

Universitatea “Babeş-Bolyai”, Cluj-Napoca
Facultatea de Matematică şi Informatică

INTELIGENȚĂ ARTIFICIALĂ

Horia F. POP

Gabriela ȘERBAN

Cluj-Napoca, 2004

Prefață

Cartea de față este destinată studenților secțiilor de Informatică și matematică-informatică ale Facultății de Matematică și Informatică, ca și tuturor celor interesați în studierea problematicei inteligenței artificiale.

Lucrarea, având ca sursă fundamentală [13], este compusă din opt capitole care abordează unele dintre cele mai importante elemente ale Inteligenței Artificiale. Pe lângă abordarea noțiunilor teoretice și descrierea multor exemple relevante, fiecare capitol conține câte o secțiune de aplicații rezolvate și probleme propuse, utile în efortul de înțelegere a materialului teoretic.

Primul capitol, **“Ce este Inteligența Artificială”**, reprezintă o definiție și o introducere în Inteligența Artificială, precum și o scurtă trecere în revistă a problematicei IA.

Capitolul al doilea, **“Probleme, spații de probleme și căutare”**, definește problema de Inteligență Artificială ca o căutare într-un spațiu de stări. Este definit sistemul de producție și sunt trecute în revistă caracteristicile problemelor și sistemelor de producție. Sunt discutate câteva tehnici de bază de rezolvare a problemelor.

Capitolul al treilea, **“Tehnici de căutare euristică”**, introduce noțiunea de tehnică euristică ca alternativă la tehnicile generale. Sunt prezentate și exemplificate mai multe tehnici euristice de rezolvare a problemelor.

Capitolul al patrulea, **“Introducere în Reprezentarea cunoștințelor”**, face o trecere în revistă a modurilor de reprezentare a cunoștințelor. Sunt trecute în revistă proprietățile acestora și diferite aspecte care intervin în reprezentarea cunoștințelor.

Capitolul al cincilea, **“Reprezentarea cunoștințelor folosind reguli”**, reprezintă o trecere în revistă a reprezentării declarative a cunoștințelor în sistemele bazate pe reguli și a modului de raționare cu aceste cunoștințe. Se face o trecere de la mecansimul simplu al programării logice la modele de raționare mai complicate.

Capitolul al șaselea, **“Jocuri”**, face o trecere în revistă a problemelor de căutare bazate pe jocuri cu doi adversari. Este prezentată procedura de căutare Minimax și sunt analizate o serie de îmbunătățiri ale acesteia.

Capitolul al șaptelea, **“Planificare”**, descrie câteva tehnici pentru combinarea strategiilor de bază de rezolvare a problemelor cu mecanisme de reprezentare a cunoștințelor, cu scopul de a construi planuri care să permită rezolvarea problemelor dificile. Domeniul ales pentru exemplificarea tehnicilor este lumea blocurilor.

Capitolul al optulea, **“Agenți Inteligenți”**, este destinat prezentării domeniului agenților

intelenți, ca fiind o importantă arie de cercetare și dezvoltare în domeniul Informaticii, în special în domeniul inteligenței artificiale. Sunt prezentate cele mai importante arhitecturi abstracte și concrete de agenți inteligenți, precum și o nouă paradigmă de programare, “programarea orientată pe agenți” (AOP).

Autorii

Cuprins

Prefață	i
1. Ce este Inteligența Artificială?	1
1.1. Problemele de IA	1
1.2. Ipotezele de bază	3
1.3. Ce este o tehnică de AI?	3
1.3.1. Exemplu: Tic-Tac-Toe	4
1.3.2. Concluzii	7
1.4. Nivelul modelului	8
1.5. Criterii de succes	9
1.6. Concluzii	9
2. Probleme, spații de probleme și căutare	11
2.1. Definirea problemei sub formă de căutare într-un spațiu de stări	11
2.2. Sisteme de producție	14
2.2.1. Strategii de control	15
2.2.2. Căutare euristică	18
2.3. Caracteristicile problemei	21
2.3.1. Problema se poate descompune?	21
2.3.2. Pașii pe drumul către soluție se pot ignora sau reface?	23
2.3.3. Este universul previzibil?	24
2.3.4. O soluție bună este absolută sau relativă?	25
2.3.5. Este soluția o stare sau un drum?	26
2.3.6. Care este rolul cunoștințelor?	27
2.3.7. Soluția cere interacțiunea cu o persoană?	27
2.3.8. Clasificarea problemelor	28
2.4. Caracteristicile sistemelor de producție	28
2.5. Aspecte în proiectarea programelor de căutare	30
2.6. Alte probleme	32
2.7. Aplicații rezolvate și probleme propuse	33

3. Tehnici de căutare euristică	45
3.1. Generează și testează	45
3.2. Hill Climbing	47
3.2.1. Hill Climbing simplu	47
3.2.2. Hill Climbing de pantă maximă (steepest-ascent)	48
3.2.3. Călire simulată (Simulated Annealing)	51
3.3. Căutare Best-First	54
3.3.1. Grafuri OR	54
3.3.2. Algoritmul A*	56
3.3.3. Agende	60
3.4. Reducerea problemei	62
3.4.1. Grafuri AND-OR	62
3.4.2. Algoritmul AO*	64
3.5. Satisfacerea Restricțiilor (Constraints Satisfaction)	66
3.6. Analiza Means-Ends	70
3.7. Aplicații rezolvate și probleme propuse	73
4. Introducere în Reprezentarea cunoștințelor	89
4.1. Moduri de reprezentare	89
4.2. Proprietăți ale reprezentării cunoștințelor	91
4.3. Aspecte ale reprezentării cunoștințelor	95
4.3.1. Atribute importante	96
4.3.2. Relații dintre atribute	96
4.3.3. Alegerea granularității reprezentării	98
4.3.4. Reprezentarea mulțimilor de obiecte	100
4.3.5. Identificarea celor mai potrivite structuri	101
4.4. Problema cadrelor	103
5. Reprezentarea cunoștințelor folosind reguli	105
5.1. Cunoștințe procedurale și declarative	105
5.2. Programare logică	106
5.3. Raționare înainte și înapoi	108
5.3.1. Sisteme de reguli cu înlănțuire înapoi	112
5.3.2. Sisteme de reguli cu înlănțuire înainte	112
5.3.3. Combinarea raționamentului înainte și înapoi	112
5.4. Potrivire	113
5.4.1. Indexarea	113
5.4.2. Potrivirea cu variabile	114
5.4.3. Potrivirea complexă și aproximativă	114

5.4.4. Rezolvarea conflictelor	115
5.5. Controlul cunoștințelor	116
5.6. Aplicații rezolvate și probleme propuse	119
6. Jocuri	123
6.1. Introducere	123
6.2. Procedura de căutare Minimax	125
6.3. Adăugarea tăieturilor alfa-beta	128
6.4. Rafinări suplimentare	131
6.4.1. Așteptarea unei perioade de liniște	131
6.4.2. Căutare secundară	132
6.4.3. Utilizarea unei arhive de mutări	133
6.4.4. Alternative la Minimax	133
6.5. Adâncire iterativă	134
6.6. Aplicații rezolvate și probleme propuse	135
7. Planificare	153
7.1. Introducere	153
7.2. Domeniu exemplu: lumea blocurilor	155
7.3. Componentele unui sistem de planificare	156
7.4. Planificare folosind stive de obiective	161
7.5. Planificare neliniară folosind declararea limitărilor	169
7.6. Planificare ierarhică	175
7.7. Sisteme reactive	176
7.8. Alte tehnici de planificare	177
7.9. Probleme propuse	177
8. Agenți inteligenți	179
8.1. Conceptul de agent inteligent	179
8.2. Structura agenților inteligenți	179
8.3. Tipuri de agenți inteligenți	181
8.4. Arhitecturi abstracte pentru agenți inteligenți	183
8.4.1. Agenți total reactivi	184
8.4.2. Percepție	185
8.4.3. Agenți cu stări	186
8.5. Arhitecturi concrete pentru agenți inteligenți	187
8.5.1. Arhitecturi bazate pe logică	188
8.5.2. Arhitecturi reactive	192
8.5.3. Arhitecturi tip Opinie-Cerere-Intenție	194
8.5.4. Arhitecturi stratificate	198

8.6. Limbaje de programare pentru agenți	199
8.6.1. Programare orientată pe agenți	199
8.6.2. “METATEM” Concurent	201
8.7. Agenți inteligenți și Obiecte	203
Bibliografie	205

Capitolul 1

Ce este Inteligența Artificială?

Ce este IA? Până în prezent s-au făcut multe încercări de definire, majoritatea inutile. Noi vom avea în vedere următoarea definiție: IA este studiul modului în care putem determina computerele să facă lucruri la care, în prezent, omul este mai bun.

Să observăm că această definiție este efemeră, din cauză că se referă la stadiul curent al informaticii. De asemenea, nu include zone cu impact foarte mare, anume acele probleme care nu pot fi rezolvate bine astăzi nici de oameni, nici de calculatoare.

1.1. Problemele de IA

Care sunt problemele conținute în IA? Multe din eforturile timpurii s-au făcut în **domenii formale**, cum ar fi **teoria jocurilor** și **demonstrarea automată a teoremelor**; exemplu: șah, checkers; încercări de demonstrare automată a teoremelor matematice (algebră, geometrie).

Teoria jocurilor și demonstrarea teoremelor partajează proprietatea că oamenii care le fac bine sunt considerați inteligenți. S-a considerat că nu vor fi probleme cu computerele în aceste zone, din cauza automatismelor implicate; au fost probleme cu explozia combinatorială.

Alte eforturi s-au făcut în domeniul rezolvării de probleme realizate zilnic – **commonsense reasoning**; exemplu: despre obiecte și relațiile dintre acestea; despre acțiuni și consecințele lor.

Pe măsura progresului cercetărilor de IA s-au abordat noi domenii: **percepție (vedere și vorbire)**, **înțelegerea limbajului natural**, **rezolvarea problemelor în domenii specializate** – diagnostic medicală și chimică.

În plus față de aceste domenii comune, mulți oameni realizează sarcini într-unul sau mai multe domenii specializate, domenii în care au învățat expertiza necesară (proiectare, descoperiri științifice, diagnostic medicală, planificare financiară).

Programe care pot rezolva probleme din aceste domenii sunt tot din domeniul IA.

Iată deci unele din scopurile cercetărilor de IA:

- **sarcini uzuale**

- percepție
 - * vedere
 - * vorbire
- limbaj natural
 - * înțelegere
 - * generare
 - * traducere
 - * commonsense reasoning
 - * controlul roboților
- **sarcini formale**
 - jocuri
 - * șah
 - * table
 - * checkers
 - * go
 - matematică
 - * geometrie
 - * logică
 - * calcul integral
 - * demonstrarea proprietăților programelor
- **sarcini experte**
 - inginerie
 - * proiectare
 - * descoperirea erorilor
 - * planificarea producției
 - analiză științifică
 - diagnoză medicală
 - planificare financiară

Înainte să studiem probleme AI specifice și tehnici de soluționare, vom discuta următoarele patru întrebări:

(1) Care sunt ipotezele de bază despre inteligență?

- (2) Ce tipuri de tehnici vor fi utile în rezolvarea problemelor de AI?
- (3) La ce nivel de detaliu încercăm să modelăm inteligența umană?
- (4) Cum vom ști dacă am reușit să construim un program inteligent?

1.2. Ipotezele de bază

La baza cercetărilor din domeniul Inteligenței Artificiale stă ipoteza **sistemului de simboluri fizice** (SSF). Sistemul de simboluri fizice (SSF) este un set de entități (**simboluri**) care sunt modele fizice care pot apărea sub formă de componente ale unui alt tip de entități (**expresii, structuri de simboluri**).

O structură de simboluri este compusă dintr-un număr de instanțe de simboluri (**tokeni**) legate într-un anumit mod fizic (de exemplu vecinătate). La orice moment de timp sistemul va conține o **colecție** de astfel de structuri, precum și **proces** ce operează pe aceste expresii pentru a produce alte expresii (**creare, modificare, reproducere, distrugere**).

Un SSF este o mașină care produce de-a lungul timpului o colecție în evoluție de sisteme de simboluri.

Ipoteza de bază a IA este **ipoteza SSF**: un SSF are mijloacele necesare și suficiente pentru a genera acțiuni inteligente. Aceasta este doar o ipoteză; nu apare nici o posibilitate de a o demonstra sau infirma cu fundamente logice; trebuie supusă validării empirice — prin experimentare. Computerele oferă mediul perfect pentru aceste experimentări — pot fi programare să simuleze orice SSF.

A devenit din ce în ce mai realizabil să se desfășoare investigații empirice asupra ipotezei SSF. În fiecare investigație de acest fel se selectează o sarcină particulară ce poate fi privită ca cerând inteligență; se va produce și testa un program ce realizează sarcina respectivă.

Importanța ipotezei SSF: **(1)** e o teorie semnificativă a naturii înțelegerii umane — de mare interes pentru psihologi; **(2)** formează baza credinței că e posibil să se construiască programe ce pot îndeplini sarcini inteligente ce acum sunt îndeplinite de oameni.

1.3. Ce este o tehnică de AI?

Problemele de AI sunt împrăștiate pe un spectru foarte larg. Aparent singura legătură dintre ele este că sunt dificile.

Există tehnici potrivite pentru soluționarea unei varietăți dintre acestea. Ce putem spune despre aceste tehnici în afară de faptul că manipulează simboluri? Cum putem spune dacă aceste tehnici sunt folositoare în rezolvarea problemelor? Ce proprietăți trebuie să aibă aceste tehnici?

Unul dintre cele mai dificile și rapide rezultate ale inteligenței artificiale este acela că inteligența cere cunoaștere. Cunoașterea are proprietăți relativ negative: este voluminoasă; este dificil de caracterizat cu precizie; este constant schimbătoare.

O tehnică de AI este o metodă care exploatează cunoașterea și verifică următoarele condiții:

- (1) **captează generalitatea** (nu trebuie memorate / reprezentate separat situații individuale);
- (2) **poate fi înțeleasă de oamenii care o oferă** (în multe domenii cunoștințele trebuie preluate de la oameni și soluțiile trebuie justificate oamenilor);
- (3) **poate fi ușor de modificat** pentru a corecta erorile și a reflecta schimbările în lume și în vederea noastră despre lume;
- (4) **poate fi folosită într-o mare varietate de situații**, chiar dacă nu e absolut exactă și/sau completă;
- (5) **poate fi folosită pentru evitarea supraîncărcării datelor**, prin sprijinirea limitării gamei de posibilități ce trebuie de obicei considerate.

Există o independență între tehnici și probleme, și anume:

- se pot rezolva probleme de AI fără tehnici de AI;
- se pot rezolva alte probleme folosind tehnici de AI.

1.3.1. Exemplu: Tic-Tac-Toe

În continuare vom analiza trei tehnici care rezolvă problema Tic-Tac-Toe și care cresc în:

- complexitate;
- utilizarea generalității;
- claritatea cunoștințelor;
- extensibilitatea abordărilor,

și deci cresc în ceea ce numim tehnici de IA.

Varianta 1

Structura datelor este următoarea:

- **tabla**: vector de 9 componente cu 0 (pătrat gol), 1 (indică X) și 2 (indică O);
- **mutările**: vector de 39 elemente, fiecăre câte un vector cu 9 elemente.

Algoritmul pentru a face o mutare:

- (1) configurația tablei o privim ca număr în baza trei; îl transformăm în baza 10;
- (2) îl privim ca index în vectorul de mutări;
- (3) configurația după mutare este vectorul indicat în vectorul de mutări.

Comentarii:

- această tehnică necesită spațiu mult;
- intrările sunt dificil de completat (este mult de lucrat);
- este improbabil ca intrările să se completeze fără erori;
- în caz de extindere la trei dimensiuni toată metoda trebuie reluat de la început (rediscutată).

Varianta 2

Structura datelor este următoarea:

- **tabla**: vector de 9 componente cu 2 (pătrat gol), 3 (indică X) și 5 (indică 0);
- **mutarea**: întreg: care mutare a jocului trebuie făcută: 1 = prima, 9 = ultima.

Algoritmul folosește trei proceduri:

- **MAKE2** încearcă să facă două pe un rând; dacă în centru nu poate, încearcă în pătratele necolțuri;
- **POSSWIN**(p) = 0 dacă jucătorul p nu poate câștiga la mutarea următoare; altfel întoarce numărul pătratului ce constituie mutare câștigătoare; poate atât câștiga, cât și să blocheze posibilitatea de câștig a adversarului; dacă produsul unei linii sau coloane = 18 ($3 \times 3 \times 2$) atunci X poate câștiga; dacă produsul = 50 ($5 \times 5 \times 2$) atunci 0 poate câștiga;
- **GO**(n) face o mutare în pătratul n ; setează $Tabla[n]=3$ dacă e mutare impară; $Tabla[n]=5$ dacă e mutare pară.

Strategie: mutările pare sunt cele ale lui 0, cele impare sunt cele ale lui X .

Mutarea 1: GO(1);

Mutarea 2: dacă $Tabla[5] = \text{blanc}$, atunci execută GO(5), altfel execută GO(1);

Mutarea 3: dacă $Tabla[9] = \text{blanc}$, atunci execută GO(9), altfel execută GO(3);

Mutarea 4: dacă $\text{POSSWIN}(X) \neq 0$, atunci execută $\text{GO}(\text{POSSWIN}(X))$, altfel execută $\text{GO}(\text{MAKE2})$;

Mutarea 5: dacă $\text{POSSWIN}(X) \neq 0$, atunci execută $\text{GO}(\text{POSSWIN}(X))$, altfel dacă $\text{POSSWIN}(0) \neq 0$ atunci execută $\text{GO}(\text{POSSWIN}(0))$, altfel, dacă $\text{Tabla}[7] = \text{blanc}$ atunci execută $\text{GO}(7)$, altfel execută $\text{GO}(3)$;

Mutarea 6: dacă $\text{POSSWIN}(0) \neq 0$ atunci execută $\text{GO}(\text{POSSWIN}(0))$, altfel dacă $\text{POSSWIN}(X) \neq 0$ atunci execută $\text{GO}(\text{POSSWIN}(X))$, altfel $\text{GO}(\text{MAKE2})$;

Mutarea 7, 8, 9: dacă $\text{POSSWIN}(\text{noi}) \neq 0$, atunci execută $\text{GO}(\text{POSSWIN}(\text{noi}))$, altfel dacă $\text{POSSWIN}(\text{ei}) \neq 0$ atunci execută $\text{GO}(\text{POSSWIN}(\text{ei}))$, altfel execută $\text{GO}(\text{oriunde e blanc})$.

Comentarii:

- nu e la fel de eficient ca timp de execuție;
- este mai eficient din punctul de vedere al spațiului ocupat;
- strategia este mai ușor de schimbat;
- totuși, strategia a fost precizată de programator;
- erorile de îndemânare ale programatorului-jucător se vor vedea în program;
- este dificil de generalizat.

Varianta 3

Structura datelor este următoarea:

- **PozițiaTablei** este o structură cu:
 - vector de 9 elemente (tabla)
 - o listă de poziții ce pot rezulta din mutarea următoare;
 - un număr, estimare a probabilității de câștig din tabla curentă.

Algoritmul este următorul:

- uită-te la pozițiile ce pot rezulta din mutările posibile;
- alege mutarea cea mai bună; (*)
- fă mutarea;

- atribuire estimată a calității celei mai bune mutări (*) de mai sus ca estimare a poziției curente;
- pentru a decide care e poziția cea mai bună:
 - dacă este câștigătoare, dă-i o valoare maximală;
 - altfel: se analizează mutările făcute de adversar la plecând de la mutarea noastră;
 - vezi care e poziția cea mai bună pentru adversar (cea mai proastă pentru noi, recursiv);
 - presupune că adversarul va face acea mutare;
 - valoarea acelei mutări este considerată ca valoare a nodului pe care îl considerăm;
 - mutarea noastră cea mai bună este cea cu cea mai mare valoare.

Comentarii:

- cere mai mult timp decât precedentele;
- poate fi extins să manevreze jocuri mai complicate;
- în loc să considerăm toate mutările posibile, se pot considera doar un subset mai probabil;
- în loc să se caute în adâncime până la câștig, se poate căuta un număr fix de pași;
- este o tehnică de Inteligență Artificială;
- este mai puțin eficientă pentru probleme foarte simple.

1.3.2. Concluzii

Al treilea program este o tehnică de AI. Este mai lent ca timp de execuție dar ilustrează câteva tehnici de AI:

căutare: o modalitate de rezolvare a problemelor pentru care nu este disponibilă nici o abordare directă și nici un cadru în care o tehnică directă poate fi scufundată;

folosirea cunoștințelor: oferă o modalitate de rezolvare a problemelor complexe exploatarea structurilor obiectelor implicate;

abstracție: oferă o modalitate de separare a caracteristicilor și variațiilor importante de cele neimportante, care altfel ar încălca programul.

Programele ce folosesc aceste tehnici au următoarele caracteristici importante:

- sunt mai puțin fragile;

- nu vor fi date peste cap de o perturbație mică la intrare;
- pot fi înțelese ușor (în ceea ce privește cunoștințele);
- funcționează pe probleme mari.

1.4. Nivelul modelului

Care este scopul nostru? Ce încercăm să facem? De ce dorim să producem programe care fac lucrurile inteligente pe care le fac oamenii? La aceste întrebări putem să răspundem în trei moduri:

- (1) dorim să producem programe care fac lucrurile în același mod în care le fac oamenii;
- (2) dorim să producem programe care pur-și-simplu fac lucruri inteligente;
- (3) dorim să producem programe care fac lucrurile în maniera care pare să fie cea mai simplă.

Programele din categoria (1) se împart în două clase:

- (a) programe care încearcă să rezolve probleme care nu sunt chiar de IA: probleme pe care computerele le pot rezolva dar pentru care folosesc mecanisme care nu sunt disponibile oamenilor;
- (b) programe care rezolvă probleme care cad clar în clasa problemelor de IA: lucruri care nu sunt triviale pentru computer.

Iată în continuare câteva motive care pot motiva necesitatea modelării gândirii umane:

- pentru testarea teoriilor psihologice despre performanța umană;
- pentru a permite computerelor să înțeleagă raționamentul uman; de exemplu un program care citește un ziar și răspunde la întrebarea ‘de ce i-au omorât teroriștii pe ostatici?’;
- pentru a permite oamenilor să înțeleagă raționamentul computerelor; în multe circumstanțe oamenii ezită să aibă încredere în rezultatele produse de computer până ce nu înțeleg modul în care mașina a produs acel rezultat;
- pentru a exploata toate cunoștințele care se pot stoarce de la oameni.

1.5. Criterii de succes

Cum știm dacă am reușit sau nu? Această întrebare este la fel de dificilă ca și întrebarea fără răspuns ‘ce este inteligența?’.

În 1950, Alan Turing a creat **testul Turing** pentru a vedea dacă o mașină se comportă inteligent. Pentru desfășurarea testului este nevoie de doi oameni și de mașina de evaluat. Un om este interogatorul, conectat la un terminal într-o cameră separată. Al doilea om este și el conectat la un terminal. Ambele terminale comunică cu mașina de testat. Interogatorul pune întrebări și primește răspunsuri prin terminal. El poate pune întrebări omului și mașinii, dar nu îi cunoaște decât ca A și B .

Scopul testului este să determine care dintre A și B este omul și care este mașina. Scopul mașinii este să-l păcălească pe interogator în așa fel încât acesta să creadă că ea este omul. Mașina are voie să facă orice pentru a reuși. Dacă reușește, putem trage concluzia că mașina poate gândi.

Problema cea mai serioasă în calea dezvoltării de mașini inteligente este cantitatea de cunoștințe necesare. Din această cauză s-ar putea ca testul Turing să nu poată fi trecut încă multă vreme. Totuși, dacă îngustăm problema și dorim cunoștințe limitate la un domeniu specific, atunci realizarea de mașini inteligente devine posibilă. Iată câteva exemple:

- programe de șah: DEEP BLUE;
- programe de analize chimice, cu rezultate de cercetare originale: DENDRAL.

Concluzie: problema dacă o mașină poate gândi este prea nebuloasă, dar se pot construi programe cu performanțe standard pentru o problemă particulară.

1.6. Concluzii

Problemele de AI sunt variate, interesante și grele. Pentru a le rezolva trebuie să realizăm două obiective importante:

- trebuie să setăm criteriile astfel încât să putem spune dacă am rezolvat problema;
- trebuie să încercăm să rezolvăm problema.

De asemenea, avem nevoie de metode care să ne ajute să rezolvăm dilema IA:

- un sistem de IA trebuie să conțină multe cunoștințe dacă trebuie să manevreze și altceva decât probleme triviale;
- pe măsură ce cantitatea de cunoștințe crește, devine din ce în ce mai dificil de accesat lucrurile pe care le dorim, deci trebuie adăugate cunoștințe în plus; dar acum sunt deja mai multe cunoștințe de manevrat, deci mai trebuie adăugate altele în plus, ș.a.m.d

Capitolul 2

Probleme, spații de probleme și căutare

Am văzut până acum o descriere sumară a tipurilor de probleme cu care se preocupă IA în mod tipic, precum și câteva tehnici pe care le oferă pentru a rezolva aceste probleme. Pentru a construi o problemă particulară avem nevoie să facem patru lucruri:

1. **Să definim problema precis.** Această definiție trebuie să includă specificații precise a ceea ce este o situație inițială și ce situații finale constituie o soluție acceptabilă pentru problemă.
2. **Să analizăm problema.** Câteva facilități foarte importante pot avea un impact imens asupra măsurii în care câteva tehnici posibile sunt corespunzătoare pentru rezolvarea problemei.
3. **Să izolăm și reprezentăm cunoștințele** necesare rezolvării problemei.
4. **Să alegem cea mai bună tehnică** de rezolvare și să o aplicăm la această problemă particulară.

În continuare vom discuta problemele 1, 2 și 4, urmând ca mai târziu să ne concentrăm pe problema 3.

2.1. Definirea problemei sub formă de căutare într-un spațiu de stări

Să luăm exemplul **jocului de șah**. Pentru a construi un program capabil să joace șah va trebui să reprezentăm poziția de pe tabla de șah corespunzătoare începerii jocului, regulile care definesc mutări legale, și pozițiile care reprezintă o victorie pentru una din părți. În plus va trebui să explicităm scopul implicit al problemei, acela de a juca nu numai un joc legal, ci de a câștiga jocul, dacă este posibil.

Este destul de ușor să oferim o descriere formală și completă a problemei. Poziția câștigătoare poate fi un vector 8×8 cu pozițiile pieselor. Poziția câștigătoare este aceea în care oponentul nu are o mutare legală și regele său este atacat. Mutările legale oferă un mod de a trece de la starea inițială la o stare finală și pot fi descrise ca un set de reguli constând din două părți: **partea stângă** care servește ca un model care se poate potrivi cu poziția curentă și o **parte dreaptă** care descrie schimbarea care trebuie făcută pe tablă pentru a reflecta mutarea. Aceste reguli pot fi scrise în câteva moduri:

(a) **Tabla înainte de mutare** \rightarrow **tabla după mutare**; trebuie indicate reguli separate pentru cele 10^{120} poziții posibile; sunt două probleme:

- nici o persoană nu poate scrie un set complet de astfel de reguli;
- nici un program nu poate manevra cu ușurință o cantitate atât de mare de reguli.

(b) Reguli descrise într-un mod cât de general posibil, cu notații pentru descrierea modelelor și substituțiilor, de exemplu:

- pion alb în pătratul (e, 2) și liber în pătratul (e, 3) și liber în pătratul (e, 4) \Rightarrow mută pionul din pătratul (e, 2) în pătratul (e, 4)

Am definit problema jocului de șah ca o problemă de deplasare printr-un **spațiu de stări** unde fiecare stare corespunde unei poziții legale pe tablă. Putem juca acum șah începând de la o stare inițială, utilizând un set de reguli pentru a ne deplasa printre stări și încercând să terminăm într-una dintr-un set de stări finale.

Această **reprezentare printr-un spațiu de stări** este naturală nu numai pentru șah, ci și pentru probleme care apar natural și care sunt mai puțin structurate ca șahul. Sigur că ar putea fi nevoie să utilizăm structuri mai complexe decât un vector.

Reprezentarea printr-un spațiu de stări formează baza celor mai multe dintre metodele de AI pe care le vom discuta. Structura sa corespunde cu structura modului de rezolvare a problemelor în două feluri importante:

- Permite o definire formală a unei probleme ca o nevoie de a converti o situație dată într-o situație dorită, prin utilizarea unui set de operații autorizate;
- Permite definirea procesului de rezolvare a unei probleme particulare ca o combinație de **tehnici cunoscute** (fiecare reprezentată ca o regulă ce definește un singur pas în spațiu) și **căutare**, tehnica generală a explorării spațiului pentru a încerca să găsim un drum de la starea curentă la o soluție.

Pentru a arăta generalitatea reprezentării printr-un spațiu de stări, o vom folosi pentru a descrie o problemă foarte diferită de șah.

Problema găleților cu apă. Se dau două găleți, de 4 litri și de 3 litri. Nici una nu are semne de măsură pe ea. Este disponibilă o pompă care poate fi folosită pentru a umple gălețile cu apă. Cum puteți obține exact 2 litri de apă în găleata de 4 litri?

Spațiul de stări este un set de perechi ordinate de întregi (x, y) , astfel încât $x = 0, 1, 2, 3, 4$ și $y = 0, 1, 2, 3$; x este numărul de litri din găleata de 4 litri iar y este numărul de litri din găleata de 3 litri. Starea inițială este $(0, 0)$, iar sparea finală este $(2, n)$, pentru orice valoare legală a lui n .

1.	$(x, y), x < 4$	\Rightarrow	$(4, y)$	umple găleata de 4 litri
2.	$(x, y), y < 3$	\Rightarrow	$(x, 3)$	umple găleata de 3 litri
3.	$(x, y), x > 0$	\Rightarrow	$(x - d, y)$	golește parte din găleata 4
4.	$(x, y), y > 0$	\Rightarrow	$(x, y - d)$	golește parte din găleata 3
5.	$(x, y), x > 0$	\Rightarrow	$(0, y)$	golește găleata 4
6.	$(x, y), y > 0$	\Rightarrow	$(x, 0)$	golește găleata 3
7.	$(x, y), x + y \geq 4, y > 0$	\Rightarrow	$(4, y - (4 - x))$	umple găleata 4 din 3
8.	$(x, y), x + y \geq 3, x > 0$	\Rightarrow	$(x - (3 - y), 3)$	umple găleata 3 din 4
9.	$(x, y), x + y < 4, y > 0$	\Rightarrow	$(x + y, 0)$	golește găleata 3 în 4
10.	$(x, y), x + y < 3, x > 0$	\Rightarrow	$(0, x + y)$	golește găleata 4 în 3

Figura 2.1: Reguli de producție pentru problema găleților cu apă

Operatorii utilizați pentru a rezolva problema sunt descriși în Figura 2.1. Operatorii sunt reprezentați tot sub forma de reguli cu două părți. În plus trebuie să explicităm unele presupuneri care nu sunt menționate în problemă. Astfel de explicitări sunt întotdeauna necesare.

Pentru a rezolva problema, în plus față de descrierea problemei mai avem nevoie de o structură de control care să cicleze printre reguli, să selecteze pe rând câte o regulă care se poate aplica, să aplice regula și să verifice dacă starea obținută este o stare finală. Cât timp rezultatul nu este cel dorit ciclul va continua. În mod evident, viteza de generare a unei soluții depinde de mecanismul utilizat pentru selectarea următoarei operații de realizat.

Pentru problema găleților, una dintre soluții constă din aplicarea regulilor în secvența 2, 9, 2, 7, 5, 9. În anumite situații problema cere găsirea **cele mai scurte secvențe** care duc la o stare finală. Această chestiune va influența mecanismul de ghidare a căutării.

O a doua chestiune se referă la regulile cu condiții suplimentare. Aceste condiții în multe cazuri nu sunt neapărat necesare, dar restrâng aria de aplicare a regulii, și în felul acesta contribuie la mărirea vitezei de generare a unei soluții.

O a treia chestiune este legată de acele reguli implicite care sunt în mod sigur permise, dar care, la o analiză superficială preliminară arată că aplicarea lor nu va conduce la o soluție (regulile 3 și 4). În general aceste reguli se vor elimina.

O a patra chestiune se referă la regulile speciale. Acestea sunt cazuri speciale ale unor reguli generale (ca de exemplu regulile $(0,2) \rightarrow (2,0)$ și $(2,y) \rightarrow (0,y)$). Astfel de reguli pot încetini mecanismul de căutare. Dacă se acordă prioritate acestor reguli speciale, în sensul că

se va verifica mai întâi dacă acestea pot fi aplicate și abia apoi se va trece la celelalte reguli, performanța sistemului poate crește.

În concluzie, pentru a construi o descriere formală a unei probleme trebuie să facem următoarele:

1. **Să definim un spațiu de stări** care conține toate configurațiile posibile ale obiectelor relevante (și poate unele configurații imposibile). Este posibil să definim un spațiu fără să enumerăm explicit toate stările care le conține.
2. **Să specificăm una sau mai multe stări** care descriu situații posibile de la care poate începe problema. Aceste stări se numesc **stări inițiale**.
3. **Să specificăm una sau mai multe stări** care ar fi acceptabile ca soluții ale problemei. Aceste stări se numesc **stări finale** sau **stări scop**.
4. **Să specificăm un set de reguli** care descriu acțiunile disponibile. Acest lucru va cere să ne gândim la următoarele chestiuni:
 - Ce presupuneri implicite sunt prezente în descrierea informală a problemei?
 - Cât de generale trebuie să fie regulile?
 - Cât de mult din munca cerută pentru rezolvarea problemei trebuie să fie reprezentată în reguli?

Problema poate fi apoi rezolvată prin utilizarea regulilor, în combinație cu o strategie de control potrivită, prin deplasarea prin spațiul problemei până când am găsit un drum de la o stare inițială la o stare finală. Astfel procesul căutării este fundamental în procesul de rezolvare a problemelor. Aceasta nu înseamnă că nu pot fi utilizate abordări mai directe. Oricând este posibil, acestea pot fi incluse ca pași în procesul de căutare, prin codificarea lor în reguli. De exemplu, în problema găleților nu folosim căutarea pentru identificarea numărului cu proprietatea că este egal cu $y - (4 - x)$.

Un ultim aspect: soluția finală poate consta fie dintr-o stare finală, fie dintr-un drum de la o stare inițială la o stare finală.

2.2. Sisteme de producție

Deoarece căutarea formează nucleul multor procese inteligente, este util să structurăm programele de AI într-un mod care să faciliteze descrierea și realizarea procesului de căutare. Sistemele de producție oferă astfel de structuri. O definiție a sistemelor de producție este dată mai jos. A nu se confunda cu alte utilizări ale cuvântului producție, ca de exemplu descrierea a ceea ce se întâmplă într-o fabrică. Un **sistem de producție** constă din:

Un set de reguli, fiecare constând dintr-o parte stângă (model) care determină aplicabilitatea regulii și o parte dreaptă care descrie operațiile care trebuie realizate dacă regula este aplicată.

Una sau mai multe baze de cunoștințe / baze de date care conțin orice informație etse necesară pentru o anumite problemă. Unele părți ale bazei de date sunt permanente, în timp ce alte părți pot aparține doar soluției unei anumite probleme. Informațiile din aceste baze de date pot fi structurate într-un mod corespunzător.

O strategie de control care specifică ordinea în care regulile vor fi comparate cu baza de date și un mod de a rezolva conflictele care apar când se potrivesc mai multe reguli în același moment.

Un mecanism de aplicare a regulilor.

Această definiție este foarte generală. Conține foarte multe sisteme, incluzând descrierile noastre ale jucătorului de șah și a rezolvitorului problemei găleților cu apă. De-aseamenea, conține o familie de interpretoare de sisteme de producție, cum ar fi:

- Limbaje de sisteme de producție de bază.
- Sisteme mai complexe, deseori hibride, numite shell-uri de sisteme experte, care oferă medii relativ complete pentru construirea sistemelor experte bazate pe cunoștințe.
- Arhitecturi generale de rezolvare a problemelor.

2.2.1. Strategii de control

Întrebare: cum decidem care regulă trebuie aplicată în continuare, în cadrul procesului de căutare a unei soluții? Această întrebare apare mai ales datorită faptului că aproape întotdeauna mai multe reguli (și uneori nici una) se potrivesc în partea stângă cu starea curentă.

- **Prima cerință a unei strategii de control bune este că cauzează mișcare.** Dacă implementăm o strategie de control simplă, care începe întotdeauna cu prima regulă aplicabilă, s-ar putea să intrăm în ciclu infinit și să nu cauzăm mișcare. Strategiile de control care nu cauzează mișcare nu vor conduce la soluție.
- **A doua cerință a unei strategii de control bune este că trebuie să fie sistematică.** Fie o strategie de control care la fiecare moment selectează aleator o regulă dintre toate regulile aplicabile. Această strategie este mai bună decât prima, deoarece cauzează mișcare. Eventual, într-un final, va produce o soluție. Dar este foarte probabil că vom reveni la aceeași stare de mai multe ori în timpul procesului și vom folosi mai mulți pași decât sunt necesari.

Iată un exemplu de strategie de control sistematică pentru problema găleților:

- Se construiește un arbore cu starea inițială ca rădăcină.
- Se generează toți descendenții rădăcinii prin aplicarea tuturor regulilor posibile stării inițiale.
- Pentru fiecare nod terminal generează toți succesorii prin aplicarea tuturor regulilor posibile.
- Continuă acest proces până la generarea unei stări finale.

Acest proces se numește **căutare Breadth-First** și constă din următorii pași:

1. Crează o variabilă numită **NODE-LIST** și setează-o la starea inițială.
2. Până când se produce o stare finală sau până când **NODE-LIST** este vidă, execută:
 - (a) Înlătură primul element din **NODE-LIST** și numește-l **E**. Dacă **NODE-LIST** era vid, atunci **STOP**.
 - (b) Pentru fiecare regulă care se potrivește cu starea descrisă de **E**, execută:
 - i. Aplică regula pentru a genera o stare nouă.
 - ii. Dacă starea nouă este o stare finală, **STOP** și întoarce această stare.
 - iii. Altfel, adaugă starea nouă la sfârșitul lui **NODE-LIST**.

Un **exemplu** de aplicare a algoritmului de căutare Breadth-First pentru problema găleților cu apă este dat în Figurile 2.2 și 2.3.

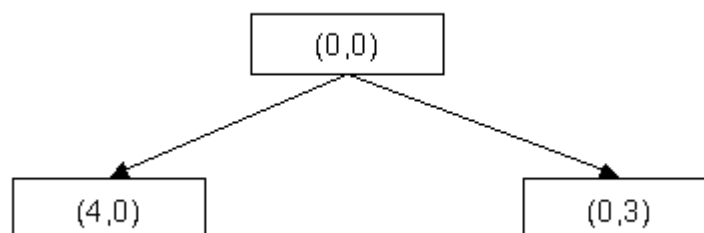


Figura 2.2: Un nivel al arborelui de căutare Breadth-First

Mai sunt disponibile și alte strategii. De exemplu, am putea să urmărim o singură cale a arborelui până produce o soluție sau până se ia o decizie de terminare a drumului. Un drum se poate termina fie când ajungem la un blocaj, fie când devine mai lung sau mai costisitor decât o anumită limită. În acest moment, apare backtracking-ul. Starea creată cel mai recent și de la care sunt disponibile reguli alternative neexplorate va fi revizitată și se va genera o nouă stare.

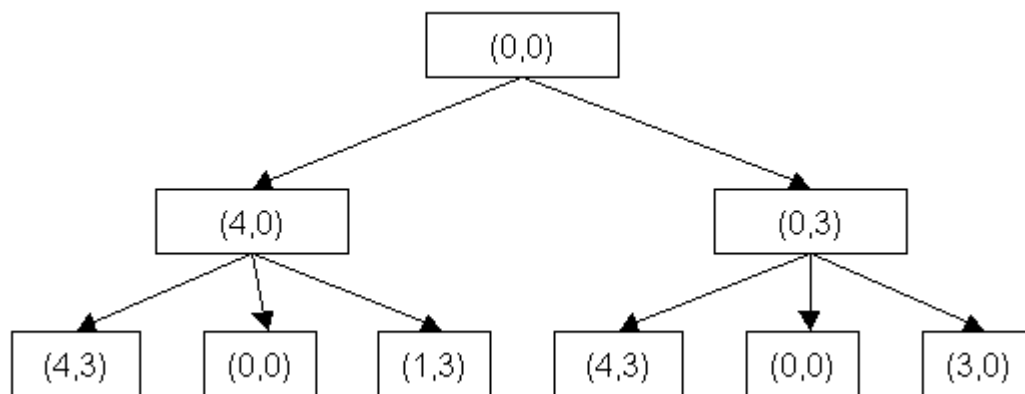


Figura 2.3: Două nivele ale arborelui de căutare Breadth-First

Această formă de backtracking se numește **backtracking cronologic** deoarece ordinea în care pașii sunt desfăcuți depinde doar de secvența temporală în care aceștia au fost făcuți inițial.

Procedura de căutare descrisă mai sus este cunoscută și sub numele de **căutare Depth-First**. Algoritmul este următorul:

1. Dacă starea inițială este o stare finală, STOP și raportează succes.
2. Altfel, execută următoarele până când se semnalează succes sau eșec.
 - (a) Generează un succesor E al stării inițiale. Dacă nu sunt succesori semnalează eșec.
 - (b) Apelează Depth-First Search cu E ca stare inițială.
 - (c) Dacă se raportează succes, semnalează succes. Altfel continuă în această buclă.

De **exemplu**, începutul arborelui de căutare Depth-First pentru problema găleșilor cu apă este ramura $(0,0) \Rightarrow (4,0) \Rightarrow (4,3)$.

Avantajele căutării Depth-First

- Cere mai puțină memorie deoarece sunt memorate doar nodul curent și drumul. Aceasta contrastează cu căutarea Breadth-First, unde se memorează întregul arbore care a fost generat până acum.
- Din întâmplare, sau dacă stările succesor sunt ordonate cu grijă, căutarea DF poate găsi o soluție fără să examineze o parte mrea mare din spațiul de căutare. Aceasta contrastează cu căutarea BF, unde întregul arbore trebuie examinat până la nivelul n pentru a putea examina vre-un nod de pe nivelul $n + 1$.

Avantajele căutării Breadth-First

- Căutarea BF nu se va împotmoli căutând pe un drum fără ieșire. Aceasta contrastează cu căutarea DF care poate merge pe un singur drum multă vreme înainte de a trece la un alt drum. Aceasta este o problemă în special în cazul în care există bucle, adică o stare are un succesor care i-a fost predecesor.
- Dacă există o soluție, căutarea BF o găsește în mod garantat. Mai mult, dacă sunt mai multe soluții, BF găsește una minimală. Spre deosebire, DF poate găsi un drum mai lung la o soluție, chiar dacă există un drum mai scurt în altă parte a arborelui.

În mod clar dorim să combinăm avantajele acestor două metode. Vom vedea ceva mai târziu cum putem face acest lucru când avem mai multe informații adiționale.

Exemplu: Problema comis-voiajorului. Un comis-voiajor are o listă de orașe, fiecare dintre care trebuie vizitate exact o dată. Sunt drumuri directe între fiecare pereche de orașe de pe listă. Găsiți drumul pe care comis-voiajorul trebuie să-l urmeze pentru cea mai scurtă călătorie care începe și se termină într-un singur (oricare) oraș.

O strategie simplă, care cauzează mișcare și care este sistematică poate rezolva problema prin explorarea tuturor drumurilor posibile și raportarea celui mai mic. Dacă sunt N orașe, numărul de drumuri posibile este $(N-1)!$, iar timpul necesar examinării unui drum este proporțional cu N . Deci timpul total cerut pentru rezolvarea problemei este proporțional cu $N!$. Pentru 10 orașe rezultă $10! = 3,628,800$ drumuri posibile. Acest fenomen de înmulțire a spațiului de stări se numește explozie combinatorială. Pentru a o combate avem nevoie de o strategie de control nouă.

O soluție este utilizarea tehnicii numite **Branch-and-Bound**. Începem să generăm drumuri complete, și memorăm cel mai scurt drum găsit până acum. Oprim explorarea oricărui alt drum până ce lungimea parțială a drumului curent devine mai mare decât cel mai scurt drum găsit până acum. Această tehnică garantează soluția, dar, deși este mai eficientă decât prima metodă, este tot exponențială.

2.2.2. Căutare euristică

Pentru a rezolva eficient multe probleme dificile este deseori necesar să facem un compromis între cerințele de mobilitate și sistematicitate și să construim o structură de control care nu mai garantează obținerea celui mai bun răspuns, ci că va produce aproape întotdeauna un răspuns foarte bun. Astfel introducem ideea de euristică. O **euristică** este o tehnică care îmbunătățește eficiența unui proces de căutare, eventual prin sacrificarea pretențiilor de completitudine. Euristicile sunt la fel ca **tururile ghidate**: sunt bune în măsura în care indică spre direcții interesante; sunt proaste în măsura în care omit elemente de interes pentru diversi indivizi. Termenul provine din limba greacă: *heuriskein* înseamnă *a descoperi*.

Unele euristici ajută la ghidarea unui proces fără pierderea condițiilor de completitudine pe care procesul le-ar fi avut înainte. Altele (multe dintre cele mai bune) pot în unele cazuri să ducă la omiterea unui drum excelent. Dar, în medie, acestea îmbunătățesc calitatea drumurilor care sunt explorate. Prin utilizarea unor euristici bune putem spera să obținem soluții bune (dar posibil ne-optimale) la probleme dificile (ca de exemplu problema comis-voiajorului) într-un timp sub cel exponențial. Există un număr de **euristici generale**, utile într-o mare varietate de probleme. În plus, se pot construi și **euristici specifice**, care exploatează cunoștințe particulare din domeniul problemei.

Un exemplu de euristică generală este **euristica celui mai apropiat vecin** (nearest neighbour), care lucrează prin selectarea la fiecare pas a celei mai bune alternative locale. Aplicarea ei la problema comis-voiajorului conduce la procedura următoare:

1. Selectează un oraș de start arbitrar.
2. Pentru a selecta următorul oraș, analizează orașele încă nevizitate și selectează-l pe cel mai apropiat. În continuare deplasează-te acolo.
3. Repetă pasul 2 până când toate orașele au fost vizitate.

Această procedură se execută într-un timp proporțional cu N^2 , o îmbunătățire semnificativă față de $N!$ și se poate demonstra o limită superioară a erorii pe care o produce. Pentru euristici de interes general este deseori posibil să demonstrăm astfel de margini de eroare, care oferă asigurări că nu plătim vitezei un preț prea mare în exactitatea soluției.

În multe probleme nu este posibil totuși să producem astfel de limite liniștitoare, din două motive:

- Pentru probleme din viața reală este deseori greu de măsurat precis valoarea unei soluții particulare.
- Pentru probleme din viața reală este deseori util de introdus euristici bazate pe cunoștințe relativ nestructurate, și astfel este aproape imposibil de efectuat o analiză matematică a efectului asupra procesului de căutare.

Iată câteva argumente în favoarea utilizării euristicilor:

- Fără euristici nu am putea evita explozia combinatorială.
- Doar rareori dorim soluția optimă; de obicei ne este la fel de utilă o bună aproximare a sa. De fapt există indicații că oamenii, când rezolvă probleme, nu sunt generatori de soluții optime ci generatori de soluții satisfăcătoare. De exemplu, căutarea unui loc de parcare: se opresc la prima soluție satisfăcătoare, chiar dacă o soluție mai bună se găsește ceva mai încolo.

- Deși aproximațiile produse de euristici ar putea să nu fie foarte bune în cel mai rău caz, aceste cele mai rele cazuri rareori apar în viața reală.
- Încercarea de a înțelege de ce o euristică funcționează sau nu conduce deseori la o înțelegere mai adâncă a problemei.

Există două metode majore prin care cunoștințe euristice specifice domeniului pot fi încorporate într-o procedură de căutare bazată pe reguli:

- În regulile însele. De exemplu, regulile jocului de șah ar putea descrie nu doar mulțimea de mutări legale, ci o mulțime de mutări “semnificative”, așa cum au fost determinate de autorul regulilor.
- Ca o funcție euristică care evaluează stări de probleme individuale și determină cât de importante sunt acestea (cât de aproape de soluție ne duc).

O **funcție euristică** este o funcție definită pe mulțimea stărilor problemei cu valori într-o mulțime de măsuri de importanță, uzual reprezentate de numere. Care aspecte ale stărilor problemei sunt considerate, cum sunt evaluate aceste stări ale problemei, și ponderile date diverselor aspecte sunt alese într-un astfel de mod încât valoarea funcției euristice la un nod dat este o estimare cât de bună posibil a dacă acel nod este pe drumul dorit către o soluție.

Funcțiile euristice bine proiectate pot juca un rol important în ghidarea eficientă a unui proces de căutare spre o soluție. Uneori funcții euristice foarte simple pot oferi o estimare destul de bună a calității stării curente sau drumului curent. În alte situații este nevoie de funcții euristice mai complexe. Iată unele funcții euristice simple:

Șah: Avantajul material al nostru asupra adversarului

Comis-voiajorul: Suma distanțelor parcurse până acum

Tic-Tac-Toe: 1 pentru fiecare rând în care putem câștiga și în care avem deja o piesă plus 2 pentru fiecare astfel de rând în care avem două piese

De notat că optimul înseamnă maxim pentru problemele 1 și 3 și minim pentru problema 2. Acest aspect nu este esențial.

Scopul unei funcții euristice este să ghideze procesul de căutare în direcția cea mai profitabilă sugerând care cale trebuie urmată atunci când sunt posibile mai multe căi. Cu cât mai exact estimează o funcție euristică meritele reale ale nodurilor, cu atât mai direct este procesul de căutare. La extremă, funcția euristică este atât de bună încât nu are loc nici o căutare, sistemul se va deplasa direct la soluție. Dar costul calculării valorilor unei astfel de funcții va depăși efortul economisit în procesul de căutare. În general există un echilibru între costul evaluării unei funcții euristice și economiile în timp de căutare pe care aceste funcții le oferă.

Până acum am descris soluțiile problemelor de IA ca centrate pe un proces de căutare. Din discuția de aici va fi clar că este vorba de un proces de căutare euristică.

Aceasta conduce la un **alt mod de a defini IA**: studiul tehnicilor pentru rezolvarea problemelor exponențiale dificile în timp polinomial prin exploatarea cunoștințelor despre domeniul problemelor.

2.3. Caracteristicile problemei

Căutarea euristică este o metodă foarte generală aplicabilă unei clase mari de probleme. Cuprinde o varietate de tehnici specifice, fiecare în mod particular eficace pentru o clasă mică de probleme. Pentru a selecta cea mai potrivită metodă (sau combinație de metode) este necesar să analizăm problema prin câteva elemente cheie:

- Problema se poate descompune într-un set de probleme (aproape) independente mai mici sau mai ușor de rezolvat?
- Pașii pe drumul către soluție se pot ignora, sau cel puțin se pot reface dacă au fost neînțelepți?
- Este universul problemei previzibil?
- Este evidentă o soluție bună a problemei fără a face comparații cu toate celelalte soluții posibile?
- Este soluția dorită o stare a lumii sau un drum către o stare?
- Este necesară o mare cantitate de cunoștințe pentru a rezolva problema, sau doar pentru a limita căutarea?
- Poate un program care primește problema să întoarcă direct soluția, sau soluția problemei implică interacțiunea dintre programi o persoană?

În continuare vom analiza fiecare din aceste chestiuni în detaliu. De observat că aceste întrebări nu implică doar enunțarea problemei, ci și descrierea caracteristicilor soluției dorite și circumstanțele în care soluția trebuie să apară.

2.3.1. Problema se poate descompune?

O problemă se poate rezolva prin împărțirea în subprobleme mai mici. Fiecare dintre acestea pot fi rezolvate prin utilizarea unei colecții mici de reguli specifice. Arborele problemei va fi generat în mod recursiv de procesul de descompunere a problemei. La fiecare pas se verifică dacă problema pe care se lucrează este solvabilă imediat. Dacă da, răspunsul se întoarce direct.

Dacă nu, programul va vedea dacă poate împărți mai departe problema în subprobleme. Dacă poate, atunci crează acele subprobleme și se autoapelează pe ele. Această tehnică se numește **descompunerea problemei (problem decomposition)**.

Un exemplu de problemă decompozabilă este **calculul primitivei unei funcții**. Aceasta se poate descompune în primitive ale unor funcții mai simple, care sunt independente între ele.

Să luăm un exemplu de problemă din **lumea blocurilor**, prezentată în Figura 2.4.



Figura 2.4: Exemplu de problemă din lumea blocurilor

Să considerăm următorii operatori:

1. CLEAR(x) [blocul x nu are nimic pe el] \Rightarrow ON(x , Masă) [ia-l pe x și pune-l pe masă]
2. CLEAR(x) și CLEAR(y) \Rightarrow ON(x , y) [pune-l pe x peste y]

Aplicarea tehnicii descompunerii problemei la această lume simplă a blocurilor va duce la arborele de soluții prezentat în Figura 2.5. În figură, scopurile sunt subliniate. Stările obținute nu sunt subliniate.

