 Inducerea șabloanelor logice

Învățarea inductivă înseamnă practic căutarea unui șablon descriptiv cu care să fie comparate instanțe noi ale problemei. Inițial, șablonul propus este vag, el fiind detaliat atunci când întâlnim o instanță fals pozitivă (o instanță incorectă clasificată corectă) sau fals negativă (o instanță corectă clasificată incorectă). Căutarea instanțelor clasificate incorect și corectarea șablonului se poate face în două moduri: metoda șablonului prezent optim și metoda angajamentului minim.

Metoda șablonului prezent optim (Mill, 1843) modifică șablonul în cazul găsirii unei instanțe incorect clasificate după metoda:

- dacă instanța este fals pozitivă, specializează șablonul pentru a nu mai acoperi acea instanță prin renunțarea la disjuncții sau adăugarea de noi termeni;

- dacă instanța este fals negativă, generalizează șablonul prin adăugarea de disjuncții sau renunțarea la unii termeni;
- dacă nu poate fi găsit un șablon valid, revin la optimul precedent.

Punctul slab al acestei metode este numărul mare de teste necesare la fiecare modificare a șablonului, căci fiecare instanță trebuie verificată din nou.

Metoda angajamentului minim (Mitchell,1978) propune păstrarea tuturor șabloanelor posibile ce sunt consistente cu instanțele testate până în momentul curent. Cu fiecare nouă instanță se elimină din *spațiul versiunilor* șabloanele inconsistente cu acea instanță. Algoritmul ce realizează acest lucru se numește *algoritmul de învățare prin eliminarea candidaților* (Mitchell,1982) și are forma:

Eliminarea candidaților(G,S)

begin

Inițializează G cu cel mai general concept din spațiu;

Inițializează S cu prima instanță pozitivă din intrare;

for (fiecare instanță pozitivă p)

begin

Șterge toate instanțele din G care nu se potrivesc cu p;

for (fiecare s din S)

if (s nu se potrivește cu p) **then** înlocuiește s cu cea mai specifică generalizare care se potrivește cu p;

Șterge din S toate instanțele mai generale decât alta din S;

Șterge din S toate instanțele mai puțin specifice decât alta din G;

end;

for (fiecare instanță negativă n)

begin

Șterge toate instanțele din S care se potrivesc cu n;

for (fiecare g din G care se potrivește cu n)

înlocuiește g cu cea mai generală specializare care nu se potrivește cu n;

Șterge din G toate instanțele mai specifice decât alta din G;

Șterge din G toate instanțele mai specifice decât alta din S;

end;

if (G=S și ambele conțin o singură instanță) **then** acea instanță este conceptul căutat;

else if (G și S sunt vide) **then** nu există un concept care să acopere toate instanțele pozitive și nici una din cele negative;

end.

Mulțimile G și S conțin informația atât din instanțele pozitive cât și din cele negative și nu mai este necesar ca ele să fie reținute separat. Algoritmul folosește G pentru a testa specificitatea conceptelor din S și S pentru a testa generalitatea conceptelor din G, cele două mulțimi definind astfel doua limite între care se va găsi șablonul optim, în caz că el există.

Problema principală în identificarea oricărui șablon este sensibilitatea algoritmilor la instanțe neconsistente. Algoritmii nu au nici o posibilitate de a decide dacă o instanță de antrenament este etichetată corect, deci în cazul unei inconsistente va eșua. Soluțiile posibile sunt fie de a menține mai multe șabloane

posibile, fiecare consistent cu majoritatea instanțelor, fie de a lua în considerare numai instanțele statistic probabile.

Complexitatea computațională

Pentru p instanțe pozitive și n instanțe negative de antrenament, complexitatea de timp a învățării prin metoda șablonului prezent optim este $O(pn)$, iar cea de spațiu $O(p+n)$.

Pentru metoda angajamentului minim, complexitatea de timp este $O(sg(p+n)+s^2p+g^2n)$ iar cea de spațiu este $O(s+g)$, unde s și g sunt dimensiunile maxime pentru mulțimile S respectiv G .

6.3.6 Învățarea Bayesiană

Învățarea în modelul Bayesian presupune menținerea unor variante posibile ale șablonului, fiecare variantă fiind evaluată prin calcularea probabilității ca ea să clasifice corect o instanță viitoare. Această probabilitate se calculează folosind rezultatele anterioare ale clasificărilor făcute de șablonul respectiv. La un moment dat, trebuie însă ca cel puțin o parte din șabloanele ipotetice să fie eliminate ca fiind posibile soluții. Acest lucru se face prin stabilirea unui șablon ca fiind cel mai probabil și eliminarea tuturor celorlalte, sau cel puțin a celor cu o probabilitate considerabil mai mică. Fie H_{MAX} clasa de șabloane cu probabilitatea maximă. Notăm acea probabilitate cu $P(H_{MAX}, D)$, unde D este mulțimea de instanțe. Pentru a identifica probabilitatea maximă, trebuie să calculăm probabilitățile tuturor ipotezelor curente, folosind regula lui Bayes:

$$P(H_i | D) = [P(D | H_i) \times P(H_i)] / P(D)$$

unde H_i este o clasa de șabloane care clasifica similar instanțele din D . Cum $P(D)$ nu poate fi influențat de algoritm, singura metodă de îmbunătățire a probabilității de corectitudine este fie mărirea șansei ca setul de instanțe să ducă la găsirea unui șablon aproximativ corect ($P(D | H_i)$), fie mărirea șansei ca un șablon să fie aproximativ corect ($P(H_i)$).

Acest model de a forma o rețea de șabloane posibile se numește învățarea folosind o rețea de fapte.

