

**FLORENTINA HRISTEA
MARIA FLORINA BALCAN**

**ASPECTE ALE CĂUTĂRII ȘI
REPREZENTĂRII CUNOȘTINȚELOR
ÎN INTELIGENȚA ARTIFICIALĂ**

**EDITURA UNIVERSITĂȚII DIN BUCUREȘTI
2004**

FMI-UNIBUC

CUVÂNT ÎNAINTE

Lucrarea de față reprezintă un suport al cursului de inteligență artificială, precum și al seminarului corespunzător, pe care cele două autoare le țin studenților din anul al patrulea, secția “Matematică - Informatică” a Facultății de Matematică și Informatică - Universitatea din București. În egală măsură, ne adresăm studenților secției “Informatică”, precum și celor de la cursurile postuniversitare de specializare în informatică ai aceleiași facultăți, care studiază această disciplină.

Deși își propune, în primul rând, să realizeze o introducere în domeniul inteligenței artificiale, lucrarea poate constitui, de asemenea, suportul unui curs special (optional) axat pe principalele aspecte ale căutării și reprezentării cunoștințelor, cu specială referire la sistemele expert și la implementarea acestora în limbajul Prolog.

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

Structurată în cinci capitole și o anexă, lucrarea de față urmărește aspectele de bază referitoare la căutare (ca tehnică de rezolvare a problemelor ce explorează în mod sistematic un spațiu de stări ale problemei) și la reprezentarea cunoștințelor. Exemplul clasic de problemă de căutare prezentat în cadrul acestui curs este acela al jocurilor. Principala aplicație fundamentală a problematicii reprezentării cunoștințelor în inteligență artificială, pe care o prezentăm aici, se referă la sistemele expert și la implementarea acestora în limbajul Prolog. De departe de a pretinde că epuizează aspectele legate de problematica reprezentării cunoștințelor, lucrarea se oprește asupra câtorva dintre cele mai importante sisteme de reprezentare a cunoștințelor, și anume: sistemele bazate pe reguli, structurile de tip slot-and-filler, rețelele semantice și cadrele. Ultimul capitol, intitulat “Raționament statistic”, își propune să întregească imaginea referitoare la tehniciile de reprezentare a cunoștințelor, ocupându-se de două tipuri de abordări statistice pentru sisteme de raționament incert (factorii de certitudine în sistemele bazate pe reguli și, respectiv, rețelele Bayesiene).

Nu în ultimul rând, considerăm lucrarea utilă și prin numeroasele exemple de programe pe care le oferă și a căror existență este apreciată de noi ca fiind indispensabilă în vederea unei înțelegeri aprofundate a tematicii abordate.

Nădăjduim că această lucrare îi va ajuta pe studenții noștri, și nu numai pe ei, să se apropie cu interes de un domeniu atât de vast, și, în același timp, de fascinant, cum este cel al inteligenței artificiale.

Autoarele

CAPITOLUL 1

CONSIDERAȚII PRELIMINARE

Inteligenta artificială poate fi privită ca fiind o știință sui-generis al cărei obiect de studiu îl constituie modul de programare a calculatoarelor în vederea săvârșirii unor operațiuni pe care, deocamdată, oamenii le efectuează mai bine. Conform acestei definiții primare, simplificate (și care nu este general acceptată) putem gândi inteligența artificială ca fiind acel domeniu al informaticii care se ocupă de *automatizarea comportamentului intelligent*.

O asemenea definiție¹ privește inteligența artificială ca fiind un domeniu al informaticii și, prin urmare, se bazează pe principii teoretice și practice ale acesteia, cum ar fi:

- *structurile de date* folosite în reprezentarea cunoștințelor;
- *algoritmi* necesari pentru aplicarea cunoștințelor;
- *limbajele și tehniciile de programare* folosite la implementarea acestor algoritmi.

În același timp, aşa cum se arată în [9], definiția pe care o comentăm aici este deficitară prin faptul că însuși conceptul de *inteligentă* nu este foarte bine înțeles și precizat. Astfel, problema definirii domeniului inteligenței artificiale devine, până la urmă, una a definirii conceptului însuși de inteligență. Ea presupune ridicarea unor întrebări de tipul: Ce se întâmplă atunci când intervene învățarea? Ce este

¹ Vezi și [11], s.v. *artificial intelligence*, p. 37.

creativitatea? Ce este intuiția? Cum sunt reprezentate cunoștințele în țesutul nervos al unei ființe vii? Este necesar ca un program de calculator intelligent să fie modelat în conformitate cu ceea ce se cunoaște despre inteligență umană sau este suficientă o abordare strict inginerească a problemei? Este posibil să se achiziționeze inteligență pe un calculator sau o entitate intelligentă necesită bogăția senzațiilor și a experienței care nu pot fi găsite decât în existența biologică? Toate aceste întrebări arată legătura inteligenței artificiale cu alte domenii, cum ar fi, în primul rând, filozofia. În același timp, toate aceste întrebări deschise, care nu au încă răspuns, au ajutat la conturarea *problemelor* și a *soluțiilor*, a *metodologilor* care constituie miezul inteligenței artificiale moderne. Inteligența artificială, o disciplină încă Tânără (prin comparație cu altele, cum ar fi matematica sau fizica) ale cărei structură, scopuri, preocupări și metode sunt mai puțin clar definite decât cele ale unor discipline mult mai mature, pune la dispoziție instrumente unice și puternice necesare tocmai în explorarea unor asemenea întrebări.

1.1. Domeniul inteligenței artificiale. Definiții și un scurt istoric

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

Așa cum se arată în [9], există mai multe definiții ale domeniului. Acestea variază de-a lungul a două mari dimensiuni. Prima dimensiune

este aceea a *procesului de gândire și a raționamentului*. Cea de-a doua adresează *comportamentul* (“behavior”). De asemenea, unele definiții măsoară *succesul* în termenii *performanței umane*, în timp ce altele îl măsoară relativ la un concept ideal de inteligență, pe care îl vom numi *rațiune*. Iată câteva definiții ale domeniului inteligenței artificiale, date conform acestor clasificări și preluate de noi din [9]:

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

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

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

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

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

Un an istoric în evoluția domeniului este 1958, an în care McCarthy se mută de la Dartmouth la MIT (“Massachusetts Institute of Technology”). Aici el aduce trei contribuții vitale, toate în același

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

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

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

În timpul primului deceniu de cercetări în inteligență artificială conceptia legată de rezolvarea problemelor era aceea a unui mecanism de

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

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

Tot în perioada 1969-1979, numărul mare de aplicații referitoare la problemele lumii reale a determinat creșterea cererii de *scheme de*

reprezentare a cunoștințelor. Au fost astfel dezvoltate diferite limbaje de reprezentare. Unele dintre ele se bazau pe *logică*², cum este cazul limbajului Prolog³, extrem de popular în Europa.

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

Începând din 1987 și până în prezent, cu precădere în ultimii ani, atât conținutul, cât și metodologia de cercetare în inteligența artificială au suferit o serie de schimbări. Astfel, se caută aplicații din lumea reală, se construiește pe teoriile existente, dar, mai ales, cercetătorii se bazează mult mai mult pe teoreme riguroase și mai puțin decât până acum pe intuiție. Spre exemplu, domeniul *recunoașterii limbajului* a ajuns să fie dominat de aşa-numitele “hidden Markov models” (HMM-uri)⁴. La rândul lor, modelele grafice (care includ, printre altele, rețelele Bayesiene, discutate în ultimul capitol al cursului de față) au devenit

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

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

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

extrem de importante pentru inteligența artificială. În general, putem spune că domeniul inteligenței artificiale se bazează pe teorii matematice riguroase din ce în ce mai mult.

1.2. Subdomenii ale inteligenței artificiale

Principalele **subdomenii** ale inteligenței artificiale sunt considerate a fi [9] următoarele:

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

tența unui robot capabil să execute anumite acțiuni atomice, cum ar fi deplasarea într-o cameră plină cu obstacole).

- **învățarea automată** (datorită căreia se realizează adaptarea la noi circumstanțe, precum și detectarea și extrapolarea unor şabloane - “patterns”. Învățarea se realizează, spre exemplu, prin intermediul așa-numitelor *rețele neurale*⁵. O asemenea rețea reprezintă un tip de sistem de inteligență artificială modelat după neuronii -celulele nervoase-dintron un sistem nervos biologic, în încercarea de a simula modul în care creierul prelucrează informațiile, învăță sau își aduce aminte⁶).

Toate aceste subdomenii ale inteligenței artificiale au anumite trăsături în comun [9], și anume:

- O concentrare asupra problemelor care **nu răspund la soluții algoritmice**; din această cauză tehnică de rezolvare a problemelor specifică inteligenței artificiale este aceea de a se baza pe o căutare euristică⁷.
- Inteligența artificială rezolvă probleme folosind și informație inexactă, care lipsește sau care nu este complet definită și utilizează formalisme de reprezentare ce constituie pentru programator o compensație față de aceste probleme.
- Inteligența artificială folosește raționamente asupra trăsăturilor calitative semnificative ale unei situații.
- Inteligența artificială folosește, în rezolvarea problemelor, mari

⁵ Este acceptat și termenul de *rețea neuronală*.

⁶ Vezi și [11], s.v. *neural network*, p. 348 și s.v. *artificial neural network*, p. 37.

⁷ Prin *euristică* se înțelege o metodă de dirijare sau optimizare a procesului de rezolvare a unei probleme, pe baza unor reguli derivate din experiența, intuiția sau inspirația programatorului. Euristica este de neîmlocuit în cazul unor probleme deosebit de complexe, cum ar fi, spre exemplu, jocul de șah. Vezi și [11], s.v. *heuristic*, p. 247.

cantități de cunoștințe specifice domeniului investigat.

Majoritatea tehniciilor din inteligența artificială folosesc pentru implementarea inteligenței cunoștințe reprezentate în mod explicit și algoritmi de căutare. În inteligența artificială există *două abordări majore* complet diferite ale problematicii domeniului: cea **simbolică** și cea **conecționistă**. Într-o abordare simbolică cunoștințele vor fi reprezentate în mod simbolic. O abordare total diferită -cea coneționistă- este aceea care urmărește să construiască programe inteligente folosind modele care fac un paralelism cu structura neuronilor din creierul uman.

Cursul de față va folosi prima dintre abordările amintite, în cadrul căreia

- **repräsentarea cunoștințelor** adresează problema reprezentării într-un limbaj formal, adică un limbaj adecvat pentru prelucrarea ulterioară de către un calculator, a întregii game de cunoștințe necesare comportamentului intelligent;
- **căutarea** este o tehnică de rezolvare a problemelor care explorează în mod sistematic un spațiu de stări ale problemei, adică de stadii succesive și alternative în procesul de rezolvare a acestoria. (Exemple de stări ale problemei pot fi configurațiile diferite ale tablei de șah în cadrul unui joc sau pașii intermediari într-un proces de raționament).

1.3. Tehnici ale inteligenței artificiale

După scurgerea primelor trei decenii de activitate în domeniul inteligenței artificiale, s-a tras concluzia că *inteligența presupune*

cunoaștere. Aceasta este probabil cea mai importantă concluzie la care s-a ajuns și ea influențează cel mai mult tehnicele specifice domeniului. Așa cum s-a mai arătat, majoritatea tehniciilor din inteligența artificială folosesc pentru implementarea inteligenței cunoștințe reprezentate în mod explicit și algoritmi de căutare. Până la urmă, se poate spune [9] că ***o tehnică a inteligenței artificiale*** este o metodă de exploatare și valorificare a cunoștințelor, care, la rândul lor, ar trebui reprezentate într-un anumit mod, și anume conform următoarelor cerințe [9]:

- Cunoștințele trebuie să înglobeze *generalizări*. Cu alte cuvinte, nu este necesară reprezentarea separată a fiecărei situații în parte. În schimb, situațiile care au în comun proprietăți importante pot fi grupate laolaltă. Dacă cunoștințele nu ar avea această proprietate, ar fi nevoie de foarte multă memorie și de multe operații de actualizare. Prin urmare, o mulțime de informații care nu posedă această proprietate nu va reprezenta cunoștințe, ci ***date***.
- Cunoștințele trebuie să poată fi înțelese de către oamenii care le furnizează. Deși, pentru majoritatea programelor, nucleul de date poate fi achiziționat în mod automat, în multe domenii ale inteligenței artificiale este necesar ca majoritatea cunoștințelor pe care un program trebuie să le aibă să fie furnizate de oameni, în termeni care să poată fi înțeleși de către aceștia.
- Cunoștințele trebuie să poată fi ușor modificate pentru a se corecta erori și pentru a reflecta atât schimbările din lumea înconjurătoare, cât și schimbările din percepția și imaginea noastră despre aceasta.
- Cunoștințele trebuie să poată fi folosite în foarte multe situații, chiar dacă uneori duc lipsă de acuratețe și nu sunt complete.

- Cunoștințele trebuie să poată fi folosite astfel încât să ajute la îngustarea gamei de posibilități care trebuie luate în considerație.

Legat de această problematică, vom mai nota aici doar următorul fapt: *cunoașterea* are anumite proprietăți, care țin de

- Completivitate (Cunoașterea este întotdeauna incompletă. Cunoștințele incorporate într-un sistem software al inteligenței artificiale referitor la o anumită aplicație sunt întotdeauna mult mai puțin complete decât ceea ce se cunoaște despre aplicația respectivă).
- Obiectivitate (Cunoașterea poate fi obiectivă sau subiectivă; este necesar să se facă o distincție între fapte și păreri. Acestea din urmă trebuie asociate agenților).
- Certitudine (Cunoașterea poate fi certă sau incertă; tocmai de aceea teoria probabilităților joacă un rol esențial în multe sisteme ale inteligenței artificiale).
- Formalizare (Cunoștințele pot fi sau nu ușor de formalizat prin intermediul unui limbaj simbolic. Oamenii operează adesea într-o manieră aproximativă, acordându-și o marjă de eroare. Aceeași posibilitate ar trebui să o aibă și sistemele inteligente, dar acestea beneficiază numai de cunoștințele explicite care le-au fost furnizate. Agenții umani se bazează foarte mult pe experiență. Sistemele trebuie să beneficieze și ele de formalizarea experienței).

Așa cum s-a mai amintit, comportamentul intelligent presupune cunoaștere. O ipoteză interesantă este aceea a lui Matt Ginsberg, care afirmă că *acțiunea inteligentă* poate fi redusă la *căutare*. Deși inteligența în mod cert nu coincide și nu poate fi redusă la căutare, aceasta este o ipoteză atractivă, întrucât: căutarea poate fi ușor înțeleasă, este ușor de

automatizat, ușor de gestionat etc. De asemenea, cunoștințele formalizate sunt ușor de utilizat în căutare.

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

- a găsi un drum în graf de la un nod inițial la un nod-scop (cu alte cuvinte, de la o situație inițială la una finală);
- a găsi un nod-scop.

Exemplul clasic de problemă de căutare care va fi prezentat în cadrul acestui curs este acela al jocurilor. Este însă interesant de analizat și alte aplicații ale căutării, cum ar fi analiza sintactică a unei propoziții, privită ca problemă de căutare esențială în subdomeniul procesării limbajului natural.

1.4. Un exemplu de problemă tipică în inteligența artificială

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

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

- **Observație:** În cazul unei astfel de probleme este mai greu de precizat care este răspunsul corect. Oferim, în acest sens, următorul exemplu⁸:

Exemplul nr. 1

Presupunem că textul reprezentând input-ul este dat de unica propoziție

“Russia massed troops on the Czech border.”

Corespunzător unui astfel de input este posibilă existența ambelor dialoguri care urmează. Aceste dialoguri au reprezentat efectiv output-ul programului numit POLITICS (Carbonell, 1980):

Dialogul nr. 1

Q: Why did Russia do this?

A: Because Russia thought that it could take political control of Czechoslovakia by sending troops.

Q: What should the United States do?

A: The United States should intervene militarily.

Dialogul nr. 2

Q: Why did Russia do this?

A: Because Russia wanted to increase its political influence over Czechoslovakia.

Q: What should the United States do?

A: The United States should denounce the Russian action in the United Nations.

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

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

O problemă din aceeași categorie, care utilizează în rezolvare tehnici specifice inteligenței artificiale, este cea prezentată în

Exemplul nr.2

Fie textul de intrare:

“Maria s-a dus să cumpere un palton nou. A găsit unul roșu care i-a plăcut mult. Când l-a adus acasă, a descoperit că merge perfect cu rochia ei preferată.”

Dorim să stabilim un algoritm prin care programul să răspundă următoarelor întrebări:

I1: *Ce s-a dus să cumpere Maria?*

I2: *Ce a găsit Maria care i-a plăcut?*

I3: *A cumpărat Maria ceva?*

Programul nr. 1

Acest program mai întâi convertește textul de intrare (reprezentând input-ul) într-o *formă structurată internă* care își propune să capteze sensul propozițiilor. De asemenea, convertește întrebările în același format. Programul găsește răspunsurile făcând corespondența dintre cele două forme structurate.

Structuri de date folosite în program:

- **CunoștințeRomana** - conține o descriere a cuvintelor, gramaticii și a interpretărilor semantice adecvate a unei submulțimi din limbă (în acest caz limba română), suficient de mari și de semnificative, care să poată acoperi textele de intrare pe care sistemul urmează să le primească. Aceste *cunoștințe* referitoare la limbă sunt folosite atât pentru a face corespondența dintre textele de intrare și o *formă internă, orientată către sens*, precum și dintre o asemenea formă și limba recunoscută de sistem. Problema stabilirii unei asemenea *baze de cunoștințe* pentru o limbă este foarte complicată și de ea se ocupă subdomeniul procesării limbajului natural.
- **InputText** - reprezintă textul de intrare în format caracter.
- **TextStructurat** - este o reprezentare structurată a conținutului textului de intrare. Această structură își propune să capteze cunoștințele cele mai importante conținute în text, independent de modul exact în care aceste cunoștințe au fost formulate în limba respectivă. Unele lucruri care nu au fost explicitate în acea limbă (spre exemplu la cine se referă pronumele din text) vor fi explicitate în această formă. *Reprezentarea*

cunoștințelor în acest mod este o chestiune esențială în toate programele inteligenței artificiale. Există trei familii importante de asemenea *sisteme de reprezentare a cunoștințelor*:

- reguli de producție (de forma “dacă x , atunci y ”);
- structuri de tip *slot-and-filler* (multe reprezentări folosesc perechi de tipul atribut-valoare, numite *sloturi*: engl. *slots*; atributele reprezintă denumirile sloturilor: *slot names*, iar valorile sunt cele care umplu aceste sloturi: *slot fillers*; vezi, de asemenea, p. 176 și Fig. 4.2).
- instrucțiuni în formă matematică (de tipul celor dintr-un sistem logic).

Sistemul de reprezentare a cunoștințelor ales de noi în acest exemplu este o structură de tip “*slot-and-filler*”. În acest sistem, de pildă, propozitia

“Ea a găsit unul roșu care i-a plăcut mult.”

se poate reprezenta în felul următor:

Eveniment1

<i>instantiere:</i>	<i>A gasi</i>
<i>temp:</i>	<i>Trecut</i>
<i>agent:</i>	<i>Maria</i>
<i>obiect:</i>	<i>Lucru1</i>

Lucru1

<i>instantiere:</i>	<i>Palton</i>
<i>culoare:</i>	<i>Rosu</i>

Eveniment2

<i>instantiere:</i>	<i>A place</i>
<i>temp:</i>	<i>Trecut</i>
<i>element modifier:</i>	<i>Mult</i>
<i>obiect:</i>	<i>Lucru1</i>

Aceasta este o descriere simplificată a conținutului propoziției. Ea nu este foarte explicită cu privire la relațiile temporale (timpul a fost marcat doar ca fiind "trecut"). Una dintre ideile cheie ale unei astfel de reprezentări este aceea că sensul entităților din reprezentare derivă din legătura, conexiunile lor cu alte entități. De aceea, în reprezentare există și alte entități, corespunzătoare unor concepte pe care programul le cunoștea înapoi de a citi această propoziție - Maria, palton (conceptul general de *palton*).

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

Algoritmul:

Convertește **InputText** în formă structurată utilizând cunoștințele conținute în **CunostinteRomana**. Această operație s-ar putea să presupună luarea în considerație a *diferite potențiale structuri*, din variate motive, spre exemplu datorită faptului că multe cuvinte dintr-o limbă pot fi ambigue. De asemenea, anumite structuri gramaticale pot fi ambigue, în procesul comunicării pronumele pot avea diferiți antecedenți etc.

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

1. Se convertește întrebarea în formă structurată, folosind din nou cunoștințele conținute în structura de date **CunostinteRomana**. Se folosește un marcator special în cadrul structurii pentru a indica acea

parte a structurii care ar trebui întoarsă ca răspuns. Acest marcator va corespunde adesea cu ocurența unui cuvânt care indică o întrebare în cadrul propoziției (cuvânt de tipul *cine* sau *ce*). Felul exact în care se efectuează această marcăre depinde de forma aleasă pentru a reprezenta **TextStructurat**. Atunci când, în reprezentare, se folosește o structură de tipul “slot-and-filler”, ca aici, un marcator special poate fi plasat în unul sau mai multe sloturi. Dacă se folosește un sistem logic, marcatorii vor apărea ca variabile în cadrul formulelor logice care reprezintă întrebarea.

2. Se opune această formă structurată celei din **TextStructurat**. (Se împerechează cele două forme).

3. Se întorc ca răspuns acele părți ale textului care se potrivesc cu segmentul cerut din întrebare.

□

Iată răspunsurile obținute în cazul nostru, adică atunci când sistemul i se pun cele trei întrebări luate în considerație în *Exemplul nr. 2*:

I1: Răspunsul direct la întrebare este “*un palton nou*”.

I2: Răspunsul direct la întrebare este “*un palton roșu*”.

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

▪ Observații:

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

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

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

“Maria s-a întrebat spre vânzătoare. Ea a întrebat unde se află raionul de jucării.”

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

Programul nr. 2

(care exemplifică ce se înțelege prin tehnică a inteligenței artificiale)

Acest program convertește textul de intrare într-o formă structurată care conține sensurile propozițiilor din text și apoi combină această formă cu alte forme structurate, care descriu cunoștințe prealabile despre obiectele și situațiile la care se referă textul. Programul răspunde la întrebări folosind această *structură de cunoștințe extinsă*.

Structuri de date folosite în program:

- **Modelul Lumii** - este o reprezentare structurată a cunoștințelor asupra lumii (universului) din fundal. Această structură conține cunoștințe despre obiecte, acțiuni și situații care sunt descrise în textul de intrare. Ea este folosită pentru a construi **TextIntegritat** plecând de la textul de intrare. Figura care urmează ilustrează un exemplu de structură ce reprezintă cunoștințele sistemului legate de mersul la cumpărături. Asemenea cunoștințe memorate legate de evenimente stereotipe formează un *script*⁹. Fig. 1.1 ne arată un script pentru cumpărături:

⁹ Un *script* este o structură care descrie o secvență stereotipă de evenimente, într-un anumit context. Astfel, scripturile își propun să reprezinte cunoștințe despre succesiuni comune de evenimente (i.e. evenimente care nu intervin în mod izolat). Informații despre succesiuni tipice de evenimente, aşa cum sunt ele reprezentate în scripturi, pot fi folosite în interpretarea unei succesiuni particolare de evenimente, care este observată la un moment dat.

Un script constă dintr-o mulțime de *sloturi*. Asociată fiecărui slot se găsește o informație despre tipurile de valori pe care le poate conține, precum și o valoare implicită, care poate fi folosită atunci când nu este disponibilă nici o altă informație.

Scripturile sunt utile deoarece, în lumea reală, există anumite *șabloane* după care evenimentele se produc. Aceste șabloane se nasc datorită *relațiilor cauzale* care intervin între evenimente. Evenimentele descrise într-un script formează un uriaș *lanț cauzal*. Începutul lanțului este mulțimea condițiilor de intrare care fac posibilă producerea primelor evenimente ale scriptului. Sfârșitul lanțului este mulțimea rezultatelor care pot

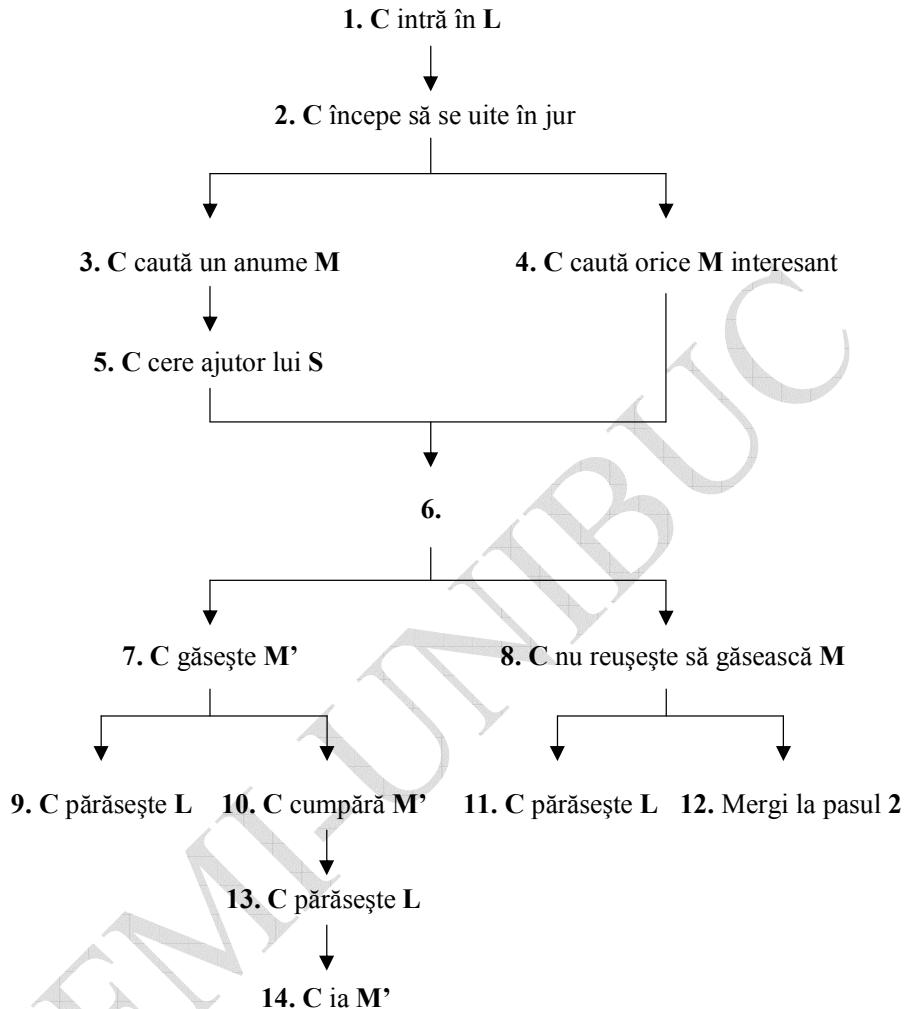


Fig. 1.1

face posibilă producerea unor evenimente sau a unor lanțuri de evenimente ulterioare. În cadrul lanțului, evenimentele sunt conectate atât la evenimente anterioare, care le-au făcut posibile, cât și la evenimente ulterioare, a căror producere o înlesnesc.

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

Notăția folosită aici este simplificată în raport cu cea din literatura existentă, de dragul simplității și al înțelegerii cât mai clare a noțiunii de script. Astfel, notația cu “prim” descrie un obiect de același tip ca cea fără “prim”, cele două simboluri (dintre care unul cu “prim”) putându-se referi sau nu la obiecte identice. În exemplul nostru, de pildă, **M** este un palton, iar **M'** un palton roșu. Ramificațiile din figură descriu drumuri alternative prin script.

- **CunostinteLimba** - la fel ca în Programul nr. 1.
- **InputText** - textul de intrare în format caracter.
- **TextIntegrat** - o reprezentare structurată a cunoștințelor conținute în textul de intrare (similară cu descrierea structurată din Programul nr.1), dar care acum este combinată cu alte cunoștințe, înrudite, din fundal.
- **InputIntrebare** - întrebarea care constituie o parte a input-ului, în format caracter.
- **StructIntrebare** - o reprezentare structurată a întrebării utilizatorului.

Algoritmul:

Convertește **InputText** în formă structurată folosind atât cunoștințele conținute în **CunostinteLimba**, cât și pe cele conținute în **ModelulLumii**. (Numărul structurilor posibile va fi mai mare acum decât în programul anterior, pentru că sunt folosite mult mai multe cunoștințe. Unele posibilități vor fi însă eliminate prin filtrarea lor cu ajutorul cunoștințelor suplimentare).

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

1. Convertește întrebarea în formă structurată, ca și în Programul 1, dar folosește, atunci când este necesar, **Modelul Lumii**, pentru a rezolva diversele ambiguități care pot interveni.
2. Împerechează această formă cu cea din **TextIntegrat**.
3. Întoarce ca răspuns acele părți din text care se potrivesc cu segmentul de întrebare cerut.

□

Iată răspunsurile obținute acum în cazul nostru, adică atunci când sistemului î se pun cele trei întrebări luate în considerație în *Exemplul nr. 2*:

I1: La fel ca în Programul nr.1.

I2: La fel ca în Programul nr.1.

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

“*Ea a cumpărat un palton roșu.*”

▪ **Observații:**

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

2. Chiar și tehniciile exploatației în acest program nu sunt adecvate pentru a răspunde tuturor întrebărilor. Cel mai important lucru care îi lipsește acestui program este un *mecanism general de inferență*, care să poată fi folosit atunci când răspunsul cerut nu este conținut în mod explicit nici chiar în **TextIntegrat**, dar decurge în mod logic din cunoștințele incluse acolo. Spre exemplu, fiind dat textul

“Duminică dimineața Maria s-a dus la cumpărături. Fratele ei a încercat să o sună atunci, dar nu a putut să-o găsească.”

ar trebui să se poată răspunde la întrebarea

“De ce nu a putut fratele Mariei să o găsească?”

prin răspunsul

“Pentru că ea nu era acasă.”

Dar, pentru a obține un asemenea răspuns, este necesar să se cunoască faptul că cineva nu poate fi în două locuri în același timp și apoi să se folosească acest fapt pentru a trage concluzia că Maria nu ar fi putut fi acasă deoarece era la cumpărături.

Problema inferenței a fost în mod temporar evitată prin construirea lui **TextIntegrat**, care avea unele inferențe incluse în el. Dar acestea nu sunt suficiente și, cum nu este posibil să se anticipateze toate inferențele normale și posibile, anumite subdomenii ale inteligenței artificiale se ocupă de furnizarea unui *mecanism general de inferență*.

- **Concluzie:**

*Programul nr. 2 reprezintă un exemplu pentru a susține faptul că o procedură eficientă de “răspunsuri la întrebări” trebuie să se bazeze în mod solid pe *cunoștințe* și pe *exploatarea computațională a acelor cunoștințe*. De altfel, însuși scopul tehniciilor inteligenței artificiale este acela de a putea susține o folosire eficientă a cunoștințelor.*

CAPITOLUL 2

TEHNICI DE CĂUTARE

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

În cazul unui *agent uman*, acesta are anumite organe (ochi, urechi etc.) pe post de senzori și anumite părți ale corpului (mâini, picioare, gură etc.) pe post de efectori. Există și agenți roboți, caz în care, spre exemplu, efectorii sunt înlocuiți cu diferite motoare. În general, prin *efector* se înțelege un mușchi, o glandă etc. capabil să răspundă unui stimul, în special unui impuls nervos.

Așa cum se afirmă în [9], menirea inteligenței artificiale este să proiecteze un *program agent*, și anume o funcție care implementează un agent ce realizează transformarea de la percepții la acțiuni.

Agenții inteligenți au, în mare, același schelet, ceea ce înseamnă că acceptă percepții din mediul înconjurător și generează acțiuni. Cele mai facile *programe agent* au o formă foarte simplă. Fiecare astfel de program folosește niște *structuri de date interne* care vor fi actualizate pe măsură ce intervin noi percepții. Asupra acestor structuri de date se operează prin *procedurile de luare de decizii* ale agentului. Aceste proceduri generează alegerea unei acțiuni, alegere care este transmisă arhitecturii (hardware-ului) pentru a fi executată.

În cele ce urmează, vom arăta cum un agent poate acționa prin stabilirea unor țeluri sau *scopuri* și vom lua în considerație secvența de acțiuni care ar putea duce la îndeplinirea acestor scopuri.

Conform [9], un scop și o mulțime de modalități de atingere a acestuia formează o *problemă*, iar explorarea a ceea ce pot face aceste modalități de atingere a scopului constituie procesul de *căutare*.

În continuare, urmând calea din [9], ne propunem să descriem un anumit tip de *agent bazat pe scopuri*, numit *agent rezolvator de probleme*. Agenții rezolvatori de probleme decid ce trebuie făcut prin găsirea unor secvențe de acțiuni care conduc la niște *stări convenabile*. Agentul trebuie să-și formuleze o viziune adecvată asupra problemei pe care o va rezolva. Tipul de problemă care rezultă din procesul de formulare va depinde de cunoștințele aflate la îndemâna agentului. În principal, este important dacă el cunoaște starea curentă și rezultatele acțiunilor.

Agenții cu care vom lucra vor adopta un *scop* și vor urmări *satisfacerea* lui.

2.1. Rezolvarea problemelor prin intermediul căutării

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

Vom considera un *scop* ca fiind o mulțime de stări ale universului, și anume acele stări în care scopul este satisfăcut. *Acțiunile* pot fi privite ca generând tranziții între stări ale universului. Agentul va trebui să afle care acțiuni îl vor conduce la o stare în care scopul este

satisfăcut. Înainte de a face asta el trebuie să decidă ce tipuri de acțiuni și de stări să ia în considerație.

Procesul decizional cu privire la acțiunile și stările ce trebuie luate în considerație reprezintă *formularea problemei*. Formularea problemei urmează după formularea scopului.

Un agent care va avea la dispoziție mai multe opțiuni imediate va decide ce să facă examinând mai întâi diferite *secvențe de acțiuni* posibile, care conduc la stări de valori necunoscute, urmând ca, în urma acestei examinări, să o aleagă pe cea mai bună. Procesul de examinare a unei astfel de succesiuni de acțiuni se numește *căutare*. Un *algoritm de căutare* primește ca *input* o *problemă* și întoarce ca *output* o *soluție* sub forma unei succesiuni de acțiuni.

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

Iată o funcție, preluată de noi din [9], care implementează un *agent rezolvator de probleme* simplu:

```
function AGENT_REZOLVATOR_DE_PROBLEME(p) return o acțiune
```

inputs: *p*, o percepție

static: *s*, o secvență de acțiuni, inițial vidă

```

stare, o descriere a stării curente a lumii (universului)
g, un scop, inițial vid
problemă, o formulare a unei probleme

stare  $\leftarrow$  ACTUALIZEAZĂ_STARE(stare, p)
if s este vid then
    g  $\leftarrow$  FORMULEAZĂ_SCOP(stare)
    problemă  $\leftarrow$  FORMULEAZĂ_PROBLEMĂ(stare, g)
    s  $\leftarrow$  CAUTĂ(problemă)
    acțiune  $\leftarrow$  RECOMANDARE(s, stare)
    s  $\leftarrow$  REST(s, stare)
return acțiune

```

În cele ce urmează nu vom discuta amănunte despre funcțiile ACTUALIZEAZĂ_STARE și FORMULEAZĂ_SCOP, ci ne vom concentra asupra *procesului de formulare a problemei* și ne vom ocupa de diferite versiuni ale funcției CAUTĂ. Funcția RECOMANDARE nu face decât să preia prima dintre acțiunile secvenței de acțiuni, iar funcția REST întoarce restul.

2.1.1. Tipuri de probleme

Există *patru tipuri de probleme* [9], fundamental diferite: probleme cu o unică stare, probleme cu stări multiple, probleme de contingență și probleme de explorare:

1. Atunci când senzorii agentului îi furnizează acestuia suficientă informație pentru ca el să își dea seama exact în ce stare se află (spunem că universul îi este accesibil) și atunci când el știe în mod exact ce

realizează fiecare dintre acțiunile sale, el poate calcula în mod exact în ce stare se va afla după orice secvență de acțiuni. Acesta este cazul cel mai simplu, numit *problemă cu o singură stare*.

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

3. Rezolvarea anumitor probleme (în special în lumea fizică, reală) necesită acțiunea senzorilor chiar în faza de execuție. Atunci când nu există o secvență de acțiuni fixată care să garanteze o soluție a problemei, agentul va trebui să calculeze un întreg *arbore de acțiuni*, nu numai o unică secvență de acțiuni. În general, fiecare ramură a arborelui se ocupă de o posibilă *contingență* ce ar putea să apară. Din acest motiv, numim acest gen de probleme - *probleme de contingență*¹⁰. Multe probleme din lumea reală sunt probleme de contingență deoarece *predicția exactă este imposibilă*. (Mersul pe stradă sau condusul mașinii reprezintă, în lumea reală, probleme de contingență). Rezolvarea problemelor de contingență necesită algoritmi extrem de complecsi. De

¹⁰ Aici *contingență* are sensul de *întâmplare*.

aceea, în cadrul acestui curs, ne vom ocupa numai de problemele cu o singură stare și cu stări multiple, care pot fi rezolvate utilizând tehnici de căutare similară. Vom lua deci în considerație numai cazuri în care *o soluție garantată constă dintr-o unică secvență de acțiuni*.

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

2.1.2. Probleme și soluții corect definite

➤ *Probleme cu o singură stare*

Elementele de bază ale definirii unei probleme sunt *stările* și *acțiunile*. Pentru a descrie stările și acțiunile, din punct de vedere formal, este nevoie de următoarele elemente [9]:

- *Starea initială* în care agentul știe că se află.
- Mulțimea acțiunilor posibile disponibile agentului. Termenul de *operator* este folosit pentru a desemna descrierea unei acțiuni, prin specificarea stării în care se va ajunge ca urmare a îndeplinirii acțiunii

respective, atunci când ne aflăm într-o anumită stare. (O formulare alternativă folosește o *funcție succesor* S . Fiind dată o anumită stare x , $S(x)$ întoarce mulțimea stărilor în care se poate ajunge din x , printr-o unică acțiune).

- *Spațiul de stări* al unei probleme reprezintă mulțimea tuturor stărilor în care se poate ajunge plecând din starea inițială, prin intermediul oricărei secvențe de acțiuni.
- Un *drum* în spațiul de stări este orice secvență de acțiuni care conduce de la o stare la alta.
- *Testul scop* este testul pe care un agent îl poate aplica unei singure descrieri de stare pentru a determina dacă ea este o *stare de tip scop*, adică o stare în care scopul este atins (sau realizat). Uneori există o mulțime explicită de stări scop posibile și testul efectuat nu face decât să verifice dacă s-a ajuns în una dintre ele. Alteori, scopul este specificat printr-o proprietate abstractă și nu prin enumerarea unei mulțimi de stări. De exemplu, în șah, scopul este să se ajungă la o stare numită “șah mat”, în care regele adversarului poate fi capturat la următoarea mutare, orice ar face adversarul. S-ar putea întâmpla ca o soluție să fie preferabilă alteia, chiar dacă amândouă ating scopul. Spre exemplu, pot fi preferate drumuri cu mai puține acțiuni sau cu acțiuni mai puțin costisitoare.
- *Funcția de cost a unui drum* este o funcție care atribuie un cost unui drum. Ea este adeseori notată prin g . Vom considera costul unui drum ca fiind suma costurilor acțiunilor individuale care compun drumul.

Împreună *starea inițială*, *mulțimea operatorilor*, *testul scop* și *funcția de cost a unui drum* definesc o **problemă**.

Tipul de dată prin care putem reprezenta o problemă este deci următorul [9]:

tip_de_dată PROBLEMĂ

componente: STARE_INIȚIALĂ, OPERATORI, TEST_SCOP,
FUNCȚIE_COST_DRUM

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

➤ **Probleme cu stări multiple**

Pentru definirea unei astfel de probleme trebuie specificate:

- o *mulțime* de stări inițiale;
- o mulțime de operatori care indică, în cazul fiecărei acțiuni, mulțimea stărilor în care se ajunge plecând de la orice stare dată;
- un test scop (la fel ca la problema cu o singură stare);
- funcția de cost a unui drum (la fel ca la problema cu o singură stare).

Un operator se aplică unei mulțimi de stări prin reunirea rezultatelor aplicării operatorului fiecărei stări din mulțime. Aici un **drum** leagă *mulțimi de stări*, iar o *soluție* este un drum care conduce la o *mulțime de stări*, dintre care *toate sunt stări scop*. Spațiul de stări este aici înlocuit de *spațiul mulțimii de stări*.