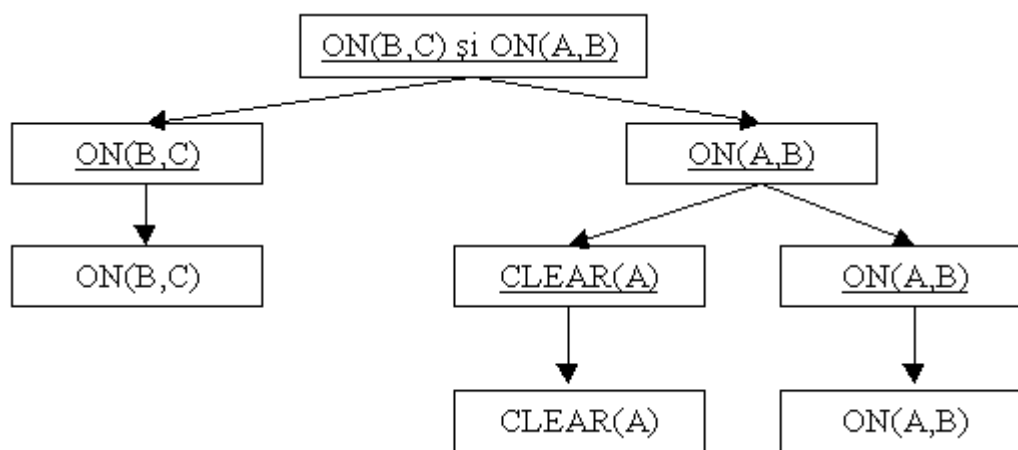


Figura 2.5: O propunere de soluție pentru problema blocurilor

Ideea este clară: împărțim problema **ON(B,C) și ON(A,B)** în problemele **ON(B,C)** și **ON(A,B)**. Pentru prima subproblemă nu trebuie decât să luăm pe B de pe masă și să îl punem peste C. A doua problemă nu mai este chiar așa de simplă. Pentru a putea pune pe A peste B trebuie ca A să fie CLEAR, ceea ce se obține prin înlăturarea lui C de pe A. Acum, dacă

încercăm să combinăm cele două subsoluții, vom eșua. Indiferent care operație o facem prima, cealaltă nu poate fi realizată așa cum am planificat. În această problemă cele două subprobleme nu sunt independente. Ele interacționează, și interacțiunea lor trebuie să fie luată în considerare la stabilirea unei soluții pentru întreaga problemă. Problema descompunerii va fi abordată în ambele cazuri ceva mai târziu.

2.3.2. Pașii pe drumul către soluție se pot ignora sau reface?

Să presupunem că avem de **demonstrat o teoremă** matematică. Incepem prin demonstrarea unei leme care credem că ne va fi utilă. În final vom observa că lema nu ne ajută la nimic. Sigur că nu este nici o problemă. Tot ceea ce trebuie să știm pentru a demonstra teorema este în continuare adevărat și nu este afectat de inutilitatea lemei. Tot ceea ce am pierdut este efortul inutil pentru demonstrarea lemei. Toate regulile valabile înainte de demonstrarea lemei sunt valabile și în continuare.

Iată în continuare o a doua problemă: **8-puzzle**. Există un cadru pătrat cu nouă spații în care sunt opt piese numerotate. Al nouălea spațiu este gol. O piesă adiacentă spațiului poate aluneca în acel spațiu. Jocul constă dintr-o poziție de start și una finală. Scopul este de a transforma poziția de start în poziție finală prin alunecarea pieselor.

Start			Scop		
2	8	3	1	2	3
1	6	4	8		4
7		5	7	6	5

Figura 2.6: Un exemplu al problemei 8-puzzle

Un exemplu este indicat în Figura 2.6. În încercarea de a rezolva puzzle-ul am făcut niște mutări prostești. De exemplu am putea începe prin plasarea lui 5 în spațiul liber. Imediat după aceasta nu ne putem răzgândi și să mutăm 6 în spațiu, pentru că spațiul este deja ocupat de 5, iar noul spațiu este altundeva. Dar putem reface operația de deplasare a lui 5, și îl putem deplasa înapoi unde era. Acum putem mișca și piesa 6. Greșelile se pot reface, dar nu mai este chiar așa ușor ca la demonstrarea teoremelor. Se cere câte un pas adițional pentru refacerea unui pas făcut greșit. În plus, operatorul trebuie să rețină pașii făcuți, pentru a putea fi desfăcuți dacă se va dovedi necesar. Structura de control a demonstratorului de teoreme nu trebuie să memoreze toate aceste informații.

Iată acum o a treia problemă: **jocul de șah**. Să presupunem că un program de jucat șah face o mutare stupidă și realizează acest lucru câțiva pași mai târziu. Nu poate să joace ca și cum nu ar fi făcut acea mutare. Nici nu poate da jocul înapoi și reface jocul din punctul în care a făcut mutarea stupidă. Tot ceea ce poate face este să se descurce cât mai bine din poziția curentă.

Aceste trei probleme ilustrează diferențele dintre trei clase de probleme importante:

- **ignorabile**, în care pașii către soluție pot fi ignorați;
- **recuperabile**, în care pașii către soluție pot fi recuperați;
- **irecuperabile**, în care pașii către soluție sunt definitivi.

Aceste trei definiții se referă la pașii către soluție și astfel par să caracterizeze sisteme de producție particulare, și nu problemele însele. Probabil, o altă formulare a aceleiași probleme ar putea conduce la o altă caracterizare a ei. În sens strict, aceasta este adevărat. Dar din punct de vedere practic, pentru multe probleme există doar o singură formulare (sau un număr mic de formulări esențiale) care descrie problema *în mod natural*.

Caracterul recuperabil al unei probleme joacă un rol important în determinarea complexității structurii de control necesare pentru rezolvarea problemei. Problemele ignorabile pot fi rezolvate folosind o structură de control care nu face backtracking. O astfel de structură este ușor de implementat. Problemele recuperabile pot fi rezolvate printr-o strategie de control ceva mai complicată, care uneori face greșeli. Structura de control trebuie implementată cu o stivă în care se vor memora deciziile, dacă acestea vor trebui ‘date înapoi’. Problemele irecuperabile vor necesita sisteme care consumă o cantitate mare de efort pe luarea deciziilor, deoarece decizia este definitivă. Unele probleme irecuperabile pot fi rezolvate prin metode de tip recuperabil, în care lanțuri întregi de pași se examinează înainte pentru a descoperi dacă unde va conduce.

2.3.3. Este universul previzibil?

Să luăm din nou jocul **8-puzzle**. De fiecare dată când facem o mutare știm exact ce se va întâmpla. Adică, este posibil să planificăm o succesiune de pași înainte de a-l face pe primul și vom ști dinainte care va fi starea rezultată. Putem folosi tehnici de planificare pentru a evita refacerea unor pași greșiți, deși s-ar putea să fie nevoie de refacerea unor pași luați în cadrul procesului de planificare. Astfel, va fi necesară o structură de control cu backtracking.

În alte jocuri această planificare nu poate fi posibilă. De exemplu, **jocurile de cărți (canastă, bridge)**. Am dori să planificăm modul în care jucăm o mână de cărți, dar nu putem face aceasta deoarece nu știm cum sunt împărțite cărțile între parteneri. Tot ce putem face este să investigăm câteva planuri și să folosim probabilitățile pentru a alege un plan care are cea mai mare probabilitate estimată de a conduce la un scor bun.

Aceste două probleme ilustrează diferențele dintr problemele cu rezultat cunoscut și cele cu rezultat necunoscut. În cazul problemelor din a doua clasă, planificarea trebuie însoțită te un proces de revizuire a planului, care trebuie executat pe măsură ce apare feedback.

Ultimele două caracteristici discutate (recuperabil versus irecuperabil și rezultat cunoscut versus necunoscut) interacționează într-un mod interesant. O modalitate de a rezolva problemele irecuperabile este planificarea unei soluții întregi înainte de a face primul pas. Dar acest proces de planificare poate fi realizat doar la problemele cu rezultat cunoscut. Deci o clasă de

probleme foarte dificile este cele **irecuperabile și cu rezultat necunoscut**. Iată câteva exemple:

- jocul de bridge;
- controlul brațului unui robot;
- asistență pentru avocatul unui individ acuzat de crimă.

2.3.4. O soluție bună este absolută sau relativă?

Fie problema de a **răspunde la întrebări** bazate pe o bază de date de fapte relativ simple, ca de exemplu:

1. Marcus era un om.
2. Marcus era din Pompei.
3. Marcus s-a născut în anul 40 AD.
4. Toți oamenii sunt muritori.
5. Toți cei din Pompei au murit la erupția vulcanului, în anul 79 AD.
6. Nici un muritor nu trăiește mai mult de 150 ani.
7. Acum este anul 1998 AD.

Fie următoarea întrebare: “Este Marcus în viață?”. Reprezentăm fiecare fapt într-un limbaj formal, cum ar fi logica predicatelor, și apoi folosim metode de inferență formală. Putem produce rapid un răspuns la problemă. Important este răspunsul, nu ne interesează care drum până la soluție îl urmăm. Dacă urmăm un drum până la soluție, nu este nici un motiv să ne întoarcem și să verificăm dacă mai există vre-unul. Iată în continuare două modalități de a decide că Marcus este mort.

- (1) Marcus era un om.
- (4) Toți oamenii sunt muritori.
- $(1, 4 \Rightarrow 8)$ Marcus este muritor.
- (3) Marcus s-a născut în anul 40 AD.
- (7) Acum este anul 1998 AD.
- $(3, 7 \Rightarrow 9)$ Marcus are vârsta de 1958 ani.

- (6) Nici un muritor nu trăiește mai mult de 150 ani.
- (8, 6, 9 \Rightarrow 10) Marcus este mort.

și

- (7) Acum este anul 1998 AD.
- (5) Toți cei din Pompei au murit în anul 79 AD.
- (7, 5 \Rightarrow 11) Toți cei din Pompei sunt morți acum.
- (2) Marcus era un Pompeian.
- (11, 2 \Rightarrow 12) Marcus este mort.

Să considerăm acum din nou problema **comis-voiajorului**. scopul nostru este să găsim drumul cel mai scurt care vizitează toate orașele exact o dată. Este evident că nu avem nici o garanție că primul drum găsit este cel mai scurt.

Aceste două probleme ilustrează diferența dintre problemele de tip orice-drum și cele de tip drum-optimum. Problemele de drum optimum sunt computațional mai dificile decât problemele de tip orice-drum. Problemele de tip orice-drum pot fi rezolvate adesea într-un timp rezonabil prin euristici care sugerează drumuri bune care se pot explora. Dacă euristicele nu sunt cele mai bune, căutarea către o soluție ar putea să nu fie cea mai directă posibil, dar aceasta nu este o problemă. Pentru problemele de drum optimum nu se pot utiliza euristici care pot rata drumul optimum. Astfel va trebui să se realizeze o căutare mult mai exhaustivă.

2.3.5. Este soluția o stare sau un drum?

Fie **problema găsirii unei interpretări consistente** pentru o anumită propoziție. De exemplu, "*Președintele de bancă a mâncat o farfurie de salată de paste cu furculița*". Sunt mai multe componente care, luate individual, au mai multe interpretări. Împreună componentele formează un întreg și își limitează reciproc interpretările. Iată câteva surse de ambiguitate:

- **bancă**: instituție sau loc de șezut;
- **a mâncat** o farfurie sau salata din ea;
- **cu furculița** se referă la salată sau la mâncare.

Din cauza interacțiunii dintre interpretările constituenților, va trebui să facem căutare pentru a găsi o interpretare corectă. Dar pentru a găsi interpretarea trebuie să producem doar interpretarea, nu este nevoie să reținem procesul prin care s-a identificat interpretarea corectă.

Fie **problema găleților cu apă**. Aici nu este suficient să identificăm că am rezolvat problema și că starea finală este (2,0). Trebuie să producem drumul care ne-a condus la soluție: o secvență de operații, numită și plan.

Aceste două exemple ilustrează diferențele dintre problemele ale căror soluție este o stare și cele ale căror soluție este drumul către o stare. La un anumit nivel chestiunea poate fi ignorată și problema poate fi formulată astfel încât să se raporteze doar o stare. Trebuie doar să redescrim stările astfel încât o stare să reprezinte un drum parțial către soluție. Deci problema nu este semnificativă. Atâta doar că deseori există o formulare naturală și economică în care o stare corespunde unei anumite situații, nu unei secvențe de operații. Răspunsul la această întrebare ne va spune dacă trebuie să înregistrăm drumul către soluție sau nu.

2.3.6. Care este rolul cunoștințelor?

Din nou să luăm ca exemplu **jocul de șah**. Pentru un program de șah avem nevoie de foarte puține informații: doar regulile pentru determinarea mutărilor legale și un mecanism de control pentru implementarea unei proceduri de căutare corespunătoare. Cunoștințe adiționale despre strategii sau tactici de joc sunt în măsură să limiteze căutarea și să grăbească soluția.

Al doilea exemplu: **problema scanării unui articol de ziar** și a identificării răspunsului la o anumită întrebare. De această dată cantitatea de cunoștințe necesare este foarte mare, pentru că trebuie memorate toate numele, datele, faptele și relațiile care apar în articolul respectiv.

Aceste două probleme ilustrează diferențele dintre probleme pentru care se cer multe cunoștințe doar pentru a limita căutarea și probleme pentru care se cer multe cunoștințe chiar pentru a putea identifica o soluție.

2.3.7. Soluția cere interacțiunea cu o persoană?

Uneori este util să realizăm programe care rezolvă problemele în moduri pe care majoritatea oamenilor nu le pot înțelege. Aceasta este în regulă pentru cazul în care nivelul de interacțiune între computer și utilizatorii umani este **problem-in-solution-out**. Dar începem să realizăm într-o măsură tot mai mare programe care cer interacțiune cu oamenii, atât pentru a oferi intrare adițională pentru program cât și pentru a oferi asigurări suplimentare pentru utilizator.

De exemplu, în cazul **demonstrației automate a teoremelor**, dacă

- (a) tot ceea ce dorim este să știm dacă există o demonstrație;
- (b) programul este capabil să găsească demonstrația singur,

atunci nu contează ce strategie urmează programul pentru a găsi soluția. Dacă însă una dintre condiții nu are loc, atunci ar putea conta foarte mult cum se găsește o demonstrație. Am putea chiar să influențăm procedeul de deducție; deocamdată oamenii sunt mai buni la strategii de nivel înalt.

Putem deci distinge între două tipuri de probleme:

Solitare, în care computerul primește o descriere a problemei și produce un răspuns fără comunicare intermediară și fără o cerere de explicații în legătură cu procesul de raționare;

Conversaționale, în care există conversație intermediară între o persoană și computer, fie pentru a oferi asistență adițională computerului, fie utilizatorului.

Desigur, această distincție nu este strictă în ceea ce privește descrierea unor probleme particulare. Anumite probleme pot fi foarte ușor privite în ambele moduri. Dar de obicei unul din tipuri va fi cel natural.

2.3.8. Clasificarea problemelor

Când probleme reale sunt examinate din punctul de vedere al tuturor acestor întrebări, devine vizibil că există câteva clase mari în care se împart problemele. Aceste clase pot fi asociate cu o strategie de control generică care este potrivită pentru rezolvarea problemei. De exemplu, să considerăm problema generică a **clasificării**. Scopul este de a examina o intrare și de a decide de care clasă dintr-un set de clase dat aparține acea intrare. Majoritatea problemelor de diagnosticare (diagnostică medicală, diagnostica erorilor) sunt exemple de clasificare. Un alt exemplu de strategie generică este **propune și rafinează**. Multe probleme de proiectare și planificare pot fi abordate cu această strategie.

Depinzând de nivelul de granularitate la care încercăm să clasificăm problemele și strategiile de control, putem propune diferite liste de scopuri și proceduri generice. Trebuie reținut că nu există un mod unic de a rezolva o problemă. Dar nu trebuie nici să considerăm fiecare problemă nouă totalmente *ab initio*. În loc de aceasta, problemele trebuie analizate cu grijă și metodele de rezolvare sortate prin tipurile de probleme pentru care sunt potrivite. Astfel vom putea să utilizăm la fiecare problemă nouă multe din ceea ce am învățat din rezolvarea altor probleme similare.

2.4. Caracteristicile sistemelor de producție

Am examinat un set de caracteristici care separă clase variate de probleme. Am considerat că sistemele de producție sunt un mod bun de a descrie operațiile care pot fi realizate într-o căutare pentru soluția problemei. Apar două întrebări:

1. Sistemele de producție pot fi descrise printr-un set de caracteristici care aruncă o lumină asupra modului în care acestea pot fi implementate?
2. Dacă da, ce relații există între tipurile de probleme și tipurile de sisteme de producție cele mai potrivite pentru a rezolva problemele?

La prima întrebare răspunsul este **da**. Fie următoarele definiții ale unor clase de sisteme.

Un **sistem de producție monotonic** este un sistem de producție în care aplicarea unei reguli nu împiedică aplicarea ulterioară a unei alte reguli care ar fi putut fi aplicată la momentul în care prima regulă a fost selectată.

Un **sistem de producție nemonotonic** este unul în care afirmația de mai sus nu este adevărată.

Un **sistem de producție parțial comutativ** este un sistem de producție cu proprietatea că dacă aplicarea unui șir de reguli transformă starea x în starea y , atunci aplicarea oricărei permutări admisibile a aceluși șir de reguli va transforma starea x în starea y .

Un **sistem de producție comutativ** este un sistem de producție monotonic și parțial comutativ.

Semnificația acestor categorii de sisteme de producție stă în relația dintre categorii și strategiile de implementare corespunzătoare. Dar să vedem cum se leagă aceste definiții de probleme specifice.

În ceea ce privește a doua întrebare, pentru orice problemă rezolvabilă există un număr infinit de sisteme de producție care descriu moduri de a localiza soluția. Unele vor fi mai naturale sau eficiente decât altele. Orice problemă care poate fi rezolvată printr-un sistem de producție, poate fi rezolvată și printr-unul comutativ (clasa cea mai restrictivă). Dar ar putea fi atât de complicat încât să fie inutil. Ar putea folosi stări individuale pentru a reprezenta șiruri de reguli ale unui sistem non-comutativ mai simplu. Deci, în semns formal nu există vre-o legătură între clasele de probleme și clasele de sisteme de producție. Dar dintr-un punct de vedere practic există o astfel de legătură între tipuri de probleme și tipurile de sisteme de producție care sunt o descriere naturală a acestor probleme. Iată câteva exemple:

	Monotonic	Non-monotonic
Parțial comutativ	(1) Demonstrarea teoremelor	(2) Navigarea roboților
Non-parțial comutativ	(3) Sinteza chimică	(4) Bridge

Sistemele de categoria (1) sunt comutative. Sunt utile la rezolvarea **problemelor ignorabile**. Acestea sunt probleme pentru care o formulare naturală conduce la posibilitatea de a ignora pași ai soluției. Această formulare naturală va fi un sistem comutativ. Problemele care implică crearea de lucruri noi mai degrabă decât modificarea lucrurilor existente sunt în general ignorabile.

Aceste sisteme sunt importante din punct de vedere al implementării, deoarece pot fi implementate fără backtracking, deși o implementare cu backtracking este utilă pentru a genera o soluție sistematică.

Sistemele de categoria (2) sunt bune pentru problemele în care schimbările care apar pot fi desfăcute și în care ordinea operațiilor nu este critică. Acesta este cazul în probleme de manipulare fizică, cum ar fi navigarea roboților pe un plan orizontal. În funcție de reprezentare, și 8-puzzle și problemele din lumea blocurilor sunt din această categorie.

Ambele sisteme sunt importante din punct de vedere al implementării deoarece tind să ducă la multe multiplicări ale stărilor individuale în cadrul procesului de căutare.

Sistemele de categoriile (3) și (4) sunt utile pentru multe probleme în care pot să apară schimbări ireversibile. De exemplu problema determinării unui proces de a produce un compus chimic. Adăugarea unei componente sau schimbarea temperaturii pot afecta proprietățile compusului. Sistemele non-parțial comutative este puțin probabil să producă stări multiple.

2.5. Aspecte în proiectarea programelor de căutare

Procesul de căutare trebuie să găsească un drum sau drumuri în arborele care conectează starea inițială cu una sau mai multe stări finale. Arborii care trebuie căutați pot fi în principiu construiți în întregime. Dar în practică majoritatea arborelui nu este contruit în întregime. Este prea mare și nu este explorat în totalitate. În loc să se genereze arborele în mod **explicit**, majoritatea programelor îl reprezintă **implicit**, prin reguli și generează explicit doar acea parte pe care se decid să o explore.

Iată în continuare cele mai importante elemente care apar în tehnicile ce vor fi discutate în capitolul următor:

- Direcția în care trebuie condusă căutarea: **înainte** (de la starea inițială la o stare finală) sau **înapoi** (de la starea finală la starea inițială);
- Modul de selectare a regulilor aplicabile (**matching**). Sistemele de producție consumă de obicei mult timp pentru căutarea regulilor de aplicat, deci este critic să avem proceduri eficiente de potrivire a regulilor cu stările.
- Modul de reprezentare a procesului de căutare (**problema reprezentării cunoștințelor**). Pentru probleme ca șah-ul un nod poate fi reprezentat în întregime printr-un vector. În situații mai complexe este inefficient și/sau imposibil să se reprezinte toate faptele din lume și să se determine toate efectele laterale pe care o acțiune le poate avea.

Problemă: Arbori de căutare sau grafe de căutare

Ne putem gândi că regulile de producție generează noduri într-un arbore de căutare. Fiecare nod poate fi expandat generând un set de succesori. Acest proces continuă până când se găsește un nod care reprezintă o soluție. Implementarea unui astfel de arbore nu cere efort de memorare mare. Totuși, acest proces generează deseori un același nod ca parte a mai multor drumuri și astfel va fi procesat de mai multe ori. Aceasta se întâmplă deoarece spațiul de căutare poate fi în realitate un graf, și nu un arbore.

Fie de exemplu problema găleților cu apă. Fie arborele de căutare din Figura 2.7. Aici nodurile (4,3) și (0,0) se generează de mai multe ori. Nodul (4,3) de două ori pe nivelul trei,

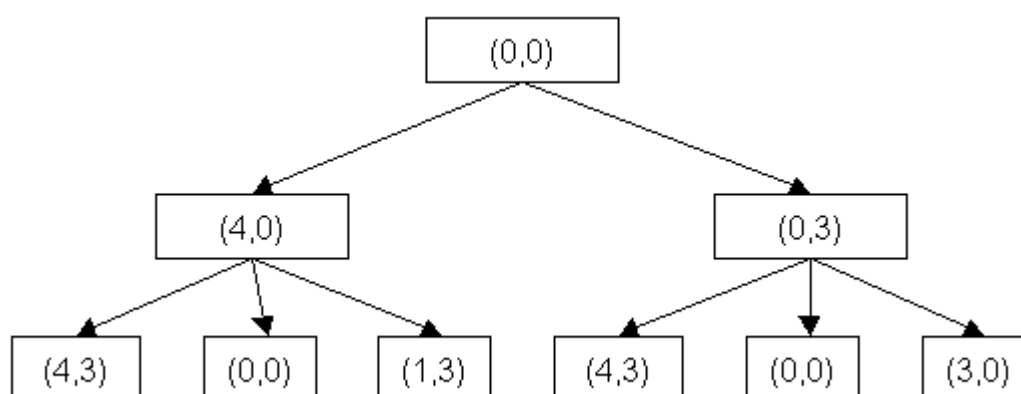


Figura 2.7: Un arbore de căutare pentru problema găleților cu apă

iar $(0,0)$ de două ori pe nivelul trei și o dată pe nivelul unu (rădăcina). Pentru a evita pierderea de timp și efort cu analizarea unor drumuri de mai multe ori, trebuie evitată generarea unui aceluiași nod de mai multe ori chiar cu prețul unei memorii adiționale. În loc să traversăm un arbore de căutare va trebui să traversăm un graf orientat. Graful diferă de arbore prin aceea că mai multe drumuri pot să ducă la un același nod. Graful corespunzător arborelui de mai sus este prezentat în Figura 2.8.

Orice procedură de căutare în arbori care memorează nodurile generate până în prezent poate fi convertită la o procedură de căutare în grafe prin modificarea acțiunii realizate la fiecare generare a vreunui nod. Din cele două proceduri de căutare sistematice discutate până acum, această condiție este verificată de Breadth-First, nu și de Depth-First. Dar, desigur, Depth-First poate fi modificat pentru a memora și nodurile care au fost expandate și apoi desfăcute. Deoarece toate nodurile sunt salvate în graful de căutare, în loc să adăugăm un nod la graf trebuie să executăm următorul algoritm.

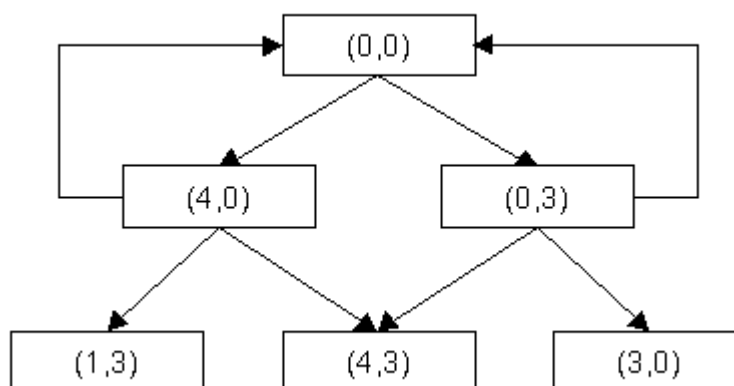


Figura 2.8: Un graf de căutare pentru problema găleților cu apă

Algoritm: Verificarea nodurilor duplicate

1. Examinează mulțimea nodurilor care au fost create până acum pentru a verifica dacă nodul nou există deja.
2. Dacă nu există, adaugă-l în graf în același mod ca pentru un arbore.
3. Dacă există, execută următoarele:
 - (a) Setează un pointer de succesori între nodul care este expandat și nodul care deja există, nu cel nou creat. Nodul creat poate fi distrus.
 - (b) Dacă se memorează drumul cel mai bun (din punctul de vedere al unui cost) la fiecare nod, verifică dacă drumul nou la nodul existent este mai bun sau mai vechi decât drumul vechi. Dacă este mai prost, nu fă nimic. Dar dacă este mai bun, memorează drumul nou ca drum corect până la nod și propagă schimbarea în cost în jos către nodurile succesor.

O problemă care poate să apară aici este că se pot introduce cicluri în graful de căutare. Un ciclu este un drum prin graf în care un nod apare de mai multe ori. În graful de mai sus există două cicluri: $(0,0)$ și $(4,0)$ și $(0,0)$ și $(0,3)$.

Tratarea procesului de căutare ca o căutare în grafe și nu în arbori reduce cantitatea de efort cheltuit pe explorarea aceluiași drum de mai multe ori. Dar aceasta cere efort suplimentar la generarea fiecărui nod pentru a vedea dacă a mai fost generat deja. Dacă este foarte probabil ca un anumit nod să fie generat de mai multe ori, atunci se recomandă folosirea căutării în grafe.

Procedurile de căutare în grafe sunt în special utile în abordarea sistemelor de producție parțial comutative (în care un set de operații va produce același rezultat indiferent de ordinea de aplicare a operațiilor). O căutare sistematică va verifica multe din permutările acestor operatori și va genera un nod de mai multe ori. Exact asta s-a întâmplat cu problema găleților cu apă.

2.6. Alte probleme

Înainte de a prezenta această listă de probleme, trebuie să reținem că IA nu este o știință care rezolvă astfel de probleme-jocuri. Multe dintre tehnicile dezvoltate pentru ele au devenit nucleul multor sisteme care rezolvă probleme foarte serioase.

Problema 1. Misionarii și canibalii

Trei misionari și trei canibali sunt de o parte a unui râu. Toți doresc să treacă de partea cealaltă. Pentru a nu fi în pericol misionarii nu doresc să rămână în minoritate de nici o parte a râului. Singurul mijloc de navigație disponibil este o barcă cu două locuri. Cum poate trece toată lumea râul fără ca misionarii să riște să fie mâncați?

Problema 2. Turnurile din Hanoi

Undeva lângă Hanoi este o mănăstire ai cărei călugări se concentrează asupra unei probleme foarte importante. În curtea mănăstirii sunt trei stâlpi. Pe acești stâlpi sunt 64 discuri, fiecare cu o gaură în centru și de raze diferite. La înființarea mănăstirii toate discurile erau plasate pe primul stâlp, fiecare situat peste discul de rază imediat mai mare. Sarcina călugărilor este să mute discurile pe stâlpul al doilea. Nu se poate muta decât un disc la un moment dat și toate celelalte discuri trebuie să fie pe unul din stâlpi. Nu se poate plasa un disc deasupra unui disc de rază mai mică. Al treilea stâlp poate fi folosit ca spațiu de manevră. Care este cel mai rapid mod de a deplasa discurile?

Problema 3. Criptaritmetică

Se consideră o problemă de aritmetică reprezentată de litere. Atribuiți o cifră zecimală fiecărei litere astfel încât operația de adunare să fie corectă. Dacă o literă apare de mai multe ori i se atribuie aceeași cifră. Două litere distincte nu vor primi aceeași cifră.

$$\begin{array}{r} \text{SEND} + \\ \text{MORE} \\ \hline \text{MONEY} \end{array}$$

$$\begin{array}{r} \text{DONALD} + \\ \text{GERALD} \\ \hline \text{ROBERT} \end{array}$$

$$\begin{array}{r} \text{CROSS} + \\ \text{ROADS} \\ \hline \text{DANGER} \end{array}$$

2.7. Aplicații rezolvate și probleme propuse

2.7.1 (Problema comis-voiajorului). Se dau n orașe, numerotate $1, 2, \dots, n$. Între orașe există legături directe, al căror cost se cunoaște. Un comis-voiajor vrea să plece dintr-un oraș, să se întoarcă în orașul de plecare, vizitând o singură dată fiecare oraș, pe un drum de cost total minim. Se cere să se furnizeze traseul comis-voiajorului.

Rezolvare

Considerând cele n orașe ca vârfuri ale unui graf neorientat, iar legăturile dintre orașe ca muchii ale grafului, problema se reduce la determinarea ciclului hamiltonian de cost total minim într-un graf neorientat (ciclul care trece prin toate vârfurile grafului și are costul total minim). Pentru această problemă nu se cunosc încă algoritmi polinomiali de rezolvare. Propunem două variante de rezolvare, cea de-a doua fiind o tehnică de Inteligență Artificială.

Specificarea problemei

Date

n – numărul de orașe

$a(n, n)$ – matricea reprezentând legăturile dintre orașe (matricea de adiacență a grafului), cu semnificația: $a[i, j] = c$, dacă între orașele i și j există o legătură directă al cărei cost este c , respectiv $a[i, j] = 0$, dacă $i = j$ sau dacă între orașele i și j nu există legătură directă

Rezultate

$o(n)$ – vectorul conținând ordinea în care vor fi vizitate orașele

c – costul total al traseului

Varianta 1

Algoritmul de rezolvare se bazează pe metoda *backtracking* și constă în a genera toate ciclurile hamiltoniene din graf și a-l reține pe cel de cost total minim.

Se vor genera toate succesiunile de orașe $x_1, x_2, \dots, x_n, x_{n+1}$ satisfăcând următoarele condiții:

- $x_1 = x_{n+1}$
- elementele vectorului sunt distincte (excepție făcând capetele, care sunt egale)
- pentru orice $i = 1, \dots, n$, x_i și x_{i+1} sunt adiacente (unite prin muchie)

Observații

Metoda folosită spre rezolvare

- garantează obținerea soluției optime;
- nu este eficientă, datorită timpului ridicat de execuție, proporțional cu $n!$ (backtrackingul nu evită explozia combinatorială).

procedura POSIBIL(a, n, k, x, c)

- 1: {verifică dacă elementul $x[k]$ este corect ales în soluție}
- 2: { c reprezintă costul traseului, dacă acesta este un ciclu hamiltonian}
- 3: **dacă** $k \neq n + 1$ **atunci**
- 4: **pentru** $i \leftarrow 1, k - 1$ **execută**
- 5: **dacă** $x[k] = x[i]$ **atunci**
- 6: **return** false
- 7: **sfârșit dacă**
- 8: **sfârșit pentru**
- 9: **altfel**
- 10: $c \leftarrow a[x[k - 1], x[k]] + a[x[k], x[1]]$
- 11: **dacă** $x[k] \neq x[1]$ **atunci**
- 12: **return** false
- 13: **sfârșit dacă**
- 14: **pentru** $i \leftarrow 2, k - 1$ **execută**
- 15: **dacă** $x[k] = x[i]$ **atunci**
- 16: **return** false
- 17: **sfârșit dacă**
- 18: $c \leftarrow c + a[x[i - 1], x[i]]$
- 19: **sfârșit pentru**

```

20: sfârșit dacă
21: dacă  $k > 1$  atunci
22:     dacă  $a[x[k-1], x[k]] = 0$  atunci
23:         return false
24:     sfârșit dacă
25: sfârșit dacă
26: return true

```

procedura COMPAR($a, n, x, c, cmin, o$)

```

1: {verifică dacă costul ciclului hamiltonian  $x[1], \dots, x[n+1]$  este minim}
2: {în caz afirmativ se reține ca fiind soluție a problemei}
3: dacă  $c < cmin$  atunci
4:      $cmin \leftarrow c$ 
5:      $o \leftarrow x$ 
6: sfârșit dacă

```

procedura BACK($a, n, x, k, cmin, o$)

```

1: {generează recursiv toate soluțiile problemei și reține soluția optimă}
2: pentru  $i \leftarrow 1, n$  execută
3:      $x[k] \leftarrow i$ 
4:     dacă POSIBIL( $a, n, k, x, c$ ) atunci
5:         dacă  $k = n + 1$  atunci
6:             COMPAR( $a, n, x, c, cmin, o$ )
7:         altfel
8:             BACK( $a, n, x, k + 1, cmin, o$ )
9:         sfârșit dacă
10:    sfârșit dacă
11: sfârșit pentru

```

programul PRINCIPAL

```

1: {generează ciclul hamiltonian de cost total minim}
2:  $cmin \leftarrow 10^6$  {citește datele}
3:  $x[1] \leftarrow 1$  {alege un oraș de plecare arbitrar}
4: {începe generarea soluției cu al doilea element}
5: BACK( $a, n, x, 2, cmin, o$ )
6: {vectorul  $o[1], \dots, o[n+1]$  reprezintă ordinea în care vor fi vizitate orașele}
7: dacă  $cmin = 10^6$  atunci
8:     scrie problema nu are soluție
9: altfel
10:    scrie soluția problemei: vectorul  $o[1], \dots, o[n+1]$ 
11: sfârșit dacă

```

Varianta 2

Algoritmul de rezolvare se bazează pe metoda *Greedy* și folosește ca metodă de rezolvare o *euristică generală*, cunoscută sub numele de *euristica celui mai apropiat vecin*. Euristica aleasă este minimizarea sumei distanțelor parcurse la orice moment.

Algoritmul este următorul: ciclul hamiltonian se va construi pas cu pas, adăugând la fiecare etapă cea mai scurtă muchie (nealeasă încă) cu următoarele proprietăți:

- este incidentă muchiei alese anterior (dacă muchia aleasă nu este prima)
- nu închide ciclu cu muchiile deja selectate (cu excepția ultimei muchii selectate care va trebui să închidă ciclul hamiltonian)

Observații

Metoda folosită spre rezolvare:

- este o metodă euristică, fiind o tehnică de IA;
- este o metodă fără revenire, un element o dată ales în soluție nu va putea fi eliminat;
- există situații în care nu se obține soluție sau nu se obține soluția optimă, dar în acest ultim caz se va găsi o aproximație “bună” a soluției optime;
- este mult mai eficientă decât prima metodă, deoarece se execută într-un timp proporțional cu n^2 .

Se poate arăta că acest algoritm nu furnizează soluția optimă. Acest lucru se întâmplă din cauza faptului că, alegerea la fiecare etapă a celei mai scurte muchii nu garantează obținerea în final a ciclului cel mai scurt.

Să presupunem că la un moment dat avem de ales între 2 muchii $[a,b]$ și $[a,c]$, $l([a,b]) < l([a,c])$ (cu $l(m)$ am notat lungimea muchiei m). Conform algoritmului vom alege muchia $[a,b]$. Presupunem că alegând muchia $[a,b]$, cea mai bună alegere în continuare va fi muchia $[b,c]$, apoi $[c,d]$ și în final $[d,a]$, pe când alegând muchia $[a,c]$ se vor alege în continuare muchiile $[c,b]$, $[b,d]$ și în final $[d,a]$. Să presupunem că $l([a,b]) + l([c,d]) > l([a,c]) + l([b,d])$, ceea ce arată că alegerea muchiei $[a,c]$ ar fi fost mai potrivită decât alegerea muchiei $[a,b]$, chiar dacă $l([a,c]) > l([a,b])$.

Un exemplu concret în care nu se obține soluția optimă ar fi următorul: se dau 5 orașe, iar matricea distanțelor dintre orașe este

$$a = \begin{pmatrix} 0 & 2 & 0 & 1 & 3 \\ 2 & 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 4 & 2 \\ 1 & 2 & 4 & 0 & 0 \\ 3 & 3 & 2 & 0 & 0 \end{pmatrix}$$

Aplicând algoritmul euristic propus anterior vom obține ciclul hamiltonian (1,2,4,3,5,1) având lungimea 13, or acesta nu este minim, deoarece se poate obține ciclul (1,2,5,3,4,1) de lungime 12.

procedura HAMILTON(a, n, o, c)

```

1: {generează ciclul hamiltonian  $o[1], \dots, o[n+1]$  având costul total  $c$ }
2:  $o[1] \leftarrow 1$  {selectează un oraș de start arbitrar}
3:  $k \leftarrow 1$  {numărul elementelor generate în soluție}
4:  $c \leftarrow 0$  {costul traseului}
5:  $m \leftarrow \{1\}$  {mulțimea orașelor selectate deja}
6:  $gasit \leftarrow false$  {s-a găsit sau nu ciclu hamiltonian}
7: cât timp  $\neg gasit \wedge k <> n$  execută {până când toate orașele au fost vizitate}
8:   {analizează orașele nevizitate încă și selectează-l pe cel mai apropiat}
9:    $min \leftarrow 10^6$ 
10:  pentru  $i \leftarrow 1, n$  execută
11:    dacă  $\neg(i \in m) \wedge (a[o[k], i] \neq 0) \wedge (a[o[k], i] < min)$  atunci
12:       $min \leftarrow a[o[k], i]$ 
13:       $v \leftarrow i$  {orașul cel mai apropiat}
14:    sfârșit dacă
15:  sfârșit pentru
16:  dacă  $min = 10^6$  atunci
17:    scrie nu există soluție
18:    return
19:  altfel
20:     $k \leftarrow k + 1$ 
21:     $o[k] \leftarrow v$ 
22:  sfârșit dacă
23: sfârșit cât timp
24: dacă  $a[o[n], o[1]] = 0$  atunci {nu se închide ciclul}
25:   scrie nu există soluție
26: altfel
27:   scrie soluția problemei: șirul  $o[1], \dots, o[n], o[1]$ 
28: sfârșit dacă

```

2.7.2 (Problema celor 8 dame). Se cere să se dispună pe o tablă de șah 8 dame, astfel încât acestea să nu se atace reciproc.

Rezolvare

Problema enunțată este o problemă de satisfacere a restricțiilor. Problemele de satisfacere a restricțiilor se rezolvă, în general, prin *backtracking*.

După o scurtă analiză a problemei, se constată că o primă cerință (care rezultă din enunț) este ca pe o linie să se dispună o singură damă, altfel spus cele 8 dame vor fi așezate fiecare pe câte o linie. Ceea ce înseamnă că pe fiecare linie i dama va trebui așezată pe o coloană $x(i)$ (vectorul “necunoscută” a problemei). O a doua condiție pentru ca damele de pe două linii oarecare i și j să nu se atace este ca $|x(i) - x(j)| \neq |i - j|$ (condiția va trebui să fie satisfăcută pentru orice $i \neq j$).

În consecință, problema damelor poate fi formalizată ca o problemă de satisfacere a restricțiilor astfel:

- în rezolvare intervin 8 variabile x_1, x_2, \dots, x_8 corespunzătoare pozițiilor damelor pe fiecare linie
- domeniul fiecărei variabile este $\{1, 2, \dots, 8\}$
- soluție este o combinație a acestor variabile
- restricțiile problemei (pentru ca damele să nu se atace reciproc) vor fi reprezentate sub forma a două predicate:

$$x(i) \neq x(j) \wedge |x(i) - x(j)| \neq |i - j|$$

Rezultate

$x(8)$ – vectorul reprezentând coloanele pe care vor fi plasate damele pe cele 8 linii ale tablei de șah

Algoritmul de rezolvare se bazează după cum am spus anterior pe metoda *backtracking* și constă în generarea tuturor valorilor posibile ale elementelor vectorului x (x_1, x_2, \dots, x_8) satisfăcând restricțiile de mai sus.

procedura POSIBIL(k, x)

- 1: {verifică dacă dama de pe linia k poate fi dispusă pe coloana $x[k]$,}
- 2: {adică dacă sunt satisfăcute restricțiile impuse de problemă}
- 3: **pentru** $i \leftarrow 1, k - 1$ **execută**
- 4: **dacă** $x[k] = x[i]$ **atunci**
- 5: **return** false
- 6: **sfârșit dacă**
- 7: **sfârșit pentru**
- 8: **pentru** $i \leftarrow 1, k - 1$ **execută**
- 9: **dacă** $k - i = |x[k] - x[i]|$ **atunci**
- 10: **return** false
- 11: **sfârșit dacă**

12: **sfârșit pentru**

13: **return** true

procedura SCRIE(x)

1: {scrie o soluție a problemei și incrementează numărul soluțiilor găsite}

2: $sol \leftarrow sol + 1$

3: **scrie** Soluția sol este:

4: **pentru** $i \leftarrow 1, 8$ **execută**

5: **scrie** pe linia i dama se va dispune pe coloana $x[i]$

6: **sfârșit pentru**

procedura BACK(x, k)

1: {generează recursiv toate soluțiile problemei}

2: **pentru** $i \leftarrow 1, 8$ **execută**

3: $x[k] \leftarrow i$

4: **dacă** POSIBIL(k, x) **atunci**

5: **dacă** $k = 8$ **atunci**

6: SCRIE(x)

7: **altfel**

8: BACK($x, k + 1$)

9: **sfârșit dacă**

10: **sfârșit dacă**

11: **sfârșit pentru**

programul PRINCIPAL

1: {generează toate posibilitățile de dispunere a damelor}

2: {dacă nu va găsi soluție, scrie mesaj}

3: $sol \leftarrow 0$ {inițializează numărul soluțiilor cu 0}

4: {începe generarea soluției cu primul element}

5: BACK($x, 2$)

Observație: Se poate generaliza, pentru rezolvarea “problemei celor n dame”.

2.7.3. Analizați și descrieți un algoritm de rezolvare pentru următoarea problemă de satisfacere a restricțiilor: “Problema colorării hărților”.

O hartă este împărțită în n domenii (zone). Fiecare domeniu trebuie colorat cu o culoare (numărul de culori cu care poate fi colorat fiecare domeniu este finit). Se cere să se furnizeze toate modalitățile de colorare a hărții astfel încât două domenii învecinate ale hărții să nu fie colorate cu aceeași culoare (se consideră învecinate domeniile situate pe direcțiile N, S, E, V, N-E, N-V, S-E, S-V).

2.7.4. Analizați următoarele probleme prezentate în partea teoretică și propuneți variante de rezolvare (algoritmi):

- Gălețile cu apă;
- 8 – puzzle;
- Misionarii și canibalii.

2.7.5 (Problema turnurilor din Hanoi). Analizați următoarea problemă și descrieți algoritmi de rezolvare.

Se dau trei tije A , B , C . Pe prima tijă sunt dispuse n discuri, în ordine descrescătoare a diametrelor (se cunoaște faptul că diametrul discului i este i). Se cere numărul minim de operații necesare pentru a muta toate discurile de pe tija A pe tija B , folosind ca tijă intermediară tija C , știind că, la un moment dat:

- nu se poate muta decât un singur disc;
- nu se poate muta un disc de diametru mai mare peste un disc de diametru mai mic.

Rezolvare

Este un exemplu de problemă decompozabilă (problema se poate descompune în subprobleme independente, care se rezolvă separat și apoi se combină soluțiile pentru a obține soluția întregii probleme). Tehnica de programare care rezolvă probleme în acest fel se numește *Divide et Impera*.

În cazul problemei Turnurilor din Hanoi, algoritmul este următorul: turnulețul de n discuri de pe tija A se descompune în două, și anume: discul n și restul, adică turnulețul de $n - 1$ discuri. Ca urmare, mutările necesare vor fi următoarele:

- dacă $n > 1$, se mută turnulețul de $n - 1$ discuri de pe tija A pe tija C , folosind ca tijă intermediară tija B (mutarea acestuia se va face ca și a celui cu n discuri, adică prin apelul recursiv al procedurii de mutare);
- se mută discul n de pe tija A pe tija B ;
- dacă $n > 1$ se mută turnulețul de $n - 1$ discuri de pe tija C pe tija B , folosind ca tijă intermediară tija A .

Se poate arăta că această ordine de mutări furnizează soluția minimă, adică numărul minim de operații.

Se va folosi procedura recursivă $hanoi(n, x, y, z)$ având următoarea semnificație: descrie succesiunea operațiilor necesare pentru a muta n discuri de pe tija x pe tija y , având ca tijă intermediară tija z .

procedura $HANOI(n, x, y, z)$

- 1: **dacă** $n = 1$ **atunci**
- 2: **scrie** se muta discul 1 de pe tija x pe tija y

3: **altfel**

4: { se mută $n - 1$ discuri de pe tija x pe tija z folosind tija intermediară z }

5: $\text{HANOI}((n - 1, x, z, y)$

6: { se mută discul n de pe tija x pe tija y }

7: **scrie** se muta discul n de pe tija x pe tija y

8: { se mută $n - 1$ discuri de pe tija z pe tija y folosind tija intermediară x }

9: $\text{HANOI}(n - 1, z, y, x)$

10: **sfârșit dacă**

programul PRINCIPAL

1: { generează succesiunea de mutări pentru obținerea soluției problemei }

2: { citește valoarea lui n }

3: { apelul inițial al procedurii recursive }

4: $\text{HANOI}(n, 'A', 'B', 'C')$

Observație: Propuneți o altă metodă de rezolvare a problemei.

2.7.6 (Problema “decupării”). Analizați următoarea problemă și descrieți algoritmi de rezolvare.

Se dă o bucată de tablă dreptunghiulară, de lungime L și lățime l , conținând găuri. Se cere să se decupeze din bucata de tablă un dreptunghi de arie maximă ce nu conține găuri.

Indicație: Este un alt exemplu (ca și cel anterior) de problemă decompozabilă. Se poate folosi pentru rezolvare tehnica *Divide et Impera*.

2.7.7. Pentru următoarele probleme descrieți o funcție euristică bună.

- Lumea blocurilor;
- Demonstrarea teoremelor;
- Misionarii și canibalii.

2.7.8. Să se scrie un algoritm pentru realizarea unei căutări *Depth-First* pe un graf. Dându-se un graf orientat G , prin matricea sa de adiacență $A(n,n)$, și un vârf x se cere să se tipărească vârfurile grafului în ordinea în care vor fi vizitate, într-o căutare de tip *Depth-First* (în adâncime).

Rezolvare

Specificarea problemei

Date

n – numărul de vârfuri ale grafului;

$a(n, n)$ – matricea reprezentând legăturile dintre vârfuri (matricea de adiacență a grafului), cu semnificația: $a[i, j] = 1$, dacă între vârfurile i și j există o legătură directă, respectiv $a[i, j] = 0$, dacă $i=j$ sau dacă între vârfurile i și j nu există legătură directă.

Rezultate

Se vor tipări vârfurile în ordinea în care sunt vizitate într-o căutare în adâncime.

Procedura de căutare Depth-First este descrisă recursiv și se bazează pe algoritmul descris în partea teoretică. Fiind vorba de o căutare pe un graf, pentru a evita vizitarea unui vârf de mai multe ori, vom introduce variabila de tip vector viz , având următoarea semnificație: $viz[i] = true$, dacă vârful i a fost deja vizitat, respectiv false în caz contrar.

Exemplu: În graful orientat G , având 6 vârfuri și matricea de adiacență

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

ordinea în care vor fi vizitate vârfurile grafului într-o căutare *Depth-First* este 1, 2, 6, 5, 4, 3.

procedura DEPTHFIRST(a, vf)

- 1: { parcurge în adâncime începând cu vârful vf }
- 2: { tipărește vârful vf și marchează ca fiind vizitat }
- 3: $viz[vf] \leftarrow true$
- 4: **scrie** vf
- 5: **pentru** $i \leftarrow 1, n$ **execută**
- 6: { dacă vârful i este adiacent cu vf și nu a fost încă vizitat, }
- 7: { se realizează o parcurgere Depth-First începând cu vârful i }
- 8: **dacă** $(a[vf, i] \neq 0) \wedge (\neg viz[i])$ **atunci**
- 9: DEPTHFIRST(a, i)
- 10: **sfârșit dacă**
- 11: **sfârșit pentru**

programul PRINCIPAL

- 1: { realizează parcurgerea Depth-First începând cu un vârf x dat }
- 2: { citește datele }
- 3: { marchează toate vârfurile grafului ca fiind nevizitate }
- 4: **pentru** $i \leftarrow 1, n$ **execută**
- 5: $viz[i] \leftarrow false$
- 6: **sfârșit pentru**

7: {apelează procedura recursivă de parcurgere, începând cu vârful x }

8: DEPTHFIRST(a, x)

2.7.9. Dați un exemplu de problemă în care căutarea *Breadth-First* este mai bună ca și căutarea *Depth-First*. Dați un exemplu de problemă în care căutarea *Depth-First* este mai bună ca și căutarea *Breadth-First*.

2.7.10. Scrieți un program pentru realizarea unei căutări *Breadth-First* pe un graf.

2.7.11. Construiți un algoritm pentru rezolvarea problemelor din lumea blocurilor.

Capitolul 3

Tehnici de căutare euristică

Am văzut că multe din problemele care sunt de inteligență artificială sunt prea complexe pentru a fi soluționate prin tehnici directe. În acest capitol se prezintă un cadru pentru descrierea metodelor de căutare și sunt discutate câteva tehnici de scop general, care sunt toate variațiuni de căutare euristică. Acestea pot fi descrise independent de problemă sau de domeniul problemei. Când sunt aplicate, eficiența lor depinde de modul în care exploatăm cunoștințele specifice domeniului, deoarece doar prin ele însele, acestea nu pot depăși explozia combinatorială la care sunt vulnerabile tehnicile de căutare. Din acest motiv, aceste tehnici se numesc **metode slabe**.

Am discutat deja tehnicile:

- Căutare Depth-First;
- Căutare Breadth-First.

În continuare vom prezenta următoarele tehnici:

- Generează și Testează;
- Hill Climbing;
- Căutare Best-First;
- Reducerea Problemei;
- Satisfacerea Restricțiilor (Constraints Satisfaction);
- Analiza Means-Ends.

3.1. Generează și testează

Este cea mai simplă strategie dintre cele pe care le vom discuta. Constă din următorii pași:

Algoritm: Generează și testează

1. Generează o soluție posibilă. Pentru unele probleme aceasta înseamnă generarea unui punct din spațiul stărilor. Pentru altele, înseamnă generarea unui drum de la o stare inițială.
2. Testează pentru a vedea dacă este realmente o soluție, prin compararea punctului ales sau capătul drumului ales cu mulțimea de stări finale acceptabile.
3. Dacă s-a găsit o soluție, STOP. Altfel, mergi la pasul 1.

Dacă generarea soluțiilor posibile se face sistematic, atunci procedura va găsi la un moment dat o soluție, dacă aceasta există. Este adevărat că dacă spațiul problemei este mare, “la un moment dat” ar putea însemna după foarte mult timp. Această tehnică se mai cunoaște sub forma următoare: *Dacă un număr suficient de maimuțe sunt plasate în fața unei mulțimi de mașini de scris și sunt lăsate singure un timp suficient de lung, după un timp maimuțele vor produce toate lucrările lui Shakespeare.*

Este o procedură de căutare de tip Depth-First, deoarece trebuie generate soluțiile înainte de a putea fi testate. Iată trei moduri de a face căutarea:

1. Căutare exhaustivă în spațiul de probleme (forma cea mai sistematică);
2. Căutare aleatoare (fără garanții de succes);
3. Căutare sistematică cu ignorarea unor drumuri improbabile (folosind funcții euristice).

Modalitatea imediată de implementare este folosind un arbore. Dacă este probabil ca stări intermediare să apară de mai multe ori, ar putea fi util să se modifice procedura pentru a traversa un graf, nu un arbore.

Exemplul 1: Se dau patru cuburi cu fețele colorate într-una din patru culori. Se cere să se aranjeze cuburile pe un rând astfel încât toate cele patru fețe ale rândului arată câte o față de cub de fiecare culoare.

O soluție – căutarea exhaustivă. A doua soluție – dacă observăm, de exemplu, că avem mai multe fețe roșii decât din celelalte culori, atunci este o idee bună să utilizăm cât de puține din acestea ca fețe externe.

Exemplul 2: Sistemul DENDRAL care inferează structura componentelor organice folosind specograme de masă și rezonanță nucleară magnetică. Folosește o tehnică numită **planifică-generează-testează** în care un proces de planificare care folosește tehnici de satisfacere a limitărilor crează o listă de structuri recomandate și contraindicate.

Observație: Planificarea produce în general soluții slabe, deoarece nu există feed-back de la lume. Dar dacă planificarea se utilizează pentru a produce părți de soluție, acestea vor fi exploatate prin procesul de generare și testare.

3.2. Hill Climbing

Este o variantă a algoritmului **Generează și Testează** în care se utilizează feed-back de la procedura de testare pentru a ajuta generatorul să decidă în care direcție să se deplaseze în spațiul de căutare. Într-o procedură **Generează și Testează** pură, funcția de testare răspunde cu **da** sau **nu**. Dar dacă funcția se îmbunătățește cu o funcție euristică (numită și funcție obiectiv), care va trebui să spună și cât de aproape de soluție suntem, atunci procedura de generare poate exploata această informație

3.2.1. Hill Climbing simplu

Cea mai simplă modalitate de a implementa hill climbing este următoarea:

Algoritm: Hill climbing simplu

1. Evaluează starea inițială. Dacă este și stare finală, STOP(starea inițială). Altfel, continuă cu starea inițială ca stare curentă.
2. Ciclează până când o soluție se găsește sau până când nu mai există operatori de aplicat stării curente:
 - (a) Selectează un operator care încă nu a fost aplicat stării curente și aplică-l pentru a produce o nouă stare.
 - (b) Evaluează starea nouă.
 - i. Dacă este o stare finală atunci STOP(stare nouă).
 - ii. Dacă nu este stare finală, dar este mai bună ca starea curentă, atunci fă-o stare curentă.
 - iii. Dacă nu este mai bună ca starea curentă, continuă ciclu.