În funcție de structura rețelei și de variabilele interne, putem diferenția patru tipuri de rețele de fapte:

- rețele în care structura este cunoscută și variabilele vizibile, în care partea învățabilă este cea a probabilităților condiționate ale șabloanelor;
- rețele cu structura necunoscută, dar variabile vizibile, în care învățarea poate reconstrui topologia rețelei;
- rețele cu structura cunoscută și variabile ascunse, care sunt analoage rețelelor neuronale;
- rețele cu structura necunoscută și variabile ascunse, pentru care nu există nici o metoda de învățare.

6.3.7 Învățarea prin încurajare

Spre deosebire de metodele de învățare supervizată prezentate mai sus, învățarea prin încurajare se face fără ca algoritmul de învățare să compare direct șablonul obținut cu rezultatele corecte pentru exemplele de antrenament. În schimb

este implementată o modalitate de a “răsplăti” sau “pedepsi” sistemul în funcție de cât de mult se apropie de rezultatul corect. Acest *feedback* este singura metodă a sistemului de învățare de a se regla pentru îmbunătățirea rezultatelor sale. Acest lucru face învățarea prin încurajare mai dificilă, căci sistemul nu mai primește informații directe despre cum și cât să se corecteze, ci doar știe dacă se apropie sau se depărtează de rezultatul optim. De asemenea, *feedback*-ul poate veni doar uneori, nu neapărat la fiecare schimbare în șablonul ipotetic, deci sistemul trebuie să aibă o modalitate de a direcționa și impulsiona singur schimbarea pentru îmbunătățirea șablonului.

Există două tipuri de informație ce pot face parte din *feedback*-ul primit de sistem:

- informație utilitară, prin care sistemul învață utilitatea unei anumite stări în care se află și care permite sistemului să caute acele stări care maximizează șansa de a găsi o soluție optimă;
- valoarea unei acțiuni, adică sistemul află potențialul unei acțiuni de a fi sau nu utilă într-o anumită stare, practic potențialul unei acțiuni de a apropia sistemul de găsirea unei soluții optime.

Învățarea prin încurajare este utilă în situațiile în care nu dispunem de un set de antrenament, nu putem identifica cu precizie instanțe valide sau eronate, fie din cauza complexității reprezentăționale, fie din cauza lipsei de informații sigure. Dispunem însă de capacitatea de a aprecia corectitudinea unei soluții obținute de sistem, fie chiar și doar comparativ cu celelalte soluții.

6.3.8 Învățarea pasivă într-un domeniu cunoscut de stări

Dacă furnizăm unui sistem de învățare un domeniu de stări prin care poate trece împreună cu un model ce calculează șansa de a trece dintr-o stare în alta, sistemul poate învăța după un set de secvențe de antrenament formate dintr-un șir de stări consecutive în urma cărora se obține o “răsplată” cunoscută, fie pozitivă sau negativă. Scopul sistemului devine astfel învățarea utilității fiecărei stări posibile.

Presupunerea la baza învățării utilității unei stări dintr-o secvență este aceea că “răsplata” obținută după o secvență este rezultatul sumei utilităților stărilor ce compun secvența, adică suma pozitivă a utilităților stărilor favorabile adunată cu suma negativă a utilităților stărilor nefavorabile dă un număr pozitiv atunci când secvența este “răsplătită”, și un număr negativ când secvența este “pedepsită”.

Există trei modalități ca un sistem să învețe utilitatea stărilor sale: învățarea naivă, învățarea adaptiv dinamică și învățarea după diferența temporală.

Învățarea naivă este reprezentată de metoda minimului de stări negative (*Least Mean Squares* – LMS) (Widrow, Hoff, 1960). Metoda calculează utilitatea unei stări ca fiind media utilității secvențelor în care ea apare. Această metodă minimizează într-adevăr numărul de stări defavorabile prin care ar trebui să treacă sistemul pentru a ajunge la un șablon satisfăcător, însă nu minimizează numărului de stări prin care ar trebui să treacă și nici nu garantează găsirea celui mai optim șablon.

Utilitatea unei stări găsită prin învățarea naivă este departe de utilitatea reală, pentru că în calculul ei nu se ține cont de faptul că utilitatea unei stări este dependentă și de suma ponderată de probabilități ale utilităților stărilor ulterioare. Utilitatea reală a stării i este calculată după formula:

$$U(i) = R(i) + \sum_j M_{ij}U(j)$$

unde $R(i)$ este răsplata stării i și M_{ij} este probabilitatea trecerii din starea i în starea j .

Învățarea adaptiv dinamică este reprezentată de orice metodă de învățare prin încurajare care calculează utilitățile stărilor sistemului printr-un algoritm dinamic, după formula de mai sus. Deși este o metodă ce dă rezultate foarte bune, ea prezintă dezavantajul că este foarte costisitoare în timp, căci căutarea în spațiul stărilor este de ordin exponențial.

Învățarea după diferența temporală folosește diferența între utilitățile stărilor succesive dintr-o secvență pentru a le ajusta la fiecare trecere prin ele. Ideea de baza este de a folosi tranzițiile observate pentru a ajusta utilitatea stării i pentru a se potrivi cu utilitatea stării succesoare j folosind ecuația:

$$U(i) \leftarrow U(i) + c(R(i) + U(j) - U(i))$$

unde c este o constantă de învățare.

Această metodă funcționează cu atât mai bine cu cât numărul de secvențe de antrenament în care apare fiecare stare este mai mare. Constanta c este stabilită anterior învățării, în funcție de dimensiunea setului de antrenare.

Diferența principală între învățarea adaptiv dinamică și învățarea după diferența temporală este că învățarea după diferența temporală ajustează utilitatea unei stări în conformitate cu un succesor iar învățarea adaptiv dinamică ajustează utilitățile stărilor în conformitate cu toți succesorii lor, în funcție de probabilități. O ajustare în cazul învățării adaptiv dinamice se propagă în întreaga listă de utilități, pe când în cazul învățării după diferența temporală o ajustare se face numai la nivelul unei tranziții locale. Pentru a diminua efectul de propagare, un sistem de învățare adaptiv dinamic poate adopta o regula euristică de genul: "ajustează utilitățile doar stărilor pentru care utilitățile succesorilor cei mai probabil au fost mult modificate în pasul curent".