▪ Un exemplu. Problema misionarilor și canibalilor

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

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

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

Pentru a formaliza problema, la primul pas ea trebuie abstractizată, prin îndepărțarea tuturor detaliilor care nu au relevanță asupra soluției.

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

numărul de canibali. Prin urmare, numai capetele unei traversări sunt importante.

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

Toate aceste considerații duc la următoarea definiție formală a problemei:

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

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

□

Un aspect deosebit de important îl constituie măsurarea performanței în rezolvarea de probleme. Astfel, eficacitatea unei căutări

poate fi măsurată în cel puțin trei moduri, și anume conform următoarelor criterii de bază:

- dacă se găsește o soluție;
- dacă s-a găsit o soluție bună (adică o soluție cu un cost scăzut al drumului);
- care este costul căutării¹¹ asociat timpului calculator și memoriei necesare pentru a găsi o soluție.

2.1.3. Căutarea soluțiilor și generarea secvențelor de acțiuni

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

Procesul de căutare construiește un *arbore de căutare*, a cărui rădăcină este un *nod de căutare* corespunzând stării inițiale. La fiecare pas, algoritmul de căutare alege un nod-frunză pentru a-l extinde. *Algoritmul de căutare general* este următorul [9]:

¹¹ Costul *total* al unei căutări se definește ca fiind suma dintre costul drumului corespunzător și costul respectivei căutări.

```

function CĂUTARE_GENERALĂ(problemă, strategie)
    return soluție sau eșec
initializează arborele de căutare folosind starea inițială a lui problemă
ciclu do
    if nu există candidați pentru extindere
        then return eșec
    alege un nod-frunză pentru extindere conform lui strategie
    if nodul conține o stare scop
        then return soluția corespunzătoare
    else extinde nodul și adaugă nodurile rezultate arborelui de
        căutare
end

```

□

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

2.1.4. Structuri de date pentru arbori de căutare

Există numeroase moduri de a reprezenta nodurile. În general, se consideră că un nod este o structură de date cu cinci componente:

- starea din spațiul de stări căreia îi corespunde nodul;
- nodul din arborele de căutare care a generat acest nod (nodul părinte);
- operatorul care a fost aplicat pentru a se genera nodul;

- numărul de noduri aflate pe drumul de la rădăcină la acest nod (*adâncimea* nodului);
- costul drumului de la starea inițială la acest nod.

Este importantă *distincția între noduri și stări*. Astfel, un nod este o structură de date folosită pentru a reprezenta arborele de căutare corespunzător unei anumite realizări a unei probleme, generată de un anumit algoritm. O stare reprezintă însă o configurație a lumii înconjurătoare. De aceea, nodurile au adâncimi și părinți, iar stările nu le au. Mai mult, este posibil ca două noduri diferite să conțină aceeași stare, dacă acea stare este generată prin intermediul a două secvențe de acțiuni diferite. Funcția EXTINDERE va fi responsabilă pentru determinarea fiecărei componente a nodului pe care îl generează.

Este necesară, de asemenea, reprezentarea colecției de noduri care așteaptă pentru a fi extinse. Această colecție de noduri poartă denumirea de *frontieră*. Cea mai simplă reprezentare ar fi aceea a unei mulțimi de noduri, iar strategia de căutare ar fi o funcție care selectează, din această mulțime, următorul nod ce trebuie extins. Deși din punct de vedere conceptual această cale este una directă, din punct de vedere computațional ea poate fi foarte scumpă, pentru că funcția strategie ar trebui să se „uite” la fiecare element al mulțimii pentru a-l alege pe cel mai bun. De aceea, *vom presupune că această colecție de noduri este implementată ca o coadă*.

Operațiile definite pe o coadă vor fi următoarele:

- CREEAZĂ_COADA(*Elemente*) - creează o coadă cu elementele date;
- GOL?(*Coadă*) - întoarce „true” numai dacă nu mai există elemente în coadă;

- ÎNLĂTURĂ_DIN_FAȚĂ(*Coada*) - înălțură elementul din față al cozii și îl transmite înapoi (return);
- COADA_FN(*Elemente, Coada*) - inserează o mulțime de elemente în coadă. Diferite feluri de funcții de acest tip produc algoritmi de căutare diferenți.

Cu aceste definiții, putem da o descriere mai formală a *algoritmului general de căutare*, conform [9]:

```
function CĂUTARE_GENERALĂ(problemă,COADA_FN) return
    o soluție sau eșec
    noduri  $\leftarrow$  CREEAZĂ_COADA(CREEAZĂ_NOD(STARE_INIȚIALĂ
        [problemă]))
    ciclu do
        if noduri este vidă
            then return eșec
        nod  $\leftarrow$  ÎNLĂTURĂ_DIN_FAȚĂ(noduri)
        if TEST_SCOP[problemă] aplicat lui STARE(nod) are succes
            then return nod
        noduri  $\leftarrow$  COADA_FN(noduri, EXTINDERE(nod, OPERATORI
            [problemă]))
    end
```

□

2.1.5. Evaluarea strategiilor de căutare

Strategiile de căutare se evaluatează conform următoarelor patru criterii:

- Completitudine: dacă, atunci când o soluție există, strategia dată garantează găsirea acesteia;
- Complexitate a timpului: durata de timp pentru găsirea unei soluții;
- Complexitate a spațiului: necesitățile de memorie pentru efectuarea căutării;
- Optimalitate: atunci când există mai multe soluții, strategia dată să o găsească pe cea mai de calitate dintre ele.

2.2. Căutarea neinformată

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

Să considerăm, de pildă, problema găsirii unui drum de la Arad la București, având în față o hartă. De la starea inițială, Arad, există trei acțiuni care conduc la trei noi stări: Sibiu, Timișoara și Zerind. O căutare neinformată nu are nici o preferință între cele trei variante. Un agent mai inteligent va observa însă că scopul, București, se află la sud-est de Arad și că numai *Sibiu* este în această direcție, care reprezintă, probabil, cea

mai bună alegeră. Strategiile care folosesc asemenea considerații se numesc *strategii de căutare informată* sau *strategii de căutare euristică*.

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

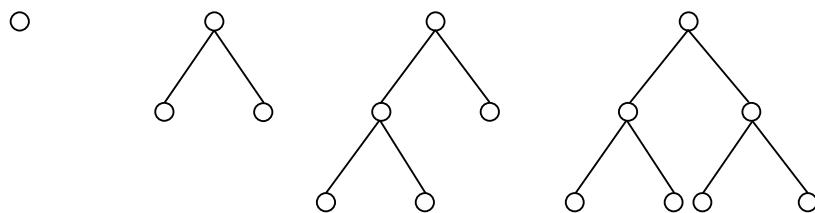
În cele ce urmează, vom aborda mai multe strategii de căutare neinformată, acestea deosebindu-se între ele prin **ordinea** în care nodurile sunt extinse.

2.2.1. Căutarea de tip breadth-first

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

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

Strategia breadth-first este foarte *sistemerică* deoarece ia în considerație toate drumurile de lungime 1, apoi pe cele de lungime 2 etc., așa cum se arată în Fig. 2.1. Dacă există o soluție, este sigur că această metodă o va găsi, iar dacă există mai multe soluții, căutarea de tip breadth-first va găsi întotdeauna mai întâi *soluția cel mai puțin adâncă*.

**Fig. 2.1**

În termenii celor patru criterii de evaluare a strategiilor de căutare, cea de tip breadth-first este *completă* și este *optimă* cu condiția ca *costul drumului să fie o funcție descrescătoare de adâncimea nodului*. Această condiție este de obicei satisfăcută numai atunci când toți operatorii au același cost.

2.2.1.1. Algoritmul de căutare breadth-first

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

Algoritmul 2.1

1. Creează o variabilă numită LISTA_NODURI și setează-o la starea inițială.
2. Până când este găsită o stare-scop sau până când LISTA_NODURI devine vidă, execută:
 - 2.1. Înlătură primul element din LISTA_NODURI și numește-l E. Dacă LISTA_NODURI a fost vidă, STOP.
 - 2.2. Pentru fiecare mod în care fiecare regulă se potrivește cu

starea descrisă în E, execută:

- 2.2.1.** Aplică regula pentru a genera o nouă stare.
- 2.2.2.** Dacă noua stare este o stare-scop, întoarce această stare și STOP.
- 2.2.3.** Altfel, adaugă noua stare la sfârșitul lui LISTA_NODURI.

2.2.1.2. Implementare în Prolog

Pentru a programa în Prolog strategia de căutare breadth-first, trebuie menținută în memorie *o mulțime de noduri candidate alternative*. Această mulțime de candidați reprezintă marginea de jos a arborelui de căutare, aflată în continuă creștere (frontiera). Totuși, această mulțime de noduri nu este suficientă dacă se dorește și extragerea unui drum-soluție în urma procesului de căutare. Prin urmare, în loc de a menține o mulțime de noduri candidate, vom menține *o mulțime de drumuri candidate*¹².

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

Căutarea este începută cu o mulțime de candidați având un singur element:

`[[NodInitial]].`

¹² Implementarea în Prolog a strategiei breadth-first propusă aici este preluată din [1].

Fiind dată o mulțime de drumuri candidate, căutarea de tip breadth-first se desfășoară astfel:

- dacă primul drum conține un nod-scop pe post de cap, atunci acesta este o soluție a problemei;
- altfel, înălătură primul drum din mulțimea de candidați și generează toate extensiile de un pas ale acestui drum, adăugând această mulțime de extensii la sfârșitul mulțimii de candidați. Execută apoi căutarea de tip breadth-first asupra mulțimii actualizate.

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

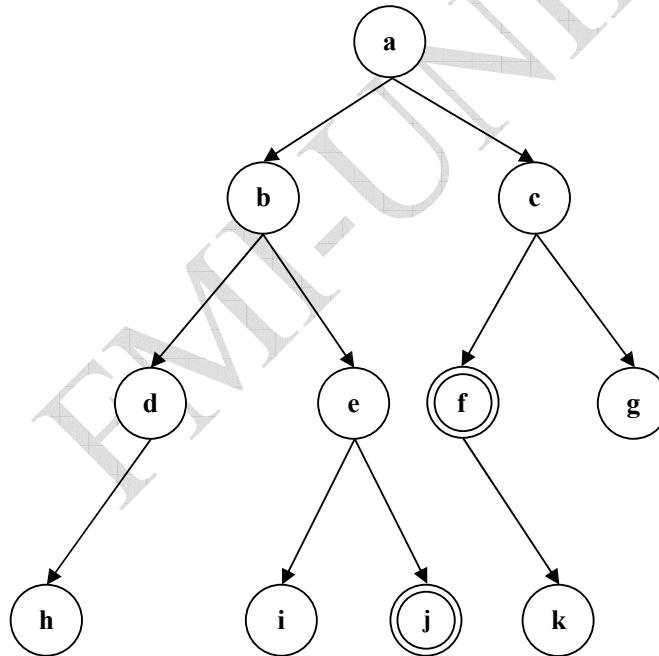


Fig. 2.2

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

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

(1) Se începe cu mulțimea de candidați inițială:

[[a]]

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

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

(Se observă reprezentarea drumurilor în ordine inversă).

(3) Se înlătură primul drum candidat, [b, a], din mulțime și se generează extensiile ale acestui drum:

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

Se adaugă lista extensiilor la sfârșitul mulțimii de candidați:

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

(4) Se înlătură [c, a] și se adaugă extensiile sale la sfârșitul mulțimii de candidați, rezultând următoarea mulțime de drumuri:

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

La următorii pași, [d, b, a] și [e, b, a] sunt extinse, iar mulțimea de candidați modificată devine:

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

Acum procesul de căutare întâlneste [f, c, a], care conține un nod scop, *f*.

Prin urmare, acest drum este returnat ca soluție.

Programul Prolog care implementează acest proces de căutare va reprezenta mulțimile de noduri ca pe niște liste, efectuând și un test care să prevină generarea unor drumuri ciclice. În cadrul acestui program,

toate extensiile de un pas vor fi generate prin utilizarea procedurii încorporate **bagof**:

```

rezolva_b(Start,Sol) :-breadthfirst([[Start]],Sol).
breadthfirst([[Nod|Drum] | _], [Nod|Drum]) :-scop(Nod) .
breadthfirst([Drum|Drumuri],Sol) :-  

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

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

```

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

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

Predicatul **extinde(Drum,DrumuriNoi)** este adevărat dacă prin extinderea mulțimii de noduri Drum obținem mulțimea numită DrumuriNoi, el generând mulțimea tuturor extensiilor acestui drum.

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

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

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

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

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

Programul 2.1

```

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

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

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

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

```

```

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

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

extinde(_, []) .

```

Interogarea Prologului se face astfel:

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

Răspunsul Prologului va fi:

```

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

```

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

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

2.2.1.3. Timpul și memoria cerute de strategia breadth-first

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

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

Prin urmare, algoritmul are o *complexitate exponențială* de $O(b^d)$. Complexitatea spațiului este aceeași cu complexitatea timpului deoarece toate nodurile frunză ale arborelui trebuie să fie menținute în memorie în același timp.

Iată câteva exemple de execuții cu factor de ramificare $b=10$, preluate de noi din [9]:

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

Se observă că cerințele de memorie sunt o problemă mai mare, pentru căutarea de tip breadth-first, decât timpul de execuție. (Este posibil să

putem aştepta 18 minute pentru a se efectua o căutare de adâncime 6, dar să nu dispunem de 111 megabytes de memorie). La rândul lor, cerințele de timp sunt majore. (Dacă problema are o soluție la adâncimea 12, o căutare neinformată de acest tip o găsește în 35 de ani). În general, problemele de căutare de *complexitate exponențială* nu pot fi rezolvate decât pe mici porțiuni.

2.2.1.4. Un exemplu. Problema misionarilor și canibalilor

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

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

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

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

respectiv la numărul de canibali și de misionari aflați inițial pe malul estic al râului.

Programul 2.2

```
%Predicatelor specifică problemei misionarilor și
%canibalilor

%Definim relația de succesiune.

%Predicatul st(S1,S2) este adevarat dacă există un
%arc între S1 și S2 în spațiul starilor.

s(st(B,NMB,NCB,NMis,NCan,M,N),st(B1,NMisB1,NCanB1,
NMis1,NCan1,M,N)):-  

    maluri_opuse(B,B1),  

    intre(K1,0,M),  

    %determinăm K1, numărul de misionari care trec  

    %cu barca  

    intre(K2,0,M),  

    %determinăm K2, numărul de canibali care trec  

    %cu barca  

    K1+K2=<M,K1+K2>0,  

    (K1=0 ; K1=\=0, K1>=K2),  

    NMB>=K1, NCB>=K2, NMisB1 is NMis+K1, NCanB1 is  

    NCan+K2,  

    NMis1 is NMB-K1, NCan1 is  

    NCB-K2, in_regula(NMisB1,NCanB1),  

    in_regula(NMis1,NCan1).

membru(X,[X|_]).  

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

```

intre(A,A,B) :- A=<B.

intre(X,A,B) :- A<B, A1 is A+1, intre(X,A1,B).

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

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

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

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

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

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

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

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

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

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

colectez(L) :- retract(elem(X)), !,

```

```

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

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

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

breadthfirst_dif([[Nod|Drum]|_]-_, [Nod|Drum]):-
    scop(Nod) . 

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

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

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

scrie(st(B,MX,CX,MY,CY,_,_)) :-
    write('Barca se afla pe malul de '),write(B),

```

```

write('.'),nl,maluri_opuse(B,B1),
(B=est, write('Pe malul de '),write(B),
write(' se afla '),write(MX),scrie_nr(MX,m),
write(' si '),write(CX),scrie_nr(CX,c),
write('.'),nl,write('Pe malul de '),write(B1),
write(' se afla '),write(MY),scrie_nr(MY,m),
write(' si '),write(CY),scrie_nr(CY,c);
B=vest, write('Pe malul de '),write(B1),
write(' se afla '),write(MY),scrie_nr(MY,m),
write(' si '),write(CY),scrie_nr(CY,c),
write('.'),nl,write('Pe malul de '),write(B),
write(' se afla '),write(MX),scrie_nr(MX,m),
write('si'),write(CX),scrie_nr(CX,c)),write('').

```

```

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

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

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

```

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

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

Răspunsul sistemului va fi **yes**

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

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

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

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

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

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

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

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

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

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

Barca se afla pe malul de vest.

Pe malul de est se afla 0 misionari si 1 canibal.

Pe malul de vest se afla 3 misionari si 2 canibali.

Barca se afla pe malul de est.

Pe malul de est se afla 1 misionar si 1 canibal.

Pe malul de vest se afla 2 misionari si 2 canibali.

Barca se afla pe malul de vest.

Pe malul de est se afla 0 misionari si 0 canibali.

Pe malul de vest se afla 3 misionari si 3 canibali.

Nr mutari: 11

2.2.2. Căutarea de tip depth-first

2.2.2.1. Prezentare generală și comparație cu strategia breadth-first

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

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

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

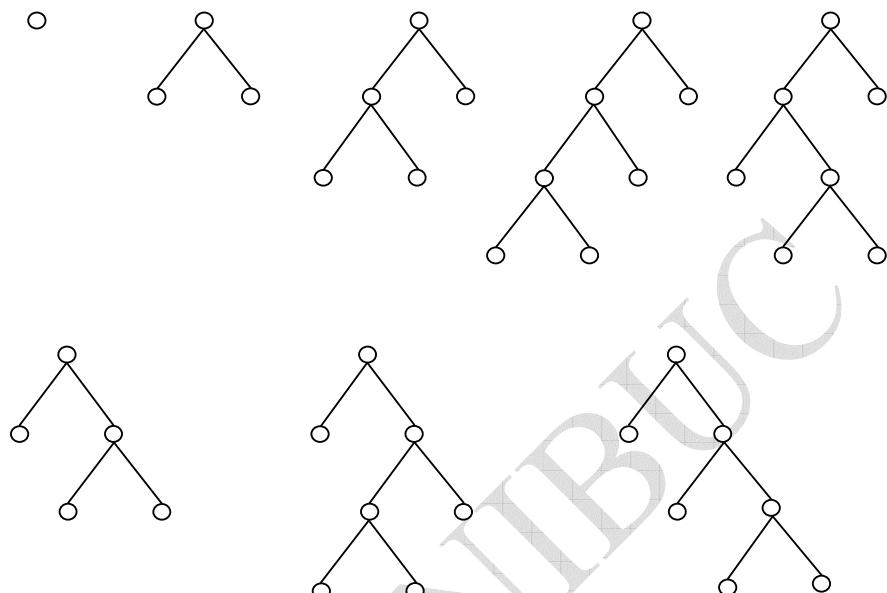


Fig. 2.3

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

Complexitatea de timp a strategiei depth-first este de $O(b^m)$.

Pentru probleme care au foarte multe soluții, căutarea de tip depth-first s-ar putea să fie mai rapidă decât cea de tip breadth-first, deoarece sunt șanse mari să fie găsită o soluție după ce se explorează numai o mică porțiune a întregului spațiu de căutare. Strategia breadth-first trebuie să investigheze toate drumurile de lungime $d-1$ înainte de a lua în

considerație pe oricare dintre drumurile de lungime d . Depth-first este de complexitate $O(b^m)$ și în cazul cel mai nefavorabil.

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

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

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

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

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

2.2.2.2. Implementare în Prolog

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

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

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

Aceasta se traduce în Prolog astfel¹³:

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

Interogarea Prologului se va face în felul următor:

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

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

Programul 2.3

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

¹³ Implementarea în Prolog a strategiei depth-first propusă aici este preluată din [1].

```

s(e,j).
s(c,f).
s(c,g).
s(f,k).

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

```

Interogarea Prologului se face astfel:

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

Răspunsul Prologului va fi:

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

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

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

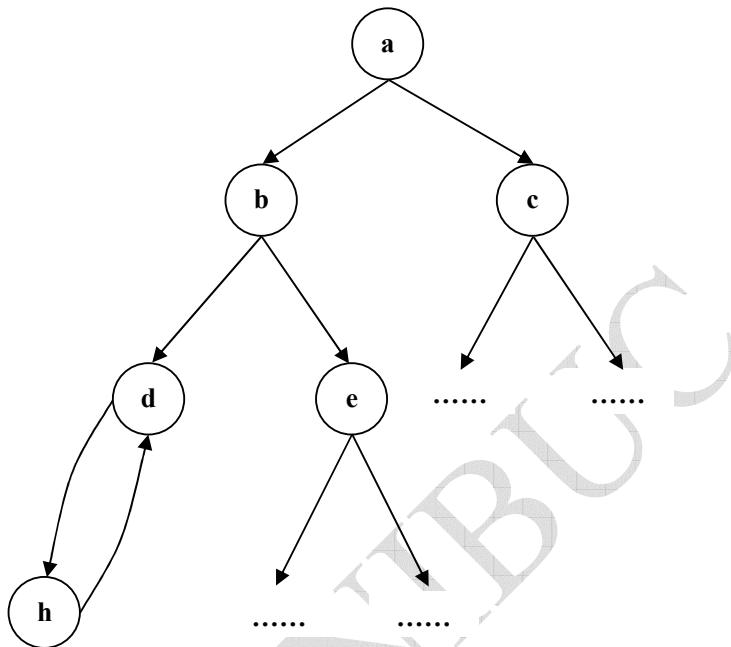


Fig. 2.4

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

depthfirst(Drum, Nod, Soluție).

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

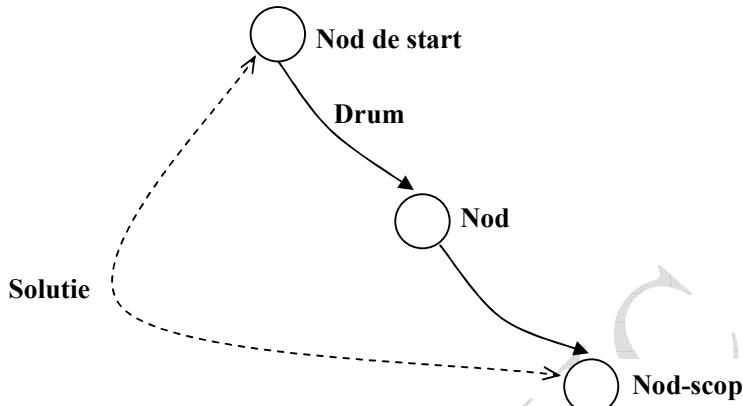


Fig.2.5

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

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

Predicatul corespunzător,

`depthfirst(Drum,Nod,Sol),`

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

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

```

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

depthfirst(Drum,Nod,[Nod|Drum]):-scop(Nod).

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

```

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

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

Programul 2.4

```

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

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

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

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

Interogarea Prologului se face astfel:

```
?- rezolva1_d(a,Sol).
```

Răspunsul Prologului va fi:

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

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

Așa cum se arată în [1], un program de acest tip nu va lucra totuși corect atunci când *spațiul stărilor este infinit*. Într-un astfel de spațiu algoritmul depth-first poate omite un nod-scop deplasându-se de-a lungul unei ramuri infinite a grafului. Programul poate atunci să exploreze în mod nedefinit această parte infinită a spațiului, fără a se apropiă vreodată de un scop. Pentru a evita astfel de *ramuri aciclice infinite*, procedura de căutare depth-first de bază poate fi, în continuare, rafinată, prin *limitarea adâncimii căutării*. În acest caz, procedura de căutare depth-first va avea următoarele argumente:

depthfirst1(Nod, Soluție, Maxdepth),

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

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

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

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

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

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

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

2.2.3. Căutarea în adâncime iterativă

2.2.3.1. Prezentare generală

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

- este optimă și completă ca și căutarea breadth-first;
- consumă numai cantitatea mică de memorie necesară căutării depth-first (cerința de memorie este liniară).

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

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

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

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

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

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

$$\begin{aligned} N_{id} &= \sum_{j=0}^d \frac{b^{j+1} - 1}{b - 1} = \frac{1}{b - 1} \sum_{j=0}^d [b^{j+1} - 1] = \frac{1}{b - 1} [\sum_{j=0}^d b^{j+1} - \sum_{j=0}^d 1] \\ &= \frac{1}{b - 1} [b(\sum_{j=0}^d b^j) - \sum_{j=0}^d 1] = \frac{1}{b - 1} [b \frac{b^{d+1} - 1}{b - 1} - (d + 1)] = \\ &= \frac{b(b^{d+1} - 1) - (b - 1)(d + 1)}{(b - 1)^2} = \frac{b^{d+2} - b - bd - b + d + 1}{(b - 1)^2} = \\ &= \frac{b^{d+2} - 2b - bd + d + 1}{(b - 1)^2} \end{aligned}$$

Spre exemplu [7], atunci când $b=10$, iar scopurile se află la mare adâncime, o căutare în adâncime iterativă de la adâncimea 1 până la adâncimea d extinde cu numai 11% noduri mai mult decât o singură căutare de tip breadth-first sau decât una în adâncime limitată la adâncimea d . Cu cât factorul de ramificare are o valoare mai mare, cu atât se micșorează costurile legate de extinderea repetată a acelorași stări. Chiar și atunci când factorul de ramificare are valoarea 2, căutarea în adâncime iterativă durează numai de aproximativ două ori mai mult decât o căutare de tip breadth-first completă [9]. Aceasta înseamnă că strategia

de căutare în adâncime iterativă are tot complexitatea de timp $O(b^d)$, iar complexitatea sa de spatiu este $O(bd)$. În general, căutarea în adâncime iterativă este metoda de căutare preferată atunci când există un spațiu al căutării foarte mare, iar adâncimea soluției nu este cunoscută.

2.2.3.2. Implementare în Prolog

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

cale(Nod1, Nod2, Drum)

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

Implementarea în Prolog a căutării în adâncime iterative este următoarea:

```

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

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

```

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

Programul 2.5

```

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

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

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

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

```

Interogarea Prologului se face astfel:

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

Răspunsul Prologului va fi:

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

Programul găsește soluția cel mai puțin adâncă, sub forma unui drum scris în ordine inversă, după care oprește căutarea. El va funcționa la fel de bine și într-un spațiu al stărilor conținând cicluri, datorită mecanismului de verificare `\+ (membru(UltimNod, Drum))`, care evită luarea în considerație a nodurilor deja vizitate.

Principalul avantaj al acestei metode este acela că ea necesită puțină memorie. La orice moment al execuției, necesitățile de spațiu se reduc la *un singur drum*, acela dintre nodul de început al căutării și nodul curent.

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

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

2.2.4. Strategii de căutare neinformată. Un exemplu

Un exemplu de problemă care poate fi rezolvată prin implementarea oricăreia dintre principalele strategii de căutare neinformată este “Problema mutării blocurilor”. Programul care urmează implementează succesiv tehnicele de căutare breadth-first, depth-first cu

detectarea ciclurilor și respectiv iterative deepening, folosind predicatele descrise anterior.

▪ Problema Mutării Blocurilor

Să considerăm că avem la dispoziție M blocuri, depozitate pe un teren. Pe suprafața terenului există un număr de N locații de depozitare. Pentru o mai bună utilizare a spațiului, blocurile pot fi așezate unele peste altele, în stive. În fiecare locație de depozitare există câte o astfel de stivă, eventual vidă. Un bloc poate fi mutat din locul său numai dacă el se află în vârful unei stive și poate fi așezat numai deasupra unei alte stive (eventual vide).

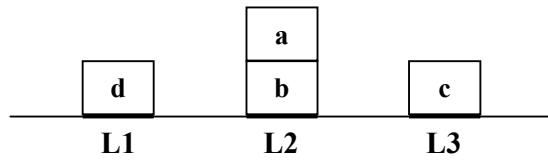
Pentru două configurații date, $C1$ și $C2$, ale așezării celor M blocuri în cele N locații, să se stabilească dacă și cum este posibil să se ajungă în configurația $C2$, plecând din configurația $C1$.

Rezolvare:

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

M = 4, N = 3

C1:



C2:

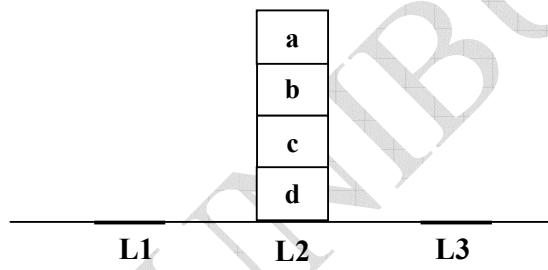


Fig 2.6

Programul 2.6

```
%Predicatul specific problemei mutarii blocurilor
%Definim relatia de succesiune

s(Lista_stive,Lista_stive_rez) :-
    membru(X,Lista_stive),
    X=[Varf|_],
    det_poz_el(Lista_stive,N,X),
    sterg_la_n(Lista_stive,Lista_stive_inter,N),
    membru(Y,Lista_stive),
```

```

det_poz_el(Lista_stive,N1,Y), N1\==N,
adaug_la_n(Varf,Lista_stive_inter,
            Lista_stive_rez,N1),
\+(permutare(Lista_stive,Lista_stive_rez)).

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

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

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

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

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

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

scop([],[],[],[]).

%Cautare de tip breadth-first

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

rezolva_b(NodInitial,Solutie):-

```

```

breadthfirst([[NodInitial]],Solutie).

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

breadthfirst([[Nod|Drum] | _], [Nod|Drum]):-scop(Nod).
breadthfirst([Drum|Drumuri],Solutie):-
    extinde(Drum,DrumNoi),
    concat(Drumuri,DrumNoi,Drumuril1),
    breadthfirst(Drumuril1,Solutie).
extinde([Nod|Drum],DrumNoi):-
    bagof([NodNou,Nod|Drum],
    (s(Nod,NodNou),
    \+ (membru(NodNou,[Nod|Drum]))),DrumNoi),
    !.
extinde(_,[]).

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

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

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

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

```

```

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

%Cautare de tip iterative deepening

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

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

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

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

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

pb_df_id:-tell('C:\\block_mut_ies_df_id_.txt'),

```

```

    initial(S),
    depth_first_iterative_deepening(S,Solutie),
    afisare(Solutie),told.

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

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

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

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

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

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

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

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

```

```

afis(L1,_,0):-af(L1).

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

af([]):-write(==).

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

afis2([],_,_).

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

det_el_n([H|_],1,H).

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

det_poz_el([H|_],1,H).

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

sterg(X,[X|L],L).

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

permutare([],[]).

```

```

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

membru(X,[X|_]).

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

concat([],L,L).

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

afisare([]).

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

```

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

```
?- pb_df_id.
```

Răspunsul sistemului va fi

```
yes
```

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

a
d b c
=====

a d
b c
=====

d
a b c
=====

b d
a c
=====

b
a d c
=====

b c
a d
=====

b
c
a d
=====

a
b
c
d
=====

2.3. Căutarea informată

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

2.3.1. Căutarea de tip best-first

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

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

1. Se presupune existența unei funcții euristice de evaluare, \hat{f} , cu rolul de a ne ajuta să decidem care nod ar trebui extins la pasul următor. Se va adopta *convenția* că valori mici ale lui \hat{f} indică nodurile cele mai bune. Această funcție se bazează pe informație specifică domeniului pentru care s-a formulat problema. Este o funcție de descriere a stărilor, cu valori reale.

2. Se extinde nodul cu cea mai mică valoare a lui $\hat{f}(n)$. În cele

ce urmează, se va presupune că extinderea unui nod va produce toți succesorii acestui nod.

3. Procesul se încheie atunci când următorul nod care ar trebui extins este un nod-scop.

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

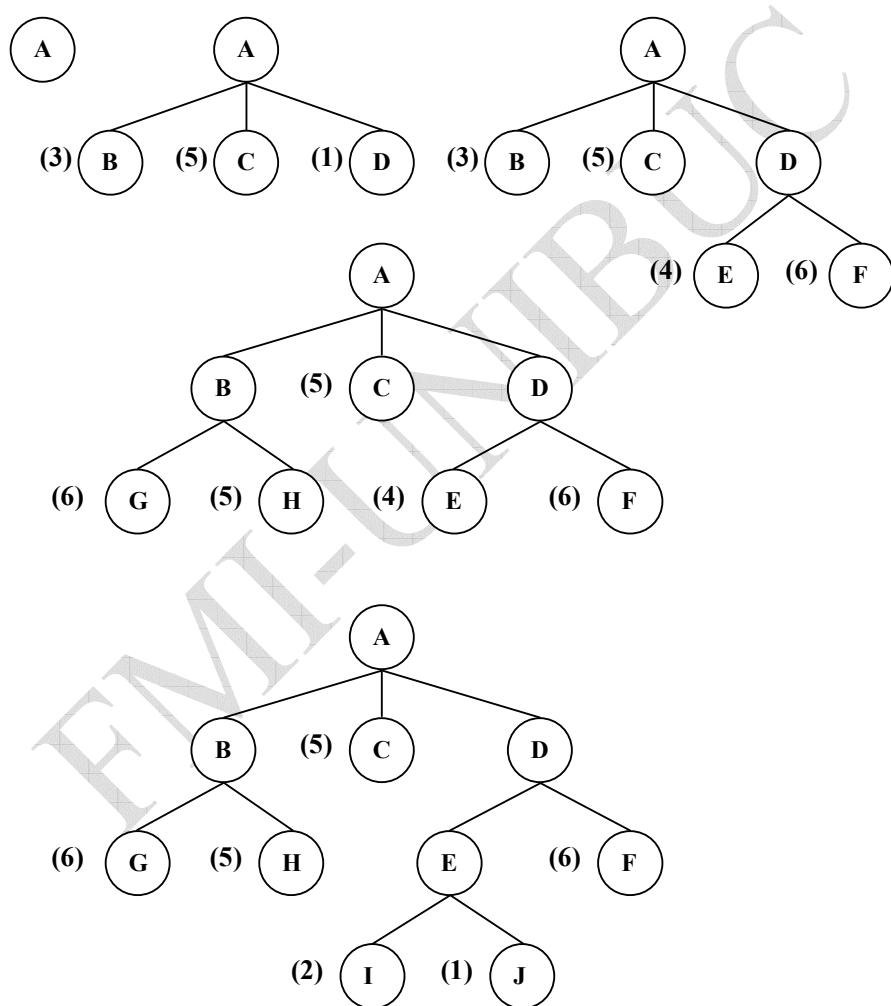


Fig. 2.7

În Fig. 2.7 există inițial un singur nod, A, astfel încât acesta va fi extins. Extinderea lui generează trei noduri noi. Funcția euristică este aplicată fiecărui dintre acestea. Întrucât nodul D este cel mai promițător, el este următorul nod extins. Extinderea lui va produce două noduri succesor, E și F, cărora li se aplică funcția euristică. În acest moment, un alt drum, și anume acela care trece prin nodul B, pare mai promițător, astfel încât el este urmat, generându-se nodurile G și H. În urma evaluării, aceste noduri par însă mai puțin promițătoare decât un alt drum, astfel încât este ales, în continuare, drumul prin D la E. E este apoi extins producând nodurile I și J. La pasul următor va fi extins nodul J, întrucât acesta este cel mai promițător. Procesul continuă până când este găsită o soluție.

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

Pentru a studia, în cele ce urmează, aspectele formale ale căutării de tip best-first¹⁴, vom începe prin a prezenta un algoritm de căutare generală bazat pe grafuri. Algoritmul este preluat din [7] și include versiuni ale căutării de tip best-first ca reprezentând cazuri particulare.

¹⁴ prezentate, aici, conform [7]

2.3.2. Algoritm de căutare general bazat pe grafuri

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

GraphSearch

1. Creează un arbore de căutare, T_r , care constă numai din nodul de start n_0 . Plasează pe n_0 într-o listă ordonată numită OPEN.
2. Creează o listă numită CLOSED, care inițial este vidă.
3. Dacă lista OPEN este vidă, EXIT cu eșec.
4. Selectează primul nod din OPEN, înlătură-l din lista OPEN și include-l în lista CLOSED. Numește acest nod n .
5. Dacă n este un nod scop, algoritmul se încheie cu succes, iar soluția este cea obținută prin urmarea în sens invers a unui drum de-a lungul arcelor din arborele T_r , de la n la n_0 . (Arcele sunt create la pasul 6).
6. Extinde nodul n , generând o mulțime, M , de succesi. Inclu-de M ca succesi ai lui n în T_r , prin crearea de arce de la n la fiecare membru al mulțimii M .
7. Reordonează lista OPEN, fie în concordanță cu un plan arbitrar, fie în mod heuristic.
8. Mergi la pasul 3.

□

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

2.3.2.1. Algoritmul A*

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

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

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

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

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

Pentru fiecare nod n , fie $\hat{h}(n)$, numit *factor heuristic*, o estimare a lui $h(n)$ și fie $\hat{g}(n)$, numit *factor de adâncime*, costul drumului de cost minim până la n găsit de A* până la pasul curent. Algoritmul A* va folosi funcția $\hat{f} = \hat{g} + \hat{h}$.

Așa cum se remarcă în [7], în definirea Algoritmului A* de până acum nu s-a ținut cont de următoarea *problemă*: ce se întâmplă dacă graful implicit în care se efectuează căutarea nu este un arbore? Cu alte cuvinte, există mai mult decât o unică secvență de acțiuni care pot conduce la aceeași stare a lumii plecând din starea inițială. (Există situații în care fiecare dintre succesorii nodului n îl are pe n ca succesor i.e. acțiunile sunt reversibile). Pentru a rezolva astfel de cazuri [7], pasul 6 al algoritmului GraphSearch trebuie înlocuit cu următorul pas 6':

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

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

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

Desigur, pentru a verifica existența acestor cicluri mai lungi, trebuie să se vadă dacă structura de date care etichetează fiecare succesor al

nodului n este egală cu structura de date care etichetează pe oricare dintre strămoșii nodului n . Pentru structuri de date complexe, acest pas poate mări complexitatea algoritmului. Pasul 6 modificat în pasul 6" face însă ca algoritmul să nu se mai întoarcă în cerc, în căutarea unui drum la scop.

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

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

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

folosesc anumite arce din graful de căutare anterior ce nu se aflau și în arborele de căutare anterior.

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

Algoritmul A*

1. Creează un graf de căutare G , constând numai din nodul inițial n_0 . Plasează n_0 într-o listă numită OPEN.
2. Creează o listă numită CLOSED, care inițial este vidă.
3. Dacă lista OPEN este vidă, EXIT cu eșec.
4. Selectează primul nod din lista OPEN, înlătură-l din OPEN și plasează-l în lista CLOSED. Numește acest nod n .
5. Dacă n este un nod scop, oprește execuția cu succes. Returnează soluția obținută urmând un drum de-a lungul pointerilor de la n la n_0 în G . (Pointerii definesc un arbore de căutare și sunt stabiliți la pasul 7).
6. Extinde nodul n , generând o mulțime, M , de succesi ai lui care nu sunt deja strămoși ai lui n în G . Instalează acești membri ai lui M ca succesi ai lui n în G .
7. Stabilește un pointer către n de la fiecare dintre membrii lui M care nu se găseau deja în G (adică nu se aflau deja nici în OPEN, nici în CLOSED). Adaugă acești membri ai lui M listei OPEN. Pentru fiecare

membru, m , al lui M , care se află deja în OPEN sau în CLOSED, redirecționează pointerul său către n , dacă cel mai bun drum la m găsit până în acel moment trece prin n . Pentru fiecare membru al lui M care se află deja în lista CLOSED, redirecționează pointerii fiecărui dintre descendenții săi din G astfel încât aceștia să țintească înapoi de-a lungul celor mai bune drumuri până la acești descendenți, găsite până în acel moment.

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

9. Mergi la pasul 3.

□

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

2.3.2.1.1. Admisibilitatea Algoritmului A*

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

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

Condiția asupra lui \hat{h} este:

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

- **Observații:**

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

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

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

Teorema 2.1

Atunci când sunt îndeplinite condițiile asupra grafurilor și asupra lui \hat{h} enunțate anterior și cu condiția să existe un drum de cost finit de la nodul inițial, n_0 , la un nod-scop, algoritmul A* garantează găsirea unui drum de cost minim la un scop.

Demonstratia teoremei:

Cea mai importantă componentă a demonstrației este următoarea lemă:

Lema 2.1

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

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

Presupunând această lemă ca fiind demonstrată (vezi p. 97), continuăm demonstrația teoremei: vom arăta că algoritmul A* se termină dacă există un scop accesibil și, mai mult, că el se termină prin găsirea unui drum optim la un scop.