Diferența față de algoritmul Generează și Testează este utilizarea unei funcții de evaluare ca o modalitate de a introduce cunoștințe în procesul de control.

Observație: Avem nevoie de o definiție pentru “stare mai bună”. Uneori înseamnă o valoare mai mare a funcției euristice, iar alteori înseamnă o valoare mai mică.

Pentru o verificare a algoritmului să vedem din nou problema celor patru cuburi. **Funcția euristică** poate fi suma culorilor distincte de pe fiecare față a rândului de cuburi. O soluție are suma egală cu 16. Apoi avem nevoie de un **set de reguli** pentru a transforma o configurație în alta: regula următoare: “ia un cub și rotește-l cu 90 grade într-o direcție”. Apoi avem nevoie de o **stare inițială**. Aceasta se poate genera aleator sau folosind euristica descrisă mai înainte.

3.2.2. Hill Climbing de pantă maximă (steepest-ascent)

O variație utilă a algoritmului Hill Climbing simplu consideră toate mutările din starea curentă și o selectează pe cea mai bună ca stare viitoare. Această metodă se numește **Hill Climbing de pantă maximă (steepest-ascent)** sau **căutare de gradient**. Notați contrastul cu metoda de bază în care se selectează prima stare mai bună ca starea curentă.

Algoritm: Hill Climbing de pantă maximă

1. Evaluează starea inițială. Dacă este și stare finală, STOP(starea inițială). Altfel, continuă cu starea inițială ca stare curentă.
2. Ciclează până când o soluție se găsește sau până când o iterație completă nu produce o schimbare în starea curentă:
 - (a) Fie SUCC o stare astfel încât orice succesori al stării curente este mai bun decât SUCC.
 - (b) Pentru fiecare operator aplicabil stării curente execută:
 - i. Aplică operatorul și generează o stare nouă.
 - ii. Evaluează starea nouă. Dacă este stare finală atunci STOP(stare nouă). Dacă nu, compară cu SUCC. Dacă este mai bună, setează $SUCC := \text{stare nouă}$. Dacă nu este mai bună nu modifica pe SUCC.
 - (c) Dacă SUCC este mai bun ca starea curentă, atunci setează stare curentă $:= SUCC$.

Observație: Există un echilibru între timpul care se cere

- pentru a selecta o mutare (mai lung la Hill Climbing de pantă maximă);
- pentru a atinge o soluție (mai lung la Hill Climbing simplu).

Observație: Ambele tehnici pot eșua în găsirea unei soluții. Acestea se pot opri nu prin găsirea unei stări soluție și prin găsirea unei stări de la care nu se poate avansa cu stări mai bune. Aceasta se va întâmpla dacă programul a atins una din următoarele stări:

Maximul local este o stare mai bună decât toți vecinii, dar este mai proastă decât o altă stare, situată la distanță. Într-un maxim local toate mutările par să ducă la stări mai slabe. Deseori maximele locale apar în vecinătatea unei soluții.

Platoul este o zonă plată în care un număr mare de stări vecine au aceeași valoare. Pe un platou direcția în care ne vom deplasa nu se poate determina numai prin comparații locale.

Creasta este un tip special de maxim local. Este o zonă în spațiul de căutare care este mai mare decât stările vecine, dar care are o pantă ascendentă care nu poate fi însă urmată prin mutări individuale.

Există câteva metode prin care putem aborda aceste probleme, chiar dacă aceste metode nu sunt garantate:

- Execută backtracking la un nod anterior și încearcă o deplasare într-o altă direcție; este o soluție rezonabilă dacă la acel nod s-a întâmplat să existe o alternativă (aproape la fel de) promițătoare. Pentru implementarea acestei strategii trebuie memorate adresele drumurilor care au fost **aproape selectate** și, dacă un drum selectat a dus la un blocaj, se va reveni la un drum aproape selectat. Aceasta este o metodă bună pentru **maxime locale**.
- Execută un salt mare într-o direcție în încercarea de a ajunge într-o altă secțiune a spațiului de căutare. Dacă singurele reguli aplicabile descriu pași mici, atunci aplică-le de mai multe ori în aceeași direcție. Aceasta este o metodă bună pentru **platouri**.
- Aplică două sau mai multe reguli înainte de a face testul. Aceasta corespunde cu deplasarea în mai multe direcții în același moment. Aceasta este o metodă bună pentru **creste**.

Chiar cu aceste tehnici ajutătoare, Hill climbing nu este intotdeauna foarte eficace. În mod particular nu este potrivită cu acele probleme în care valoarea funcției euristice scade brusc imediat ce ne depărtăm de soluție. Este o metodă locală, prin aceasta înțelegând că pentru a decide care este următoarea mutare se uită doar la efectele imediate ale mutărilor. Partajează avantajul altor metode locale (**nearest neighbour**) de a fi mai puțin exploziv combinatoriale decât metodele globale.

Pe de altă parte, deși este adevărat că hill-climbing se uită doar o mutare în față, această examinare ar putea să exploateze o anumită cantitate de informație globală, dacă această informație globală este inserată în funcția euristică.

Fie de exemplu problema din lumea cuburilor din Figura 3.1, și să presupunem că avem la dispoziție aceiași operatori despre care am discutat înainte.

Fie următoarea **funcție euristică locală**: **Adună câte un punct pentru fiecare bloc care stă pe obiectul corect (bloc sau masă) și scade câte un punct pentru fiecare bloc care stă pe obiectul greșit.**

Valoarea stării finale este 8, valoarea stării inițiale este $6 - 2 = 4$ ($C + D + E + F + G + H - A - B$). Dinspre starea inițială există o singură mutare posibilă:

A pe masă, cu un scor de $7 - 1 = 6$

De aici sunt disponibile trei mutări, indicate în Figura 3.2:

- A înapoi pe H, cu un scor de $6 - 2(A, B) = 4$
- H peste A, cu un scor de $6 - 2(H, B) = 4$
- H pe masă, cu un scor de $6 - 2(H, B) = 4$

Inițial	Final
A	H
H	G
G	F
F	E
E	D
D	C
C	B
B	A

Figura 3.1: O problemă Hill-Climbing

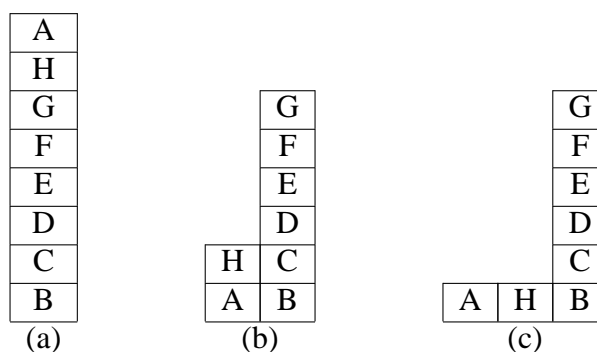


Figura 3.2: Trei mutări posibile

Procedura se oprește aici cu un maxim local, deoarece toate cele trei mutări produc stări mai proaste. Maximul local obținut nu este maximul global (soluția problemei). Dificultatea este că prin examinarea structurilor de suport doar la nivel local, starea curentă pare mai bună deoarece sunt mai multe blocuri care stau pe obiectele corecte. Pentru a rezolva această problemă este necesar să dezasamblăm o structură locală bună (stiva de la B la H) deoarece este în contextul global greșit.

Am putea să blamăm Hill-climbing pentru eșec, dar de asemenea putem blama funcția euristică și putem încerca să o modificăm. Să considerăm următoarea funcție euristică globală: **Pentru fiecare bloc care stă pe structura de suport corectă (adică structura completă, de la masă în sus) adună câte un punct pentru fiecare bloc din structură. Pentru fiecare bloc care stă pe structura de suport greșită scade câte un punct pentru fiecare bloc din structură.**

Folosind această funcție euristică, starea finală are scorul **28**, iar starea inițială are scorul **-28**. Prima mutare conduce la o stare cu scorul **-21**, iar cele trei mutări posibile din acea stare conduc la stări cu scorurile (a) **-28**, (b) **-16**, (c) **-15**. De această dată algoritmul Hill-Climbing de pantă maximă va selecta mutarea (c), care este cea corectă. Această funcție euristică prinde două aspecte esențiale:

- structurile incorecte sunt negative și trebuie demontate;

- structurile corecte sunt pozitive și trebuie construite (sau construit peste ele).

Ca urmare, aceeași procedură care a dus la un rezultat greșit cu funcția anterioară, acum va duce la rezultatul corect.

Observații:

1. Nu este întotdeauna posibil să contruim funcții euristice atât de bune; fie nu avem cunoștințele necesare, fie acestea nu sunt utilizabile din punct de vedere computațional.
2. Ca exemplu extrem să presupunem că o funcție euristică face propria sa căutare a soluției, în felul acesta putând apoi să calculeze exact costul identificării soluției plecând de la starea curentă. Dar în felul acesta procedura de căutare locală este convertită într-o procedură globală, și avantajele căutării locale sunt pierdute.
3. Metoda este utilă atunci când este combinată cu alte metode care sunt capabile să îi dea startul în acea zonă a spațiului de stări care este o vecinătate generală a soluției.

3.2.3. Călire simulată (Simulated Annealing)

Călire simulată este o variație a algoritmului Hill climbing în care, la începutul procesului se pot face unele deplasări defavorabile (în josul dealului). Ideea este să facem o explorare suficient de mare a spațiului de căutare cât mai repede posibil, astfel încât soluția finală să fie relativ insensibilă la starea de start. Această operație ar trebui să micșoreze riscul de a fi prinși într-un maxim local, platou sau creastă.

Pentru a fi compatibili cu terminologia standard din discuțiile despre călire simulată, în această secțiune vom vorbi despre **funcție obiectiv** (în loc de funcție euristică), respectiv despre **minimizarea** (în loc de maximizarea) funcției obiectiv.

Procesul computațional de călire simulată este modelat după procesul fizic de **călire**, în care substanțe fizice, cum ar fi metalul, sunt topite (urcate la un nivel de energie înalt) și apoi răcite treptat până când se atinge o stare solidă cu anumite proprietăți. Scopul procesului este de a produce o stare finală de energie minimală. Astfel procesul este unul de valley descending în care funcția obiectiv este nivelul de energie.

De obicei substanțele fizice evoluează de la o stare de energie mare la una de energie mai scăzută, și deci procesul de valley descending apare natural. Totuși, există o probabilitate să apară o tranziție la o stare de energie mai înaltă. Această probabilitate este dată de funcția

$$p = e^{-\Delta E/kT}$$

unde ΔE este schimbarea pozitivă în nivelul de energie, T este temperatura și k este constanta lui Boltzman (și descrie corespondența între unități de temperatură și unități de energie). Astfel, în procesul de călire fizică, probabilitatea unei mișcări în sus mai mari este mai mică

decât probabilitatea unei mișcări mai mici. În plus, probabilitatea ca o mișcare în sus să aibă loc scade odată cu scăderea temperaturii. Ca urmare astfel de mișcări sunt posibile la începutul procesului, când temperatura este mare, și devin din ce în ce mai improbabile la sfârșit, pe măsură ce temperatura scade.

Viteza cu care sistemul se răcește se numește **schemă (orar) de călire** (annealing schedule). Dacă răcirea este prea rapidă, se formează regiuni stabile de energie înaltă, adică apare un minim local și nu global. Dacă, însă, se folosește o schemă mai înceată, este mai probabil să se formeze o structură cristalină, care corespunde unui minim global. Schema de călire optimă pentru fiecare problemă se descoperă, de obicei, în mod empiric.

Toată această teorie a călirii fizice se poate utiliza pentru definirea unui proces analog, de călire simulată. Aici, ΔE va reprezenta schimbarea în valoarea funcției obiectiv. Semnificația pentru kT este mai puțin evidentă. Vom încorpora pe k în T și vom selecta pentru T valori care vor influența comportarea algoritmului. Astfel, ajungem la formula de probabilitate revizuită:

$$p' = e^{-\Delta E/T}$$

Va trebui să alegem o schemă de valori pentru T (pe care o vom numi, totuși, temperatură). **Algoritmul de călire simulată** este ușor diferit de procedura Hill-Climbing simplă. Cele trei diferențe sunt următoarele:

- Trebuie memorată și actualizată schema de călire;
- Se pot accepta și deplasări la stări mai slabe;
- Este o idee bună să memorăm, alături de starea curentă, cea mai bună stare găsită până acum. Astfel, dacă starea finală este mai proastă decât acea stare intermediară (din cauza ghinionului din acceptarea unor deplasări mai proaste), starea intermediară este disponibilă.

Algoritm: Călire simulată

1. Evaluează starea inițială. Dacă este o stare finală, atunci RETURN(starea inițială). Altfel, continuă cu starea inițială ca stare curentă.
2. Inițializează BEST cu starea curentă.
3. Inițializează T conform schemei de călire.
4. Ciclează până când se găsește o soluție sau până când nu există operatori de aplicat stării curente.
 - (a) Selectează un operator care încă nu a fost aplicat stării curente și aplică-l pentru a produce o stare nouă.

(b) Evaluează starea nouă. Calculează

$$\Delta E = (\text{valoarea stării curente}) - (\text{valoarea stării noi})$$

- i. Dacă starea nouă este stare finală, atunci RETURN(starea nouă).
- ii. Dacă nu e stare finală, dar este mai bună decât starea curentă, atunci fă-o stare curentă și setează BEST la această stare.
- iii. Dacă nu este mai bună decât starea curentă, atunci fă-o stare curentă cu probabilitatea p' definită mai sus. Acest pas se implementează de obicei prin apelarea unui generator de numere aleatoare pentru a produce un număr în $[0,1]$. Dacă numărul este mai mic decât p' , atunci operația este acceptată. Altfel, nu execută nimic.

(c) Actualizează T conform schemei de călire.

5. RETURN(BEST).

Pentru a implementa acest algoritm, avem nevoie să selectăm o schemă de călire. Aceasta are trei componente (opțional, patru):

1. Valoarea inițială care se va folosi pentru T ;
2. Criteriul utilizat pentru a decide când trebuie redusă temperatura sistemului;
3. Mărimea cu care se va reduce temperatura la fiecare modificare;
4. (Opțional) Decizia relativă la momentul de oprire.

Procedeu se folosește deseori pentru probleme în care numărul de mutări posibile este foarte mare (vezi problema comis-voiajorului). În astfel de situații nu ar fi rezonabil să încercăm toate soluțiile, ci ar fi util să exploatăm criterii care implică numărul de operații efectuate de la ultima îmbunătățire.

Experimental s-a observat că cel mai bun mod de a selecta o schemă de călire este prin încercarea mai multor alternative și prin studierea efectului asupra calității soluției și vitezei de convergență. De exemplu, să remarcăm că, pe măsură ce T se apropie de zero, probabilitatea de a accepta o deplasare la o stare mai proastă tinde la zero și tehnica devine Hill-Climbing simplu. Al doilea lucru de remarcat este că ceea ce contează în calculul probabilității de acceptare a unei mutări este raportul $\Delta E/T$. Astfel este important ca valorile lui T să fie scalate astfel încât acest raport să fie semnificativ. De exemplu, T ar putea fi inițializat la o astfel de valoare încât pentru ΔE mediu, p' să fie 0.5.

3.3. Căutare Best-First

Până acum am discutat doar două strategii de control sistematice: **Căutare Breadth-First** și **Căutare Depth-First** (cu mai multe varietăți). Aici vom discuta o metodă nouă, **Căutarea Best-First**, care este o combinație a celor două tehnici.

3.3.1. Grafuri OR

Căutarea Depth-First este bună deoarece permite generarea unei soluții fără o examinare a tuturor ramurilor arborelui de căutare. Iar căutarea breadth-first este bună deoarece nu poate fi prinsă pe drumuri moarte (blocate). O posibilitate de a combina aceste două avantaje este de a urma un singur drum la un moment dat, dar cu schimbarea drumului oricând un drum concurent devine mai interesant decât drumul curent.

La fiecare pas al procesului de căutare selectăm nodul cel mai promițător dintre cele pe care le-am generat până acum, prin aplicarea unei funcții euristice pe toate aceste noduri. Apoi utilizăm regulile aplicabile pentru expandarea nodului ales și generarea succesorilor săi. Dacă unul din succesori este soluție, ne oprim. Dacă nu, toate aceste noduri noi sunt adăugate la mulțimea de noduri generate până acum. Din nou, nodul cel mai promițător este selectat și procesul continuă. Ceea ce se întâmplă de obicei este că apare puțină căutare depth-first pe măsură ce drumul cel mai promițător este explorat. Dar, dacă o soluție nu este finalmente găsită, acel drum va începe să pară mai puțin promițător decât altul care mai înainte a fost ignorat. În acel punct, drumul care a devenit mai promițător este selectat și analizat, dar drumul vechi nu este uitat. Căutarea poate reveni la acesta oricând toate celelalte drumuri devin mai proaste și când acesta este din nou drumul cel mai promițător.

Figura 3.3 arată începutul unui proces de căutare Best-First. Valorile din paranteză indică valoarea funcției euristice pe un anumit nod și reprezintă o estimare a costului obținerii unei soluții plecând din acel nod.

Să observăm că această procedură este foarte asemănătoare cu procedura Hill-Climbing de pantă maximă, cu două excepții.

- (1) În Hill-Climbing odată cu selectarea unei mutări, toate celelalte sunt respinse și nu vor mai fi considerate niciodată. Aceasta produce comportarea liniară caracteristică algoritmului. În căutarea Best-First, odată cu selectarea unei mutări, celelalte mutări sunt păstrate pentru a putea fi considerate mai târziu dacă drumul selectat devine mai puțin promițător.

Observație: Există o variație a căutării Best-First, numită **Beam Search**, unde doar cele mai promițătoare n stări sunt memorate pentru a fi ulterior luate în considerare. Procedura este mai eficientă relativ la consumul de memorie, dar introduce posibilitatea ocolirii unei soluții prin reducerea prea rapidă a arborelui de căutare.

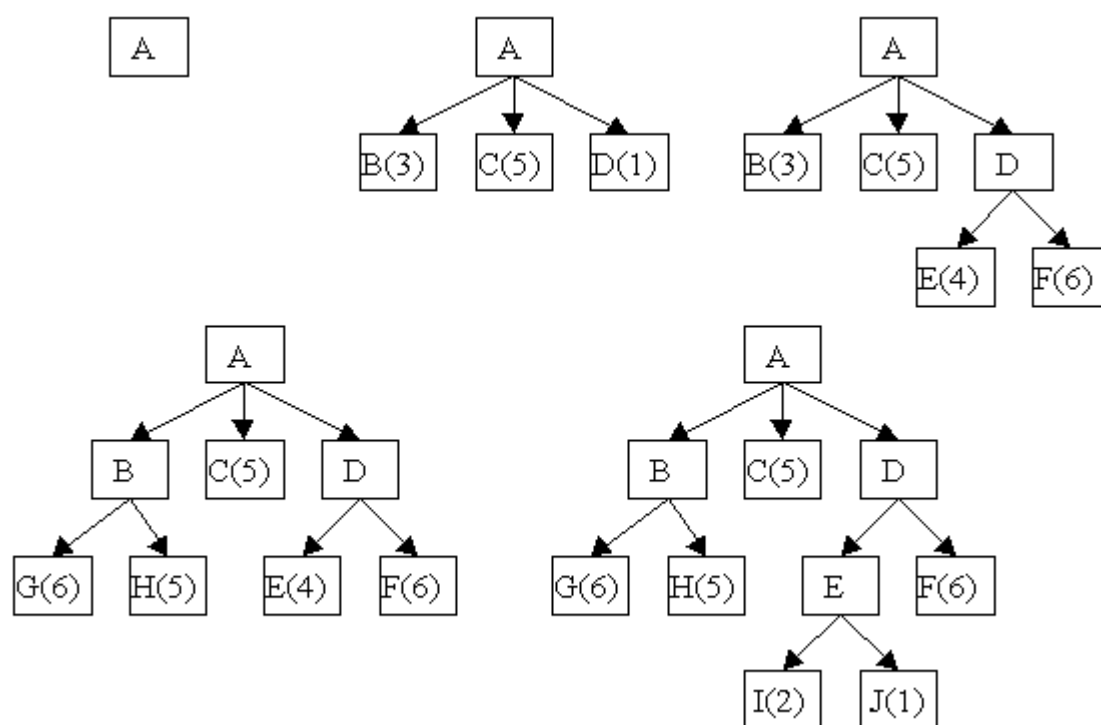


Figura 3.3: O căutare Best-First

- (2) În căutarea Best-First este selectată cea mai bună stare disponibilă, chiar dacă această stare are o valoare mai mică decât starea care tocmai a fost explorată. Aceasta contrastează cu Hill-Climbing, care se va opri dacă nu există stări succesori cu valori mai bune decât starea curentă.

Uneori este important să operăm această căutare pe un graf ordonat, nu pe un arbore, pentru a evita analizarea unui drum de mai multe ori. Fiecare nod va conține, pe lângă o informație relativă la cât de promițător este, un pointer înapoi, către cel mai bun nod care l-a generat și o listă a nodurilor care au fost generate din el. Lista succesorilor va face posibilă propagarea îmbunătățirilor în jos, înspre succesorii săi. Un astfel de graf se va numi **graf OR**, deoarece fiecare dintre ramurile sale se reprezintă un drum alternativ de rezolvare a problemei.

Pentru implementarea acestei proceduri de căutare vom avea nevoie de două liste de noduri:

OPEN – noduri care au fost generate și au atașată valoarea funcției euristice, dar nu au fost încă examinate (nu le-au fost generați succesorii); această listă este o coadă cu prioritate.

CLOSED – noduri care deja au fost examinate; dacă căutăm un arbore avem nevoie de această listă, pentru a evita generarea repetată a unor noduri.

Vom avea nevoie de o funcție euristică care estimează meritele fiecărui nod pe care îl generăm. Această funcție va permite algoritmului să caute drumul cel mai promițător. Să o notăm

cu f' , ca să indicăm că este o aproximare a funcției f care întoarce valoarea corectă, exactă, a nodului. Pentru multe aplicații această funcție o definim ca sumă a două componente, g și h' . Funcția g este o măsură (nu o estimare) a costului trecerii din starea inițială în starea curentă, iar funcția h' este o estimare a costului adițional de a obține starea finală din starea curentă. Astfel funcția combinată f' reprezintă o estimare a costului obținerii stării finale din starea inițială de-a lungul drumului pe care ne aflăm. Să mai notăm că h' trebuie să fie o estimare a costului unui nod (cu cât mai bun este nodul, cu atât mai mică este valoarea funcției), nu a calității nodului (cu cât mai bun este nodul, cu atât mai mare este valoarea funcției). De asemenea, g trebuie să fie nenegativă, pentru a evita ciclurile din grafe, întrucât drumurile cu cicluri vor părea mai bune decât cele fără cicluri.

Modul de operare a algoritmului este foarte simplu și este sumarizat în continuare.

Algoritm: Căutare Best-First

1. Incepe cu $OPEN := \{\text{starea inițială}\}$
2. Până când se găsește o stare finală, sau până când $OPEN$ este vidă, execută:
 - (a) Alege nodul cel mai bun din $OPEN$.
 - (b) Generează-i succesorii.
 - (c) Pentru fiecare succesori execută:
 - i. Dacă nu a fost generat deja, evaluează-l, adaugă-l la $OPEN$ și înregistrează-i părinții.
 - ii. Dacă a fost generat deja, schimbă-i părintele dacă acest drum nou este mai bun decât cel dinainte. În acest caz, actualizează costul obținerii acestui nod (valoarea lui g) și a oricărui nod succesori al acestuia.

Din păcate, este rar cazul în care algoritmi de traversare a grafurilor sunt suficient de simpli ca să se scrie corect. Si este și mai rar cazul în care este simplu să se garanteze corectitudinea algoritmilor. În secțiunea următoare vom descrie acest algoritm în detaliu mai mare ca un exemplu a proiectării și analizei unui program de căutare în grafe.

3.3.2. Algoritm A*

Algoritm de căutare Best-First prezentat mai sus este o simplificare a unui algoritm numit A^* . Acest algoritm folosește funcțiile f' , g și h' și listele $OPEN$ și $CLOSED$ descrise mai sus.

Algoritm: A^*

1. **Începe** cu $OPEN := \{\text{starea inițială}\}$; $g(s.i.) := 0$; calculează $h'(s.i.)$; $f'(s.i.) := h'(s.i.) + 0$; $CLOSED := \{\}$
2. **Până când se găsește un nod final**, repetă următoarea procedură:

- Dacă nu există noduri în OPEN, raportează eșec;
- Dacă există noduri în OPEN, atunci:
 - BEST := nodul din OPEN cu cel mai mic f' ;
 - Scoate BEST din OPEN; plasează-l în CLOSED;
 - Verifică dacă BEST este stare finală;
 - Dacă DA, exit și raportează soluția (BEST sau drumul la el);
 - Dacă NU, generează succesorii lui BEST, dar nu pointa BEST la aceștia; Pentru fiecare astfel de SUCCesor execută următoarele:
 - (a) Legătura părinte a lui SUCC să indice înapoi la BEST;
 - (b) $g(\text{SUCC}) := g(\text{BEST}) + \text{costul drumului de la BEST la SUCC}$;
 - (c) **Dacă SUCC apare în OPEN**, atunci
 - * Nodul din OPEN se numește OLD;
 - * Elimină SUCC și adaugă OLD la succesorii lui BEST;
 - * Dacă drumul tocmai găsit până la SUCC este mai ieftin decât drumul cel mai bun până la OLD ($g(\text{SUCC}) \leq g(\text{OLD})$), atunci
 - Părintele lui OLD trebuie resetat la BEST;
 - Actualizează $g(\text{OLD})$ și $f'(\text{OLD})$.
 - (d) **Dacă SUCC nu apare în OPEN, dar apare în CLOSED**, atunci
 - * Nodul din CLOSED se numește OLD;
 - * Adaugă OLD la lista succesurilor lui BEST;
 - * Execută pasul 2(c)iii;
 - * Dacă am găsit un drum mai bun la OLD, îmbunătățirea trebuie propagată la succesorii lui OLD, astfel:
 - OLD indică la succesorii săi; aceștia – la ai lor, ș.a.m.d.; fiecare ramură se termină fie cu un nod în OPEN, fie fără succesori;
 - fă o traversare Depth-First cu începere din OLD, cu schimbarea lui g și f' ai nodurilor traversate, cu terminare la un nod fără succesori sau un nod până la care s-a găsit deja un drum cel puțin la fel de bun;
 - * condiția “drum cel puțin la fel de bun” înseamnă următoarele:
 - dacă cel mai bun părinte al nodului este cel de unde venim, atunci continuă propagarea;
 - dacă nu, verifică dacă drumul nou actualizat este mai bun decât drumul memorat;
 - dacă este mai bun, resetează părintele și continuă propagarea;
 - dacă nu, oprește propagarea.

(e) **Dacă SUCC nu apare nici în OPEN, nici în CLOSED**, atunci

- * Aduagă SUCC în OPEN și la lista succesorilor lui BEST;
- * Calculează $f'(\text{SUCC}) = g(\text{SUCC}) + h'(\text{SUCC})$.

Iată în continuare câteva observații interesante despre acest algoritm.

Observația 1 privește rolul funcției g . Ne permite să alegem care nod să-l expandăm nu numai pe baza a cât de aproape este de starea finală, ci și a cât de aproape este de starea inițială.

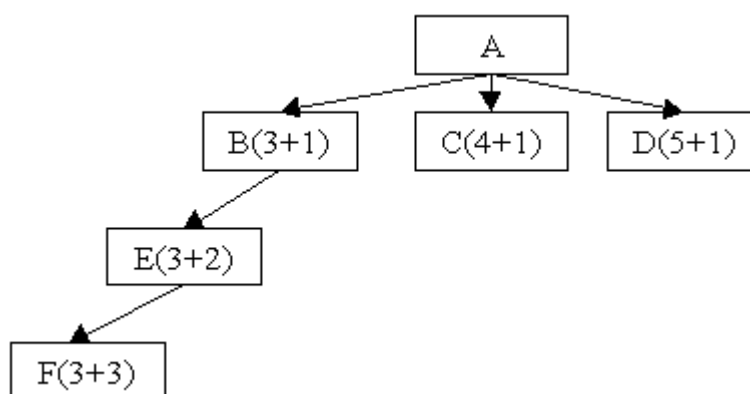
- Dacă ne interesează să ajungem la o soluție, indiferent cum, vom seta întotdeauna $g := 0$;
- Dacă ne interesează să ajungem la o soluție în numărul minim de pași, atunci costul trecerii de la o stare la o stare succesor $:= 1$;
- Dacă dorim drumul cel mai ieftin, și avem costurile operatorilor disponibili, atunci costul trecerii de la o stare la starea succesor vor reflecta costurile operatorilor.

Observația 2 privește rolul funcției h' , estimarea funcției h , distanța unui nod față de nodul final.

- Dacă h' este un estimator perfect al lui h , atunci \mathbf{A}^* va converge imediat la soluție, fără căutare.
- Cu cât h' este mai bun, cu atât mai aproape vom fi de o căutare directă. Dacă, pe de altă parte, valoarea lui h' este întotdeauna zero, căutarea va fi controlată de g .
- Dacă valoarea lui g este de asemenea întotdeauna zero, strategia de căutare va fi aleatoare.
- Dacă valoarea lui g este întotdeauna 1, căutarea va fi Breadth-First. Toate nodurile de pe un nivel vor avea aceeași valoare a lui g , și astfel vor avea valori ale lui f' mai mici decât nodurile de pe nivelul următor.
- Ce se întâmplă dacă h' nu este nici perfect, nici zero? Dacă putem garanta că h' nu supraestimează niciodată pe h , algoritmul \mathbf{A}^* garantează găsirea unui drum optimal către o stare finală (determinat de g), dacă un astfel de drum există. Aceasta se poate vedea în câteva exemple la care vom reveni mai târziu.

Definiție: Un algoritm de căutare care garantează găsirea unui drum optimal către o stare finală, dacă un astfel de drum există, se numește **admisibil**.

Observația 3 privește relația dintre arbori și grafe. Algoritmul \mathbf{A}^* a fost prezentat în forma sa cea mai completă, aplicat pe grafe. Poate fi simplificat să fie aplicat la arbori, prin aceea că nu se va mai verifica dacă un nod nou este deja în listele OPEN și CLOSED. Aceasta face generarea nodurilor mai rapidă, dar poate duce la efectuarea de mai multe ori a unei operații de căutare, dacă nodurile sunt duplicate.

Figura 3.4: h' subestimează pe h

În anumite condiții algoritmul A^* se poate arăta că este optimal în aceea că generează mai puține noduri în procesul găsirii unei soluții la o problemă. În alte condiții nu este optimal.

Și acum iată două exemple relative la relația dintre h' și h .

Exemplul 1: h' subestimează pe h . Fie arborele din Figura 3.4.

Nodul rădăcină: nodul A
 Succesorii lui A: B(3+1), C(4+1), D(5+1)
 Succesorul lui B: E(3+2)
 Succesorul lui E: F(3+3)

Toate costurile arcelor sunt 1. În paranteza, f' este indicat ca $h' + g$.

1. Dintre succesorii lui A, B are valoarea minimă, 4. Deci, este expandat primul.
2. E are valoarea 5, egală cu a lui C. Presupunem că rezolvăm conflictul în favoarea lui E.
3. F are valoarea 6, mai mare decât a lui C, deci vom expanda în continuare pe C. De remarcat că $h(B)=h(E)=h(F)=3$, deci atât B, E cât și F sunt evaluate la 3 mișcări de soluție. Totuși, nu am avansat deloc.

Exemplul 2: h' supraestimează pe h . Fie arborele din Figura 3.5.

Nodul rădăcină: nodul A
 Succesorii lui A: B(3+1), C(4+1), D(5+1)
 Succesorul lui B: E(2+2)
 Succesorul lui E: F(1+3)
 Succesorul lui F: F(0+4)

Mergând pe același raționament ca în cazul anterior producem soluția G, cu un drum de lungime 4. Să presupunem în acest caz că unul din succesorii lui D este o soluție, în felul acesta având un drum de lungime 2.

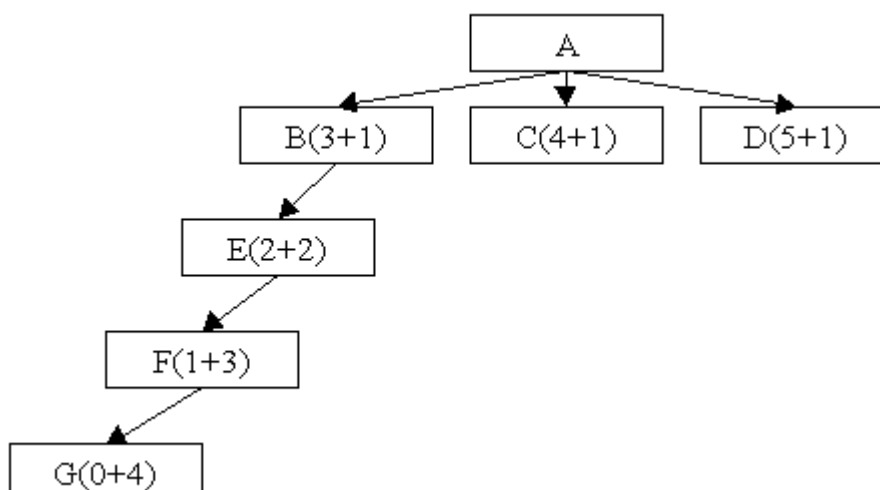


Figura 3.5: h' supraestimează pe h

Observație: Supraevaluarea lui $h(D)$ ne va împiedica să o găsim. D arată atât de prost încât eventual vom găsi o soluție mai rea fără măcar să-l expandăm pe D . Dacă h' poate supraestima pe h , nu putem garanta că obținem cea mai bună soluție decât dacă expandăm tot graful.

Teorema “Dacă h' nu supraestimează pe h atunci A^* este admisibil” nu are o semnificație practică, deoarece singurul mod prin care putem garanta că h' nu supraestimează pe h este de a face $h' = 0$, ceea ce ne întoarce la breadth-first, care e admisibil, dar ineficient. O soluție este următorul

Corolar. [Scăderea Acceptabilă a Admisibilității (Graceful Decay of Admissibility)]

Dacă h' supraestimează rar pe h cu mai mult decât δ , atunci algoritmul A^* va găsi rar o soluție al cărei cost să fie cu peste δ mai mare decât costul soluției optime.

3.3.3. Agende

În discuția noastră despre căutare best-first în grafe OR, am presupus că putem evalua în mod independent mai multe drumuri la același nod. De exemplu, la problema găleților cu apă, valoarea poziției (4,3) este independentă de faptul că există două drumuri prin care această stare poate fi atinsă. Acest lucru nu este adevărat în toate situațiile.

Definiție:: Se numește **agendă** o listă de **taskuri** pe care un sistem le poate realiza. Fiecare task are asociate următoarele:

- o listă de motive pentru care taskul a fost propus;
- un număr (rating) care reprezintă ponderea globală a dovezilor care sugerează că taskul este util.

Algoritm: Căutare condusă de o agendă

1. Execută până când s-a atins o stare finală sau agenda este vidă:

- (a) Alege taskul cel mai promițător. Taskul poate fi reprezentat în orice formă dezirabilă: o declarație explicită a ceea ce urmează să fie făcut, sau o indicare a nodului care urmează să fie expandat.
- (b) Execută taskul prin punerea la dispoziție a numărului de resurse determinate de importanța sa. Resursele importante sunt timpul și spațiul. Execuția taskului va genera alte taskuri. Pentru fiecare din ele, execută:
 - i. Vezi dacă este pe agendă.
 - Dacă da, vezi dacă motivul pentru care a fost realizat este deja pe lista sa de motive.
 - Dacă da, ignoră această operație.
 - Dacă nu, adaugă motivul la listă.
 - Dacă nu, adaugă taskul pe agendă.
 - ii. Calculează ratingul taskului, combinând dovezile din toate motivațiile. Aceste motivații nu vor avea aceleași ponderi. Ponderea este o indicație a importanței motivației.

O problemă importantă este: cum găsim la fiecare ciclu taskul cel mai promițător?

- soluție este să se păstreze lista de taskuri sortată după rating (inserările se vor face la locul corect, iar modificarea ratingului va duce la mutarea taskului în listă). Dar aceasta presupune cheltuirea unui timp suplimentar cu păstrarea listei de taskuri în ordinea corectă.
- a doua soluție este mai ieftină în timp consumat, dar ar putea să nu producă taskul cel mai promițător. La fiecare modificare a ratingului, dacă noul rating intră în primele zece, atunci deplasează taskul la locul corect. Altfel, lasă-l acolo unde se află, sau inserează-l la sfârșit. În plus, din când în când fă o trecere prin agendă și sortează-o.

O structură de control condusă de o agendă este utilă și în cazul în care anumite taskuri (sau noduri) produc motivații negative despre meritele altor task-uri. În aceste situații ar trebui verificat dacă taskurile afectate nu ar trebui să iasă din lista primelor zece.

Există anumite domenii pentru care sistemul nu este potrivit. Mecanismul agendei presupune că dacă **acum** există un motiv bun de a face ceva, atunci și **mai târziu** va exista acel motiv bun pentru a face acea acțiune, cu singura excepție în cazul în care a apărut ceva mai bun. Acest lucru nu este valabil întotdeauna.

3.4. Reducerea problemei

Până acum am considerat strategii de căutare pentru grafuri OR, prin care dorim să găsim o singură cale către o stare finală. Astfel de structuri reprezintă faptul că vom ști cum să ajungem de la o stare la o stare finală, dacă putem descoperi cum să ajungem de la acea stare la o stare finală situată de-a lungul unuia dintre drumurile care pleacă de acolo.

3.4.1. Grafuri AND-OR

Un alt tip de structură, **graful** (sau **arborele**) **AND-OR**, este util pentru reprezentarea soluțiilor problemelor care pot fi rezolvate prin descompunerea lor în mai multe probleme mai mici, fiecare dintre acestea trebuind apoi rezolvate. Această descompunere generează arce pe care le vom numi **arce AND**. Un arc AND pointează către mai mulți succesori, care trebuie rezolvați pentru ca arcul să indice o soluție. Ca și în cazul grafelor OR, dintr-un nod pot porni mai multe arce, acestea indicând mai multe moduri de a se rezolva o problemă. Din această cauză structura se numește graf AND-OR, nu doar graf AND. În Figura 3.6 este prezentat un graf AND-OR simplu. Arcele AND sunt indicate cu o linie care unește toate arcele componente.

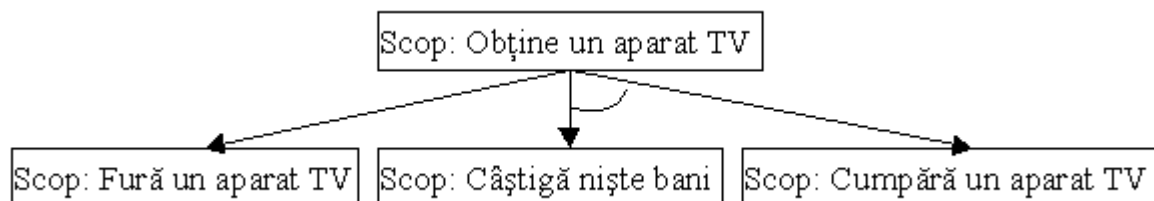


Figura 3.6: Un graf AND-OR simplu

Pentru a găsi soluții într-un graf AND-OR avem nevoie de un algoritm similar cu Best-First, dar care să prezinte abilitatea de a manevra arcele AND. Acest algoritm trebuie să găsească un drum de la starea inițială la o mulțime de noduri reprezentând stările soluție. Să notăm că poate fi necesar să atingem mai multe stări finale, deoarece fiecare braț al unui arc AND trebuie să ducă la propria sa stare finală.

Pentru a vedea de ce algoritmul Best-First nu este adecvat pentru căutarea grafurilor AND-OR, să analizăm exemplul prezentat în Figura 3.7.

Valorile nodurilor reprezintă valorile funcției f' . Presupunem pentru simplitate că fiecare operație are același cost, deci fiecare arc cu un singur succesor are costul 1 și fiecare arc AND cu n succesori are costul n .

Exemplul (a): Dacă ne uităm doar la noduri, atunci trebuie să alegem pentru expandare acel nod cu f' minim, deci C. Dacă însă analizăm toată informația pe care o avem la dispoziție, trebuie să expandăm pe B, deoarece dacă expandăm pe C trebuie să expandăm și pe D, cu un cost total de $f'(C) + f'(D) + 2 = 9$.

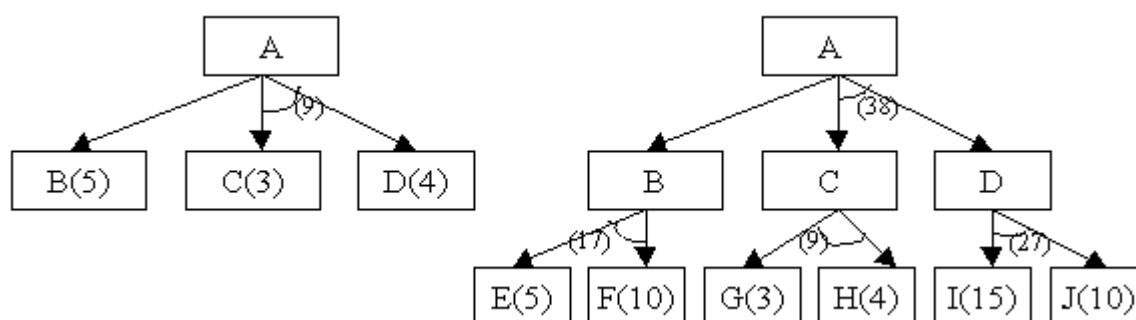


Figura 3.7: Grafe AND-OR

Exemplul (b): Nodul cel mai promițător este G (cu valoarea 3). G face parte chiar din arcul cel mai promițător, $G+H$ (cu valoarea $3+7+2=9$). Dacă expandăm pe G și H trebuie să expandăm și pe I și J. Astfel, G nu face parte din drumul cel mai bun de până acum; acesta din urmă este $A \Rightarrow B \Rightarrow E + F$ (cu valoarea 18). Deci, nodul care trebuie examinat în continuare este E sau F, nicidecum G.

Pentru a descrie un algoritm pentru căutarea grafelor AND-OR avem nevoie de o mărime numită PRAG, care reprezintă costul peste care căutarea se abandonează. Ideea este că orice soluție cu un cost mai mare ca PRAG este prea scumpă pentru a fi de interes practic, chiar dacă poate fi găsită.

Algoritm: Reducerea Problemei

1. Inițializează graful cu nodul de start.
2. Ciclează până când nodul de start este etichetat REZOLVAT sau până când costul său depășește valoarea PRAG:
 - (a) Traversează graful, începând cu nodul de start și urmând drumul cel mai bun de până acum; acumulează mulțimea de noduri care sunt pe acest drum și nu au fost încă expandați sau etichetați cu REZOLVAT.
 - (b) Alege unul din aceste noduri neexpandate și expandează-l. Dacă nu are succesori, fă valoarea acestui nod = PRAG. Altfel, adaugă-i succesorii la graf și pentru fiecare calculează f' (utilizează h' și ignoră g). Dacă f' pe un anumit nod este zero, atunci marchează acel nod cu REZOLVAT.
 - (c) Schimbă f' estimat pe nodul nou expandat pentru a reflecta noua informație oferită de succesorii săi. Propagă această schimbare înapoi prin graf. Dacă un nod conține un arc succesori ai cărui descendenți sunt marcați REZOLVAT, marchează nodul REZOLVAT. La fiecare nod vizitat pe drumul în sus, marchează ca parte din drumul cel mai bun arcul succesori cel mai promițător. Aceasta poate duce la modificarea drumului cel mai bun.

Exemple. Un prim exemplu este prezentat în Figura 3.8. Arcele marcate ca cele mai bune sunt indicate cu săgeată. Acest exemplu ilustrează modul de acțiune al metodei Reducerii Problemei.

În exemplul din Figura 3.9 algoritmul nu produce drumul cu costul cel mai mic. În acest caz, drumul mai bun, și anume cel care ajunge la E prin J, este mai mulg decât cel prin C.

În exemplul din Figura 3.10 algoritmul nu ține seama de interdependențele dintre soluții și lucrează în plus.

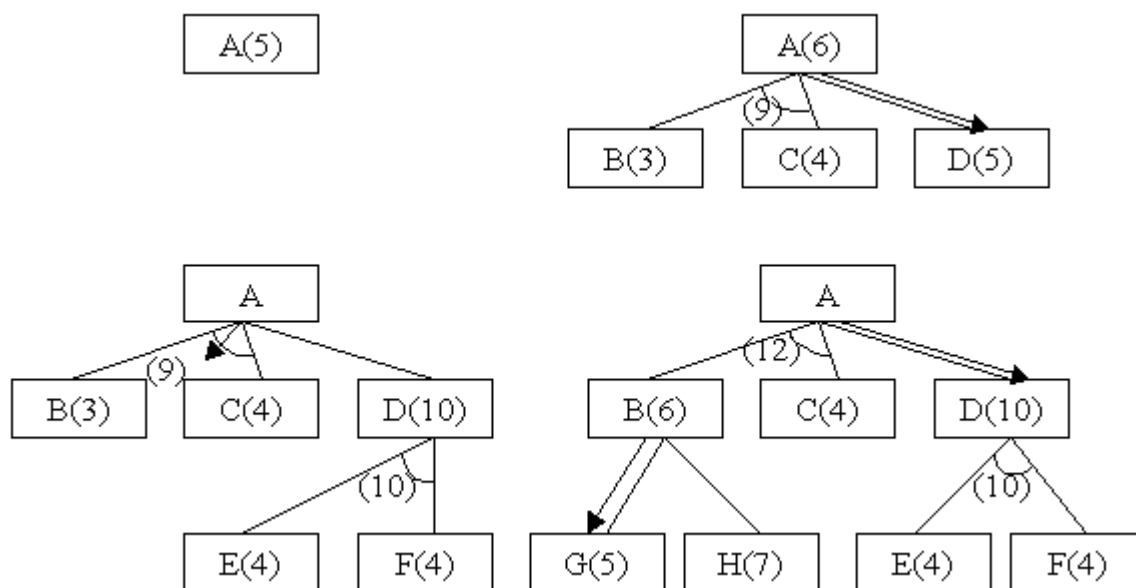


Figura 3.8: Modul de acțiune al metodei Reducerii Problemei

3.4.2. Algoritmul AO*

Algoritmul de reducere a problemelor pe care tocmai l-am vazut este o simplificare a unui algoritm descris în 1973, numit AO*.

Algoritmul va folosi o singură structură, numită GRAPH, reprezentând partea din graful de căutare care a fost generată deja. Fiecare nod din graf va indica atât către succesorii săi imediați cât și către predecesorii săi imediați. Fiecare nod va avea asociată valoarea funcției h' . Nu vom memora g . Această valoare nu este necesară din cauza traversării to-down a celui mai bun drum, care garantează că doar nodurile de pe cel mai bun drum sunt considerate pentru expandare. Deci h' va servi ca estimare a costului unui drum de la un nod la o stare finală.

Algoritm: AO*

1. $GRAPH := \{s.i.\}$; $INIT := s.i.$; calculează $h'(INIT)$.
2. Până când $INIT$ este etichetat SOLVED sau până când $h'(INIT)$ devine mai mare ca PRAG, repetă procedura:

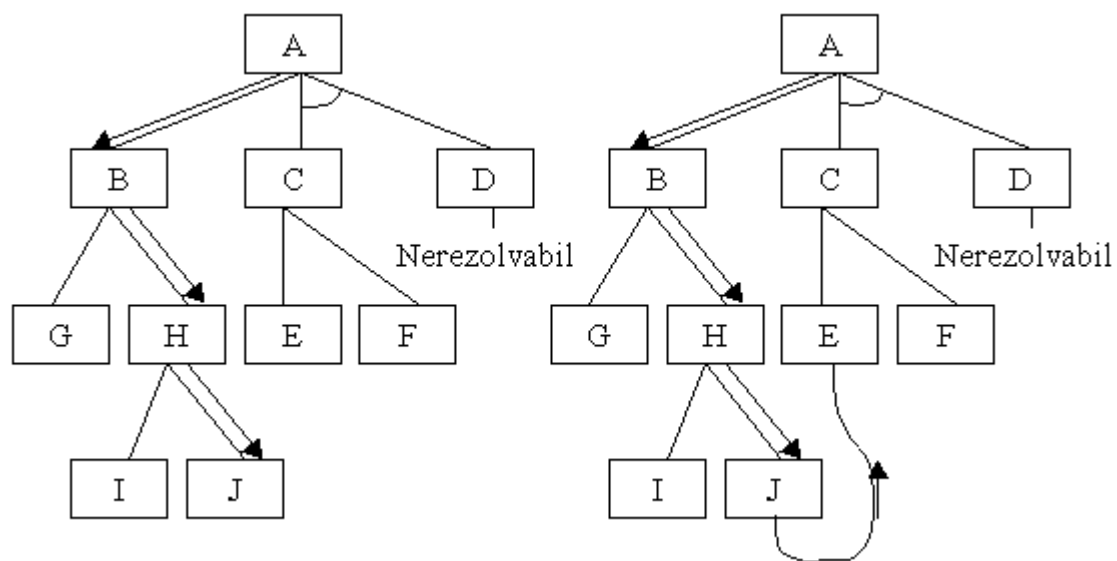


Figura 3.9: Un drum lung poate fi mai bun

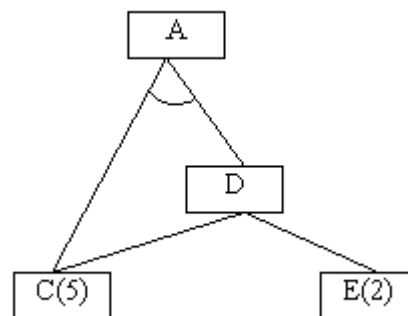


Figura 3.10: Subgoaluri care interacționează

- Selecționează pentru expansiune unul din nodurile etichetate care apar pe drumurile de la INIT și numește acel nod NOD.
- Generează succesorii lui NOD. Dacă nu există, atribuie $h'(\text{NOD}) := \text{PRAG}$. Dacă există, pentru fiecare SUCC care nu este înaintaș al lui NOD execută:
 - Adaugă SUCC la GRAPH.
 - Dacă SUCC este nod terminal, etichetează-l SOLVED și atribuie-i $h'(\text{SUCC}) := 0$.
 - Dacă SUCC nu e terminal, calculează $h'(\text{SUCC})$.
- Propagă noile informații descoperite în susu în graf. Pentru aceasat execută următoarele: Fie S un set de noduri etichetate SOLVED sau ale căror valori h' au fost schimbate și au nevoie de propagare la părinți. fie $S := \{\text{NOD}\}$. Până când S este vid execută:

- i. Dacă este posibil, selectează din S un nod ai cărui descendenți apar în S . Dacă nu există, selectează orice nod din S . Numește-l CURENT și șterge-l din S .
- ii. Determină costul fiecărui arc ce iese din CURENT. Costul arcului este egal cu suma valorilor h' ale fiecărui nod de la sfârșit plus costul propriu-zis al arcului. $h'(\text{CURENT}) := \text{minimul valorilor astfel determinate}$.
- iii. Marchează cel mai bun drum care iese din CURENT, prin marcarea arcului cu cost minim așa cum a fost determinat în pasul anterior.
- iv. Marchează CURENT ca SOLVED dacă toate nodurile conectate la el prin arcele nou etichetate au fost marcate SOLVED.
- v. Dacă CURENT a fost etichetat SOLVED sau dacă costul CURENT a fost schimbat, noua stare trebuie propagată în sus în graf. Deci, adaugă la S toți înaintașii lui CURENT.

Observația 1: În legătură cu propagarea înapoi a informațiilor de cost. La pasul 2(c)v, algoritmul va insera toți înaintașii nodului în mulțime. Aceasta înseamnă că propagarea costului se va face și peste noduri care se știe că nu sunt foarte bune.

Observația 2: În legătură cu terminarea ciclului de la pasul 2(c). Deoarece GRAPH poate conține cicluri nu sunt garanții că acest proces se va termina doar pentru că ajunge în partea de sus a grafului. Există, totuși, un alt motiv pentru care procesul se oprește. **Care anume?**

3.5. Satisfacerea Restricțiilor (Constraints Satisfaction)

Multe probleme de Inteligență Artificială pot fi tratate ca probleme de **satisfacere a restricțiilor**, unde scopul este de a descoperi o anumită stare a problemei care satisface o mulțime de restricții date. Exemple ale acestui tip de probleme includ problemele de criptaritmetică și multe probleme de etichetare perceptuală din lumea reală. Sarcinile de proiectare pot fi și acestea considerate probleme de satisfacere a restricțiilor în care un proiect trebuie creat în interiorul unor limite fixate în timp, cost și materiale.