6.3.9 Învățarea pasivă într-un domeniu necunoscut de stări

Deoarece învățarea după diferența temporală și învățarea naivă nu țin cont de modelul M de calcul al probabilităților de trecere de la o stare la alta, ele funcționează la fel și în cazul unui domeniu de stări necunoscut.

Chiar dacă nu cunoaștem modelul M , dacă domeniul de stări este suficient de puțin complex, putem estima un model M calculând probabilitatea ca din starea i să trecem în starea j ca fiind S_{ij}/S_i , unde S_{ij} este numărul de tranziții de la i la j iar S_i numărul total al tranzițiilor de la i la altă stare.

6.3.10 Învățarea activă

Diferența dintre un sistem pasiv și unul activ de învățare este aceea că sistemul pasiv folosește o strategie predefinită pentru a decide ce pași să urmeze, iar un sistem activ decide ce urmează să facă în funcție de cum estimează că va fi afectat rezultatul învățării. Pentru a reprezenta un sistem activ de învățare, unui domeniu de stări îi atașăm modelul M ce nu va calcula probabilitatea de trecere din starea i în starea j , ci probabilitatea de trecere din starea i în starea j în urma acțiunii a . Utilitatea unei stări va fi calculată după formula:

$$U(i) = R(i) + \max_a \sum_j M_{ij}^a U(j)$$

Mai este utilă definirea unei funcții de calcul a utilității acțiunii a în starea i . Putem calcula acest lucru după formula:

$$U_a(i) = \max_a Q(a, i)$$

unde $Q(a,i)$ reprezintă valoarea efectuării acțiunii a în starea i rezultată în urma unei secvențe de antrenament în care are loc acea acțiune.

Practic un sistem activ de învățare își auto-evaluează acțiunile posibile la fiecare pas de execuție, selectând cele care măresc probabilitatea atingerii unei stări optime.

6.3.11 Algoritmii genetici

Algoritmii genetici sunt inspirați de mecanismul selecției naturale. În natură, organismele care nu sunt adaptate mediului înconjurător nu mai evoluează și dispar în timp, iar cele adaptabile continuă să se dezvolte. De asemenea, o generație a oricărui organism tinde să păstreze trăsăturile predecesorilor la care mai adaugă eventual trăsături noi dezvoltate. Algoritmii genetici simulează aceste comportamente.

Un algoritm genetic are un mecanism de a selecta dintr-o “populație” de șabloane pe cele mai “adaptate” pentru a le dezvolta ulterior, renunțând la celelalte. Un **indiviz** este reprezentat printr-un șir de biți ce codifică atributele sale definitorii. Fiecare element component al acelui șir de biți se numește **genă**.

Funcționarea unui algoritm genetic constă în trecerea prin mai multe generații ale unei populații inițiale. La fiecare pas algoritmul selectează indivizii cu probabilitatea cea mai mare de a fi optimi. Indivizii astfel selectați sunt folosiți pentru crearea generației viitoare folosind părți din codul lor de gene pentru a forma alte șiruri de gene ce definesc alți indivizi. Algoritmul mai atribuie și o șansă ca la fiecare nou cod de gene format să aibă loc o mutație ce va modifica la întâmplare unul sau mai mulți biți.

Avantajul unui algoritm de acest tip este că nu necesită nici un fel de cunoștințe anterioare pentru a rezolva orice fel de problemă învățabilă, dar rezolvarea durează mai mult decât dacă am folosi alte metode.

Algoritmii evolutivi sunt o formă mai avansată a algoritmilor genetici în care un individ reprezintă o structură mai complexă, cum ar fi o secvență de program, iar formarea de noi indivizi se face după metode mai elaborate.

Sistemele de clasificare sunt algoritmi genetici care dezvoltă reguli de clasificare a unor instanțe. Fiecărei reguli i se atribuie un cost și are un set de alte reguli ca condiții și alt set ca rezultate. La fiecare pas, se selectează regulile ce pot fi aplicate în starea curentă a sistemului, și se alege cea care va fi aplicată în funcție de costul sau și de relațiile sale cu celelalte reguli. Costurile regulilor sunt ajustate în funcție de contribuția lor la atingerea obiectivului sistemului de clasificare.

6.3.12 Învățarea bazată pe cunoștințe

Metodele de învățare prezentate mai sus se bazează pe generalizarea unui set de instanțe pentru a forma un concept. Locul și modul în care învățarea

generalizează un șablon este ales numai în funcție de regularitățile identificate în setul de instanțe de antrenament. Aceste metode nu țin cont de semantica acelor instanțe, ci numai de reprezentarea lor formală.

Învățarea care folosește o sursă anterioară de cunoștințe referitoare la domeniul problemei pentru a ghida formarea unui concept nou se numește **învățare supervizată deductivă** sau **învățare bazată pe cunoștințe**. Avantajele acestei metode de învățare sunt semnificative: setul de instanțe de antrenament poate fi mult redus, iar șansa de a ajunge la un șablon corect este mai mare, datorită eliminării unor generalizări făcute pe baza unor similarități aparente, dar false.

Șabloanele posibile nu sunt extrase din regularități identificate în instanțele de antrenament, ci din cunoștințele anterioare ce explică măcar una din clasificările cunoscute. Învățarea deductivă nu produce nici o regulă nouă ce nu poate fi dedusă din baza de cunoștințe. Orice rezultat al unei învățări deductive ar putea fi deci obținut și printr-un mecanism de inferență pe baza regulilor cunoscute. Utilitatea învățării deductive stă însă în faptul că este o metodă mult mai rapidă de a extrage concepte din reguli cunoscute, căci oferă un mecanism de a direcționa căutarea spre conceptul dorit.