Arătăm că A se termină:* să presupunem că algoritmul nu se termină. În acest caz, A* continuă să extindă noduri la infinit și, la un moment dat, începe să extindă noduri la o adâncime mai mare în arborele

de căutare decât orice limitare finită a adâncimii. (S-a presupus că graful în care se face căutarea are factor de ramificare finit). Întrucât costul fiecărui arc este mai mare decât $\varepsilon > 0$, valorile lui \hat{g} (și, prin urmare, și cele ale lui \hat{f}) ale tuturor nodurilor din OPEN vor depăși, până la urmă, pe $f(n_0)$. Dar acest lucru contrazice Lema 2.1.

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

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

Să presupunem acum că A* se termină prin găsirea unui scop care nu este optim, de exemplu prin găsirea lui n_{g2} cu $f(n_{g2}) > f(n_0)$ și în condițiile în care există un scop optim, $n_{g1} \neq n_{g2}$, cu $f(n_{g1}) = f(n_0)$.

Atunci când are loc terminarea în nodul n_{g2} , $\hat{f}(n_{g2}) \geq f(n_{g2}) > f(n_0)$. Dar, chiar înainte ca A* să îl selecteze pe n_{g2} pentru extindere, conform Lemei 2.1, a existat un nod n^* în OPEN și aflat pe un drum optim, cu $\hat{f}(n^*) \leq f(n_0)$. Prin urmare, A* nu ar fi putut să îl selecteze pe n_{g2} pentru extindere, deoarece A* selectează întotdeauna acel nod având cea mai mică valoare a lui \hat{f} și $\hat{f}(n^*) \leq f(n_0) < f(n_{g2})$. Deci teorema a fost demonstrată.

□

Demonstrația lemei:

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

- (1) sunt valabile la începutul algoritmului;
 - (2) dacă sunt valabile înainte de extinderea unui nod, vor continua să fie valabile și după extinderea nodului.
- (1) *Cazul de bază:* la începutul căutării, când numai nodul n_0 a fost selectat pentru extindere, nodul n_0 este în OPEN, el este un drum optim la scop și A* a găsit acest drum. De asemenea, $\hat{f}(n_0) \leq f(n_0)$ deoarece $\hat{f}(n_0) = \hat{h}(n_0) \leq f(n_0)$. Astfel, nodul n_0 poate fi, în acest stadiu, nodul n^* al lemei.
- (2) *Pasul de inducție:* presupunem adevărate concluziile lemei la momentul la care au fost extinse m noduri ($m \geq 0$) și, folosind această presupunere, arătăm că ele sunt adevărate la momentul la care au fost extinse $m+1$ noduri.

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

Cazul I. Dacă n^* **nu** este selectat pentru extindere la pasul $m+1$, n^* are aceleași proprietăți pe care le avea înainte, deci pasul de inducție este demonstrat în acest caz. Fig. 2.8 corespunde acestei situații și prezintă Algoritmul A* selectând nodul n_1 ca fiind al $(m+1)$ -lea nod extins:

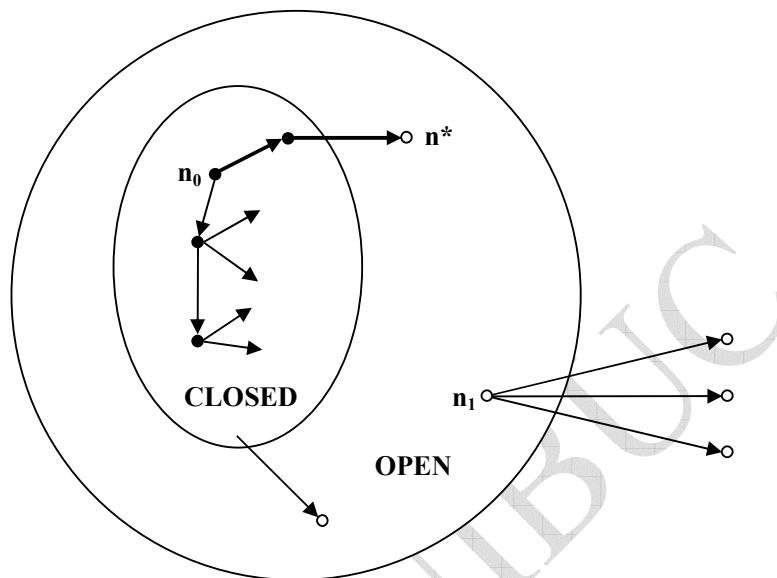


Fig. 2.8

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

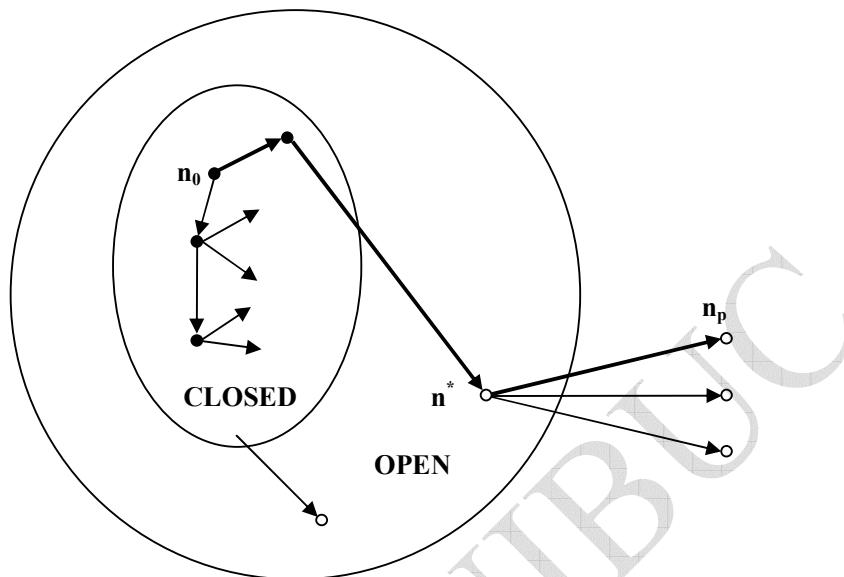


Fig. 2.9

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

$$\begin{aligned}
 \hat{f}(n^*) &= \hat{g}(n^*) + \hat{h}(n^*) \leq && \text{(prin definiție)} \\
 &\leq g(n^*) + h(n^*) \leq && \text{deoarece } \begin{cases} \hat{g}(n^*) = g(n^*) \\ \hat{h}(n^*) \leq h(n^*) \end{cases} \\
 &\leq f(n^*) \leq && \text{deoarece } g(n^*) + h(n^*) = f(n^*) \quad \text{prin} \\
 &&& \text{definiție} \\
 &\leq f(n_0) && \text{deoarece } f(n^*) = f(n_0), \text{ întrucât } n^* \text{ se} \\
 &&& \text{află pe un drum optim, ceea ce completează demonstrația lemei.}
 \end{aligned}$$

□

Definiția 2.1

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

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

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

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

Teorema 2.2

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

□

Rezultă de aici că A^*_1 extinde cel puțin tot atâtea noduri câte extinde A^*_2 și, prin urmare, algoritmul mai informat A^*_2 este și mai eficient. În concluzie, se caută o funcție \hat{h} ale cărei valori sunt cât se poate de apropiate de cele ale funcției h (pentru o căt mai mare eficiență a căutării), dar fără să le depășească pe acestea (pentru admisibilitate).

Desigur, pentru a evalua eficiența *totală* a căutării, trebuie luat în considerație și costul calculării lui \hat{h} .

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

2.3.2.1.2. Condiția de consistență

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

Definiția 2.2

Se spune că \hat{h} îndeplinește condiția de consistență dacă, pentru orice astfel de pereche (n_i, n_j) de noduri din graful de căutare,

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

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

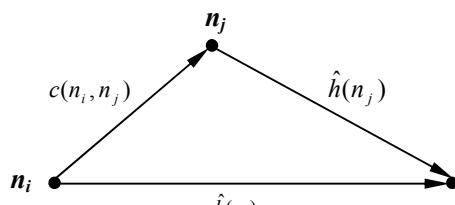
Condiția de consistență mai poate fi formulată și sub una din formele următoare:

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

ceea ce conduce la următoarea *interpretare* a ei: de-a lungul oricărui drum din graful de căutare, estimarea făcută asupra costului optim rămas

pentru a atinge scopul nu poate descrește cu o cantitate mai mare decât costul arcului de-a lungul acelui drum. Se spune că funcția euristică este *local consistentă* atunci când se ia în considerație costul cunoscut al unui arc.

Condiția de consistență:



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

Condiția de consistență implică faptul că valorile funcției \hat{f} corespunzătoare nodurilor din arborele de căutare descresc monoton pe măsură ce ne îndepărțăm de nodul de start.

- Fie n_i și n_j două noduri în *arborele de căutare* generat de algoritmul A*, cu n_j succesor al lui n_i . Atunci, dacă condiția de consistență este satisfăcută, avem:

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

Demonstrație:

Pentru a demonstra acest fapt, se începe cu condiția de consistență:

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

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

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

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

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

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

□

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

Există următoarea teoremă referitoare la condiția de consistență:

Teorema 2.3

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

Demonstratie:

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

Fie $\xi = (n_0, n_1, \dots, n_l, n_{l+1}, \dots, n = n_k)$ o secvență de noduri din G care constituie un drum optim de la n_0 la n . Fie n_l ultimul nod din ξ extins de A*. Întrucât n_l este ultimul “nod închis” din ξ , adică ultimul

din lista CLOSED, știm că n_{l+1} este în lista OPEN și deci reprezintă un candidat pentru extindere.

Pentru orice nod n_i și succesorul său n_{l+1} din ξ , avem¹⁵

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

atunci când condiția de consistență este satisfăcută.

Tranzitivitatea relației \geq ne dă:

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

În particular,

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

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

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

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

S-a stabilit deja că

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

Din (1) și (2) rezultă

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

Prin urmare,

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

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

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

□

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

2.3.2.1.3. Optimalitatea Algoritmului A*

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

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

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

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

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

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

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

Combinând (1) cu (2) obținem:

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

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

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

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

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

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

2.3.2.1.4. Completitudinea Algoritmului A*

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

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

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

- există un nod cu *factor de ramificare infinit*;
- există un drum cu un *cost finit*, dar care are un *număr infinit de noduri*. (Acest lucru ar fi posibil conform paradoxului lui Zeno, care vrea să arate că o piatră aruncată spre un copac nu va ajunge

niciodată la acesta. Astfel, se imaginează că traectoria pietrei este împărțită într-un sir de faze, fiecare dintre acestea acoperind jumătate din distanța rămasă până la copac. Aceasta conduce la un număr infinit de pași cu un cost total finit).

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

2.3.2.1.5. Complexitatea Algoritmului A*

S-a arătat (demonstrație pe care nu o vom reproduce în cadrul restrâns al acestui curs) că o creștere exponențială va interveni, în afara cazului în care eroarea în funcția euristică nu crește mai repede decât logaritmul costului efectiv al drumului. Cu alte cuvinte, *condiția pentru o creștere subexponențială* este:

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

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

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

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

2.3.2.2. Iterative Deepening A* (IDA*)

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

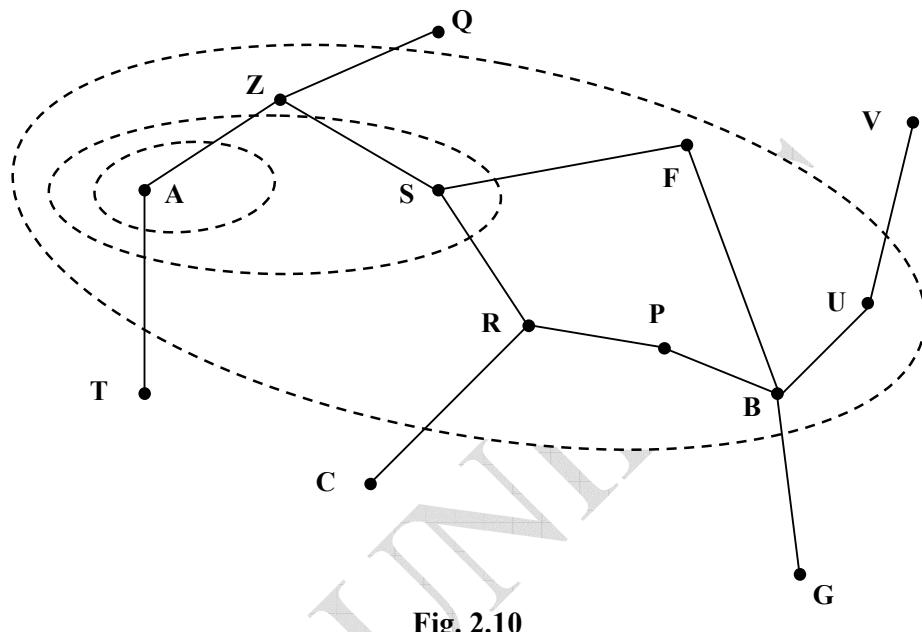
În cadrul acestui algoritm fiecare iterație reprezintă o căutare de tip depth-first, iar căutarea de tip depth-first este modificată astfel încât ea să folosească *o limită a costului* și nu o limită a adâncimii.

Faptul că în cadrul algoritmului A* f nu descrește niciodată de-a lungul oricărui drum care pleacă din rădăcină ne permite să trasăm, din punct de vedere conceptual, *contururi* în spațiul stărilor. Astfel, în interiorul unui contur, toate nodurile au valoarea $f(n)$ mai mică sau egală cu o aceeași valoare. În cazul algoritmului IDA* fiecare iterație extinde toate nodurile din interiorul conturului determinat de costul f curent, după care se trece la conturul următor. De îndată ce căutarea în interiorul unui contur dat a fost completată, este declanșată o nouă iterație, folosind un nou cost f , corespunzător următorului contur. Fig. 2.10 prezintă căutări iterative în interiorul câte unui contur.

Algoritmul IDA* este complet și optim cu aceleași amendamente ca și A*. Deoarece este de tip depth-first *nu necesită decât un spațiu proporțional cu cel mai lung drum pe care îl explorează*.

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

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



2.3.3. Implementarea în Prolog a căutării de tip best-first

Pentru o mai bună înțelegere a metodei, în cele ce urmează, prezentarea căutării de tip best-first, în vederea implementării ei în Prolog, se va face conform abordării din [1].

Astfel, vom imagina căutarea de tip best-first funcționând în felul următor: căutarea constă dintr-un număr de subprocese "concurente", fiecare explorând alternativa sa, adică propriul subarbore. Subarborii au subarbori, care vor fi la rândul lor explorați de subprocese ale subproceselor, și.a.m.d.. Dintre toate aceste subprocese doar unul este

activ la un moment dat și anume cel care se ocupă de alternativa cea mai promițătoare (adică alternativa corespunzătoare celei mai mici \hat{f} -valori). Celelalte procese așteaptă până când \hat{f} -valorile se schimbă astfel încât o altă alternativă devine mai promițătoare, caz în care procesul corespunzător acesteia devine activ. Acest mecanism de activare-dezactivare poate fi privit după cum urmează: procesului corespunzător alternativei curente de prioritate maximă î se alocă un buget și, atâta vreme cât acest buget nu este epuizat, procesul este activ. Pe durata activității sale, procesul își expandează propriul subarbore, iar în cazul atingerii unei stări-scop este anunțată găsirea unei soluții. Bugetul acestei funcționări este determinat de \hat{f} -valoarea corespunzătoare celei mai apropiate alternative concurente.

Ilustrăm această idee în exemplul următor. Considerăm orașele s, a, b, c, d, e, f, g, t unite printr-o rețea de drumuri ca în Fig. 2.11. Aici fiecare drum direct între două orașe este etichetat cu lungimea sa; numărul din căsuță alăturată unui oraș reprezintă distanța în linie dreaptă între orașul respectiv și orașul t. Ne punem problema determinării celui mai scurt drum între orașul s și orașul t utilizând strategia best-first. Definim în acest scop funcția \hat{h} bazându-ne pe distanța în linie dreaptă între două orașe. Astfel, pentru un oraș X, definim

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

unde $dist(X, t)$ reprezintă distanța în linie dreaptă între X și t.

În acest exemplu, căutarea de tip best-first este efectuată prin intermediul a două procese, P₁ și P₂, ce explorează fiecare câte una din

cele două căi alternative. Calea de la s la t via nodul a corespunde procesului P_1 , iar calea prin nodul e corespunde procesului P_2 .

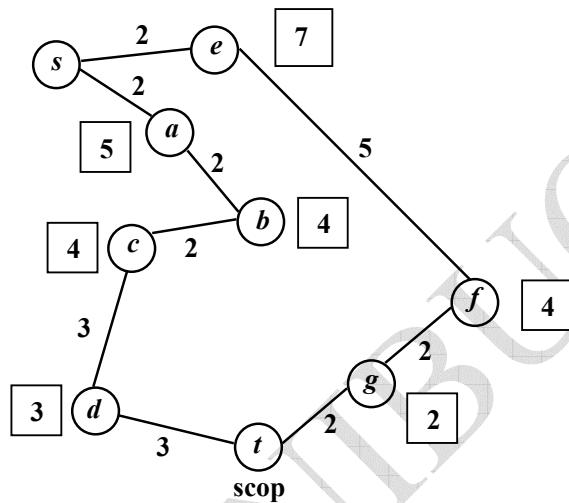


Fig. 2.11

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

Căutarea schițată mai sus pornește din nodul inițial și este continuată cu generarea unor noduri noi, conform relației de succesiune. În timpul acestui proces, este generat un arbore de căutare, a cărui rădăcină este nodul de start. Acest arbore este expandat în direcția cea mai promițătoare conform \hat{f} - valorilor, până la găsirea unei soluții.

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

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

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

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

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

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

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

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

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

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

Argumentele sale au următoarele semnificații:

- **Drum** reprezintă calea între nodul de start al căutării și **Arb**
- **Arb** este arborele (subarborele) curent de căutare
- **Limita** este \hat{f} - limita pentru expandarea lui **Arb**
- **Rez** este un indicator a cărui valoare poate fi "da", "nu", "imposibil"

- **Solutie** este o cale de la nodul de start ("prin **Arb1**") către un nod-scop (în limita **Limita**), dacă un astfel de nod-scop există.

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

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

Vom prezenta în continuare o implementare a unei variante a metodei best-first [1], în SICStus Prolog, implementare care folosește considerentele anterioare.

Strategia best-first

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

```
bestfirst(Nod_initial,Solutie) :-  

  expandeaza([],l(Nod_initial,0/0),9999999,_,_,  

  da,Solutie).
```

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

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

```
expandeaza(Drum,l(N,F/G),Limita,Arb1,Rez,Sol) :-  

  F=<Limita,  

  (bagof(M/C,(s(N,M,C), \+ (membru(M,Drum))),Succ),!,  

  listasucc(G,Succ,As),  

  cea_mai_buna_f(As,F1),  

  expandeaza(Drum,t(N,F1/G,As),Limita,Arb1, Rez,Sol);  

  Rez=imposibil).
```

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

```
expandeaza(Drum,t(N,F/G,[A|As]),Limita,Arb1,Rez,  

Sol) :-  

  F=<Limita,  

  cea_mai_buna_f(As,BF),  

  min(Limita,BF,Limita1),
```

```
expandeaza([N|Drum],A,Limita1,A1,Rez1,Sol),
continua(Drum,t(N,F/G,[A1|As]),Limita,Arb1,
Rez1,Rez,Sol).
```

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

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

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

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

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

```
continua(_,_,_,_da,da,Sol).
continua(P,t(N,F/G,[A1|As]),Limita,Arb1,nu,Rez,Sol):-
    insereaza(A1,As,NAs),
    cea_mai_buna_f(NAs,F1),
    expandeaza(P,t(N,F1/G,NAs),Limita,Arb1,Rez,Sol).
continua(P,t(N,F/G,[_|As]),Limita,Arb1,imposibil,Rez,
Sol):-cea_mai_buna_f(As,F1),
    expandeaza(P,t(N,F1/G,As),Limita,Arb1,Rez,Sol).
```

```
listasucc(_,[],[]).
listasucc(G0,[N/C|NCs],Ts):-
```

```

G is G0+C,
h(N,H),
F is G+H,
listasucc(G0,NCs,Ts1),
insereaza(l(N,F/G),Ts1,Ts).

```

%Predicatul insereaza(A,As,As1) este utilizat pentru
 %inserarea unui arbore A intr-o lista de arbori As,
 %mentionand ordinea impusa de \hat{f} -valorile lor.

```

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

```

```

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

```

```

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

```

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

```

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

```

%In cazul unei liste de arbori vide, \hat{f} -valoarea
 %determinata este foarte mare.

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

- **s (Nod, Nod1, Cost)**

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

- **scop (Nod)**

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

- **h (Nod, H)**

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

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

2.3.3.1. Exemple

▪ Problema 8-puzzle

Se consideră un tablou pătratic de dimensiune 3x3, împărțit în 9 câmpuri pătrate de latură 1. Opt dintre aceste câmpuri conțin câte o plăcuță etichetată cu un număr între 1 și 8, în timp ce al nouălea câmp nu conține nimic. O plăcuță poate fi deplasată de pe câmpul sau pe un

câmp vecin, dacă astfel nu se suprapune peste altă plăcuță, deci dacă acest ultim câmp nu conține nici o plăcuță. Fiind date două configurații $C1, C2$, ale plăcuțelor pe tabloul $3x3$, se cere să se spună dacă și cum este posibil să se ajungă din configurația $C1$ în configurația $C2$ printr-un număr minim de mutări.

Rezolvare:

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

Pentru rezolvarea problemei pot fi luate în considerație mai multe funcții euristice [1]. Astfel, pentru o stare (configurație) S , putem defini:

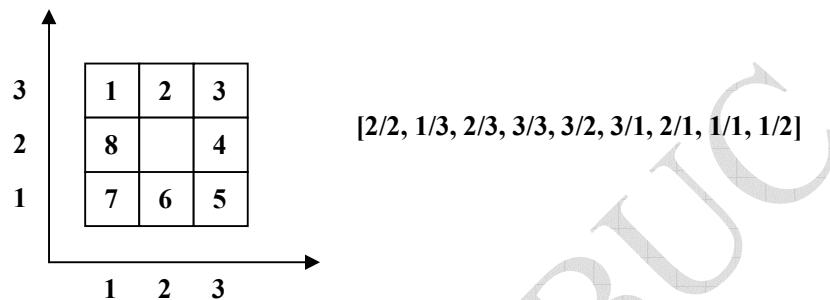
- $\hat{h}_1(S)$ = numărul de plăcuțe care nu sunt la locul lor în starea S față de starea finală;
- $\hat{h}_2(S)$ = suma distanțelor pe verticală și pe orizontală de la poziția fiecărei plăcuțe în configurația curentă (S), la poziția respectivei plăcuțe în configurația finală.

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

- (1) poziția curentă a locului liber (neocupat de nici o plăcuță),

- (2) poziția curentă a plăcuței etichetate cu 1,
- (3) poziția curentă a plăcuței etichetate cu 2 etc.

Exemplu:



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

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

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

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

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

- indicele plăcuței aflate în centrul tabloului este 1;

- indicele unei plăcuțe aflate într-o poziție diferită de centru este 0, dacă acea plăcuță este urmată (în sens invers-trigonometric) de succesorul său din configurația finală;
- în orice altă situație, indicele este egal cu 2.

De exemplu, pentru configurația S_a din Fig.2.13 a), $\sec v(S_a) = 6$.

Fig. 2.13 prezintă trei exemple de configurații inițiale:

1	3	4
8		2
7	6	5

a)

2	8	3
1	6	4
7		5

b)

2	1	6
4		8
7	5	3

c)

Fig. 2.13

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

În ciuda calităților ei, euristică \hat{h}_3 nu este totuși admisibilă. Cu alte cuvinte, ea nu garantează că soluția cea mai scurtă este găsită întotdeauna înaintea uneia mai lungi. Motivul este acela că funcția aleasă

nu satisfacă condiția de admisibilitate pentru toate nodurile din spațiul stărilor. Spre exemplu, pentru configurația inițială S_a , avem:

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

$$h(S_a) = 4$$

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

mandist (P1, P2, D)

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

Programul 2.7

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

%Definim relația de succesiune; considerăm ca toate
%arcele au costul 1.

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

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

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

```

interschimba(Go1,Placuta,[Placuta|Ts],[Go1|Ts]) :-  

    distman(Go1,Placuta,1).  

interschimba(Go1,Placuta,[T1|Ts],[T1|Ts1]) :-  

    interschimba(Go1,Placuta,Ts,Ts1).  

membru(H,[H|_]).  

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

h([Go1|Placute],H) :-  

    scop([Go1|Placute_final]),  

    disttot(Placute,Placute_final,D),  

    secv(Placute,S),  

    H is D+3*S.  

secv([Prima|Placute],S) :-  

    secv([Prima|Placute],Prima,S).  

secv([Placutal,Placuta2|Placute],Prima,S) :-  

    scor(Placutal,Placuta2,S1),  

    secv([Placuta2|Placute],Prima,S2),  

    S is S1+S2.  

secv([Ultima],Prima,S) :- scor(Ultima,Prima,S).  

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

%Placuta plasata in centru are scorul 1.  

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

```

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

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

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

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

```
expandeaza(Drum,l(N,F/G),Limita,Arb1,Rez,Sol) :-
  F=<Limita,
  (bagof(M/C,(s(N,M,C), \+ (membru(M,Drum))),Succ),!,
  listasucc(G,Succ,As),
  cea_mai_buna_f(As,F1),
  expandeaza(Drum,t(N,F1/G,As),Limita,Arb1, Rez,Sol);
  Rez=imposibil).
```

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

```
expandeaza(Drum,t(N,F/G,[A|As]),Limita,Arb1,Rez,
Sol) :-
  F=<Limita,
  cea_mai_buna_f(As,BF),
  min(Limita,BF,Limita1),
  expandeaza([N|Drum],A,Limita1,A1,Rez1,Sol),
  continua(Drum,t(N,F/G,[A1|As]),Limita,Arb1,
  Rez1,Rez,Sol).
```

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

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

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

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

%Caz 5: In cazul unor \hat{f} -valori mai mari decat Bound, %arborele nu mai poate fi extins.

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

```
continua(P, t(N, F/G, [A1|As]), Limita, Arb1, nu, Rez, Sol):-  
    insereaza(A1, As, NAs),  
    cea_mai_buna_f(NAs, F1),  
    expandeaza(P, t(N, F1/G, NAs), Limita, Arb1, Rez, Sol).
```

```
continua(P, t(N, F/G, [_|As]), Limita, Arb1, impossibil, Rez,  
Sol):-  
    cea_mai_buna_f(As, F1),  
    expandeaza(P, t(N, F1/G, As), Limita, Arb1, Rez, Sol).
```

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

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

```

insereaza(l(N,F/G),Ts1,Ts).

% Predicatul insereaza(A,As,As1) este utilizat pentru
% inserarea unui arbore A intr-o lista de arbori As,
% menținând ordinea impusă de  $\hat{f}$ -valorile lor.

```

```

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

insereaza(A,[A1|As],[A1|As1]):-insereaza(A,As,As1).

```

```

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

```

```

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

```

% Predicatul cea_mai_buna_f(As,F) este utilizat pentru
% a determina cea mai bună \hat{f} -valoare a unui arbore din
% lista de arbori As, dacă aceasta lista este nevidă;
% lista As este ordonată după \hat{f} -valorile subarborilor
% constituenți.

```

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

```

% În cazul unei liste de arbori vide, \hat{f} -valoarea
% determinată este foarte mare.

Interogarea Prologului se face după cum urmează:

```
?- pb.
```

Răspunsul sistemului va fi

```
yes
```

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

2	8	3
1	6	4
7		5

în starea finală

1	2	3
8		4
7	6	5

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

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

1	2	3
8	4	
7	6	5

1	2	3
8	4	
7	6	5

■ Problema misionarilor și canibalilor

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

Rezolvare:

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

Prezentăm, în continuare, două exemple de funcții euristice. Pentru o stare dată S, notăm cu $n_c(\text{est})$ numărul de canibali de pe malul de est și cu $n_m(\text{est})$ numărul de misionari de pe același mal. Definim:

$$\hat{h}_1(S) = \left\lceil \frac{(n_c(\text{est}) + n_m(\text{est}))}{M} \right\rceil$$

$$\hat{h}_2(S) = \begin{cases} 2 \left\lceil \frac{(n_c(\text{est}) + n_m(\text{est}) - 2)}{(M - 1)} \right\rceil + 1, & \text{daca } n_c(\text{est}) + n_m(\text{est}) \neq 0 \\ & \text{si barca se află pe malul de est} \\ 2 \left\lceil \frac{(n_c(\text{est}) + n_m(\text{est}) - 1)}{(M - 1)} \right\rceil + 2, & \text{daca } n_c(\text{est}) + n_m(\text{est}) \neq 0 \\ & \text{si barca se află pe malul de vest} \\ 0, & \text{daca } n_c(\text{est}) + n_m(\text{est}) = 0 \end{cases}$$

Funcția euristică \hat{h}_2 descrie situația în care un transport est-vest se face cu cât mai mulți pasageri (eventual cu M - capacitatea totală a bărcii), iar unul vest-est se face cu cât mai puțini (adică unul singur), fără a ține cont de relația dintre numărul de canibali și cel de misionari de pe cele două maluri. Dacă, în plus, intervine și constrângerea privitoare la această relație numerică între canibali și misionari, cu siguranță că transportul va fi făcut într-un număr de drumuri mai mare sau egal cu cel specificat de euristica h_2 . Prin urmare, această funcție euristică este admisibilă. Se observă ușor că și funcția euristică \hat{h}_1 este admisibilă, dar și că aceasta este mai puțin informată decât \hat{h}_2 .

2.3.4. Concluzii

1. Până acum, *cunoștințele* au fost folosite numai în procesul formulării unei probleme în termeni de *stări* și de *operatori*. Fiind însă dată o problemă bine definită, singurul loc în care putem aplica cunoștințele este în funcția ce organizează coada. Această funcție trebuie să determine următorul nod care va fi extins. De obicei, cunoștințele care permit determinarea acestuia sunt furnizate de către o *funcție de evaluare* care întoarce un număr menit să descrie cât de oportună sau neoportună este extinderea unui anumit nod la pasul următor. Atunci când nodurile sunt ordonate astfel încât cel care este cel mai bine evaluat să fie primul extins, strategia care rezultă poartă denumirea de *căutare de tip best-first*.
2. Una dintre cele mai simple strategii de căutare de tip best-first constă în minimizarea costului estimat pentru a fi atins scopul. Cu alte cuvinte, nodul a cărui stare este considerată ca fiind cea mai apropiată de starea-scop este întotdeauna primul extins. Pentru majoritatea problemelor, costul atingerii scopului plecându-se dintr-o anumită stare nu poate fi determinat în mod exact, dar poate fi estimat. O funcție care calculează astfel de estimări ale costului se numește *funcție euristică* și este, de obicei, notată cu h :

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

O căutare de tip best-first care folosește această funcție h pentru a selecta următorul nod care va fi extins se numește o *căutare de tip Greedy*.

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

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

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

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

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

Căutarea de tip best-first care folosește acest f ca funcție de evaluare și o funcție h admisibilă se numește *căutare de tip A**. A* este *complet* și *optimal*.

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

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

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

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

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

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

FMI-UNIBUC

CAPITOLUL 3

JOCURILE CA PROBLEME DE CĂUTARE

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

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

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

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

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

În cele ce urmează, ne vom referi la tehnici de joc corespunzătoare unor *jocuri de două persoane cu informație completă*, cum ar fi șahul. În cazul jocurilor interesante, arborii rezultați sunt mult prea complecsi pentru a se putea realiza o căutare exhaustivă, astfel încât sunt necesare abordări de o natură diferită. Una dintre metodele clasice se bazează pe „*principiul minimax*”, implementat în mod eficient sub forma Algoritmului Alpha-Beta (bazat pe aşa-numita *tehnică de alpha-beta retezare*).

3.1. O definire formală a jocurilor

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

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

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

- *starea inițială*, care include poziția de pe tabla de joc și o indicație referitoare la cine face prima mutare;
- *o mulțime de operatori*, care definesc mișările permise (“legale”) unui jucător;

- *un test terminal*, care determină momentul în care jocul ia sfârșit;
- *o funcție de utilitate* (numită și *funcție de plată*), care acordă o valoare numerică rezultatului unui joc; în cazul jocului de șah, spre exemplu, rezultatul poate fi *câștig*, *pierdere* sau *remiză*, situații care pot fi reprezentate prin valorile 1, -1 sau 0.

Dacă un joc ar reprezenta o problemă standard de căutare, atunci acțiunea jucătorului MAX ar consta din căutarea unei secvențe de mutări care conduc la o stare terminală reprezentând o stare câștigătoare (conform funcției de utilitate) și din efectuarea primei mutări aparținând acestei secvențe. Acțiunea lui MAX interacționează însă cu cea a jucătorului MIN. Prin urmare, MAX trebuie să găsească *o strategie* care va conduce la o stare terminală câștigătoare, indiferent de acțiunea lui MIN. Această strategie include mutarea corectă a lui MAX corespunzătoare fiecărei mutări posibile a lui MIN. În cele ce urmează, vom începe prin a arăta cum poate fi găsită strategia optimă (sau rațională), deși în realitate nu vom dispune de timpul necesar pentru a o calcula.

3.2. Algoritmul Minimax

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

informație și încearcă întotdeauna să se mute la acea stare care este cea mai nefavorabilă lui MAX.

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

Algoritmul Minimax

1. Generează întregul arbore de joc, până la stările terminale.
2. Aplică funcția de utilitate fiecărei stări terminale pentru a obține valoarea corespunzătoare stării.
3. Deplasează-te înapoi în arbore, de la nodurile-frunze spre nodul-rădăcină, determinând, corespunzător fiecărui nivel al arborelui, valorile care reprezintă utilitatea nodurilor aflate la acel nivel. Propagarea acestor valori la niveluri anterioare se face prin intermediul nodurilor-părinte succesive, conform următoarei reguli:
 - dacă starea-părinte este un nod de tip MAX, atribuie-i maximul dintre valorile avute de fiilor săi;
 - dacă starea-părinte este un nod de tip MIN, atribuie-i minimul dintre valorile avute de fiilor săi.
4. Ajuns în nodul-rădăcină, alege pentru MAX acea mutare care conduce la valoarea maximă.

□

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

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

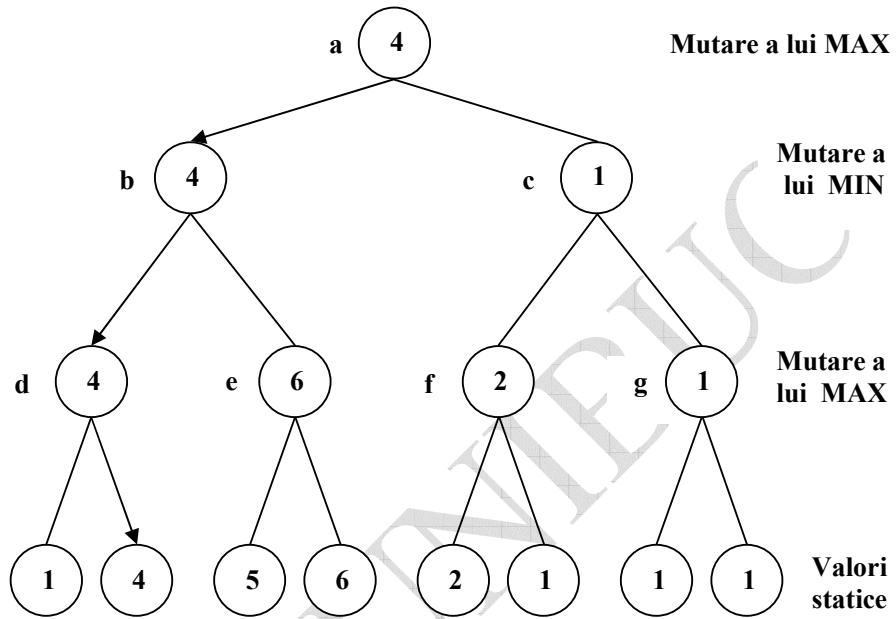


Fig. 3.1

Valorile pozițiilor de la ultimul nivel sunt determinate de către funcția de utilitate și se numesc *valori statice*. Valorile minimax ale nodurilor interne sunt calculate în mod dinamic, în manieră bottom-up, nivel cu nivel, până când este atins nodul-rădăcină. Valoarea rezultată, corespunzătoare acestuia, este 4 și, prin urmare, cea mai bună mutare a lui MAX din poziția *a* este *a-b*. Cel mai bun răspuns al lui MIN este *b-d*. Această secvență a jocului poartă denumirea de *variație principală*. Ea definește jocul optim de tip minimax pentru ambele părți. Se observă

că valoarea pozițiilor de-a lungul variației principale nu variază. Prin urmare, mutările corecte sunt cele care *conservează valoarea jocului*.

Dacă adâncimea maximă a arborelui este m și dacă există b “mutări legale” la fiecare punct, atunci complexitatea de timp a Algoritmului Minimax este $O(b^m)$. Algoritmul reprezintă o căutare de tip depth-first (deși aici este sugerată o implementare bazată pe recursivitate și nu una care folosește o coadă de noduri), astfel încât cerințele sale de spațiu sunt numai liniare în m și b .

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

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

Este de observat distincția care se face, în acest cadru, între *arborele de joc* și *arborele de căutare*. Acesta din urmă este de obicei o parte a arborelui de joc (cea superioară), și anume acea parte care este

în mod explicit generată de către procesul de căutare. Prin urmare, poziții terminale ale căutării nu reprezintă, neapărat, poziții terminale ale jocului.

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

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

3.2.1. Funcții de evaluare

O *funcție de evaluare* întoarce o *estimatie*, realizată dintr-o poziție dată, a utilității așteptate a jocului. Ea are la bază evaluarea șanselor de câștigare a jocului de către fiecare dintre părți, pe baza calculării caracteristicilor unei poziții. În mod evident, performanța unui program referitor la jocuri este extrem de dependentă de calitatea funcției de evaluare utilizate.

Funcția de evaluare trebuie să îndeplinească anumite condiții evidente: ea trebuie să concorde cu funcția de utilitate în ceea ce privește stările terminale, calculele efectuate nu trebuie să dureze prea mult și ea trebuie să reflecte în mod corect șansele efective de câștig. O valoare a funcției de evaluare acoperă mai multe poziții diferite, grupate laolaltă într-o *categorie* de poziții etichetată cu o anumită valoare. Spre exemplu, în jocul de șah, fiecare *pion* poate avea valoarea 1, un *nebun* poate avea valoarea 3 și... În poziția de deschidere evaluarea este 0 și toate pozițiile până la prima captură vor avea aceeași evaluare. Dacă MAX reușește să captureze un nebun fără a pierde o piesă, atunci poziția rezultată va fi evaluată la valoarea 3. Toate pozițiile de acest fel ale lui MAX vor fi grupate într-o *categorie* etichetată cu “3”. Funcția de evaluare trebuie să reflecte șansa ca o poziție aleasă la întâmplare dintr-o asemenea categorie să conducă la câștig (sau la pierdere sau la remiză) pe baza experienței anterioare.

Funcția de evaluare cel mai frecvent utilizată presupune că valoarea unei piese poate fi stabilită independent de celelalte piese

existente pe tablă. Un asemenea tip de funcție de evaluare se numește *funcție liniară ponderată*, întrucât are o expresie de forma

$$w_1f_1 + w_2f_2 + \dots + w_nf_n,$$

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

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

3.2.2. Implementare în Prolog

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

miminax(Poz, SuccBun, Val)

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

minimax(Poz, SuccBun, Val) :

Poz este o poziție, **Val** este valoarea ei de tip minimax;

cea mai bună mutare de la **Poz** conduce la poziția **SuccBun**.

La rândul ei, relația

mutari(Poz, ListaPoz)

coresponde regulilor jocului care indică mutările “legale” (admise):

ListaPoz este lista pozițiilor succesor legale ale lui **Poz**. Predicatul **mutari** va eșua dacă **Poz** este o poziție de căutare terminală (o frunză a arborelui de căutare). Relația

celmaibun(ListaPoz , PozBun , ValBuna)

selectează cea mai bună poziție **PozBun** dintr-o listă de poziții candidate **ListaPoz**. **ValBuna** este valoarea lui **PozBun** și, prin urmare, și a lui **Poz**. „Cel mai bun” are aici sensul de maxim sau de minim, în funcție de partea care execută mutarea.