Prin vizualizarea unei probleme ca o problemă de satisfacere a restricțiilor, este deseori posibil să reducem substanțial efortul de căutare cerut, spre comparație cu o metodă care încearcă să formeze o soluție parțială în mod direct prin alegerea unor valori specifice pentru componente ale unei soluții posibile. De exemplu, o procedură de căutare evidentă pentru a rezolva o problemă de criptaritmetică poate opera într-un spațiu de stări de soluții parțiale, în care literelor le sunt atribuite numere. O schemă de control Depth-First poate urma un drum de atribuire până când s-a descoperit fie o soluție, fie o inconsistență. Spre deosebire de aceasta, o abordare de satisfacere a restricțiilor pentru rezolvarea problemei evită să facă încercări în ceea ce privește atribuirii particulare, și amână această decizie până în momentul în care este nevoie să o ia. Setul inițial de restricții, care spune că fiecare număr corespunde unei singure litere și că suma numerelor trebuie să respecte enunțul problemei, este extins pentru a include

restricții care pot fi deduse din regulile aritmeticii. Apoi, deși s-ar putea ca încercările să fie încă necesare, numărul de încercări permise este mai mic și astfel gradul de căutare este redus.

Satisfacerea restricțiilor este o procedură de căutare care opează într-un spațiu de mulțimi de restricții. Starea inițială conține restricțiile care sunt date în enunțul problemei. O stare finală este orice stare care a fost limitată “suficient”, unde “suficient” trebuie definit pentru fiecare problemă. De exemplu, pentru criptaritmetică, suficient înseamnă că fiecărei litere i-a fost asociată o valoare numerică unică.

Satisfacerea restricțiilor este un proces în doi pași. Mai întâi, restricțiile sunt descoperite și propagate cât de departe posibil prin sistem. Apoi, dacă, totuși, nu am descoperit o soluție, începe căutarea. Se face o încercare în legătură cu valoarea unui anumit obiect și această încercare este adăugată ca limitare nouă. Din nou poate începe propagarea cu această limitare, și așa mai departe.

Primul pas, propagarea, provine din faptul că de obicei sunt dependențe între restricții. Aceste dependențe apar din faptul că multe restricții implică mai mult de un obiect și multe obiecte sunt implicate în mai mult de o limitare. Deci, de exemplu, să presupunem că începem cu o limitare, $N = E + 1$. Atunci, dacă adăugăm limitarea $N = 3$, putem propaga această limitare pentru a obține o limitare mai puternică pentru E , și anume $E = 2$. Propagarea restricțiilor de asemenea apare din prezența regulilor de inferență care permit deducerea unor constrângeri adiționale din constrângerile existente. Propagarea restricțiilor se termină într-unul din următoarele două cazuri. Mai întâi, se poate detecta o contradicție. În acest caz nu există soluții consistente cu toate restricțiile cunoscute. Dacă contradicția implică doar restricții care snt parte din specificația problemei (spre deosebire de acele restricții care provin din încercări), atunci nu există soluție. Al doilea motiv posibil pentru terminare este că nu se mai pot face deducții pe baza setului de restricții existent. Dacă nu s-a identificat o soluție pe baza restricțiilor existente, atunci este nevoie de căutare pentru a putea porni din nou mecanismul deducției de restricții.

În acest punct începe al doilea pas. Trebuie făcute unele ipoteze asupra unei modalități de a întări restricțiile. În cazul problemei de criparitmetică, de exemplu, aceasta înseamnă ghicirea unei valori particulare pentru o anumită literă. După aceasta, propagarea restricțiilor poate începe din nou din această stare. Dacă se găsește o soluție, aceasta se va afișa. Dacă sunt necesare mai multe încercări ca cea de mai sus, acestea se vor putea face. Dacă s-a detectat o contradicție, se poate folosi tehnica backtracking pentru a face o încercare nouă și pentru a continua de acolo. Această procedură poate fi enunțată mai precis după cum urmează:

Algoritm: Satisfacerea Restricțiilor

1. Propagă restricțiile disponibile. Pentru aceasta inițializează OPEN cu mulțimea tuturor obiectelor cărora trebuie să li se atribue valori într-o soluție completă. Apoi execută următorii pași până când se detectează o inconsistență sau până când OPEN este vid:
 - (a) Selectează un obiect OB din OPEN. Întărește cât de mult mulțimea restricțiilor care

se aplică lui OB.

- (b) Dacă această mulțime esre diferită de mulțimea care a fost determinată ultima oară când OB a fost examinat, adaugă la OPEN toate obiectele care partajează cu OB cel puțin o restricție.
 - (c) Înltură OB din OPEN.
2. Dacă reuniunea restricțiilor descoperite mai sus definește o soluție, atunci STOP și raportează soluția.
 3. Dacă reuniunea restricțiilor descoperite mai sus definește o contradicție, atunci STOP și întoarce eșec.
 4. Dacă nici una din condițiile de mai sus nu apare, este necesar să facem o încercare în legătură cu un anumit obiect. Pentru aceasta, ciclează până când o soluție se găsește sau până când toate soluțiile posibile au fost eliminate:
 - (a) Selectează un obiect a cărui valoare nu a fost încă determinată și selectează o modalitate de întărire a restricțiilor acelui obiect.
 - (b) Apelează recursiv satisfacerea restricțiilor cu setul curent de restricții, îmbunătățit cu limitarea produsă mai sus.

Acest algoritm a fost enunțat în modul cel mai general posibil. Pentru a fi plicat într-un domeniu particular, este nevoie de două tipuri de reguli: reguli care definesc modul în care restricțiile pot fi corect propagate și reguli care sugerează încercări, în cazul în care încercările sunt necesare. Merită, totuși, notat că în anumite domenii de probleme încercările ar putea să nu fie necesare.

Problemă:

$$\begin{array}{r} \text{SEND} + \\ \text{MORE} \\ \hline \text{MONEY} \end{array}$$

Starea inițială:

- Literele au valori distincte două câte două.
- Suma cifrelor trebuie să respecte adunarea din problemă.

Figura 3.11: O problemă de criptaritmetică

Pentru a vedea cum funcționează acest algoritm, să considerăm problema de criptaritmetică arătată în Figura 3.11. Starea finală este o stare de problemă în care pentru toate literele au fost atribuite cifre într-un astfel de mod încât toate restricțiile inițiale sunt satisfăcute.

Procesul de determinare a soluției este un proces în cicluri. La fiecare ciclu se realizează două operații semnificative (corespunzând pașilor 1 și 4 din algoritm):

1. restricțiile sunt propagate prin utilizarea regulilor care corespund proprietăților aritmetice.
2. Se ghicește o valoare pentru o anumită literă a cărei valoare încă nu a fost determinată.

În primul pas de obicei nu este importantă ordinea în care se face propagarea, de vreme ce, oricum, toate propagările posibile se vor face înainte de terminare. În al doilea pas, totuși, ordinea în care se fac încercările poate avea un impact substanțial asupra gradului de căutare necesară. Câteva euristici simple pot fi de folos la selectarea celei mai bune încercări de efectuat. De exemplu, dacă există o literă care are doar două valori posibile și o a doua literă cu șase valori posibile, șansele ca prima încercare să fie cea corectă sunt mai mari la prima literă decât la a doua. O altă euristică utilă este aceea că dacă există o literă care participă la mai multe restricții, atunci este o idee bună să o preferăm în favoarea unei alte litere care participă la mai puține restricții. O încercare efectuată asupra unei litere puternic limitate de obicei conduce mai repede fie la o contradicție, fie la generarea mai multor restricții. Pe de altă parte, o încercare asupra unei litere mai puțin limitate produce mai puțină informație.

Rezultatul primilor pași ai procesului este prezentat în Figura 3.12. Deoarece restricțiile nu dispar la nivelurile inferioare, pentru fiecare nivel sunt arătate doar restricțiile noi. Pentru rezolvitor nu va mult fi mai dificil să privească restricțiile ca o mulțime de liste în loc de o singură listă mai lungă. În plus, această abordare este eficientă atât din punct de vedere al memoriei ocupate cât și al ușurinței metodei backtracking. O altă abordare rezonabilă ar fi să memorăm restricțiile într-o singură listă, dar să memorăm la fiecare nod schimbările care trebuie refăcute la backtracking. În Figura 3.12, C1, C2, C3 și C4 indică cifrele de transport între coloane, numerotate de la dreapta la stânga.

În ceea ce privește procesul deductiv, vom face câteva observații. Să notăm că ceea ce se cere de la regulile de propagare a restricțiilor este ca acestea să nu producă restricții false. Nu este necesar să producă toate restricțiile legale. De exemplu, am fi putut raționa până la rezultatul că C1 trebuie să fie egal cu 0. Am fi putut face aceasta observând că pentru ca C1 să fie 1, trebuie să fie validă relația $2 + D = 10 + Y$. Pentru aceasta, D ar trebui să fie 8 sau 9. Dar atât S cât și R trebuie să fie 8 sau 9, și trei litere nu pot partaja două valori. Deci, C1 nu poate fi 1. Dacă am fi realizat aceasta de la început, o anumită căutare s-ar fi putut evita. Dar, deoarece regulile de propagare a restricțiilor pe care le-am folosit nu sunt atât de sofisticate, au condus la efectuarea unei anumite căutări. Dacă căutarea durează mai mult sau mai puțin decât propagarea restricțiilor depinde de cât de mult durează raționarea cerută de propagarea restricțiilor.

O a doua observație se referă la faptul că deseori există două tipuri de restricții. Primul tip este simplu: aceste restricții listează valorile posibile pentru un anumit obiect. Al doilea

tip este mai complex: aceste restricții descriu relații dintre obiecte. Ambele tipuri de restricții joacă același rol în procesul de satisfacere a restricțiilor, și în exemplul de criptaritmetică au fost tratate identic. Totuși, pentru unele probleme poate fi util să reprezentăm cele două tipuri de restricții în mod diferit. restricțiile simple, de listare a valorilor posibile, sunt întotdeauna dinamice și trebuie reprezentate explicit în fiecare stare a problemei. restricțiile mai complicate, care exprimă relații între obiecte, sunt dinamice în domeniul criptaritmeticii, deoarece sunt diferite pentru fiecare problemă de criptaritmetică. Dar în multe alte domenii sunt statice. În aceste situații, ar putea fi eficient din punct de vedere computațional nu să reprezentăm aceste restricții explicit în descrierile stărilor, ci să le codificăm direct în algoritm. Dar esențialmente algoritmul este același în ambele cazuri.

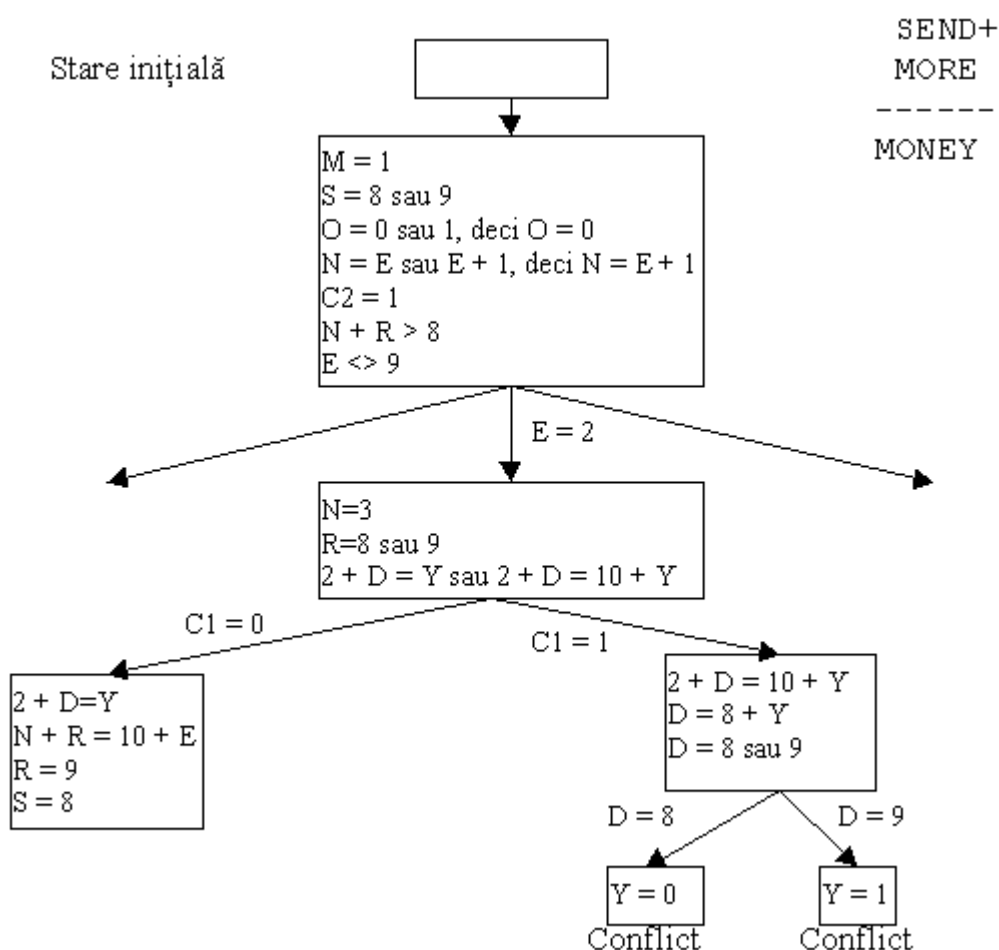


Figura 3.12: Rezolvarea unei probleme de criptaritmetică

3.6. Analiza Means-Ends

Până acum am prezentat o colecție de strategii de căutare care pot raționa fie înainte, fie înapoi, dar, pentru o anumită problemă trebuie aleasă una dintre direcții. Deseori este potrivită o

combinație a celor două direcții. Printr-o astfel de strategie mixtă ar deveni posibil să rezolvăm la început părțile majore ale unei probleme, după care să revenim și să rezolvăm problemele mici care apar la combinarea soluțiilor. O tehnică cunoscută sub numele **analiza means-ends** ne permite acest lucru.

Procesul analizei means-ends se centrează pe detectarea diferențelor dintre starea curentă și starea scop. Imediat ce o astfel de diferență este izolată, trebuie identificat un operator care poate reduce diferența. Dar probabil că acel operator nu se poate aplica stării curente. Deci ne fixăm o subproblemă de a ajunge la o stare în care operatorul să poată fi aplicat. Tipul de înlănțuire înapoi în care operatorii sunt selectați și apoi sub-scopurile sunt fixate pentru a stabili precondițiile operatorului se numește **operator subgoalig**. Dar probabil că operatorul nu produce exact starea finală pe care o dorim. Atunci avem o a doua subproblemă, aceea de a ajunge din starea pe care o produce în starea finală. Dar dacă diferența a fost selectată corect și dacă operatorul este realmente efectiv în reducerea diferenței, atunci cele două subprobleme ar trebui să fie mai simplu de rezolvat decât problema originală. Procesul analizei means-ends poate fi apoi aplicat recursiv. Pentru a focaliza atenția sistemului mai întâi asupra problemelor dificile, diferențele pot primi niveluri de prioritate. Diferențele cu prioritate mai mare pot fi tratate înaintea celor cu prioritate mai mică.

Precum celelalte tehnici de rezolvare a problemelor pe care le-am discutat, analiza means-ends se bazează pe o serie de reguli care pot transforma o stare a problemei în alta. Aceste reguli de obicei nu sunt reprezentate cu descrierile complete ale stărilor de fiecare parte, ci ca o parte stângă care descrie condițiile care trebuie satisfăcute pentru ca regula să fie aplicabilă (aceste condiții se numesc **precondițiile** regulii), și o parte dreaptă care descrie acele aspecte ale stării problemei care vor fi schimbate prin aplicarea regulii. O structură de date separată, numită **tabel de diferențe**, indexează regulile după diferențele pe care aceste reguli le pot reduce.

Operator	Precondiții	Rezultat
PUSH (obj, loc)	at (robot, obj) + large (obj) + clear (obj) + armempty	at (obj, loc) + at (robot, loc)
CARRY (obj, loc)	at (robot, obj) + small (obj)	at (obj, loc) + at (robot, loc)
WALK (loc)	nimic	at (robot, loc)
PICKUP (obj)	at (robot, obj)	holding (obj)
PUTDOWN (obj)	holding (obj)	holding (obj)
PLACE (obj1, obj2)	at (robot, obj2) + holding (obj1)	on (obj1, obj2)

Figura 3.13: Operatorii robotului

Ca exemplu să considerăm un robot simplu. Operatorii disponibili sunt descriși în Figura 3.13, împreună cu precondițiile și rezultatele. Figura 3.14 arată tabelul de diferențe care descrie când este aplicabil fiecare operator. Să notăm că uneori pot exista mai mulți operatori care pot reduce o anumită diferență și că un anumit operator poate reduce mai multe diferențe.

	Push	Carry	Walk	Pickup	Putdown	Place
Deplasează obiect	*	*				
Deplasează robot			*			
Curăță obiect				*		
Pune obiect pe obiect						*
Golește brațul					*	*
Ia un obiect				*		

Figura 3.14: Un tabel de diferențe

Să presupunem că robotul din acest domeniu primește problema deplasării unei mese cu două lucruri pe ea dintr-o cameră în alta. Obiectele trebuie, de asemenea, deplasate. Principala diferență dintre starea de start și starea finală este locul în care se află masa. Pentru a reduce această diferență se pot aplica fie PUSH, fie CARRY. Dacă se alege CARRY, condițiile sale trebuie satisfăcute. Aceasta produce două noi diferențe: locul în care se află robotul și dimensiunea mesei. Locul robotului se poate trata cu WALK, dar nu există operatori pentru schimbarea dimensiunii unui obiect. Deci acest drum duce la un blocaj. Urmăm cealaltă alternativă și încercăm să aplicăm PUSH (vezi Figura 3.15). Dar, pentru a putea aplica PUSH trebuie să reducem diferențele dintre A și B și dintre C și D.

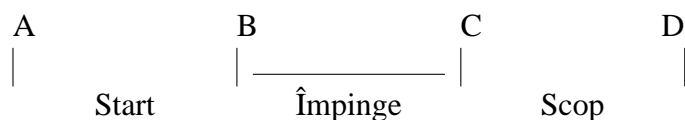


Figura 3.15: Situația problemei folosind metoda Analizei Means-Ends

PUSH are patru condiții, dintre care două produc diferențe între starea inițială și starea finală: robotul trebuie să fie la masă și masa trebuie să fie curată. Deoarece masa este deja mare și brațul robotului este gol, aceste două condiții pot fi ignorate. Robotul poate fi adus la locul corect folosind WALK. Și suprafața mesei poate fi curățată cu două utilizări ale lui PICKUP. Dar, după o aplicare a lui PICKUP, încă o încercare produce o altă diferență: brațul trebuie să fie gol. Pentru reducerea acestei diferențe se folosește PUTDOWN.

Imediat ce s-a aplicat PUSH, starea problemei este aproape de starea finală. Obiectele mai trebuie plasate pe masă cu PLACE. Dar PLACE nu poate fi aplicat imediat: trebuie eliminată o altă diferență, deoarece robotul trebuie să aibă obiectele. Situația problemei în acest moment este prezentată în Figura 3.16. Diferența finală dintre C și E se poate reduce folosind WALK pentru a duce robotul la obiecte, urmat de PICKUP și CARRY.

Procesul pe care l-am ilustrat (și pe care îl notăm, pe scurt, **AME**) poate fi sumarizat astfel:

Algoritm: Analiza Means-Ends (CURRENT, SCOP)

1. Compară stările CURRENT și SCOP. Dacă nu sunt diferențe între ele, STOP.

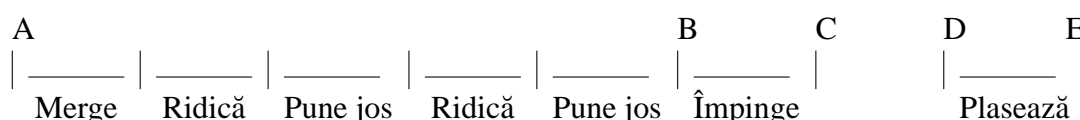


Figura 3.16: Situația problemei folosind metoda Analizei Means-Ends

2. Altfel, selectează diferența cea mai importantă și redu-o prin efectuarea următorilor pași până când se raportează succes sau eșec:
 - (a) Selectează un operator O neîncercat care este aplicabil la diferența curentă. Dacă nu există astfel de operatori, raportează eșec.
 - (b) Încearcă să aplici O la $CURRENT$. Generează descrierile celor două stări: O_START , stare în care sunt satisfăcute precondițiile lui O , și $O_REZULTAT$, stare care ar fi produsă dacă O s-ar aplica pe O_START .
 - (c) Dacă apelurile $FIRST_PART := AME(CURRENT, O_START)$ și $LAST_PART := AME(O_REZULTAT, SCOP)$ se termină cu succes, raportează succes și întoarce ca rezultat al apelului original concatenarea dintre $FIRST_PART$, O și $LAST_PART$.

În discuție au fost omise multe dintre detaliile acestui proces. În particular, ordinea în care diferențele sunt considerate poate fi critică. Este important ca diferențele semnificative să fie reduse înaintea celorlalte mai puțin critice. În caz contrar, se poate pierde un efort considerabil.

Procesul simplu pe care l-am descris nu este, de obicei, adecvat pentru rezolvarea problemelor complexe. Numărul de permutări ale diferențelor poate deveni prea mare. Lucrul asupra reducerii unei diferențe poate interfera cu planul pentru reducerea alteia. În plus, în domeniile complexe tabelele de diferențe ar fi imense.

3.7. Aplicații rezolvate și probleme propuse

3.7.1. Analizați comparativ următorii algoritmi:

- *Hill-climbing* și *Algoritmul de călire simulată*;
- *Algoritmul AO** și *Algoritmul de planificare*;
- *Algoritmul Hill-climbing de pantă maximă* și *algoritmul de căutare Best-First*.

Ilustrați deosebirile pe exemple concrete.

3.7.2. *Descrieți un algoritm de căutare eficient pentru rezolvarea următoarei probleme: Să se afle rădăcina pătrată a numărului 123454321. Să se analizeze problema și să se specifice euristica aleasă în rezolvare. Generalizare pentru:*

- un număr de forma $1234 \dots n \dots 4321$;
- un număr palindrom oarecare (număr palindrom este un număr care citit de la stânga la dreapta sau de la dreapta la stânga reprezintă același număr).

3.7.3. Dați exemplu de spațiu de căutare în care algoritmul AO* nu furnizează soluția corectă. Justificați răspunsul.

3.7.4 (Problema pătratului). Să se descrie o euristică admisibilă pentru problema pătratului și să se propună un algoritm de rezolvare al problemei.

Se consideră un pătrat cu $n \times n$ căsuțe. Fiecare căsuță conține un număr între 1 și $n \times n - 2$. Două căsuțe sunt ocupate cu numărul 0. Fiecare număr natural, diferit de 0, apare o singură dată în cadrul pătratului. Știind că 0 își poate schimba poziția cu orice număr natural aflat deasupra, la dreapta, la stanga sau jos, în raport cu poziția în care se află numărul 0, se cere numărul minim de mutări prin care se poate ajunge de la o configurație inițială la o configurație finală.

Exemplu

3 2 0		1 2 3
1 0 4	→	5 0 4
5 7 6		0 7 6
Inițial		Final

Rezolvare

Algoritmul propus spre rezolvare este algoritmul A*, folosit în tehnica *Branch and Bound* (vezi partea de teorie).

Problema se va rezolva pe un arbore ale cărui noduri vor conține configurații, rădăcina arborelui fiind configurația inițială.

Pentru a asigura găsirea celei mai scurte soluții, se va asocia fiecărui nod (configurație) o funcție euristică f' , obținută ca suma a altor două funcții g și h' ($f' = g + h'$), unde:

g – măsură a costului de trecere din starea inițială în starea curentă;

h' – estimare a costului nodului (costul de a ajunge din starea curentă în starea finală).

Pentru problema pătratului, propunem două variante de alegere a funcției h' :

(a) numărul de cifre nenule prin care configurația curentă diferă de configurația finală;

Exemplu

$$a = \begin{pmatrix} 1 & 3 & 2 \\ 4 & 5 & 7 \\ 6 & 0 & 0 \end{pmatrix} \rightarrow h'(a) = 0 + 1 + 1 + 0 + 0 + 1 + 1 = 4$$

- (b) se calculează ca sumă, pentru fiecare cifră în parte a numărului de mutări necesare pentru a o putea aduce în poziția corespunzătoare configurației finale (distanța Manhattan). De exemplu, avem $h'(a) = 0 + 1 + 1 + 2 + 1 + 2 + 2 = 9$.

Observații

- nici una dintre funcțiile propuse anterior nu supraestimează pe h (numărul de mutări necesar pentru a ajunge la soluția finală), ceea ce asigură obținerea soluției;
- a doua funcție h' aproximează mai bine decât prima efortul de a ajunge la soluția finală (în acest caz se va ajunge mai repede la rezultat);
- Având aleasă euristica pentru rezolvare, aplicarea algoritmului A^* nu mai ridică probleme.

În continuare vom prezenta codul sursă al programului BORLAND PASCAL care rezolvă problema pătratului, aplicând algoritmul A^* și folosind observațiile făcute anterior. Nu mai revenim asupra pașilor de aplicare ai algoritmului A^* , aceștia fiind prezentați în partea de teorie.

uses

crt;

const

n = 3;

type

mat = array[1..n, 1..n] of byte;

const

ci: mat = ((3,2,0), (1,0,4), (5,7,6)); {configuratia initiala}
cf: mat = ((1,2,3), (5,0,4), (0,7,6)); {configuratia finala}
dx: array[1..4] of -1..1 = (-1,0,1,0); {pozitiile relative pe directiile}
dy: array[1..4] of -1..1 = (0,-1,0,1); {N, S, E, V - mutarile}

type

lista = ^nod;

nod = record

inf: mat; {configuratia curenta}

g, h: byte; {functiile g si h'}

sus, {legatura spre parinte}

legs, {leg. spre urmatorul nod din lista succesorilor}

leg: lista; {leg. spre urmatorul nod din open sau closed}

end ;

function *h* (*a*, *b*: *mat*; *n*: *integer*): *byte*;

{ se calculeaza valoarea functiei *h* atasata configuratiei curente *a*, *b* fiind configuratia finala }

var

i, *j*, *k*, *l*, *s*: *byte*;

begin

s := 0;

for *l* := 1 **to** *n* **do**

for *k* := 1 **to** *n* **do**

for *i* := 1 **to** *n* **do**

for *j* := 1 **to** *n* **do**

if (*a*[*l*,*k*] <> 0) **and** (*a*[*l*,*k*] = *b*[*i*,*j*]) **then**

s := *s* + *abs*(*i*-*l*) + *abs*(*j*-*k*);

h := *s*;

end ;

function *egal* (*a*, *b*: *mat*): *boolean*;

{ se verifica daca cele doua configuratii *a* si *b* coincid }

var

i, *j*: *integer*;

begin

for *i* := 1 **to** *n* **do**

for *j* := 1 **to** *n* **do**

if *a*[*i*,*j*] <> *b*[*i*,*j*] **then**

begin

egal := *false*;

exit;

end ;

egal := *true*;

end ;

procedure *apare* (*p*, *cap_open*, *cap_close*: *lista*;

var *apare_o*, *apare_c*: *boolean*;

var *q_o*, *q_c*, *prec_o*, *prec_c*: *lista*);

{ verifica daca configuratia corespunzatoare nodului *p* apare in open sau closed. In caz afirmativ, se retine adresa din lista unde apare configuratia, precum si adresa nodului precedent }

```

var
  r: lista;
begin
  apare_o := false;
  apare_c := false;
  r := cap_open;    {nodul curent cu care se parcurge lista open}
  q_o := nil;
  prec_o := nil;
  while (r <> nil) and not apare_o do
    if egal(p^.inf, r^.inf) then
      begin
        apare_o := true;
        q_o := r;
      end
    else
      begin
        prec_o := r;
        r := r^.leg;
      end;
  r := cap_close;    {nodul curent cu care se parcurge lista closed}
  q_c := nil;
  prec_c := nil;
  while (r <> nil) and not apare_c do
    if egal(p^.inf, r^.inf) then
      begin
        apare_c := true;
        q_c := r;
      end
    else
      begin
        prec_c := r;
        r := r^.leg;
      end;
  end;

procedure scrie(p: lista);
  { tipareste solutia problemei, pornind de la nodul curent, mergand pe legatura de tip
  parinte: "sus" }
var

```

```

    i, j: integer;
begin
    if p<>nil then
        begin
            scrie(p^.sus);
            readln;
            for i := 1 to n do
                begin
                    for j := 1 to n do
                        write(p^.inf[i,j]: 2);
                    writeln;
                end;
            writeln;
        end;
    end;
end;

procedure expandare(p: lista; var cap_succ, coada_succ: lista);
{ procedura care realizeaza expandarea configuratiei curente (p^.inf), prin efectuarea
  tuturor mutarilor posibile si generarea listei succesorilor }
var
    k, i, j, l1, l2, c1, c2: integer;
    config: mat;
    q: lista;
begin
    k := 0;
    for i := 1 to n do
        for j := 1 to n do
            if p^.inf[i,j] = 0 then
                if k = 0 then
                    begin
                        l1 := i;      { linia primului 0 }
                        c1 := j;      { coloana primului 0 }
                        k := 1;
                    end
                else
                    begin
                        l2 := i;      { linia celui de-al doilea 0 }
                        c2 := j;      { coloana celui de-al doilea 0 }
                    end
                end;
        end;
    end;
end;

```

```

cap_succ := nil;
coada_succ := nil;
for k := 1 to 8 do      { se efectueaza cele 8 mutari posibile }
  begin
    config := p^.inf;
    if k <= 4 then
      if (l1+dx[k] in [1..n]) and
         (c1+dy[k] in [1..n]) and
         (config[l1+dx[k], c1+dy[k]] <> 0) then
        begin
          config[l1, c1] := config[l1+dx[k], c1+dy[k]];
          config[l1+dx[k], c1+dy[k]] := 0;
        end
      else if (l2+dx[k-4] in [1..n]) and
              (c2+dy[k-4] in [1..n]) and
              (config[l2+dx[k-4], c2+dy[k-4]] <> 0) then
        begin
          config[l2, c2] := config[l2+dx[k-4], c2+dy[k-4]];
          config[l2+dx[k-4], c2+dy[k-4]] := 0;
        end
      end
    if not egal(p^.inf, config) then
      { daca noua configuratie difera de configuratia parinte }
      begin
        { genereaza noua configuratie si se introduce in lista succesorilor }
        new(q);
        q^.sus := p;
        q^.g := p^.g+1;
        q^.inf := config;
        q^.h := h(q^.inf, cf, n);
        q^.legs := nil;
        if cap_succ = nil then
          begin
            cap_succ := q;
            coada_succ := q;
          end
        else
          begin
            coada_succ^.legs := q;
            coada_succ := q;
          end
        end
      end
    end
  end

```

```

        end ;
    end ;
end ;
end ;

procedure aleg_open(var cap_open, coada_open, cap_close, coada_close,
p:lista) ;
{ selecteaza din open nodul avand valoarea minima; il scoate din open si il introduce in
closed }

var
    min, min1, i, j: integer;
    prec, q, precp: lista;
begin
    min := cap_open^.g+cap_open^.h;
    min1 := cap_open^.h;
    p := cap_open;
    q := cap_open;           { nodul curent }
    prec := nil;           { precedentul nodului curent }
    while q<>nil do
        begin
            if (min>q^.g+q^.h) or ( (min=q^.g+q^.h) and (min1>q^.h) ) then
                begin           { s-a gasit un nod mai promitator }
                    min := q^.g+q^.h;
                    min1 := q^.h;
                    p := q;           { nodul cu valoarea minima }
                    precp := prec;   { precedentul nodului cu valoarea minima }
                end;
                prec := q;
                q := q^.leg;
            end;

            { se scoate din open nodul p }
            if p=cap_open then
                cap_open := cap_open^.leg
            else if p=coada_open then
                begin
                    coada_open := precp;
                    precp^.leg := nil;
                end

```



```

else
    precp^.leg := p^.leg;

    { se introduce in closed nodul p }
    p^.leg := nil;
    if cap_close=nil then      { in closed nu mai exista alte noduri }
        begin
            cap_close := p;
            coada_close := p;
        end
    else
        begin
            { se adauga nodul la sfarsitul listei closed }
            coada_close^.leg := p;
            coada_close := p;
        end;
end;

procedure scot_close (p, prec: lista; var cap_close, coada_close: lista);
{ scoate nodul p din lista closed - precedentul nodului este prec }
begin
    if p=cap_close then
        cap_close := cap_close^.leg
    else if p=coada_close then
        begin
            coada_close := prec;
            prec^.leg := nil;
        end
    else
        prec^.leg := p^.leg;
end;

procedure adaug_open (p: lista; var cap_open, coada_open: lista);
{ se adauga nodul p la sfarsitul listei open }
begin
    p^.leg := nil;
    p^.h := h(p^.inf, cf, n);
    if cap_open=nil then
        begin
            cap_open := p;

```

```

    coada_open := p;
    end
else
    begin
        coada_open^.leg := p;
        coada_open := p;
    end
end;

procedure b_b;
{ genereaza sirul minim de mutari pentru a ajunge de la configuratia initiala la
  configuratia finala }
var
    cap_open, coada_open, {primul si ultimul element din lista open}
    cap_close, coada_close, {primul si ultimul element din lista closed}
    cap_succ, coada_succ: lista; {primul si ultimul element din lista succesori}
    p, q, q_o, q_c, prec_o, prec_c: lista;
    gata, t, apare_o, apare_c: boolean;
begin
    {crearea listei open}
    new(cap_open);
    coada_open := cap_open;
    cap_open^.inf := ci;
    cap_open^.g := 0;
    cap_open^.sus := nil;
    cap_open^.h := h(ci, cf, n);
    cap_open^.leg := nil;

    {lista closed la inceput e vida}
    cap_close := nil;
    coada_close := nil;
    gata := false;      {inca nu s-a gasit solutia}
    while (cap_open<>nil) and (not gata) do
        begin
            aleg_open(cap_open, coada_open, cap_close, coada_close, p);
            if h(p^.inf, cf, n)=0 then      {s-a ajuns la configuratia finala}
                begin
                    writeln('Solutia este:');
                    scrie(p);

```

```

    gata := true;
end
else
    begin
        { se genereaza succesorii configuratiei curente }
        expandare(p, cap_succ, coada_succ);
        { se prelucreaza fiecare succesor conform algoritmului A* }
        while cap_succ <> nil do
            begin
                t := false;
                cap_succ^.g := p^.g + 1;
                apare(cap_succ, cap_open, cap_close, apare_o, apare_c, q_o, q_c,
prec_o, prec_c);
                if apare_o or apare_c then
                    { nodul apare in open sau closed }
                    if apare_o then
                        { apare in open }
                        if cap_succ^.g < q_o^.g then
                            { e mai bun decat nodul din open }
                            begin
                                q_o^.g := cap_succ^.g;
                                q_o^.sus := p;
                            end
                        else
                            { nodul nou generat va putea fi sters }
                            t := true
                        else
                            { apare in close }
                            if cap_succ^.g < q_c^.g then
                                { e mai bun decat nodul din closed }
                                begin
                                    q_c^.g := cap_succ^.g;
                                    q_c^.sus := p;
                                    { se scoate nodul din closed }
                                    scot_close(q_c, prec_c, cap_close, coada_close);
                                    { se adauga in open }
                                    adaug_open(q_c, cap_open, coada_open);
                                end
                            else
                                {

```

```

        {nodul nou generat va putea fi sters}
        t := true
    else
        {nodul nu apare nici in open, nici in closed}
    begin
        cap_succ^.sus := p;
        {se adauga in open}
        adaug_open(cap_succ, cap_open, coada_open);
    end;
    q := cap_succ;
    {se trece la urmatorul succesor}
    cap_succ := cap_succ^.legs;
    if t then
        dispose(q);    {nodul poate fi eliberat}
    end;
end;
end;

if not gata then
    begin
        writeln ('Nu exista solutie a problemei!');
        readln;
    end;
end;

{ programul principal }
begin
    clrscr;
    b_b;
    readln;
end.

```

3.7.5 (8-puzzle). Să se descrie o euristică admisibilă pentru problema 8-puzzle și să se propună un algoritm de rezolvare al problemei.

3.7.6 (Problema căsuțelor adiacente). Să se descrie o euristică admisibilă pentru problema căsuțelor adiacente și să se propună un algoritm de rezolvare al problemei.

Se consideră $2n$ căsuțe adiacente situate pe aceeași linie. Două căsuțe adiacente sunt goale, $n - 1$ căsuțe conțin caracterul 'A', iar celelalte $n - 1$ conțin caracterul 'B'. O mutare constă în interschimbarea conținutului a două căsuțe adiacente nevide cu cele două căsuțe libere. Se cere numărul minim de mutări (în cazul în care există soluție) prin care se poate ajunge de la o configurație inițială la o configurație finală în care toate caracterele 'A' apar înaintea caracterelor 'B', poziția căsuțelor goale putând fi oricare.

Exemplu

ABBABBAA**AB	AAAAAB**BBBB	* – spațiu
Inițial	Final	

3.7.7 (Problema căsuțelor). Să se descrie o euristică admisibilă pentru problema căsuțelor și să se propună un algoritm de rezolvare al problemei.

Se consideră $2n$ căsuțe adiacente situate pe aceeași linie. Două căsuțe adiacente sunt goale, $n - 1$ căsuțe conțin caracterul 'A', iar celelalte $n - 1$ conțin caracterul 'B'. O mutare constă în interschimbarea conținutului a două căsuțe adiacente nevide cu cele două căsuțe libere. Se cere numărul minim de mutări prin care se poate ajunge de la o configurație inițială la o configurație finală dată. Dacă nu există soluție a problemei, se dă mesaj.

Exemplu

ABBABBAA**AB	ABABBA**ABAB	* – spațiu
Inițial	Final	

3.7.8. Se consideră arborele de căutare din Figura 3.17. În paranteze sunt trecute valorile funcției de cost atașate nodurilor arborelui. Indicați ordinea în care sunt verificate nodurile arborelui într-o căutare Best-First.

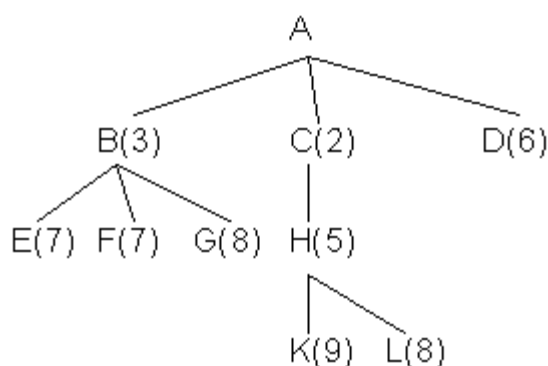


Figura 3.17: Arborele de căutare pentru Problema 3.7.8

Rezolvare

Într-o căutare *Best-First*, la fiecare pas se selectează nodul cel mai promițător (având valoarea funcției de cost cea mai mică) din mulțimea nodurilor generate până la un moment dat.

Ordinea în care se vor analiza nodurile arborelui anterior va fi următoarea:

- se alege spre expandare nodul C (din mulțimea de noduri [B, C, D] are cea mai mică valoare a funcției de cost – 2);
- se alege spre expandare nodul B (din mulțimea de noduri [B, H, D] are cea mai mică valoare a funcției de cost – 3);
- se alege spre expandare nodul H (din mulțimea de noduri [E, F, G, H, D] are cea mai mică valoare a funcției de cost – 5);
- se va continua cu expandarea nodului D (din mulțimea de noduri [E, F, G, K, L, D] are cea mai mică valoare a funcției de cost – 6).

3.7.9. Se consideră arborele de căutare din Figura 3.18. În paranteze sunt trecute valorile funcției de cost atașate nodurilor arborelui. Indicați ordinea în care sunt verificate nodurile arborelui într-o căutare *Best-First*.

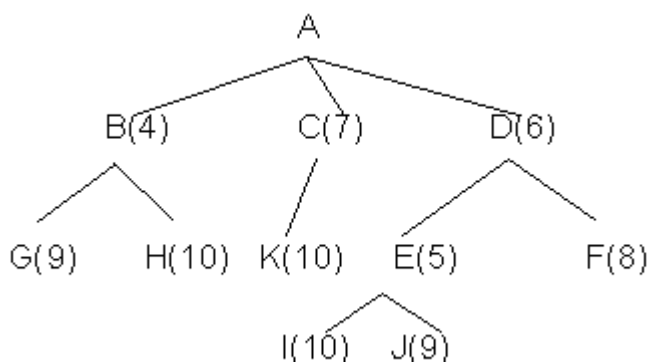


Figura 3.18: Arborele de căutare pentru Problema 3.7.9

3.7.10. Să se rezolve următoarea problemă de criptaritmetică:

$$\begin{array}{r}
 BAD \quad + \\
 ADC \\
 \hline
 DDCE
 \end{array}$$

Se vor lua în considerare următoarele restricții:

- $B = 8$;

- $A \leq 2$;
- două litere distincte nu pot avea aceeași cifră asociată;
- adunarea trebuie să fie corectă din punct de vedere matematic.

Rezolvare

Notăm cu C1, C2, C3 cifrele de transport pe coloane, numerotate de la dreapta spre stânga.

Arborele de analiză este prezentat în Figura 3.19. Soluția problemei este unică și este : $A=2, B=9, C=3, D=1, E=4$.

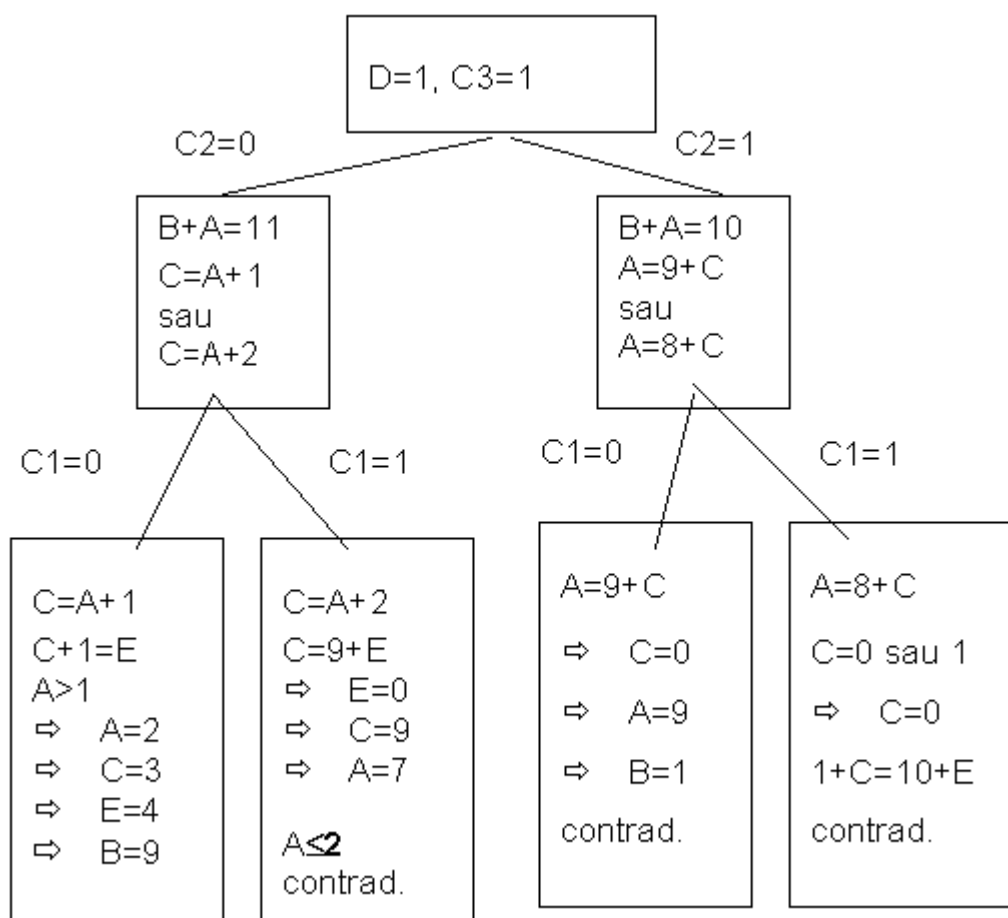


Figura 3.19: Arborele de analiză pentru Problema 3.7.10

3.7.11. Să se rezolve următoarea problemă de criptaritmetică:

$$\begin{array}{r}
 BDC \quad + \\
 CBF \\
 \hline
 AACB
 \end{array}$$

Se vor lua în considerare următoarele restricții:

- $B=8$;
- două litere distincte nu pot avea aceeași cifră asociată;
- adunarea trebuie să fie corectă din punct de vedere matematic.

3.7.12. Să se rezolve următoarea problemă de criptaritmetică:

$$\begin{array}{r} PSR \quad + \\ SPO \\ \hline LLSP \end{array}$$

Se vor lua în considerare următoarele restricții:

- $P=7$;
- două litere distincte nu pot avea aceeași cifră asociată;
- adunarea trebuie să fie corectă din punct de vedere matematic.

Capitolul 4

Introducere în Reprezentarea cunoștințelor

Până acum am discutat despre cadrul de bază pentru construirea programelor de rezolvare a problemelor prin căutare. Aceste metode sunt suficient de generale, astfel încât am putut să discutăm despre ele fără să me referim la modul de reprezentare a cunoștințelor de care ele au nevoie. Totuși, puterea acestor metode de rezolvare a problemelor este limitată tocmai datorită generalității lor. Așa cum vom vedea în continuare, anumite modele de reprezentarea cunoștințelor permit utilizarea unor mecanisme de rezolvare a problemelor mai specifice, mai puternice.

4.1. Moduri de reprezentare

Pentru a rezolva problemele complexe întâlnite în inteligența artificială, avem nevoie atât de o cantitate mare de cunoștințe cât și de unele mecanisme care să permită manipularea acelor cunoștințe pentru a crea soluții ale problemelor noi. În toată discuția care urmează vom întâlni două tipuri de entități:

Fapte: adevăruri într-un domeniu relevant. Acestea sunt lucrurile pe care dorim să le reprezentăm.

Reprezentări ale faptelor într-un anumit formalism. Acestea sunt lucrurile pe care vom putea să le manipulăm.

Un mod de a gândi structura acestor entități este pe două nivele:

Nivelul cunoștințelor, la care toate faptele (inclusiv comportarea fiecărui agent și scopurile curente) sunt descrise.

Nivelul simbolurilor, la care reprezentările obiectelor de la nivelul cunoștințelor sunt definite în funcție de simbolurile care pot fi manipulate de program.

În loc să gândim aceste niveluri ca fiind subordonate unul altuia, ne vom orienta către fapte, reprezentări și corespondențele bi-direcționale care trebuie să existe între acestea. Vom numi aceste legături **corespondențe de reprezentare** (*representation mappings*). Corespondența de reprezentare înainte realizează corespondența de la fapte la reprezentări. Corespondența de reprezentare înapoi merge în sens invers, de la reprezentări la fapte.

O anumită reprezentare a faptelor este atât de uzuală încât merită o mențiune specială: propozițiile din limbajul natural. Indiferent de reprezentarea faptelor pe care o utilizăm în program, am putea avea nevoie și de o reprezentare a acelor fapte în limbajul natural, pentru a facilita transferul de informație în și din sistem. În acest caz trebuie să avem și funcții de corespondență de la propozițiile din limbajul natural la reprezentarea pe care o vom folosi de fapt în program, și de la aceasta înapoi la propoziții.

Să analizăm un exemplu. Fie propoziția următoare:

Lăbuș este un câine.

Faptul reprezentat de această propoziție poate fi reprezentat în logică sub forma:

câine(Lăbuș)

Să presupunem acum că avem o reprezentare logică a faptului că toți câinii au cozi:

$$\forall x : \text{câine}(x) \rightarrow \text{arecoadă}(x)$$

Apoi, folosind mecanismele deductive ale logicii putem genera un nou obiect:

arecoadă(Lăbuș)

Folosind o funcție de corespondență înapoi potrivită putem genera propoziția:

Lăbuș are coadă.

Trebuie să reținem că de obicei funcțiile de corespondență nu sunt bijective. În majoritatea cazurilor, nici nu sunt funcții, ci relații: fiecare obiect din domeniu poate fi pus în corespondență cu mai multe elemente din codomeniu, și mai multe elemente din domeniu pot fi puse în corespondență cu același element din codomeniu. De exemplu, afirmația “Fiecare câine are coadă” poate însemna că fiecare câine are cel puțin o coadă, sau că există o anumită coadă pe care o are fiecare câine.

Uneori o bună reprezentare face ca operarea programului de raționare să fie nu doar corectă, ci banală. Un exemplu este acela al problemei tablei de șah mutilate:

Problema tablei de șah mutilate. Să considerăm o tablă de șah normală din care au fost îndepărtate două pătrate, din colțuri opuse. Problema constă în acoperirea în întregime a restului pătratelor cu piese de domino, fiecare acoperind două pătrate. Nu se acceptă suprapuneri ale pieselor de domino una peste alta, sau depășiri ale cadrului tablei mutilate de către piesele de domino. Este acest lucru posibil?

Un mod de a rezolva problema este de a încerca enumerarea exhaustivă a tuturor îmbinărilor de piese de domino posibile. Dar să presupunem că dorim o soluție mai inteligentă. Să

considerăm trei reprezentări ale tablei de șah mutilate:

- (a) Reprezentare grafică, în care pătratele tablei nu sunt colorate. Această reprezentare nu sugerează în mod direct răspunsul problemei.
- (b) Reprezentare grafică, în care pătratele tablei sunt colorate în alb și negru și lipsesc colțurile negre. Această reprezentare poate sugera un răspuns.
- (c) Reprezentare analitică: Avem 30 pătrate negre și 32 pătrate albe. Această reprezentare, combinată cu faptul suplimentar că fiecare domino trebuie să acopere exact un pătrat alb și unul negru, sugerează în mod direct răspunsul problemei.

Inclusiv pentru rezolvatorii umani, o schimbare a reprezentării poate produce o diferență enormă în ușurința rezolvării problemei. Figura 4.1 prezintă modul de reprezentare a faptelor sugerat de acest exemplu. Linia întreruptă de sus reprezintă procesul abstract de raționare pe care un program trebuie să îl modeleze. Linia solidă de jos reprezintă procesul concret de raționare pe care programul îl realizează. Acest program modelează cu succes procesul abstract în măsura în care, atunci când corespondența de reprezentare înapoi este aplicată, sunt generate faptele finale corespunzătoare. Dacă fie modul de operare al programului sau una din corespondențele de reprezentare nu sunt fidele problemei modelate, faptele finale nu vor fi probabil cele dorite.

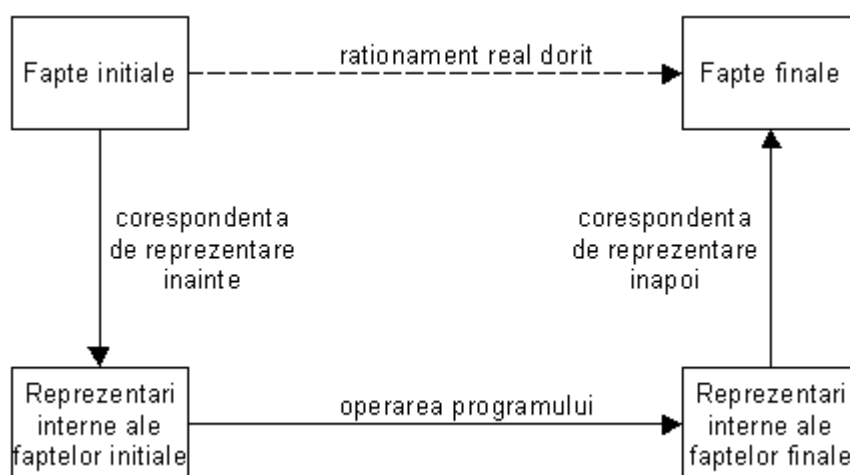


Figura 4.1: Reprezentarea faptelor

4.2. Proprietăți ale reprezentării cunoștințelor

Un sistem bun de reprezentare a cunoștințelor într-un domeniu particular ar trebui să verifice următoarele patru proprietăți:

Adecvare Reprezentațională – abilitatea de a reprezenta toate tipurile de cunoștințe care sunt necesare în acel domeniu.

Adecvare Inferențială – abilitatea de a manipula structurile reprezentationale într-un astfel de mod încât să permită deducerea de structuri noi corespunzând cunoștințelor noi deduse din cele vechi.

Eficiență Inferențială – abilitatea de a incorpora în structura cunoștințelor informație adițională care poate fi folosită pentru focalizarea atenției mecanismului de inferență în direcțiile cele mai promițătoare.

Eficiență Achizițională – abilitatea de a achiziționa cu ușurință informații noi. Cazul cel mai simplu implică inserarea directă de către o persoană a cunoștințelor noi în baza de date. Ideal, programul ar trebui să fie capabil să controleze achiziția de cunoștințe.

Din păcate încă nu a fost identificat un sistem care să optimizeze toate capabilitățile pentru toate tipurile de cunoștințe. Ca rezultat, există mai multe tehnici de reprezentarea cunoștințelor. Multe programe se bazează pe mai multe tehnici în același timp.

Cunoștințe relaționale simple

Cel mai simplu mod de reprezentare a faptelor declarative este ca o mulțime de relații asemănătoare cu cele folosite în sistemele de gestiune a bazelor de date. Motivul pentru care o astfel de reprezentare este simplă este că oferă posibilități de inferență foarte reduse. Dar cunoștințele reprezentate în acest mod pot servi ca date de intrare pentru motoare de inferență mai puternice. Sistemele de gestiune a bazelor de date sunt proiectate tocmai pentru a oferi un suport pentru reprezentarea cunoștințelor relaționale. Astfel, nu vom discuta aici acest tip de structură de reprezentare a cunoștințelor.

Cunoștințe moștenite

Cunoștințele relaționale corespund unei mulțimi de atribute și valori asociate care, împreună, descriu obiectele bazei de cunoștințe. Cunoștințele despre obiecte, atributele acestora și valorile lor nu trebuie să fie neapărat simple. De exemplu, este posibil să extindem reprezentarea de bază sugerată în secțiunea anterioară cu mecanisme de inferență care să opereze pe structura reprezentării. Pentru ca acestea să fie eficiente, structura trebuie proiectată pentru a corespunde mecanismelor de inferență dorite. Una din formele de inferență cele mai utile este **moștenirea proprietăților**, în care elementele unor clase moștenesc atribute și valori din clase mai generale, în care acestea sunt incluse.