6.3.13 Învățarea bazată pe explicații

Învățarea bazată pe explicații (EBL – Explanation Based Learning) pleacă de la o bază de cunoștințe anterioare despre domeniul problemei și caută să găsească explicația clasificărilor făcute în instanțele de antrenament. Când o explicație este găsită, ea este generalizată pentru a include cât mai multe instanțe similare.

Fie baza de cunoștințe următoare:

```
rotund(X) ∧ ușor(X) => minge(X)
mic(X) => ușor(X)
fără_colțuri(X) => rotund(X)
```

Din această bază de cunoștințe și din instanța de antrenament:

```
culoare(obj,roșu) ∧ mic(obj) ∧ rotund (obj) => minge(obj)
```

dorim să obținem un șablon pentru a identifica obiecte cu proprietatea *minge(obj)*. Un algoritm de învățare bazat pe explicații va găsi justificarea instanței pe baza cunoștințelor, și anume:

```
mic(obj) => ușor(obj)
minge(obj) → uşor(obj) ∧ rotund(obj) =>
               rotund (obj)
```

Prin generalizare, va fi format șablonul:

```
mic(X) ∧ rotund(X) => minge(X)
```

Forma generală a algoritmului de învățare bazată pe explicații, când P este premiza, C – concluzia unei reguli din baza de cunoștințe, iar S – șablonul curent propus, este:

```

EBL (P, C, S)
begin
  Ts = cel mai general unificator al PxS si CxS;
  S = SxTs;
  Tg = cel mai general unificator al PxG si CxG;
  G = GxTg;
  if (P și C sunt compatibile cu o instanță de antrenament)
    begin
      Ts = cel mai general unificator al PxS si CxS;
      S = SxTs;
    end;
end.

```

G reprezintă lista substituțiilor de generalizare (înlocuire a unei constante cu o variabilă în premiza unei reguli).

6.3.14 Învățarea bazată pe relevanță

Învățarea bazată pe relevanță este o formă de învățare automată bazată pe o baza de cunoștințe ce conține informații privind relevanța unor anumite proprietăți ale unei instanțe, relativ la apartenența sa la conceptul țintă. De exemplu, proprietatea unui obiect de a fi rotund are o relevanță mare în privința apartenenței sale la conceptul *minge*, în schimb proprietatea sa de a avea culoarea roșie este irelevantă.

Aceasta metodă presupune ca baza de cunoștințe să conțină dependențe funcționale care să contribuie la definirea conceptului. Algoritmul de învățare va încerca, pe această bază, să găsească implicații consistente cu proprietățile cunoscute ale instanțelor de antrenament.

6.3.15 Învățarea inductivă bazată pe cunoștințe

Învățarea inductivă bazată pe cunoștințe folosește regulile cunoscute și instanțele de antrenament pentru a încerca să formeze reguli noi care să explice proprietățile care nu sunt justificabile pe baza cunoștințelor anterioare. De exemplu, pentru baza de cunoștințe:

```

rotund(X) ∧ ușor(X) ∧ sare(X) => minge(X)
mic(X) => ușor(X)

```

și instanțele:

```

rotund(obj1) ∧ mic(obj1) ∧ elastic (obj1) ∧ sare (obj1) =>
minge(obj1)
rotund(obj2) ∧ ușor(obj2) ∧ elastic (obj2) ∧ sare (obj2) =>
minge(obj2)
mic(obj3) ∧ sare (obj3) => nu_minge(obj3)

```

un algoritm de învățare inductivă ar observa concordanța între proprietățile unui obiect *ușor*, *elastic* și care *sare*. Regula obținută ar fi:

```

ușor(X) ∧ elastic (X) => sare(X)

```

Principalele variante ale învățării inductive bazate pe explicații sunt **rezoluția inversă** și **învățarea top-down**.

Rezoluția inversă se bazează pe faptul că dacă un concept poate clasifica o instanță doar pe baza setului de cunoștințe atunci această clasificare se poate demonstra printr-o rezoluție. Scopul este de a inversa rezoluția pentru a afla ipotezele inițiale de la care a plecat.

Astfel, pentru un rezolvent C sunt obținute două clauze $C1$ și $C2$ din care, pe baza regulilor cunoscute, se poate obține C . Clauzele finale obținute astfel devin premisele unei reguli a cărei concluzie este C .

Învățarea *top-down* este o generalizare a arborilor de decizie pentru a acoperi logica clasică. Ideea este de a pleca cu o regulă foarte generală, care va fi specializată pentru a acoperi doar datele cunoscute. Proprietățile unei instanțe sunt reprezentate ca literalii, iar șablonul este un set de clauze ce simulează un arbore de decizie.

Plecând de la un concept-țintă și un set de instanțe de antrenament, un algoritm de învățare top-down construiește o clauză Horn, inițial vidă, care clasifică o instanță în conceptul țintă. La această clauză se adaugă treptat literalii care vor îmbunătăți cel mai mult acuratețea clasificărilor. Algoritmul continuă până la obținerea unei clauze care este satisfăcută de o parte a instanțelor pozitive de antrenament și de nici o instanță negativă de antrenament. Clauza apoi este generalizată pentru a acoperi toate instanțele pozitive de antrenament. Alegerea literalilor ce urmează să fie adăugați se face pe baza unor reguli euristice.

6.3.16 Dezavantajele învățării supervizate

Toate metodele de învățare prezentate în secțiunile precedente presupun existența unui set de instanțe de antrenament despre care știm dacă aparțin sau nu unui concept țintă. Ele sunt folosite pentru a verifica și direcționa modificarea șablonului învățat de algoritm.