O implementare a principiului Minimax este realizată de următorul program Prolog, preluat din [1]:

Principiul Minimax

minimax(Poz, SuccBun, Val) :-

% mutările legale de la Poz produc ListaPoz

mutări(Poz, ListaPoz) , ! ,

celmaibun(ListaPoz , SuccBun , Val) ;

%Poz nu are succesișri și este evaluat

%în mod static

staticval(Poz, Val) .

```

celmaibun([Poz], Poz, Val) :-  

    minimax(Poz, _, Val), !.  

celmaibun([Poz1 | ListaPoz], PozBun, ValBuna) :-  

    minimax(Poz1, _, Val1),  

    celmaibun(ListaPoz, Poz2, Val2),  

    maibine(Poz1, Val1, Poz2, Val2, PozBun, ValBuna).  

% Poz0 mai bună decât Poz1  

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

    % Min face o mutare la Poz0  

    % Max preferă valoarea maximă  

    mutare_min(Poz0),  

    Val0 > Val1, !  

    ;  

    % Max face o mutare la Poz0  

    % Min preferă valoarea mai mică  

    mutare_max(Poz0),  

    Val0 < Val1, !.  

% Altfel, Poz1 este mai bună decât Poz0  

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

```

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

Tehnica pe care o vom examina, în cele ce urmează, este numită în literatura de specialitate **alpha-beta pruning** (“alpha-beta retezare”). Atunci când este aplicată unui arbore de tip minimax standard, ea va

întoarce aceeași mutare pe care ar furniza-o și Algoritmul Minimax, dar într-un timp mai scurt, încărcăt realizează o retezare a unor ramuri ale arborelui care nu pot influența decizia finală. Principiul general al acestei tehnici constă în a considera un nod oarecare n al arborelui, astfel încât jucătorul poate alege să facă o mutare la acel nod. Dacă același jucător dispune de o alegere mai avantajoasă, m , fie la nivelul nodului părinte al lui n , fie în orice punct de decizie aflat mai sus în arbore, atunci n nu va fi niciodată atins în timpul jocului. Prin urmare, de îndată ce, în urma examinării unora dintre descendenții nodului n , ajungem să deținem suficientă informație relativ la acesta, îl putem înălătura.

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

prost pe care îl poate obține MAX, presupunând că MIN joacă perfect. În mod similar, dacă MIN are valoarea beta 6, el nu mai trebuie să ia în considerație nici un nod de tip MAX situat sub el care are valoarea 6 sau o valoare mai mare decât acest număr. Cele două reguli pentru încheierea căutării, bazată pe valori alpha și beta, pot fi formulate după cum urmează [5]:

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

Dacă, referitor la o poziție, se arată că valoarea corespunzătoare ei se află în afara intervalului alpha-beta, atunci această informație este suficientă pentru a ști că poziția respectivă nu se află de-a lungul *variației principale*, chiar dacă nu este cunoscută valoarea exactă corespunzătoare ei. Cunoașterea valorii exacte a unei poziții este necesară numai atunci când această valoare se află între alpha și beta.

Din punct de vedere formal, putem defini o *valoare de tip minimax* a unui nod intern, P , $V(P, \text{alpha}, \text{beta})$, ca fiind “*suficient de bună*” dacă satisfac următoarele cerințe:

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

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

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

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

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

1. Începe din poziția a .
2. Mutare la b .
3. Mutare la d .
4. Alege valoarea maximă a succesorilor lui d , ceea ce conduce la $V(d) = 4$.
5. Întoarce-te în nodul b și execută o mutare de aici la e .
6. Ia în considerație primul succesor al lui e a cărui valoare este 5. În acest moment, MAX, a cărui mutare urmează, are garanțiată, aflându-se în poziția e , cel puțin valoarea 5, indiferent care ar fi celealte alternative plecând din e . Această informație este suficientă pentru ca MIN să realizeze că, la nodul b , alternativa e este inferioară alternativei d . Această concluzie poate fi trasă fără a cunoaște valoarea *exactă* a lui e . Pe această bază, cel de-al doilea succesor al lui e poate fi neglijat, iar nodului e i se poate atribui valoarea *aproximativă* 5.

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

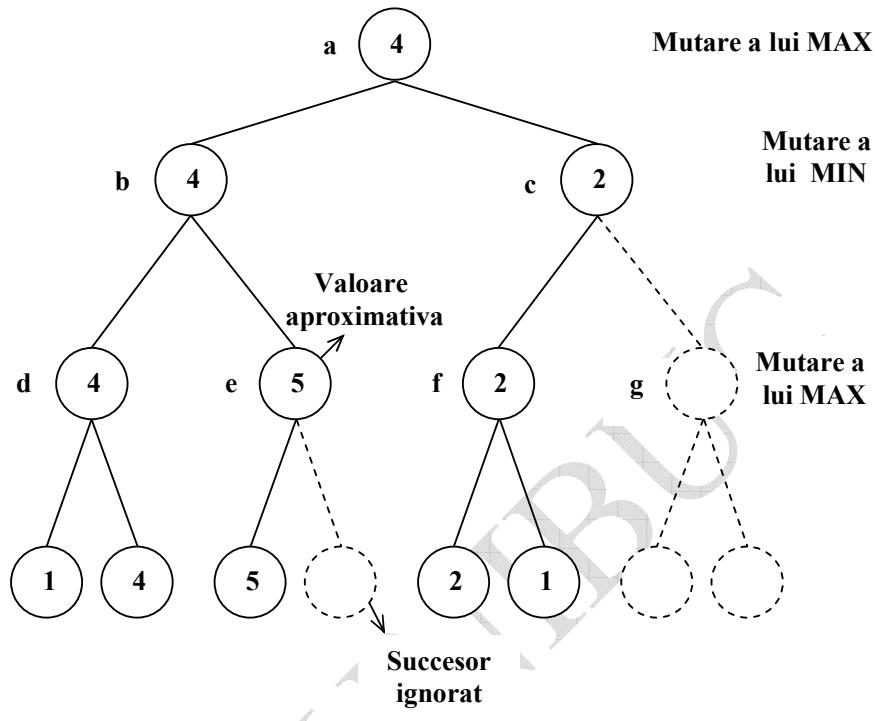


Fig. 3.2

Căutarea de tip alpha-beta retează nodurile figurate în mod discontinuu. Ca rezultat, câteva dintre valorile intermediare nu sunt exacte (nodurile *c*, *e*), dar aproximările făcute sunt suficiente pentru a determina atât valoarea corespunzătoare rădăcinii, cât și variația principală, în mod exact. Un alt exemplu [5] de aplicare a Algoritmului Alpha-Beta este cel din Fig. 3.4. Astfel, corespunzător spațiului de stări din Fig. 3.3, alpha-beta retezarea produce arborele de căutare din Fig. 3.4, în care stările fără numere nu sunt evaluate.

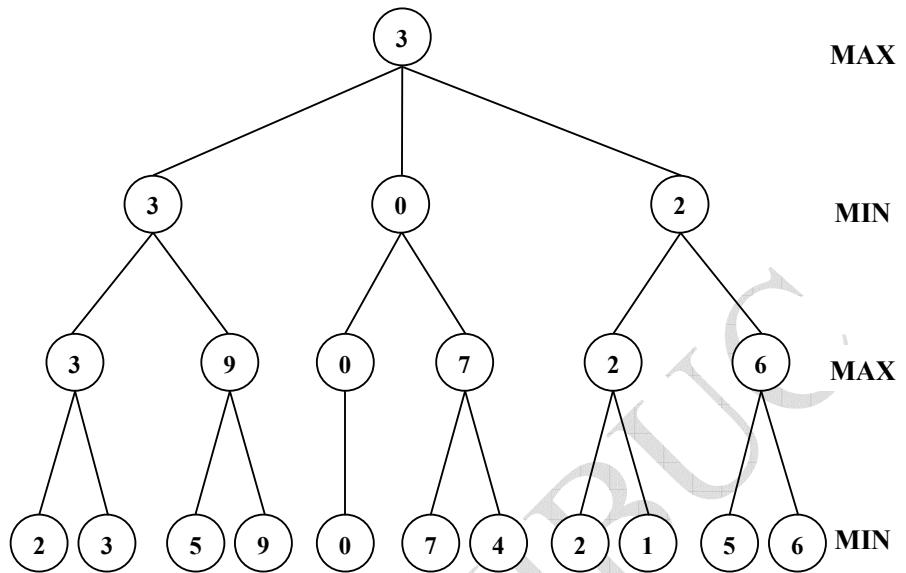


Fig. 3.3

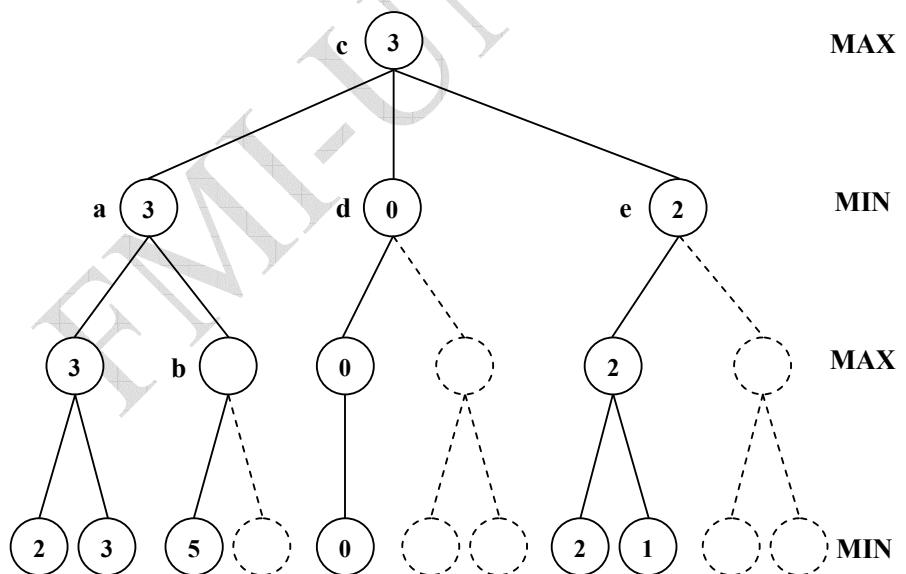


Fig. 3.4

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

3.3.1. Implementare în Prolog

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

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

unde **PozBuna** reprezintă un succesor “suficient de bun” al lui **Poz**, astfel încât valoarea sa, **Val**, satisfacă cerințele (1):

$$\text{Val} = V(\text{Poz}, \text{Alpha}, \text{Beta})$$

Procedura

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

găsește, în lista **ListăPoz**, o poziție suficient de bună, **PozBuna**, astfel încât valoarea de tip minimax, **Val**, a lui **PozBuna**, reprezintă o aproximare suficient de bună relativ la **Alpha** și **Beta**.

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

Relația

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

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

O implementare a Algoritmului Alpha-Beta este realizată de următorul program Prolog, preluat, de asemenea, din [1]:

Algoritmul Alpha-Beta

```

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

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

%nu există alt candidat

destuldebun([],_,_,Poz,Val,Poz,Val) :- !.
destuldebun(_,Alpha,Beta,Poz,Val,Poz,Val) :-
    %atingere limită superioară
    mutare_min(Poz), Val > Beta, !;
    %atingere limită inferioară
    mutare_max(Poz), Val < Alpha, !.
destuldebun(ListaPoz,Alpha,Beta,Poz,Val,PozBuna,
ValBuna) :-

```

```

%rafinare limite
limitenoi(Alpha,Beta,Poz,Val,AlphaNou,BetaNou) ,
limitarebuna(ListaPoz,AlphaNou,BetaNou,Poz1 ,
                  Val1) ,
maibine(Poz,Val,Poz1,Val1,PozBuna,ValBuna) .

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

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

```

3.3.2. Considerații privitoare la eficiență

Eficiența Algoritmului Alpha-Beta depinde de *ordinea în care sunt examinați succesorii*. Este preferabil să fie examinați mai întâi succesorii despre care se crede că ar putea fi cei mai buni. În mod evident, acest lucru nu poate fi realizat în întregime. Dacă el ar fi posibil, funcția care ordonează succesorii ar putea fi utilizată pentru a se juca un

joc perfect. În ipoteza în care această ordonare ar putea fi realizată, s-a arătat că Algoritmul Alpha-Beta nu trebuie să examineze, pentru a alege cea mai bună mutare, decât $O(b^{d/2})$ noduri, în loc de $O(b^d)$, ca în cazul Algoritmului Minimax. Aceasta arată că factorul de ramificare efectiv este \sqrt{b} în loc de b – în cazul jocului de şah 6, în loc de 35. Cu alte cuvinte, Algoritmul Alpha-Beta poate “privi înainte” la o adâncime dublă față de Algoritmul Minimax, pentru a găsi același cost.

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

Așa cum se remarcă în [9], toate rezultatele cu privire la complexitatea jocurilor, ca și a problemelor de căutare, în general, se bazează pe un *model arborescent idealizat*. Spre exemplu, modelul folosit pentru Algoritmul Alpha-Beta presupune că toate nodurile au

același factor de ramificare, b ; că toate drumurile ating limita de adâncime fixată, d ; că evaluările frunzelor sunt distribuite în mod aleator de-a lungul ultimului nivel al arborelui. Această ultimă presupunere are cel mai serios neajuns: dacă o mutare din partea superioară a arborelui reprezintă o greșală dezastroasă, atunci majoritatea descendenților săi vor părea neavantajoși jucătorului care a făcut greșala. Valoarea unui nod pare a fi strâns corelată cu valorile nodurilor frate ale acestuia. Gradul de corelare depinde foarte mult de fiecare joc în parte și de poziția particulară corespunzătoare rădăcinii. Astfel, cercetările legate de jocuri conțin și o componentă de știință empirică, care în mod inevitabil eludează puterea analizei matematice.

3.4. Un exemplu. Jocul X și 0

Pentru a ilustra folosirea celor doi algoritmi minimax și alpha-beta, considerăm drept exemplu aşa-numitul **Joc X și 0**, care face parte din categoria jocurilor între două persoane, cu informație perfectă (completă).

Jocul X și 0

Se consideră un tablou pătratic de dimensiune 3×3 , ale cărui câmpuri sunt marcate alternativ de către doi jucători. Unul dintre jucători marchează cu X, iar celălalt cu 0. Câștigă acel jucător care marchează primul, cu simbolul său și în întregime, fie o linie orizontală,

fie una verticală (coloană), fie o linie diagonală a tabloului. Să se determine jucătorul câștigător și configurația finală a tabloului.

Pentru rezolvarea acestei probleme vom defini funcția de evaluare conform [7]. Pentru o poziție p , definim $e(p)$ astfel:

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

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

	0	
	X	

Fig. 3.5

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

st(J,Tablou,N,MAX)

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

Programul 3.1

```
%jocul X si 0
%predicate specifice pentru jocul X si 0
juc_opus(x,0).
juc_opus(0,x).

%initial, tabloul este complet nemarcat
tablou_initial([gol,gol,gol,gol,gol,gol,gol,gol,
                gol]). 

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

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

```

tablou_final([_,C,_,'_','_','_','_'],C):-C\==gol,!.
tablou_final([_,_,C,_,'_','_','_','_'],C):-C\==gol,!.
tablou_final([C,_,'_','_','_','_','_'],C):-C\==gol,!.
tablou_final([_,_,C,'_','_','_'],C):-C\==gol,!.
tablou_final([_,_,C,_,'_'],C):-C\==gol,!.

tablou_final(Tablou,gol):- \+ (membru(gol,Tablou)).

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

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

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

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

%definim functia de evaluare statica

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

staticval(st(_,Tablou,_,MAX),Val):-
    juc_opus(MAX,MIN),

```

```

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

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

nr_lin_deschise(XO,L_D,Tablou) :-
    linial(Tablou,L1,XO),
    linia2(Tablou,L2,XO),
    linia3(Tablou,L3,XO),
    coloana1(Tablou,C1,XO),
    coloana2(Tablou,C2,XO),
    coloana3(Tablou,C3,XO),
    diagonalal1(Tablou,D1,XO),
    diagonalal2(Tablou,D2,XO),
    L_D is L1+L2+L3+C1+C2+C3+D1+D2.

membru_1([],_).
membru_1([H|T],XO) :-
    membru(H,[XO,gol]),membru_1(T,XO).

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

```

```

linial([X,Y,Z|_],1,XO):-membru_1([X,Y,Z],XO),!.
linial(_,0,_).

linia2([_,_,_,X,Y,Z|_],1,XO):-membru_1([X,Y,Z],XO),!.
linia2(_,0,_).

linia3([_,_,_,_,_,X,Y,Z],1,XO):-
    membru_1([X,Y,Z],XO),!.
linia3(_,0,_).

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

coloana1([X,_,_,Y,_,_,Z|_],1,XO):-
    membru_1([X,Y,Z],XO),!.
coloana1(_,0,_).

coloana2([_,X,_,_,Y,_,_,Z|_],1,XO):-
    membru_1([X,Y,Z],XO),!.
coloana2(_,0,_).

coloana3([_,_,X,_,_,Y,_,_,Z],1,XO):-
    membru_1([X,Y,Z],XO),!.
coloana3(_,0,_).

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

diagonalal([X,_,_,_,Y,_,_,_,Z],1,XO):-
    membru_1([X,Y,Z],XO),!.
```

```

diagonala1(_,0,_).

diagonala2([_,_,X,_,Y,_,Z|_],1,XO) :-
    membru_1([X,Y,Z], XO), !.

diagonala2(_,0,_).

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

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

%algoritmul minimax

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

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

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

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

%algoritmul alpha-beta

alphabeta(Poz,Alpha,Beta,PozBuna,Val) :-
    mutari(Poz,ListaPoz), !,

```

```

limitarebuna(ListaPoz,Alpha,Beta,PozBuna,
            Val) ;
staticval(Poz,Val) .

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

destuldebun([],_,_,Poz,Val,Poz,Val):-! .
destuldebun(_,Alpha,Beta,Poz,Val,Poz,Val):-!
    mutare_min(Poz),Val>Beta,!;
    mutare_max(Poz),Val<Alpha,!.

destuldebun(ListaPoz,Alpha,Beta,Poz,Val,PozBuna,
            ValBuna) :-
    limitenoi(Alpha,Beta,Poz,Val,AlphaNou,BetaNou),
    limitarebuna(ListaPoz,AlphaNou,BetaNou,Poz1,
                Vall),
    maibine(Poz,Val,Poz1,Vall,PozBuna,ValBuna) .

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

%predicate specifice pentru jocul x si 0

x_si_o_minimax:-
```

```

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

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

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

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

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

joc_minimax(st(_,Tablou,_,MAX)) :-
```

```

tablou_final(Tablou,J),!,anunt_castigator(J,MAX).

joc_minimax(st(J,Tablou,N,MAX)):-  

    MAX\==J,! ,juc_opus(J,J_O),  

    mutare_MIN(Tablou,Tablou_urM,J),nl,  

    afis_tablou(Tablou_urM),  

    joc_minimax(st(J_O,Tablou_urM,N,MAX)).  

joc_minimax(st(J,Tablou,N,MAX)):-  

    MAX==J,juc_opus(J,J_O),  

    minimax(st(J,Tablou,N,MAX),  

    st(J_O,Tablou_urM,_,MAX),_),  

    nl,write('Mutarea mea este: '),nl,nl,  

    afis_tablou(Tablou_urM),nl,  

    joc_minimax(st(J_O,Tablou_urM,N,MAX)).  

joc_alpha_beta(st(_,Tablou,_,MAX)):-  

    tablou_final(Tablou,J),!,  

    anunt_castigator(J,MAX).

joc_alpha_beta(st(J,Tablou,N,MAX)):-  

    MAX\==J,! ,juc_opus(J,J_O),  

    mutare_MIN(Tablou,Tablou_urM,J),nl,  

    afis_tablou(Tablou_urM),  

    joc_alpha_beta(st(J_O,Tablou_urM,N,MAX)).  

joc_alpha_beta(st(J,Tablou,N,MAX)):-  

    MAX==J,juc_opus(J,J_O),  

    alphabeta(st(J,Tablou,N,MAX),  

    -500,500,st(J_O,Tablou_urM,_,MAX),_),  

    nl,write('Mutarea mea este: '),nl,nl,  

    afis_tablou(Tablou_urM),nl,  

    joc_alpha_beta(st(J_O,Tablou_urM,N,MAX)).  


```

```

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

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

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

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

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

```

```

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

scrie_elem(X) :- X==gol, !, write('.');
    X==x, !, write('X');
    write('0').

```

Interogarea Prologului se face apelând predicatele **x_si_o_minimax** dacă se dorește folosirea algoritmului minimax și respectiv **x_si_o_alpha_beta** pentru folosirea algoritmului alpha-beta. Iată niște exemple de execuție ale programului:

1. Unul dintre jucători câștigă

```

?- x_si_o_minimax.
Incepe jocul...
Vrei sa fii cu X? (da/nu)
|: nu.
Adancimea: 6.

```

```

1 2 3
1 ...
2 ...
3 ...

```

Mutarea mea este:

1	2	3
1	.	.
2	.	X
3	.	.

Trebuie sa marchezi.

Specifică campul pe care îl vei marca.

Linia: 1.

Coloana: 2.

1	2	3
1	.	0
2	.	X
3	.	.

Mutarea mea este:

1	2	3
1	.	0
2	.	X
3	.	X

Trebuie sa marchezi.

Specifică campul pe care îl vei marca.

Linia: 1.

Coloana: 1.

1	2	3
1	0	0
2	.	X
3	.	X

Mutarea mea este:

1	2	3
1	0	0
2	.	X
3	.	X

Trebuie sa marchezi.

Specifică campul pe care îl vei marca.

Linia: 3.

Coloana: 1.

1	2	3
1	0	0 X
2	.	X .
3	0	. X

Mutarea mea este:

1	2	3
1	0	0 X
2	.	XX
3	0	. X

Ai pierdut!

Yes

2. Remiză

?- x_si_o_minimax.
 Incepe jocul...
 Vrei sa fii cu X? (da/nu)
 |; da.

Adancimea: 6.

1	2	3
1	...	
2	...	
3	...	

Trebuie sa marchezi.
 Specifica campul pe care il vei marca.
 Linia: 2.
 Coloana: 2.

1	2	3
1	...	
2	.	X .
3	...	

Mutarea mea este:

1	2	3
1	...	
2	.	X .
3	..	0

Trebuie sa marchezi.

Specifică campul pe care îl vei marca.

Linia: 1.

Coloana: 1.

1	2	3
1	X	.
2	.	X
3	.	0

Mutarea mea este:

1	2	3
1	X	.
2	.	X
3	0	.

Trebuie sa marchezi.

Specifică campul pe care îl vei marca.

Linia: 3.

Coloana: 2.

1	2	3
1	X	.
2	.	X
3	0	X

Mutarea mea este:

1	2	3
1	X	0
2	.	X
3	0	X

Trebuie sa marchezi.

Specifică campul pe care îl vei marca.

Linia: 2.

Coloana: 1.

1	2	3
1	X	0
2	X	X
3	0	X

Mutarea mea este:

1	2	3
1	X	0
2	X	X
3	0	X

Trebuie sa marchezi.

Specifică campul pe care îl vei marca.

Linia: 1.

Coloana: 3.

1	2	3
1	X	0
2	X	X
3	0	X

Jocul s-a încheiat cu remiza

yes

FMI-UNIBUC

CAPITOLUL 4

ASPECTE ALE REPREZENTĂRII CUNOȘTINȚELOR

Pentru a rezolva problemele complexe întâlnite în domeniul inteligenței artificiale este nevoie atât de o mare cantitate de cunoștințe, cât și de mecanisme specifice pentru manipularea acestor cunoștințe cu scopul găsirii de soluții ale unor noi probleme. Cele două tipuri diferite de entități pe care le tratează diversele modalități de reprezentare a cunoștințelor sunt:

- *Faptele* - reprezentând adevăruri într-o lume relevantă. Acestea sunt entitățile pe care dorim să le reprezentăm.
- *Reprezentări ale faptelor într-un anumit formalism*. Acestea sunt entitățile care vor putea fi efectiv manipulate.
Structurarea acestor entități se poate face la două niveluri diferite:
 - *Nivelul cunoștințelor*, nivel la care sunt descrise faptele (inclusiv comportamentul fiecărui agent și scopurile lui curente).
 - *Nivelul simbolurilor*, la care reprezentările obiectelor de la nivelul cunoștințelor sunt definite în termenii unor simboluri care pot fi manipulate de către programele de calculator.

Un bun sistem de reprezentare a cunoștințelor într-un anumit domeniu ar trebui să posede [8] următoarele patru proprietăți:

- *Adecvare în reprezentare* - abilitatea de a reprezenta toate tipurile de cunoștințe care sunt necesare într-un anumit domeniu.

- *Adecvare inferențială* - abilitatea de a manipula structurile reprezentătionale într-un asemenea mod încât să poată fi deriveate structuri noi, corespunzătoare cunoștințelor noi deduse din cele vechi.
- *Eficiența inferențială* - abilitatea de a încorpora în structura de cunoștințe informații adiționale care să poată fi folosite pentru a canaliza atenția mecanismului de inferență în direcțiile cele mai promițătoare.
- *Eficiența în achiziție* - abilitatea de a achiziționa ușor informații noi. În mod ideal, însusi programul ar trebui să poată controla achiziția de cunoștințe.

Nici un sistem unic nu poate optimiza toate aceste caracteristici, nu în ultimul rând datorită tipurilor extrem de diverse de cunoștințe care există, majoritatea programelor bazându-se pe tehnici multiple.

4.1. Tipuri de cunoștințe

4.1.1. Cunoștințe relationale simple

Cea mai simplă modalitate de reprezentare a faptelor declarative constă în folosirea unei mulțimi de relații de același tip cu cele utilizate în sistemele de baze de date. Un exemplu de sistem relational este cel din Fig. 4.1. Cunoștințele relationale din acest tabel corespund unei mulțimi de attribute și de valori asociate, care împreună descriu obiectele bazei de cunoștințe.

Student	Vârstă	An de studiu	Note la informatică
Popescu Andrei	18	I	8-9
Ionescu Maria	18	I	9-10
Hristea Oana	20	I	7-8
Pârvu Ana	19	II	8-9
Savu Andrei	19	II	7-8
Popescu Ana	20	III	9-10

Fig. 4.1

Această reprezentare este simplă deoarece, în sine, nu posedă și nu furnizează capacitatea de inferență. Dar cunoștințele reprezentate în acest mod pot constitui input-ul adecvat pentru motoare de inferență mult mai puternice. Sistemele de baze de date sunt proiectate tocmai cu scopul de a furniza suportul necesar cunoștințelor relaționale.

4.1.2. Cunoștințe care se moștenesc

Cunoștințele despre obiecte, atributele lor și valorile acestora sunt adesea mult mai complexe decât permite modul de reprezentare din Fig. 4.1. În particular, este posibil ca reprezentarea de bază să fie îmbogățită cu mecanisme de inferență care operează asupra structurii reprezentării. Pentru ca această modalitate de reprezentare să fie eficientă, structura trebuie proiectată în aşa fel încât ea să corespundă mecanismelor de inferență dorite. Una dintre cele mai utilizate forme de inferență este *moștenirea proprietăților*, prin care elemente aparținând anumitor clase *moștenesc atrbute și valori* provenite de la clase mai generale, în care sunt incluse. Pentru a admite moștenirea proprietăților, *obiectele* trebuie

să fie organizate în *clase*, iar clasele trebuie să fie aranjate în cadrul unei ierarhii. În Fig. 4.2 sunt reprezentate cunoștințe legate de jocul de fotbal, cunoștințe organizate într-o structură de acest tip. În această reprezentare, liniile desemnează attribute. Nodurile figurate prin dreptunghiuri reprezintă obiecte și valori ale atributelor obiectelor. Aceste valori pot fi, la rândul lor, privite ca obiecte având attribute și valori și.a.m.d.. Săgețile corespunzătoare liniilor sunt orientate de la un obiect la valoarea lui (de-a lungul liniei desemnând atributul corespunzător).

În acest exemplu, structura corespunzătoare reprezintă o *structură de tip “slot-and-filler”*. Ea mai poate fi privită și ca o *rețea semantică* sau ca o colecție de *cadre*. În cel din urmă caz, fiecare cadru individual reprezintă colecția atributelor și a valorilor asociate cu un anumit nod. O diferențiere exactă a acestor tipuri de reprezentări este greu de făcut datorită marii flexibilități care există în reprezentarea cunoștințelor. În general, termenul de sistem de cadre implică existența unei mai mari structurări a atributelor și a mecanismelor de inferență care le pot fi aplicate decât în cazul rețelelor semantice. Vom reveni asupra ambelor modalități de reprezentare a cunoștințelor.

În exemplul din Fig. 4.2., toate obiectele și majoritatea atributelor care intervin corespund domeniului sportiv al jocului de fotbal și nu au o semnificație generală. Singurele două excepții sunt atributul *isa*, utilizat pentru a desemna *inclusiunea între clase* și atributul *instanțiere*, folosit pentru a arăta *apartenența la o clasă*. Aceste două attribute, extrem de folosite, se află la baza *măștenirii proprietăților* ca tehnică de inferență.

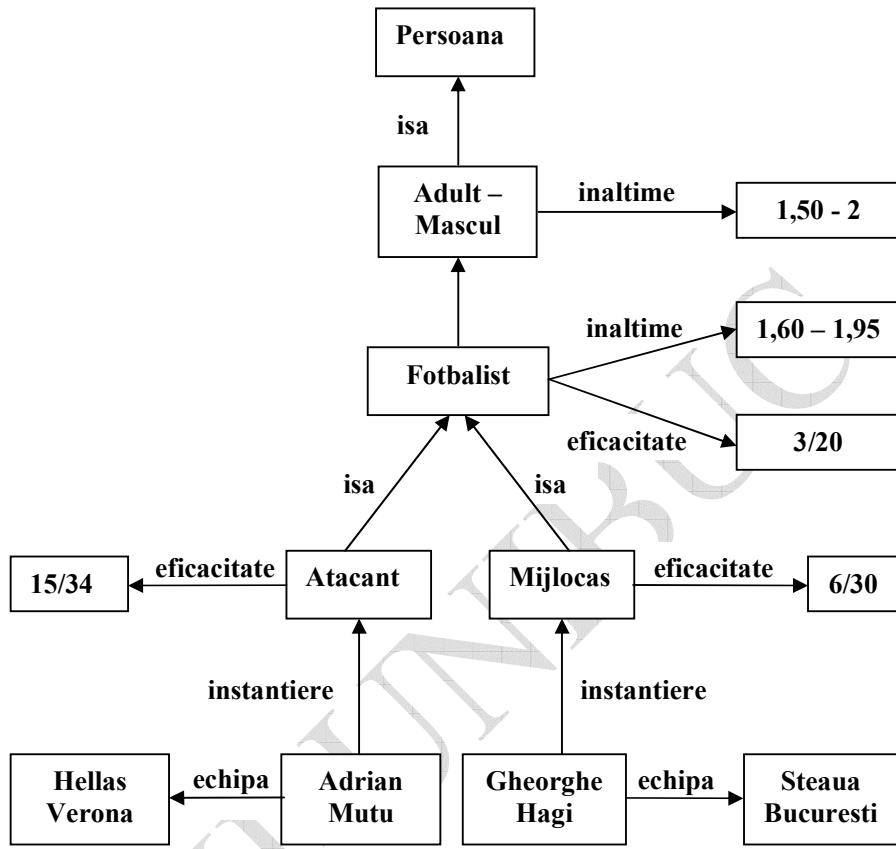


Fig. 4.2

Utilizând această tehnică de inferență, *baza de cunoștințe* poate asigura atât regăsirea faptelor care au fost memorate în mod explicit, precum și a faptelor care derivă din cele memorate în mod explicit, ca în următorul exemplu:

$$\text{eficacitate}(\text{Adrian_Mutu}) = 15/34$$

Pentru a găsi răspunsul la această interogare, întrucât nu există nici o valoare pentru eficacitate memorată în mod explicit corespunzător lui

Adrian_Mutu, a fost urmat atributul *instanțiere* până la *Atacant* și a fost extrasă valoarea memorată acolo. Se poate acum observa una dintre caracteristicile negative ale moștenirii proprietăților, și anume aceea că această tehnică de inferență poate produce valori implicate care nu sunt garantate a fi corecte, dar care reprezintă cele mai bune aproximări în lipsa unor informații exacte. Această tehnică de inferență continuă să fie printre cele mai folosite.

4.1.3. Cunoștințe inferențiale

Moștenirea proprietăților este o formă de inferență puternică, dar nu este suficientă pentru a construi un sistem de reprezentare complet, care, de cele mai multe ori, combină mai multe tehnici de reprezentare a cunoștințelor.

Puterea logicii tradiționale este adesea utilă pentru a se descrie toate inferențele necesare. Desigur, astfel de cunoștințe nu sunt utile decât în prezența unei proceduri de inferență care să le poată exploata. Procedura de inferență necesară în acest caz este una care implementează regulile logice de inferență standard. Există multe asemenea proceduri, dintre care unele fac raționamente de tipul “*înainte*”, de la fapte date către concluzii, iar altele raționează “*înapoi*”, de la concluziile dorite la faptele date. Una dintre procedurile cele mai folosite de acest tip este *rezoluția*, care folosește strategia contradicției.

În general, logica furnizează o structură puternică în cadrul căreia sunt descrise legăturile dintre valori. Ea se combină adesea cu un alt limbaj puternic de descriere, cum ar fi o ierarhie de tip *isa*.

4.1.4. Cunoștințe procedurale

Reprezentarea cunoștințelor descrisă până în prezent s-a concentrat asupra faptelor statice, declarative. Un alt tip de cunoștințe extrem de utile sunt *cunoștințele procedurale sau operaționale*, care specifică ce anume trebuie făcut și când.

Cunoștințele procedurale pot fi reprezentate în programe în diverse moduri. Cea mai simplă modalitate este, desigur, cea sub formă de cod, într-un anumit limbaj de programare. În acest caz, mașina folosește cunoștințele atunci când execută codul pentru a efectua o anumită sarcină. Acest mod de reprezentare a cunoștințelor procedurale nu este însă cel mai fericit din punctul de vedere al adecvării inferențiale, precum și al eficienței în achiziție. Din această cauză s-au căutat alte modalități, diferite, de reprezentare a cunoștințelor procedurale, astfel încât acestea să poată fi manipulate relativ ușor atât de către oameni, cât și de către alte programe.

Cea mai folosită tehnică de reprezentare a cunoștințelor procedurale în programele de inteligență artificială este aceea a utilizării *regulilor de producție*. Atunci când sunt îmbogățite cu informații asupra felului în care trebuie să fie folosite, regulile de producție sunt mai procedurale decât alte metode existente de reprezentare a cunoștințelor.

Regulile de producție, numite și reguli de tip **if-then**, sunt instrucțiuni condiționale, care pot avea diverse interpretări, cum ar fi:

- **if** precondiție P **then** concluzie C
- **if** situație S **then** acțiune A

- **if** condițiile C1 și C2 sunt verificate **then** condiția C nu este verificată.

Regulile de producție sunt foarte utilizate în proiectarea *sistemelor expert*. Vom reveni asupra lor, precum și a implementării lor în limbajul Prolog.

A face o distincție clară între cunoștințele declarative și cele procedurale este foarte dificil. Diferența esențială între ele este dată de modul în care cunoștințele sunt folosite de către procedurile care le manipulează.

4.2. Clase de metode pentru reprezentarea cunoștințelor

Principalele tipuri de reprezentări ale cunoștințelor sunt reprezentările *bazate pe logică* și cele de tip “*slot-filler*” (“deschizătură-umplutură”).

Reprezentările bazate pe logică aparțin unor două mari categorii, în funcție de instrumentele folosite în reprezentare, și anume:

- Logica - mecanismul principal îl constituie inferența logică.
- Regulile (folosite, de pildă, în sistemele expert) - principalele mecanisme sunt “*înlănțuirea înainte*” și “*înlănțuirea înapoi*”.

O regulă este similară unei implicații logice, dar nu are o valoare proprie (regulile sunt *aplicate*, ele nu au una dintre valorile true sau false).

Reprezentările de tip slot-filler folosesc două categorii diferite de structuri:

- Rețele semantice și grafuri conceptuale - o reprezentare “distribuită” (concepe legate între ele prin diverse relații). Principalul mecanism folosit este *căutarea*.
- Cadre și scripturi - o reprezentare structurată (grupuri de concepe și relații); sunt foarte utile în reprezentarea tipicității. Principalul mecanism folosit este *împerecherea* (potrivirea) *șablonelor* (tiparelor).

4.3. Reprezentarea cunoștințelor și sistemele expert

Un *sistem expert* este un program care se comportă ca un expert într-un domeniu relativ restrâns. Caracteristica majoră a sistemelor expert, numite și *sisteme bazate pe cunoștințe*, este aceea că ele se bazează pe cunoștințele unui expert uman în domeniul care este studiat. Mai exact, ele se sprijină pe cunoștințele expertului uman asupra strategiilor de rezolvare a problemelor specifice domeniului. Astfel, la baza sistemelor expert se află utilizarea în rezolvarea problemelor a unor mari cantități de cunoștințe specifice domeniului.

Sistemele expert¹⁶ trebuie să poată rezolva probleme care necesită cunoștințe într-un anumit domeniu. Prin urmare, ele trebuie să posede aceste cunoștințe într-o anumită formă. Din această cauză ele se mai numesc și sisteme bazate pe cunoștințe. Nu toate sistemele bazate pe cunoștințe constituie însă sisteme expert. Un sistem expert trebuie, în plus, să fie capabil să *explice* utilizatorului comportamentul său și

¹⁶ A căror tratare se face aici conform [1].

deciziile luate, la fel cum o fac experții umani. Această caracteristică referitoare la generarea explicațiilor este necesară în special în domeniile în care intervine *incertitudinea*, cum ar fi diagnosticarea medicală. Numai în acest fel utilizatorul poate detecta o posibilă fisură în raționamentul sistemului.

O caracteristică suplimentară care este adesea cerută sistemelor expert este, prin urmare, abilitatea de a trata incertitudinea sau starea de a fi incomplet. Astfel, informațiile asupra problemei care trebuie rezolvată pot fi incomplete sau de natură să nu inspire încredere, iar relațiile din domeniul problemei pot fi aproximative. Spre exemplu, un anumit medicament poate genera anumite probleme, dar *cel mai adesea* nu o face.

Pentru a construi un sistem expert este necesară, în general, definirea următoarelor două funcții:

- o *funcție de rezolvare a problemei*, funcție capabilă să folosească cunoștințele specifice domeniului și care trebuie să poată trata incertitudinea.
- o *funcție de interacțiune cu utilizatorul*, care asigură și generarea de explicații asupra intențiilor sistemului și a deciziilor luate de acesta în timpul și după procesul de rezolvare a problemei.

Fiecare dintre aceste funcții poate fi foarte complicată și depinde în mod nemijlocit de domeniul de aplicație și de cerințele practice care pot să apară.

4.3.1. Structura de bază a unui sistem expert

Un sistem expert conține trei module principale, și anume:

- o bază de cunoștințe;
- un motor de inferență;
- o interfață cu utilizatorul.

Baza de cunoștințe cuprinde cunoștințele specifice unui domeniu, inclusiv fapte simple referitoare la domeniul de aplicație, reguli care descriu relațiile și fenomenele proprii domeniului și, eventual, metode, euristică și idei pentru rezolvarea problemelor domeniului.

Motorul de inferență utilizează în mod activ cunoștințele din baza de cunoștințe.

Interfața cu utilizatorul asigură comunicarea dintre utilizator și sistem, oferind utilizatorului o privire asupra procesului de rezolvare a problemei executat de către motorul de inferență. De cele mai multe ori motorul de inferență și interfața sunt privite ca un unic modul, numit *shell* (în engleză).

Structura de bază a unui sistem expert este prezentată în Fig. 4.3:

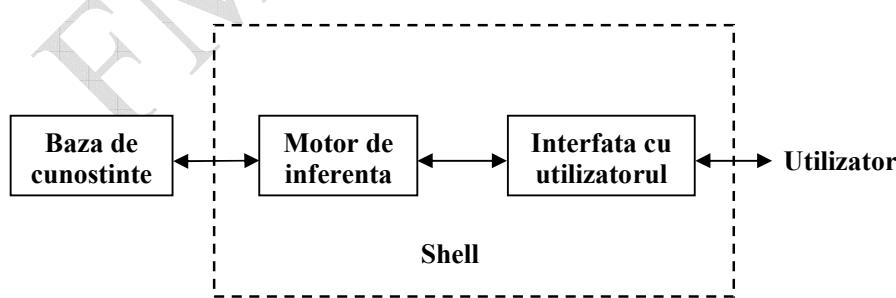


Fig. 4.3

Schema din Fig. 4.3 separă cunoștințele de algoritmii care folosesc aceste cunoștințe. Această separare este convenabilă din următoarele motive: baza de cunoștințe depinde în mod clar de aplicație. Pe de altă parte, învelișul este, în principiu, independent de domeniu. O modalitate de a dezvolta sisteme expert pentru diverse aplicații, în aceste condiții, constă din dezvoltarea unui *shell* care poate fi folosit în mod universal, cu utilizarea unei baze de cunoștințe diferite corespunzător fiecărei aplicații. Desigur, toate bazele de cunoștințe utilizate vor trebui să se conformeze unui același formalism care este înțeles de către *shell*. În practică, această abordare nu va avea succes decât în cazul în care domeniile de aplicație sunt extrem de similare, dar, chiar dacă modificări ale învelișului vor fi necesare, atunci când se trece de la un domeniu la altul, principiile de bază pot fi reținute.

4.3.2. Reprezentarea cunoștințelor cu reguli if-then

Regulile de tip *if-then*, numite și *reguli de producție*, constituie o formă naturală de exprimare a cunoștințelor și au următoarele caracteristici suplimentare:

- *Modularitate*: fiecare regulă definește o cantitate de cunoștințe relativ mică și independentă de celelalte.
- *Incrementabilitate*: noi reguli pot fi adăugate bazei de cunoștințe în mod relativ independent de celealte reguli.
- *Modificabilitate* (ca o consecință a modularității): regulile vechi pot fi modificate relativ independent de celealte reguli.
- Susțin *transparența* sistemului.

Această ultimă proprietate este o caracteristică importantă a sistemelor expert. Prin *transparența sistemului* se înțelege abilitatea sistemului de a explica deciziile și soluțiile sale. Regulile de producție facilitează generarea răspunsului pentru următoarele două tipuri de întrebări ale utilizatorului:

- întrebare de tipul “*cum*”: *Cum* ai ajuns la această concluzie?
- întrebare de tipul “*de ce*”: *De ce* te interesează această informație?

Regulile de tip **if-then** adesea definesc relații logice între concepțele aparținând domeniului problemei. Relațiile pur logice pot fi caracterizate ca aparținând așa-numitelor *cunoștințe categorice*, adică acelor cunoștințe care vor fi întotdeauna adevărate. În unele domenii însă, cum ar fi diagnosticarea în medicină, predomină cunoștințele “moi” sau *probabiliste*. În cazul acestui tip de cunoștințe, regularitățile empirice sunt valide numai până la un anumit punct (adesea, dar nu întotdeauna). În astfel de cazuri, regulile de producție pot fi modificate prin adăugarea la interpretarea lor logică a unei calificări de verosimilitate, obținându-se reguli de forma următoare:

if condiție A then concluzie B cu certitudinea F

Pentru a exemplifica folosirea regulilor de producție, vom lua în considerație baza de cunoștințe din Fig. 4.4, care își propune să trateze problema scurgerii de apă în apartamentul din aceeași figură. O scurgere de apă poate interveni fie în baie, fie în bucătărie. În ambele situații, scurgerea provoacă o problemă (inundație) și în hol. Această bază de cunoștințe¹⁷ simplistă nu presupune decât existența defectelor unice: problema poate fi la baie sau la bucătărie, dar nu în ambele locuri în

¹⁷ Preluată de noi, pentru exemplificare, din [1].

același timp. Ea este reprezentată în Fig. 4.4 sub forma unei *rețele de inferență*:

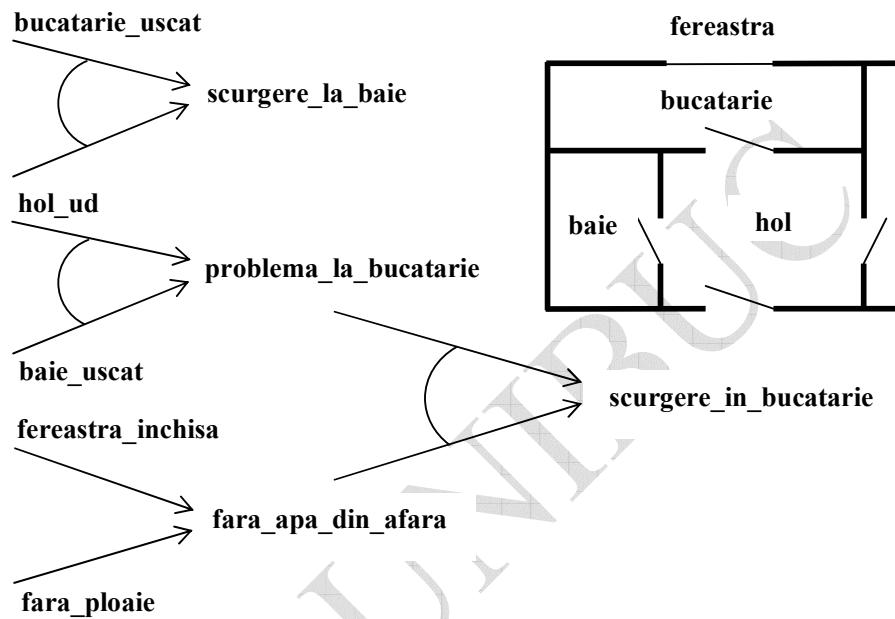


Fig. 4.4

Nodurile rețelei corespund propozițiilor, iar legăturile corespund regulilor din baza de cunoștințe. Arcele care conectează unele dintre legături indică conexiunea conjunctivă existentă între propozițiile corespunzătoare. În consecință, regula referitoare la existența unei probleme în bucătărie, în cadrul acestei rețele, este:

if hol_ud si baie_uscat then problema_la_bucatarie.

4.3.3. Înlănțuire înainte și înapoi în sistemele bazate pe reguli

Atunci când cunoștințele sunt reprezentate într-o anumită formă, este nevoie de o procedură de raționament care să tragă concluzii deriveate din baza de cunoștințe. În cazul regulilor de tip **if-then**, există două modalități de a raționa, ambele extrem de ușor de implementat în Prolog [1], și anume:

- înlănțuire înapoi (“backward chaining”);
- înlănțuire înainte (“forward chaining”).

4.3.3.1. Înlănțuirea înapoi

Raționamentul de tip “înlănțuire înapoi” pleacă de la o ipoteză și apoi parurge în sensul “înapoi” rețeaua de inferență. În cazul bazei de cunoștințe din Fig. 4.4, spre exemplu, una dintre ipotezele de la care se poate pleca este *surgere_in_bucatarie*. Pentru ca această ipoteză să fie confirmată, este nevoie ca *problema_la_bucatarie* și *fara_apă_din_afara* să fie adevărate. Prima dintre acestea este confirmată dacă se constată că holul este ud și baia este uscată. Cea de-a doua este confirmată dacă se constată, de pildă, că fereastra este închisă.

Acest tip de raționament a fost numit “înlănțuire înapoi” deoarece el urmează un lanț de reguli care pleacă de la ipoteză (*surgere_in_bucatarie*) și se îndreaptă către faptele evidente (*hol_ud*). Acest mod de a raționa este extrem de simplu de implementat în Prolog, întrucât reprezintă însuși mecanismul de raționament încorporat în acest limbaj. Cea mai simplă și mai directă modalitate de implementare este

cea care enunță regulile din baza de cunoștințe sub forma unor reguli Prolog, ca în exemplele următoare:

```
problema_la_bucatarie:-
hol_ud,
baie_uscat.

fara_apa_din_afara:-
fereastra_inchisa
;
fara_ploaie.
```

Faptele observate ca fiind evidente pot fi reprezentate sub forma unor fapte Prolog de tipul:

```
hol_ud.
baie_uscat.
fereastra_inchisa.
```

Ipoteza de la care s-a plecat poate fi acum verificată printr-o interogare a Prologului de tipul următor:

```
?- surgere_in_bucatarie.
yes
```

În această implementare, în cazul regulilor, a fost folosită însăși sintaxa limbajului Prolog. Această abordare prezintă anumite dezavantaje, printre care faptul că expertul în domeniu care utilizează baza de cunoștințe trebuie să fie familiarizat cu limbajul Prolog, întrucât

el trebuie să citească regulile, să le poată modifica și să poată specifica reguli noi. Un al doilea dezavantaj al acestei implementări este faptul că baza de cunoștințe nu se poate distinge, din punct de vedere sintactic, de restul programului. Pentru a face diferența dintre baza de cunoștințe și restul programului mai clară, sintaxa regulilor expert poate fi modificată prin folosirea unor operatori definiți de utilizator. Spre exemplu, pot fi folosiți ca operatori *if*, *then*, *and* și *or*, declarați în felul următor:

```
:- op(800,fx,if).
:- op(700,xfx,then).
:- op(300,xfy,or).
:- op(200,xfy,and).
```

Regulile pot fi scrise atunci sub forma:

```
if
    hol_ud and bucatarie_uscat
then
    scurgere_la_baie.

if
    fereastra_inchisa or fara_ploaie
then
    fara_apă_din_afara.
```

Faptele observate pot fi enunțate sub forma unei proceduri pe care o vom numi **fapta**:

```
fapta(hol_ud).
fapta(baie_uscat).
fapta(fereastra_inchisa).
```

În această nouă sintaxă este nevoie de un nou interpreter pentru reguli. Un astfel de interpreter poate fi definit sub forma procedurii

```
este_adevarat(P)
```

unde propoziția **P** este fie dată prin intermediul procedurii **fapta**, fie poate fi derivată prin utilizarea regulilor. Noul interpreter, aşa cum este el propus în [1], este următorul:

```

:- op(800,fx,if).
:- op(700,xfx,then).
:- op(300,xfy,or).
:- op(200,xfy, and).

este_adevarat(P) :-
    fapta(P).

este_adevarat(P) :-
    if Conditie then P,
    este_adevarat(Conditie).

este_adevarat(P1 and P2) :-
    este_adevarat(P1),
    este_adevarat(P2).

este_adevarat(P1 or P2) :-
    este_adevarat(P1)
    ;
    este_adevarat(P2).

```

Se observă că acest interpreter pentru reguli **if-then** de tip “înlănțuire înapoi” continuă să lucreze înapoi în maniera depth-first.

Interogarea Prologului se face acum în felul următor:

```
? : -este_adevarat(surgere_in_bucatarie).
yes
```

Principalul dezavantaj al procedurilor de inferență relativ simple prezentate până acum constă în faptul că utilizatorul trebuie să enunțe toate informațiile relevante de la început, înaintea declanșării raționamentului, sub formă de fapte. Există, prin urmare, pericolul ca utilizatorul să enunțe prea puține sau prea multe fapte. De aceea, este de preferat ca el să furnizeze informațiile în mod interactiv, în cadrul unui dialog, atunci când acest lucru devine necesar. O asemenea abordare poate fi văzută în [1].

4.3.3.2. Înlănțuirea înainte

Înlănțuirea înainte nu începe cu o ipoteză, ci face un raționament în direcție opusă, de la partea cu **if** la partea cu **then**. În cazul exemplului studiat, de pildă, după ce se observă că holul este ud iar baia este uscată, se trage concluzia că există o problemă la bucătărie.

Interpretorul pentru înlănțuirea înainte pe care îl vom prezenta aici și care este preluat din [1], presupune că regulile sunt, ca și înainte, de forma

if Conditie then Concluzie

unde **Conditie** poate fi o expresie de tipul **AND/OR**. Pentru simplitate vom continua să presupunem că regulile nu conțin variabile. Interpretorul începe cu ceea ce este deja cunoscut (și enunțat prin intermediul relației

fapta), trage toate concluziile posibile și adaugă aceste concluzii (folosind **assert**) relației **fapta**:

```

inainte:-  

    fapta_noua_dedusa(P),           %o noua fapta  

    !,  

    write('Dedus:'), write(P), nl,  

    assert(fapta (P)),  

    inainte                         %continua  

    ;  

    write('Nu mai exista fapte').   %Toate faptele  
          %au fost deduse

fapta_noua_dedusa(Concl):-  

    if Cond then Concl,             %o regula  

    not fapta(Concl),  

    fapta_compusa(Cond).          %Concluzia  
          %regulii nu este  
          %inca o fapta  
          %Conditia este  
          %adevarata?

fapta_compusa(Cond):-  

    fapta(Cond).                  %fapta simpla

fapta_compusa(Cond1 and Cond2):-  

    fapta_compusa(Cond1),          %Ambii conjuncti  

    fapta_compusa(Cond2).          %sunt adevarati

fapta_compusa(Cond1 or Cond2):-  

    fapta_compusa(Cond1)  

    ;  

    fapta_compusa(Cond2).

```

Interogarea Prologului se face în felul următor:

```
?-inainte.  
Dedus: problema_la_bucatarie  
Dedus: fara_apă_din_afara  
Dedus: scurgere_in_bucatarie  
Nu mai există fapte
```

4.3.3.3. Concluzii

Regulile **if-then** formează lanțuri de forma
 informație input →...→ informație dedusa

Acste două tipuri de informație sunt cunoscute sub diverse denumiri în literatura de specialitate, denumiri care depind, în mare măsură, de contextul în care ele sunt folosite. Informația de tip input mai poartă denumirea de *date* sau *manifestări*. Informația dedusă constituie *ipotezele* care trebuie demonstate sau *cauzele* manifestărilor sau *diagnostice* sau *explicații*.

Atât înlătuirea înainte, cât și cea înapoi presupun *căutare*, dar direcția de căutare este diferită pentru fiecare în parte. Înlătuirea înapoi execută o căutare de la scopuri înspre date, din care cauză se spune despre ea că este *orientată către scop*. Prin contrast, înlătuirea înainte caută pornind de la date înspre scopuri, fiind *orientată către date*.

Întrebarea firească care se ridică este cea referitoare la care tip de raționament este preferabil. Răspunsul depinde în mare măsură de problema dată. În general, dacă se dorește verificarea unei anumite ipoteze, atunci înlătuirea înapoi, pornindu-se de la respectiva ipoteză,

pare mai naturală. Dacă însă există o multitudine de ipoteze și nu există o anumită motivație pentru testarea cu prioritate a uneia dintre ele, atunci înlănțuirea înainte va fi preferabilă. Această metodă se recomandă și în cazul sistemelor în care datele se achiziționează în mod continuu, iar sistemul trebuie să detecteze apariția oricărei situații reprezentând o anomalie. În acest caz, fiecare schimbare în datele de intrare poate fi propagată înainte, pentru a se constata dacă ea indică vreo eroare a procesului monitorizat sau o schimbare a nivelului de performanță.

În alegerea metodei de raționament poate fi utilă însăși forma rețelei în cauză. Astfel, un număr mic de noduri de date și unul ridicat al nodurilor scop pot sugera că fiind mai adekvată înlănțuirea înainte. Un număr redus al nodurilor scop și multe noduri corespunzătoare datelor indică înlănțuirea înapoi ca fiind preferabilă. Este totuși de notat faptul că majoritatea sistemelor expert sunt infinit mai complexe și necesită o combinare a celor două tipuri de raționament, adică o combinare a înlănțuirii în ambele direcții.

4.3.4. Generarea explicațiilor

Una dintre caracteristicile regulilor de producție care fac din acestea o modalitate naturală de exprimare a cunoștințelor în cadrul sistemelor expert este faptul că ele *susțin transparența sistemului*. Prin transparența sistemului se înțelege abilitatea acestuia de a explica deciziile și soluțiile sale. Regulile de producție facilitează generarea răspunsului pentru următoarele două tipuri de întrebări ale utilizatorului:

- Întrebare de tipul “cum”: *Cum* ai ajuns la această concluzie?

- Întrebare de tipul “de ce”: *De ce* te interesează această informație?

În cele ce urmează, ne vom ocupa de primul tip de întrebare. O tratare a întrebărilor de tipul “de ce”, care necesită interacțiunea utilizatorului cu procesul de raționament, poate fi consultată în [1].

În cazul întrebărilor de tipul “cum”, explicația pe care sistemul o furnizează cu privire la modul în care a fost dedus răspunsul său constituie un *arbore de demonstrație* a modului în care concluzia finală decurge din regulile și faptele aflate în baza de cunoștințe.

Fie “ \leq ” un operator infixat. Atunci arborele de demonstrație al unei propoziții poate fi reprezentat în una dintre următoarele forme, în funcție de necesități:

1. Dacă **P** este o faptă, atunci arborele de demonstrație este **P**.
2. Dacă **P** a fost dedus folosind o regulă de forma

if Cond then P

atunci arborele de demonstrație este

P \leq DemCond

unde **DemCond** este un arbore de demonstrație a lui **Cond**.

3. Fie **P1** și **P2** propoziții ale căror arbori de demonstrație sunt **Dem1** și **Dem2**. Dacă **P** este de forma **P1 and P2**, atunci arborele de demonstrație corespunzător este **Dem1 and Dem2**. Dacă **P** este de forma **P1 or P2**, atunci arborele de demonstrație este fie **Dem1**, fie **Dem2**.

Construcția arborilor de demonstrație în Prolog este directă și se poate realiza prin modificarea predicatului **este_adevarat**, introdus în § 4.3.3.1, în conformitate cu cele trei cazuri enunțate mai sus. Un astfel

de predicat **este_adevarat** modificat ar putea fi următorul, preluat de noi din [1]:

```
%este_adevarat(P,Dem) daca Dem constituie o
%demonstratie a faptului ca P este adevarat

:-op(800,xfx,<=).

este_adevarat(P,P) :-
    fapta(P).
este_adevarat(P,P<= DemCond) :-
    if Cond then P,
    este_adevarat(Cond,DemCond).
este_adevarat(P1 and P2, Dem1 and Dem2) :-
    este_adevarat(P1,Dem1),
    este_adevarat(P2,Dem2).
este_adevarat(P1 or P2, Dem) :-
    este_adevarat(P1,Dem)
;
    este_adevarat(P2,Dem).
```

4.3.5. Introducerea incertitudinii

Reprezentarea cunoștințelor luată în discuție până acum pleacă de la presupunerea că domeniile problemelor sunt *categorice*. Aceasta înseamnă că răspunsul la orice întrebare este fie adevărat, fie fals. Regulile care intervenau erau de aceeași natură, reprezentând așa-numite implicații categorice. Totuși, majoritatea domeniilor expert nu sunt

categorice. Atât datele referitoare la o anumită problemă, cât și regulile generale pot să nu fie certe. Incertitudinea poate fi modelată prin atribuirea unei calificări, alta decât adevărat sau fals, majorității aserțiunilor. Gradul de adevăr poate fi exprimat prin intermediul unui număr real aflat într-un anumit interval - spre exemplu, un număr între 0 și 1 sau între -5 și +5. Astfel de numere cunosc, în literatura de specialitate, o întreagă varietate de denumiri, cum ar fi *factor de certitudine, măsură a încrederii sau certitudine subiectivă*.

În cele ce urmează, vom exemplifica prin extinderea reprezentării bazate pe reguli de până acum cu o schemă simplă de incertitudine, preluată din [1]. Fiecarei propoziții i se va adăuga un număr între 0 și 1 ca *factor de certitudine*. Reprezentarea folosită va consta dintr-o pereche de formă:

Propoziție: FactorCertitudine

Această notație va fi aplicată și regulilor. Astfel, următoarea formă va defini o regulă și gradul de certitudine până la care acea regulă este validă:

If Condiție then Concluzie: Certitudine.

În cazul oricărei reprezentări cu incertitudine este necesară specificarea modului în care se combină certitudinile propozițiilor și ale regulilor. Spre exemplu, să presupunem că sunt date două propoziții **P1** și **P2** având certitudinile **c(P1)** și respectiv **c(P2)**. Atunci putem defini

$$c(P1 \text{ and } P2) = \min(c(P1), c(P2))$$

$$c(P1 \text{ or } P2) = \max(c(P1), c(P2))$$

Dacă există regula

if P1 then P2: C

cu **C** reprezentând factorul de certitudine, atunci

$$c(P2) = c(P1)*C$$

Pentru simplitate, vom presupune, în cele ce urmează, că nu există mai mult de o regulă susținând o aceeași afirmație. Dacă ar exista două astfel de reguli în baza de cunoștințe, ele ar putea fi transformate, cu ajutorul operatorului **OR**, în reguli echivalente care satisfac această presupunere. Implementarea în Prolog a unui interpretor de reguli corespunzător schemei de incertitudine descrise aici, preluată din [1], va presupune specificarea de către utilizator a estimărilor de certitudine corespunzătoare datelor observate (nodurile cel mai din stânga ale rețelei) prin relația

dat(Propozitie, Certitudine).

Iată un asemenea interpretor pentru reguli cu factor de certitudine:

```
% certitudine (Propozitie, Certitudine)
certitudine(P,Cert) :-
    dat(P,Cert).

certitudine(Cond1 and Cond2, Cert) :-
    certitudine(Cond1,Cert1),
    certitudine(Cond2,Cert2),
    minimum(Cert1,Cert2,Cert).

certitudine(Cond1 or Cond2, Cert) :-
    certitudine(Cond1,Cert1),
    certitudine(Cond2,Cert2),
    maximum(Cert1,Cert2,Cert).
```

```
certitudine(P,Cert) :-
  if Cond then P:C1,
  certitudine(Cond,C2),
  Cert is C1*C2.
```

Regulile bazei de cunoștințe studiate anterior pot fi acum rafinate ca în următorul exemplu :

```
if hol_ud and baie_uscat
then
  problema_la_bucatarie: 0.9.
```

O situație în care holul este ud, baia este uscată, bucătăria nu este uscată, fereastra nu este închisă și credem - dar nu suntem siguri - că afară nu plouă, poate fi specificată după cum urmează:

```
dat(hol_ud,1).
dat(baie_uscat,1).
dat(bucatarie_uscat,0).
dat(fara_ploaie,0.8).
dat(fereastra_inchisa,0).
```

Interogarea Prologului referitoare la o scurgere în bucătărie se face astfel:

```
?- certitudine(scurgere_in_bucatarie, C).
C = 0.8
```

Factorul de certitudine **C** este obținut după cum urmează: faptul ca holul este ud iar baia este uscată indică o problemă în bucătărie cu certitudine

0.9. Întrucât a existat o oarecare posibilitate de ploaie, factorul de certitudine corespunzător lui *fara_apă_din_afara* este 0.8. În final, factorul de certitudine al lui *scurgere_in_bucatarie* este calculat ca fiind $\min(0.8, 0.9) = 0.8$.

Chestiunea manevrării incertitudinii în sistemele expert a fost îndelung dezbatută în literatura de specialitate, cele prezentate aici, constituind, fără îndoială, o schemă mult simplificată de tratare a acestei probleme. Abordări matematice bazate pe teoria probabilităților există, în egală măsură. Ele vor fi amintite de noi în § 5.1.1 al lucrării de față. Ceea ce li se reproșează, cel mai adesea, este faptul că abordări corecte din punct de vedere probabilistic necesită fie informație care nu este întotdeauna disponibilă, fie anumite presupuneri simplificatoare care, de regulă, nu sunt justificate în aplicațiile practice și care fac, din nou, ca abordarea să nu fie suficient de riguroasă din punct de vedere matematic.

Una dintre cele mai cunoscute și mai utilizate scheme cu factori de certitudine este cea dezvoltată pentru sistemul MYCIN¹⁸, un sistem expert folosit în diagnosticarea infecțiilor bacteriene. Factorii de certitudine MYCIN au fost concepuți pentru a produce rezultate care păreau corecte experților, din punct de vedere intuitiv. Alți cercetători au argumentat conceperea unor factori de certitudine bazați într-o mai mare măsură pe teoria probabilităților, iar alții au experimentat scheme mult mai complexe, proiectate pentru a modela mai bine lumea reală. Factorii MYCIN continuă însă să fie utilizați cu succes în multe aplicații cu informație incertă.

¹⁸ Pentru care vezi § 5.1.1 al lucrării de față.

4.3.6. Un exemplu de sistem expert. Implementare în Prolog

Pentru exemplificarea noțiunilor descrise anterior, prezentăm, în continuare, implementarea în Prolog a unui sistem expert (SEPPCO) destinat asistării clienților unei firme de turism, în alegerea unei oferte pentru petrecerea concediului de odihnă, din totalitatea celor existente.

În cadrul acestui sistem, cunoștințele sunt reprezentate cu ajutorul regulilor de tip **if-then** (dacă-atunci), iar inferența utilizată este de tip înlănțuire înapoi cu incertitudine (în prezența incertitudinii). Incertitudinea este modelată prin intermediul factorilor de certitudine/incertitudine de tip MYCIN (o descriere mai formală a factorilor MYCIN este realizată în §5.1.1). În exemplul din această secțiune, considerăm că factorii de certitudine sunt cuprinși între -100 și 100; -100 corespunde valorii cu siguranță “fals”, iar 100 corespunde valorii cu siguranță “adevărat”. Să mai remarcăm faptul că acești factori nu reprezintă probabilități.

Shell-ul sistemului SEPPCO este o versiune modificată a shell-ului unui sistem expert cu inferență de tip înlănțuire înapoi prezentat în [6] (capitolele 3 și 4 și Anexa B).

Prezentăm, în continuare, baza de cunoștințe a sistemului SEPPCO. După cum se va vedea, conținutul bazei apare într-o formă foarte apropiată de cea a limbajului natural.

Baza de cunoștințe sistem SEPPCO

scopul este loc_concediu.

regula 1

daca buget_disponibil este redus
atunci in_romania fc 90.

regula 2
daca buget_disponibil este mediu
atunci in_romania fc 70.

regula 3
daca buget_disponibil este mare
atunci in_romania fc 50.

regula 4
daca departare este aproape
atunci in_romania.

regula 5
daca departare este departe
atunci in_romania fc 40.

regula 6
daca in_romania si
la_mare si
tip_oferta este sejur_1_luna si
buget_disponibil este mare si
anotimp este vara
atunci loc_concediu este neptun fc 80.

regula 7
daca in_romania si
la_mare si
tip_oferta este sejur_2_saptamani si
buget_disponibil este mare si
anotimp este vara

```
atunci loc_concediu este mamaia fc 90.  
  
regula 8  
daca in_romania si  
    la_mare si  
    tip_oferta este sejur_2_saptamani si  
    anotimp este vara  
atunci loc_concediu este costinesti fc 60.  
  
regula 9  
daca in_romania si  
    not la_mare si  
    tip_oferta este excursie si  
    anotimp este vara  
atunci loc_concediu este manastiri_oltenia fc 70.  
  
regula 10  
daca in_romania si  
    not la_mare si  
    tip_oferta este excursie si  
    anotimp este vara  
atunci loc_concediu este manastiri_moldova fc 60.  
  
regula 11  
daca not la_mare si  
    anotimp este vara  
atunci loc_concediu este delta_dunarii.  
  
regula 12  
daca not la_mare si  
    tip_oferta este sejur_1_luna si  
    anotimp este vara  
atunci loc_concediu_vara este busteni.
```

```
regula 13
daca la_mare si
    departare este foarte_departe si
    buget_disponibil este mare si
    anotimp este vara
atunci loc_concediu_vara este bahamas fc 80.

regula 14
daca not la_mare si
    departare este foarte_departe si
    buget_disponibil este mare si
    tip_oferta este excursie si
    anotimp este vara
atunci loc_concediu este valea_loirei.

regula 15
daca departare este aproape si
    not la_mare si
    buget_disponibil este mediu si
    anotimp este vara
atunci loc_concediu_vara este sinaia fc 70.

regula 16
daca la_mare si
    buget_disponibil este mare si
    anotimp este iarna
atunci loc_concediu este rio_de_janeiro.

regula 17
daca buget_disponibil este mare si
    not la_mare si
    departare este foarte_departe si
    tip_oferta este excursie si
```

```
        anotimp este iarna
atunci loc_concediu este austria_germania_franta fc
90.

regula 18
daca departare este foarte_departe si
    not la_mare si
        tip_oferta este sejur_2_saptamani si
        buget_disponibil este mare si
            anotimp este iarna
atunci loc_concediu este chamonix fc 60.

regula 19
daca departare este aproape si
    not la_mare si
        tip_oferta este sejur_2_saptamani si
        buget_disponibil este mare si
            anotimp este iarna
atunci loc_concediu este poiana_brasov.

regula 20
daca in_romania si
    not la_mare si
        tip_oferta este sejur_2_saptamani si
            anotimp este iarna
atunci loc_concediu este busteni fc 70.

intreaba anotimp
optiuni (vara iarna)
afiseaza 'In ce anotimp preferati sa va petreceti
concediul?'.

intreaba tip_oferta
```

```

optiuni (sejur_2_saptamani sejur_1_luna excursie)
afiseaza 'Preferati sa mergeti intr-o excursie, ori
sa petreceti un sejur intr-o statiune?'.

intreaba la_mare
optiuni (da nu)
afiseaza 'Preferati sa petreceti concediul la mare?'.

intreaba departare
optiuni (aproape departe foarte_departe)
afiseaza 'Preferati ca locul de petrecere a
concediului sa fie mai aproape, ori mai departe de
localitatea unde locuiti?'.

intreaba buget_disponibil
optiuni (redus mediu mare)
afiseaza 'Ce tip de buget alocati pentru petrecerea
concediului?'.

```

Din punctul de vedere al implementării în Prolog se poate spune că, în general, există două modalități de a lucra cu o bază de cunoștințe exprimată printr-un limbaj aproape natural. Prima dintre ele se bazează pe definirea unor operatori și aceasta este abordarea folosită în implementarea sistemului prezentat în capitolul 15 din [1]. Această abordare a fost folosită și de noi în descrierea făcută în paragraful anterior. Cea de-a doua modalitate (exemplificată aici în implementarea sistemului SEPPCO) folosește, în esență, abilitatea Prolog-ului de a lucra cu gramatici de tip DCG (“definite clause grammar” – o extensie a gramaticilor independente de context). În această abordare, din punctul de vedere al programatorului, problema analizei sintactice (parsing) se

reduce doar la specificarea unei gramatici DCG și, de aceea, am putea spune că această variantă oferă mai multă flexibilitate programatorului. O descriere amănunțită a acestei modalități de lucru există în [2].

Prezentăm, în continuare, câteva caracteristici ale sistemului SEPPCO, precum și principalele predicate ale implementării sale în Prolog.

Înainte de toate, vom face specificarea că, ori de câte ori o valoare a unui atribut este determinată, fie cu ajutorul unei reguli, fie cu ajutorul utilizatorului, o pereche de tipul (Atribut,Valoare) este salvată în spațiul de lucru, împreună cu factorul de certitudine *FC* asociat. Mai clar, informațiile referitoare la faptele cunoscute la un moment dat (informații care sunt, în esență, perechi de tipul (atribut, valoare)) sunt menținute în spațiul de lucru prin intermediul unor termeni Prolog de tipul: **fapt(av(Atr, Val), FC, Reguli)**. Aici **Atr** și **Val** reprezintă un atribut și respectiv o valoare asociată acestuia, iar **FC** reprezintă factorul de certitudine asociat faptului nostru. Argumentul **Reguli** specifică lista regulilor care au dus în mod direct la derivarea faptului și acest argument este util pentru generarea explicațiilor la întrebările de tipul **cum**. Formatul intern al regulilor specificate în baza de cunoștințe este următorul:

```
regula(N, premise(Lista), concluzie(Scop,FC))
```

Primul argument, **N**, reprezintă un atom, care identifică în mod unic regula (mai exact, **N** este un întreg pozitiv). **Lista** este o listă de elemente de tipul **av(Atr, Val)**, iar **Scop** este și el un element de forma **av(Atr, Val)**.

Menționăm, de asemenea, că prin intermediul unor propoziții de tipul **Scopul este Atribut** putem specifica în baza de cunoștințe întrebările generale (scopurile principale) la care dorim ca sistemul nostru să răspundă. În formatul intern Prolog, o propoziție **Scopul este Atribut** se traduce/ se reprezintă prin **scop(Atribut)**.

O caracteristică importantă a sistemului SEPPCO (în contextul prezenței incertitudinii) este aceea că inferența nu este oprită atunci când este găsit primul răspuns posibil la o întrebare; dimpotrivă, toate variantele posibile/”rezonabile” sunt explorate și raportate. În general, există două posibilități prin care incertitudinea poate pătrunde/apărea într-un sistem expert: prin intermediul regulilor cu concluzie incertă și prin intermediul răspunsurilor utilizatorului. În aceste condiții, situațiile în care trebuie specificată modalitatea prin care incertitudinea este propagată în sistem se referă, în esență, la următoarele cazuri:

- reguli cu concluzia incertă;
- reguli cu premise incerte;
- date incerte introduse de utilizator;
- combinarea premiselor incerte cu concluzii incerte;
- actualizarea unor fapte incerte în spațiul de lucru folosind, de asemenea, fapte incerte;
- impunerea unui prag de incertitudine pentru o premisă a unei reguli, prag care determină aplicarea/neaplicarea regulii respective.

Specificăm, în continuare, pe scurt, modalitățile în care toate aceste situații sunt tratate în SEPPCO. Menționăm, mai întâi, că lucrăm

în ipoteza că membrul stâng al unei reguli este întotdeauna o conjuncție de “scopuri elementare”. Remarcăm faptul că, în general, într-un sistem expert, membrul stâng al unei reguli poate avea o formă mai generală decât cea permisă în SEPPCO, spre exemplu poate fi de tipul conjuncții de disjuncții de scopuri elementare; pentru aceste situații au fost propuse diverse modalități de a se lucra cu factorii de (in)certitudine [3].

După cum se poate vedea în baza de cunoștințe a sistemului SEPPCO, anumite reguli au un factor de (in)certitudine asociat lor; aceste reguli sunt numite reguli cu concluzia incertă. Semnificația factorului FC asociat unei reguli R este următoarea: dacă premisele regulii R sunt adevărate cu certitudinea 100, atunci se poate deduce că concluzia este cunoscută cu certitudinea FC . De exemplu, dacă concluzia unei reguli R are certitudine $FC=50$ și dacă premisele sale sunt cunoscute cu certitudinea 100, atunci faptul din concluzia lui R este dedus cu certitudinea 50. În situația în care există premise în membrul stâng al unei reguli R , care au factorul de certitudine diferit de 100, factorul de certitudine al membrului stâng al regulii R este ales ca fiind minimul factorilor asociați premiselor lui R .

Dacă membrul stâng al unei reguli R este incert (și factorul său asociat este $StFC$) și dacă regula este „cu concluzie incertă” având factorul asociat RFC , atunci factorul de incertitudine FC al faptului determinat prin aplicarea regulii (adică al concluziei) este calculat astfel:

$$FC = \frac{RFC \cdot StFC}{100}$$

Pentru a evita folosirea regulilor al căror membru stâng au un factor de certitudine pozitiv, dar foarte mic, considerăm valoarea 20 ca

fiind valoarea pozitivă minimă a factorului de certitudine asociat membrului stâng al unei reguli aplicabile R . Există cazuri în care mai multe reguli conduc la aceeași concluzie. În aceste situații este de dorit ca fiecare dintre ele să contribuie la factorul de incertitudine FC al faptului rezultat. De aceea folosim următoarea strategie: dacă o regula R ce conduce la o concluzie deja existentă în spațiul de lucru (prin intermediul unui fapt F) se poate aplica la un moment dat, atunci noul factor de certitudine asociat faptului F se calculează astfel:

$$nou_FC(X, Y) = X + \frac{Y \cdot (100 - X)}{100}, \text{ dacă } X, Y > 0$$

$$nou_FC(X, Y) = 100 \cdot \frac{(X + Y)}{(100 - \min(X, Y))}, \text{ dacă } X > 0, Y \leq 0$$

$$nou_FC(X, Y) = 100 \cdot \frac{(X + Y)}{(100 - \min(X, Y))}, \text{ dacă } X \leq 0, Y > 0$$

$$nou_FC(X, Y) = -nou_FC(-X, -Y), \text{ dacă } X, Y < 0$$

În formulele anterioare, X reprezintă factorul de certitudine al faptului existent (F), Y este factorul de certitudine al faptului determinat prin aplicarea regulei R , iar $nou_FC(X, Y)$ este noul factor de certitudine asociat faptului considerat (F).

Părtind scurt, motorul de inferență al sistemului SEPPCO trebuie să îndeplinească următoarele funcții: să combine factorii de certitudine în modul specificat anterior, să mențină în spațiul de lucru faptele derivate și să actualizeze informațiile relevante. În plus, este nevoie ca toate informațiile referitoare la un anume atribut să fie determinate (atunci când acest lucru este necesar) și adăugate în spațiul de lucru.

Trebuie remarcat faptul că, în implementarea de față, presupunem că, initial, în spațiul de lucru nu avem nici un fapt cunoscut, ci numai reguli și alte informații referitoare la întrebările pe care utilizatorul le poate pune (toate fiind exact informațiile care se regăsesc în baza de cunoștințe).

Principalul predicat al programului, predicat care reprezintă de fapt „motorul” inferenței, este **realizare_scop**:

```
realizare_scop(Scop, FC, Iстория)
```

Argumentele sale au următoarele semnificații:

- **Scop** are fie forma **av(Atr,Val)**, fie forma **not av(Atr,Val)**; **Av** este atributul despre care dorim să aflăm informații la un moment dat și, de obicei, atunci când acest predicat este folosit, **Atr** este deja instanțiat.
- **FC** reprezintă factorul de incertitudine ce va fi asociat faptului nostru în final.
- **Iстория** reține “istoria” și este folosit pentru a răspunde la întrebarea “de ce?”

Predicatul tratează, în esență, trei cazuri, după cum urmează:

- dacă în baza de cunoștințe există un fapt care ne oferă informații referitoare la **Atr**, atunci problema este rezolvată și alte posibilități (reguli, informații de la utilizator) sunt eliminate; deoarece în baza de cunoștințe apare o informație referitoare la **Atr**, înseamnă că toate variantele de răspuns pentru **Atr** au fost explorate deja (acest lucru se întamplă deoarece, aşa cum s-a

specificat, lucrăm în ipoteza că, inițial, baza de cunoștințe nu conține nici un fapt cunoscut);

- altfel, dacă utilizatorul poate fi interogat referitor la **Atr**, atunci acesta este întrebat despre **Atr** și răspunsul său este înregistrat (adăugat în spațiul de lucru); predicatul folosit pentru interogare este **interrogheaza**;
- altfel, se determină pe rând toate regulile care au atributul **Atr** în concluzie și se încearcă demonstrarea premiselor din regulile respective; dacă acestea pot fi demonstate, atunci “incertitudinea premiselor” este combinată cu (in)certitudinea concluziei pentru a se determina factorul de certitudine al rezultatului datorat aplicării fiecărei reguli în parte.

Remarcăm faptul că, în cel de-al treilea caz, se iau în calcul toate regulile în care **Atr** intervene în concluzie, chiar dacă acestea se referă la valori diferite pentru **Atr**. Specificăm, de asemenea, faptul că sistemul nostru permite ca premisa unei reguli să fie și de forma **not Scop**; în această situație, predicatul **realizare_scop** este utilizat pentru **Scop**, iar factorul de certitudine pentru **not Scop** este definit/considerat ca fiind opusul factorului de certitudine pentru **Scop**.

Menționăm că, prin intermediul unor termeni de tipul **interrogabil(Atr,Mesaj,Optiuni)**, sistemul SEPPCO menține în spațiul de lucru informații referitoare la atrbutele despre care putem obține informații de la utilizator. Aceste informații sunt preluate din baza de cunoștințe în momentul în care ea este încărcată.

Predicatul **interogheaza** este utilizat în cel de-al doilea caz prezentat în descrierea predicatului **realizare_scop**, și anume atunci când nu există în spațiul de lucru fapte referitoare la un anume atribut, dar se pot obține informații referitoare la **Atr** de la utilizator. Forma sa este:

interogheaza(Atr,Mesaj,Optiuni,Istorie)

Argumentele sale au următoarele semnificații:

- **Atr** este atributul despre care dorim să obținem informații;
- **Mesaj** reprezintă mesajul/întrebarea care va fi adresată utilizatorului atunci când acesta va fi interogat; acest mesaj poate fi obținut prin intermediul predicatului **interrogabil**;
- **Optiuni** este de fapt lista în care sunt specificate valorile posibile ale atributului **Atr**; ca și în cazul atributului **Mesaj**, **Optiuni** poate fi determinat prin intermediul predicatului **interrogabil**;
- Argumentul **Istorie** reprezintă istoria și este folosit pentru a răspunde la întrebările de tipul “**de ce?**”

Specificăm faptul că răspunsul dorit este fie de forma **M** (unde **M** este un membru din lista de opțiuni **Optiuni**), fie de forma **M fc Numar** (unde **Numar** este un factor de certitudine diferit de 100). Mai remarcăm doar că în prima situație se consideră că factorul de certitudine al răspunsului asociat este maxim.

Cel de-al treilea caz prezentat în descrierea predicatului **realizare_scop** se referă la situația în care sunt efectiv utilizate reguli

pentru a determina informații despre **Atr**. Predicatul folosit în această situație este predicatul **fg**, predcat care forcează folosirea tuturor regulilor care au **Atr** în concluzie, precum și combinarea factorilor de certitudine ale faptelor deriveate, în modul dorit și descris anterior. După cum se va vedea în textul programului, predicatele principale folosite în definiția predicatului **fg** sunt: **demonstreaza**, **ajusteaza**, **actualizeaza**. Predicatul **demonstreaza** este utilizat pentru a considera toate premisele din lista de premise a fiecărei reguli de interes, apelând **realizare_scop** pentru fiecare dintre premise; factorul de certitudine final asociat membrului stâng al fiecărei reguli este calculat ca fiind minimul factorilor de certitudine din lista de premise asociate reguli respective. Predicatul **ajusteaza** este folosit pentru a combina incertitudinea asociată unei reguli cu incertitudinea premiselor sale. În final, predicatul **actualizeaza** tratează situația în care concluzia datorată aplicării unei reguli se regăsește deja în spațiul de lucru prin intermediul unui fapt; acest predcat este folosit în cazul în care concluziile mai multor reguli se referă la aceeași pereche (atribut, valoare).

Principalul predcat al shell-ului este predicatul **pornire**, predcat care determină un ciclu de execuție, în acesta din urmă opțiunile fiind **incarca**, **consulta**, **reinitiaza**, **afisare_fapte**, **iesire**, **cum**. Opțiunile **incarca**, **afisare_fapte**, **consulta**, **cum** sunt implementate prin intermediul predicatorilor **incarca**, **scopuri_princ**, **cum**, **afiseaza_fapte**. Predicatul **scopuri_princ** încearcă satisfacerea, pe rând, a fiecărui scop care

a fost specificat în baza de cunoștințe; el reprezintă punctul de pornire al inferenței (de tip înlănțuire înapoi). Predicatul **incarca** este utilizat pentru încărcarea bazei de cunoștințe în spațiul de lucru; cel mai important predicat care intervine în descrierea predicatului **incarca** este predicatul **incarca_reguli**, care este definit în felul următor:

```
incarca_reguli :-  
    repeat, citeste_propozitie(L),  
    proceseaza(L), L == [end_of_file], nl.
```

Așa cum arată definiția sa, predicatul **incarca_reguli** forțează citirea, pe rând, a fiecărei propoziții din fișierul de intrare (fișier determinat în interiorul predicatului **incarca**). Fiecarei propoziții ii este asociată o listă de atomi (prin intermediul predicatului **citeste_propozitie**) și fiecare astfel de listă de atomi este “transformată” într-un termen Prolog, prin intermediul predicatului **proceseaza**. Definirea predicatului **citeste_propozitie** se bazează pe o implementare sugerată în [2]. Predicatul **proceseaza** este definit astfel:

```
proceseaza([end_of_file]):-!.  
proceseaza(L) :- trad(R,L,[]), assertz(R), !.
```

Argumentul său, **L**, reprezintă o listă de atomi, despre care considerăm că este validă dacă ea corespunde cuvintelor unei fraze din fișierul în care este specificată baza de cunoștințe a sistemului SEPPCO. În esență, predicatul **trad** este folosit pentru traducerea unei fraze valide

din baza de cunoștințe (frază specificată prin lista de atomi corespunzătoare) în formatul intern al Prologului. Așa cum am specificat deja, această traducere este implementată folosind gramatici de tip DCG. Dacă lista de atomi **L** nu reprezintă o frază validă, atunci predicatul **trad** și, prin urmare, și predicatul **proceseaza**, eșuează. Remarcăm faptul că, în definirea predicatului **trad**, se specifică în mod clar ce înseamnă fraza validă.

Reamintim că informațiile referitoare la faptele cunoscute la un moment dat sunt menținute în spațiul de lucru prin intermediul unor termeni Prolog de tipul **fapt(av(Atr, Val), FC, Reguli)**, unde argumentul **Reguli** specifică lista regulilor care au dus în mod direct la derivarea faptului nostru. Folosind această caracteristică, implementarea predicatului **cum** (predicat folosit pentru generarea explicațiilor la întrebările de tipul „cum?”) este foarte naturală (a se vedea textul programului).

Predicatul **reinitiaza** este utilizat pentru a reduce spațiul de lucru la o formă identică cu cea imediat de după încărcarea bazei de cunoștințe (o formă în care, în spațiul de lucru, se regăsesc numai reguli și informații referitoare la întrebările pe care utilizatorul le poate pune, dar nu și fapte cunoscute (deduse/ derivate)).

Prezentăm, în continuare, o implementare în Prolog a sistemului nostru, implementare ce folosește predicatele și considerentele anterioare:

```
: -use_module(library(lists)) .  
: -use_module(library(system)) .  
: -op(900,fy,not) .
```

```
:dynamic fapt/3.
:-dynamic interogat/1.
:-dynamic scop/1.
:-dynamic interogabil/3.
:-dynamic regula/3.

not(P):-P,! ,fail.
not(_).

scrie_lista([]):-nl.
scrie_lista([H|T]) :-
    write(H), tab(1),
    scrie_lista(T).

afiseaza_fapte :-
    write('Fapte existente in baza de cunostinte:'),
    nl,nl, write(' (Atribut,valoare) '), nl,nl,
    listeaza_fapte,nl.

listeaza_fapte:-
    fapt(av(Atr,Val),FC,_),
    write('('),write(Atr),write(','),
    write(Val), write(')'),
    write(','), write(' certitudine '),
    FC1 is integer(FC),write(FC1),
    nl,fail.

listeaza_fapte.

lista_float_int([],[]).
lista_float_int([Regula|Reguli],[Regula1|Reguli1]):-
```

```

(Regula \== utiliz,
Regula1 is integer(Regula);
Regula ==utiliz, Regula1=Regula),
lista_float_int(Reguli,Reguli1).

pornire :-
    retractall(interrogat(_)),
    retractall(fapt(_,_,_)),
    repeat,
    write('Introduceti una din urmatoarele optiuni: '),
    nl,nl,
    write(' (Incarca Consulta Reinitiaza Afisare_fapte
Cum Iesire) '),
    nl,nl,write('|: '),citeste_linie([H|T]),
    executa([H|T]), H == iesire.

executa([incarca]) :-
    incarca,! ,nl,
    write('Fisierul dorit a fost incarcat'),nl.

executa([consulta]) :-
    scopuri_princ,!.

executa([reinitiaza]) :-
    retractall(interrogat(_)),
    retractall(fapt(_,_,_)),! .

executa([afisare_fapte]) :-
    afiseaza_fapte,!.

executa([cum|L]) :- cum(L),! .
executa([iesire]):-! .
executa([_|_]) :-
```

```

write('Comanda incorecta! '),nl.

scopuri_princ :-
    scop(Atr), determina(Atr), afiseaza_scop(Atr), fail.
scopuri_princ.

determina(Atr) :-
    realizare_scop(av(Atr,_),_, [scop(Atr)]), !.
determina(_).

afiseaza_scop(Atr) :-
    nl, fapt(av(Atr,Val), FC, _),
    FC >= 20, scrie_scop(av(Atr,Val), FC),
    nl, fail.
afiseaza_scop(_):-nl,nl.

scrie_scop(av(Atr,Val), FC) :-
    transformare(av(Atr,Val), X),
    scrie_lista(X), tab(2),
    write(' '),
    write('factorul de certitudine este '),
    FC1 is integer(FC), write(FC1).

realizare_scop(not Scop, Not_FC, Istorie) :-
    realizare_scop(Scop, FC, Istorie),
    Not_FC is - FC, !.
realizare_scop(Scop, FC, _) :-
    fapt(Scop, FC, _), !.
realizare_scop(Scop, FC, Istorie) :-
    pot_interoga(Scop, Istorie),

```

```

    !,realizare_scop(Scop,FC,Istorie).

realizare_scop(Scop,FC_current,Istorie) :-
    fg(Scop,FC_current,Istorie).

fg(Scop,FC_current,Istorie) :-
    regula(N, premise(Lista), concluzie(Scop,FC)),
    demonstreaza(N,Lista,FC_premise,Istorie),
    ajusteaza(FC,FC_premise,FC_nou),
    actualizeaza(Scop,FC_nou,FC_current,N),
    FC_current == 100,!.

fg(Scop,FC,_) :- fapt(Scop,FC,_).

pot_interoga(av(Atr,_),Istorie) :-
    not interogat(av(Atr,_)),
    interogabil(Atr,Optiuni,Mesaj),
    interogheaza(Atr,Mesaj,Optiuni,Istorie),nl,
    asserta(interogat(av(Atr,_))).

cum([]) :- write('Scop? '),nl,
           write('|:'),citeste_linie(Linie),nl,
           transformare(Scop,Linie), cum(Scop).

cum(L) :- transformare(Scop,L),nl, cum(Scop).

cum(not Scop) :-
    fapt(Scop,FC,Reguli),
    lista_float_int(Reguli,Reguli1),
    FC < -20,transformare(not Scop,PG),
    append(PG,[a,fost,derivat,cu, ajutorul, 'regulilor:
'|Reguli1],LL),
    scrie_lista(LL),nl,afis_reguli(Reguli),fail.

```

```

cum(Scop) :-  

    fapt(Scop,FC,Reguli),  

    lista_float_int(Reguli,Reguli1),  

    FC > 20, transformare(Scop,PG),  

    append(PG,[a,fost,derivat,cu, ajutorul, 'regulilor:  

'|Reguli1],LL),  

    scrie_lista(LL),nl,afis_reguli(Reguli),  

    fail.  

cum(_).  

afis_reguli([]).  

afis_reguli([N|X]) :-  

    afis_regula(N),  

    premisele(N),  

    afis_reguli(X).  

afis_regula(N) :-  

    regula(N, premise(Lista_premise),  

    concluzie(Scop,FC)),NN is integer(N),  

    scrie_lista(['regula ',NN]),  

    scrie_lista([' Daca']),  

    scrie_lista_premise(Lista_premise),  

    scrie_lista([' Atunci']),  

    transformare(Scop,Scop_tr),  

    append([''],Scop_tr,L1),  

    FC1 is integer(FC),append(L1,[FC1],LL),  

    scrie_lista(LL),nl.  

scrie_lista_premise([]).  

scrie_lista_premise([H|T]) :-  


```

```

transformare(H,H_tr) ,
tab(5),scrie_lista(H_tr),
scrie_lista_premise(T) .

transformare(av(A,da) ,[A]) :- !.
transformare(not av(A,da) , [not,A]) :- !.
transformare(av(A,nu) ,[not,A]) :- !.
transformare(av(A,V) ,[A,este,V]) .

premisele(N) :-
    regula(N, premise(Lista_premise) , _),
    !, cum_premise(Lista_premise).

cum_premise([]).
cum_premise([Scop|X]) :-
    cum(Scop),
    cum_premise(X).

interogheaza(Atr,Mesaj,[da,nu],Istorie) :-
    !, write(Mesaj), nl,
    de_la_utiliz(X,Istorie,[da,nu]),
    det_val_fc(X,Val,FC),
    asserta( fapt(av(Atr,Val),FC,[utiliz]) ) .
interogheaza(Atr,Mesaj,Optiuni,Istorie) :-
    write(Mesaj), nl,
    citeste_opt(VLista,Optiuni,Istorie),
    assert_fapt(Atr,VLista).

citeste_opt(X,Optiuni,Istorie) :-
    append(['('],Optiuni,Opt1),

```

```

append(Opt1,[')'],Opt),
scrie_lista(Opt),
de_la_utiliz(X,Istorie,Optiuni).

de_la_utiliz(X,Istorie,Lista_opt) :-
repeat,write(' : '),citeste_linie(X),
proceseaza_raspuns(X,Istorie,Lista_opt).

proceseaza_raspuns([de_ce],Istorie,_) :-
nl,afis_istorie(Istorie),!,fail.

proceseaza_raspuns([X],_,Lista_opt):-
member(X,Lista_opt).

proceseaza_raspuns([X,fc,FC],_,Lista_opt):-
member(X,Lista_opt),float(FC).

assert_fapt(Atr,[Val,fc,FC]) :-
!,asserta( fapt(av(Atr,Val),FC,[utiliz]) ).

assert_fapt(Atr,[Val]) :-
asserta( fapt(av(Atr,Val),100,[utiliz]) ).

det_val_fc([nu],da,-100).
det_val_fc([nu,FC],da,NFC) :- NFC is -FC.
det_val_fc([nu,fc,FC],da,NFC) :- NFC is -FC.
det_val_fc([Val,FC],Val,FC).
det_val_fc([Val,fc,FC],Val,FC).
det_val_fc([Val],Val,100).

afis_istorie([]) :- nl.
afis_istorie([scop(X)|T]) :-
scrie_lista([scop,X]),!,

```

```

afis_istorie(T) .
afis_istorie([N|T]) :-  

    afis_regula(N), !, afis_istorie(T) .

demonstreaza(N,ListaPremise,Val_finala,Istorie) :-  

    dem(ListaPremise,100,Val_finala,[N|Istorie]), !.

dem([],Val_finala,Val_finala,_).
dem([H|T],Val_actuala,Val_finala,Istorie) :-  

    realizare_scop(H,FC,Istorie),
    Val_interm is min(Val_actuala,FC),
    Val_interm >= 20,
    dem(T,Val_interm,Val_finala,Istorie).

actualizeaza(Scop,FC_nou,FC,RegulaN) :-  

    fapt(Scop,FC_vechi,_),
    combina(FC_nou,FC_vechi,FC),
    retract( fapt(Scop,FC_vechi,Reguli_vechi) ),
    asserta( fapt(Scop,FC,[RegulaN | Reguli_vechi]) ), !.
actualizeaza(Scop,FC,FC,RegulaN) :-  

    asserta( fapt(Scop,FC,[RegulaN]) ) .

ajusteaaza(FC1,FC2,FC) :-  

    X is FC1 * FC2 / 100,  

    FC is round(X).

combina(FC1,FC2,FC) :-  

    FC1 >= 0, FC2 >= 0,  

    X is FC2*(100 - FC1)/100 + FC1,  

    FC is round(X).

combina(FC1,FC2,FC) :-  


```

```

FC1 < 0, FC2 < 0,
X is - ( -FC1 -FC2 * (100 + FC1)/100) ,
FC is round(X).

combina(FC1,FC2,FC) :-  

    (FC1 < 0 ; FC2 < 0) ,  

    (FC1 > 0 ; FC2 > 0) ,  

    FCM1 is abs(FC1), FCM2 is abs(FC2) ,  

    MFC is min(FCM1,FCM2) ,  

    X is 100 * (FC1 + FC2) / (100 - MFC) ,  

    FC is round(X).

incarca :-
    write('Introduceti numele fisierului care doriti sa  

fie incarcat: '), nl, write('|:'), read(F),
    file_exists(F), !, incarca(F).
incarca:- write('Nume incorect de fisier! '), nl, fail.

incarca(F) :-
    retractall(interrogat(_)), retractall(fapt(_,_,_)),
    retractall(scop(_)), retractall(interrogabil(_,_,_)),
    retractall(regula(_,_,_)),
    see(F), incarca_reguli, seen, !.

incarca_reguli :-
    repeat, citeste_propozitie(L),
    proceseaza(L), L == [end_of_file], nl.

proceseaza([end_of_file]):-! .
proceseaza(L) :-
    trad(R,L,[]), assertz(R), ! .

```

```

trad(scop(X)) --> [scopul,este,X].
trad(scop(X)) --> [scopul,X].
trad(interogabil(Atr,M,P)) -->
    [intreaba,Atr],lista_optiuni(M),afiseaza(Atr,P).
trad(regula(N,premise(Daca),concluzie(Atunci,F))) -->
    identificator(N),daca(Daca),atunci(Atunci,F).
trad('Eroare la parsare'-L,L,_).

lista_optiuni(M) -->
    [optiuni,'()',lista_de_optiuni(M)].
lista_de_optiuni([Element]) --> [Element,'()''].
lista_de_optiuni([Element|T]) -->
    [Element],lista_de_optiuni(T).

afiseaza(_,P) --> [afiseaza,P].
afiseaza(P,P) --> [].
identificator(N) --> [regula,N].

daca(Daca) --> [daca],lista_premise(Daca).

lista_premise([Daca]) --> propoz(Daca),[atunci].
lista_premise([Prima|Celalalte]) -->
    propoz(Prima),[si],lista_premise(Celalalte).
lista_premise([Prima|Celalalte]) -->
    propoz(Prima),[','],lista_premise(Celalalte).

atunci(Atunci,FC) --> propoz(Atunci),[fc],[FC].
atunci(Atunci,100) --> propoz(Atunci).

propoz(not av(Atr,da)) --> [not,Atr].
propoz(av(Atr,Val)) --> [Atr,este,Val].

```

```

propoz(av(Attr,da)) --> [Attr] .

citereste_linie([Cuv|Lista_cuv]) :-  

    get0(Car),  

    citeste_cuvant(Car, Cuv, Car1),  

    rest_cuvinte_linie(Car1, Lista_cuv).

% -1 este codul ASCII pt EOF

rest_cuvinte_linie(-1, []):-!.  

rest_cuvinte_linie(Car, []) :-(Car==13;Car==10), !.  

rest_cuvinte_linie(Car, [Cuv1|Lista_cuv]) :-  

    citeste_cuvant(Car,Cuv1,Car1),  

    rest_cuvinte_linie(Car1,Lista_cuv).

citereste_propozitie([Cuv|Lista_cuv]) :-  

    get0(Car),citeste_cuvant(Car, Cuv, Car1),  

    rest_cuvinte_propozitie(Car1, Lista_cuv).

rest_cuvinte_propozitie(-1, []):-!.  

rest_cuvinte_propozitie(Car, []) :-(Car==46), !.  

rest_cuvinte_propozitie(Car, [Cuv1|Lista_cuv]) :-  

    citeste_cuvant(Car,Cuv1,Car1),  

    rest_cuvinte_propozitie(Car1,Lista_cuv).

citereste_cuvant(-1,end_of_file,-1):-!.  

citereste_cuvant(Caracter,Cuvant,Caracter1) :-  

    caracter_cuvant(Caracter),!,  

    name(Cuvant, [Caracter]),get0(Caracter1).
citereste_cuvant(Caracter, Numar, Caracter1) :-  

    caracter_numar(Caracter),!,  

    name(Numar, [Caracter]),get0(Caracter1).

```

```

    citeste_tot_numarul(Caracter, Numar, Caracter1).

    citeste_tot_numarul(Caracter, Numar, Caracter1) :-
        determina_lista(Lista1, Caracter1),
        append([Caracter], Lista1, Lista),
        transforma_lista_numar(Lista, Numar).

determina_lista(Lista, Caracter1) :-
    get0(Caracter),
    (caracter_numar(Caracter),
     determina_lista(Lista1, Caracter1),
     append([Caracter], Lista1, Lista);
    \+(caracter_numar(Caracter)),
     Lista=[], Caracter1=Caracter).

transforma_lista_numar([], 0).
transforma_lista_numar([H|T], N) :-
    transforma_lista_numar(T, NN),
    lungime(T, L), Aux is exp(10, L),
    HH is H-48, N is HH*Aux+NN.

lungime([], 0).
lungime([_|T], L) :-
    lungime(T, L1),
    L is L1+1.

% 39 este codul ASCII pt ' '

    citeste_cuvant(Caracter, Cuvant, Caracter1) :-  

        Caracter==39, !,  

        pana_la_urmatorul_apostrof(Lista_caractere),

```

```

L=[Caracter|Lista_caractere] ,
name(Cuvant, L),get0(Caracter1) .

pana_la_urmatorul_apostrof(Lista_caractere) :- 
    get0(Caracter),
    (Caracter == 39,Lista_caractere=[Caracter];
    Caracter\==39,
    pana_la_urmatorul_apostrof(Lista_caractere1),
    Lista_caractere=[Caracter|Lista_caractere1]).

citeste_cuvant(Caracter,Cuvant,Caracter1) :- 
    caractere_in_interiorul_unui_cuvant(Caracter),!,
    ((Caracter>64,Caracter<91),!,
    Caracter_modificat is Caracter+32;
    Caracter_modificat is Caracter),
    citeste_intreg_cuvantul(Caractere,Caracter1),
    name(Cuvant,[Caracter_modificat|Caractere]).

citeste_intreg_cuvantul(Lista_Caractere,Caracter1) :- 
    get0(Caracter),
    (caractere_in_interiorul_unui_cuvant(Caracter),
    ((Caracter>64,Caracter<91),!,
    Caracter_modificat is Caracter+32;
    Caracter_modificat is Caracter),
    citeste_intreg_cuvantul(Lista_Caracterel, Caracter1),
    Lista_Caractere=[Caracter_modificat|Lista_Caracterel]
    ; \+(caractere_in_interiorul_unui_cuvant(Caracter)),
    Lista_Caractere=[], Caracter1=Caracter).

citeste_cuvant(_,Cuvant,Caracter1) :-

```

```

get0(Caracter),
citeste_cuvant(Caracter,Cuvant,Caracter1).

caracter_cuvant(C) :-
member(C,[44,59,58,63,33,46,41,40]).

% am specificat codurile ASCII pentru , ; : ? ! . ) ( 

caracter_in_interiorul_unui_cuvant(C) :-
C>64,C<91;C>47,C<58;
C==45;C==95;C>96,C<123.

caracter_numar(C):-C<58,C>=48.

```

Prezentăm, în continuare, un exemplu complex de execuție a programului. În acest exemplu, sunt evidențiate cele mai importante caracteristici ale sistemului SEPCCO, prin intermediul unor situații special alese pentru a releva modul în care sistemul propus tratează incertitudinea. Un exercițiu util (chiar recomandabil) ar putea fi rularea exemplului împreună cu urmărirea îndeaproape a mecanismului de producere a răspunsurilor, în fiecare caz în parte.

Exemplu de utilizare

| ?- pornire.

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: incarca

Introduceti numele fisierului care doriti sa fie incarcat:

|:'C:/ sist_seppco.txt'.

Fisierul dorit a fost incarcat

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

[: consulta
 'Ce tip de buget alocati pentru petrecerea concediului?'
 (redus mediu mare)
 : mare

'Preferati ca locul de petrecere a concediului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'
 (aproape departe foarte_departe)
 : departe

'Preferati sa petreceti concediul la mare?'
 : da

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'
 (sejur_2_saptamani sejur_1_luna excursie)
 : sejur_2_saptamani

'In ce anotimp preferati sa va petreceti concediul?'
 (vara iarna)
 : vara

loc_concediu este costinesti
 factorul de certitudine este 42
 loc_concediu este mamaia
 factorul de certitudine este 63

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

[: afisare_fapte
 Fapte existente în baza de cunostinte:

(Atribut,valoare)

(loc_concediu,costinesti), certitudine 42
 (loc_concediu,mamaia), certitudine 63
 (anotimp,vara), certitudine 100
 (tip_oferta,sejur_2_saptamani), certitudine 100
 (la_mare,da), certitudine 100
 (in_romania,da), certitudine 70
 (departare,departe), certitudine 100

(buget_disponibil,mare), certitudine 100

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum

Scop?

|:loc_concediu este costinesti

loc_concediu este costinesti a fost derivat cu ajutorul regulilor: 8

regula 8

Daca

 in_romania

 la_mare

 tip_oferta este sejur_2_saptamani

 anotimp este vara

Atunci

 loc_concediu este costinesti 60

in_romania a fost derivat cu ajutorul regulilor: 5 3

regula 5

Daca

 departare este departe

Atunci

 in_romania 40

departare este departe a fost derivat cu ajutorul regulilor: utiliz

regula 3

Daca

 buget_disponibil este mare

Atunci

 in_romania 50

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

la_mare a fost derivat cu ajutorul regulilor: utiliz

tip_oferta este sejur_2_saptamani a fost derivat cu ajutorul regulilor: utiliz

anotimp este vara a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: reinitiaza

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: consulta

'Ce tip de buget alocati pentru petrecerea concediului?'

(redus mediu mare)

: mare

'Preferati ca locul de petrecere a concediului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'

(aproape departe foarte_departe)

: aproape

'Preferati sa petreceti concediul la mare?'

: da

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'

(sejur_2_saptamani sejur_1_luna excursie)

: sejur_1_luna

'In ce anotimp preferati sa va petreceti concediul?'

(vara iarna)

: vara

loc_concediu este neptun

factorul de certitudine este 80

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum

Scop?

|:loc_concediu este neptun

loc_concediu este neptun a fost derivat cu ajutorul regulilor: 6

regula 6

Daca

in_romania

la_mare

tip_oferta este sejur_1_luna

buget_disponibil este mare
anotimp este vara

Atunci
loc_concediu este neptun 80

in_romania a fost derivat cu ajutorul regulilor: 4 3

regula 4

Daca
departare este aproape
Atunci
in_romania 100

departare este aproape a fost derivat cu ajutorul regulilor: utiliz

regula 3

Daca
buget_disponibil este mare
Atunci
in_romania 50

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

la_mare a fost derivat cu ajutorul regulilor: utiliz

tip_oferta este sejur_1_luna a fost derivat cu ajutorul regulilor: utiliz

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

anotimp este vara a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: reinitiaza

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: consulta

'Ce tip de buget alocati pentru petrecerea concediului?'

(redus mediu mare)

: mare

'Preferati ca locul de petrecere a condeiului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'
 (aproape departe foarte_departe)
 : aproape fc 90

'Preferati sa petreceti condeiul la mare?'
 : da

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'
 (sejur_2_saptamani sejur_1_luna excursie)
 : sejur_1_luna fc 60

'In ce anotimp preferati sa va petreceti condeiul?'
 (vara iarna)
 : vara fc 70

loc_condeiu este neptun
 factorul de certitudine este 48

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum_Iesire)

|: cum loc_condeiu este neptun

loc_condeiu este neptun a fost derivat cu ajutorul regulilor: 6

regula 6
 Daca
 in_romania
 la_mare
 tip_oferta este sejur_1_luna
 buget_disponibil este mare
 anotimp este vara
 Atunci
 loc_condeiu este neptun 80

in_romania a fost derivat cu ajutorul regulilor: 4 3
 regula 4

Daca
 departare este aproape
 Atunci
 in_romania 100

departare este aproape a fost derivat cu ajutorul regulilor: utiliz

regula 3

Daca

 buget_disponibil este mare

Atunci

 in_romania 50

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

la_mare a fost derivat cu ajutorul regulilor: utiliz

tip_oferta este sejur_1_luna a fost derivat cu ajutorul regulilor: utiliz

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

anotimp este vara a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: afisare_fapte

Fapte existente în baza de cunostinte:

(Atribut,valoare)

(loc_concediu,neptun), certitudine 48

(anotimp,vara), certitudine 70

(tip_oferta,sejur_1_luna), certitudine 60

(la_mare,da), certitudine 100

(in_romania,da), certitudine 95

(departare,aproape), certitudine 90

(buget_disponibil,mare), certitudine 100

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: reinitiaza

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: consulta

'Ce tip de buget alocati pentru petrecerea concediului?"

(redus mediu mare)

: mare

'Preferati ca locul de petrecere a condeiului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'
 (aproape departe foarte_departe)
 : departe

'Preferati sa petreceti condeiul la mare?'
 : nu fc 30

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'
 (sejur_2_saptamani sejur_1_luna excursie)
 : excursie

'In ce anotimp preferati sa va petreceti condeiul?'
 (vara iarna)
 : vara

loc_condeiu este delta_dunarii
 factorul de certitudine este 30
 loc_condeiu este manastiri_oltenia
 factorul de certitudine este 21

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum loc_condeiu este delta_dunarii

loc_condeiu este delta_dunarii a fost derivat cu ajutorul regulilor: 11

regula 11

Daca

not la_mare
 anotimp este vara

Atunci

loc_condeiu este delta_dunarii 100

not la_mare a fost derivat cu ajutorul regulilor: utiliz

anotimp este vara a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum loc_condeiu este manastiri_oltenia

loc_concediu este manastiri oltenia a fost derivat cu ajutorul regulilor: 9

regula 9

Daca

in_romania

not la_mare

tip_oferta este excursie

anotimp este vara

Atunci

loc_concediu este manastiri oltenia 70

in_romania a fost derivat cu ajutorul regulilor: 5 3

regula 5

Daca

departare este departe

Atunci

in_romania 40

departare este departe a fost derivat cu ajutorul regulilor: utiliz

regula 3

Daca

buget_disponibil este mare

Atunci

in_romania 50

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

not la_mare a fost derivat cu ajutorul regulilor: utiliz

tip_oferta este excursie a fost derivat cu ajutorul regulilor: utiliz

anotimp este vara a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: afisare_fapte

Fapte existente in baza de cunostinte:

(Atribut,valoare)

(loc_concediu,delta_dunarii), certitudine 30

(loc_concediu,manastiri_moldova), certitudine 18

(loc_concediu,manastiri_oltenia), certitudine 21
 (anotimp,vara), certitudine 100
 (tip_oferta,excursie), certitudine 100
 (la_mare,da), certitudine -30
 (in_romania,da), certitudine 70
 (departare,departe), certitudine 100
 (buget_disponibil,mare), certitudine 100

Introduceti una din urmatoarele optiuni:
 (Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: reinitiaza

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: consulta

'Ce tip de buget alocati pentru petrecerea condeiului?'

(redus mediu mare)

: mare

'Preferati ca locul de petrecere a condeiului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'

(aproape departe foarte_departe)

: departe

'Preferati sa petreceti condeiul la mare?'

: da fc 90

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'

(sejur_2_saptamani sejur_1_luna excursie)

: sejur_1_luna

'In ce anotimp preferati sa va petreceti condeiul?'

(vara iarna)

: iarna fc 50

loc_concediu este rio_de_janeiro

factorul de certitudine este 50

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: cum loc_concediu este rio_de_janeiro

loc_concediu este rio_de_janeiro a fost derivat cu ajutorul regulilor: 16

regula 16

Daca

la_mare

buget_disponibil este mare

anotimp este iarna

Atunci

loc_concediu este rio_de_janeiro 100

la_mare a fost derivat cu ajutorul regulilor: utiliz

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

anotimp este iarna a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: afisare_fapte

Fapte existente în baza de cunostinte:

(Atribut, valoare)

(loc_concediu,rio_de_janeiro), certitudine 50

(anotimp,iarna), certitudine 50

(tip_oferta,sejur_1_luna), certitudine 100

(la_mare,da), certitudine 90

(in_romania,da), certitudine 70

(departare,departe), certitudine 100

(buget_disponibil,mare), certitudine 100

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: reinitiaza

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: consulta

'Ce tip de buget alocati pentru petrecerea concediului?'

(redus mediu mare)

: mare fc 35

'Preferati ca locul de petrecere a condeiului sa fie mai aproape, ori mai departe de localitatea unde locuiti?'

(aproape departe foarte_departe)
: foarte_departe

'Preferati sa petreceti condeiul la mare?'

: nu fc 90

'In ce anotimp preferati sa va petreceti condeiul?'

(vara iarna)
: iarna fc 80

'Preferati sa mergeti intr-o excursie, ori sa petreceti un sejur intr-o statiune?'

(sejur_2_saptamani sejur_1_luna excursie)
: sejur_2_saptamani

loc_condeiu este chamonix
factorul de certitudine este 21

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum_Iesire)

|: cum loc_condeiu este chamonix

loc_condeiu este chamonix a fost derivat cu ajutorul regulilor: 18

regula 18
Daca
departare este foarte_departe
not_la_mare
tip_oferta este sejur_2_saptamani
buget_disponibil este mare
anotimp este iarna

Atunci
loc_condeiu este chamonix 60

departare este foarte_departe a fost derivat cu ajutorul regulilor: utiliz

not_la_mare a fost derivat cu ajutorul regulilor: utiliz

tip_oferta este sejur_2_saptamani a fost derivat cu ajutorul regulilor: utiliz

buget_disponibil este mare a fost derivat cu ajutorul regulilor: utiliz

anotimp este iarna a fost derivat cu ajutorul regulilor: utiliz

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: afisare_fapte

Fapte existente în baza de cunoștințe:

(Atribut,valoare)

(loc_concediu,chamonix), certitudine 21
 (tip_oferta,sejur_2_saptamani), certitudine 100
 (anotimp,iarna), certitudine 80
 (la_mare,da), certitudine -90
 (departare,foarte_departe), certitudine 100
 (in_romania,da), certitudine 18
 (buget_disponibil,mare), certitudine 35

Introduceti una din urmatoarele optiuni:

(Incarca Consulta Reinitiaza Afisare_fapte Cum Iesire)

|: iesire

yes

4.4. Rețele semantice și cadre

Sistemele bazate pe reguli nu constituie singurul cadru de lucru pentru reprezentarea cunoștințelor. Alte două moduri de lucru posibile constau în utilizarea aşa-numitelor *rețele semantice* și, respectiv, *cadre*, la care ne vom referi, pe scurt, în cele ce urmează. Acestea diferă de reprezentările bazate pe reguli în primul rând prin faptul că se concentrează asupra reprezentării, într-un mod structurat, a unor multimi foarte mari de fapte. Multimea de fapte reprezentată este *structurată* și, în măsura în care este posibil, *comprimată*: o serie de fapte nu sunt reprezentate în mod explicit atunci când ele pot fi reconstruite prin mecanismul de inferență. Atât rețelele semantice, cât și cadrele sunt ușor de reprezentat în Prolog, aşa cum se va vedea în cele ce urmează.

4.4.1. Rețelele semantice

O rețea semantică constă din entități și relații între acestea. Ea se reprezintă sub forma unui *graf orientat* ale cărui arce leagă vârfuri etichetate. Cel mai adesea nodurile grafului corespund entităților, în timp ce relațiile sunt figurate ca legături între noduri etichetate cu denumirile relațiilor. În Fig. 4.5 este prezentată, spre exemplificare, o rețea semantică:

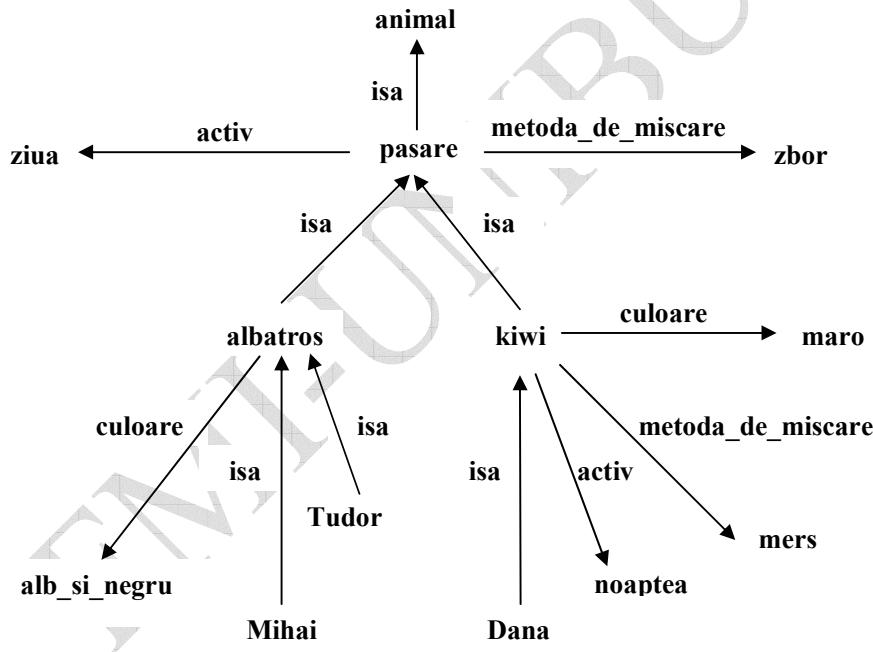


Fig. 4.5

Exemplul a fost preluat de noi din [1], iar rețeaua este folosită pentru reprezentarea unor fapte de tipul următor:

- O pasăre este un anumit tip de animal.
- Metoda uzuală de mișcare a păsărilor este zborul.
- Un albatros este o pasăre.
- Mihai și Tudor sunt albatroși.

Se observă că relația de tip **isa** uneori realizează legătura dintre *o clasă* de obiecte și *o altă clasă* având o sferă mai largă și reprezentând un element supraordonat (*animal* este o clasă supraordonată a lui *pasare*), iar alteori face legătura între *o instanțiere* a unei clase și însăși clasa respectivă (*Mihai* este un *albatros*). O astfel de rețea este imediat “tradusă” sub forma unor fapte Prolog de tipul:

```
isa(pasare,animal).
isa(mihai,albatros).
metoda_de_miscare(pasare,zbor).
metoda_de_miscare(kiwi,mers).
```

Pe lângă faptele de acest tip, care sunt enunțate în mod explicit, alte fapte pot fi deduse pe baza aceleiași rețele. *Principiul tipic de inferență* încorporat în rețea este, aşa cum s-a mai menționat, *moștenirea proprietăților*. Faptele sunt moștenite prin intermediul relației **isa**. În Prolog putem enunța faptul că metoda de mișcare este moștenită, urcând de-a lungul ierarhiei isa, sub forma următoare:

```
metoda_de_miscare(X,Metoda) :-
    isa(X,SuperX),
    metoda_de_miscare(SuperX,Metoda).
```

Pentru a nu enunța câte o regulă de moștenire distinctă corespunzător fiecărei relații care poate fi moștenită, se va enunța o regulă mult mai generală referitoare la fapte, fie ele enunțate în mod explicit în cadrul rețelei, fie moștenite:

```
fapta(Fapta) :- %Fapta nu este o variabila;
Fapta, !. %Fapta explicită în rețea

fapta(Fapta) :-
Fapta=.. [Rel,Arg1,Arg2],
isa(Arg1,SuperArg), %urcând de-a lungul
SuperFapta=.. [Rel,SuperArg,Arg2]. %ierarhiei isa
```

Rețeaua semantică din Fig. 4.5 poate fi acum interogată în modul următor:

```
?-fapta(metoda_de_miscare(dana, Metoda)) .  
Metoda=mers
```

Această metodă de mișcare a fost moștenită de la faptul că păsările kiwi merg, aceasta fiind o faptă dată în mod explicit în cadrul rețelei. Pe de altă parte, metoda de mișcare a lui Mihai este moștenită de la *clasa pasare* și ea este, prin urmare, zborul:

```
?-fapta(metoda_de_miscare(mihai, Metoda)) .  
Metoda= zbor
```

Unul dintre subdomeniile inteligenței artificiale în care este extrem de folosită reprezentarea cunoștințelor prin intermediul rețelelor semantice este acela al *prelucrării limbajului natural*. Astfel, una dintre cele mai utile reprezentări ale *cunoștințelor lexicale* este cea sub formă de rețea semantică. Avantajele oferite de rețelele semantice în reprezentarea cunoștințelor lexicale sunt multiple. Printre acestea amintim faptul că rețelele semantice ușurează *construcția lexiconului*, permitând moștenirea proprietăților. În același timp, ele furnizează o mulțime foarte bogată de legături între sensurile cuvintelor, ceea ce facilitează *dezambiguizarea*. Aceasta din urmă este, de regulă, realizată printr-o reprezentare ierarhică a sensurilor cuvintelor, cel mai adesea captată tocmai prin intermediul rețelelor semantice. Folosindu-se astfel de ierarhii pot fi definite restricții selecționale și pot fi utilizate aceste constrângeri pentru a reduce numărul de sensuri posibile ale unui cuvânt.

▪ Un alt exemplu de rețea semantică. WordNet

Un exemplu celebru de rețea semantică (din domeniul prelucrării limbajului natural) este WordNet, datorită căreia *procesarea cunoștințelor* a dobândit noi dimensiuni.

WordNet reprezintă în primul rând o *bază de date lexicală interactivă*, dezvoltată în ultimii 15 ani, pentru limba engleză, la Universitatea Princeton, de către un grup de cercetători condus de profesorul George Miller. În același timp, WordNet poate fi privită ca un *dicționar semantic*, deoarece cuvintele sunt localizate pe baza *afinităților conceptuale* cu alte cuvinte, spre deosebire de cazul dicționarelor clasice,

unde cuvintele sunt ordonate alfabetic. Deși este similară unui tezaur, WordNet este mult mai utilă aplicațiilor inteligenței artificiale, întrucât este înzestrată cu o bogată mulțime de relații între cuvinte și sensuri ale cuvintelor. WordNet este implementată în limbajele Prolog, C și Java.

WordNet conține majoritatea substantivelor, verbelor, adjecțivelor și adverbelor limbii engleze, organizate în mulțimi de sinonime numite **synset-uri**. Fiecare *synset* reprezintă un *concept*.

Rețeaua WordNet structurează informația lexicală în termeni de sensuri ale cuvintelor. Ea face corespondența dintre formele tip ale cuvintelor și sensurile acestora utilizând categoria sintactică ca parametru. Astfel, cuvintele aparținând aceleiași categorii sintactice, care pot fi folosite pentru a exprima același înțeles, sunt grupate într-un același synset. Cuvintele polisemantice aparțin mai multor synset-uri. Spre exemplu, cuvântul englezesc *computer* are două sensuri definite în WordNet, ceea ce face ca el să aparțină la două synset-uri diferite, după cum urmează:

(1) {computer, data processor, electronic computer, information processing system}

și

(2) {calculator, reckoner, figurer, estimator, computer}.

În versiunea sa curentă (versiunea 1.6), WordNet conține 129509 cuvinte organizate în 99643 synset-uri, rețeaua utilizând un număr de 229152 noduri. Cuvintele și conceptele sunt legate între ele prin *relații semantice*. Există în total 299711 asemenea relații. Toate aceste numere

sunt însă aproximative, întrucât WordNet continuă să crească. Versiunea 1.7.1 este acum accesibilă în egala măsură, la adresa:

<http://www.cogsci.princeton.edu/~wn/obtain/>

Relațiile semantice se stabilesc între cuvinte, între cuvinte și synset-uri, precum și între synset-uri. Fiecare cuvânt țineste către unul sau mai multe synset-uri, fiecare dintre acestea corespunzând unui anumit sens al cuvântului respectiv. Prin urmare, diferite cuvinte pot ținti către un același sens (synset). Bogăția mulțimii de relații stabilite între synset-uri este ceea ce face ca rețeaua semantică WordNet să fie atât de puternică și de interesantă pentru diverse tipuri de aplicații. Exemple de relații semantice existente în WordNet sunt **sinonimia** (*synonymy*), folosită pentru a forma synset-urile, **hiperonimia** (*hyponymy*) și **hiponimia** (*meronymy*), corespunzând relației de tip *isa* și respectiv relației inverse (*reverse isa*), **meronimia** (*meronymy*), corespunzând relației *parte-din*, relația **cauzală** referitoare la verbe și altele. O importanță deosebită este atașată relațiilor de hiperonimie și de hiponimie ca relații între synset-uri.

Cu ajutorul relației de hiperonimie (sau de tip *isa*) conceptele de *substantiv* și de *verb* sunt structurate sub formă de *ierarhii*. Cele de *adjectiv* și de *adverb* au o structură diferită (*cluster*). În WordNet există 11 ierarhii substantivale și 512 ierarhii verbale. Semantica relației de tip *isa* permite unui concept să moștenească toate proprietățile hiperonimelor sale. În plus, proprietățile tipice ale unui concept sunt enunțate sub formă de glosă atașată fiecărui concept în parte. Fiecare glosă include

o definiție, una sau mai multe explicații suplimentare și unul sau mai multe exemple.

WordNet reprezintă o bază de date lexicală a limbii engleze care a fost adoptată pe scară largă pentru o întreagă varietate de *aplicații practice* din domeniul inteligenței artificiale, cu precădere din subdomeniul procesării limbajului natural. Mulți cercetători care utilizează WordNet, în special în domeniul inteligenței artificiale, consideră că aceasta reprezintă o *bază de cunoștințe* lexicală și o valorifică ca atare. *Procesarea cunoștințelor* a dobândit noi dimensiuni în S.U.A. datorită existenței WordNet. În același timp, comunitatea științifică internațională se arată extrem de interesată de dezvoltarea unor baze de date lexicale de tip WordNet pentru cât mai multe limbi, în încercarea de a crea o *infrastructură ontologică uniformă*. Astfel, întrucât mulțimea de bază a relațiilor care leagă între ele conceptele rămâne aceeași, indiferent de limbă, *algoritmii de inferență* pentru extragerea informației pot rămâne aceiași.

Possiblele aplicații ale WordNet în cele mai variate domenii (*regăsirea informației, extragerea informației, dezambiguizarea, generarea limbajului natural, învățarea, dicționarele electronice, achiziția de cunoștințe* și.a.) sunt citate în peste 300 de lucrări științifice.

Este de menționat faptul că, la mijlocul anilor '90, datorită multiplelor aplicații dezvoltate pe baza WordNet, a fost puternic resimțită nevoia de a se crea baze de date asemănătoare și pentru alte limbi, în special pentru cele europene. Un imens efort științific și finanțier a fost lansat în Europa Occidentală, pentru a se crea aşa-numita **EuroWordNet**, utilizând varianta americană WordNet ca model. Acest efort științific s-a

concretizat în anul 1996, în cadrul proiectului de cercetare - dezvoltare **EuroWordNet**, sub conducerea Universității din Amsterdam:

<http://www.hum.uva.nl/~ewn/>

În prezent există câte o bază de date lexicală de tip WordNet pentru limbile daneză, italiană și spaniolă (fiecare aflată în continuă îmbunătățire) și se lucrează la unele similare pentru limbile germană, franceză și estoniană. Tot în prezent se pune problema creării unor astfel de baze de date lexicale interactive pentru limbile din Europa Centrală și de Est, folosindu-se varianta WordNet a limbii engleze ca model și adaptând-o specificului fiecărei limbi în parte. Proiectul **BalkaNet**, finanțat de Comisia Europeană, se ocupă în prezent de aceste limbi:

<http://www.ceid.upatras.gr/Balkanet/>

Eforturile cercetătorilor (informaticienilor) se concentrează și asupra problemei generării automate a unor baze de date de tip WordNet corespunzătoare diverselor limbii, generare care să pornească de la rețeaua semantică WordNet a limbii engleze. În cazul limbii române acest studiu a fost realizat, referitor la substantivele și adjectivele românești, de către echipa RORIC-LING de la Universitatea din București, în cadrul proiectului **BALRIC-LING** (finanțat tot de Comisia Europeană) și este descris în pagina de web a acestui proiect:

<http://phobos.cs.unibuc.ro/roric>

4.4.2. Cadre

În reprezentarea bazată pe cadre faptele sunt concentrate în jurul obiectelor. Cuvântul “obiect” are aici fie sensul unui obiect fizic concret, fie al unui concept mai abstract, cum ar fi o clasă de obiecte sau chiar o situație.

Un *cadru* este o structură de date ale cărei componente sunt numite *slot-uri* (tăietură, deschizătură). Sloturile pot găzdui informații de diverse naturi. Aici se pot găsi valori simple, referințe la alte slot-uri sau chiar proceduri care pot calcula valoarea slot-ului pe baza altor informații. Un slot poate să fie vid, el putând fi umplut prin mecanismul de inferență. Ca și în cazul rețelelor semantice, cel mai frecvent principiu de inferență este moștenirea proprietăților. Atunci când un cadru reprezintă o clasă de obiecte (albatros) iar un alt cadru reprezintă o superclasă a acestei clase (pasăre), cadrul corespunzător clasei poate moșteni valori de la cadrul reprezentând superclasa.

Un cadru, prin urmare, este o colecție de atrbute (slot-uri) și de valori asociate (eventual și de constrângeri asupra acelor valori), care descriu o anumită entitate a lumii. Uneori, un cadru descrie o entitate într-un sens absolut, alteori el reprezintă acea entitate numai dintr-un anumit punct de vedere. Un cadru unic este arareori util. În schimb, se pot construi întregi sisteme de cadre cu ajutorul unor colecții de cadre care sunt conectate între ele prin faptul că valoarea unui atrbut al unui anumit cadru este un alt cadru. În particular, este util să se atribuie cât mai multă structură atât nodurilor, cât și legăturilor unei rețele. Deși actualmente nu se face o distincție clară între o rețea semantică și un

sistem de cadre, putem spune că, pe masură ce sistemul dobândește mai multă structură, el se apropiie din ce în ce mai mult de un aşa-numit *sistem de cadre*.

Preluând același exemplu din [1] vom plasa o serie de cunoștințe despre păsări în cadre, după cum urmează:

CADRU: pasare

un_fel_de: animal

metoda_de_miscare: zbor

activ: ziua

Acest cadru reprezintă clasa păsărilor. Iată cadrele pentru două subclase ale acestei clase - albatros și kiwi:

CADRU: albatros

un_fel_de: pasare

culoare: alb_si_negru

marime: 115

CADRU: kiwi

un_fel_de: pasare

metoda_de_miscare: mers

activ: noaptea

culoare: maro

marime: 40

Albatros, ca element tipic al clasei păsărilor, moștenește metoda de mișcare (zborul), precum și faptul că este activ în timpul zilei, de la cadrul pasare. Din această cauză, nu se menționează nimic relativ la **metoda_de_miscare** și **activ** în cadrul referitor la albatros. Pe de altă

parte, kiwi este o pasăre atipică și nu respectă valorile pentru **metoda_de_miscare** și **activ** ale clasei păsărilor.

Un cadru se poate referi și la o anumită *instantiere* a unei clase, ca în cazul albatrosului numit Mihai:

CADRU: Mihai
instantiere_pentru: albatros
marime: 120

Se observă faptul că, în cazul primelor cadre, relația **un_fel_de** este o relație între o clasă și o superclasă, în timp ce relația **instantiere_pentru** din ultimul exemplu este o relație între un membru (element) al unei clase și clasa respectivă.

Informația inserată în cadre poate fi reprezentată în Prolog ca o mulțime de fapte, câte o faptă Prolog corespunzând fiecărei valori a unui slot. Formatul pentru aceste fapte ales în [1] și pe care îl prezentăm aici este următorul:

Nume_cadru(Slot,Valoare).

Avantajul acestui format este acela că toate faptele referitoare la un anumit cadru sunt colectate laolaltă de către relația a cărei denumire este chiar numele cadrului. Două exemple de cadre reprezentate în acest format sunt următoarele:

```
%Cadrul kiwi:  

kiwi(un_fel_de,pasare).  

kiwi(metoda_de_miscare,mers).  

kiwi(activ,noaptea).  

kiwi(marime,40).  

kiwi(culoare,maro).
```

```
%Cadrul mihai:  
mihai.instantiere_pentru,albatros).  
mihai.marime,120).
```

Pentru a putea folosi o astfel de mulțime de cadre este necesară existența unei proceduri pentru regăsirea faptelor referitoare la valorile corespunzătoare slot-urilor. O procedură Prolog cu acest rol este propusă în [1] ca fiind de forma

```
valoare(Cadru,Slot,Valoare)
```

unde **Valoare** este valoarea corespunzătoare slot-ului **Slot** în cadrul **Cadru**. Dacă slot-ul este plin - valoarea sa este dată în mod explicit în cadrul - atunci aceasta este valoarea; altfel, această valoare poate fi obținută prin inferență, de pildă ca o consecință a moștenirii proprietăților. Pentru a găsi o valoare prin moștenire, trebuie să ne deplasăm de la cadrul curent la un cadrul mai general, în conformitate cu relația **un_fel_de** sau cu relația **instantiere_pentru** dintre cadre. În acest fel se ajunge la un “cadrupărinte”, iar valoarea poate fi găsită în mod explicit în acest cadrupărinte sau prin continuarea procesului de moștenire.

Acest proces de regăsire a informației din cadre, fie că este o regăsire directă, fie că este o regăsire prin moștenire, poate fi exprimat în Prolog după cum urmează:

```
valoare(Cadru,Slot,Valoare) :-  
    %valoare regasita in mod direct  
    Interogare=..[Cadru,Slot,Valoare],  
    call(Interogare), !.
```

```
valoare(Cadru,Slot,Valoare) :-
```

```
%un cadru mai general
parinte(Cadru,CadruParinte) ,
valoare(CadruParinte,Slot,Valoare) .

parinte(Cadru,CadruParinte) :- 
    (Interogare=..[Cadru,un_fel_de,CadruParinte] ;
     Interogare=..[Cadru,instantiere_pentru,
                    CadruParinte]),
     call(Interogare).
```

Interrogarea Prologului se poate face atunci în felul următor:

```
?-valoare(mihai,activ,TimpMihai).
TimpMihai = ziua

?-valoare(kiwi,activ,TimpKiwi).
TimpKiwi = noaptea
```

Inferența între cadre este un proces extrem de subtil, care ridică diverse probleme ce nu vor fi tratate aici, ele neconstituind obiectul acestui curs. Dintre acestea amintim, în primul rând, cazul mai complicat de inferență în care, corespunzător unui slot, este dată o *procedură de calcul* a valorii, în locul valorii efective. O altă subtilitate a inferenței între cadre o constituie aşa-numita *moștenire multiplă*. Această problemă se ridică atunci când un cadru are mai multe cadre părinte (conform relației *instantiere_pentru* sau relației *un_fel_de*). În acest caz o valoare a unui slot moștenită poate proveni de la mai mult de un cadru părinte, problema care se pune fiind aceea a adoptării valorii corecte. Soluția propusă în [1], de pildă, este aceea de a se prelua prima valoare

întâlnită, găsită în urma efectuării unei *căutări de tip depth-first* printre cadrele care ar putea furniza această valoare. Alte soluții există, în egală măsură, principala concluzie care se desprinde în urma acestei succinte prezentări fiind aceea că *repräsentarea cunoștințelor și procesul de căutare se află într-o strânsă interdependentă*.

4.5. Considerații finale

Sistemele de reprezentare a cunoștințelor amintite sau discutate de noi până în prezent au fost: structurile de tip *slot-and-filler*, sistemele bazate pe reguli, rețelele semantice și cadrele. Capitolul 5 al lucrării de față va introduce sistemele de raționament statistic și se va ocupa de două tipuri de abordări statistice pentru sisteme de raționament incert (factorii de certitudine în sistemele bazate pe reguli și, respectiv, rețelele Bayesiene). Indiferent de tipul de structură aleasă pentru reprezentarea cunoștințelor, a devenit însă evident faptul că aceste sisteme trebuie nu numai să încorporeze cunoștințe, ci, în egală măsură, să furnizeze o mulțime de proceduri de inferență de bază, cum ar fi cea referitoare la moștenirea proprietăților.

Structurile de tip slot-and-filler s-au dovedit foarte valoroase pentru memorarea și regăsirea eficientă a cunoștințelor în programele inteligenței artificiale. Ele au însă o performanță scăzută atunci când se pune problema reprezentării unor aserțiuni de tipul regulilor “dacă x, y și z , atunci trage concluzia w ”. Calculul predicatelor este mult mai adekvat pentru reprezentarea unor asemenea aserțiuni, dar prezintă, la rândul său, dezavantajul de a fi inefficient pentru raționamentul general care le

utilizează. Reprezentările de tip slot-and-filler sunt mai orientate către semantică, ceea ce face ca procedurile lor de raționament să fie mai variate, mai eficiente și mai strâns legate de tipuri specifice, particulare de cunoștințe. Pe de altă parte, în cadrul structurilor slot-and-filler este dificil să se exprime aserțiuni mai complexe decât moștenirea.

Rețelele semantice sunt proiectate cu scopul principal de a capta relații, legături și raporturi semantice între entități. Ele sunt, de regulă, utilizate împreună cu o mulțime de reguli de inferență, proiectate special pentru a putea trata în mod corect tipurile de arce specifice rețelei.

Sistemele bazate pe cadre sunt în general mai structurate decât rețelele semantice și ele conțin o mulțime chiar mai mare de reguli de inferență specializezate, inclusiv cele care implementează o întreagă matrice de reguli de moștenire implicate, precum și alte proceduri, cum ar fi verificarea consistenței.

EMI-UNIBUC

CAPITOLUL 5

RAȚIONAMENT STATISTIC

Diversele tehnici de reprezentare a cunoștințelor pot fi augmentate cu ajutorul unor măsuri statistice care descriu niveluri ale dovezilor existente, precum și ale convingerilor unui agent. În cele ce urmează, vom prezenta și folosi unele instrumente furnizate de statistica Bayesiană pentru a realiza această augmentare, atât de necesară în descrierea părerilor care nu reprezintă certitudini, dar în favoarea cărora există suficiente dovezi.

Se poate spune că principalul avantaj al raționamentului de tip probabilist, față de raționamentul de tip logic, rezidă în faptul că cel dintâi permite agentului să ajungă la decizii raționale chiar și atunci când nu există suficientă informație care să dovedească că o acțiune dată va avea efectul dorit.

5.1. Abordări statistice pentru sisteme de raționament incert

Un important țel al multor sisteme de rezolvare a problemelor este acela de a colecționa probe, pe măsură ce sistemul avansează și de a-și modifica comportamentul pe baza acestor dovezi. Statistica Bayesiană este o teorie statistică care poate modela acest tip de comportament.

Dacă notăm cu H evenimentul ca ipoteza (notată H) să fie adevărată, iar cu E evenimentul obținut prin observarea "dovezii" E ,

atunci conceptul fundamental al statisticii Bayesiene poate fi considerat ca fiind acela de *probabilitate condiționată*,

$$P(H|E),$$

reprezentând probabilitatea ca ipoteza H să fie adevărată, atunci când se observă dovada E . Pentru a calcula această probabilitate trebuie luate în considerație probabilitatea prealabilă sau *a priori* a lui H (i.e. probabilitatea pe care am atribui-o lui H în lipsa oricăror probe), precum și gradul până la care E furnizează dovezi în favoarea lui H . Acest lucru se realizează prin definirea unui univers conținând o mulțime exhaustivă de ipoteze H_i , care se exclud reciproc și între care încercăm să discernem. Fie

E = dovezile obținute printr-un experiment auxiliar.

$P(H_i)$ = probabilitatea *a priori* ca ipoteza H_i să fie adevărată.

$P(E|H_i)$ = probabilitatea de a observa dovezile E (probabilitatea să se realizeze E), atunci când ipoteza H_i este adevărată.

$P(H_i|E)$ = probabilitatea *a posteriori* ca ipoteza H_i să fie adevărată, fiind date dovezile E (știind că s-a realizat E).

Ca modalitate de calcul, probabilitatea *a posteriori* $P(H_i|E)$ se obține în mod Bayesian. Dacă notăm prin k numărul ipotezelor posibile, atunci formula lui Bayes, în varianta necondiționată, calculează această probabilitate *a posteriori* în felul următor (vezi Anexa 1):

$$P(H_i | E) = \frac{P(E | H_i) \cdot P(H_i)}{\sum_{n=1}^k P(E | H_n) \cdot P(H_n)}$$

Teorema lui Bayes poate sta la baza raționamentului incert. În general, probabilitatea $P(A|B)$ descrie probabilitatea condiționată a lui A atunci când singurele dovezi de care dispunem sunt reprezentate de B . Dacă însă există și alte dovezi relevante, atunci acestea trebuie și ele luate în considerație. Atunci când sunt date un corp prealabil de dovezi e și o nouă observație E , trebuie calculată probabilitatea condiționată a posteriori $P(H | E, e)$, prin aplicarea formulei lui Bayes în versiune condiționată (vezi Anexa 1). Avem:

$$P(H | E, e) = P(H | E) \cdot \frac{P(e | E, H)}{P(e | E)}$$

Într-o lume arbitrar de complexă, dimensiunea mulțimii repartițiilor de probabilitate multidimensionale, care este necesară pentru a calcula această funcție, crește la valoarea 2^n dacă sunt luate în considerație n propoziții diferite. Așa cum se arată în [8], utilizarea teoremei lui Bayes face ca problema să devină insolubilă din mai multe motive:

- Problema achiziționării de cunoștințe devine insurmontabilă, întrucât trebuie furnizate prea multe probabilități.
- Spațiul necesar pentru memorarea tuturor probabilităților este prea mare.
- Timpul necesar pentru calcularea tuturor probabilităților este prea mare.

În ciuda acestor neajunsuri, statistica Bayesiană constituie o bază atractivă pentru un *sistem de raționament incert*. Din această cauză, au fost dezvoltate câteva mecanisme care îi exploatează puterea, făcând, în același timp, problema rezolvabilă, din punct de vedere computațional. Două dintre aceste mecanisme, asupra cărora ne vom opri în continuare, constau în atașarea unor factori de certitudine regulilor de producție și respectiv în folosirea rețelelor Bayesiene.

5.1.1. Factori de certitudine și sisteme bazate pe reguli

Abordarea pe care o propunem aici a fost preluată din [8] și provine de la sistemul MYCIN [Shortliffe, 1976; Buchanan și Shortliffe, 1984; Shortliffe și Buchanan, 1975], care își propune să recomande terapii adecvate pacienților cu infecții bacteriene. MYCIN este un exemplu de *sistem expert*, care interacționează cu medicul pentru a dobândi datele clinice de care are nevoie.

MYCIN reprezintă majoritatea cunoștințelor sale legate de diagnostic ca pe o mulțime de reguli, fiecare regulă având asociat un factor de certitudine. Sistemul folosește aceste reguli pentru a face un raționament de tip *înlănțuire înapoi* de la scopul său de a detecta organisme semnificative care pot cauza maladiei și până la datele clinice disponibile. De îndată ce identifică asemenea organisme, MYCIN încearcă să selecteze o terapie prin care boala să fie tratată. Pentru a înțelege cum exploatează sistemul informația incertă trebuie stabilit *cum combină acesta estimările de certitudine ale fiecărei reguli în parte pentru a produce o estimare finală a certitudinii concluziilor sale.*

O întrebare naturală care se ridică, în egală măsură, având în vedere observațiile anterioare privind insolubilitatea spre care conduce raționamentul Bayesian pur, este următoarea: ce compromisuri trebuie să facă tehnica MYCIN și care sunt riscurile asociate acestor compromisuri?

Un *factor de certitudine* (notat aici prin $FC[h, e]$) sau, mai simplu, prin FC) este definit în termenii a două componente:

- $MI[h, e]$ - o măsură (între 0 și 1) a încrederei în ipoteza h fiind dată dovada e . MI măsoară gradul până la care dovezile existente susțin ipoteza. Valoarea este 0 dacă aceste dovezi esuează în susținerea ipotezei.
- $MN[h, e]$ - o măsură (între 0 și 1) a neîncrederei în ipoteza h fiind dată dovada e . MN măsoară gradul până la care dovezile existente susțin *negația ipotezei*. Valoarea este 0 dacă aceste dovezi susțin ipoteza.

Fiind date aceste două măsuri, putem defini factorul de certitudine ca fiind

$$FC[h, e] = MI[h, e] - MN[h, e]$$

Întrucât orice dovadă particulară fie susține, fie neagă o ipoteză și întrucât fiecare regulă MYCIN corespunde unei anumite dovezi (deși aceasta ar putea fi o dovadă compusă), este suficient un singur număr corespunzător fiecărei reguli pentru a defini atât MI , cât și MN și, prin urmare, factorul de certitudine, FC .

Factorii de certitudine ai regulilor MYCIN sunt furnizați de către experții care scriu aceste regule. Pe măsură însă ce sistemul raționează, *acești factori de certitudine trebuie combinați*, pentru a reflecta dovezi

multiple și reguli multiple aplicate aceleiași probleme. Cele trei tipuri de combinații care trebuie luate în considerație sunt reflectate în Fig. 5.1. Astfel, în Fig. 5.1 (a) mai multe reguli furnizează dovezi referitoare la o singură ipoteză. În Fig. 5.1 (b) mai multe propoziții trebuie luate în considerație împreună pentru a ne forma o părere. În Fig. 5.1 (c) output-ul corespunzător unei reguli furnizează input pentru o alta.

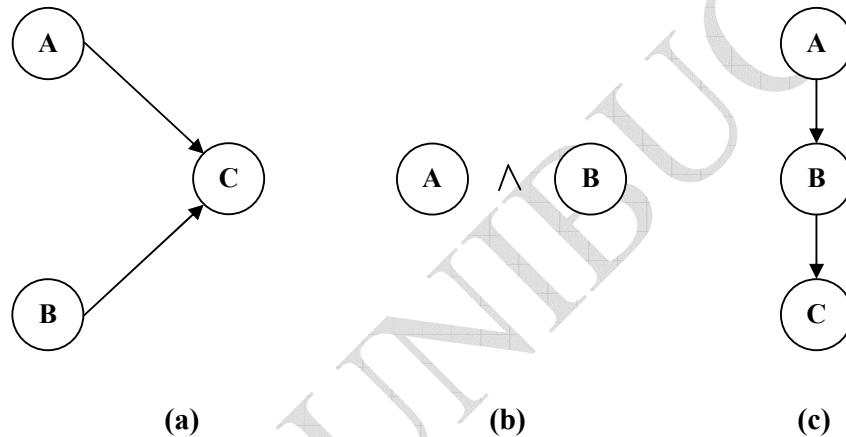


Fig. 5.1

Înainte de a stabili formulele care trebuie folosite pentru a realiza aceste combinații, trebuie avute în vedere anumite *proprietăți pe care funcția de combinare trebuie să le satisfacă*, și anume:

- Întrucât ordinea în care dovezile sunt colectate este arbitrară, funcțiile de combinare trebuie să fie comutative și asociative.
- Până când este atinsă certitudinea, dovezile suplimentare ar trebui să crească valoarea lui MI . (În mod similar, dovezile care infirmă ipoteza ar trebui să crească valoarea lui MN).

- Dacă sunt înlănțuite inferențe incerte, atunci rezultatul ar trebui să fie mai puțin cert decât fiecare inferență în parte.

Acceptând necesitatea acestor proprietăți¹⁹, vom considera mai întâi situația din Fig. 5.1 (a), în care *mai multe probe se combină pentru a se determina factorul de certitudine al unei ipoteze*. Măsura încrederii și a neîncrederii într-o ipoteză, fiind date două observații, s_1 și s_2 , se calculează după cum urmează:

$$MI[h, s_1 \wedge s_2] = \begin{cases} 0, & \text{daca } MN[h, s_1 \wedge s_2] = 1 \\ MI[h, s_1] + MI[h, s_2] \cdot (1 - MI[h, s_1]), & \text{altfel} \end{cases}$$

$$MN[h, s_1 \wedge s_2] = \begin{cases} 0, & \text{daca } MI[h, s_1 \wedge s_2] = 1 \\ MN[h, s_1] + MN[h, s_2] \cdot (1 - MN[h, s_1]), & \text{altfel} \end{cases}$$

$FC[h, s_1 \wedge s_2]$ se calculează pe baza lui $MI[h, s_1 \wedge s_2]$ și $MN[h, s_1 \wedge s_2]$.

Se observă că, dacă se coroborează mai multe dovezi ale aceleiași ipoteze, atunci valoarea absolută a lui FC va crește. Dacă sunt introduse dovezi conflictuale, atunci valoarea absolută a lui FC va descrește.

În situația din Fig. 5.1 (b) este necesar *calculul factorului de certitudine al unei conjuncții de ipoteze*. FC se calculează pe baza valorilor MI și MN . Formulele folosite de MYCIN pentru calculul lui MI , corespunzător unei conjuncții și respectiv unei disjuncții de ipoteze, sunt următoarele:

$$MI[h_1 \wedge h_2, e] = \min(MI[h_1, e], MI[h_2, e])$$

¹⁹ Descrierea sistemului MYCIN este făcută aici conform [8].

$$MI[h_1 \vee h_2, e] = \max(MI[h_1, e], MI[h_2, e])$$

MN poate fi calculat în mod analog.

Fig. 5.1 (c) prezintă cazul în care *regulile sunt înlănuite laolaltă*, ceea ce are ca rezultat faptul că ieșirea incertă a uneia dintre reguli trebuie să constituie intrarea alteia. Soluția dată acestei probleme se va ocupa și de cazul în care trebuie atribuită o măsură a incertitudinii intrărilor inițiale. Aceste situații sunt frecvente și corespund cazurilor în care dovezile provin ca urmare a unui experiment sau al unui test de laborator ale cărui rezultate nu sunt suficient de exacte. În aceste cazuri, factorul de certitudine al unei ipoteze trebuie să ia în considerație atât tăria cu care probele sugerează ipoteza, cât și nivelul de încredere în aceste probe. MYCIN furnizează o regulă de înlănuire care este definită după cum urmează.

Fie $MI'[h, s]$ măsura încrederii în h atunci când suntem absolut siguri de validitatea lui s . Fie e dovada care ne-a determinat să credem în s (spre exemplu, măsurările efectuate cu ajutorul instrumentelor de laborator sau rezultatele aplicării altor reguli). Atunci:

$$MI[h, s] = MI'[h, s] \cdot \max(0, FC[s, e])$$

Întrucât valorile *FC inițiale* din sistemul MYCIN sunt date de către experți care scriu regulile, nu este necesar să se formuleze o definiție mai exactă a factorului de certitudine decât cea dată până acum. Autorii sistemului inițial au dat însă o astfel de definiție, exprimând pe MI (care poate fi privit ca o descreștere proporțională a neîncrederii în h ca rezultat al observării lui e) după cum urmează:

$$MI[h, e] = \begin{cases} 1, & \text{daca } P(h) = 1 \\ \frac{\max[P(h|e), P(h)] - P(h)}{1 - P(h)}, & \text{altfel} \end{cases}$$

În mod similar, MN este descreșterea proporțională a încrederii în h ca rezultat al observării lui e :

$$MN[h, e] = \begin{cases} 1, & \text{daca } P(h) = 0 \\ \frac{\min[P(h|e), P(h)] - P(h)}{-P(h)}, & \text{altfel} \end{cases}$$

Acstea definiții s-au dovedit a fi incompatibile cu viziunea Bayesiană asupra probabilității condiționate. Modificări minore le fac însă compatibile [Heckerman, 1986]. În particular, putem redefini MI astfel:

$$MI[h, e] = \begin{cases} 1, & \text{daca } P(h) = 1 \\ \frac{\max[P(h|e), P(h)] - P(h)}{(1 - P(h)) \cdot P(h|e)}, & \text{altfel} \end{cases}$$

Definiția lui MN trebuie schimbată în mod similar.

Cu aceste reinterpretări nu mai există nici un conflict fundamental între tehniciile MYCIN și cele sugerate de statistica Bayesiană, MYCIN nefiind un sistem care conduce la insolubilitate.

Fiecare FC dintr-o regulă MYCIN reprezintă contribuția unei reguli individuale la încrederea pe care MYCIN o acordă unei anumite ipoteze. Într-un anume sens, acest factor reprezintă o probabilitate condiționată, $P(H|E)$. Dar, într-un sistem Bayesian pur, $P(H|E)$ descrie probabilitatea condiționată a lui H atunci când singurele dovezi relevante sunt date de E . Dacă există și alte probe, atunci trebuie luate în

considerație repartiției de probabilitate multidimensionale. Acesta este momentul în care MYCIN se distanțează de un sistem Bayesian pur, devenind mai eficient în execuție, dar cu riscul de a avea un comportament mai puțin intuitiv. În particular, *formulele MYCIN pentru cele trei situații din Fig. 5.1 fac presupunerea că toate regulile sunt independente*. Sarcina garantării independenței revine expertului care scrie regulile. Fiecare dintre scenariile combinate ale Fig. 5.1 devine vulnerabil atunci când această presupunere de independență este încălcată.

Abordarea care utilizează factori de certitudine și sisteme bazate pe reguli face, prin urmare, *presupuneri puternice privitoare la independență*, datorită cărora este ușor de folosit în practică. În același timp, *regulile trebuie scrise în aşa fel încât ele să reflecte dependențele importante*. Abordarea stă la baza multor programe, precum MYCIN, ca și a unei game largi de sisteme diferite, care au fost construite pe platforma EMYCIN [van Melle et al., 1981]. Platforma EMYCIN constituie o generalizare (numită *shell*) a lui MYCIN, în cadrul căreia au fost înălțurate toate regulile specifice domeniului. Unul dintre motivele pentru care acest cadru de lucru este util, în ciuda limitărilor sale, este acela că, într-un sistem robust, valorile exacte care sunt folosite nu au o importanță prea mare. Această abordare pare, de asemenea, a imita suficient de bine [Shultz et al., 1989] felul în care oamenii manipulează certitudinile.

5.1.2. Rețele Bayesiene

Factorii de certitudine au fost descriși ca un mecanism de reducere a complexității unui sistem de raționament Bayesian prin intermediul unor aproximări ale formalismului. O *abordare alternativă* este cea bazată pe *rețele Bayesiene* [Pearl, 1988], abordare care conservă formalismul bazându-se, în schimb, pe modularitatea universului a cărui modelare se încearcă.

Ideea de bază a acestei abordări este aceea că, pentru a descrie lumea reală, nu este necesară folosirea unei tabele imense de repartiții de probabilitate multidimensionale în care să fie listate probabilitățile tuturor combinațiilor posibile de evenimente. Majoritatea evenimentelor sunt independente de celealte, astfel încât interacțiunile dintre ele nu trebuie luate în considerație. În schimb, se poate folosi o *repräsentare locală* având rolul de a descrie grupuri de evenimente care interacționează. Astfel, în contextul folosirii regulii lui Bayes, relațiile de independență condiționată dintre variabile pot reduce substanțial numărul probabilităților condiționate care trebuie specificate. Structura de date folosită cel mai frecvent pentru a reprezenta dependența dintre variabile și pentru a da o specificație concisă a repartiției de probabilitate multidimensionale este *rețeaua Bayesiană*.

Definiția 5.1

O *rețea Bayesiană* este un graf cu următoarele proprietăți:

- (1) Graful este direcționat și aciclic.
- (2) O mulțime de variabile aleatoare constituie nodurile rețelei.

(3) O mulțime de arce direcționate conectează perechi de noduri.

În mod intuitiv, semnificația unui arc direcționat de la nodul X la nodul Y este aceea că X are o influență directă asupra lui Y .

(4) Fiecare nod îi corespunde un tabel de probabilități condiționate care cuantifică efectele pe care părinții le au asupra nodului respectiv. (Părinții unui nod sunt toate acele noduri din care pleacă arce direcționate înspre acesta).

Un expert în domeniu poate, de obicei, să decidă relativ ușor relațiile directe de dependență condiționată care sunt valabile în domeniu, acest lucru fiind mult mai ușor decât specificarea probabilităților propriu-zise. După stabilirea topologiei rețelei Bayesiene, nu mai este necesară decât specificarea probabilităților condiționate ale acelor noduri care participă în dependențe directe, urmând ca acestea să fie apoi folosite în calculul oricărora alte probabilități.

În mod concret, se construiește un graf direcționat aciclic care reprezintă *relațiile de cauzalitate* dintre variabile. Variabilele dintr-un astfel de graf pot fi propoziționale (caz în care pot lua valorile TRUE sau FALSE) sau pot fi variabile care primesc valori de un alt tip (spre exemplu o temperatură a corpului sau măsurători făcute de către un dispozitiv de diagnosticare).

În Fig. 5.2 este prezentat un astfel de graf al cauzalității (rețea). Un asemenea graf ilustrează relațiile de cauzalitate care intervin între nodurile pe care le conține. Pentru a-l putea folosi ca bază a unui raționament de tip probabilist sunt necesare însă mai multe informații. În particular este necesar să cunoaștem, pentru o valoare a unui nod

părinte, ce dovezi sunt furnizate referitor la valorile pe care le poate lua nodul fiu.

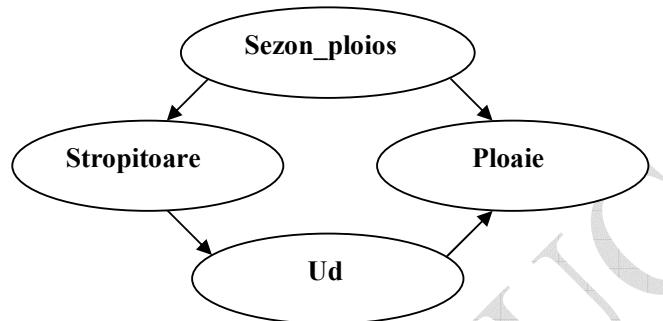


Fig. 5.2

Probabilitățile condiționate corespunzătoare pot fi date sub forma unui tabel de tipul:

Atribut	Probabilitate
$p(Ud Stropitoare, Ploaie)$	0.95
$p(Ud Stropitoare, \neg Ploaie)$	0.9
$p(Ud \neg Stropitoare, Ploaie)$	0.8
$p(Ud \neg Stropitoare, \neg Ploaie)$	0.1
$p(Stropitoare Sezon ploios)$	0.0
$p(Stropitoare \neg Sezon ploios)$	1.0
$p(Ploaie Sezon ploios)$	0.9
$p(Ploaie \neg Sezon ploios)$	0.1
$p(Sezon ploios)$	0.5

Din acest tabel, preluat de noi, spre exemplificare, din [8], se poate citi, de pildă, probabilitatea a priori de a avea un sezon ploios ca fiind 0.5. Dacă suntem în timpul unui sezon ploios, probabilitatea de a avea ploaie

într-o noapte dată este 0.9. Dacă sezonul nu este ploios, această probabilitate este numai 0.1.

Pentru a putea folosi o asemenea reprezentare în rezolvarea problemelor este necesar un mecanism de calcul al influenței oricărui nod arbitrar asupra oricărui alt nod. Pentru a obține acest mecanism de calcul este necesar ca graful inițial să fie convertit la un graf nedirecționat, în care arcele să poată fi folosite pentru a se transmite probabilități în oricare dintre direcții, în funcție de locul din care provin dovezile. Este, de asemenea, necesar un mecanism de folosire a grafului care să garanteze transmiterea corectă a probabilităților. Spre exemplu, deși este adevărat că iarba udă poate constitui o doavadă de ploaie și că existența ploii constituie o doavadă în favoarea ierbii ude, trebuie să garantăm faptul că nici un ciclu nu este vreodată traversat astfel încât iarba udă să constituie o doavadă a ploii, care să fie considerată apoi o doavadă a existenței ierbii ude și.a.m.d..

Există trei mari clase de algoritmi [8] care efectuează aceste calcule, ideea care stă la baza tuturor acestor metode fiind aceea că nodurile au domenii limitate de influență. Metoda cea mai folosită este probabil cea a transmiterii mesajelor [Pearl, 1988], abordare care se bazează pe observația că, pentru a calcula probabilitatea unui nod A condiționat de ceea ce se știe despre celelalte noduri din rețea, este necesară cunoașterea a trei tipuri de informații:

- suportul total care sosește în A de la nodurile sale părinte (reprezentând cauzele sale);
- suportul total care sosește în A de la fiile acestuia (reprezentând simptomele sale);

- intrarea în matricea fixată de probabilități condiționate care face legătura dintre nodul A și cauzele sale.

Au fost dezvoltate câteva metode de propagare a mesajelor de primele două tipuri, precum și de actualizare a probabilităților corespunzătoare nodurilor. Structura rețelei determină abordarea care va fi folosită. Spre exemplu, în rețelele unic conectate (aceleia în care există un singur drum între fiecare pereche de noduri), se poate folosi un algoritm mai simplu (a se vedea § 5.1.2.2.1 al lucrării de față) decât în cazul celor conectate multiplu.

5.1.2.1. Relații de independentă condiționată în rețele Bayesiene

O rețea Bayesiană exprimă independentă condiționată a unui nod și a predecesorilor săi, fiind dați părinții acestuia și folosește această independentă pentru a proiecta o metodă de construcție a rețelelor. Pentru a stabili *algoritmi de inferență* trebuie însă să stim dacă se verifică independențe condiționate mai generale. Fiind dată o rețea dorim să putem ”citi” dacă o mulțime de noduri X este independentă de o altă mulțime Y , fiind dată o mulțime de ”noduri dovezi” E . Metoda de stabilire a acestui fapt se bazează pe noțiunea de *separare dependentă de direcție* sau **d-separare**, definită după cum urmează:

Definiția 5.2

O mulțime de noduri E *d-separă* două mulțimi de noduri X și Y dacă orice drum nedirecționat de la un nod din X la un nod din Y este **blocat** condiționat de E .

Definiția 5.3

Fiind dată o mulțime de noduri E , spunem că un drum este *blockat* condiționat de E dacă există un nod Z aparținând drumului, pentru care una dintre următoarele trei condiții se verifică:

- (1) Z aparține lui E și Z are o săgeată a drumului intrând în E și o săgeată a drumului ieșind din E .
- (2) Z aparține lui E și Z are ambele săgeți ale drumului ieșind din E .
- (3) Nici Z și nici vreunul dintre descendenții săi nu aparțin lui E , iar ambele săgeți ale drumului țintesc înspre Z .

Fig. 5.3 ilustrează cele trei cazuri posibile:

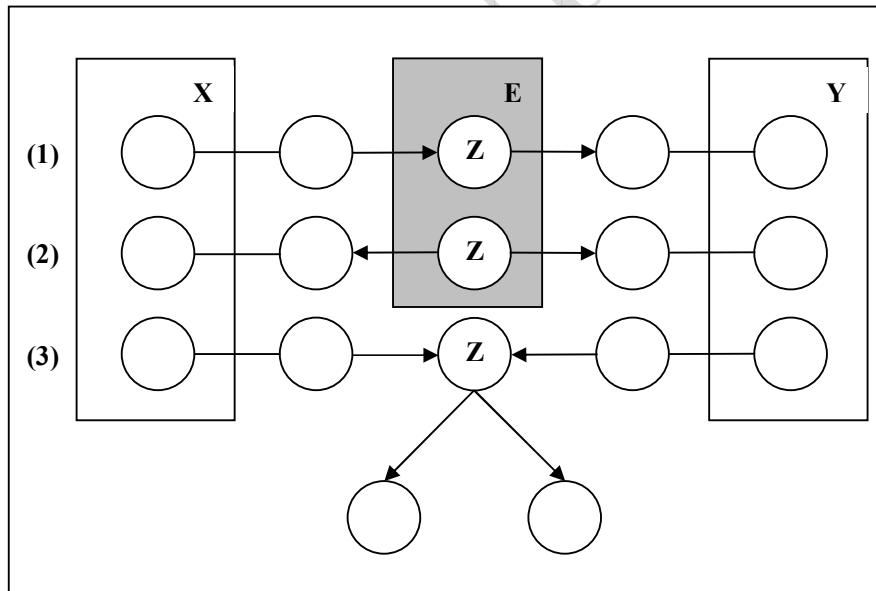


Fig. 5.3

S-a demonstrat că dacă orice drum nedirecționat de la un nod din X la un nod din Y este d-separat de E , atunci X și Y sunt independente condiționat de E . Aceasta este teorema fundamentală a rețelelor Bayesiene, demonstrată de Verma și Pearl. Demonstrația acestei teoreme este suficient de complicată și nu ne vom opri asupra ei în cadrul restrâns al cursului de față. Reținem faptul că noțiunea de d-separare este fundamentală în construcția algoritmilor de inferență. Procesul construirii și folosirii rețelelor Bayesiene nu utilizează d-separarea.

5.1.2.2. Inferență în rețele Bayesiene

Sarcina principală a oricărui sistem probabilist de inferență este aceea de a calcula probabilități a posteriori de tipul

$$P(\text{Interogare}|\text{Dovezi})$$

corespunzător unei mulțimi de *variabile de interogare* condiționat de valori exacte ale unor *variabile dovezi*. (În exemplul considerat, *Ud* este o variabilă de interogare, iar *Stropitoare* și *Sezon_ploios* ar putea fi variabile dovezi).

Rețelele Bayesiene sunt suficient de flexibile pentru ca orice nod să poată servi fie ca o variabilă de interogare, fie ca o variabilă dovdă. În general, un agent primește valori ale variabilelor dovezi de la senzorii săi (sau în urma altor raționamente) și întreabă despre posibilele valori ale altor variabile astfel încât să poată decide ce acțiune trebuie întreprinsă.

Literatura de specialitate discută patru tipuri distincte de inferență, care poate fi realizată de rețelele Bayesiene:

- inferență de tip diagnostic (de la efecte la cauze);
- inferență cauzală (de la cauze la efecte);
- inferență intercauzală (între cauze ale unui efect comun);
- inferențe mixte (reprezentând combinații a două sau mai multe dintre inferențele anterioare).

Exemplu: Setând efectul *Stropitoare* la valoarea ”adevărat” și cauza *Sezon_ploios* la valoarea ”fals”, ne propunem să calculăm

$$P(Ud|Stropitoare, \neg Sezon_ploios).$$

Aceasta este o inferență mixtă, care reprezintă o utilizare simultană a inferenței de tip diagnostic și a celei cauzale.

Rețelele Bayesiene s-au dovedit extrem de folositoare în realizarea mai multor tipuri de sarcini. Paragraful următor se va concentra asupra celei mai frecvente și mai utile dintre acestea, și anume determinarea probabilităților condiționate a posteriori ale variabilelor de interogare.

5.1.2.2.1. Un algoritm pentru răspunsul la interogări

În cele ce urmează, ne propunem să stabilim un algoritm de calcul al probabilităților condiționate a posteriori ale variabilelor de interogare. Algoritmul găsit va fi de tip înlănțuire înapoi prin faptul că pleacă de la variabila de interogare și urmează drumurile de la acel nod până la nodurile dovezi. Datorită complicațiilor care se pot naște atunci când două drumuri diferite converg la același nod, algoritmul pe care îl vom prezenta și care este preluat din [9] se referă numai la *rețele unic conectate*, cunoscute și sub denumirea de **polyarbori**. Reamintim faptul

că, în astfel de rețele, există cel mult un drum nedirecționat între oricare două noduri ale rețelei. Algoritmii pentru rețele generale, asupra cărora nu ne vom opri în cadrul restrâns al acestui curs, vor folosi algoritmii referitor la polyarbori ca principală subrutină.

Fig. 5.4 prezintă o rețea generică unic conectată. În această rețea nodul X are părinții $\mathbf{U} = U_1 \dots U_m$ și fișii $\mathbf{Y} = Y_1 \dots Y_n$. Corespondent fiecărui fiu și fiecărui părinte a fost desenat un dreptunghi care include toți descendenții nodului și toți strămoșii lui (cu excepția lui X). Proprietatea de *unică conectare* înseamnă că toate dreptunghiurile sunt disjuncte și că nu există legături care să le conecteze între ele. Se presupune că X este variabilă de interogare și că există o mulțime E de variabile dovezi²⁰. Se urmărește calcularea probabilității condiționate

$$P(X|E)$$

În mod evident, dacă însuși X este o variabilă dovedă din E , atunci calcularea lui $P(X|E)$ este banală. Vom presupune, de aceea, că X nu aparține lui E .

Rețeaua din Fig. 5.4 este partită în conformitate cu părinții și cu fișii variabilei de interogare X . Pentru a concepe un algoritm, va fi util să ne putem referi la diferite porțiuni ale dovezilor. Prima distincție pe care o vom face este următoarea:

- E_X^+ reprezintă **suportul cauzal** pentru X - variabilele dovezi aflate "deasupra" lui X , care sunt conectate la X prin intermediul părinților săi.

²⁰ $E = \{E_X^-, E_X^+\} = \{U_1, \dots, U_m, Y_1, \dots, Y_n\}$, mulțime de variabile aleatoare discrete.

- E_X^- reprezintă **suportul probatoriu** pentru X - variabilele dovezi aflate "dedesubtul" lui X și care sunt conectate la X prin intermediul fiilor săi.

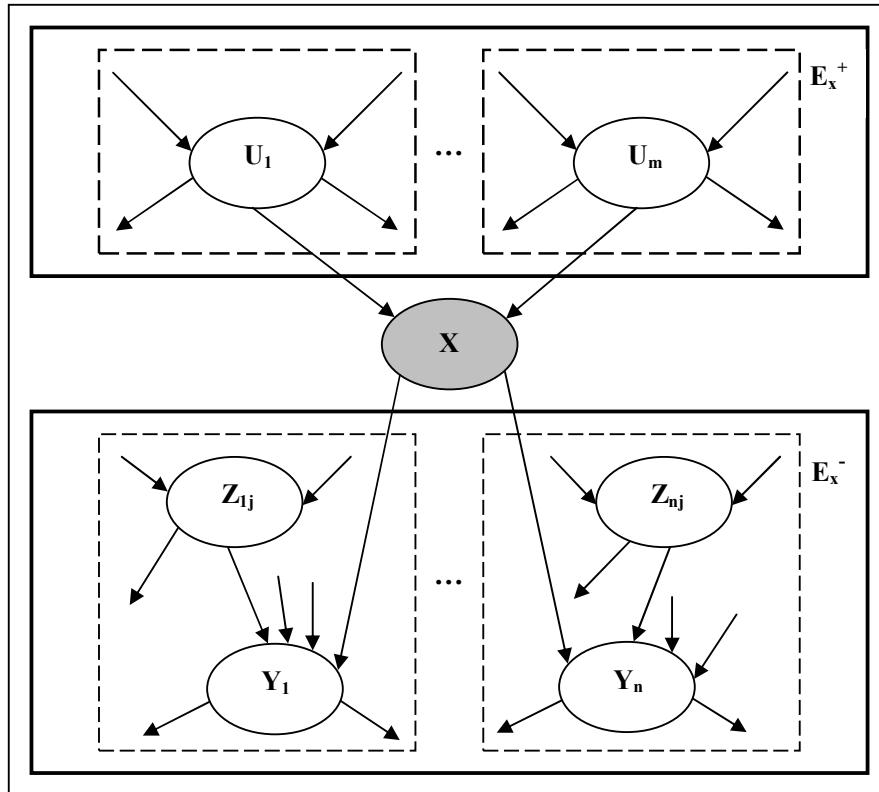


Fig. 5.4

Notația $E_{U_i \setminus X}$ va fi folosită pentru a se face referire la toate dovezile conectate cu nodul U_i , mai puțin cele prin drumul de la X . Cu această ultimă notație, putem partiționa pe E_X^+ în $E_{U_1 \setminus X}, \dots, E_{U_m \setminus X}$ și deci putem exprima pe E_X^+ sub forma următoare:

$$E_X^+ = \bigcup_{i=1}^m E_{U_i \setminus X} \quad (1)$$

În mod similar, $E_{Y_i \setminus X}$ semnifică mulțimea tuturor dovezilor conectate la Y_i prin intermediul părintilor săi, cu excepția lui X . Mulțimea E_X^- poate fi deci partaționată în mulțimile $E_{Y_1 \setminus X}, \dots, E_{Y_n \setminus X}$ și este de forma

$$E_X^- = \bigcup_{i=1}^n E_{Y_i \setminus X} \quad (2)$$

Se observă că mulțimea tuturor dovezilor E poate fi scrisă ca E_X (toate dovezile conectate la X) și ca $E_{X \setminus}$ (toate dovezile conectate la X , fără excepție).

Strategia generală pentru calculul lui $P(X | E)$ este atunci următoarea:

- Exprimă $P(X | E)$ în termenii contribuțiilor lui E_X^+ și E_X^- .
- Calculează contribuția mulțimii E_X^+ calculând efectul ei asupra părintilor lui X și apoi transmițând acest efect lui X . (Calcularea efectului asupra fiecărui părinte al lui X este o secvență recursivă a problemei calculării efectului asupra lui X).
- Calculează contribuția mulțimii E_X^- calculând efectul ei asupra fiilor lui X și apoi transmițând acest efect lui X . (Calcularea efectului asupra fiecărui fiu al lui X reprezintă o secvență recursivă a problemei calculării efectului asupra lui X).

Totalitatea dovezilor, E , constă din dovezile aflate "deasupra" lui X și din cele aflate "dedesubtul" lui X , întrucât s-a făcut presupunerea că X însuși nu se află în E . Prin urmare, avem

$$P(X|E) = P(X|E_X^-, E_X^+)$$

Pentru a separa contribuțiile lui E_X^+ și E_X^- , vom aplica versiunea condiționată a regulii lui Bayes, păstrând pe E_X^- ca doavadă fixată în fundal:

$$P(X|E_X^-, E_X^+) = \frac{P(X|E_X^+)P(E_X^-|X, E_X^+)}{P(E_X^-|E_X^+)}$$

Întrucât X d-separă în cadrul rețelei pe E_X^+ de E_X^- , putem folosi independența condiționată pentru a simplifica al doilea termen al numărătorului. De asemenea, putem trata $1/P(E_X^-|E_X^+)$ ca pe o constantă de normalizare, obținând:

$$P(X|E) = \alpha P(X|E_X^+)P(E_X^-|X) \quad (3)$$

Prin urmare, este necesar să calculăm cei doi termeni $P(X|E_X^+)$ și $P(E_X^-|X)$. Vom începe prin tratarea primului, care se calculează relativ ușor.

Vom calcula $P(X|E_X^+)$ luând în considerație toate configurațiile posibile ale părintilor lui X , precum și cât de probabile sunt acestea fiind dată mulțimea E_X^+ . În cazul fiecărei configurații date, probabilitatea lui X se cunoaște direct din tabelul probabilităților condiționate.

Fie \mathbf{U} vectorul părinților U_1, \dots, U_m și fie \mathbf{u} o atribuire de valori pentru aceștia²¹. În calculele care urmează vom folosi faptul că $E_{U_i \setminus X}$ d-separă pe U_i de toate celelalte dovezi din E_X^+ . Înțând cont de acest fapt și de formula (1), obținem egalitatea:

$$P(\{U_i = u_i\} | E_X^+) = P(\{U_i = u_i\} | E_{U_i \setminus X}) \quad i = \overline{1, m} \quad (4)$$

Luând în considerație evenimentul sigur, putem exprima probabilitatea $P(X | E_X^+)$ sub următoarea formă:

$$P(X | E_X^+) = P\left(X \cap \bigcup_{(u_1, \dots, u_m)} \{U_1 = u_1, \dots, U_m = u_m\} | E_X^+\right) \quad (5)$$

Întrucât membrul drept din (5) reprezintă probabilitatea unei reuniuni de mulțimi disjuncte, avem:

$$P(X | E_X^+) = \sum_{(u_1, \dots, u_m)} P\left(X \cap \{U_1 = u_1, \dots, U_m = u_m\} | E_X^+\right) \quad (6)$$

Aplicând, în continuare, versiunea condiționată a formulei de înmulțire a probabilităților membrului drept din (6), obținem:

$$\begin{aligned} P(X | E_X^+) &= \sum_{(u_1, \dots, u_m)} P(\{U_1 = u_1, \dots, U_m = u_m\} | E_X^+) \cdot \\ &\cdot P(X | \{U_1 = u_1, \dots, U_m = u_m\}, E_X^+) \end{aligned} \quad (7)$$

Probabilitatea $P(\{U_1 = u_1, \dots, U_m = u_m\} | E_X^+)$, care intervene în membrul drept al formulei (7), poate fi explicitată dacă se ține cont de independența variabilelor aleatoare U_1, \dots, U_m , precum și de faptul că

²¹ Spre exemplu, dacă există doi părinți Booleeni, U_1 și U_2 , atunci \mathbf{u} trece peste patru atribuiri posibile, dintre care una este [true, false].

$E_{U_i \setminus X}$ d-separă pe U_i de toate celelalte dovezi din E_X^+ , care a fost partiționat în $E_{U_1 \setminus X}, \dots, E_{U_m \setminus X}$. Întrucât U_1, \dots, U_m sunt independente,

$$P(\{U_1 = u_1, \dots, U_m = u_m\} | E_X^+) = \prod_{i=1}^m P(\{U_i = u_i\} | E_X^+) \quad (8)$$

și, ținând cont de (1), avem:

$$\prod_{i=1}^m P(\{U_i = u_i\} | E_X^+) = \prod_{i=1}^m P(\{U_i = u_i\} | E_{U_i \setminus X}) \quad (9)$$

Cea de-a doua probabilitate condiționată care intervine în membrul drept al formulei (7) poate fi explicitată ținându-se cont de faptul că U d-separă pe X de E_X^+ :

$$P(X | \{U_1 = u_1, \dots, U_m = u_m\}, E_X^+) = P(X | \{U_1 = u_1, \dots, U_m = u_m\}) \quad (10)$$

Ținând cont de (7), (8), (9) și (10), obținem:

$$P(X | E_X^+) = \sum_{(u_1, \dots, u_m)} P(X | \{U_1 = u_1, \dots, U_m = u_m\}) \prod_{i=1}^m P(\{U_i = u_i\} | E_{U_i \setminus X}) \quad (11)$$

Introducând expresia lui $P(X | E_X^+)$ dată de (11) în formula (3), rezultă:

$$P(X | E) = \alpha P(E_X^- | X) \sum_{\mathbf{u}} P(X | \mathbf{u}) \prod_{i=1}^m P(\{U_i = u_i\} | E_{U_i \setminus X}) \quad (12)$$

Ecuația (12) sugerează deja un algoritm. Astfel, $P(X | \mathbf{u})$ este repartitia lui X condiționată de realizarea $(U_1 = u_1, \dots, U_m = u_m)$. Valoarea acestei probabilități poate fi luată din tabelul de probabilități condiționate asociat lui X . Calculul fiecărei probabilități $P(\{U_i = u_i\} | E_{U_i \setminus X})$ reprezintă o secvență recursivă a problemei inițiale, aceea de a calcula $P(X | E)$, adică $P(X | E_{X \setminus \cdot})$. Vom mai nota aici faptul că mulțimea variabilelor

dovezi care intervin în apelarea recursivă reprezintă o submulțime a celor din apelarea inițială, ceea ce constituie un indiciu că procedura de calcul se va termina într-un număr finit de pași, ea reprezentând într-adevăr un algoritm.

În continuare ne propunem să calculăm probabilitatea $P(E_X^- | X)$, care intervine în formula (12), urmărind, în egală măsură, obținerea unei soluții recursive.

Întrucât $E_{Y_i \setminus X}$ d-separă pe Y_i de toate celelalte dovezi din E_X^- , rezultă independența dovezilor din fiecare "casetă" Y_i față de celelalte, condiționat de X . Forma lui E_X^- este cea dată de formula (2). Datorită independenței dovezilor de sub X avem:

$$P(E_X^- | X) = \prod_{i=1}^n P(E_{Y_i \setminus X} | X) \quad (13)$$

Fie \mathbf{Y} vectorul fiilor Y_1, \dots, Y_n și fie $\mathbf{y} = (y_1, \dots, y_n)$ o realizare a acestuia. În cele ce urmează, vom ține cont de valorile fiilor lui $X, Y_i (i = \overline{1, n})$, dar va trebui să includem și părinții fiecărui Y_i . Fie \mathbf{Z}_i părinții lui Y_i , alții decât X și fie \mathbf{z}_i o atribuire de valori ale părinților. Luând în considerație evenimentul sigur $\bigcup_{y_i} \{Y_i = y_i\}$ și respectiv $\bigcup_{\mathbf{z}_i} \{\mathbf{Z}_i = \mathbf{z}_i\}$, avem:

$$\begin{aligned} P(E_{Y_i \setminus X} | X) &= P\left(E_{Y_i \setminus X} \cap \bigcup_{y_i} \{Y_i = y_i\} \cap \bigcup_{\mathbf{z}_i} \{\mathbf{Z}_i = \mathbf{z}_i\} | X\right) = \\ &= \sum_{y_i} \sum_{\mathbf{z}_i} P\left(E_{Y_i \setminus X}, \{Y_i = y_i\}, \{\mathbf{Z}_i = \mathbf{z}_i\} | X\right) \end{aligned}$$

Aplicând, în formula anterioară, versiunea condiționată a formulei de înmulțire a probabilităților, obținem:

$$P(E_{Y_i \setminus X} | X) = \sum_{y_i} \sum_{\mathbf{z}_i} P(\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\} | X) P(E_{Y_i \setminus X} | X, \{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}) \quad (14)$$

Mulțimea dovezilor $E_{Y_i \setminus X}$ reprezintă reuniunea a două submulțimi disjuncte și independente de variabile aleatoare:

$$E_{Y_i \setminus X} = E_{Y_i \setminus X}^+ \bigcup E_{Y_i}^-$$

Prin urmare, avem:

$$\begin{aligned} P(E_{Y_i \setminus X} | X, \{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}) &= P(E_{Y_i \setminus X}^+ | X, \{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}) \cdot \\ &\cdot P(E_{Y_i}^- | X, \{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}) \end{aligned} \quad (15)$$

Dar $E_{Y_i}^-$ este o mulțime de variabile aleatoare independente de X și \mathbf{Z}_i , iar $E_{Y_i \setminus X}^+$ constituie o mulțime de variabile aleatoare independente de X și Y_i . În aceste condiții, repartițiile condiționate din (15) devin:

$$P(E_{Y_i \setminus X} | X, \{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}) = P(E_{Y_i}^- | \{Y_i = y_i\}) \cdot P(E_{Y_i \setminus X}^+ | \{\mathbf{Z}_i = \mathbf{z}_i\}) \quad (16)$$

Cea de-a doua repartitie condiționată din membrul drept al lui (16) apare ca o repartitie a posteriori și poate fi exprimată cu formula lui Bayes (în versiune necondiționată):

$$P(E_{Y_i \setminus X}^+ | \{\mathbf{Z}_i = \mathbf{z}_i\}) = \frac{P(E_{Y_i \setminus X}^+) \cdot P(\{\mathbf{Z}_i = \mathbf{z}_i\} | E_{Y_i \setminus X}^+)}{P(\{\mathbf{Z}_i = \mathbf{z}_i\})} \quad (17)$$

Introducând (17) în (16) obținem:

$$P(E_{Y_i \setminus X} | X, \{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\}) =$$

$$= P(E_{Y_i}^- | \{Y_i = y_i\}) \cdot \frac{P(E_{Y_i \setminus X}^+) P(\{\mathbf{Z}_i = \mathbf{z}_i\} | E_{Y_i \setminus X}^+)}{P(\{\mathbf{Z}_i = \mathbf{z}_i\})} \quad (18)$$

Introducând acum expresia lui $P(E_{Y_i \setminus X}^+ | X, \{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\})$ dată de (18) în formula (14), rezultă:

$$\begin{aligned} P(E_{Y_i \setminus X}^- | X) &= \sum_{y_i} \sum_{\mathbf{z}_i} P(\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\} | X) \cdot P(E_{Y_i}^- | \{Y_i = y_i\}) \cdot \\ &\quad \cdot \frac{P(E_{Y_i \setminus X}^+) P(\{\mathbf{Z}_i = \mathbf{z}_i\} | E_{Y_i \setminus X}^+)}{P(\{\mathbf{Z}_i = \mathbf{z}_i\})} \end{aligned} \quad (19)$$

Revenind acum la repartiția lui E_X^- condiționată de X , dată de formula (13) și luând în considerație (19), obținem:

$$\begin{aligned} P(E_X^- | X) &= \prod_{i=1}^n \sum_{y_i} P(E_{Y_i}^- | \{Y_i = y_i\}) \sum_{\mathbf{z}_i} P(\{Y_i = y_i, \mathbf{Z}_i = \mathbf{z}_i\} | X) \cdot \\ &\quad \cdot \frac{P(E_{Y_i \setminus X}^+) P(\{\mathbf{Z}_i = \mathbf{z}_i\} | E_{Y_i \setminus X}^+)}{P(\{\mathbf{Z}_i = \mathbf{z}_i\})} \end{aligned} \quad (20)$$

Aplicând în (20) versiunea condiționată a formulei de înmulțire a probabilităților, rezultă:

$$\begin{aligned} P(E_X^- | X) &= \prod_{i=1}^n \sum_{y_i} P(E_{Y_i}^- | \{Y_i = y_i\}) \cdot \sum_{\mathbf{z}_i} \frac{P(E_{Y_i \setminus X}^+) P(\{\mathbf{Z}_i = \mathbf{z}_i\} | E_{Y_i \setminus X}^+)}{P(\{\mathbf{Z}_i = \mathbf{z}_i\})} \cdot \\ &\quad \cdot P(\{\mathbf{Z}_i = \mathbf{z}_i\} | X) \cdot P(\{Y_i = y_i\} | X, \{\mathbf{Z}_i = \mathbf{z}_i\}) \end{aligned} \quad (21)$$

Dar \mathbf{Z} și X sunt d-separate, ceea ce înseamnă că $P(\{\mathbf{Z}_i = \mathbf{z}_i\} | X) = P(\{\mathbf{Z}_i = \mathbf{z}_i\})$. Putem, de asemenea, înlocui pe $P(E_{Y_i \setminus X}^+)$ cu o constantă de normalizare β_i . În aceste condiții, egalitatea (21) devine:

$$P(E_X^- | X) = \prod_{i=1}^n \sum_{y_i} P(E_{Y_i}^- | \{Y_i = y_i\}) \sum_{\mathbf{z}_i} \beta_i P(\{\mathbf{Z}_i = \mathbf{z}_i\} | E_{Y_i \setminus X}^+) \cdot \\ \cdot P(\{Y_i = y_i\} | X, \{\mathbf{Z}_i = \mathbf{z}_i\})$$

Combinând toți β_i într-o unică constantă de normalizare β și ținând cont de semnificația lui $E_{Y_i \setminus X}^+$, obținem:

$$P(E_X^- | X) = \beta \prod_i \sum_{y_i} P(E_{Y_i}^- | y_i) \sum_{\mathbf{z}_i} P(y_i | X, \mathbf{z}_i) \prod_j P(z_{ij} | E_{Z_j \setminus Y_i}) \quad (22)$$

Se observă că fiecare dintre termenii expresiei finale date de (22) este ușor de evaluat:

- $P(E_{Y_i}^- | y_i)$ este o secvență recursivă a calculării lui $P(E_X^- | X)$;
- $P(y_i | X, \mathbf{z}_i)$ se citesc din tabelul de probabilități condiționate asociat lui Y_i ;
- $P(z_{ij} | E_{Z_j \setminus Y_i})$ reprezintă o secvență recursivă a problemei inițiale, adică a calculării lui $P(X | E)$, mai precis a lui $P(X | E_{X \setminus V})$.

Problema transformării acestor calcule într-un *algoritm* este relativ simplă. Așa cum se arată în [9], vor fi necesare două subrutine de bază, care vor calcula $P(X | E_{X \setminus V})$ și respectiv $P(E_{X \setminus V}^- | X)$, unde prin V am notat variabila care desemnează dovezile exceptate atunci când sunt luate în considerație cele conectate la nodul X .

În Algoritmul 5.1 subroutinesa SUPPORT-EXCEPT(X, V) calculează $P(X | E_{X \setminus V})$ folosind o generalizare a ecuației (12), în timp ce subroutinesa DOVEZI-EXCEPT(X, V) calculează $P(E_{X \setminus V}^- | X)$ folosind o generalizare a ecuației (22).

Pentru a simplifica prezentarea, se va presupune că rețeaua este fixată și deja ”aprovisionată” cu dovezi, precum și că variabilele dovezi satisfac predicatul DOVEZI?. Probabilitățile $P(X|\mathbf{U})$, unde \mathbf{U} reprezintă vectorul părinților lui X , sunt disponibile în tabelul de probabilități condiționate asociat lui X . Calculul expresiilor $\alpha \dots$ și $\beta \dots$ se face prin normalizare.

În general, un agent primește valori ale variabilelor dovezi prin intermediul senzorilor săi sau în urma unor raționamente și se interesează (întreabă) despre posibilele valori ale altor variabile, astfel încât să poată decide ce acțiune trebuie întreprinsă în continuare. Algoritmul 5.1, preluat de noi din [9], calculează distribuția de probabilitate condiționată pentru o variabilă de interogare dată. El reprezintă un algoritm de tip înlănțuire-înapoi pentru rezolvarea interogărilor probabiliste asupra unui polyarbore.

Algoritmul 5.1