Pentru a permite moștenirea proprietăților, obiectele trebuie organizate în clase și clasele trebuie aranjate într-o ierarhie generalizată, reprezentată sub formă de graf, ca în exemplul din

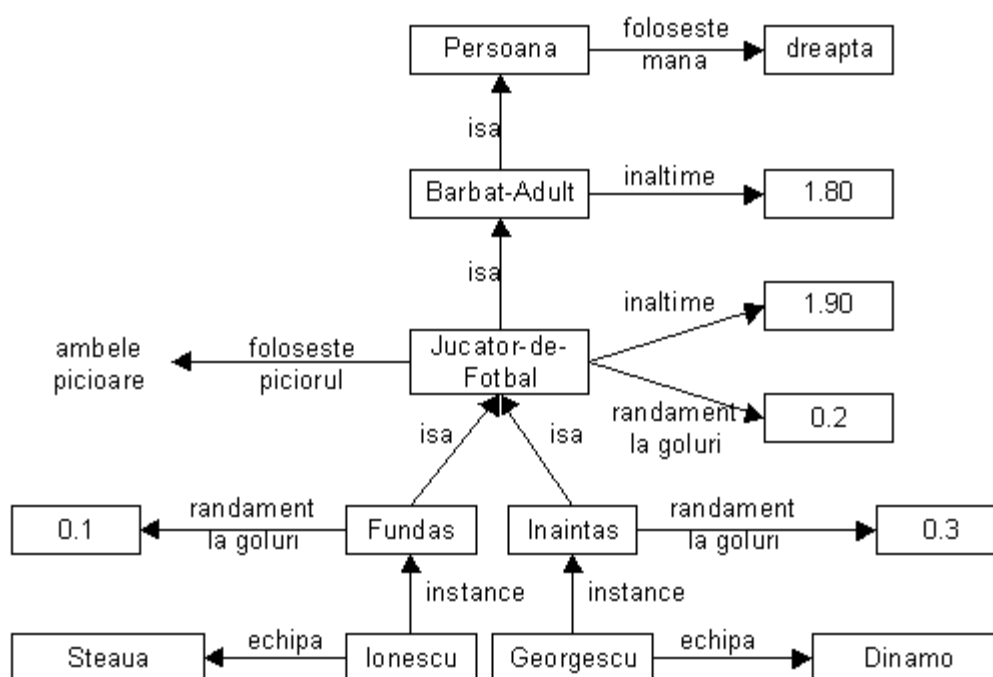


Figura 4.2: Cunoștințe moștenite

Figura 4.2. Liniile reprezintă atribute, dreptunghiurile reprezintă obiecte și valori ale acestora. Aceste valori pot fi, la rândul lor, vazute ca obiecte având atribute și valori, ș.a.m.d. Săgețile de pe linii indică de la un obiect la valoarea acestuia corespunzătoare liniei atributului respectiv. Structura din Figura 4.2 se numește **structură slot-and-filler**, **rețea semantică** sau **colecție de cadre**. În ultimul caz, fiecare cadru individual reprezintă colecția de atribute și valorile asociate unui nod particular. Figura 4.3 prezintă nodul jucătorului de fotbal reprezentat sub formă de cadru.

Jucător-de-Fotbal	
isa:	Bărbat-Adult
șuteur:	(EQUAL picior)
înălțime:	1.90
media-de-goluri	0.252

Figura 4.3: Reprezentarea unui nod sub formă de cadru

Toate obiectele și majoritatea atributelor din Figura 4.3 corespund domeniului jucătorului de fotbal și ca atare sunt fără semnificație practică. Singurele excepții sunt atributul **isa**, utilizat pentru a arăta incluziunea unor clase și atributul **instance**, utilizat pentru a arăta apartenența la o clasă. Aceste două atribute specifice (și general valabile) oferă baza pentru moștenirea proprietăților ca o tehnică de inferență. Utilizând această tehnică, baza de cunoștințe poate accepta atât regăsirea faptelor care au fost explicit memorate cât și a faptelor care pot fi deduse din cele explicit memorate.

O formă idealizată a algoritmului de moștenire a proprietăților este următoarea:

Algoritm: Moștenirea Proprietăților

Pentru a regăsi valoarea V corespunzătoare atributului A al obiectului O :

1. Găsește O în baza de cunoștințe.
2. Dacă există o valoare pentru atributul A , întoarce acea valoare.
3. Altfel, vezi dacă există o valoare pentru atributul **instance**. Dacă nu, întoarce eșec.
4. Altfel, deplasează-te la nodul corespunzător acelei valori și caută o valoare pentru atributul A . Dacă există, întoarce acea valoare.
5. Altfel, repetă ciclul următor până când nu există valoare pentru atributul **isa** sau până când se găsește un răspuns:
 - (a) Citește valoarea atributului **isa** și deplasează-te la acel nod.
 - (b) Vezi dacă există o valoare pentru atributul A . Dacă există, întoarce acea valoare.

Această procedură este simplistă. Nu spune ce trebuie să facem dacă există mai multe valori ale atributelor **instance** și **isa**. Dar descrie mecanismul de bază al moștenirii. Putem aplica această procedură pentru a deduce soluții pentru trei tipuri de întrebări:

- (a) Referitor la valori memorate explicit în baza de date.
- (b) Referitor la valori ale unor atribute care nu sunt definite pentru obiectul analizat, dar sunt definite pentru obiectul către care indică atributul **instance** sau **isa**. De remarcat una din caracteristicile critice ale moștenirii proprietăților, aceea că poate produce valori implicite care nu sunt garantate ca fiind corecte, dar care reprezintă valori acceptabile în absența unor informații mai precise. Să mai remarcăm că valoarea primului atribut pe care îl vom găsi este mai specifică decât valoarea aceluiași atribut, dar asociat unui obiect mai depărtat, pentru că acesta din urmă va fi mai general.
- (c) Referitor la valori ale unor atribute care nu sunt definite pentru obiectul analizat, dar sunt definite pentru obiectul către care indică atributul **instance** sau **isa**, cu deosebirea că atributul nu va avea asociată o valoare, ci o regulă pentru deducerea valorii. În această situație procesul trebuie reluat în mod recursiv pentru a identifica valorile necesare realizării deducției.

Cunoștințe inferențiale

Moștenirea proprietăților este o formă puternică de inferență, dar nu este singura formă utilă. Uneori toată puterea logicii tradiționale este necesară pentru a descrie inferențele necesare. Sigur că bazele de cunoștințe sunt inutile dacă nu există o procedură de inferență care

le poate exploata. Procedura de inferență necesară este una care implementează regulile logice standard ale inferenței. Sunt multe astfel de proceduri, unele care raționează înainte, de la fapte la concluzie, și altele care raționează înapoi, de la concluzii la fapte. Una dintre cele mai utilizate proceduri este rezoluția, care exploatează o demonstrație prin metoda reducerii la absurd.

Cunoștințe procedurale

Un alt tip de cunoștințe, la fel de util ca cele concentrate pe fapte declarative, statice, sunt cunoștințele operaționale, procedurale, care specifică ce să facem și când anume. Cunoștințele procedurale pot fi reprezentate în programe în multe moduri. Cel mai utilizat mod este sub formă de cod în anumite limbaje de programare precum LISP. Mașina utilizează cunoștințele când execută codul pentru realizarea unei sarcini. Din păcate această modalitate de reprezentare a cunoștințelor este slabă relativ la proprietățile de adecvare inferențială (deoarece este foarte dificil de scris un program care poate raționa despre comportarea unui alt program) și eficiență achizițională (deoarece procesul de actualizare și depanare a unor secvențe mari de cod devine greoi).

Cea mai utilizată tehnică pentru reprezentarea cunoștințelor procedurale în programele de inteligență artificială este utilizarea regulilor de producție. Figura 4.4 arată un exemplu de regulă de producție pe care se bazează jucătorul tipic de fotbal.

Dacă:	ești aproximativ la 30 metri
	ești în ofensivă
	portarul advers nu e atent sau e ieșit din poartă
Atunci:	șutează pe poarta adversă

Figura 4.4: Cunoștințe procedurale sub formă de reguli de producție

Regulile de producție, în particular cele care sunt extinse cu informații despre cum trebuie folosite sunt mai procedurale decât alte metode de reprezentare discutate în acest capitol. Dar este dificil să facem o distincție majoră între cunoștințele declarative și cele procedurale. Diferența semnificativă este în modul în care cunoștințele sunt folosite de procedurile care le manipulează.

4.3. Aspecte ale reprezentării cunoștințelor

Înainte să trecem la o discuție a mecansimelor specifice care au fost utilizate la reprezentarea diferitelor tipuri de cunoștințe reale, trebuie să discutăm unele elemente care le deosebesc:

1. Sunt anumite attribute ale obiectelor atât de fundamentale încât apar în aproape fiecare domeniu? Dacă sunt, avem nevoie să ne asigurăm că acestea sunt manevrate în mod

corespunzător în fiecare din mecanismele pe care le vom descrie. Dacă astfel de atribute există, care sunt acestea?

2. Există relații importante care există între atributele obiectelor?
3. La ce nivel trebuie reprezentate cunoștințele? Există o mulțime bună de **primitive** în care toate cunoștințele se pot diviza? Este util să utilizăm astfel de primitive?
4. Cum trebuie reprezentate mulțimile de obiecte?
5. Fiind dat un volum mare de cunoștințe memorate într-o bază de date, cum pot fi accesate părțile relevante, când acestea sunt necesare?

În cele ce urmează vom discuta pe scurt fiecare dintre aceste întrebări.

4.3.1. Atribute importante

Există două atribute care au o semnificație foarte generală, și pe care le-am utilizat: **instance** și **isa**. Aceste atribute sunt importante deoarece acestea sunt baza moștenirii proprietăților. Ceea ce este relevant la acestea nu este numele lor (sunt denumite în mod diferit în diverse sisteme de IA), ci faptul că ele reprezintă incluziunea claselor și apartenența la clase, și că incluziunea claselor este tranzitivă. Aceste relații pot fi reprezentate explicit (de exemplu în sistemele slot-and-filler) sau implicit (în sistemele bazate pe logică, printr-o mulțime de predicate care descriu clase particulare).

4.3.2. Relații dintre atribute

Atributele pe care le folosim pentru a descrie obiecte sunt ele însele entități pe care le reprezentăm. Ce proprietăți independente de cunoștințele specifice pe care le codifică au aceste atribute? Există patru astfel de proprietăți care trebuie menționate aici:

1. Atribute inverse;
2. Existența într-o ierarhie **isa**;
3. Tehnici pentru raționarea cu valori;
4. Atribute cu o singură valoare.

Atribute inverse

În situația utilizării relațiilor dintre două entități, până acum am privit aceste relații ca atribute. În felul acesta, dacă punem accentul pe una dintre entități, vom genera un atribut al acestei entități, având ca valoare cealaltă entitate. Evident, lucrurile pot fi privite și invers, generând

un atribut al celei din urmă entități, având ca valoare prima entitate. În multe cazuri este util să reprezentăm ambele viziuni ale relației. Există două moduri de a face acest lucru.

Prima modalitate este de a folosi o singură reprezentare care ignoră focalizarea pe una dintre entități. Reprezentările logice sunt interpretate de obicei în acest fel. De exemplu, declarația

echipă(Lăcătuș, Steaua)

poate fi în mod egal privită ca o declarație despre Lăcătuș sau despre Steaua. Modul în care această declarație este utilizată depinde de celelalte declarații pe care le conține un sistem.

A doua abordare este de a folosi atribute care focalizează atenția pe una dintre entități, dar să le folosim în perechi, unul fiind inversul celuilalt. În această abordare vom reprezenta informația echipă cu cele două atribute sub forma:

- un atribut asociat cu Lăcătuș:

echipă = Steaua

- un atribut asociat cu Steaua:

jucători = Lăcătuș, ...

Această abordare este luată în considerare în rețelele semantice și în sistemele bazate pe cadre. Este de obicei însoțită de un instrument de achiziție a cunoștințelor care garantează consistența inverselor prin forțarea declarării acestora și verificarea că la fiecare adăugare a unei valori pentru un anumit atribut că valoarea corespunzătoare este adăugată atributului invers.

O ierarhie de atribute isa

După cum există clase de obiecte și submulțimi specializate ale acestor clase, există și atribute și specializări ale acestora. De exemplu, pentru obiectul **persoană** atributul **înălțime** este o specializare a atributului mai general **dimensiune-fizică**, care este, în schimb, o specializare a atributului **atribut-fizic**. Aceste relații generalizare-specializare sunt importante pentru atribute pentru același motiv pentru care sunt importante pentru alte concepte – acestea suportă moștenirea. În cazul atributelor, suportă moștenirea informațiilor despre lucruri precum limitări ale valorilor pe care atributele le pot lua și mecanismele pentru calcularea acestor valori.

Tehnici pentru raționarea cu valori

Uneori valorile atributelor sunt specificate explicit când se crează o bază de cunoștințe. Deseori un sistem de raționare trebuie să raționeze despre valori care nu au fost precizate explicit. Câteva tipuri de informație pot juca un rol în această raționare:

- Informație despre tipul valorii. De exemplu, valoarea atributului înălțime trebuie să fie un număr precizat într-o unitate de măsură.
- Limitări asupra valorii, deseori în termenii unor relații între entități. De exemplu, vârsta unei persoane nu poate fi mai mare decât vârsta fiecăruia dintre părinții săi.

- Reguli pentru calculul valorii, atunci când va fi necesar. Aceste reguli se numesc reguli **înapoi**, sau reguli **if-needed** (dacă este necesar).
- Reguli care descriu acțiuni care ar trebui să fie luate dacă o valoare devine cunoscută la un moment dat. Aceste reguli se numesc reguli **înainte**, sau reguli **if-added** (dacă se adaugă).

Atribute cu o singură valoare

Un tip foarte util de atribut este acela care se garantează că are o singură valoare. De exemplu, un jucător poate avea, în orice moment, o singură înălțime și poate fi membru al unei singure echipe. Dacă un astfel de atribut are deja o valoare și o a doua valoare se atribuie, atunci fie a apărut o schimbare în domeniul problemei, fie a apărut o contradicție în baza de cunoștințe, și care trebuie rezolvată. Sistemele de reprezentare a cunoștințelor au folosit diferite abordări pentru a suporta atributele cu o singură valoare:

- Introducerea unei notații explicite pentru interval de timp. Dacă două valori distincte sunt atribuite aceluiași interval de timp, semnalează o contradicție.
- Presupunerea faptului că singurul interval de timp care ne interesează este cel de acum. Deci, dacă o valoare nouă este atribuită, aceasta va înlocui vechea valoare.
- Absența unui suport explicit. Sistemele bazate pe logică sunt în această categorie. Dar în aceste sisteme constructorii bazelor de cunoștințe pot adăuga axiome care afirmă că dacă un atribut are o valoare atunci se știe că nu are nici o altă valoare.

4.3.3. Alegerea granularității reprezentării

Indiferent de tipul de reprezentare pe care îl alegem, este necesar să putem răspunde la întrebarea “La ce nivel de detaliu ar trebui reprezentată lumea?” Un alt mod în care se pune această întrebare este “Care ar trebui să fie primitivele noastre?”. Ar trebui să fie un număr mic de primitive de nivel scăzut, sau ar trebui să fie un număr mare de primitive care să acopere un interval mare de granularități?

Avantajul principal al convertirii tuturor declarațiilor dintr-o reprezentare în funcție de o mulțime mică de primitive este că regulile utilizate pentru realizarea deducțiilor din acele cunoștințe trebuie să fie scrise doar pentru acea mulțime de primitive, nu pentru diferitele moduri în care cunoștințele au apărut. Există o serie de programe de IA care se bazează pe baze de cunoștințe descrise în funcție de un număr mic de primitive de nivel scăzut.

Există câteva argumente împotriva utilizării primitivelor de nivel scăzut. Primul este că fapte simple de nivel înalt ar putea să ceară o capacitate de memorare mare dacă sunt scrise în funcție de primitive de nivel scăzut. O mare parte din această memorie este efectiv pierdută,

deoarece reprezentarea de nivel scăzut a unui concept de nivel înalt va apare de mai multe ori, câte o dată pentru fiecare referire a conceptului respectiv.

O a doua problemă este aceea că inițial cunoștințele sunt prezentate sistemului într-o formă de nivel relativ înalt, cum ar fi limbajul natural. În această situație trebuie depus un efort consistent pentru convertirea cunoștințelor în formă primitivă. În plus, această reprezentare primitivă detaliată ar putea să nu fie necesară. Atât în înțelegerea limbajului cât și în interpretarea lumii, apar multe lucruri care mai târziu se divedesc irelevante. Din motive de eficiență, este de dorit să memorăm aceste lucruri la un nivel cât de înalt posibil, și apoi să analizăm în detaliu doar acele informații care par importante.

A treia problemă este că în multe domenii nu este clar care ar trebui să fie primitivele. Si chiar în domenii în care ar putea exista o mulțime de primitive evidentă, ar putea să nu existe suficientă informație în fiecare utilizare a construcțiilor de nivel înalt astfel încât să fie posibilă convertirea acestora în componente primitive. În orice caz, nu există nici un mod în care să evităm reprezentarea faptelor într-o varietate de granularități.

Să luăm un exemplu, acela al reprezentării relațiilor de rudenie. Există o mulțime clară de primitive: **mamă**, **tată**, **fiu**, **fiică**, și posibil, **frate** și **soră**. Să presupunem că ni se spune că Maria este verișoara Ioanei. O încercare de a descrie această relație în funcție de primitive ar putea produce una din interpretările următoare:

- Maria = fiică(frate(mamă(Ioana)))
- Maria = fiică(soră(mamă(Ioana)))
- Maria = fiică(frate(tată(Ioana)))
- Maria = fiică(soră(tată(Ioana)))

Dacă nu știm că Maria este o persoană de genul feminin, atunci mai avem încă patru posibilități. Cum în general nu putem alege una dintre aceste reprezentări, singura posibilitate este să reprezentăm acest fapt folosind relația neprimitivă **văr** (**verișoară**).

O altă posibilitate de a rezolva această problemă este să schimbăm primitivele. Am putea să folosim mulțimea: **părinte**, **copil**, **frate**, **bărbat**, **femeie**. (Să remarcăm că aici primitiva **frate** nu presupune un gen implicit, ci se va folosi pentru reprezentarea atât a relației frate cât și a relației soră.) Astfelm faptul că Maria este verișoara Ioanei se reprezintă astfel:

- Maria = copil(frate(părinte(Ioana)))

Dar cum primitivele folosite încorporează oarecari generalizări care pot sau pot să nu fie potrivite. Concluzia este că chiar în domeniile foarte simple, mulțimea corectă de primitive nu este evidentă.

Problema alegerii granularității corecte a reprezentării unui bagaj de cunoștințe nu este ușoară. Cu cât mai scăzut este nivelul de reprezentare, cu atât mai puțină inferență se cere

pentru a raționa, dar cu atât mai multă inferență se cere pentru a converti reprezentarea din limbajul natural și mai mult spațiu este necesar pentru memorare.

4.3.4. Reprezentarea mulțimilor de obiecte

Un aspect important este acela de a putea reprezenta mulțimi de obiecte. Un motiv este acela că există proprietăți valide pentru o mulțime de obiecte, dar care nu sunt valide pentru un obiect oarecare. De exemplu, să considerăm afirmația “Vorbitori de limbă engleză sunt peste tot în lume”. Singurul mod de a reprezenta acest fapt este să reprezentăm mulțimea vorbitorilor de engleză și să îi asociem acesteia proprietatea din afirmație, deoarece, în mod evident, nici un vorbitor de engleză particular nu poate fi găsit peste tot în lume. Un al doilea motiv pentru care este utilă reprezentarea mulțimilor este situația în care o proprietate este valabilă pentru toate (sau aproape toate) obiectele unei mulțimi. În acest caz este mai eficient să asociem proprietatea cu mulțimea obiectelor, nu explicit cu fiecare obiect în parte. De exemplu, am făcut acest lucru în cazul structurilor slot-and-filler.

Există trei moduri în care se pot reprezenta mulțimile. Cel mai simplu este prin asocierea unui nume, așa cum am procedat în Secțiunea 4.2, cu exemplul de rețea semantică. Această reprezentare simplă permite asocierea predicatelor cu mulțimi, dar nu oferă informații despre mulțimea reprezentată. Nu ne spune, de exemplu, cum să determinăm dacă un element este membru al mulțimii sau nu.

Există două moduri de definire a unei mulțimi și a elementelor acesteia. Primul mod este de a preciza lista membrilor. O astfel de specificare se numește **definiție prin extensiune** (*extensional definition*). Al doilea mod este de a oferi o regulă care, când un obiect este evaluat, întoarce rezultat adevărat sau fals în funcție de apartenența obiectului la mulțime. O astfel de regulă se numește **definiție prin intensiune** (*intensional definition*).

Aceste două reprezentări diferă prin aceea că ele nu corespund în mod necesar una alteia. Unei definiții prin extensiune i se pot asocia mai multe definiții prin intensiune. Astfel, dacă este trivial să verificăm dacă două mulțimi reprezentate prin extensiune sunt identice, acest lucru ar putea fi foarte dificil în cazul reprezentării prin intensiune.

Reprezentările prin intensiune au două proprietăți care lipsesc reprezentărilor prin extensiune. Prima este că ele pot fi folosite pentru a descrie mulțimi infinite și mulțimi ale căror elemente nu sunt cunoscute în întregime. A doua proprietate este că putem permite acestora să depindă de parametri care se pot schimba, cum ar fi timpul sau localizarea în spațiu. Mulțimea reprezentată în acest fel se va schimba odată cu schimbarea valorilor acestor parametri. De exemplu, fie afirmația “Președintele Statelor Unite era Republican”, făcută în contextul în care președintele de astăzi este Democrat. Această afirmație are două sensuri. Primul este că persoana care este astăzi președinte, la un moment dat în trecut a fost Republican. Acest sens este evident cu o reprezentare prin extensiune a noțiunii “Președintele Statelor Unite”. Al doilea sens este că la un moment dat a existat o persoană care era președinte și care era Republican.

Pentru a reprezenta noțiunea “Președintele Statelor Unite” în această interpretare, avem nevoie de o descriere prin intensiune care să depindă de timp.

4.3.5. Identificarea celor mai potrivite structuri

Problema pe care o ridicăm acum este cum să localizăm structuri de cunoștințe potrivite care au fost memorate anterior. De exemplu să presupunem că avem un script (o descriere a unei clase de evenimente în funcție de context, participanți și subevenimente) care descrie secvența tipică de evenimente dintr-un restaurant. Acest script ne va permite să analizăm textul:

Marian a mers aseară la Casa Universtarilor. A comandat un grătar mare, a plătit și a plecat.

și să răspundem cu “da” la întrebarea “Marian a luat cina aseară?”

Să remarcăm că nicăieri nu este precizat explicit faptul că Marian a mâncat ceva. Dar faptul că cineva, când merge la restaurant la mâncă ceva, este conținut în scriptul restaurant. Dacă știm să folosim scriptul restaurant, vom putea răspunde la întrebare. Pentru a putea fi eficient, sistemul va trebui să aiba multe scripturi, despre multe situații. Dar nicăieri în textul nostru nu este precizat cuvântul “restaurant”. Atunci cum va selecta sistemul scriptul restaurant?

Pentru a avea acces la structura corectă pentru descrierea unei stiații particulare, este necesar să rezolvăm problemele următoare:

1. Cum să realizăm o selecție inițială a celei mai potrivite structuri.
2. Cum să completăm detaliile relevante din situația curentă.
3. Cum să identificăm o structură mai bună, dacă cea aleasă se dovedește nepotrivită.
4. Ce să facem dacă nici o structură nu este potrivită.
5. Când să creăm și când să ne reamintim o structură nouă.

Nu există o metodă general valabilă pentru rezolvarea tuturor acestor probleme. Unele tehnici de reprezentare a cunoștințelor rezolvă câteva dintre acestea. În continuare vom discuta câteva soluții pentru două probleme: prima și a treia.

Selectarea structurii inițiale

Selectarea structurii care să corespundă unei situații particulare în rezolvarea problemei este o problemă dificilă. Există câteva moduri în care această selecție se poate face.

- Indexează structurile folosind cuvinte semnificative din limbajul natural, care pot fi folosite pentru a le descrie. De exemplu, vom asocia fiecare verb cu o structură care îi descrie

semnificația. Această abordare ar putea fi nepotrivită chiar pentru selectarea structurilor simple, deoarece multe verbe au mai multe semnificații absolut distincte. De asemenea, această abordare este utilă doar în situația în care există o descriere a problemei de rezolvat în limbaj natural.

- Consideră fiecare concept major ca un pointer la toate structurile (precum scripturi) în care ar putea fi implicat. Aceasta poate produce mai multe mulțimi de structuri posibile. Vom selecta acea structură care aparține intersecției mulțimilor corespunzătoare tuturor conceptelor menționate de problema. Dezavantajul este acela că dacă descrierea problemei conține un număr fie și redus de concepte străine de domeniul problemei, atunci intersecția structurilor asociate va fi vidă. Aceasta s-ar putea întâmpla dacă am fi menționat că “Marian a mers aseară cu bicicleta la Casa Universitarilor”. De asemenea, ar putea fi necesară o putere de calcul extrem de mare pentru determinarea tuturor mulțimilor de structuri și a intersecției acestora. Dacă acest calcul se poate face în paralel, atunci timpul cerut pentru producerea unui răspuns devine rezonabil.
- Localizează un element cheie major în descrierea problemei și folosește-l pentru selectarea structurii inițiale. Pe măsură ce apare un alt element cheie, folosește-l pentru rafinarea selecției inițiale sau pentru realizarea uneia complet noi. Dificultatea este că în unele situații nu se poate identifica ușor un element cheie major. O a doua problemă este că este necesar să anticipăm care sunt elementele cheie care vor fi importante și care nu. Dar importanța relativă a elementelor cheie se poate schimba esențial de la o situație la alta.

Nici una din aceste propuneri nu pare să fie răspunsul complet la problemă. Deseori de dovedește că, cu cât mai complexe sunt structurile cunoștințelor, cu atât mai dificil este să identificăm cea mai potrivită structură.

Revizuirea selecției când este necesar

Odată ce am identificat o structură potrivită, trebuie să încercăm să realizăm o potrivire mai detaliată a ei cu problema de rezolvat. În funcție de reprezentările folosite, detaliile procesului de potrivire vor varia (legarea variabilelor de obiecte, sau compararea valorilor atributelor). Dacă pot fi identificate valori care satisfac restricțiile impuse de structura de cunoștințe, acestea sunt plasate în structură. Dacă nu se pot identifica valori potrivite, trebuie selectată o altă structură. Modul în care precedentul proces de potrivire a eșuat poate oferi informații utile despre ce anume să încercă în continuare. Procesul instanțierii unei structuri într-o situație particulară nu se desfășoară ușor. Când procesul se împotmolește, este deseori necesar să abandonăm efortul și să o luăm de la capăt. Se pot întâmpla unul din următoarele lucruri:

- Selectează fragmente din structura curentă, care corespund situației și potrivește-le cu alternativele posibile. Alege cea mai bună potrivire. Dacă structura coruentă a fost aproape

potrivită, cea mai mare parte a efortului realizat pentru consturirea substructurilor se conservă.

- Produce o scuză pentru eșecul structurii curente și continuă să o folosești. O parte a structurii ar trebui să conțină informații despre caracteristicile pentru care scuzele sunt acceptabile. De asemenea, există euristici generale precum faptul că o structură este mai probabil să fie potrivită dacă o anumită caracteristică lipsește, decât dacă o caracteristică necorespunzătoare este prezentă. De exemplu, un om cu un singur picior este mai plauzibil decât un om cu coadă.
- Caută legături specifice între structuri pentru a sugera noi direcții în care să se exploreze.
- Dacă structurile de cunoștințe sunt memorate într-o ierarhie **isa**, traversează ierarhia în sus până când se găsește o structură suficient de generală și care nu contrazice evidența. Fie utilizează această structură, dacă este suficient de specifică pentru a oferi cunoștințele necesare, fie încearcă să creezi o structură nouă exact sub cea găsită.

4.4. Problema cadrelor

Am văzut câteva metode de reprezentare a cunoștințelor care ne permit să formăm descrieri complexe ale stărilor pentru un program de căutare. O altă problemă de discutat este reprezentarea eficientă a **șirurilor de stări** care provin dintr-un proces de căutare. Pentru probleme complexe structurate greșit aceasta este o problemă serioasă.

Fie un anumit domeniu al problemei. Există multe obiecte și relații dintre acestea, și descrierea unei stări trebuie să includă multe fapte. O strategie este să memorăm descrierea stării ca o listă de fapte. Dar, în timpul procesului de căutare, aceste liste pot deveni foarte lungi. Multe dintre fapte nu se vor schimba, și totuși fiecare fapt va fi reprezentat la fiecare nod, și vom consuma inutil memorie. În plus, vom pierde timp cu crearea nodurilor și copierea faptelor. Problema reală este însă de a deduce care sunt faptele care ar trebui să difere de la un nod la altul.

Problema reprezentării faptelor care se schimbă, precum și a celor care nu se schimbă, se numește **problema cadrelor**. În unele domenii partea dificilă este să reprezentăm faptele. În altele, problema este identificarea celor în schimbare. Pentru a suporta un raționament bazat pe fapte în schimbare, unele sisteme utilizează o mulțime de axiome, numite **axiomele cadrelor**, care descriu lucrurile care nu se schimbă atunci când un operator este aplicat stării n pentru a produce starea $n+1$. Lucrurile care se schimbă trebuie precizate ca parte a operatorului.

Putem reprezenta stările în schimbare dacă începem cu descrierea stării inițiale și realizarea schimbărilor la acea descriere, așa cum sunt ele indicate de regulile aplicate. Aceasta rezolvă problema pierderii de memorie și timp la copierea informației de la un nod la următorul. Și funcționează perfect până în momentul în care trebuie să recurgem la backtracking, moment în

care trebuie să refacem anumite stări anterioare (cu excepția cazului în care schimbările pot fi ignorate). Dar cum știm ce schimbări în descrierea stării trebuie refăcute? Există două moduri în care putem rezolva problema:

- Nu modificăm deloc starea inițială. La fiecare nod memorăm o indicație asupra schimbărilor care trebuie efectuate. Când este necesar să ne referim la descrierea stării curente, analizăm descrierea nodului inițial și indicațiile memorate la nodurile pe drumul de la nodul inițial la cel curent. Astfel, backtracking-ul este foarte ușor dar referirea la descrierea stărilor destul de complexă.
- Modifică descrierea stării inițiale conform necesităților, dar memorează la fiecare nod o indicație despre operațiile necesare dacă în cazul backtracking-ului peste acel nod. Apoi, oricând backtracking-ul este necesar, verifică fiecare nod de-a lungul drumului și realizează operațiile indicate în descrierea stării.

Uneori aceste soluții nu sunt suficiente. Am putea avea nevoie de informații relativ la o situație anterioară. Într-un astfel de caz vom folosi **variabile de stări** care vor memora aceste informații de la o stare la alta.

Nu există un răspuns simplu la problema reprezentării cunoștințelor sau la problema cauzelor. Dar este important să ni le amintim când avem în vedere strategii de căutare, deoarece reprezentarea cunoștințelor și procesul de căutare depind puternic între ele.

Capitolul 5

Reprezentarea cunoștințelor folosind reguli

În acest capitol vom discuta despre utilizarea regulilor la codificarea cunoștințelor, problemă importantă deoarece sistemele de raționare bazate pe reguli au jucat un rol foarte important în evoluția sistemelor de inteligență artificială.

5.1. Cunoștințe procedurale și declarative

Așa cum am vazut, una din modalitățile de reprezentare a cunoștințelor se bazează pe utilizarea declarațiilor logice, împărțite în fapte și reguli. **Reprezentarea declarativă** este aceea reprezentare în care sunt specificate cunoștințe, dar nu este specificat modul de utilizare a acelor cunoștințe. Pentru a utiliza o reprezentare declarativă trebuie să o extindem cu un program care specifică ce este de făcut cu cunoștințele și cum anume. De exemplu, o mulțime de declarații logice poate fi combinată cu un demonstrator prin rezoluție al teoremelor pentru a obține un program complet de rezolvare a problemelor.

Pe de altă parte, declarațiile logice pot fi văzute și ca un **program**, nu ca **date** ale unui program. În acest declarațiile de implicație (regulile) definesc căi de raționare (în spațiul cunoștințelor), iar declarațiile atomice (faptele) oferă puncte de plecare (sau, dacă raționăm înapoi, puncte de terminare) ale acelor căi. Aceste căi de raționare definesc căile de execuție posibile ale programului cam în același mod în care structurile de control tradiționale, precum *if-then-else*, definesc căile de execuție în programele tradiționale. Cu alte cuvinte, putem privi declarațiile logice ca reprezentări procedurale ale cunoștințelor. **Reprezentarea procedurală** este aceea reprezentare în care informația de control care este necesară pentru utilizarea cunoștințelor se consideră că este încorporată în cunoștințe. Pentru a utiliza o reprezentare procedurală avem nevoie să o extindem cu un interpretor care poate executa instrucțiunile date în cunoștințe.

Este important de notat că deși o reprezentare procedurală încorporează informația de control în baza de cunoștințe, acest lucru este valabil doar în măsura în care interpretorul cunoștin-

țelor este capabil să recunoască această informație de control.

5.2. Programare logică

Programarea logică este o paradigmă de programare în care declarațiile logice sunt tratate ca programe. Astăzi există câteva sisteme de programare logică, cel mai important dintre acestea fiind PROLOG. Un program PROLOG este descris ca o serie de declarații logice, fiecare dintre ele fiind o clauză Horn. O clauză Horn este o clauză care are cel mult un literal pozitiv. Astfel, p , $\neg p \vee q$ și $p \rightarrow q$ sunt clauze Horn. Ultima nu pare clauză, și pare să aibă doi literalii pozitivi. Dar știm că orice expresie logică poate fi convertită la forma clauzală. Dacă facem acest lucru, clauza produsă este $\neg p \vee q$.

Faptul că programele PROLOG sunt compuse doar din clauze Horn și nu din expresii logice arbitrare are două consecințe importante. Prima este că din cauza reprezentării uniforme se poate scrie un interpretor simplu și eficient. A doua consecință este mai importantă. Logica clauzelor Horn este decidabilă.

Structura de control impusă unui program PROLOG de către interpretorul PROLOG este bazată pe declarații logice de două tipuri: fapte și reguli. Intrarea unui program este o concluzie care trebuie demonstrată. Se aplică raționamentul înapoi pentru a demonstra concluzia fiind date declarațiile din program. Programul este citit de sus în jos, de la dreapta la stânga, iar căutarea este de tipul Depth-First și se realizează folosind backtracking.

Figurile 5.1 și 5.2 arată un exemplu de bază de cunoștințe simplă reprezentată în notăție logică standard și în PROLOG. Ambele reprezentări conțin două tipuri de declarații, **faptele**, care conțin doar constante, și **regulile**, care conțin și variabile. Faptele reprezintă declarații despre obiecte specifice. Regulile reprezintă declarații despre clase de obiecte.

$$\begin{aligned} \forall x : \text{animal}(x) \wedge \text{mic}(x) &\rightarrow \text{animal_de_apartament}(x) \\ \forall x : \text{pisică}(x) \vee \text{câine}(x) &\rightarrow \text{animal}(x) \\ \forall x : \text{pechinez}(x) &\rightarrow \text{câine}(x) \wedge \text{mic}(x) \\ &\text{pechinez}(\text{puffy}) \end{aligned}$$

Figura 5.1: O reprezentare în logică

Să notăm câteva diferențe sintactice între reprezentările logice și în PROLOG:

1. În logică variabilele sunt cuantificate în mod explicit. În PROLOG, cuantificarea este implicită prin modul în care variabilele sunt interpretate. Distincția dintre variabile și constante se face în PROLOG prin modul de scriere al acestora: numele variabilelor începe cu majusculă, și numele constantelor începe cu minusculă sau cu număr.

```

animal_de_apartament(X) :- animal(X), mic(X).
animal(X) :- pisică(X).
animal(X) :- câine(X).
câine(X) :- pechinez(X).
mic(X) :- pechinez(X).
pechinez(puffy).

```

Figura 5.2: O reprezentare în PROLOG

2. În logică există simboluri explicite pentru *și* (\wedge) și *sau* (\vee). În PROLOG există un simbol explicit pentru *și* ($,$), dar nu este nici unul pentru *sau*. În loc de aceasta, disjuncțiile trebuie reprezentate ca o listă de declarații alternative, oricare dintre ele putând oferi baza pentru concluzie.
3. În logică implicațiile de forma “ p implică q ” sunt scrise sub forma $p \rightarrow q$. În PROLOG, aceeași implicație se scrie invers, sub forma $q :- p$. Această formă este naturală în PROLOG, deoarece interpretorul lucrează întotdeauna înapoi, de la o concluzie spre fapte, și această formă permite fiecărei reguli să înceapă cu componenta care trebuie să se potrivească cu concluzia. Această componentă se numește *capul* regulii.

Primele două diferențe provin în mod natural din faptul că programele PROLOG sunt mulțimi de clauze Horn care au fost transformate după cum urmează:

1. Dacă clauza Horn nu conține literal negativi (adică conține exact un literal pozitiv), atunci las-o așa cum este.
2. Altfel, rescrie clauza Horn ca o implicație, combinând toți literalii negativi în ipoteza implicației și lasă singurul literal pozitiv (dacă există) în concluzia implicației.

Această procedură face ca o clauză, care inițial era formată dintr-o disjuncție de literalii (dintre care toți cu excepția unuia sunt negativi) să fie transformată într-o singură implicație a cărei ipoteză este o conjuncție de literalii (care acum sunt pozitivi). Mai mult, într-o clauză toți literalii sunt implicit cuantificați universal. Dar când aplicăm această transformare, orice variabilă care a apărut în formă negativă și care acum apare în ipoteza implicației, devine cuantificată existențial, în timp ce variabilele din concluzie sunt în continuare cuantificate universal. De exemplu, clauza PROLOG

$$P(x) : \neg Q(x, y)$$

este echivalentă cu expresia logică

$$\forall x : \exists y : Q(x, y) \rightarrow P(x)$$

O diferență esențială între reprezentările logice și în PROLOG este că reprezentarea PROLOG are o strategie de control fixă, și deci declarațiile în PROLOG definesc o cale de căutare particulară la orice întrebare. În contrast, declarațiile logice definesc doar mulțimea de răspunsuri pe care le justifică, și nu spun nimic relativ la modul în care trebuie ales răspunsul corect, dacă sunt mai multe.

Strategia de control de bază în PROLOG este simplă. Începem cu declarația unei probleme, care poate fi privită ca o concluzie de demonstrat. Căutăm declarații care pot demonstra concluzia. Considerăm fapte, care demonstrează concluzia direct și considerăm orice regulă al cărei cap se potrivește cu concluzia. Pentru a decide dacă un fapt sau o regulă se poate aplica problemei, invocă o procedură de unificare standard. Raționează înapoi de la concluzie până când se găsește un drum care se termină și care conține declarații din program. Drumurile sunt avute în vedere utilizând o strategie Depth-First și backtracking. La fiecare moment consideră opțiunile în ordinea în care acestea apar în program. Dacă o concluzie are mai multe părți conjunctive, demonstrează aceste părți în ordinea în care apar, propagând legările variabilelor așa cum sunt determinate în timpul unificării.

Se pot spune multe despre programarea în stil PROLOG față de programarea în stil LISP. Un mare avantaj al programării logice este că programatorul trebuie doar să specifice reguli și fapte, deoarece mecanismul de căutare este încorporat în interpretor. Dezavantajul este că strategia de control este fixată. Este foarte dificil de scris cod PROLOG care să folosească alte strategii de căutare, și este chiar mai dificil să de aplicat cunoștințe specifice domeniului problemei pentru a limita căutarea. PROLOG permite controlarea căutării prin intermediul unui operator nelogic numit **tăietură**. O tăietură se poate include într-o regulă pentru a specifica un punct peste care nu se va reveni înapoi cu backtracking.

5.3. Raționare înainte și înapoi

Scopul unei proceduri de căutare este de a descoperi un drum prin spațiul problemei, de la o configurație inițială la o stare finală. Chiar dacă PROLOG caută doar de la o stare finală, o astfel de căutare poate fi realizată în două direcții:

- Înainte, de la stările inițiale către stările finale;
- Înapoi, de la stările finale la stările inițiale.

Modelul sistemului de producție al procesului de căutare ne permite să vedem raționarea înainte și înapoi ca procese simetrice. Să considerăm problema rezolvării unei configurații particulare a problemei 8-puzzle. Regulile care trebuie folosite pentru rezolvarea acestui puzzle sunt scrise în Figura 5.3. Folosind aceste reguli putem încerca să rezolvăm configurația din Figura 2.6 într-unul din următoarele două moduri:

Raționare înainte, de la stările inițiale. Începem construirea unui arbore de șiruri de mutări care pot fi soluții ale problemei, prin considerarea configurației/configurațiilor inițiale ca rădăcină a arborelui. Se generează următorul nivel al arborelui prin identificarea tuturor regulilor a căror parte **stângă** se potrivește cu nodul rădăcină și se folosește partea dreaptă pentru crearea configurațiilor noi. Se generează următorul nivel prin considerarea fiecărui nod generat la nivelul precedent și prin aplicarea tuturor regulilor a căror parte stângă se potrivește cu acesta. Procesul continuă până când se generează o configurație care se potrivește cu o stare finală.

Raționare înapoi, de la stările finale. Începem construirea unui arbore de șiruri de mutări care pot fi soluții ale problemei, prin considerarea configurației/configurațiilor finale ca rădăcină a arborelui. Se generează următorul nivel al arborelui prin identificarea tuturor regulilor a căror parte **dreaptă** se potrivește cu nodul rădăcină. Acestea sunt toate regulile care, dacă ar fi aplicate, ar genera stările pe care le dorim. Se folosește partea stângă pentru crearea configurațiilor de la nivelul doi al arborelui. Se generează următorul nivel prin considerarea fiecărui nod generat la nivelul precedent și prin aplicarea tuturor regulilor a căror parte dreaptă se potrivește cu acesta. Procesul continuă până când se generează o configurație care se potrivește cu o stare inițială. Această metodă de raționare înapoi este deseori numită **raționare direcționată de scop**.

Să presupunem că pătratele din puzzle sunt numerotate astfel:

1	2	3
4	5	6
7	8	9

- Pătratul 1 este gol și Pătratul 2 conține piesa $n \rightarrow$ Pătratul 2 este gol și Pătratul 1 conține piesa n
- Pătratul 1 este gol și Pătratul 4 conține piesa $n \rightarrow$ Pătratul 4 este gol și Pătratul 1 conține piesa n
- Pătratul 2 este gol și Pătratul 1 conține piesa $n \rightarrow$ Pătratul 1 este gol și Pătratul 2 conține piesa n

Figura 5.3: O parte din regulile folosite la rezolvarea problemei 8-puzzle

Să remarcăm că aceleași reguli pot fi folosite atât pentru raționarea înainte, cât și pentru raționarea înapoi. Pentru a raționa înainte, partea stângă a regulilor (precondiția) se potrivește cu starea curentă și partea dreaptă (rezultatul) se folosește la generarea stărilor noi, până când se atinge un nod final. Pentru a raționa înapoi, partea dreaptă a regulilor (rezultatul) se potrivește cu starea curentă și partea stângă (precondiția) se folosește la generarea stărilor noi,

reprezentând noi stări finale care trebuie atinse. Aceasta continuă până când unul din aceste noduri finale se potrivește cu un nod inițial.

În cazul problemei 8-puzzle, diferența dintre raționarea înainte și raționarea înapoi nu este mare. Aproximativ același număr de drumuri va fi explorat în ambele cazuri. Acest lucru nu este însă adevărat. În funcție de topologia spațiului problemei, poate fi semnificativ mai eficient să căutăm într-una din direcții și nu în cealaltă.

Patru factori influențează decizia relativ la care dintre cele două moduri de raționare este mai bun:

1. **Există mai multe stări de start sau stări finale?** Am dori să ne deplasăm de la numărul mai mic de stări la numărul mai mare de stări, în felul acesta mai ușor de găsit.
2. **În care dintre direcții factorul de ramificare** (numărul mediu de noduri care pot fi atinse direct de la un singur nod) **este mai mare?** Am dori să ne deplasăm în direcția în care factorul de ramificare este mai mic.
3. **Programului i se va cere să justifice procesul de raționare?** Dacă da, este important să ne deplasăm în direcția care corespunde într-o mai mare măsură modului de gândire al utilizatorului.
4. **Ce tip de evenimente va declanșa procesul de rezolvare a problemei?** Dacă este vorba de apariția unui fapt nou, atunci raționarea înainte este rezonabilă. Dacă este o interogare relativ la care se cere un răspuns, atunci raționarea înapoi este mai naturală.

Pentru lămurirea acestor considerații vom vedea câteva exemple.

Exemplul 1. Pare mai ușor să ne deplasăm dintr-un loc nefamiliar către casă, decât de acasă către un loc nefamiliar. Factorul de ramificare este aproximativ același. Dar, pentru problema găsirii drumului potrivit, sunt mult mai multe locuri pe care le putem asocia cu noțiunea “acasă” decât locuri pe care să le putem asocia cu locul nefamiliar. Orice loc din care știm cum să ajungem acasă poate fi considerat “acasă”. Concluzia este că, dacă nodul de start este acasă și nodul final este locul nefamiliar, ar trebui să ne planificăm drumul folosind raționamentul înapoi.

Exemplul 2. Problema integrării numerice. Spațiul problemei este un set de formule, unele dintre care conțin expresii integrale. Starea de start este o formulă dată, care conține unele expresii integrale. Starea finală este o formulă echivalentă cu formula inițială și care nu conține nici o expresie integrală. Deci avem o singură stare inițială, ușor de identificat și un număr uriaș de stări finale posibile. Astfel, pentru a rezolva problema trebuie să raționăm înainte, de la starea inițială unică către o stare finală din mulțimea stărilor finale posibile, în loc să plecăm de la expresii arbitrare fără integrale, să folosim regulile de diferențiere și să încercăm să generăm expresia integrală dată.

În situațiile de mai sus elementul decisiv în deciderea direcției de raționare a fost numărul relativ de stări inițiale și stări finale, iar factorul de ramificare a fost aproximativ același. Când factorul de ramificare nu este același, trebuie luat în calcul.

Exemplul 3. Problema demonstrării automate a teoremelor. Starea finală este teorema care trebuie demonstrată. Stările inițiale sunt o mulțime restrânsă de axiome. Nici una dintre mulțimile acestor două tipuri de stări nu este semnificativ mai mare decât cealaltă. În schimb, să considerăm factorul de ramificare în fiecare din cele două direcții. Dintr-un număr mic de axiome putem deduce un număr uriaș de teoreme. În sens invers, de la acest număr uriaș de teoreme trebuie să se revină la mulțimea inițială de axiome. Deci factorul de ramificare este semnificativ mai mare în cazul raționamentului înainte față de raționamentul înapoi. Aceasta sugerează că este mai bine să raționăm înapoi când dorim să demonstrăm teoreme. Acest lucru a fost realizat deja atât de matematicieni cât și de proiectanții programelor de demonstrare automată a teoremelor.

Exemplul 4. Sisteme expert în medicină. Al treilea factor care influențează direcția de raționare este necesitatea de a genera justificări coerente ale procesului deductiv. Acest lucru este valabil în cazul programelor care realizează sarcini foarte importante, ca de exemplu sistemele expert în medicină. Medicii acceptă cu greu diagnosticul oferit de un program dacă acesta nu își poate explica raționamentul într-un mod satisfăcător pentru medic.

Multe dintre tehnicile de căutare descrise în Capitolul 3 pot fi folosite pentru a căuta fie înainte, fie înapoi. Prin descrierea procesului de căutare ca aplicare a unei mulțimi de reguli de producție, este ușor să se descrie algoritmi de căutare specifici fără referire la direcția în care are loc căutarea.

Putem, de asemenea, să căutăm simultan atât înainte, de la starea inițială, cât și înapoi, de la starea finală, până când cele două drumuri se întâlnesc. Această strategie se numește **căutare bidirecțională**. Rezultate empirice sugerează că pentru căutarea oarbă această strategie de tipul divide-et-impera este eficace. Alte rezultate sugerează că acest lucru nu mai este valabil în cazul căutărilor euristice, informate.

Căutarea bidirecțională poate fi ineficace deoarece este posibil ca cele două drumuri, cel înainte și cel înapoi, să treacă unul pe lângă celălalt fără să se întâlnească. Sigur, este de preferat să realizăm pașii înainte și înapoi în măsura în care aceștia se dovedesc cei mai profitabili pentru problema în cauză. Multe aplicații de IA au fost scrise folosind o combinație de raționare înainte și înapoi, și multe medii de programare de IA oferă suport explicit pentru un astfel de raționament hibrid.

Deși în principiu același set de reguli se poate folosi pentru raționarea în ambele direcții, în practică este util să definim două clase de reguli, fiecare codificând un anumit tip de cunoștințe:

Reguli înainte, care codifică cunoștințe despre modul în care se răspunde la anumite configurații de intrare.

Reguli înapoi, care codifică cunoștințe despre modul în care se obțin anumite rezultate.

Separând regulile în aceste două clase, practic adăugăm fiecareia o informație despre modul în care trebuie folosită la rezolvarea problemelor.

5.3.1. Sisteme de reguli cu înlănțuire înapoi

Sistemele de reguli cu înlănțuire înapoi, dintre care un exemplu este PROLOG, sunt potrivite pentru rezolvarea problemelor direcționată de scop. De exemplu, un sistem e interogare ar folosi înlănțuirea înapoi pentru a raționa și a răspunde la întrebările utilizatorului.

În PROLOG regulile sunt limitate la clauze Horn. Aceasta permite indexarea rapidă, deoarece toate regulile aplicabile la deducerea unui anumit fapt au același cap. Regulile sunt potrivite folosind procedura de unificare. Unificarea încearcă să găsească un set de legări pentru variabile astfel încât un fapt final să corespundă cu capul unei reguli. În PROLOG regulile sunt potrivite în ordinea în care sunt scrise.

Alte sisteme cu înlănțuire înapoi permit utilizarea unor reguli mai complexe.

5.3.2. Sisteme de reguli cu înlănțuire înainte

În anumite situații dorim să căutăm direcționați de datele de intrare. De exemplu, reacția mâinii de a se feri în momentul când a simțit o sursă puternică de căldură. Această comportare este modelată mai natural prin raționare înainte.

Potrivirea este de obicei mai dificilă în cazul sistemelor cu înlănțuire înainte decât al celor cu înlănțuire înapoi. De exemplu, să considerăm o regulă care verifică anumite condiții în descrierea stării și apoi adaugă o declarație. După ce regula s-a declanșat, condițiile sale probabil sunt valide în continuare, deci ar putea să se mai declanșeze imediat. Vom avea nevoie de un mecanism care să prevină declanșări repetate, în special când stările rămân neschimbate.

5.3.3. Combinarea raționamentului înainte și înapoi

Uneori anumite aspecte ale unei probleme sunt manipulate mai bine folosind înlănțuire înainte și alte aspecte folosind înlănțuire înapoi. De exemplu, să considerăm un program de diagnostic medicală cu înlănțuire înainte. Va primi 20 fapte despre starea sănătății unui pacient, apoi va raționa înainte pe aceste fapte pentru a deduce natura și cauza suferinței. Acum să presupunem că la un anumit moment partea stângă a unei reguli a fost *aproape* satisfăcută (de exemplu 90% din precondiții au fost validate). Ar putea fi ficient să aplicăm raționamentul înapoi pentru a satisface restul de 10% din precondiții într-un mod direcționat, și nu să așteptăm ca raționarea înainte să deducă faptul din întâmplare. Sau probabil restul de 10% de condiții cer teste medicale suplimentare, în care caz raționamentul înapoi poate fi folosit pentru interogarea utilizatorului.

Decizia dacă aceleași reguli se pot folosi pentru raționarea în ambele direcții depinde chiar de reguli. Dacă ambele părți ale regulii conțin declarații pure, atunci se pot potrivi declarațiile

dintr-o parte și se pot adăuga la descrierea stării declarațiile din cealaltă parte. Dacă proceduri arbitrare sunt permise în partea dreaptă a unei reguli, aceea regulă nu va mai fi reversibilă. Când se folosesc reguli ireversibile, în momentul scrierii lor trebuie luată o decizie despre direcția în care are loc căutarea.

5.4. Potrivire

Până acum am sugerat că procedurile de căutare inteligente implică alegerea dintre mai multe reguli care pot fi aplicate la un anumit moment, a acelei reguli care este cel mai probabil să conducă la o soluție. Pentru a realiza aceasta este nevoie să facem o **potrivire** (*matching*) între starea curentă și condițiile regulilor. Modul în care realizăm acest lucru este esențial pentru succesul unui sistem bazat pe reguli.

5.4.1. Indexarea

Un mod de a selecta regulile aplicabile este de a face o căutare simplă în mulțimea regulilor, să comparăm condițiile fiecărei reguli cu starea curentă și să extragem toate regulile care se potrivesc. Dar această soluție simplă ridică două probleme:

- Pentru a rezolva probleme interesante va fi necesar să folosim un număr mare de reguli. Scanarea tuturor regulilor la fiecare pas al procesului de căutare va fi o operație extrem de ineficientă.
- Nu întotdeauna este evident dacă condițiile unei reguli sunt satisfăcute de o anumită stare.

Uneori există moduri ușoare de a aborda prima problemă. În loc să scanăm toate regulile, folosim starea curentă ca index în mulțimea regulilor și selectăm imediat regulile care se potrivesc. Deseori există un echilibru între ușurința scrierii regulilor (care crește odată cu utilizarea descrierilor de nivel înalt) și simplitatea procesului de potrivire (care scade odată cu utilizarea descrierilor de nivel înalt).