Necesitatea existenței acestor instanțe este punctul slab al învățării supervizate, căci ele introduc o serie de probleme, cum ar fi:

Problema limbajului de reprezentare: trebuie găsită o modalitate de a reprezenta instanțele și conceptul-țintă într-o formă care să permită verificarea, generalizarea și specializarea unor șabloane. Un limbaj simplu de reprezentare poate fi găsit relativ ușor pentru orice domeniu, dar el poate duce la spații foarte mari de căutare a conceptului-țintă.

Inconsistența datelor: instanțele de antrenament și eventualele cunoștințe anterioare ale sistemului pot fi inconsistente cu existența unui concept țintă. Erorile pot proveni atât din clasificarea instanțelor cât și din incapacitatea limbajului de reprezentare ales de a descrie conceptul țintă.

Descrierea conceptului țintă: un algoritm de învățare poate să nu găsească nici un șablon corespunzător conceptului țintă sau poate să găsească mai multe șabloane corespunzătoare. În cazul în care se identifică mai multor șabloane posibile, este probabil ca ele să fie inconsistente între ele, deci insuficient de generale.

6.4. Învățarea nesupervizată

Învățarea nesupervizată elimină complet necesitatea unor instanțe de antrenament, deci și problemele legate de acestea. Scopul învățării nesupervizate nu este definit anterior ca un concept țintă, algoritmul fiind lăsat singur să identifice concepte posibile.

În general, învățarea nesupervizată presupune existența unor instanțe neclasificate, un set de reguli euristice pentru crearea de noi instanțe și evaluarea unor concepte deduse, eventual un model general al spațiului de cunoștințe în care se găsesc aceste instanțe. Un algoritm de învățare nesupervizată construiește concepte pentru a clasifica instanțele, le evaluează și le dezvoltă pe cele considerate “interesante” de regulile euristice. În general, concepte interesante sunt considerate cele care acoperă o parte din instanțe, dar nu pe toate.

Învățarea nesupervizată permite identificarea unor concepte complet noi plecând de la date cunoscute. Încercări de a aplica acest tip de învățare în cercetarea științifică au dus la rezultate semnificative. Astfel AM (Davis și Lenat, 1982) pleca de la un set de concepte de bază din teoria mulțimilor, un set de operații de creare a noi concepte prin modificarea și combinarea celor existente, și un set de reguli euristice pentru a alege conceptele interesante. Algoritmul a descoperit numerele naturale, conceptul de număr prim, precum și o serie de alte concepte din teoria numerelor. BACON (Langley, 1987) a fost o încercare de a dezvolta un model computațional de dezvoltare a unor legi cantitative științifice noi. Folosind date privind relația între distanțele dintre planete și soare și perioada lor de revoluție, BACON a re-descoperit legile lui Kepler privind mișcarea planetelor.

Totuși aceste încercări s-au dovedit limitate în rezultate. Principalul factor ce limitează numărul și relevanța conceptelor învățate de acest gen de algoritmi este faptul că ele nu pot învăța noi metode de a crea și evalua concepte. Pentru a obține rezultate mai relevante, ar trebui întâi descris un set mult mai complex de operații pentru crearea de noi concepte, precum și niște reguli euristice mai flexibile pentru evalua aceste concepte.

6.4.1 Identificarea claselor de instanțe

Un domeniu în care învățarea nesupervizată și-a dovedit utilitatea este cel al identificării automate de clase în mulțimi neclasificate de instanțe. Această problemă presupune existența unei mulțimi de obiecte negrupate și a unor mijloace de a găsi și măsura similarități între aceste obiecte. Scopul unui algoritm de identificare a unor clase de obiecte este de a grupa obiectele într-o ierarhie de clase după criterii cum ar fi maximizarea similarității obiectelor din aceeași clasă. Ierarhia de clase este reprezentată de obicei ca un arbore, fii unui nod reprezentând categorii distincte dar incluse în categoria părinte. În privința învățării automate aplicate la identificarea claselor de instanțe există două abordări principale: **taxonomia numerică** și **gruparea conceptuală**.

Taxonomia numerică se bazează pe reprezentarea unui obiect ca o colecție de attribute cuantificabile. Dacă reprezentăm un obiect ca un vector numeric de proprietăți, atunci similaritatea dintre două obiecte cu n proprietăți reprezintă distanța dintre reprezentările vectorilor respectivi în spațiul n -dimensional. Un algoritm de clasificare bazată pe taxonomia numerică pentru mulțimea de obiecte O are forma:


```

CalsificareTaxonomica(O)
begin
  do{
    selectează din O perechea (X,Y) de obiecte cu cel mai mare
    grad de similaritate;
    elimină X și Y din O;
    formează o clasă nouă definită de  $f(X,Y)$ , unde  $f$  este de
    obicei funcția medie aritmetică;
    while (există un obiect din O pentru care gradul de
    similaritate cu clasa nou definită este mai mare de un prag
    S)
      begin
        adaugă obiectul la clasă;
        elimină obiectul din O;
      end;
  }while (O nu este mulțime vidă)
end.

```

Algoritmul poate fi extins și la obiecte reprezentate ca seturi de simboluri și nu vectori numerici. Singurul obstacol este găsirea unei metode de a măsura similaritățile dintre obiecte. O soluție simplă este de a calcula gradul de similaritate ca fiind proporția dintre numărul de proprietăți comune și numărul total de proprietăți. Cu cât această proporție va fi mai aproape de 1, cu atât obiectele respective vor avea șansa mai mare de a aparține aceleiași clase.

Dezavantajul acestei metode de clasificare stă în faptul că nu ține cont de rolul semantic al proprietăților obiectelor. De exemplu, pentru obiectele:

```

obiect1 = (rotund, roșu, mic)
obiect2 = (pătrat, roșu, mic)
obiect3 = (rotund, verde, mare)

```

va fi găsit un grad de similaritate mai mare între *obiect1* și *obiect2* decât între *obiect1* și *obiect3*, deși proprietatea mai relevantă ar trebui să fie *rotund* și nu *roșu* și *mic*. Această metodă formează deci clase prin aprecieri subiective asupra apropierii dintre obiecte, și nu folosește nici un mecanism de a diferenția proprietățile în funcție de relevanța lor în clasificare.