```

function INTEROGARE-REȚEA( $X$ ) return
    o distribuție de probabilitate a valorilor lui  $X$ 
    input:  $X$ , o variabilă aleatoare
    SUPPORT-EXCEPT( $X$ , nul)

function SUPPORT-EXCEPT( $X, V$ ) return  $P(X|E_{X\setminus V})$ 
    if DOVEZI ? ( $X$ ) then return distribuția observată pentru  $X$ 
    else
        calculează  $P(E_{X\setminus V}^- | X) = \text{DOVEZI-EXCEPT } (X, V)$ 

```

```

U  $\leftarrow$  PĂRINTI [X]
if U este vid
    then return  $\alpha P(E_{X \setminus V}^- | X)P(X)$ 
else
    pentru fiecare  $U_i$  in U do
        calculează și memorează  $P(U_i | E_{U_i \setminus X}) =$ 
        =SUPPORT-EXCEPT ( $U_i, X$ )
    return  $\alpha P(E_{X \setminus V}^- | X) \sum_u P(X | u) \prod_i P(U_i | E_{U_i \setminus X})$ 

function DOVEZI-EXCEPT (X,V) return  $P(E_{X \setminus V}^- | X)$ 
    Y  $\leftarrow$  FII [X] - V
    if Y este vid
        then return o repartiție uniformă
    else
        pentru fiecare  $Y_i$  in Y do
            calculează  $P(E_{Y_i}^- | y_i) =$ DOVEZI-EXCEPT ( $Y_i$ , nul)
             $Z_i \leftarrow$  PĂRINTI [ $Y_i$ ] - X
            pentru fiecare  $Z_{ij}$  in  $Z_i$ 
                calculează  $P(Z_{ij} | E_{Z_{ij} \setminus Y_i}) =$ 
                =SUPPORT-EXCEPT ( $Z_{ij}, Y_i$ )
    return  $\beta \prod_i \sum_{y_i} P(E_{Y_i}^- | y_i) \sum_{\mathbf{z}_i} P(y_i | X, \mathbf{z}_i) \prod_j P(z_{ij} | E_{Z_{ij} \setminus Y_i})$ 

```

□

Calculele efectuate de Algoritmul 5.1 presupun apeluri recursive care se extind de la X de-a lungul tuturor drumurilor din rețea. Recursivitatea se încheie atunci când sunt atinse noduri-dovezi, noduri-rădăcină (care nu au părinti) și noduri-frunză (care nu au fi). Fiecare apel recursiv exclude nodul de la care s-a produs apelul, astfel încât fiecare nod din arbore este tratat o singură dată. Prin urmare, algoritmul este liniar raportat la numărul de noduri ale rețelei. Algoritmul se comportă în acest fel deoarece rețeaua dată reprezintă un *polyarbore*. Dacă ar exista mai mult de un singur drum între o pereche de noduri, atunci fie procedurile recursive ar lua în calcul aceleași dovezi de mai multe ori, fie execuția nu s-ar încheia într-un număr finit de pași.

Algoritmul prezentat, de tip înlănuire-înapoi, este cel mai simplu algoritm pentru polyarbore. Principalul neajuns al unui asemenea algoritm este acela că el calculează repartiția de probabilitate pentru o unică variabilă. Dacă se dorește determinarea repartițiilor a posteriori pentru toate variabilele care nu constituie variabile dovezi, atunci programul implementând Algoritmul 5.1 ar trebui executat pentru fiecare dintre acestea, ceea ce ar mări timpul de execuție. În acest caz este preferabilă o abordare de tip înlănuire-înainte, care pleacă de la variabilele dovezi. În cazul unei contabilizări adecvate, calculele pot fi efectuate într-un timp liniar. Versiunea algoritmului care folosește înlănuirea-înainte poate fi privită ca reprezentând o ”propagare a mesajelor” prin rețea. Din această cauză implementarea pe calculatoare paralele este relativ simplă și pot fi făcute analogii interesante cu propagarea mesajelor între neuronii din creier.

EMI-UNIBUC

ANEXA 1. Notații și formule matematice

1. Repartiții de probabilitate multidimensionale

Fie $E = (X_1, \dots, X_n)$ un vector aleator format din n variabile discrete. Repartiția sa, $P_0(X_1, \dots, X_n)^{-1}$, este complet specificată de valorile

$$\begin{aligned} p(x_1, \dots, x_n) &= P(X_1 = x_1, \dots, X_n = x_n) \geq 0 \\ \sum_{(x_1, \dots, x_n)} p(x_1, \dots, x_n) &= 1, \end{aligned}$$

unde (x_1, \dots, x_n) este o "realizare" a lui E .

În general, pentru a desemna $P_0(X_1, \dots, X_n)^{-1}$, se folosește notația simplificată $P(E)$.

În acest context, are loc următoarea formulă de înmulțire a probabilităților:

$$\begin{aligned} P(X_1 = x_1, \dots, X_n = x_n) &= P(X_1 = x_1) \cdot P(X_2 = x_2 | X_1 = x_1) \cdot \dots \\ &\quad \cdot P(X_n = x_n | X_1 = x_1, \dots, X_{n-1} = x_{n-1}) \end{aligned}$$

2. Repartiții condiționate

Fie Y o variabilă aleatoare discretă și $E = (X_1, \dots, X_n)$ un vector aleator cu componente discrete. Corpul de evenimente generate de E este finit, având ca generatori pe $\{X_1 = x_1, \dots, X_n = x_n\}$.

Pentru fiecare realizare (x_1, \dots, x_n) a lui E , notăm probabilitatea condiționată a unui eveniment A cu

$$P(A | (x_1, \dots, x_n)) = \frac{P(A \cap \{X_1 = x_1, \dots, X_n = x_n\})}{P(X_1 = x_1, \dots, X_n = x_n)}$$

Corespunzător variabilei Y putem lua în considerație:

- repartitia lui Y , $P(Y)$, dată de valorile

$$p(y) = P(Y = y);$$

- repartitia lui Y condiționată de E , $P(Y|E)$, care este dată, pentru fiecare realizare (x_1, \dots, x_n) , de valorile

$$p(y | (x_1, \dots, x_n)) = P(\{Y = y\} | (x_1, \dots, x_n)).$$

În acest context se verifică versiunea condiționată a formulei de înmulțire a probabilităților:

$$\begin{aligned} P(X_1 = x_1, \dots, X_n = x_n | C) &= P(X_1 = x_1 | C) \cdot P(X_2 = x_2 | C, X_1 = x_1) \cdot \dots \\ &\quad \cdot P(X_n = x_n | C, X_1 = x_1, \dots, X_{n-1} = x_{n-1}). \end{aligned}$$

3. Formula lui Bayes

Fie E și Y cu aceeași semnificație din §1 și §2.

- Formula lui Bayes în *versiune necondiționată* este utilizată pentru a exprima o repartiție a posteriori

$$P(E | Y = y) = \frac{P(E) \cdot P(\{Y = y\} | E)}{P(\{Y = y\})}$$

unde:

$P(E)$ este repartiția a priori a lui E ;

$\{Y = y\}$ este evenimentul observat ;

$P(E | Y = y)$ este repartiția a posteriori a lui E ;

$P(\{Y = y\} | E)$ este complet determinată de valorile

$$P(\{Y = y\} | \{X_1 = x_1, \dots, X_n = x_n\}).$$

- Formula lui Bayes în *versiune condiționată* este utilizată pentru a exprima o repartiție a posteriori condiționată

$$P(Y | E, e) = \frac{P(Y | E) \cdot P(e | E, Y)}{P(e | E)}$$

unde:

$P(Y | E)$ este o probabilitate condiționată, a priori ;

e reprezintă o nouă dovedă ;

$P(Y | E, e)$ este o probabilitate condiționată, a posteriori.

FMI-UNIBUC

BIBLIOGRAFIE

1. BRATKO, I., *Prolog Programming for Artificial Intelligence*. Second Edition. Workingham, Addison - Wesley, 1990.
2. CLOCKSin, W. F., MELLISH, C. S., *Programming in Prolog*. Springer-Verlag, 1994.
3. DURKIN, J., *Expert Systems Design and Development*. Prentice Hall, 1994.
4. FLOREA, A. M., *Elemente de inteligență artificială. Vol. I, Principii și modele*. Lito. UPB, Bucuresti, 1993.
5. LUGER, G. F., STUBBLEFIELD, W. A., *Artificial Intelligence. Structures and Strategies for Complex Problem Solving*. Third Edition. Addison – Wesley, 1998.
6. MERRITT, D., *Building Expert Systems in Prolog*. Springer-Verlag, 1989.
7. NILLSON, N., *Artificial Intelligence: A New Synthesis*. Morgan Kauffman, 1998.
8. RICH, E., KNIGHT, K., *Artificial Intelligence*. Second Edition. Tata McGraw-Hill Publishing Company Limited, New Delhi, 1993.
9. RUSSELL, S. J., NORVIG, P., *Artificial Intelligence. A Modern Approach*. Prentice - Hall International, Inc., 1995.
10. *SICSTUS PROLOG USER'S MANUAL*. Intelligent Systems Laboratory, Sweedish Institute of Computer Science;
<http://www.sics.se/isl/sicstuswww/site/documentation.html>

11. MICROSOFT PRESS. *DICȚIONAR DE CALCULATOARE.*
(Traducere de Nicolae Pora). Editura Teora, Bucuresti, 1999.

FMI-UNIBUC

CUPRINS

CUVÂNT ÎNAINTE	3
1. CONSIDERAȚII PRELIMINARE	5
1.1. Domeniul inteligenței artificiale. Definiții și un scurt istoric	6
1.2. Subdomenii ale inteligenței artificiale	12
1.3. Tehnici ale inteligenței artificiale	14
1.4. Un exemplu de problemă tipică în inteligența artificială	17
2. TEHNICI DE CĂUTARE	31
2.1. Rezolvarea problemelor prin intermediul căutării	32
2.1.1. Tipuri de probleme	34
2.1.2. Probleme și soluții corect definite	36
2.1.3. Căutarea soluțiilor și generarea secvențelor de acțiuni	41
2.1.4. Structuri de date pentru arbori de căutare	42
2.1.5. Evaluarea strategiilor de căutare	45
2.2. Căutarea neinformată	45
2.2.1. Căutarea de tip breadth-first	46
2.2.1.1. Algoritmul de căutare breadth-first	47
2.2.1.2. Implementare în Prolog	48
2.2.1.3. Timpul și memoria cerute de strategia breadth-first	54

2.2.1.4. Un exemplu. Problema misionarilor și canibalilor	55
2.2.2. Căutarea de tip depth-first	61
2.2.2.1. Prezentare generală și comparație cu strategia breadth-first	61
2.2.2.2. Implementare în Prolog	63
2.2.3. Căutarea în adâncime iterativă	71
2.2.3.1. Prezentare generală	71
2.2.3.2. Implementare în Prolog	73
2.2.4. Strategii de căutare neinformată. Un exemplu	75
2.3. Căutarea informată	85
2.3.1. Căutarea de tip best-first	85
2.3.2. Algoritm de căutare general bazat pe grafuri	88
2.3.2.1. Algoritmul A*	89
2.3.2.1.1. Admisibilitatea Algoritmului A*	94
2.3.2.1.2. Condiția de consistență	101
2.3.2.1.3. Optimalitatea Algoritmului A*	105
2.3.2.1.4. Completitudinea Algoritmului A*	106
2.3.2.1.5. Complexitatea Algoritmului A*	107
2.3.2.2. Iterative Deepening A* (IDA*)	108
2.3.3. Implementarea în Prolog a căutării de tip best-first	109
2.3.3.1. Exemple	118
2.3.4. Concluzii	131
3. JOCURILE CA PROBLEME DE CĂUTARE	135
3.1. O definire formală a jocurilor	137

3.2. Algoritmul Minimax	138
3.2.1. Funcții de evaluare	143
3.2.2. O implementare în Prolog	144
3.3. O implementare eficientă a principiului Minimax: Algoritmul Alpha-Beta	146
3.3.1. Implementare în Prolog	152
3.3.2. Considerații privitoare la eficiență	154
3.4. Un exemplu. Jocul X și 0	156
4. ASPECTE ALE REPREZENTĂRII CUNOȘTINȚELOR 173	
4.1. Tipuri de cunoștințe	174
4.1.1. Cunoștințe relaționale simple	174
4.1.2. Cunoștințe care se moștenesc	175
4.1.3. Cunoștințe inferențiale	178
4.1.4. Cunoștințe procedurale	179
4.2. Clase de metode pentru reprezentarea cunoștințelor	180
4.3. Reprezentarea cunoștințelor și sistemele expert	181
4.3.1. Structura de bază a unui sistem expert	183
4.3.2. Reprezentarea cunoștințelor cu reguli if-then	184
4.3.3. Înlățuire înainte și înapoi în sistemele bazate pe reguli	187
4.3.3.1. Înlățuirea înapoi	187
4.3.3.2. Înlățuirea înainte	191
4.3.3.3. Concluzii	193
4.3.4. Generarea explicațiilor	194
4.3.5. Introducerea incertitudinii	196

4.3.6. Un exemplu de sistem expert. Implementare în Prolog	201
4.4. Rețele semantice și cadre	242
4.4.1. Rețele semantice	243
4.4.2. Cadre	251
4.5. Considerații finale	256
5. RAȚIONAMENT STATISTIC	259
5.1. Abordări statistice pentru sisteme de raționament incert	259
5.1.1. Factori de certitudine și sisteme bazate pe reguli	262
5.1.2. Rețele Bayesiene	269
5.1.2.1. Relații de independență condiționată în rețele Bayesiene	273
5.1.2.2. Inferență în rețele Bayesiene	275
5.1.2.2.1. Un algoritm pentru răspunsul la interogări	276
ANEXA1. NOTAȚII ȘI FORMULE MATEMATICE	291
BIBLIOGRAFIE	295