Exemplul 1. În PROLOG și multe sisteme de demonstrare automată a teoremelor, regulile sunt indexate după predicatele pe care le conțin., deci toate regulile care pot fi aplicate pentru demonstrarea unui fapt pot fi accesate destul de repede.

Exemplul 2. În cazul jocului de șah, regulile pot fi indexate după piese și pozițiile lor.

În ciuda limitărilor acestei abordări, o formă oarecare de indexare este foarte importantă în operarea eficientă a sistemelor bazate pe reguli.

5.4.2. Potrivirea cu variabile

Problema selectării regulilor aplicabile este mai dificilă când condițiile conțin nu descrieri ale unei anumite situații ci proprietăți de complexitate variabilă pe care situațiile trebuie să le respecte. Procesul de verificare a existenței unei potriviri între o situație dată și condițiile unei reguli implică deseori un proces de căutare semnificativ.

În multe sisteme bazate pe reguli este nevoie să determinăm toată mulțimea regulilor care se potrivesc cu descrierea stării curente. Sistemele de raționare înapoi folosesc de obicei backtracking în adâncime (*Depth-First*) pentru selectarea regulilor, dar sistemele de raționare înainte utilizează **strategii de rezolvare a conflictelor** pentru selectarea unei reguli dintre cele aplicabile. Deși este posibil să apelăm procedura de unificare în mod repetat, pentru fiecare element din descrierea stării și fiecare regulă, este de preferat să considerăm o abordare specifică problemei de potrivire **multe-multe** (*many-many*), în care mai multe reguli sunt potrivite cu mai multe elemente din descrierea stării.

Un algoritm eficient de potrivire multe-multe este RETE, care se bazează pe elementele următoare:

Natura temporară a datelor. De obicei regulile nu modifică radical descrierea stărilor, ci adaugă sau șterg câteva elemente. Dacă o regulă nu s-a potrivit în etapa anterioară este foarte probabil că nu se va potrivi în etapa curentă. RETE folosește schimbările din condițiile stărilor pentru a determina care reguli s-ar putea aplica și care nu. Potrivirea completă se verifică doar pentru acei candidați care ar putea fi afectați de fluxul datelor.

Similarități structurale ale regulilor. Reguli diferite pot partaja un număr mare de condiții. De exemplu, regulile pentru identificarea animalelor. RETE memorează regulile astfel încât acestea partajează structurile în memorie. Mulțimile de condiții care apar în mai multe reguli sunt potrivite o singură dată.

Persistența consistenței legărilor variabilelor. Chiar dacă ar putea fi verificate toate condițiile unei reguli considerate individual, ar putea exista conflicte în ceea ce privește legările variabilelor. Aceste legări pot împiedica declanșarea unei reguli. Nu este necesar să determinăm consistența legării de fiecare dată când o nouă condiție este satisfăcută. RETE memorează calculele anterioare și le poate fuziona cu noile informații de legare.

5.4.3. Potrivirea complexă și aproximativă

Un proces de potrivire mai complicat este necesar în situația în care condițiile unei reguli specifică proprietăți care nu sunt menționate explicit în descrierea stării curente. În această situație este necesar un set de reguli separat pentru a descrie modul în care unele proprietăți se pot deduce din altele. Un proces de potrivire și mai complex apare în situația în care condițiile se potrivesc **aproximativ** cu situația curentă. Aceasta se întâmplă deseori în situații care implică descrierea fenomenelor fizice (de exemplu, descrierea vorbirii).

Potrivirea aproximativă este dificil de tratat deoarece pe măsură ce creștem toleranța permisă la potrivire creștem implicit și numărul de reguli care se potrivesc și, în consecință, dimensiunea procesului de căutare. Dar potrivirea aproximativă este superioară potrivirii exacte, mai ales în situații unde potrivirea exactă poate duce la absența regulilor care se potrivesc unei anumite descrieri. Pentru rezolvarea acestei probleme există două abordări distincte: tehnicile simbolice și sistemele conexioniste.

5.4.4. Rezolvarea conflictelor

Rezultatul procesului de potrivire este o listă de reguli ale căror precondiții au fost potrivite cu descrierea stării curente. În urma acestui proces au fost generate anumite legări de variabile. Metoda de căutare va trebui să decidă în continuare ordinea în care vor fi aplicate regulile. Dar uneori este util să incorporăm o parte din această decizie în procesul de potrivire. Această fază a procesului de potrivire se numește **rezolvarea conflictelor**.

Există trei abordări fundamentale ale problemei rezolvării conflictelor într-un sistem de producție:

- Atribuire o preferință pe baza regulii care s-a potrivit.
- Atribuire o preferință pe baza obiectelor care s-au potrivit.
- Atribuire o preferință pe baza acțiunii pe care ar realiza-o regula care s-a potrivit.

Preferințe bazate pe reguli

Există două moduri în care s'epot atribui preferințe pe baza regulilor. Prima și cea mai simplă este să considerăm că regulile au fost specificate într-o anumită ordine, ca de exemplu ordinea fizică în care sunt prezentate sistemului. Apoi prioritatea regulilor este stabilită în ordinea în care acestea apar. Această schemă este folosită în PROLOG.

Cealaltă schemă este de a da prioritate regulilor particulare față de cele generale. Scopul acestor reguli specifice este de a permite luarea în considerare a cunoștințelor pe care le folosește un rezolvitor atunci când rezolvă o problemă direct, fără căutare. Dacă considerăm toate regulile care se potrivesc, adăugarea unor astfel de reguli specifice duce la creșterea dimensiunii căutării. Pentru a evita aceasta, mecanismul de potrivire va respinge regulile mai generale decât alte reguli care se potrivesc. Există două moduri de a deduce că o regulă este mai generală decât alta:

- Dacă mulțimea precondițiilor unei reguli conține toate precondițiile altei reguli (și încă altele), atunci a doua regulă este mai generală decât prima.
- Dacă precondițiile unei reguli sunt aceleași cu cele ale altei reguli, cu excepția faptului că în primul caz sunt specificate variabile iar în al doilea constante, atunci prima regulă este mai generală decât a doua.

Preferințe bazate pe obiecte

Un alt mod în care procesul de potrivire poate ușura mecanismul de căutare este de a ordona potrivirile pe baza importanței obiectelor potrivite. Există mai multe moduri în care acest lucru poate fi realizat. De exemplu, ELIZA, un program de IA care simulează dialogul cu un psihiatru, utilizează faptul că unele cuvinte cheie din exprimarea utilizatorului sunt marcate ca fiind mai importante decât altele. Procedura de potrivire va returna acea potrivire care implică cuvântul cheie cel mai important. În afirmația făcută de utilizator “Eu știu că toți râd de mine” ELIZA recunoaște două cuvinte cheie, “eu” și “toți”. Deoarece ELIZA știe că “toți” este mai important decât “eu”, nu va răspunde cu afirmația “Spui că știi că toți râd de tine” (care se referă la “eu”) ci cu întrebarea “La cine anume te gândești?” (care se referă la “toți”). Acest tip de abordare este util dacă la orice moment se încearcă o singură regulă aleasă.

O altă formă de potrivire folosind priorități apare ca funcție a poziției obiectelor care sunt potrivite în descrierea stării curente. De exemplu, dacă dorim să modelăm comportarea memoriei scurte umane (*short-term memory*, STM), vom prefera acele potriviri care acționează asupra obiectelor care au intrat cel mai recent în STM. Vom utiliza obiectele mai vechi doar în măsura în care elementele noi cu produc o potrivire.

Preferințe bazate pe stări

Să presupunem că există câteva reguli care așteaptă să fie declanșate. Un mod de a selecta cea mai importantă este de a le declanșa temporar pe toate și de a examina rezultatele fiecăreia. Apoi, folosind o funcție euristică care poate evalua fiecare stare produsă, vom compara meritele rezultatelor și vom selecta regula preferată. Celelalte reguli fie le eliminăm, fie le păstrăm pentru mai târziu.

Această abordare este identică cu procedura de căutare Best-First, descrisă în Capitolul 3. Deși conceptual este considerată o strategie de rezolvare a conflictelor, de obicei este implementată ca tehnică de căutare care operează pe stările generate de aplicarea regulilor.

5.5. Controlul cunoștințelor

În timp ce programele inteligente cer căutare, căutarea este un proces greu de stăpânit din punct de vedere computațional, cu excepția cazului în care este limitată de cunoștințe despre domeniul problemei. În bazele de cunoștințe mari, care conțin mii de reguli, inabilitatea de a controla căutarea este o preocupare majoră. Când avem la dispoziție multe drumuri de raționare posibile, este esențial să nu fie urmate drumurile fără sens. Cunoștințele despre acele drumuri care este probabil să conducă mai repede la o stare finală sunt deseori numite **cunoștințe pentru controlul căutării**. Acestea pot lua mai multe forme:

1. Cunoștințe despre care dintre stări sunt de preferat altora.

2. Cunoștințe despre care dintre reguli trebuie aplicate într-o anumită situație.
3. Cunoștințe despre ordinea în care trebuie urmărite subgoal-urile.
4. Cunoștințe despre șiruri de reguli care este util să fie aplicate.

În Capitolul 3 am văzut cum un prim mod de a reprezenta cunoștințe este folosind funcții de evaluare euristică. Există multe alte moduri de a reprezenta cunoștințe de control. De exemplu, **regulile pot fi etichetate și partiționate**. Un sistem de diagnoza medicală poate folosi mulțimi distincte de reguli pentru boli bacteriologice și pentru boli imunologice. Dacă sistemul încearcă să verifice un anumit fapt prin raționare înapoi, probabil poate elimina din start una din cele două mulțimi de reguli. O altă metodă este de **a atribui regulilor măsuri de cost și de probabilitate a succesului**. Rezolvitorul problemei poate în acest fel să folosească analiza decizională probabilistică pentru a selecta în fiecare moment o alternativă eficientă din punct de vedere al costului.

Problema noastră este **cum să reprezentăm cunoștințele despre cunoștințe**. Din acest motiv cunoștințele pentru controlul căutării mai sunt denumite și **metacunoștințe**. O modalitate de reprezentare a acestora este reprezentarea declarativă folosind reguli, ca în Figura 5.4. Vom vedea în continuare două sisteme de IA care își reprezintă cunoștințele de control folosind reguli: SOAR și PRODIGY.

În condițiile A și B,

Regulile care (nu) menționează X

- deloc
- în partea stângă
- în partea dreaptă

vor

- fi în mod clar inutile
- fi probabil inutile
- ...
- fi probabil foarte utile

Figura 5.4: Sintaxa unei reguli de control

SOAR este o arhitectură generală pentru construirea sistemelor inteligente și se bazează pe o mulțime de ipoteze specifice despre structura rezolvării umane de probleme. Aceste ipoteze sunt deduse din ceea ce știm despre memoria pe termen scurt, efectele practicii, ș.a. În SOAR:

1. Memoria pe termen lung este stocată ca o mulțime de reguli.

2. Memoria pe termen scurt (*memorie de lucru*) este o zonă tampon care este influențată de percepții și care servește ca zonă de stocare pentru fapte deduse de regulile din memoria pe termen lung. Este analoagă cu descrierea stărilor în rezolvarea problemelor.
3. Toată activitatea de rezolvare a problemelor are loc pe măsura traversării spațiului de stări. Există câteva clase de activități de rezolvare a problemelor, incluzând raționarea despre care stări trebuie explorate, care reguli trebuie aplicate la un moment dat, și ce efect vor avea acele reguli.
4. Toate rezultatele finale și intermediare ale rezolvării problemelor sunt memorate pentru a putea fi folosite în viitor.

În ceea ce privește a treia caracteristică, la începutul execuției, SOAR construiește un spațiu inițial al problemei. În momentul în care trebuie să decidă aplicarea unei reguli dintr-o mulțime de reguli aplicabile, SOAR nu folosește o strategie fixă de rezolvare a conflictelor, ci consideră problema acestei alegeri ca fiind o problemă substanțială și construiește un spațiu auxiliar al problemei. SOAR are și reguli pentru a indica preferința de a aplica într-o anumită situație un întreg șir de reguli, facilitate utilă mai ales atunci când SOAR învață.

Putem să scriem reguli bazate pe preferința pentru unele stări în defavoarea altora. Astfel de reguli pot fi folosite pentru a implementa strategiile de căutare de bază din Capitolele 2 și 3. Astfel putem observa că toate aceste metode sunt subsumate de o arhitectură care raționează cu cunoștințe explicite pentru controlul căutării.

PRODIGY este un sistem general de rezolvare a problemelor care încorporează câteva mecanisme de învățare distincte. O bună parte din învățare este direcționată spre constatarea automată a unor etape de reguli de control pentru îmbunătățirea căutării într-un anumit domeniu. PRODIGY poate construi reguli pentru controlul căutării într-unul din următoarele moduri:

- Prin codificarea explicită de către programatori.
- Prin analiza statică a operatorilor domeniului.
- Prin analiza urmelor propriei comportări în rezolvarea problemelor.

PRODIGY învață reguli de control din experiență, atât din succese cât și din eșecuri. Dacă parcurge un drum care duce la eșec, va încerca să găsească o explicație, apoi va folosi explicația pentru a construi cunoștințe de control pentru evitarea în viitor a acestei căutări inutile.

Un motiv pentru care un drum poate duce la eșec este că subgoal-urile pot interacționa între ele. Astfel, în procesul rezolvării unui goal se poate anula soluția obținută pentru un subgoal anterior. Cunoștințele pentru controlul căutării ne pot oferi date despre ordinea în care ar trebui să abordăm subgoal-urile.

În încheiere vom discuta despre două probleme care par paradoxale. Prima se numește **problema utilității**. Pe măsură ce adăugăm mai multe cunoștințe de control, sistemul poate

căuta într-un mod mai judicios. Aceasta reduce numărul nodurilor expandate. Totuși, în efortul de deliberare în legătură cu pasul care trebuie urmat în continuare, sistemul trebuie să considere toate regulile de control. În anumite situații (în special pentru sisteme care generează cunoștințe de control în mod automat), este mai eficient să ignorăm cunoștințele de control.

A doua problemă privește **complexitatea interpretorului sistemului de producție**. Am vazut în acest capitol o tendință către reprezentarea explicită a din ce în ce mai multor cunoștințe despre cum ar trebui realizată căutarea. Un interpretor de sisteme de producție trebuie să știe cum să aplice regulile și meta-regulile, deci aceste interpretoare trebuie să fie mai complexe pe măsură ce ne depărtăm de sistemele simple cu raționare înapoi, precum PROLOG.

5.6. Aplicații rezolvate și probleme propuse

5.6.1. Care este dezavantajul reprezentării cunoștințelor folosind rețele semantice? Ilustrați pe un exemplu concret.

5.6.2. Fie problema de a răspunde la întrebări bazate pe următoarea bază de fapte:

1. Marcus era om.
2. Marcus era din Pompei.
3. Marcus s-a născut în anul 40 AD.
4. Toți oamenii sunt muritori.
5. Toți cei din Pompei au murit la erupția vulcanului, în anul 79 AD.
6. Nici un muritor nu trăiește mai mult de 150 de ani.
7. Acum este anul 1998 AD.

Să se scrie un program *Prolog* care să răspundă la întrebarea “Este Marcus în viață?”.

Rezolvare

Introducem următoarele predicate:

- (a) $om(X)$ cu semnificația “ $\langle X \rangle$ este om”;
- (b) $dinPompei(X)$ cu semnificația “ $\langle X \rangle$ este din Pompei”;
- (c) $muritor(X)$ cu semnificația “ $\langle X \rangle$ este muritor”;
- (d) $nascut(X, N)$ cu semnificația “ $\langle X \rangle$ s-a născut în anul $\langle N \rangle$ ”;
- (e) $mort(X, N)$ cu semnificația “ $\langle X \rangle$ a murit în anul $\langle N \rangle$ ”;

- (f) $mort(X)$ cu semnificația “ $\langle X \rangle$ este mort”;
- (g) $acum(N)$ cu semnificația “Acum este anul $\langle N \rangle$ ”.

Cunoscând faptul că propoziția

dacă p și q atunci r

se reprezintă în PROLOG folosind clauza

$r :- p, q.$

vom transcrie faptele date în limbaj natural, folosind următoarele clauze într-un program PROLOG.

```
domains
    el=symbol
predicates
    om(el)
    dinPompei(el)
    muritor(el)
    nascut(el,integer)
    mort(el,integer)
    mort(el)
    acum(integer)
clauses
    om(marcus).                % propozitia (a)
    dinPompei(marcus).         % propozitia (b)
    nascut(marcus,40).         % propozitia (c)
    muritor(X) :-              % propozitia (d)
        om(X).
    mort(X,79) :-              % propozitia (e)
        dinPompei(X),
        !.
    mort(X) :-                 % propozitia (f)
        muritor(X),
        acum(M),
        nascut(X,P),
        M-P > 150,
        !.
    mort(X) :-
        mort(X,\_).
    acum(1998).                % propozitia (g)
```


Întrebarea va fi pusă sistemului sub forma *mort(marcus)*, răspunsul fiind *Yes*, ceea ce înseamnă în limbaj natural “*Marcus nu mai este în viață*”.

5.6.3. Fie problema de a răspunde la întrebări bazate pe următoarea bază de fapte:

- (a) Toți copii cu vârsta de peste 7 ani merg la școală.
- (b) Maria este mama lui Dan.
- (c) Toți frații lui Dan sunt născuți înainte de 1990.
- (d) George este copilul Mariei.
- (e) Școlarii cu vârsta de peste 10 ani sunt acum în vacanță.
- (f) Acum este anul 2000.

Să se scrie un program *Prolog* care să răspundă la întrebarea “*Este George în vacanță?*”.

5.6.4. Fie problema de a răspunde la întrebări bazate pe următoarea bază de fapte:

- (a) Toți prietenii lui George având vârsta de peste 10 ani merg la școală.
- (b) Maria este sora lui Dan.
- (c) Toți frații lui Dan sunt născuți înainte de 1990.
- (d) George este prieten cu toți frații lui Dan.
- (e) Acum este anul 2000.

Să se scrie un program *Prolog* care să răspundă la întrebarea “*Merge Maria la școală?*”.

5.6.5. Fie problema de a răspunde la întrebări bazate pe următoarea bază de fapte:

- (a) Toți elevii cu vârsta de peste 15 ani de la liceul “X” merg în excursie.
- (b) Ion este tatăl lui Marius.
- (c) Toți frații lui Mihai sunt elevi la liceul “X”.
- (d) Mihai este copilul lui Ion.
- (e) Copiii lui Ion care sunt elevi la liceul “X” au cel puțin 15 ani.
- (f) Acum este anul 2000.

Să se scrie un program *Prolog* care să răspundă la întrebarea “*Va merge Marus în excursie?*”.

5.6.6. Fie problema de a răspunde la întrebări bazate pe următoarea bază de fapte:

- (a) Toți prietenii lui Dan având vârsta de peste 10 ani locuiesc în cartierul “X”.
- (b) Ana este sora lui George.
- (c) Toți frații lui George sunt născuți înainte de 1987.
- (d) Dan este prieten cu toți frații lui George cu vârsta de peste 12 ani.
- (e) Acum este anul 2000.

Să se scrie un program *Prolog* care să răspundă la întrebarea “Locuiește Ana în cartierul $\langle X \rangle$?”.

Capitolul 6

Jocuri

6.1. Introducere

Sunt două motive pentru care jocurile par să fie un domeniu potrivit pentru explorarea inteligenței artificiale:

- Jocurile oferă probleme structurate, în care este foarte ușor să se măsoare succesul sau eșecul.
- În mod evident jocurile nu cer o cantitate mare de cunoștințe. Jocurile au fost gândite pentru a fi rezolvabile prin căutare simplă de la poziția de start până la o poziție câștigătoare.

Primul motiv este în continuare valabil și este motivul principal al interesului continuu din domeniul jocurilor jucate de mașini. Din păcate, al doilea motiv nu adevărat decât pentru jocurile cele mai simple. De exemplu, pentru jocul de șah următoarele observații sunt valabile:

- Factorul de ramificare mediu este 35.
- Într-un joc mediu, fiecare jucător poate face 50 de mutări.
- Deci pentru a examina arborele complet ar trebui să examinăm 35^{100} poziții.

Prin urmare este evident că un program care folosește căutarea simplă nu va putea selecta nici măcar prima mutare în timpul vieții adversarului său. Este necesară folosirea unei proceduri de căutare euristică.

Procedurile de căutare pe care le-am analizat până acum sunt esențialmente proceduri generează-și-testează, în care testarea are loc după ce generatorul a depus un efort de o complexitate variabilă. La o extremă, generatorul generează soluții propuse întregi, pe care apoi testerul le evaluează. La cealaltă extremă, generatorul generează mutări individuale, care sunt evaluate independent de taser și cea mai promițătoare este selectată. Pentru a îmbunătăți eficiența unui program de rezolvare a problemelor bazat pe căutare trebuie realizate două lucruri:

- Îmbunătățirea procedurii de generare astfel încât să fie generate doar mutările (sau drumurile) bune.
- Îmbunătățirea procedurii de testare astfel încât mutările (sau drumurile) cele mai bune să fie recunoscute și explorate cu prioritate.

În programele de jocuri este esențial ca ambele operații să fie realizate. În ceea ce privește jocul de șah, dacă folosim un generator al mutărilor legale, sunt generate foarte multe mutări. Astfel testerul, care trebuie să răspundă repede, nu poate fi absolut corect. Dacă, însă, în loc să folosim un generator de mutări legale folosim un **generator de mutări plauzibile**, care generează un număr mic de mutări promițătoare, procedura de testare poate să consume un timp mai mare pentru evaluarea mutărilor generate. Desigur, pentru ca generatorul și testerul să fie eficienți, pe măsură ce jocul avansează este din ce în ce mai important să fie folosite euristici. Astfel, performanța globală a sistemului poate să crească.

Desigur, căutarea nu este singura tehnică avută în vedere. În unele jocuri, în anumite momente sunt disponibile tehnici mai directe. De exemplu, în șah deschiderile și închiderile sunt stilizate, și deci este mai bine să fie jucate prin căutare într-o bază de date de modele memorate. Pentru a juca un joc, trebuie să combinăm tehnicile orientate pe căutare cu cele mai directe.

Modalitatea ideală de a folosi o procedură de căutare pentru a găsi o soluție a unei probleme este de a genera deplasări în spațiul problemei până când se atinge o stare finală. Pentru jocuri complexe, deseori nu este posibil să căutăm până la identificarea unei stări câștigătoare, chiar dacă folosim un generator bun de mutări plauzibile. Adâncimea și factorul de ramificare ale arborelui generat sunt prea mari. Totuși este posibil să căutăm într-un arbore pe o anumită adâncime, de exemplu doar 10 sau 20 de mutări. Apoi, pentru a selecta cea mai bună mutare, stările generate trebuie comparate pentru a o descoperi pe cea mai avantajoasă. Aceasta se realizează cu o **funcție de evaluare statică**, care folosește informațiile disponibile pentru a evalua stările individuale prin estimarea probabilității acestora de a conduce la câștigarea jocului. Această funcție este similară cu funcția h' a algoritmului A^* . Funcția de evaluare statică poate fi aplicată pozițiilor generate de mutările propuse. Din cauza dificultății construirii unei astfel de funcții care să fie și foarte corectă, este de preferat să generăm câteva nivele din arborele stărilor și abia atunci să aplicăm funcția.

Este foarte important modul în care construim funcția de evaluare statică. De exemplu, pentru jocul de șah, o funcție de evaluare statică foarte simplă se bazează pe avantajul pieselor: calculează suma valorilor pieselor albe (A), a pieselor negre (N) și calculează raportul A/N . O variantă mai sofisticată este o combinație liniară a mai multor factori importanți, printre care valoarea pieselor, posibilitatea de a avansa, controlul centrului, mobilitatea. Au fost construite și funcții neliniare.

O posibilitate de a îmbunătăți o funcție bazată pe ponderarea unor factori este ca decizia în legătură cu ponderile factorilor să se bazeze pe experiența din jocurile anterioare: ponderile factorilor care au dus la câștig vor crește, cele ale factorilor care dus la înfrângere vor scădea.

Dar trebuie să luăm în calcul și posibilitatea ca victoria noastră să fie cauzată de faptul că inamicul a jucat prost, nu că noi am fi jucat bine. Problema deciziei referitoare la care dintre o serie de acțiuni este responsabilă pentru un anumit rezultat se numește **problema atribuirii creditului** (*credit assignment*).

Am discutat despre două componente ale unui program de joc: un generator de mutări plauzibile bun și o funcție de evaluare statică bună. Pe lângă acestea avem nevoie și de o procedură de căutare care permite analizarea a cât de multe mutări din spațiul stărilor, înainte ca acestea să fie efectiv operate. Avem la dispoziție mai multe strategii:

- Algoritmul **A***, pentru care funcția **h'** este aplicată nodurilor terminale, și valorile sunt propagate înapoi către vârful arborelui. Procedura nu este adecvată pentru jocuri deoarece jocurile sunt operate de două persoane adverse.
- Procedura **Minimax**, care se bazează pe alternanța jucătorilor. În arborele stărilor, un nivel al stărilor generate de mutările unui jucător va fi urmat de un nivel al stărilor generate de mutările adversarului său. Trebuie să avem în vedere interesele opuse ale celor doi jucători.
- Algoritmul **B***, aplicabil atât arborilor standard de rezolvare a problemelor, cât și arborilor de jocuri.

6.2. Procedura de căutare Minimax

Procedura de căutare Minimax este o procedură de căutare Depth-First limitată. Ideea este să începem la poziția curentă și să utilizăm generatorul de mutări plauzibile pentru a genera mulțimea pozițiilor succesori. Acum putem aplica acestor poziții funcția de evaluare statică și putem selecta poziția cea mai promițătoare. Valoarea acestei poziții poate fi trecută ca valoare a poziției inițiale, deoarece poziția inițială este la fel de bună ca cea mai bună poziție produsă. Vom presupune că funcția de evaluare statică produce valori cu atât mai mari cu cât mai bună este poziția pentru noi. Astfel, în această fază scopul nostru este să **maximizăm** valoarea produsă de funcție.

Dar cum știm că funcția de evaluare statică nu este absolut corectă, an dori să mergem cu căutarea mai departe. Dorim să vedem ce se întâmplă cu fiecare dintre pozițiile produse după ce adversarul va face o mutare. Pentru aceasta, în loc să aplicăm funcția de evaluare statică pozițiilor generate denoi, vom aplica generatorul de mutări plauzibile pentru a genera câte o mulțime de mutări bune ale adversarului pentru fiecare poziție generată de noi. Dacă dorim să ne oprim acum, după două mutări, putem aplica funcția de evaluare statică. Dar acum trebuie să avem în vedere că adversarul decide mutarea pe care o va face și valoarea care va fi transferată pozițiilor de pe nivelul superior. Scopul lui este să **minimizeze** valoarea funcției. Prin urmare, pentru fiecare mutare candidat a noastră, vom putea identifica o mutare candidat a

adversarului. Concluzia care se impune este ca vom prefera să facem acea mutare pentru care mutarea candidat a adversarului (cea mai bună din punctul său de vedere) va produce poziția cea mai bună din punctul nostru de vedere. La nivelul deciziei adversarului, întotdeauna a fost selectată și trimisă spre nivelul superior valoarea minimă. La nivelul deciziei noastre, întotdeauna a fost selectată și trimisă spre nivelul superior valoarea maximă. Figura 6.1 prezintă un exemplu de propagare a valorilor în arbore.

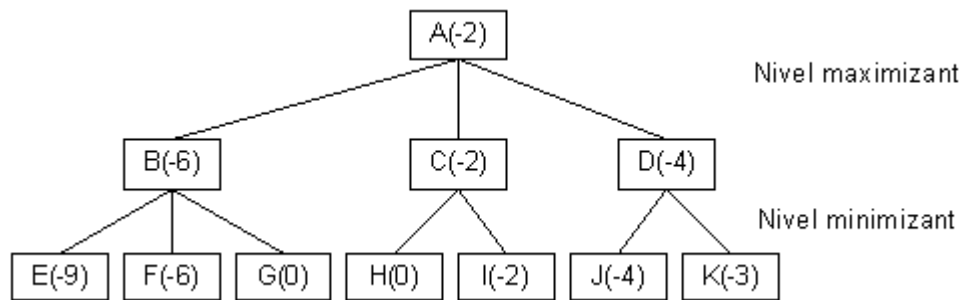


Figura 6.1: Propagarea valorilor spre nivelul superior într-o căutare pe două niveluri

În continuare vom descrie mai precis procedura Minimax. Este o procedură recursivă care se bazează pe două proceduri auxiliare specifice jocului:

MOVEGEN(Poziție, Jucător) – Generatorul de mutări plauzibile, care întoarce o listă de noduri reprezentând mutările care pot fi operate de Jucător la Poziția specificată. Cei doi jucători îi numim UNU și DOI. În jocul de șah pot fi numiți ALB și NEGRU.

STATIC(Poziție, Jucător) – Funcția de evaluare statică, care întoarce un număr reprezentând calitatea Poziției din punctul de vedere al Jucătorului. O soluție este să se determine $STATIC(Poziție, UNU)$ și, dacă este necesar, se va considera $STATIC(Poziție, DOI) := -STATIC(Poziție, UNU)$.

O problemă critică a procedurii Minimax este de a decide când să oprim recursivitatea și să apelăm funcția de evaluare statică. Dintre factorii care influențează această decizie, amintim următorii:

- A câștigat una dintre părți?
- Câte niveluri de mutări am explorat?
- Cât de promițător este drumul curent?
- Cât de mult timp a rămas?
- Cât de stabilă este configurația?

Pentru procedura generală Minimax descrisă aici vom folosi următoarea funcție:

DEEP-ENOUGH(Poziție, Adâncime) – se presupune că evaluează toți factorii de mai sus; întoarce TRUE dacă căutarea trebuie oprită la nivelul curent și FALSE dacă trebuie continuată.

O problemă care provine din recursivitatea procedurii Minimax este că are nevoie să întoarcă o structură cu doi parametri. Vom defini două funcții care vor putea extrage valorile celor doi parametri, astfel:

VALUE – valoarea drumului selectat.

PATH – drumul propriu-zis. Vom întoarce tot drumul, chiar dacă probabil va fi necesară doar prima mutare.

Procedura Minimax va fi apelată cu trei parametri, astfel:

MINIMAX(Poziție, Adâncime, Jucător) – parametrii reprezintă Poziția curentă, Adâncimea curentă și Jucătorul care urmează să mute.

Deci apelul pentru determinarea celei mai bune mutări de la poziția CURRENT este

MINIMAX(CURRENT, 0, UNU)

sau

MINIMAX(CURRENT, 0, DOI)

în funcție de jucătorul care urmează să mute.

Algoritm: MINIMAX(Poziție, Adâncime, Jucător)

1. Dacă DEEP-ENOUGH(Poziție, Adâncime) atunci întoarce structura

VALUE := STATIC(Poziție, Jucător)

PATH := Nil

2. Altfel generează un nivel al arborelui apelând funcția MOVEGEN(Poziție, Jucător) și atribuie variabilei SUCCESSORS lista întoarsă.

3. Dacă SUCCESSORS este vidă atunci nu se pot face mutări și întoarce structura de la pasul 1.

4. Dacă SUCCESSORS nu este vidă, examinează fiecare element și memorează-l pe cel mai bun. Inițializează BEST-SCORE la valoarea minimă pe care o poate întoarce funcția STATIC. Pentru fiecare element SUCC din lista SUCCESSORS, execută următoarele:

(a) Atribuie RESULT-SUCC := MINIMAX (SUCC, Adâncime+1, ADVERSAR (Jucător))

(b) Atribuie NEW-VALUE := -VALUE(RESULT-SUCC)

(c) Dacă $\text{NEW-VALUE} > \text{BEST-SCORE}$, am identificat un succesor mai bun decât cei de până acum. Memorează aceasta în modul următor:

(i) Atribue $\text{BEST-SCORE} := \text{NEW-VALUE}$

(ii) Cel mai bun drum este cel de la CURRENT la SUCC și apoi de la SUCC în continuare așa cum a fost determinat de apelul recursiv la MINIMAX . Deci setează BEST-PATH la rezultatul atașării lui SUCC în fața lui $\text{PATH}(\text{RESULT-SUCC})$.

5. Acum știm valoarea poziției și drumul de urmat. Deci întoarce structura

```
VALUE := BEST-SCORE
PATH  := BEST-PATH
```

Pentru a vedea cum funcționează algoritmul, executați-l pe arborele din Figura 6.1. Procedura este foarte simplă, dar poate fi îmbunătățită semnificativ cu câteva rafinări.

6.3. Adăugarea tăieturilor alfa-beta

Un lucru bun în legătură cu tehnicile Depth-First este acela că eficiența lor poate fi îmbunătățită prin utilizarea tehnicilor branch-and-bound în care soluții parțiale care sunt în mod clar mai slabe decât soluții cunoscute pot fi abandonate mai repede. Dar, cum procedura tradițională Depth-First a fost modificată pentru a lua în considerare pe ambii jucători, este de asemenea necesar să modificăm strategia Branch-and-Bound pentru a include două limite, pentru ambii jucători: o limită superioară și una inferioară. Această strategie modificată se numește **trunchiere alfa-beta**. Presupune actualizarea a două valori de prag, prima reprezentând limita minimă a valorii pe care o poate primi un nod maximizat (limită numită **alfa**), și a doua reprezentând limita maximă a valorii pe care o poate primi un nod minimizat (limită numită **beta**).

Pentru exemplificarea funcționării valorilor alfa și beta, să considerăm Figura 6.2. În căutarea acestui arbore se caută tot subarborele cu rădăcina în B și se descoperă că valoarea pe care A o poate lua este cel puțin 3. Când această valoare a lui alfa este pasată către F, ne va permite să nu mai explorăm nodul L. După ce K este examinat vedem că I are valoarea minimă 0, care este mai mică decât 3, deci nu are rost să mai analizăm ramificațiile lui I. Jucătorul maximizant știe că nu trebuie să mute de la C la I, pentru că astfel ar obține cel mult 0, în loc de 3, ca în cazul mutării la B. După trunchierea explorării lui I, se examinează J, care produce valoarea 5, atribuită lui F (maximul dintre 0 și 5). Această valoare devine valoarea beta pentru nodul C. Indică faptul că C ia o valoare de cel mult 5. Continuăm cu expandarea lui G. Examinăm pe M, și are valoarea 7, trecută lui G ca tentativă. Dar 7 este comparat cu beta (5), este mai mare, iar jucătorul de la nivelul C trebuie să minimizeze, și ca atare nu va selecta G, care va conduce

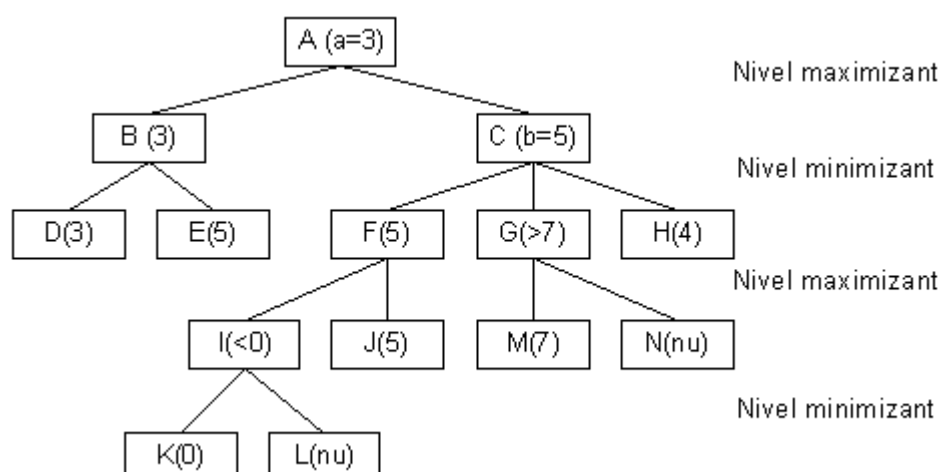


Figura 6.2: Tăieturile alfa și beta (nodurile L și N nu mai sunt examinate)

la un scor de cel puțin 7. Jucătorul are o mutare alternativă, F, cu valoarea 5, și astfel nu este necesar să se exploreze celelalte ramificații ale lui G.

Să notăm că, pentru determinarea necesității trunchierii căutării, la nivelurile maximizante este utilizat doar beta, iar la nivelurile minimizante, doar alfa. Totuși, trebuie cunoscute și celelalte valori, deoarece trebuie trecute prin recursivitate la nivelurile inferioare. Deoarece algoritmul Minimax este recursiv, vom avea nevoie de un mecanism de a transmite valorile alfa și beta. Vom folosi doi parametri suplimentari, și în loc să ne referim la aceștia ca alfa și beta, ne vom referi ca valoare-utilizată (variabila USE-THRESH) și valoare-trecută-la-următorul-nivel (variabila PASS-THRESH). Deoarece la fiecare nivel este folosită exact una dintre cele două valori, iar celaltă va fi folosită la nivelul imediat următor, această abordare permite utilizarea unei singure proceduri pentru ambii jucători. USE-TRESH va fi trecută la nivelul următor ca PASS-THRESH, și PASS-THRESH ca USE-THRESH.

La un nivel maximizant, alfa este PASS-THRESH și beta este USE-THRESH, iar la un nivel minimizant, beta este PASS-THRESH și alfa este USE-THRESH. La fiecare nivel vom folosi USE-THRESH pentru trunchieri și vom actualiza PASS-THRESH pentru a fi folosit la nivelul următor.

Pentru a vedea acest lucru, să presupunem că suntem la un nivel maximizant, de exemplu la nodul A, în vârful arborelui. Acolo se setează alfa ca fiind valoarea celui mai bun succesor găsit până acum. Ori, valoarea celui mai bun succesor este beta. Lucrurile se schimbă ușor dacă nu suntem la nivelul rădăcinii. Astfel, la un nivel maximizant, alfa trebuie setat la maximul dintre valoarea pe care o avusese la nivelul maximizant imediat mai înalt și cea mai mare valoare găsită la acest nivel.

Practic, în tot ceea ce facem, modul de calcul al lui PASS-THRESH este identic cu cel al lui BEST-SCORE, și deci putem elimina BEST-SCORE ca variabilă separată. Astfel procedura va avea următorii parametri:

MINIMAX-A-B(Poziție, Adâncime, Jucător, Use-Thresh, Pass-Thresh)

Apelul inițial va fi următorul:

MINIMAX-A-B(CURRENT, 0, UNU, valoare-maximă-pentru-STATIC,
valoare-minimă-pentru-STATIC)

Aceste două valori inițiale pentru USE-THRESH și PASS-THRESH sunt cele mai proaste valori pe care le pot lua.

Algoritm: MINIMAX-A-B(Poziție, Adâncime, Jucător, Use-Thresh, Pass-Thresh)

1. Dacă DEEP-ENOUGH(Poziție, Adâncime) atunci întoarce structura

VALUE := STATIC(Poziție, Jucător)
PATH := Nil

2. Altfel generează un nivel al arborelui apelând funcția MOVEGEN(Poziție, Jucător) și atribuie variabilei SUCCESSORS lista întoarsă.
3. Dacă SUCCESSORS este vidă atunci nu se pot face mutări și întoarce structura de la pasul 1.
4. Dacă SUCCESSORS nu este vidă, examinează fiecare element și memorează-l pe cel mai bun. Pentru fiecare element SUCC din lista SUCCESSORS, execută următoarele:
 - (a) Atribuie RESULT-SUCC := MINIMAX-A-B (SUCC, Adâncime+1, ADVERSAR (Jucător), - Pass-Thresh, - Use-Thresh)
 - (b) Atribuie NEW-VALUE := -VALUE(RESULT-SUCC)
 - (c) Dacă NEW-VALUE > Pass-Thresh, am identificat un succesori mai bun decât cei de până acum. Memorează aceasta în modul următor:
 - (i) Atribuie BEST-SCORE := NEW-VALUE1
 - (ii) Cel mai bun drum este cel de la CURRENT la SUCC și apoi de la SUCC în continuare așa cum a fost determinat de apelul recursiv la MINIMAX-A-B. Deci setează BEST-PATH la rezultatul atașării lui SUCC în fața lui PATH (RESULT-SUCC).
 - (d) Dacă Pass-Thresh (care reflectă cea mai bună valoare) nu este mai bun decât Use-Thresh, atunci trebuie să oprim examinarea acestei ramificații. Dar ambele variabile au fost inversate. Deci dacă Pass-Thresh \geq Use-Thresh, atunci revine imediat cu valoarea

VALUE := Pass-Thresh
PATH := BEST-PATH

5. Întoarce structura

VALUE := Pass-Thresh

PATH := BEST-PATH

Eficiența procedurii alfa-beta depinde în mod esențial de ordinea în care sunt examinate drumurile. Dacă cele mai rele drumuri sunt examinate mai întâi, atunci nu va apare nici o trunchiere. Dacă, în schimb, drumurile cele mai bune ar fi cunoscute dinainte, astfel încât s-ar putea garanta că sunt examinate cu prioritate, nu ar trebui să ne mai deranjăm cu un proces de căutare. Dacă, totuși, am cunoaște eficiența tehnicii de trunchiere în cazul perfect, am avea o limită superioară pe care să o folosim în alte situații.

Procedura alfa-beta poate fi extinsă astfel încât să elimine drumuri suplimentare care par să fie doar puțin mai bune decât alte drumuri care au fost deja explorate. La pasul 4(d) am abandonat procesul de căutare dacă drumul în curs de explorare nu era mai bun decât un alt drum deja explorat. De exemplu, în Figura 6.3, după generarea nodului G observăm că valoarea acestuia, de 3.2, este doar puțin mai bună decât valoarea 3 a celui mai bun nod de până atunci, B. Știm că dacă vom face mutarea B scorul garantat este 3, astfel că probabil este de preferat să terminăm explorarea nodului C acum, și să explorăm în continuare alte părți ale arborelui. Terminarea explorării unui subarboare care oferă posibilități slabe de îmbunătățire se numește **trunchiere de inutilitate** (*futility cutoff*).

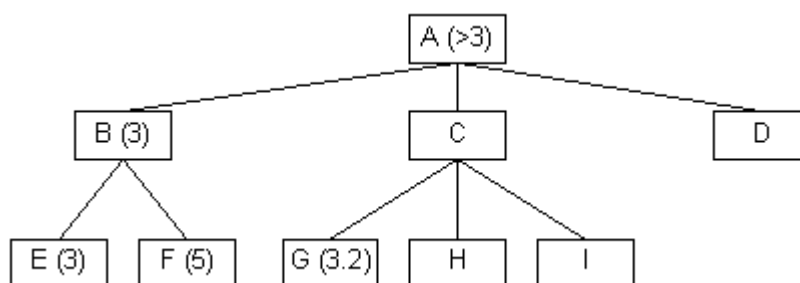


Figura 6.3: O trunchiere de inutilitate

6.4. Rafinări suplimentare

Pe lângă trunchierea alfa-beta există o mare varietate de alte modificări ale procedurii minimax, și care pot îmbunătăți performanța. În continuare vom discuta patru dintre cele mai importante îmbunătățiri.

6.4.1. Așteptarea unei perioade de liniște

Unul dintre factorii care ar trebui să fie luați uneori în considerare în determinarea momentului în care să oprim intrarea în adâncime în arborele de căutare este dacă situația este relativ

stabilă. Să considerăm situația din Figura 6.4, primul arbore. După expandarea nodului B, se obține al doilea arbore. Acum estimarea valorii nodului B se schimbă radical și am putea decide că B nu este o mutare bună, astfel că am putea opri examinarea subarborelui lui B. De exemplu, acest lucru se poate întâmpla în jocul de șah în timpul unui schimb de piese, când valoarea poziției se schimbă pe termen scurt datorită inițierii schimbului.

Pentru a ne asigura că astfel de măsuri pe termen scurt nu influențează negativ decizia noastră pentru alegerea mutării, ar trebui să continuăm căutarea până când estimarea valorii nodului B nu se va schimba radical de la un nivel la următorul. Această fază se numește **așteptarea unei perioade de liniște**. Dacă avem răbdare vom putea ajunge la situația descrisă de al treilea arbore din Figura 6.4, în care din nou mutarea în poziția B pare rezonabilă, deoarece, de exemplu, deja a avut loc și cealaltă jumătate a schimbului de piese.

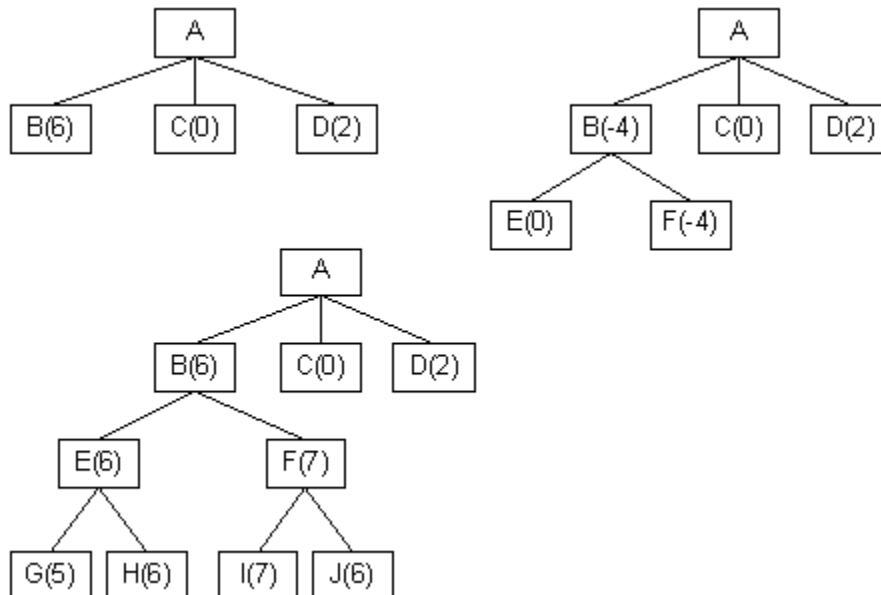


Figura 6.4: Așteptarea unei perioade de liniște

Așteptarea unei perioade de liniște ne ajută să evităm **efectul de orizont**, în care un eveniment inevitabil defavorabil poate fi întârziat prin tactici variate până când nu mai apare în porțiunea din arborele de joc pe care o explorează Minimax. Efectul de orizont poate influența, de asemenea, percepția mutărilor favorabile. Efectul poate face ca o mutare să pară bună în ciuda faptului că mutarea ar fi mai bună dacă ar fi amânată dincolo de orizont. Chiar cu luarea în considerare a perioadelor de liniște, toate programele de căutare cu adâncime fixă sunt supuse unui efect de orizont subtil.

6.4.2. Căutare secundară

O posibilitate de a combate efectul de orizont este de a verifica de mai multe ori o mutare selectată, pentru a ne asigura că nu există o capcană la câteva mutări depărtate. Să presupunem

că examinăm un arbore de joc cu o adâncime medie de șase mutări și, pe baza acestei căutări selectăm o anumită mutare. Deși ar fi fost prea scump să căutăm întregul arbore până la o adâncime de opt, este rezonabil să căutăm doar ramura selectată pe o adâncime suplimentară de două niveluri, pentru a ne asigura că mutarea aleasă este bună. Această tehnică se numește **căutare secundară**.

O formă utilă a căutării secundare este așa-numita **extensie singulară**. Ideea este că dacă un nod terminal este considerat mult superior fraților săi și dacă valoarea căutării depinde în mod critic de corectitudinea valorii acelui nod, atunci acel nod este expandat pe încă un nivel. Aceasta permite programului de căutare să se concentreze pe combinații tactice, de forță. Utilizează un criteriu pur sintactic, selectând liniile de joc interesante fără să apeleze la cunoștințe suplimentare.

6.4.3. Utilizarea unei arhive de mutări

Pentru jocuri complicate luate ca întreg nu este realizabilă selecția unei mutări prin căutarea configurației curente într-un catalog și extragerea mutării corecte. Catalogul ar fi imens și imposibil de construit. Dar pentru anumite segmente ale unor jocuri, această abordare este rezonabilă. În șah atât deschiderile cât și închiderile sunt puternic stilizate. În aceste situații performanța unui program poate fi mult îmbunătățită prin luarea în considerare a unei liste de mutări recomandate (numită **arhivă de mutări**). Utilizarea arhivei de mutări la deschiderea și închiderea jocului combinată cu utilizarea procedurii Minimax pentru partea de mijloc este un bun exemplu al modului în care cunoștințele și căutarea pot fi combinate pentru a produce rezultate mai eficiente decât ar putea produce fiecare pe cont propriu.

6.4.4. Alternative la Minimax

Minimax are câteva aspecte care, chiar cu îmbunătățirile descrise, sunt problematice. De exemplu, se bazează în mod esențial pe ipoteza că adversarul va selecta întotdeauna cea mai bună mutare. Această presupunere este acceptabilă în situațiile de câștig, în care poate fi găsită o mutare bună pentru noi. Dar în situațiile de înfrângere ar putea fi mai bine să ne asumăm riscul că adversarul va face o greșeală. De exemplu, să presupunem că avem de ales între două mutări, ambele conducând la situații foarte proaste dacă adversarul joacă perfect, dar una dintre ele este puțin mai proastă decât cealaltă. Să presupunem în continuare că mutarea mai puțin promițătoare ar putea conduce la o situație foarte bună pentru noi dacă adversarul ar face o singură greșeală. Deși procedura Minimax ar selecta mutarea mai bună, ar trebui să alegem cealaltă mutare, care este probabil puțin mai proastă, dar posibil mult mai bună. Pentru a putea lua astfel de decizii în cunoștință de cauză trebuie să avem acces la modelul stilului de joc al adversarului, astfel încât să poată fi estimată posibilitatea apariției diferitelor greșeli. Dar o astfel de analiză este foarte greu de realizat.

6.5. Adâncire iterativă

Mai multe idei pentru căutarea arborilor de jocuri pentru doi jucători au condus la noi algoritmi pentru căutări euristice de tipul celor descrise în Capitolul 3. O astfel de idee este **adâncirea iterativă** (*iterative deepening*). În loc să se realizeze o căutare în arborele jocului pe un număr fin de nivele, în prima fază se realizează o căutare pe un singur nivel, se aplică funcția de evaluare statică rezultatelor fiecărei mutări posibile, apoi se inițiază o nouă procedură Minimax pentru o adâncime de două nivele, urmată apoi de o căutare pe trei nivele, apoi patru nivele, etc. Numele “adâncire iterativă” derivă din faptul că la fiecare iterație se caută pe o adâncime mai mare cu un nivel.

Sunt mai multe motive pentru care ne interesează și iterațiile intermediare, nu doar cea finală. Mai întâi, programele de jocuri sunt supuse limitărilor în timp. Deoarece nu se poate cunoaște dinainte cât timp va dura operația de căutare, un program ar putea intra în criză de timp. Folosind adâncirea iterativă, procesul de căutare poate fi oprit în orice moment și cea mai bună mutare identificată poate fi jucată efectiv. În plus, iterațiile anterioare pot oferi informații în legătură cu modul în care trebuie ordonate mutările pentru a fi examinate mai eficient (vezi procedura alfa-beta).

În continuare vom descrie un algoritm care combină cele mai bune aspecte ale algoritmilor Depth-First și Breadth-First: algoritmul Adâncire Iterativă Depth-First:

Algoritm: Adâncire Iterativă Depth-First (DFID)

1. Setează $\text{SEARCH-DEPTH} := 1$.
2. Realizează o căutare Depth-First pe adâncimea SEARCH-DEPTH . Dacă se identifică un drum soluție, returnează-l.
3. Altfel, incrementează SEARCH-DEPTH cu 1 și reia de la pasul 2.

Problema strategiei de căutare Depth-First este că nu poate ști dinainte care este adâncimea soluției în spațiul de stări. Algoritmul DFID evită problema trunchierilor fără sacrificarea eficienței, și, de fapt, DFID este algoritmul de căutare neinformată optimal conform criteriilor de spațiu și timp.

În ceea ce privește căutarea informată, euristică, adâncirea iterativă poate fi folosită pentru îmbunătățirea performanței algoritmului A^* . Deoarece dificultatea majoră a algoritmului A^* este cantitatea mare de memorie necesară pentru a actualiza listele de noduri, adâncirea iterativă poate fi foarte utilă.

Algoritm: Adâncire Iterativă A^* (IDA^*)

1. Setează $\text{THRESHLD} := \text{evaluarea euristică a stării de start}$.
2. Realizează o căutare Depth-First, eliminând orice ramură când valoarea funcției de cost totale ($g+h'$) depășește THRESHOLD . Dacă în timpul căutării se identifică un drum soluție, returnează-l.

3. Altfel, incrementează THRESHOLD cu valoarea minimă cu care a fost depășită la pasul anterior și reia de la pasul 2.

Precum A^* , algoritmul IDA^* garantează identificarea unei soluții optimale, în condițiile în care h' este o euristică admisibilă. Din cauza tehnicii de căutare Depth-First, IDA^* este foarte eficientă relativ la spațiu. IDA^* a fost primul algoritm de căutare euristică utilizat la identificarea drumurilor soluții pentru problema 15-puzzie (versiunea 4x4 a problemei 8-puzzle) în condiții de timp și spațiu rezonabile.