Un alt dezavantaj constă în faptul că această metodă de clasificare nu construiește definiția conceptelor pe baza cărora clasifică obiectele. Clasele sunt reprezentate extensional (ca o enumerare de obiecte) și nu intensional, ca un șablon de recunoaștere a obiectelor fiecărei clase.

Gruparea conceptuală rezolvă această problemă prin folosirea unor metode de învățare automată pentru a defini concepte plecând de la o bază de cunoștințe asupra domeniului clasificării. Forma generală a unui algoritm de învățare pentru clasificarea conceptuală, așa cum a fost descris în CLUSTER/2 (Michalski și Stepp, 1983), este:

```

ClasificareConceptuala(O)
begin

```

```

do{
  selectează k obiecte din O (la întâmplare sau folosind o
  funcție de selecție);
do{
  for ( fiecare obiect selectat ) folosind obiectul ca
  instanță pozitivă și celelalte obiecte selectate ca
  instanțe negative, construiește o definiție generală a
  conceptului clasei construite în jurul obiectului selectat;
  clasifică toate obiectele din O folosind definițiile
  claselor obținute;
  for (fiecare definiție)
    while ( obiecte din celelalte clase nu sunt acoperite de
    această definiție ) generalizează definiția fiecărei
    clase;
  folosind o metodă numerică, caută elementul cel mai
  apropiat de centrul fiecărei k clase formate;
} while (nu s-au format clase satisfăcătoare și numărul de
încercări e mai mic decât un prag stabilit);
} while (nu au fost obținute clase satisfăcătoare);
end.

```

6.4.2 Structura cunoștințelor taxonomice

Algoritmii de învățare supervizată și metodele de clasificare prezentate mai sus definesc o clasă printr-o serie de attribute necesare și suficiente pentru apartenența la acea clasă. Deși eficientă în multe situații, această definiție nu permite o clasificare flexibilă și structurată așa cum întâlnim în conceptele umane. De exemplu, un om poate recunoaște un obiect ca fiind un exemplu mai bun al unei categorii, iar alt obiect un exemplu mai prost. O vrabie este un exemplu mai bun al conceptului *pasăre* decât un penguin. Această diferențiere nu este permisă de definiția clasică a unei clase.

Teoria asemănării familiale (Wittgestein, 1953) dă o definiție mai apropiată de cea umană pentru o clasă. Astfel, o clasă este definită de sistemul complex de similarități între membrii ei, nu de o serie de attribute necesare și suficiente pentru apartenența la acea clasă. Această definiție permite, în extremis, ca membrii unei clase să nu aibă nici măcar o proprietate comună. Un exemplu ar fi cel al clasei *sport*, ai cărei membri diferă prin numărul de jucători, regulile de joc, precum și alte proprietăți, totuși clasa este bine definită și neambiguă.

Clasificările umane mai diferă de cele formale și prin identificarea unor clase de bază mult mai relevante în clasificare decât generalizările sau specializările lor. Conceptul *autoturism* este mai util în descrierea unui obiect decât generalizarea sa *vehicul* sau o specializare a sa *cabrioletă*. În spatele unui astfel de concept de bază sunt mai multe informații recunoscute de un clasificator uman, dar nu și de un sistem automat de clasificare. Orice metodă ce își propune să identifice clase noi de obiecte trebuie să țină cont de mecanismul clasificării umane.

COBWEB (Fisher, 1987) este un algoritm de clasificare ce își propune să se apropie de modelul uman. COBWEB recunoaște clase de bază și grade de apartenență la clase. COBWEB este un algoritm de învățare incrementală

ce definește un număr optim de clase plecând de la un set de instanțe neclasificate.

Clasele sunt reprezentate printr-un set de proprietăți, fiecare valoare posibilă a unei proprietăți având atașată probabilitatea $P(p_i=v_{ij} \mid c_k)$ ca proprietatea p_i să aibă valoarea v_{ij} pentru un obiect aparținând clasei c_k . Atunci când primește o instanță nouă, COBWEB consideră utilitatea plasării instanței într-o clasă existentă sau a creării unei noi clase plecând de la acea instanță. Criteriul folosit pentru evaluarea calitativă a unei clasificări se numește *utilitatea claselor* (Gluck și Corter, 1985). Acest criteriu încearcă să maximizeze atât probabilitatea ca obiectele din aceeași categorie să aibă valori comune pentru o proprietate cât și probabilitatea ca obiecte din categorii diferite să aibă valori diferite ale aceleiași proprietăți. Utilitatea claselor se calculează pentru toate clasele c_k , toate proprietățile p_i și toate valorile posibile v_{ij} ale acelei proprietăți, ca:

$$\sum_k \sum_i \sum_j P(p_i=v_{ij}) P(p_i=v_{ij} \mid c_k) P(c_k \mid p_i=v_{ij})$$

Maximizarea acestei sume înseamnă îmbunătățirea modului în care sunt construite clasele.

Algoritmul COBWEB este definit, pentru intrările *Nod* și *Instanță*, după cum urmează:

```

Cobweb(Nod, Instanță)
begin
  if (Nod este nod frunză) then
    begin
      crează doi fii ai nodului Nod, F1 și F2;
      inițializează probabilitățile din F1 cu cele din Nod;
      inițializează probabilitățile din F2 cu cele din
      Instanță;
      adaugă Instanță la Nod, modificând corespunzător
      probabilitățile;
    end;
  else
    begin
      adaugă Instanță la Nod, modificând corespunzător
      probabilitățile;
      for ( fiecare descendent D al Nod) calculează
      utilitatea claselor obținute prin includerea Instanță
      în D;
      fie S1 scorul obținut de cel mai bun set de clase
      D1;
      fie S2 scorul obținut de al doilea mai bun set de
      clase D2;
      fie S3 scorul obținut dacă am plasa Instanță într- o
      clasă nouă;
      fie S4 scorul obținut dacă am reuni D1 și D2 într- un
      set de clase;
      fie S5 scorul obținut dacă am înlocui D1 cu fii săi;
    end;
  end;

```

```

if ( S1 este cel mai bun scor ) then
    Cobweb(D1, Instanță);
else
    if ( S3 este cel mai bun scor ) then
        inițializează probabilitățile din noua clasă cu
        cele din Instanță;
    else
        if ( S4 este cel mai bun scor ) then
            begin
                fie Dn setul de clase obținut prin reunirea
                D1 și D2;
                Cobweb(Dn, Instanță);
            end;
        else
            if ( S5 este cel mai bun scor ) then
                begin
                    înlocuiește D1 cu fii săi;
                    Cobweb (Nod, Instanță);
                end;
        end;
end.

```

Algoritmul face o căutare în spațiul taxonomiilor posibile folosind utilitatea claselor pentru a evalua și selecta seturile posibile de clase. Inițial, pleacă cu o singură clasă cu probabilitățile primei instanțe primite. Pentru fiecare instanță ulterioară, algoritmul parcurge arborele de clase (format inițial dintr-o singură clasă) și evaluează cel mai bun pas, dintre variantele:

- adaugă o nouă clasă formată din instanța nouă;
- adaugă instanța nouă la clasa existentă la care se potrivește cel mai bine;
- reunește două clase existente în una singură și adaugă instanța la clasa rezultată;
- înlocuiește o clasă existentă cu fiii ei și adaugă instanța la clasa rezultată la care se potrivește cel mai bine.

COBWEB este un algoritm eficient ce produce taxonomii cu un număr rezonabil de clase. Prezintă tendința de a grupa instanțe în clase de bază, și datorită utilizării probabilităților, acceptă noțiunea de grad de apartenență la o clasă.

6.5. Concluzii

Învățarea automată tradițională se bazează pe un model conceptual rațional al lumii fizice. Pe baza acestui model s-au obținut structuri de reprezentare tot mai complexe, strategii de căutare mai eficiente și progrese semnificative atât în crearea unor sisteme ce simulează aspecte ale inteligenței biologice, cât și în înțelegerea modului în care funcționează inteligența umană. Ideea unui model conceptual rațional al lumii fizice se bazează însă pe tradiția filozofică raționalistă, care însă nu modelează și felul în care raționează inteligența biologică. Inteligența umană are la bază atât raționamente logice și științifice cât și raționamente empirice, esențiale mai ales în interpretarea necunoscutului.

Acest lucru a dus la crearea de noi metode de modelare a inteligenței, cum ar fi rețelele neuronale și algoritmi genetici, sau la includerea unor raționamente empirice similare celor umane în algoritmi de învățare mai tradiționali. Rezultatele

semnificative date de aplicarea acestor noi idei promit crearea unor sisteme tot mai inteligente, până la crearea primei inteligențe artificiale reale. Probabil, prima inteligență artificială veritabilă nu va fi creată de programatori umani, ci de un sistem artificial ce va *învăța* să fie inteligent observând *istanțele* date de raționamentele umane.

Metodele învățării automate au devenit instrumente de bază în implementarea sistemelor complexe de inteligență artificială. Există numeroase centre de cercetare și conferințe anuale având ca principal domeniu dezvoltarea și evaluarea metodologiilor de învățare automată (vezi bibliografia). Tendința generală de a crea sisteme informatice tot mai independente și adaptabile face studiul *Învățării Automate* unul dintre cele mai atractive domenii de cercetare în *Inteligența Artificială*.

Datorită limitărilor de spațiu, acest capitol prezintă doar o parte din algoritmi de învățare existenți, având ca scop doar atragerea interesului cititorului pentru documentare suplimentară în acest domeniu.

Cerințe pentru studenți

- Să fie familiarizați cu cel puțin două metode de învățare automată (dintre care una – ID3).
- Să poată recunoaște cele mai indicate metode de învățare aplicabile unei probleme reale.

Probleme

P6.1 Dați exemple de aplicații pentru care învățarea automată nu poate fi aplicată în practică. Explicați.

P6.2 Dați exemple de metode de învățarea automată pentru fiecare din categoriile:

- învățare supervizată, fără surse de cunoaștere
- învățare supervizată, cu surse de cunoaștere
- învățare nesupervizată

P6.3 Un supervisor dă scoruri pentru stările unui joc de șah (situația de pe tablă la un moment dat). Un program de învățare poate recunoaște mutările optime la un moment dat, alegând cea care duce la starea cu scor maximal. Dacă însă programul de învățare nu ar dispune de exemplele de antrenament, cum ar putea învăța mutările optime?

P6.4 Pentru construirea unui set de exemple de antrenament putem genera stări ce vor fi evaluate de un supervisor, urmând una din metodele următoare:

- se generează la întâmplare o stare legală a pieselor de pe tablă;
- se generează o stare plecând de la alta deja evaluată, din care se execută o mutare legală

Care din cele două metode este mai eficientă (în sensul maximizării stărilor relevante pentru învățare – cu câștig maxim de informație).

P6.5 Construiți arborele ID3 pentru instanțele de antrenament din tabela 6.2. Apoi repetați pentru învățarea conceptului de *umiditate*, în loc de *optim ski*, utilizând atributele corespunzătoare din instanțele de antrenament.