6.6. Aplicații rezolvate și probleme propuse

6.6.1. Fie arborele de joc prezentat în Figura 6.5. Valorile atașate nodurilor terminale reprezintă valori ale funcției de evaluare statice. Știind că primul jucător e cel maxim:

1. Ce mutare va alege?
2. Ce noduri terminale nu vor fi analizate, folosind tăietura $\alpha - \beta$?

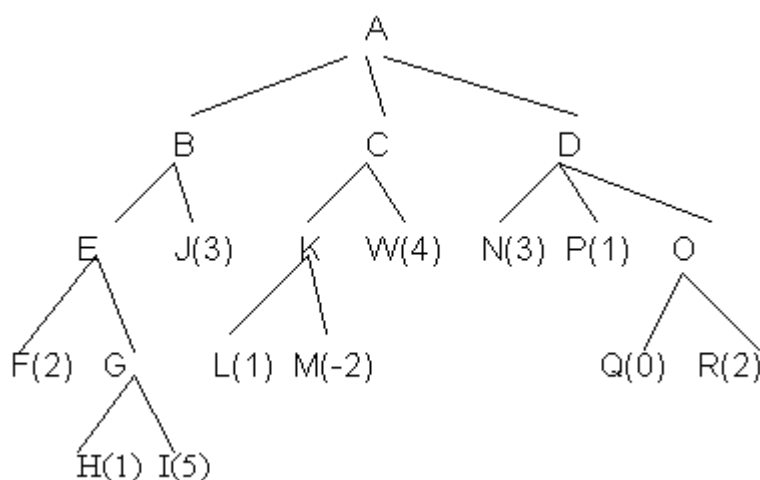


Figura 6.5: Arborele de joc pentru Problema 6.6.1

Rezolvare

După cum se știe, într-un astfel de arbore joc alternează nivelul maximizant (la care jucătorul alege cea mai mare valoare dintre valorile fiilor săi) cu nivelul minimizant (la care jucătorul alege cea mai mică valoare dintre valorile fiilor săi). Știind că primul nivel este nivel maximizant, propagarea valorilor în arbore este ilustrată în Figura 6.6.

În concluzie, mutarea aleasă va fi B și are valoarea 2 (aceasta fiind valoarea obținută în rădăcină în urma propagărilor). Spre simplificare, am înlocuit un nod neterminal X cu valoarea ce a fost obținută prin propagarea valorilor de la fiii lui X.

Folosind tăietura $\alpha - \beta$ în căutarea unui arbore de joc, se calculează două valori:

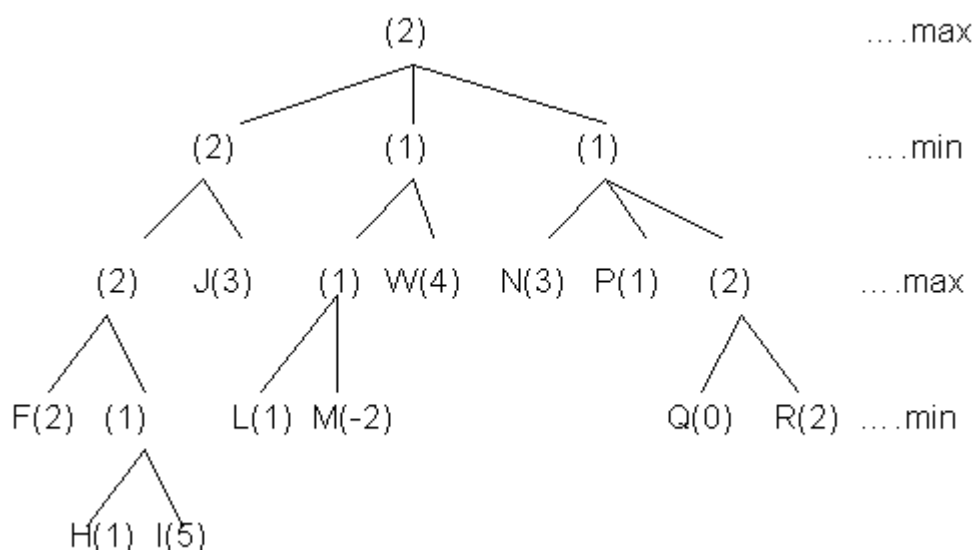


Figura 6.6: Propagarea valorilor in arborele de joc pentru Problema 6.6.1

- α – valoarea minimă la un nivel maximizant;
- β – valoarea maximă la un nivel minimizant.

În aceste condiții, se trunchiază căutarea la nodurile la care $\alpha > \beta$. Aplicând această observație, vom obține:

1. se analizează F; rezultă că valoarea lui E este ≥ 2 ($\alpha = 2$);
2. se analizează H; rezultă că valoarea lui G este ≤ 1 ($\beta = 1$);
3. nu se mai analizează I, deoarece $\alpha > \beta$ (I nu furnizează posibilitate de îmbunătățire a valorii nodului E);
4. se cunoaște valoarea lui E (2); rezultă că valoarea lui B este ≤ 2 ($\beta = 2$);
5. se analizează J; rezultă că valoarea lui B este 2; rezultă că valoarea lui A este ≥ 2 ($\alpha = 2$);
6. se analizează L și M; rezultă că valoarea lui K este 1; rezultă că valoarea lui C este ≤ 1 ($\beta = 1$);
7. nu se mai analizează W, deoarece $\alpha > \beta$;
8. se cunoaște valoare lui C (1);
9. se analizează N; rezultă că valoarea lui D este ≤ 3 ($\beta = 3$);
10. se analizează P; rezultă că valoarea lui D este ≤ 1 ($\beta = 1$);

11. nu se mai analizează O, deoarece $\alpha > \beta$;

În concluzie nu se analizează următoarele noduri terminale: I, W, Q și R.

6.6.2. Fie arborele de joc prezentat în Figura 6.7. Valorile atașate nodurilor terminale reprezintă valori ale funcției de evaluare statice. Știind că primul jucător e cel maxim:

1. Ce mutare va alege?

2. Ce noduri terminale nu vor fi analizate, folosind tăietura $\alpha - \beta$?

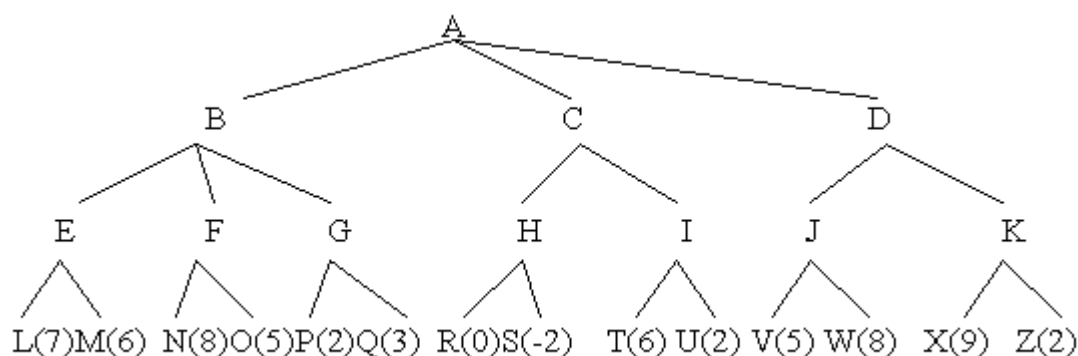


Figura 6.7: Arborele de joc pentru Problema 6.6.2

6.6.3. Fie arborele de joc prezentat în Figura 6.8. Valorile atașate nodurilor terminale reprezintă valori ale funcției de evaluare statice. Știind că primul jucător e cel minim:

1. Ce mutare va alege?

2. Ce noduri terminale nu vor fi analizate, folosind tăietura $\alpha - \beta$?

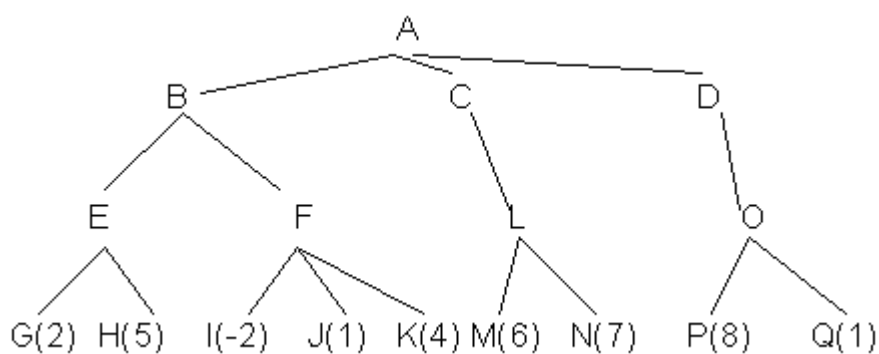


Figura 6.8: Arborele de joc pentru Problema 6.6.3

6.6.4. Dați exemplu de arbore de căutare pe trei nivele astfel încât folosind tăietura $\alpha - \beta$ să fie analizate jumătate din numărul nodurilor terminale ale arborelui. Se vor specifica valorile funcției de evaluare statice atașate nodurilor terminale ale arborelui. Se va justifica alegerea făcută.

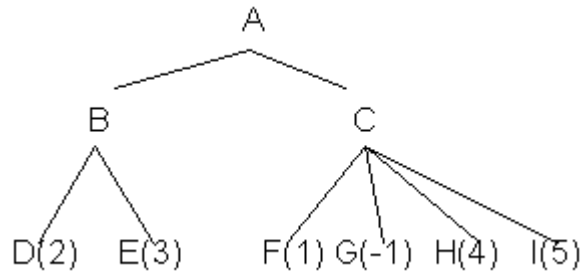


Figura 6.9: Arborele de joc pentru Problema 6.6.4

Rezolvare

Fie arborele de joc prezentat în Figura 6.9. Presupunând că primul jucător este cel maxim, folosind tăietura $\alpha - \beta$ nu se vor analiza nodurile G, H, I. Mutarea aleasă va fi B (cu valoarea 2). Raționamentul este următorul:

1. se analizează D și E; rezultă că valoarea lui E este 2; rezultă că valoarea lui A este ≥ 2 ($\alpha = 2$);
2. se analizează F; rezultă că valoarea lui C este ≤ 1 ($\beta = 1$);
3. nu se mai analizează G, H, I, deoarece $\alpha > \beta$ (cele trei noduri nu furnizează posibilitate de îmbunătățire a valorii nodului A).

6.6.5. Dați exemplu de arbore de căutare pe patru nivele astfel încât folosind tăietura $\alpha - \beta$ să fie analizate:

1. $1/4$ din numărul nodurilor terminale ale arborelui;
2. trei noduri terminale ale arborelui.

Se vor specifica valorile funcției de evaluare statice atașate nodurilor terminale ale arborelui. Se va justifica alegerea făcută.

6.6.6. Dacă numărul nodurilor terminale ale unui arbore de joc este n , care va fi numărul nodurilor terminale analizate utilizând algoritmul de căutare $MM-\alpha - \beta$? Se va justifica răspunsul.

6.6.7. Care este numărul nodurilor terminale ale unui arbore de joc, știind că folosind tăietura $\alpha - \beta$ se analizează doar 10 noduri? Se va justifica răspunsul.

6.6.8. *Descrieți un spațiu de căutare în care adâncirea iterativă funcționează mai prost decât căutarea în adâncime. Justificați răspunsul.*

6.6.9. *Să se modifice algoritmul de joc MINIMAX pentru un joc cu 3 persoane. Justificați modificările făcute.*

6.6.10. *Să se specifice explicit pentru jocul Tic-Tac-Toe (X-0) funcția de evaluare statică și generatorul de mutări.*

Rezolvare

Vom descrie în continuare programul Borland Pascal care implementează jocul Tic-Tac-Toe în situația în care ambii jucători joacă perfect. După cum se va vedea ca efect al execuției programului (de fapt acest lucru se poate demonstra), în cazul în care ambii jucători joacă foarte bine, fără să greșească, jocul se termină cu REMIZĂ (adică nu câștigă nici unul din jucători).

Ca și algoritm de rezolvare al jocului s-a folosit algoritmul MINIMAX fără tăieturi (prezentat în partea de teorie).

Varianta de program ce va fi prezentată poate fi îmbunătățită:

- folosind varianta cu tăieturi $\alpha - \beta$ a algoritmului MiniMax;
- mutările celui de-al doilea jucător să nu fie generate prin program (adică să nu fie mutările optime), ci să poată fi preluate de la tastatură (acest lucru însemnând că jocul celui de-al doilea jucător ar putea fi “cu greșeli”).

uses

crt;

type

tabla = array[1..9] of 0..2;

lista = ^nod;

nod = record

inf: tabla;

leg: lista;

end;

procedure *init* (**var** *a: tabla*);

begin

fillchar(a, sizeof(a), 2);

end;

function *final* (*a: tabla; var castig0, castig1, remiza: boolean*): *boolean*;

{ verifica daca s-a ajuns la finalul jocului: variabilele booleene indica ce jucator a castigat jocul, sau daca e remiza }

var

i, j, n0, n1, nt: integer;

begin

castig0 := false;

castig1 := false;

remiza := false;

nt := 0; {numarul total de piese ocupate pe tabla}

i := 1;

while (*i* ≤ 3) **and** (**not** *castig0*) **and** (**not** *castig1*) **do** {*caut pe linia i*}

begin

n0 := 0; {numarul de piese 0}

n1 := 0; {numarul de piese 1}

for *j := 1 to 3 do* {*parcurs linia i*}

begin

if *a[3*i-2+j-1]=0 then*

inc(n0);

if *a[3*i-2+j-1]=1 then*

inc(n1);

end;

nt := nt+n0+n1;

if *n0=3 then*

castig0 := true

else if *n1=3 then*

castig1 := true

else

i := i+1;

end;

if (**not** *castig0*) **and** (**not** *castig1*) **then**

begin

i := 1;

while (*i* ≤ 3) **and** (**not** *castig0*) **and** (**not** *castig1*) **do** {*caut pe coloane*}

begin

n0 := 0; {numarul de piese 0}

n1 := 0; {numarul de piese 1}

for *j := 1 to 3 do* {*parcurs coloana i*}

begin

if *a[i+3*(j-1)]=0 then*

```

    inc(n0) ;
    if a[i+3*(j-1)] = 1 then
        inc(n1) ;
    end ;
    if n0 = 3 then
        castig0 := true
    else if n1 = 3 then
        castig1 := true
    else
        i := i+1 ;
    end ;
end ;
if (not castig0) and (not castig1) then
begin
    n0 := 0 ;
    n1 := 0 ;
    for j := 1 to 3 do {verific diagonala principala}
        begin
            if a[1+4*(j-1)] = 0 then
                inc(n0) ;
            if a[1+4*(j-1)] = 1 then
                inc(n1) ;
            end ;
        end
    if n0 = 3 then
        castig0 := true
    else if n1 = 3 then
        castig1 := true
    else {verific diagonala secundara}
        begin
            n0 := 0 ;
            n1 := 0 ;
            for j := 1 to 3 do
                begin
                    if a[3+2*(j-1)] = 0 then
                        inc(n0) ;
                    if a[3+2*(j-1)] = 1 then
                        inc(n1) ;
                    end ;
                end
            if n0 = 3 then

```

```

        castig0 := true
    else if n1=3 then
        castig1 := true
    end;
end;
if (not castig0) and (not castig1) and (nt=9) then
    remiza := true;
final := castig0 or castig1 or remiza;
end;

```

function *Static* (*a:tabla; juc: integer*): *integer*;

{ valoarea functiei de evaluare statice din punctul de vedere al jucatorului juc; suma1 = se aduna 1 pentru fiecare rand in care pot castiga si am deja 1 piesa si 2 pentru fiecare rand in care pot castiga si am deja 2 piese; suma2 = se aduna 1 pentru fiecare rand in care adversarul poate castiga si 2 pentru fiecare rand in care adversarul poate castiga si are deja 2 piese; functia de evaluare statica (daca juc e 1) = suma1-suma2; altfel va fi suma2-suma1; eu sunt jucatorul 1, adversarul e jucatorul 2 }

var

s0, s1, n0, n1, i, j, k: integer;
castig0, castig1, remiza: boolean;

begin

if *final(a, castig0, castig1, remiza)* **then** {daca pozitia este finala}

begin

if *castig1* **then**

begin

s1 := 30000;

s0 := 0;

end;

if *castig0* **then**

begin

s1 := 0;

s0 := 30000;

end;

if *remiza* **then**

begin

s1 := 0;

s0 := 0;

end;

end

```

else
  begin
    s0 := 0;
    s1 := 0;
    i := 1;
    while (i<=9) do
      begin
        j := i;
        { numar pe linia "i div 3 + 1" cate piese am eu si cate are }
        { adversarul; 1 sunt piesele mele, 0 ale adversarului }
        n0 := 0;
        n1 := 0;
        for k := 1 to 3 do {parcurg linia}
          begin
            if a[j+k-1]=1 then
              inc(n1);
            if a[j+k-1]=0 then
              inc(n0);
          end;
        if n0=0 then {pot castiga pe linia curenta}
          s1 := s1+n1;
        if n1=0 then {adversarul poate castiga pe linia curenta}
          s0 := s0+n0;
        i := i+3; {se trece la urmatoarea linie a tablei}
      end;

    for i := 1 to 3 do {numar pe coloane}
      begin
        n0 := 0; {numarul de piese 0}
        n1 := 0; {numarul de piese 1}
        for j := 1 to 3 do {parcurg coloana i}
          begin
            if a[i+3*(j-1)]=0 then
              inc(n0);
            if a[i+3*(j-1)]=1 then
              inc(n1);
          end;
        if n0=0 then {pot castiga pe coloana curenta}
          s1 := s1+n1;
      end;
  end;

```

```

    if  $n1=0$  then {adversarul poate castiga pe coloana curenta}
         $s0 := s0+n0$ ;
    end;

 $n0 := 0$ ;
 $n1 := 0$ ;
for  $j := 1$  to 3 do {verific diagonala principala}
    begin
        if  $a[1+4*(j-1)]=0$  then
            inc( $n0$ );
        if  $a[1+4*(j-1)]=1$  then
            inc( $n1$ );
        end;
    if  $n0=0$  then {pot castiga pe diagonala principala}
         $s1 := s1+n1$ ;
    if  $n1=0$  then {adversarul poate castiga pe diagonala principala}
         $s0 := s0+n0$ ;

 $n0 := 0$ ;
 $n1 := 0$ ;
for  $j := 1$  to 3 do {verific diagonala secundara}
    begin
        if  $a[3+2*(j-1)]=0$  then
            inc( $n0$ );
        if  $a[3+2*(j-1)]=1$  then
            inc( $n1$ );
        end;
    if  $n0=0$  then {pot castiga pe diagonala secundara}
         $s1 := s1+n1$ ;
    if  $n1=0$  then {adversarul poate castiga pe diagonala secundara}
         $s0 := s0+n0$ ;
    end;
if  $juc=1$  then
     $Static := s1-s0$ 
else
     $Static := s0-s1$ ;
end;

procedure move_gen ( $a$ : tabla;  $juc$ : integer; var  $cap\_succ$ : lista);

```


{ generatorul de mutari al jucatorului juc pornind din pozitia data de tabla a; se genereaza lista inlantuita a mutarilor posibile, al carei cap este cap_succ }

var

k, v: integer;

q: lista;

coada_succ: lista;

begin

if *juc=1* **then**

v := 1 {primul jucator (calculatorul) e cu 1, al doilea cu 0}

else

v := 0;

cap_succ := nil;

coada_succ := nil;

for *k := 1 to 9* **do** *{se efectueaza mutarile posibile}*

if *a[k]=2* **then**

begin

a[k] := v;

{genereaza noua configuratie si se introduce in lista succesorilor}

new(q);

q^.inf := a;

q^.leg := nil;

if *cap_succ=nil* **then**

begin

cap_succ := q;

coada_succ := q;

end

else

begin

coada_succ^.leg := q;

coada_succ := q;

end;

a[k] := 2;

end;

end;

function *egal (a, b: tabla): boolean;*

{ verifica daca 2 configuratii coincid }

var

i: integer;

begin

for $i := 1$ **to** 9 **do**

if $a[i] <> b[i]$ **then**

begin

$egal := false;$

$exit;$

end;

$egal := true;$

end;

function $apare(a: tabla; cap: lista) : boolean;$

{ se verifica daca configuratia a apare in lista inlantuita al carei prim element este cap }

var

$p: lista;$

begin

$p := cap;$

while $p <> nil$ **do**

begin

if $egal(p^.inf, a)$ **then** { s-a gasit configuratia in lista }

begin

$apare := true;$

$exit;$

end;

$p := p^.leg;$

end;

$apare := false;$

end;

procedure $eliberare(p: lista);$

{ elibereaza zona de heap ce incepe de la adresa p }

var

$q: lista;$

begin

while $p <> nil$ **do**

begin

$q := p;$

$p := p^.leg;$

$dispose(q);$

```

    end ;
end ;

procedure creare (succ: lista; var p: lista; q: lista) ;
{ se adauga succ ca prim element in lista q si rezulta lista p }
var
    ultim, r: lista;
begin
    new(p) ;
    p^.inf := succ^.inf;
    p^.leg := nil;
    ultim := p;
    while q<>nil do
        begin
            new(r) ;
            r^.inf := q^.inf;
            r^.leg := nil;
            ultim^.leg := r;
            ultim := r;
            q := q^.leg;
        end ;
    end ;
end ;

procedure MiniMax (a: tabla; ad: integer; juc: integer; var val_optima: integer;
    var cap_cale_optima: lista; cap_noduri_generate: lista) ;
{ aplicarea algoritmului MiniMax, cu furnizarea urmatoarelor rezultate: val_optima -
    valoarea optima a caii alese, cap_cale_optima - capul listei inlantuite corespunzatoare
    caii optime. Ad - reprezinta adancimea jocului, juc reprezinta jucatorul curent. Pentru
    a evita generarea unei configuratii de mai multe ori, se retine o lista inlantuita cu
    nodurile deja generate, al carei cap este cap_noduri_generate }
var
    succ, cap_succ, cap_cale_optima_succ, p: lista;
    best, val_noua, val_optima_succ: integer;
    castig0, castig1, remiza: boolean;
begin
    if final(a, castig0, castig1, remiza) then {s-a terminat jocul}
        begin
            val_optima := Static(a, juc) ;
            cap_cale_optima := nil;

```

```

    end
else
    begin
        move_gen(a,juc,cap_succ); {se genereaza un nou nivel}
        if cap_succ=nil then {nu exista succesor}
            begin
                val_optima := Static(a,juc);
                cap_cale_optima := nil;
            end
        else
            {examineaza fiecare element din lista succesorilor si retine cel mai bun}
            begin
                best := -MAXINT; {valoarea minima pe care o poate lua Static}
                succ := cap_succ;
                new(cap_cale_optima);
                cap_cale_optima^.leg := nil;
                while succ<>nil do
                    {daca succesorul nu a fost deja generat}
                    if not apare(succ^.inf,cap_noduri_generate) then
                        begin
                            {adauga succesorul in lista nodurilor deja generate}
                            new(p);
                            p^.inf := succ^.inf;
                            p^.leg := cap_noduri_generate;
                            cap_noduri_generate := p;
                            MiniMax(succ^.inf, ad+1,3-juc, val_optima_succ, cap_cale_optima_succ,
cap_noduri_generate);
                            {se sterge succ din lista nodurilor deja generate}
                            p := cap_noduri_generate;
                            cap_noduri_generate := cap_noduri_generate^.leg;
                            dispose(p);
                            val_noua := -val_optima_succ;
                            if val_noua>best then
                                {s-a gasit un succesor mai bun}
                                begin
                                    best := val_noua;
                                    {creaza calea optima prin adagarea lui succ}
                                    {in capul caii optime gasite pentru succesor}
                                    eliberare(cap_cale_optima);
                                end
                            end
                        end
                    end
                end
            end
        end
    end

```

```

        creare(succ, cap_cale_optima, cap_cale_optima_succ);
    end;
    succ := succ^.leg;
    eliberare(cap_cale_optima_succ);
    end;
    { se elibereaza zona de memorie pentru lista succesorilor }
    eliberare(cap_succ);
    val_optima := best;
    end;
end;
end;

procedure scrie (a: tabla; juc: integer);
{ tipareste configuratia curenta sub forma de matrice - juc este jucatorul care muta }
var
    i: integer;
begin
    writeln('Muta jucatorul ', juc);
    writeln;
    writeln(' _ _ _ ');

    { prima linie }
    for i := 1 to 3 do
        begin
            write(' | ');
            case a[i] of
                0: write('0');
                1: write('X');
                2: write(' ');
            end;
        end;
    writeln(' | ');
    writeln(' _ _ _ ');

    { a doua linie }
    for i := 1 to 3 do
        begin
            write(' | ');
            case a[3+i] of

```

```

    0: write( '0' );
    1: write( 'X' );
    2: write( ' ' );
    end;
end;
writeln( '|' );
writeln( ' _ _ _ ' );

{ a treia linie }
for i := 1 to 3 do
    begin
        write( '|' );
        case a[6+i] of
            0: write( '0' );
            1: write( 'X' );
            2: write( ' ' );
        end;
    end;
    writeln( '|' );
    writeln( ' _ _ _ ' );
    readln;
end;

procedure tipar (cap_cale_optima: lista);
var
    p: lista;
    juc: integer;
    castig0, castig1, remiza: boolean;
begin
    clrscr;
    juc := 1;    { jucatorul curent - care muta }
    scrie(cap_cale_optima^.inf, juc);    { tipareste tabla curenta }
    p := cap_cale_optima^.leg;
    while p^.leg <> nil do
        begin
            juc := 3-juc;    { alterneaza mutarile celor 2 jucatori }
            scrie(p^.inf, juc);
            p := p^.leg;
        end;
    end;
end;

```

```

    {ultima mutare}
    scrie(p^.inf, 3-juc);
    writeln;
    if final(p^.inf, castig0, castig1, remiza) then
        if castig0 then
            writeln('Castiga jucatorul 2')
        else if castig1 then
            writeln('Castiga jucatorul 1')
        else if remiza then
            writeln('Remiza');
        eliberare(cap_cale_optima);
        readln;
    end;

procedure joc;
{ se initializeaza jocul - primul jucator (cel cu 1) muta aleator }
var
    a:tabla;
    x, val_optima: integer;
    p, cap_cale_optima, cap_noduri_generate: lista;
begin
    init(a);
    randomize;
    x := random(9)+1;      { se genereaza aleator o pozitie pe tabla }
    a[x] := 1;             { muta primul jucator }
    { continua jocul cel de-al doilea jucator }
    new(cap_noduri_generate);
    cap_noduri_generate^.leg := nil;
    cap_noduri_generate^.inf := a;
    MiniMax(a, 1, 2, val_optima, cap_cale_optima, cap_noduri_generate);
    { se adauga prima configuratie in capul listei rezultat }
    new(p);
    p^.inf := a;
    p^.leg := cap_cale_optima;
    cap_cale_optima := p;
    { se tipareste evolutia jocului si rezultatul final }
    tipar(cap_cale_optima);
end;

```

```
{ programul principal }  
begin  
    joc;  
end.
```


Capitolul 7

Planificare

7.1. Introducere

Pentru rezolvarea celor mai multe probleme este necesar să combinăm unele din strategiile fundamentale de rezolvare a problemelor descrise în Capitolul 3 cu unul sau mai multe mecanisme de reprezentare a cunoștințelor. De multe ori este util să împărțim problema în subprobleme mai mici și să rezolvăm aceste subprobleme separat în măsura în care acest lucru este posibil. În final va trebui să combinăm soluțiile parțiale pentru a forma o soluție completă a problemei. Există două moduri în care este important să putem realiza această descompunere.

Mai întâi trebuie să evităm efortul de a recalcula întreaga stare a problemei atunci când ne deplasăm de la o stare la alta. Dorim să considerăm doar acea parte a unei stări care este posibil să se fi modificat. Problema cadrului (*frame problem*), adică problema determinării lucrurilor care se modifică și a celor care nu se modifică, devine din ce în ce mai importantă pe măsură ce crește complexitatea problemei.

Al doilea mod în care descompunerea poate ușura identificarea soluției unei probleme dificile este divizarea unei singure probleme dificile în mai multe probleme mai ușor de rezolvat. Algoritmul AO* oferă o posibilitate de a realiza acest lucru atunci când este posibil să împărțim problema originală într-un număr de subprobleme complet separate, ceea ce deseori nu este posibil. Multe probleme pot fi considerate **aproape decompozabile** (*almost decomposable*), ceea ce înseamnă că pot fi descompuse în mai multe subprobleme care interacționează foarte limitat. De exemplu, problema mutării mobilei dintr-o cameră. Subproblemele, anume mutarea fiecărei piese de mobilier, interacționează în mod limitat în sensul că piesele de mobilier vor trebui mutate într-o anumită ordine, pentru a nu face mișcări inutile.

Au fost propuse mai multe metode pentru realizarea acestor două tipuri de descompuneri. Aceste metode propun moduri de **descompunere a problemei** originale în subprobleme corespunzătoare și moduri de **înregistra și gestiona interacțiunile** dintre subprobleme pe măsură ce sunt detectate în timpul procesului de rezolvare a problemei. Utilizarea acestor metode se numește **planificare** (*planning*).

În limbaj obișnuit termenul “planificare” se referă la procesul calucării câtorva pași ai procedurii de rezolvare înainte de a fi executat vre-unul. Există categorii de probleme pentru care deosebirea dintre a planifica o operație și a o executa este foarte mică (de exemplu probleme pentru care operațiile care s-au dovedit greșite pot fi ignorate sau desfăcute). Dacă la un moment dat se identifică un blocaj, prin backtracking se poate explora un drum nou. Dar există, de asemenea, categorii de probleme pentru care această deosebire este critică (de exemplu probleme pentru care operațiile care s-au dovedit greșite nu pot fi nici ignorate nici desfăcute). În astfel de situații planificarea devine extrem de importantă. Dacă realizarea operațiilor este irevocabilă, simularea lor nu este. Astfel putem evita lumea reală căutând soluții într-o lume simulată, în care backtracking-ul este autorizat. După ce am găsit o soluție, o putem executa în lumea reală.

Dar această abordare atinge o altă caracteristică a domeniului problemei: universul este previzibil? Într-o problemă pentru care universul nu este previzibil, nu putem fi siguri dinainte care va fi rezultatul unei operații, dacă lucrăm prin simulare. În cel mai bun caz putem considera **mulțimea rezultatelor posibile**, eventual într-o ordine în funcție de probabilitatea apariției acestor rezultate. Putem produce planuri pentru fiecare rezultat posibil la fiecare pas, dar deoarece în general multe dintre rezultatele posibile sunt foarte improbabile, ar fi o pierdere de timp să formulăm planuri exhaustive.

Există două posibilități pentru evitarea acestei situații. Putem analiza lucrurile câte un pas la un moment dat, fără să încercăm să planificăm. Această abordare este considerată pentru **sistemele reactive**. Opțiunea este de a produce un singur plan care este probabil să funcționeze bine. Dacă planul eșuează o posibilitate este să îl ignorăm și să reluăm procesul de planificare plecând de la situația curentă.

Pe de altă parte rezultatul neașteptat nu invalidează tot restul planului, ci doar o parte din acesta. Probabil o modificare minoră a planului, cum ar fi adăugarea unui pas suplimentar, îl va face din nou aplicabil.

În lumea reală este dificil să ajungem în situația în care fiecare aspect este previzibil. În totdeauna trebuie să fim pregătiți să avem planuri care eșuează. Dacă am produs planul prin descompunerea problemei în subprobleme, atunci impactul eșecului unui anumit pas asupra planului ar putea fi destul de localizat, ceea ce este un argument în favoarea descompunerii problemei. Pe lângă reducerea complexității combinatoriale a procesului de rezolvare a problemelor, se reduce și complexitatea procesului de revizuire dinamică a planului.

De fiecare dată când executăm o operație este de preferat nu doar să memorăm pașii executați, ci și să asociem acestora motivul pentru care au fost executați. Astfel, când un pas eșuează, vom putea determina care dintre părțile planului depind de acest pas și, prin urmare, ar putea fi nevoie să fie modificate.

7.2. Domeniu exemplu: lumea blocurilor

Tehnicile pe care le vom discuta pot fi aplicate într-o mare varietate de domenii de probleme. Pentru a permite compararea metodelor, le vom analiza pe un singur domeniu suficient de complex pentru a justifica utilitatea tuturor mekansimelor descrise și suficient de simplu pentru a putea găsi exemple ușor de urmărit. Un astfel de domeniu este **lumea blocurilor**. Avem la dispoziție o suprafață plată pe care se pot plasa blocuri. Există un număr de blocuri în formă de cub având aceleași dimensiuni. Blocurile pot fi plasate unul peste altul. Există un braț de robot care poate manevra blocurile. Robotul poate ține un singur bloc la un moment dat. Fiecare bloc poate avea un singur bloc pe el la un moment dat. Acțiunile care pot fi realizate de robot includ:

UNSTACK(A, B) – Ia blocul A din poziția sa curentă de pe blocul B. Brațul robotului trebuie să fie gol și blocul A nu trebuie să aibă blocuri pe el.

STACK(A, B) – Plasează blocul A peste B. Brațul trebuie să țină blocul A, și blocul B nu trebuie să aibă blocuri pe el.

PICKUP(A) – Ia blocul A din poziția sa curentă de pe masă. Brațul robotului trebuie să fie gol și blocul A nu trebuie să aibă blocuri pe el.

PUTDOWN(A) – Plasează blocul A pe masă. Brațul trebuie să țină blocul A.

Pentru a specifica atât condițiile în care au loc operațiile cât și rezultatele realizării operațiilor, vom folosi predicatele:

ON(A, B) – Blocul A este pe blocul B.

ONTABLE(A) – Blocul A este pe masă.

CLEAR(A) – Blocul A nu are blocuri pe el.

HOLDING(A) – Brațul robotului ține blocul A.

ARMEMPTY – Brațul robotului este gol.

În această lume a blocurilor sunt valabile multe declarații logice. De exemplu,

$$\exists x : \text{HOLDING}(x) \rightarrow \neg \text{ARMEMPTY}$$

$$\forall x : \text{ONTABLE}(x) \rightarrow \neg \exists y : \text{ON}(x, y)$$

$$\forall x : [\neg \exists y : \text{ON}(y, x)] \rightarrow \text{CLEAR}(x)$$

7.3. Componentele unui sistem de planificare

În cazul sistemelor de rezolvare a problemelor bazate pe tehnicile elementare discutate în Capitolul 3 a fost necesar să realizăm fiecare din următoarele funcții:

1. Alege cea mai bună regulă de aplicat pe baza celei mai bune informații euristice disponibile.
2. Aplică regula aleasă pentru a determina noua stare a problemei.
3. Detectează dacă s-a identificat o soluție.
4. Detectează blocajele, astfel încât ele să poată fi abandonate și efortul sistemului să poată fi direcționat în direcții mai interesante.

În cazul sistemelor mai complexe pe care le vom discuta în continuare se cer tehnici pentru realizarea tuturor acestor funcții. Deseori este importantă și o a cincea operație:

5. Detectează dacă s-a identificat o soluție aproape corectă și utilizează tehnici speciale pentru a o transforma într-o soluție absolut corectă.

Înainte să discutăm metode de planificare specifice vom arunca o privire la modul în care aceste operații pot fi realizate.

Alegerea regulilor

Cea mai utilizată tehnică pentru selectarea regulilor potrivite este izolarea unei mulțimi de diferențe dintre starea finală dorită și starea curentă și apoi identificarea regulilor relevante pentru reducerea acelor diferențe. Dacă sunt identificate mai multe reguli, se pot exploata o varietate de informații euristice pentru a alege regula de aplicat. Această tehnică se bazează pe metoda de analiză means-ends, descrisă în Capitolul 3.

Aplicarea regulilor

În sistemele simple pe care le-am discutat aplicarea regulilor era o operație simplă. Fiecare regulă specifică starea care ar rezulta din aplicarea ei. Acum va trebui să putem gestiona reguli care specifică o parte restrânsă a stării complete a problemei. Pentru aceasta există mai multe posibilități.

O primă posibilitate este să descriem pentru fiecare acțiune fiecare dintre schimbările pe care le aduce descrierii stării. În plus sunt necesare unele declarații care să precizeze că restul descrierii stării rămâne nemodificat. Un exemplu al acestei abordări este descrierea unei stări sub forma unei mulțimi de predicate reprezentând faptele adevărate în acea stare. Fiecare stare

este reprezentată explicit ca parametru al predicatelor. De exemplu, starea curentă $S0$ ar putea fi caracterizată de

$$\text{ON}(A, B, S0) \wedge \text{ONTABLE}(B, S0) \wedge \text{CLEAR}(A, S0)$$

și regula care descrie operatorul $\text{UNSTACK}(x, y)$ va fi

$$\begin{aligned} \text{CLEAR}(x, s) \wedge \text{ON}(x, y, s) \rightarrow & [\text{HOLDING}(x, \text{DO}(\text{UNSTACK}(x, y), s)) \wedge \\ & \text{CLEAR}(y, \text{DO}(\text{UNSTACK}(x, y), s))] \end{aligned}$$

Aici DO este o funcție care specifică starea care rezultă din aplicarea unei anumite acțiuni asupra unei anumite stări. Dacă executăm $\text{UNSTACK}(A, B)$ în starea $S0$ atunci folosind axioma despre UNSTACK și ipoteza despre $S0$ putem demonstra că în starea $S1$ care rezultă din operația UNSTACK este valabil

$$\text{HOLDING}(A, S1) \wedge \text{CLEAR}(B, S1)$$

Despre $S1$ mai știm că B este pe masă, dar cu ceea ce avem până acum nu putem deduce aceasta. Pentru a permite astfel de deducții implicite avem nevoie de un set de reguli numite **axiomele cadrului** (*frame axioms*), care descriu acele componente ale stării care nu sunt afectate de operatori. De exemplu, avem nevoie să spunem că

$$\text{ONTABLE}(z, s) \rightarrow \text{ONTABLE}(z, \text{DO}(\text{UNSTACK}(x, y), s))$$

$$[\text{ON}(m, n, s) \wedge \neg \text{EQUAL}(m, x)] \rightarrow \text{ON}(m, n, \text{DO}(\text{UNSTACK}(x, y), s))$$

Avantajul acestei abordări este că un mecanism unic, rezoluția, poate realiza toate operațiile necesare pe descrierea stărilor. Totuși, prețul plătit este numărul foarte mare de axiome necesar dacă descrierile stărilor problemei sunt complexe.

Pentru gestionarea domeniilor complexe avem nevoie de un mecanism care nu cere un număr mare de axiome cadru explicite. Un exemplu de astfel de mecanism este cel folosit de sistemul de rezolvare a problemelor STRIPS și descendenții săi. Fiecare operator este descris de o listă de predicate noi pe care operatorul le face adevărate și o listă de predicate vechi pe care operatorul le face false. Cele două liste se numesc **ADD** și respectiv **DELETE**. Pentru fiecare operator este specificată și o a treia listă, **PRECONDITION**, care conține toate predicatele care trebuie să fie adevărate pentru ca operatorul să fie aplicabil. Axiomele cadrului sunt specificate implicit în STRIPS. Orice predicat neinclus în listele **ADD** sau **DELETE** ale unui operator nu este afectat de acel operator. Operatorii de genul STRIPS care corespund operațiilor pe care le-am discutat sunt următorii:

STACK(x, y)**P:** $CLEAR(y) \wedge HOLDING(x)$ **D:** $CLEAR(y) \wedge HOLDING(x)$ **A:** $ARMEMPTY \wedge ON(x, y)$ **UNSTACK(x, y)****P:** $ON(x, y) \wedge CLEAR(x) \wedge ARMEMPTY$ **D:** $ON(x, y) \wedge ARMEMPTY$ **A:** $HOLDING(x) \wedge CLEAR(y)$ **PICKUP(x)****P:** $ONTABLE(x) \wedge CLEAR(x) \wedge ARMEMPTY$ **D:** $ONTABLE(x) \wedge ARMEMPTY$ **A:** $HOLDING(x)$ **PUTDOWN(x)****P:** $HOLDING(x)$ **D:** $HOLDING(x)$ **A:** $ONTABLE(x) \wedge ARMEMPTY$

Prin faptul că axiomele cadrului devin implicite cantitatea de informație necesară pentru fiecare operator a fost redusă semnificativ. Aceasta înseamnă că la introducerea unui atribut nou nu este necesar să introducem o axiomă nouă pentru fiecare din operatorii existenți. În ceea ce privește utilizarea axiomelor cadrului în determinarea descrierilor de stări, să notăm că pentru descrieri complexe cea mai mare paore a descrierilor rămân nemodificate. Pentru a evita preluarea informației de la o stare la alta, va trebui să nu mai reprezentăm starea ca parte explicită a predicatelor. Astfel, vom avea o singură bază de date de predicate care întotdeauna descriu starea curentă. De exemplu, descrierea stării S0 din discuția anterioară va fi

$$ON(A, B) \wedge ONTABLE(B) \wedge CLEAR(A)$$

și descrierea stării obținute după aplicarea operatorului UNSTACK(A, B) va fi

$$ONTABLE(B) \wedge CLEAR(A) \wedge CLEAR(B) \wedge HOLDING(A)$$

Acest lucru se deduce prin utilizarea listelor ADD și DELETE specificate ca parte a operatorului UNSTACK.

Actualizarea descrierii unei stări unice este o modalitate bună de actualizare a efectelor unui șir de operatori dat. Dar în cadrul procesului de determinare a șirului de operatori corect, în situația explorării unui șir incorect trebuie să putem reveni la starea originală pentru a putea încerca un alt șir de operatori. Pentru aceasta tot ceea ce trebuie să facem este să memorăm la fiecare nod schimbările care s-au realizat în baza de date globală. Dar schimbările sunt specificate în listele ADD și DELETE ale fiecărui operator. Deci, tot ceea ce va trebui să indicăm este operatorul aplicat la fiecare nod, cu argumentele variabile legate de valori constante. De exemplu să considerăm aceeași stare inițială pe care am folosit-o în exemplele anterioare. De această dată o vom denumi “nodul 1”. Descrierea ei în format STRIPS este

$$\text{ON}(A, B) \wedge \text{ONTABLE}(B) \wedge \text{CLEAR}(A)$$

Asupra nodului 1 vom folosi operatorul UNSTACK(A, B) iar asupra nodului 2, produs de acesta vom folosi operatorul PUTDOWN(A). Descrierea stării corespunzătoare nodului 3 este următoarea:

$$\text{ONTABLE}(B) \wedge \text{CLEAR}(A) \wedge \text{CLEAR}(B) \wedge \text{ONTABLE}(A)$$

Dacă acum dorim să analizăm un alt drum, va trebui să facem backtracking peste nodul 3. Vom **adăuga** predicatele din lista DELETE ale operatorului PUTDOWN, folosit pentru a trece de la nodul 2 la nodul 3, și vom **șterge** predicatele din lista ADD ale aceluiași operator PUTDOWN. După acestea vom obține descrierea stării nodului 2, identică cu cea pe care am obținut-o în exemplul anterior, după aplicarea operatorului UNSTACK la situația inițială:

$$\text{ONTABLE}(B) \wedge \text{CLEAR}(A) \wedge \text{CLEAR}(B) \wedge \text{HOLDING}(A)$$

Deoarece în domeniile complexe o specificare implicită a axiomelor cadrului este foarte importantă, toate tehnicile pe care le vom analiza exploatează descrierile de tip STRIPS ale operatorilor disponibili.

Detectarea unei soluții

Cum vom ști dacă starea curentă este stare soluție? În sistemele simple de rezolvare a problemelor, se verifică pur și simplu descrierile stărilor. Dar dacă stările nu sunt reprezentate explicit ci folosind o mulțime de proprietăți relevante, atunci această problemă devine mai complexă și modul de rezolvare depinde de modul în care sunt reprezentate descrierile stărilor. Pentru a putea descoperi dacă două reprezentări se potrivesc, trebuie să putem raționa cu reprezentări.

O tehnică de reprezentare care a servit ca bază pentru multe sisteme de planificare este logica predicatelor, atractivă din cauza mecansimului inductiv pe care îl oferă. Să presupunem

că predicatul $P(x)$ este parte a stării finale. Pentru a vedea dacă $P(x)$ este satisfăcut într-o stare oarecare încercăm să demonstrăm $P(x)$ cunoscând declarațiile care descriu acea stare și axiomele care definesc domeniul problemei. Dacă putem construi o demonstrație, atunci procesul de rezolvare a problemei se termină. Dacă nu, atunci trebuie să propunem un șir de operatori care ar putea rezolva problema. Acest șir poate fi testat în același mod ca și starea inițială, prin încercarea de a demonstra $P(x)$ din axiomele și descrierea stării obținute prin aplicarea șirului respectiv de operatori.

Detectarea blocajelor

Pe măsură ce un sistem de planificare caută un șir de operatori care să rezolve o anumită problemă, trebuie să poată detecta când explorează un drum care nu poate conduce niciodată la soluție, sau care pare improbabil să conducă la soluție. Același mecanism de raționare care poate fi folosit la detectarea unei soluții poate fi deseori folosit și la detectarea unui blocaj.

Dacă procesul de căutare raționează înainte, poate elimina orice drum care conduce la o stare de unde starea finală nu poate fi atinsă precum și orice drum care, chiar dacă nu exclude o soluție, nu pare să conducă la o stare mai apropiată de soluție decât starea de origine.

De asemenea, dacă procesul de căutare raționează înapoi, poate opri un drum deoarece este sigur că starea inițială nu poate fi atinsă sau deoarece se realizează un progres prea mic. În raționarea înapoi obiectivul problemei se descompune în sub-obiective care, fiecare, pot conduce la un set de sub-obiective adiționale. Uneori este ușor să detectăm că nu există nici un mod în care să fie satisfăcute simultan toate sub-obiectivele dintr-o mulțime.

Corectarea unei soluții aproape corecte

Tehnicile pe care le vom discuta sunt deseori utile în rezolvarea problemelor **aproape decompozabile**. O modalitate de a rezolva astfel de probleme este să presupunem că sunt complet decompozabile, să rezolvăm subproblemele separat și să verificăm dacă după combinarea sub-soluțiilor, acestea conduc efectiv la o soluție a problemei originale. Desigur că dacă acest lucru este adevărat, nu mai este nimic altceva de făcut. Dar dacă nu este adevărat, avem la dispoziție o varietate de posibilități de continuare. Cea mai simplă este de a renunța la soluția propusă, să căutăm alta și să sperăm că va fi mai bună. Deși este o soluție simplă, poate să conducă la pierderea unei cantități mari de efort.

O abordare ceva mai bună este să analizăm situația produsă prin aplicarea șirului de operații care corespund soluției propuse și să comparăm această situație cu obiectivul urmărit. În cele mai multe cazuri diferența dintre cele două stări va fi mai mică decât diferența dintre starea inițială și cea finală. În acest moment sistemul de rezolvare a problemelor poate fi apelat din nou pentru a elimina această diferență nouă. Apoi prima soluție va fi combinată cu a doua pentru a forma o soluție a problemei originale.

O modalitate mai bună de a corecta o soluție aproape corectă este să utilizăm cunoștințe specifice despre ceea ce a funcționat greșit și să aplicăm o căutare directă. De exemplu să presupunem că motivul pentru care soluția propusă este inadecvată este că unul dintre operatori nu poate fi aplicat deoarece în momentul în care ar fi trebuit aplicat precondițiile sale nu erau satisfăcute. Aceasta se poate întâmpla dacă operatorul ar avea două precondiții iar șirul de operatori care face adevărată a doua precondiție o anulează pe prima. Dar, probabil, această problemă nu ar apărea dacă am încerca să satisfacem precondițiile în ordine inversă.

O modalitate și mai bună de a corecta soluții incomplete este să le lăsăm incomplet specificate până în ultimul moment posibil. Atunci, când este disponibilă cât de multă informație posibilă, vom completa specificația într-un astfel de mod încât să nu apară conflicte. Această abordare este numită **strategia celui mai mic angajament** (*least-commitment strategy*) și poate fi aplicată în mai multe moduri. Unul este de a amâna decizia asupra ordinii de realizare a operațiilor. În exemplul anterior, în loc să decidem în ce ordine vom satisface o mulțime de precondiții ar trebui să lăsăm ordinea nespecificată până la sfârșit. Apoi vom analiza efectele fiecărei substituții pentru a determina dependența dintre ele, după care vom putea alege și ordinea.

7.4. Planificare folosind stive de obiective

Una dintre primele tehnici dezvoltate pentru rezolvarea problemelor compuse care pot interacționa a fost utilizarea unei stive de obiective. Această abordare, folosită de STRIPS, se bazează pe utilizarea unei stive unice care conține atât scopuri cât și operatori care au fost propuși pentru a rezolva acele scopuri. Rezolvatorul problemei se bazează, de asemenea, pe o bază de date care descrie situația curentă și o mulțime de operatori descriși sub forma listelor PRECONDITION, ADD și DELETE. Pentru a vedea cum funcționează această metodă o vom aplica pe exemplul din Figura 7.1.

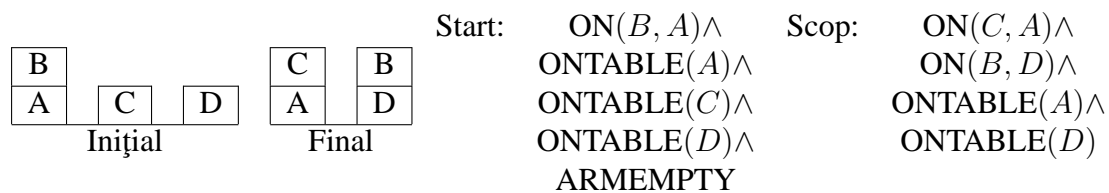


Figura 7.1: O problemă simplă din lumea blocurilor

La începutul procesului de rezolvare a problemei, stiva de scopuri este

$$ON(C, A) \wedge ON(B, D) \wedge ONTABLE(A) \wedge ONTABLE(D)$$

Dorim să separăm această problemă în patru subprobleme, câte una pentru fiecare componență a scopului original. Două din subprobleme, $ONTABLE(A)$ și $ONTABLE(D)$ sunt deja

adevărate în stare inițială. Deci vom lucra numai pe celelalte două. În funcție de ordinea în care dorim să abordăm subproblemele, pot fi create două stive de scopuri ca prim pas al procesului de rezolvare, după cum urmează (fiecare linie reprezintă un scop din stivă și OTAD este prescurtarea pentru $ONTABLE(A) \wedge ONTABLE(D)$):

$$\begin{array}{cc}
 ON(C, A) & ON(B, D) \\
 ON(B, D) & ON(C, A) \\
 ON(C, A) \wedge ON(B, D) \wedge OTAD & ON(C, A) \wedge ON(B, D) \wedge OTAD \\
 (1) & (2)
 \end{array}$$

La fiecare pas al procesului de rezolvare care se termină cu succes va fi analizat scopul din vârful stivei. Când se găsește un șir de operatori care satisface scopul, acela este aplicat descrierii stării, producând o nouă stare. Apoi, plecând de la această stare, scopul care este în acel moment în vârful stivei este explorat și se încearcă satisfacerea lui. Acest proces continuă până când stiva de scopuri este vidă. Apoi, ca o verificare finală, scopul original este comparat cu starea finală obținută prin aplicarea operatorilor selectați. Dacă există vre-o componentă a scopului care nu este satisfăcută de acea stare, lucru care s-ar putea întâmpla dacă componenta a fost produsă la un moment și anulată mai târziu, atunci acele părți nerezolvate sunt reinsertate în stivă și procesul este reluat.

Pentru a exemplifica această procedură să presupunem că decidem să explorăm alternativa 1. Alternativa 2 va produce o soluție atât de banală încât nu este interesantă pentru analiza noastră. În ceea ce privește explorarea alternativei 1, vom verifica mai întâi dacă $ON(C, A)$ este adevărat în starea curentă. Deoarece nu este, verificăm operatorii care ar putea face ca $ON(C, A)$ să fie adevărat. Din cei patru operatori pe care îi avem în vedere, doar unul este util, **STACK**, și va trebui aplicat cu parametrii C și A . Deci, plasăm pe stivă **STACK**(C, A) în loc de $ON(C, A)$, și rezultă

$$\begin{array}{c}
 \mathbf{STACK}(C, A) \\
 ON(B, D) \\
 ON(C, A) \wedge ON(B, D) \wedge OTAD
 \end{array}$$

STACK(C, A) a înlocuit $ON(C, A)$ deoarece după ce am executat **STACK** este sigur că $ON(C, A)$ va avea loc. Pentru a aplica **STACK**(C, A) condițiile sale trebuie să aibă loc, deci trebuie stabilite ca subscopuri. Din nou trebuie să separăm un scop compus,

$$CLEAR(A) \wedge HOLDING(C)$$

Pentru a decide ordinea în care vom verifica aceste subscopuri vom folosi cunoștințe euristice. **HOLDING**(x) este foarte ușor de obținut. Cel mult este necesar să punem jos alte blocuri și să luăm în mână blocul x . Dar **HOLDING** este de asemenea ușor de anulat. Pentru a realiza orice altă operație, robotul trebuie să-și folosească brațul. Ca atare, dacă vom încerca să

rezolvăm mai întâi **HOLDING**, după care robotul își va folosi brațul, este aproape sigur că în final **HOLDING** nu va mai fi adevărat. Deci, ne vom baza pe euristica conform căreia dacă **HOLDING** este unul din subscopurile de atins, ar trebui să fie analizat ultimul. O astfel de informație euristică poate fi conținută în lista de **PRECONDITII**, indicând predicatele pur și simplu în ordinea în care trebuie analizate. Aceasta produce următoarea stivă

$$\begin{aligned} & \text{CLEAR}(A) \\ & \text{HOLDING}(C) \\ & \text{CLEAR}(A) \wedge \text{HOLDING}(C) \\ & \text{STACK}(C, A) \\ & \text{ON}(B, D) \\ & \text{ON}(C, A) \wedge \text{ON}(B, D) \wedge \text{OTAD} \end{aligned}$$

În continuare vom vedea dacă **CLEAR(A)** este adevărat. Deoarece nu este, vom încerca să aplicăm singurul operator care îl va face adevărat. Aceasta produce stiva de scopuri:

$$\begin{aligned} & \text{ON}(B, A) \\ & \text{CLEAR}(B) \\ & \text{ARMEMPTY} \\ & \text{ON}(B, A) \wedge \text{CLEAR}(B) \wedge \text{ARMEMPTY} \\ & \text{UNSTACK}(B, A) \\ & \text{HOLDING}(C) \\ & \text{CLEAR}(A) \wedge \text{HOLDING}(C) \\ & \text{STACK}(C, A) \\ & \text{ON}(B, D) \\ & \text{ON}(C, A) \wedge \text{ON}(B, D) \wedge \text{OTAD} \end{aligned}$$

În acest moment comparăm elementul din vârful stivei de scopuri, **ON(B, A)**, cu ipoteza problemei și vedem că este satisfăcut și deci îl putem elimina din stivă. Și următorul element, **CLEAR(B)** este satisfăcut. Deși acest lucru nu este declarat explicit, poate fi demonstrat cu ușurință de către un demonstrator de teoreme. Deci eliminăm și **CLEAR(B)** din stivă. Al treilea element din stivă, **ARMEMPTY**, este și el adevărat și îl putem elimina. Următorul element din stivă este scopul combinat reprezentând toate condițiile lui **UNSTACK(B, A)**. Vom verifica dacă este satisfăcut. Verificarea trebuie făcută deoarece rezolvarea unuia dintre subscopuri este posibil să fi anulat soluția unui alt subscop. În acest caz nu este nici o problemă și eliminăm din stivă și scopul combinat.