Tabela 6.2 *Instanțe de antrenare pentru exercițiul 6.5*

Zi	prognoza	cer	temperatura	umiditatea	vântul	Optim ski?
1	optimistă	senin	cald	mare	slab	nu
2	optimistă	înnourat	cald	mare	puterninc	nu
3	neutră	înnourat	cald	mare	slab	da
4	proastă	senin	optim	mare	slab	da
5	proastă	senin	frig	normală	slab	da
6	proastă	senin	frig	normală	puterninc	nu
7	neutră	senin	frig	normală	puterninc	da
8	optimistă	înnourat	optim	mare	slab	nu
9	optimistă	senin	frig	normală	slab	da
10	proastă	înnourat	optim	normal	slab	da
11	optimistă	înnourat	optim	normală	puterninc	da
12	neutră	senin	optim	mare	puterninc	da
13	neutră	senin	cald	normală	slab	da
14	proastă	înnourat	optim	mare	puterninc	nu

P6.6 Construiți arborele ID3 pentru descrierea funcției:

$$(A \wedge B) \text{ xor } (C \vee D)$$

P6.7 Dacă $A1$ și $A2$ sunt arbori de decizie și $A2$ îl conține pe $A1$, poate fi $A2$ rescris ca un arbore format din $A1$ și un alt arbore de decizie $A3$ atașat la una sau mai multe noduri terminale din $A1$? Explicați și dați exemple.

P6.8 Dați o soluție mai rapidă decât ID3 pentru învățarea automată a unor reguli de clasificare pentru instanțele din tabela 6.2.

P6.9 Ce poate fi învățat aplicând EBL pentru instanțele de antrenament:

```
Iași (aproape (Botoșani), aeroport (da), drum (Botoșani))
Botoșani (aproape (Iași), aeroport (nu), drum (Iași))
```

și baza de cunoștințe:

```
A: drum (B) ^ aproape (B) => condus (A, B)
A: aeroport (da) => zbor (A, any)
condus (A, B) ^ zbor (B, any) => zbor (A, any)
```

P6.10 Descrieți o metodă de a reformula automat conceptul țintă pentru un algoritm de învățare supervizat în cazul insuficienței datelor de antrenament. Exemple.

Bibliografie

- Alpaydin E. 2004. *Introduction To Machine Learning*, MIT Press.
- Davis, R. și Lenat, D.B. 1982. *Knowledge-Based Systems in Artificial Intelligence*. New York: McGraw-Hill International Book Company, pp. 1-225.
- DeJong G, Mooney R. 1986 *Explanation-based learning. An alternative view*, Machine Learning.
- Fisher, D. H. 1987. *Knowledge acquisition via incremental conceptual clustering*. Machine Learning 2: 139–172.
- Fisher, D.H. 1991. *Concept Formation: Knowledge and Experience in Unsupervised Learning*, Lawrence Erlbaum.
- Gluck, M. A. și Corter, J. E. 1985. *Information, uncertainty, and the utility of categories*. În Proceedings of the Seventh Annual Conference of the Cognitive Science Society, 283--287. Hillsdale, NJ: Lawrence Erlbaum.
- Jebara T. 2004. *Machine Learning: discriminative and generative*, Springer.
- Langley, P. 1987. *A General Theory of Discrimination Learning*. In Production System Models of Learning and Development, Eds D.Klahr, P.Langley & R.Neches, MIT Press.
- Michalski R.S. și Stepp R.E. 1983. *Learning from observation: Conceptual clustering*. În R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, Machine Learning: An AI Approach, volume 1, pages 331-364. Morgan Kaufmann Publishers.
- Mill, J. S. 1843. *Collected Works*, Toronto: University of Toronto Press, 1963-89), XIII, 566.
- Mitchell T.M. 1997. *Machine Learning*, McGraw-Hill.
- Mitchell, T.M. 1982. *Generalization as search*. Artificial intelligence, March, 18(2), 203–226.
- Mitchell, T.M. 1978. *Version spaces: An approach to concept learning*. PhD thesis, Department of Electrical Engineering, Stanford University. Also Stanford CS reports STAN-CS-78-711, HPP-79-2.
- Quinlan J.R. 1986. *Induction of Decision Trees*, Machine Learning.
- Quinlan J.R. 1993. *Programs for Machine Learning*, Morgan Kaufmann.

Quinlan, J.R. 1983. *Learning Efficient Classification Procedures And Their Application To Chess End Games.*, Michalski, Ryszard S., J. G. Carbonell . T. M. Mitchell, : Machine Learning - An Artificial Intelligence Approach, . 463-482. Tioga Publishing Company.

Shannon C.E. 1948. *A Mathematical Theory of Communication*, Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656.

Shavlik J., Mooney R.J., Towell G.G. 1991. *Machine Learning*, Morgan Kaufmann.

Utgoff P. 1988. *ID5: An incremental ID3*. In Proceedings of the Fifth International Conference on Machine Learning, pages 107-120, Ann Arbor, MI, Morgan Kaufmann.

Valiant, L. G. 1984. *A theory of the learnable*. Communications of the ACM 1984 pp1134-1142.

Weiss S.M., Kuliowski G 1993 *Computational systems that learn: classification and prediction methods from statistics, neural nets, machine learning and expert systems*, Morgan Kaufmann.

Widrow, B. și Hoff, M. E. 1960. *Adaptive switching circuits*. în IRE WESCON, pag 96-104, New York. Convention Record.

Wittgenstein L. 1953. *Philosophical Investigations*, New york, Macmillan.

Site-uri utile:

Colecție de site-uri despre Machine Learning: <http://www.aaai.org/AITopics/html/machine.html>

Cursuri disponibile on-line: <http://www.cs.iastate.edu/~honavar/Courses/cs673/machine-learning-courses.html>

Conferințe și asociații:

European Conference on Machine Learning: <http://www.ecmlpkdd2007.org/>

Knowledge Discovery and Data: <http://www.acm.org/sigs/sigkdd/kdd/2007/>

Association for the Advancement of Artificial Intelligence's National Conferene on Artificial Intelligence: <http://www.aaai.org/Conferences/AAAI/aaai07.php>