În acest moment vârful stivei nu este un scop ci un operator, **UNSTACK(B, A)**. Condițiile sale sunt adevărate, deci îl putem aplica stării inițiale. Procesul de rezolvare va continua pe starea produsă prin această aplicare. Pe de o parte vom reține că primul operator al șirului de operatori ai soluției propuse este **UNSTACK(B, A)**. Baza de date care corespunde stării actuale este

$$\text{ONTABLE}(A) \wedge \text{ONTABLE}(C) \wedge \text{ONTABLE}(D) \wedge \text{HOLDING}(B) \wedge \text{CLEAR}(A)$$

Stiva de scopuri este

$$\begin{aligned} & \text{HOLDING}(C) \\ & \text{CLEAR}(A) \wedge \text{HOLDING}(C) \\ & \text{STACK}(C, A) \\ & \text{ON}(B, D) \\ & \text{ON}(C, A) \wedge \text{ON}(B, D) \wedge \text{OTAD} \end{aligned}$$

În continuare încercăm să satisfacem $\text{HOLDING}(C)$. Există doi operatori care ar putea face pe $\text{HOLDING}(C)$ adevărat: $\text{PICKUP}(C)$ și $\text{UNSTACK}(C, x)$, unde x poate fi orice bloc diferit de C . Fără să aruncăm o privire mai departe nu ne putem da seama care este operatorul potrivit. Vom crea două ramuri ale arborelui de căutare, corespunzând următoarelor două stive de scopuri:

$\text{ONTABLE}(C)$	$\text{ON}(C, x)$
$\text{CLEAR}(C)$	$\text{CLEAR}(C)$
ARMEMPTY	ARMEMPTY
$\text{ONTABLE}(C) \wedge \text{CLEAR}(C) \wedge$	$\text{ONTABLE}(C) \wedge \text{CLEAR}(C) \wedge$
ARMEMPTY	ARMEMPTY
$\text{PICKUP}(C)$	$\text{UNSTACK}(C, x)$
$\text{CLEAR}(A) \wedge \text{HOLDING}(C)$	$\text{CLEAR}(A) \wedge \text{HOLDING}(C)$
$\text{STACK}(C, A)$	$\text{STACK}(C, A)$
$\text{ON}(B, D)$	$\text{ON}(B, D)$
$\text{ON}(C, A) \wedge \text{ON}(B, D) \wedge \text{OTAD}$	$\text{ON}(C, A) \wedge \text{ON}(B, D) \wedge \text{OTAD}$
(1)	(2)

Să observăm că pentru alternativa 2, stiva conține variabila x care apare în trei locuri și trebuie să se potrivească de fiecare dată cu același bloc. Este important ca de fiecare dată când se introduce o variabilă, numele ei să nu coincidă cu numele nici unei variabile existente. În plus, de fiecare dată când un obiect este potrivit cu o variabilă, legarea trebuie memorată astfel încât următoarele apariții ale variabilei să fie legate la același obiect.

Cum decidem care dintre cele două alternative este de preferat? În ceea ce privește alternativa 2, trebuie să luăm blocul C de pe un bloc oarecare (UNSTACK). Dar pentru a putea face acest lucru, el trebuie să fie pe un bloc. Ori, în acest moment, C este pe masă. Ar însemna că pentru a îl putea lua de pe un bloc va trebui să îl punem pe bloc, ceea ce pare pierdere de timp. Deci ar trebui urmată alternativa 1. Dar programul nu poate face o astfel de analiză. Să presupunem totuși, că urmăm alternativa 2. Pentru a satisface $\text{ON}(C, x)$ trebuie să aplicăm $\text{STACK}(C, x)$. Deci stiva de scopuri va fi următoarea:

$$\begin{aligned}
& \text{CLEAR}(x) \\
& \text{HOLDING}(C) \\
& \text{CLEAR}(x) \wedge \text{HOLDING}(C) \\
& \text{STACK}(C, x) \\
& \text{CLEAR}(C) \\
& \text{ARMEMPTY} \\
& \text{ON}(C, x) \wedge \text{CLEAR}(C) \wedge \text{ARMEMPTY} \\
& \text{UNSTACK}(C, x) \\
& \text{CLEAR}(A) \wedge \text{HOLDING}(C) \\
& \text{STACK}(C, A) \\
& \text{ON}(B, D) \\
& \text{ON}(C, A) \wedge \text{ON}(B, D) \wedge \text{OTAD}
\end{aligned}$$

Dar să notăm acum că una din condițiile lui **STACK** este **HOLDING(C)**. Aceasta este ceea ce am încercat să obținem prin aplicarea lui **UNSTACK**, care ne-a cerut să aplicăm **STACK**. Deci suntem de unde am plecat. De fapt, avem câteva condiții în plus. În acest punct acest drum poate fi terminat ca neproductiv și ne putem întoarce la alternativa 1.

Alternativa 1 folosește **PICKUP** pentru a ajunge în situația ca robotul să țină în mână blocul C. Vârful stivei, **ONTABLE(C)**, este satisfăcut și poate fi eliminat. Următorul element, **CLEAR(C)** și el este satisfăcut, și va fi eliminat. Următoarea condiție este **ARMEMPTY**, care nu este satisfăcut, deoarece **HOLDING(B)** este adevărat. Doi operatori îl pot face adevărat pe **ARMEMPTY**: **STACK(B, x)** și **PUTDOWN(B)**, adică fie îl punem pe B peste un alt bloc, fie îl punem direct pe masă. Dacă ne-am uita înainte, am vedea că în final trebuie să-l punem pe B peste D, lucru pe care îl vom face chiar acum. Programul poate ajunge la concluzia asta prin compararea elementelor listei **ADD** a operatorilor posibili cu restul stivei de scopuri. Dacă unul din operatori are efectul de a face adevărate și alte scopuri, acest operator ar trebui ales. Deci vom aplica **STACK(B, D)** și vom lega pe x de D. Rezultatul este stiva de scopuri

$$\begin{aligned}
& \text{CLEAR}(D) \\
& \text{HOLDING}(B) \\
& \text{CLEAR}(D) \wedge \text{HOLDING}(B) \\
& \text{STACK}(B, D) \\
& \text{ONTABLE}(C) \wedge \text{CLEAR}(C) \wedge \text{ARMEMPTY} \\
& \text{PICKUP}(C) \\
& \text{CLEAR}(A) \wedge \text{HOLDING}(C) \\
& \text{STACK}(C, A) \\
& \text{ON}(B, D) \\
& \text{ON}(C, A) \wedge \text{ON}(B, D) \wedge \text{OTAD}
\end{aligned}$$

CLEAR(D) și **HOLDING(B)** sunt adevărate, precum și scopul compus următor. Putem aplica operația **STACK(B, D)** care va produce următoarea stare:

$$\text{ONTABLE}(A) \wedge \text{ONTABLE}(C) \wedge \text{ONTABLE}(D) \wedge \text{ON}(B, D) \wedge \text{ARMEMPTY}$$

Toate condițiile lui PICKUP(C) sunt satisfăcute, și, după eliminarea lor din stivă putem executa și PICKUP(C). În continuare, și condițiile lui STACK(C, A) sunt satisfăcute și, ca urmare, și STACK(C, A) va fi executat. În continuare ne vom ocupa de al doilea subscop inițial, ON(B, D). Acesta este deja satisfăcut de una din operațiile efectuate pentru satisfacerea primului subscop. Deci eliminăm ON(B, D) din stivă. Vom verifica și scopul combinat, ultimul din stivă. Deoarece și acesta este adevărat, procesul de rezolvare se oprește aici, iar planul raportat va fi

1. UNSTACK(B, A)
2. STACK(B, D)
3. PICKUP(C)
4. STACK(C, A)

În acest exemplu am văzut un mod în care informația euristică poate fi aplicată pentru a ghida procesul de căutare, un mod în care un drum neprofitabil poate fi detectat și un mod în care considerarea unor interacțiuni între scopuri poate ajuta efortul de a produce o soluție globală bună. Pentru probleme mai dificile aceste metode pot eșua. Pentru a vedea de ce, vom încerca să rezolvăm problema din Figura 7.2.

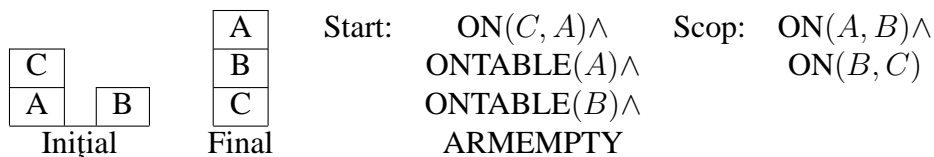


Figura 7.2: O problemă ceva mai dificilă din lumea blocurilor

Există două moduri în care putem începe rezolvarea problemei. Aceste moduri corespund următoarelor două stive de scopuri:

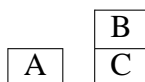
$\text{ON}(A, B)$	$\text{ON}(B, C)$
$\text{ON}(B, C)$	$\text{ON}(A, B)$
$\text{ON}(A, B) \wedge \text{ON}(B, C)$	$\text{ON}(A, B) \wedge \text{ON}(B, C)$
(1)	(2)

Să presupunem că alegem alternativa 1. La un moment dat vom ajunge la următoarea stivă de scopuri:

$$\begin{aligned} & \text{ON}(C, A) \\ & \text{CLEAR}(C) \\ & \text{ARMEMPTY} \\ & \text{ON}(C, A) \wedge \text{CLEAR}(C) \wedge \text{ARMEMPTY} \\ & \text{UNSTACK}(C, A) \\ & \text{ARMEMPTY} \\ & \text{CLEAR}(A) \wedge \text{ARMEMPTY} \\ & \text{PICKUP}(A) \\ & \text{CLEAR}(B) \wedge \text{HOLDING}(A) \\ & \text{STACK}(A, B) \\ & \text{ON}(B, C) \\ & \text{ON}(A, B) \wedge \text{ON}(B, C) \end{aligned}$$

5. UNSTACK(A, B)
6. PUTDOWN(A)
7. PICKUP(B)
8. STACK(B, C)

Acum starea problemei este prezentată în Figura 7.4



$$\text{ON}(B, C) \wedge \text{ONTABLE}(A) \wedge \text{ONTABLE}(C) \wedge \text{ARMEMPTY}$$

Figura 7.4: Starea problemei ceva mai târziu

Acum vom avea de verificat scopul final de pe stivă,

$$\text{ON}(A, B) \wedge \text{ON}(B, C)$$

Vom descoperi că nu este satisfăcut. În procesul rezolvării scopului $\text{ON}(B, C)$ am anulat scopul $\text{ON}(A, B)$. Diferența dintre scopul nostru și starea curentă este $\text{ON}(A, B)$, pe care îl vom plasa din nou pe stivă. Se găsește secvența de operatori

9. PICKUP(A)
10. STACK(A, B)

Scopul combinat este din nou verificat. De această dată este satisfăcut. Planul complet descoperit de noi este următorul:

1. UNSTACK(C, A)
2. PUTDOWN(C)
3. PICKUP(A)
4. STACK(A, B)
5. UNSTACK(A, B)
6. PUTDOWN(A)
7. PICKUP(B)

8. STACK(B, C)
9. PICKUP(A)
10. STACK(A, B)

Deși acest plan rezolvă problema, nu este foarte eficient. Metoda pe care am folosit-o nu este capabilă să găsească planuri eficiente pentru rezolvarea acestei probleme.

Există două abordări pe care le putem avea în vedere pentru producerea unui plan bun. Prima este să căutăm moduri în care să corectăm un plan pentru a îl face mai eficient. În acest caz, această operație este ușoară. Vom căuta în plan locurile în care am efectuat o operație după care am anulat-o imediat. Dacă găsim astfel de locuri, ambele operații pot fi eliminate din plan. Dacă aplicăm această regulă planului nostru, vom vedea că putem elimina pașii 4 și 5, după care vom elimina pașii 3 și 6. Planul rezultat

1. UNSTACK(C, A)
2. PUTDOWN(C)
3. PICKUP(B)
4. STACK(B, C)
5. PICKUP(A)
6. STACK(A, B)

conține numărul minim de operatori necesari pentru a rezolva problema. Dar pentru aplicații mai complexe operațiile care interferează pot fi mult distanțate în plan și astfel mult mai dificil de detectat. În plus am consumat un efort substanțial pentru producerea unor pași care ulterior au fost anulați. Ar fi mult mai bine dacă am putea identifica o procedură de construire a planurilor care să construiască direct planuri eficiente. O astfel de procedură este descrisă în secțiunea următoare.

7.5. Planificare neliniară folosind declararea limitărilor

Metoda de planificare folosind stive de scopuri abordează problemele care implică scopuri multiple prin rezolvarea scopurilor în ordine, câte unul la un moment dat. Un plan generat de această metodă conține un șir de operatori pentru atingerea primului scop, urmat de un șir complet de operatori pentru al doilea scop, ș.a.m.d. Așa cum am văzut problemele dificile creează interacțiuni între scopuri. Operatorii folosiți pentru rezolvarea unei subprobleme pot interfera cu soluția unei subprobleme anterioare. Majoritatea problemelor cer un plan combinat

în care mai multe subprobleme sunt abordate în același timp. Un astfel de plan se numește **plan neliniar** deoarece nu este compus dintr-un șir liniar de subplanuri complete.

Ca un exemplu al necesității unui plan neliniar să revenim la problema descrisă în Figura 7.2. Un plan bun pentru soluția la această problemă este următorul:

1. Începe lucrul la scopul $ON(A, B)$ prin descoperirea lui A prin plasarea lui C pe masă.
2. Rezolvă scopul $ON(B, C)$ prin plasarea lui B peste C.
3. Continuă rezolvarea scopului $ON(A, B)$ prin plasarea lui A peste B.

Vom explora în continuare câteva euristici și algoritmi pentru rezolvarea problemelor neliniare, cum este și aceasta. Printre soluțiile avute în vedere amintim: modificarea algoritmului stivelor de scopuri pentru a opera cu **mulțimi de scopuri** și utilizarea **declarării limitărilor** (*constraint posting*).

Ideea declarării limitărilor, tehnică utilizat la sistemele MOLGEN și TWEAK, este de a construi un plan aplicarea incrementală a operatorilor, ordonarea parțială a operatorilor și legarea variabilelor cu operatorii. În orice moment în cadrul procesului de rezolvare putem avea o mulțime de operatori utili dar probabil că nu avem nici o idee despre modul în care aceștia trebuie ordonați. O soluție este o mulțime parțial ordonată și parțial instanțiată de operatori. Pentru a genera un plan real convertim ordinea parțială în oricare dintr-un număr de ordini totale. Iată pe scurt diferența dintre metoda de declarare a limitărilor și metodele de planificare discutate până acum:

	Căutare în spațiul de stări	Căutare prin declararea limitărilor
Deplasările în spațiu	Modifică starea prin intermediul operatorilor	Adăugarea operatorilor Ordonarea operatorilor Legarea variabilelor Alt mod de a limita planul
Modelarea timpului	Adâncimea nodului în spațiul de căutare	Mulțime de operatori parțial ordonată
Planul memorat ca	O serie de tranziții de stări	Un singur nod

În continuare vom examina câteva operații de planificare neliniară într-un mediu de declarare a limitărilor. Să generăm în mod incremental un plan neliniar pentru rezolvarea problemei din Figura 7.2. Vom începe cu un plan vid. Vom analiza starea finală și vom fixa pașii pentru atingerea scopului. Analiza *means-ends* ne spune să alegem doi pași cu postcondițiile $ON(A, B)$ și $ON(B, C)$:

CLEAR(B)	CLEAR(C)
*HOLDING(A)	*HOLDING(B)
STACK(A, B)	STACK(B, C)
ARMEMPTY	ARMEMPTY
ON(A, B)	ON(A, B)
\neg CLEAR(B)	\neg CLEAR(B)
\neg HOLDING(A)	\neg HOLDING(A)

Fiecare pas este scris cu condițiile deasupra sa și postcondițiile dedesubt. Postcondițiile care trebuie șterse sunt marcate cu simbolul negării (\neg). În acest moment cei doi pași nu sunt ordonați unul față de celălalt. Tot ceea ce știm este că într-un târziu va trebui să îi executăm pe amândoi. Nici unul nu poate fi executat imediat deoarece unele din condițiile lor nu sunt satisfăcute. O condiție nesatisfăcută este marcată cu o stea (*). Ambele condiții *HOLDING nu sunt satisfăcute deoarece la momentul inițial brațul robotului este gol.

Introducerea unor pași noi pentru rezolvarea scopurilor sau condițiilor se numește **adăugarea pașilor**, și este una din euristicele pe care le vom folosi în generarea planurilor neliniare. Iată euristicele pe care le vom folosi în acest exemplu:

1. **Adăugarea pașilor** (*step addition*) – Crearea unor pași noi pentru un plan.
2. **Promovarea** (*promotion*) – Introducerea limitării ca un pas să apară înaintea altuia într-un plan final.
3. **Revalidarea** (*declobbering*) – Plasarea unui pas (posibil nou) s_2 între doi pași mai vechi s_1 și s_3 , astfel încât s_2 revalidează unele condiții ale lui s_3 care au fost negate de s_1 .
4. **Stabilirea simplă** (*simple establishment*) – Atribuirea unei valori pentru o variabilă pentru a asigura condițiile unui anumit pas.
5. **Separarea** (*separation*) – Prevenirea atribuirii anumitor valori pentru o variabilă.

Pentru a obține condițiile celor doi pași specificați mai sus putem folosi din nou adăugarea unor pași:

*CLEAR(B)	*CLEAR(B)
ONTABLE(A)	ONTABLE(B)
*ARMEMPTY	*ARMEMPTY
PICKUP(A)	PICKUP(B)
\neg ONTABLE(A)	\neg ONTABLE(B)
\neg ARMEMPTY	\neg ARMEMPTY
HOLDING(A)	HOLDING(B)

Adăugarea acestor doi pași nu este suficientă pentru satisfacerea precondițiilor pașilor STACK, deoarece nu există condiționări de ordonare între acești pași. Dacă într-un plan eventual pașii PICKUP ar fi executați după pașii STACK, atunci precondițiile pașilor STACK ar trebui să fie satisfăcute de o altă mulțime de pași. Astfel vom introduce condiționări de ordonare oricând utilizăm adăugarea pașilor. În acest caz dorim să spunem că fiecare pas PICKUP trebuie executat înaintea pasului STACK corespunzător:

$$\text{PICKUP}(A) \leftarrow \text{STACK}(A, B)$$

$$\text{PICKUP}(B) \leftarrow \text{STACK}(B, C)$$

Acum avem în planul nostru patru pași parțial ordonați și patru precondiții nerezolvate. Condiția *CLEAR(A) este nerezolvată deoarece blocul A nu este descoperit în starea inițială și condiția *CLEAR(B) este nerezolvată pentru că, deși blocul B este descoperit în starea inițială, în plan există pasul STACK(A, B) cu postcondiția $\neg \text{CLEAR}(B)$, și acel pas ar putea precede pasul care are *CLEAR(B) ca precondiție. Pentru a obține precondiția CLEAR(B) utilizăm o a doua euristică cunoscută sub numele de **promovare**, ceea ce înseamnă că vom declara o limitare că în plan un anumit pas trebuie să preceadă un alt pas. Putem obține CLEAR(B) dacă declarăm că pasul PICKUP(B) trebuie să apară înaintea pasului STACK(A, B):

$$\text{PICKUP}(B) \leftarrow \text{STACK}(A, B)$$

Să analizăm cele două precondiții nerealizate *ARMEMPTY. În timp ce în starea inițială robotul are brațul gol, fiecare din cei doi operatori PICKUP conține postcondiția $\neg \text{ARMEMPTY}$. Fiecare operator poate face ca celălalt să nu mai fie executat. Putem folosi promovarea pentru a obține cel puțin una din cele două precondiții:

$$\text{PICKUP}(B) \leftarrow \text{PICKUP}(A)$$

Deoarece situația inițială conține un braț gol și nici un pas care precede PICKUP(B) nu poate anula acest lucru, toate precondițiile lui PICKUP(B) sunt satisfăcute.

O a treia euristică, numită **revalidare**, ne poate ajuta să obținem precondiția *ARMEMPTY a pasului PICKUP(A). PICKUP(B) produce $\neg \text{ARMEMPTY}$, dar dacă putem insera între pașii PICKUP(B) și PICKUP(A) un alt pas care să facă valid ARMEMPTY, atunci precondiția va fi satisfăcută. Pasul inserat va fi STACK(B, C), deci declarăm limitarea:

$$\text{PICKUP}(B) \leftarrow \text{STACK}(B, C) \leftarrow \text{PICKUP}(A)$$

Singura precondiție nerezolvată este *CLEAR(A), din pasul PICKUP(A). Pentru obținerea ei putem folosi adăugarea pașilor:

$$\begin{array}{c}
*ON(x, A) \\
*CLEAR(x) \\
*ARMEMPTY \\
\hline
UNSTACK(x, A) \\
\hline
\neg ARMEMPTY \\
CLEAR(A) \\
HOLDING(A) \\
\neg ON(x, A)
\end{array}$$

Introducem variabila x deoarece singura postcondiție care ne interesează este $CLEAR(A)$. Blocul de pe A este nerelevant. Declararea limitărilor ne permite să creăm planuri incomplete relativ la ordinea pașilor. Variabilele ne permit să evităm să facem instanțieri de operatori nenecesare.

Din păcate acum avem trei precondiții nerezolvate. Putem obține $ON(x, A)$ cu ușurință prin limitarea valorii lui x la blocul C . Este în regulă, deoarece blocul C este pe blocul A în starea inițială. Această euristică se numește **stabilirea simplă** și, în forma sa cea mai generală ne permite să declarăm că două propoziții diferite trebuie să fie instanțiate la aceeași propoziție. În cazul nostru

$$x = C \text{ în pasul } UNSTACK(x, A)$$

Există pași care anulează precondițiile $CLEAR(C)$ și $ARMEMPTY$, dar putem folosi promovarea pentru a avea grijă de acest lucru:

$$UNSTACK(x, A) \leftarrow STACK(B, C)$$

$$UNSTACK(x, A) \leftarrow PICKUP(A)$$

$$UNSTACK(x, A) \leftarrow PICKUP(B)$$

Dintre euristicile pe care le-am analizat până acum adăugarea unui pas este cea mai problematică deoarece întotdeauna trebuie să verificăm dacă noul pas invalidează unele precondiții ale unui pas existent. Pasul $PICKUP(B)$ cere $ARMEMPTY$, dar acesta este anulat de noul pas $STACK(x, A)$. Un mod de a rezolva problema apărută este să adăugăm la plan un pas de revalidare:

$$\begin{array}{c}
HOLDING(C) \\
\hline
PUTDOWN(C) \\
\hline
\neg HOLDING(C) \\
ONTABLE(x) \\
ARMEMPTY
\end{array}$$

Acest pas este ordonat astfel:

$$\text{UNSTACK}(x, A) \leftarrow \text{PUTDOWN}(C) \leftarrow \text{PICKUP}(B)$$

Să remarcăm că am văzut două tipuri de revalidări: unul în care un pas existent este folosit la revalidarea altuia, și un al doilea în care este introdus un pas de revalidare nou. În acest moment precondițiile pasului PUTDOWN tocmai introdus sunt satisfăcute. Sunt satisfăcute și toate precondițiile ale tuturor pașilor, deci problema este rezolvată. Rămâne să folosim ordonările și legările de variabile indicate în plan pentru a construi un plan concret:

1. UNSTACK(C, A)
2. PUTDOWN(C)
3. PICKUP(B)
4. STACK(B, C)
5. PICKUP(A)
6. STACK(A, B)

În această problemă am folosit patru euristici: adăugarea pașilor, promovarea, revalidarea și stabilirea simplă. Aceste operații, aplicate în ordinea corectă, nu sunt întotdeauna suficiente pentru a rezolva orice problemă de planificare neliniară. Pentru aceasta este nevoie și de a cincea euristică, **separarea**. Să presupunem că pasul C1 este posibil să preceadă pasul C2 și că C1 este posibil să nege o precondiție a lui C2. Spunem “posibil” deoarece propozițiile pot conține variabile. Separarea ne permite să stabilim o limitare care precizează că cele două propoziții nu trebuie instanțiate în același mod într-un plan.

Planificatorul TWEAKS împreună cu cele cinci operații de modificare a planului sunt suficiente pentru rezolvarea **oricărei** probleme neliniare rezolvabile. Algoritmul care exploatează operațiile de modificare a planului este simplu.

Algoritm: Planificare neliniară (TWEAK)

1. Inițializează S la mulțimea propozițiilor din starea scop.
2. Șterge din S o propoziție nerezolvată P.
3. Rezolvă P folosind una din cele cinci operații de modificare a planului.
4. Revizuieste toți pașii din plan incluzând orice pas introdus prin adăugarea pașilor, pentru a vedea dacă există vre-o precondiție nesatisfăcută. Adaugă la S noua mulțime de precondiții nesatisfăcute.

5. Dacă S este vidă, termină planul prin convertirea ordinii parțiale de pași într-o ordine totală și instanțiază orice variabile după cum este necesar.
6. Altfel mergi la pasul 2.

Nu orice șir de operații de modificare a planului conduce la o soluție. Nedeterminismul pașilor 2 și 3 trebuie implementat ca un fel de procedură de căutare, care poate fi ghidată de euristici. De exemplu, dacă atât promovarea cât și adăugarea unui pas rezolvă o anumită dificultate, atunci probabil este bine ca mai întâi să se încerce promovarea. TWEAK folosește backtracking de tip Breadth-First direcționat de dependențe precum și euristici de ordine.

7.6. Planificare ierarhică

Pentru a rezolva probleme dificile un rezolvitor de probleme ar putea să fie nevoit să genereze planuri lungi. Pentru a face aceasta în mod eficient este important să poată elimina unele detalii ale problemei până când se va găsi o soluție care abordează aspectele principale. Apoi se poate face o încercare de a completa și detaliile corespunzătoare. Primele astfel de încercări au implicat utilizarea macro-operatorilor, în care operatori mari erau construiți din operatori mai mici. Dar în acest fel din descrierea operatorilor nu a fost eliminat nici un detaliu. O abordare mai bună a fost dezvoltată în sistemul ABSTRIPS, care planifica într-o ierarhie de **spații abstracte**. În fiecare dintre acestea erau ignorate condițiile aflate la un nivel de abstractizare mai scăzut.

De exemplu să presupunem că dorim să vizităm un prieten aflat într-o altă țară, dar avem la dispoziție o sumă de bani limitată. Ete rezonabil să verificăm mai întâi prețul biletelor de avion sau tren, deoarece partea cea mai dificilă a problemei este să identificăm o modalitate de transport rezonabilă. Până când am rezolvat această parte a problemei nu are rost să ne preocupăm de planficarea drumului până la aeroport sau gară și de rezolvarea parcurii.

Modalitatea de rezolvare folosită de ABSTRIPS este următoarea: Se atribuie fiecărei condiții o **valoare critică** care reflectă dificultatea presupusă de satisfacere a condiției. Se rezolvă problema complet considerând doar condițiile ale căror valori critice sunt cele mai mari posibile. Pentru aceasta se operează la fel ca STRIPS dar condițiile necritice sunt ignorate. După aceasta folosește planu construit ca schelet pentru un plan complet și consideră condițiile de la nivelul critic imediat următor. Planul se va extinde cu operatori care satisfac aceste condiții. Din nou, în selectarea operatorilor sunt ignorate toate condițiile mai puțin critice decât nivelul considerat. Acest proces este continuat prin luarea în considerare a condițiilor din ce în ce mai puțin critice până ce toate au fost considerate toate condițiile regulilor originale.

Deoarece explorează în întregime planuri la un anumit nivel de detaliu înainte să analizeze detaliile de nivel mai scăzut, acest proces a fost numit **căutare Length-First**. Este evident că atribuirea celor mai potrivite valorilor critice este esențială pentru succesul acestei metode de

planificare ierarhică. Acele precondiții pe care nici un operator nu le poate satisface sunt cele mai critice.

7.7. Sisteme reactive

Până acum am descris un proces de planificare deliberativ, în care înainte de a acționa se construiește un plan pentru realizarea întregului obiectiv. Problema de a decide ce anume să facem poate fi abordată într-un mod cu totul diferit. Ideea **sistemelor reactive** este de a evita complet planificarea și de a folosi situația observabilă ca o indicație la care să putem reacționa.

Un sistem reactiv trebuie să aibă acces la o bază de cunoștințe care să descrie acțiunile care trebuie luate și circumstanțele în care aceste acțiuni trebuie luate. Un sistem reactiv este foarte diferit de celelalte tipuri de sisteme de planificare pe care le-am discutat deoarece alege câte o acțiune la un moment dat și nu anticipează și selectează un șir întreg de acțiuni înainte de a face primul pas.

Un exemplu simplu de sistem reactiv este termostatul. Obiectivul unui termostat este de a păstra constantă temperatura dintr-o încăpere. Putem imagina o soluție care cere un efort substanțial de planificare și care să țină seama de modul de creștere și scădere a temperaturii în timpul zilei, de modul în care căldura se disipează în încăpere și între încăperi, ș.a.m.d. Dar un termostat real folosește reguli simple de forma perechilor **situație-acțiune**:

1. Dacă temperatura în încăpere este cu k grade peste temperatura dorită, atunci pornește instalația de aer condiționat.
2. Dacă temperatura în încăpere este cu k grade sub temperatura dorită, atunci oprește instalația de aer condiționat.

Se dovedește că sistemele reactive sunt capabile să surprindă comportări surprinzător de complexe, în special în aplicații reale cum ar fi deplasarea unui robot. Principalul avantaj al sistemelor reactive asupra planificatorilor tradiționali este că sistemele reactive operează robust în domenii care sunt dificil de modelat absolut complet și absolut corect. Sistemele reactive renunță complet la modelare și își bazează acțiunile pe percepția lor. În domeniile complexe și imprevizibile abilitatea de a planifica un șir exact de pași înainte ca aceștia să fie aplicați are o valoare înouă. Un alt avantaj al sistemelor reactive este că au o abilitate de răspuns foarte mare, deoarece evită explozia combinatorială implicată în planificarea deliberată. Aceasta le face atractive pentru aplicații în timp real precum conducerea unei mașini și mersul pe jos.

Multe aplicații de IA solicită o deliberare semnificativă care este de obicei implementată ca un proces de căutare internă. Deoarece sistemele reactive nu se bazează pe un model al lumii sau pe structuri explicite ale obiectivului urmărit, performanța lor în astfel de aplicații este limitată.

Eforturile de a înțelege comportarea sistemelor rective au servit la ilustrarea multor limitări ale planificatorilor tradiționali. De exemplu, este vital să alternăm planificarea și execuția planului. Un sistem inteligent cu resurse limitate trebuie să decidă când să înceapă să gândească, când să nu mai gândească și când să acționeze. De asemenea, când sistemul interacționează cu mediul obiectivele de urmărit apar în mod natural. Este nevoie de un mecanism de suspendare a planificării pentru ca sistemul să se poată concentra asupra obiectivelor cu prioritate mare. Unele situații cer atenție imediată și acțiune rapidă. Pentru acest motiv unii planificatori deliverativi includ subsisteme reactive (adică mulțimi de reguli de genul situație-acțiune) bazate pe experiența de rezolvare a problemelor. Astfel de sisteme în timp învață să fie reactive.

7.8. Alte tehnici de planificare

Alte tehnici de planificare pe care nu le-am discutat includ următoarele:

Tabele triunghiulare – Oferă un mod de a înregistra obiectivele pe care fiecare operator trebuie să le satisfacă și obiectivele care trebuie să fie adevărate pentru ca operatorul să poată fi executat corect. Dacă se întâmplă ceva neprevăzut în timpul execuției unui plan, tabelul oferă informațiile necesare pentru a corecta planul.

Metaplanificare – O tehnică de raționare nu doar despre problema de rezolvat, ci și despre însuși procesul de planificare.

Macro-operatori – Permit unui planificator să construiască noi operatori care reprezintă șiruri de operatori utilizate frecvent.

Planificare bazată pe cazuri – Reutilizează planurile vechi pentru a construi planuri noi.

7.9. Probleme propuse

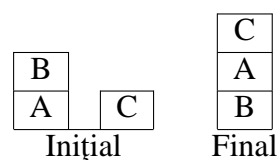


Figura 7.5: Configurația problemei 7.9.1

7.9.1. Se consideră problema din lumea blocurilor prezentată în Figura 7.5. Se urmărește să se ajungă din starea inițială în starea finală, folosind un algoritm de planificare.

(a) Arătați cum rezolvă sistemul STRIPS problema.

(b) Arătați cum rezolvă sistemul TWEAK problema.

Capitolul 8

Agenți inteligenți

Toate problemele inteligenței artificiale pot fi reunite sub conceptul de *agent inteligent*. Inteligența artificială se ocupă cu descrierea și construirea agenților care primesc percepții de la mediul înconjurător și execută acțiuni.

8.1. Conceptul de agent inteligent

Un **agent** este o entitate ce poate *percepe* mediul său înconjurător prin senzorii săi și poate *acționa* asupra acestuia prin *acțiuni*. *Agenții inteligenți* reprezintă un domeniu al Inteligenței Artificiale ce oferă o nouă metodă de rezolvare a problemelor și un nou mod de interacțiune între calculator și utilizator.

Un agent uman are ochi, urechi și alte organe pentru simțuri (senzori) și picioare, mâini, gură și alte părți ale corpului ca efectori. Un agent robotic poate substitui camera de luat vederi și lentilele infraroșii ca senzori și motoarele ca efectori. Un agent generic este prezentat în Figura 8.1. Scopul programatorului este de a proiecta agenți care se comportă bine în mediul lor.

Una din sarcinile unui *agent* este de a asista utilizatorul realizând sarcini în locul acestuia, sau învățându-l pe utilizator ce trebuie să facă.

8.2. Structura agenților inteligenți

Un *agent* este caracterizat prin

- partea de *arhitectură*, așa numitul *comportament* al agentului – acțiunea efectuată în urma unei secvențe perceptuale;
- partea de *program*, așa numita parte internă a agentului.

Sarcina Inteligenței Artificiale este legată de realizarea părții de program a agentului: o funcție care implementează modul de trecere de la percepție la acțiune. Acest program trebuie

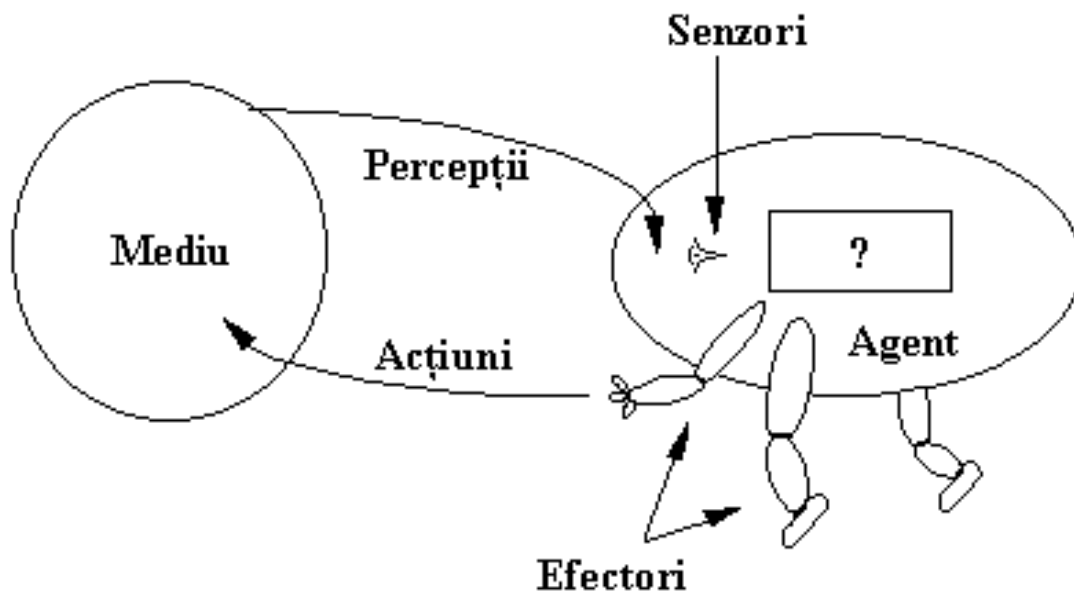


Figura 8.1: Agent interacționând cu mediul prin intermediul senzorilor și efectorilor

să fie compatibil cu arhitectura agentului. Arhitectura realizează interfața între percepția dată de senzori și program, rulează programul și asigură efectuarea acțiunilor alese pe măsură ce acestea sunt generate. Arhitectura poate fi realizată folosind diferite tehnici cum ar fi scenariile, cadrele.

Un *agent rațional ideal* va fi considerat acela care acționează în sensul maximizării performanțelor sale, pe baza informațiilor pe care le are despre mediul (furnizate de secvența perceptuală) și pe baza cunoașterii proprii pe care o are încorporată în structura sa internă.

Astfel, un agent inteligent artificial va fi înzestrat cu o cunoaștere inițială și cu capacitatea de a învăța. Învățarea asigură *autonomia* agentului – capacitatea de a-și deduce comportamentul din propria sa experiență. Agenții care operează doar pe baza cunoștințelor inițiale pe care le dețin vor fi numiți *neautonomi*, datorită faptului că ei vor acționa cu succes doar în măsura în care aceste cunoștințe rămân valabile, pierzându-și astfel flexibilitatea.

Un agent inteligent are deasemenea o *funcție de performanță (utilitate)*, care măsoară performanța acțiunilor sale. Funcția de performanță este, în mod obișnuit, unică pentru toți agenții care acționează într-un mediu dat. Performanța este dealtfel o funcție care asociază unei stări un număr real, ca fiind o măsură a gradului de succes al stării (o stare pe care agentul o preferă în raport cu alte stări are funcția de performanță mai mare).

Pe baza funcției de performanță agentul e capabil să rezolve așa numitele situații conflictuale în care, la un moment dat, poate opta pentru mai multe stări (succesor), nici una dintre acestea neputând fi sigur realizabile. În astfel de situații, funcția de performanță furnizează o cale prin care succesul poate fi ponderat pe baza importanței stărilor.

Toate programele agent vor avea același schelet: primesc percepții de la un mediu și generează acțiuni (Figura 8.2). Fiecare program agent va folosi structuri de date interne care vor fi reactualizate pe măsură ce apar noi percepții. Cu aceste structuri de date vor lucra procedurile de luare a deciziilor pentru a genera acțiuni posibile.

Partea de *program* a unui agent inteligent ar putea fi schematizată sub forma din Figura 8.2.

funcția AGENT-PROGRAM(*percepție*)

static : *memorie*

memorie ← ACTUALIZEAZĂ-MEMORIE(*memorie*, *percepție*)

acțiune ← ALEGE-ACTIUNEA-OPTIMĂ(*memorie*)

memorie ← ACTUALIZEAZĂ-MEMORIE(*memorie*, *acțiune*)

returnează *acțiune*

sf-funcție

Figura 8.2: Scheletul unui agent inteligent

Un astfel de agent program utilizează anumite structuri interne de date, care sunt actualizate pe măsură ce un nou semnal sosește. Agentul primește la intrare un singur semnal, sub forma unei percepții asupra mediului înconjurător. La fiecare apel, memoria agentului este actualizată pentru a reflecta noul semnal primit. Apoi, pe baza unor proceduri de decizie, se va alege acțiunea optimă, care apoi va fi transmisă arhitecturii agentului pentru a fi executată, apoi se memorează faptul că acțiunea a fost aleasă.

Anumiți agenți program sunt construiți astfel încât secvența perceptuală primită la intrare nu este stocată în memorie, mai ales în situații complexe în care secvențele sunt mari și este inefficientă memorarea lor.

Priviți ca și sisteme (soft) complexe care au abilitatea de a se adapta în medii dinamice și schimbătoare, *agenții* ridică o problemă fundamentală și anume modul în care își folosesc “experiența” anterioară și informațiile senzoriale primite din mediu pentru a decide următoarele mișcări, modul în care să le abordeze și să colaboreze cu alți agenți.

8.3. Tipuri de agenți inteligenți

Există numeroase tipuri de agenți inteligenți, în funcție de informația făcută explicită și folosită în procesul de decizie. Programele agent diferă prin modul în care iau în considerare percepțiile, acțiunile, scopurile și mediul. Câteva clase de agenți inteligenți sunt următoarele:

- agenți reflex (agenți bazați pe reflexe simple);
- agenți care păstrează o imagine a lumii;
- agenți bazați pe scop;
- agenți bazați pe utilitate.

Agenți bazați pe reflexe simple

Un agent bazat pe reflexe simple (agent reflex) răspunde imediat la o secvență de intrare. Un agent reflex realizează o conexiune între situația curentă (percepută) și o acțiune. Aceste conexiuni sunt memorate sub forma unor reguli de producție sau reguli condiție – acțiune.

Programul unui agent reflex este foarte simplu. El caută regula a cărei parte de condiție se potrivește cu situația curentă (cum este ea definită de percepție) și efectuează acțiunea asociată cu această regulă.

Agenți care păstrează o imagine a lumii

Un agent uman alege o anumită acțiune în funcție de condițiile exterioare și de starea sa internă. În anumite situații un agent artificial trebuie să mențină o anumită informație asupra stării interne pentru a distinge între stări ale lumii care generează aceeași secvență perceptuală dar sunt în mod esențial diferite (necesită acțiuni diferite).

Modificarea stării interne necesită ca programul agentului să includă două tipuri de informații(cunoștințe):

1. informații despre felul cum lumea (universul, domeniul de competență al agentului) evoluează independent de agent;
2. informații despre felul în care acțiunile agentului afectează lumea.

Agentul este capabil să interpreteze o nouă secvență perceptuală (o nouă situație) în lumina cunoașterii pe care o are despre lume (codificată în starea internă). El folosește informația despre cum evoluează lumea pentru a păstra o urmă a părții nevăzute a lumii. Agentul trebuie să știe în ce fel acțiunile sale interferează cu starea lumii.

Unui agent reflex i se pot asocia stări interne. În acest caz starea curentă se combină cu starea internă pentru a genera o nouă stare curentă. Acțiunea agentului se bazează pe căutarea unei reguli a cărei condiție se potrivește cu situația curentă așa cum este definită de secvența perceptuală și de starea internă. Apoi se efectuează acțiunea asociată cu această regulă.

Agenți bazați pe scop

Acești agenți acționează în sensul îndeplinirii scopului lor. Cunoașterea stării curente a mediului nu este întotdeauna suficientă pentru a selecta o acțiune corectă. Uneori agentul are nevoie și de o anumită informație asupra scopului, care descrie situațiile ce sunt de dorit. Programul agentului poate combina această informație cu cea despre rezultatele acțiunilor sale pentru a alege acțiunile care permit atingerea țelului propus.

Agenți bazați pe utilitate

Scopul nu este întotdeauna suficient pentru a genera cea mai bună strategie (sau comportare) a unui agent. Scopul realizează o distincție casantă între stările favorabile și cele nefavorabile. Asociind fiecărei stări o măsură a performanței putem compara diferite stări, sau secvențe de stări, din punctul de vedere al satisfaciabilității sau utilității lor pentru agent.

Utilitatea este o funcție U definită pe mulțimea S a stărilor cu valori în mulțimea numerelor reale:

$$U : S \rightarrow R$$

Funcția de utilitate permite luarea unor decizii raționale în cazul scopurilor conflictuale. În cazul în care există mai multe scopuri, dar nici unul nu poate fi atins cu certitudine, funcția de utilitate furnizează o metodă prin care probabilitatea de succes și importanța scopurilor nu pot fi comparate.

Putem admite că utilitatea unei stări măsoară gradul de satisfacție pe care o are agentul dacă atinge starea respectivă. Prin urmare un agent bazat pe utilitate tinde să-și maximizeze propria satisfacție.

8.4. Arhitecturi abstracte pentru agenți inteligenți

În acest paragraf vom prezenta câteva modele generale și proprietăți ale agenților, fără a exista vreo legătură cu modul de implementare al acestora. Se va realiza și o formalizare a prezentării *agenților inteligenți*. Vom presupune că stările ce compun mediul unui agent pot fi descrise sub forma unei mulțimi $S = \{s_1, s_2, \dots\}$, *stările mediului*. La orice moment, mediul se presupune a fi într-una din aceste stări. Capacitatea de acțiune a agentului se presupune a fi reprezentată sub forma unei mulțimi $A = \{a_1, a_2, \dots\}$ de *acțiuni*.

Astfel, din punct de vedere abstract un *agent* poate fi privit ca fiind o funcție

$$\text{agent} : S^* \rightarrow A,$$

funcție care asignează acțiuni unor secvențe de stări ale mediului. Un agent modelat de o astfel de funcție se va numi *agent standard*. Intuitiv, un agent va decide ce acțiuni să efectueze în baza experienței anterioare. Aceste experiențe sunt reprezentate sub forma unor secvențe de stări ale mediului – acele stări pe care agentul le-a întâlnit deja.

Comportamentul (nedeterminist) al mediului poate fi modelat sub forma unei funcții

$$\text{mediu} : S \times A \rightarrow \wp(S),$$

care asignează unei perechi (s, a) o mulțime de stări $\text{mediu}(s, a)$ – ce pot rezulta în urma efectuării acțiunii a în starea s . Dacă toate elementele codomeniului au cardinalul unitar, atunci

mediul se va numi *determinist*, iar comportamentul poate fi stabilit cu exactitate.

Interacțiunea unui agent cu mediul său se poate reprezenta sub forma unui *istoric*

$$h : s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{u-1}} s_u \xrightarrow{a_u} \dots,$$

s_0 fiind starea inițială a mediului, a_u fiind cea de-a u -a acțiune efectuată, iar s_u fiind cea de-a u -a stare a mediului.

Dacă $\text{agent} : S^* \rightarrow A$ este un *agent*, $\text{mediu} : S \times A \rightarrow \wp(S)$ este un *mediu*, iar s_0 este starea inițială a mediului, atunci secvența

$$h : s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{u-1}} s_u \xrightarrow{a_u} \dots$$

reprezintă un posibil *istoric* al agentului în mediu, dacă și numai dacă următoarele două condiții sunt verificate:

1. $\forall u \in \mathbb{N}, a_u = \text{agent}((s_0, s_1, \dots, s_u))$,
2. $\forall u \in \mathbb{N}$ astfel încât $u > 0, s_u \in \text{mediu}(s_{u-1}, a_{u-1})$.

Comportamentul specific al unui agent $\text{agent} : S^* \rightarrow A$ într-un mediu $\text{mediu} : S \times A \rightarrow \wp(S)$ este mulțimea tuturor “istoricelor” care satisfac proprietățile anterioare. Dacă o anumită proprietate Φ este valabilă pentru fiecare istoric al agentului, atunci Φ va fi considerată ca fiind o *proprietate invariantă* a agentului în mediu.

Se va nota $\text{ist}(\text{agent}, \text{mediu})$ mulțimea tuturor “istoricelor” agentului în mediu. În acest sens, doi agenți ag_1 și ag_2 se vor numi *comportamental echivalenți* în raport cu mediul mediu , dacă și numai dacă $\text{ist}(ag_1, \text{mediu}) = \text{ist}(ag_2, \text{mediu})$ și *comportamental echivalenți* dacă și numai dacă sunt comportamental echivalenți în raport cu orice mediu.

În general, sunt de un mai mare interes agenții ale căror interacțiuni cu propriul mediu nu se termină, altfel spus al căror comportament specific este *infini*.

8.4.1. Agenți total reactivi

Anumite categorii de agenți decid ce acțiune să aleagă la un moment dat, fără a ține cont de *istoric*-ul lor, altfel spus luarea unei decizii nu are nici o legătură cu acțiunile trecute ale agentului. Un astfel de agent se va numi *total reactiv*, datorită faptului că poate răspunde *direct* mediului în care acționează.

Formal, comportamentul unui astfel de agent va fi o funcție

$$\text{agent} : S \rightarrow A.$$

Ceea ce înseamnă că pentru fiecare agent total reactiv, există un agent standard echivalent; reciproca nu este, în general, valabilă.

Un exemplu de astfel de agent total reactiv este un termostat (ce dispune de un senzor pentru a detecta temperatura camerei). Mediul unui astfel de agent este unul *fizic*, acțiunile disponibile fiind doar două, astfel încât

$$\text{agent}(s) = \begin{cases} \text{căldura PORNITĂ} \\ \text{căldura OPRITĂ} \end{cases}$$

și agentul poate la orice moment să reacționeze imediat: decizia despre acțiunea pe care o va efectua este *independentă* de acțiunile sale anterioare.

8.4.2. Percepție

După cum se poate observa, modelul *abstract* al unui agent (după cum a fost prezentat anterior) presupune două aspecte: arhitectura (partea de construcție) agentului și proiectarea (design-ul) funcției de decizie a agentului (funcția *agent*).

Este important ca modelul să fie *rafinat*, adică să împartă agentul în subsisteme, în maniera în care se procedează și în ingineria soft. O simplă analiză a celor două subsisteme ce alcătuiesc un *agent*, conduce la următoarele concluzii:

- *proiectarea* (design-ul) agentului (de fapt partea de *program* a agentului) se referă la datele și structurile de control ce vor fi prezente în descrierea agentului;
- *arhitectura* agentului se referă la partea internă – structurile de date, operațiile care vor trebui aplicate acestor structuri, precum și controlul fluxului între structurile de date.

În primul rând, funcția de decizie a agentului trebuie separată în două subsisteme: subsistemul *percepție* și subsistemul *acțiune* (Figura 8.4.2):

- funcția *percepție* se referă la abilitatea agentului de a-și observa (percepe) mediul;
- funcția *acțiune* se referă la procesul de aplicare a deciziilor.

Spre exemplu, pentru un agent soft (programul **xbiff** de sub UNIX – care monitorizează mail-urile primite de utilizator și care indică dacă acesta are sau nu mail-uri necitite), senzorii care percep mediul ar putea fi o serie de comenzi ale sistemului prin care se pot obține informații despre mediul soft (de exemplu comenzile **ls**, **finger**, sau altele).

De observat că mediul acestui agent este unul *soft*, funcțiile prin care obține informații despre mediu sunt funcții *soft*, iar acțiunile pe care le execută sunt tot acțiuni *soft* (schimbarea unei icoane pe ecran sau executarea unui program).

Rezultatul (ieșirea) funcției *percepție* este un *semnal* (percepție) – o secvență perceptuală primită la intrare. Dacă P este o mulțime nevidă de percepții, atunci funcția *percepție* va fi

$$\text{percepție} : S \rightarrow P,$$

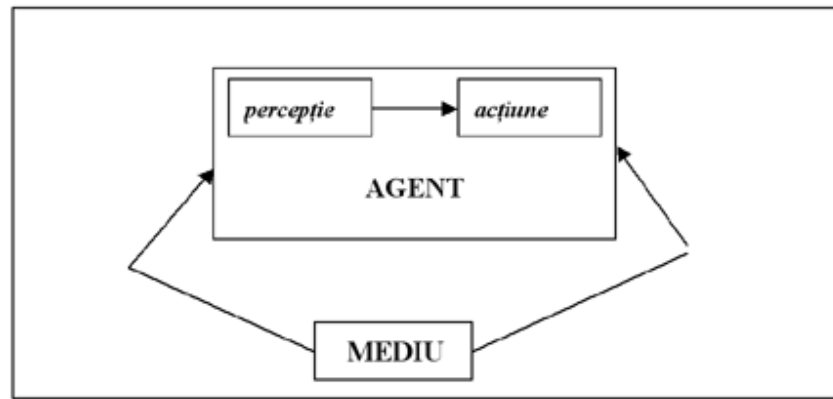


Figura 8.3: Subsistemele *percepție* și *acțiune* ale unui agent

funcție ce asignează stărilor mediului percepții, iar funcția *acțiune* va fi

$$\text{acțiune} : P^* \rightarrow A,$$

funcție ce asignează secvențelor de percepții (semnale) acțiuni.

Definițiile anterioare conduc la următoarea proprietate caracteristică agenților și percepției acestora.

Proprietate: Presupunem că $s_1 \in S$ și $s_2 \in S$ sunt două stări ale mediului unui agent A astfel încât $s_1 \neq s_2$, dar $\text{percepție}(s_1) = \text{percepție}(s_2)$ (agentul va primi aceeași informație perceptuală în două stări distincte). În acest caz, agentul *nu poate deosebi* stările s_1 și s_2 (stările fiind *neindentificabile*).

De exemplu, presupunem că avem un agent care are legătură cu doar două fapte p și q (reprezentate sub formă de propoziții logice) despre mediul în care acționează. Mai mult, faptele p și q sunt independente una de alta, iar agentul percepe efectiv (prin senzorii săi) doar faptul p . În acest caz, putem spune că mulțimea S a stărilor mediului poate fi reprezentată ca având exact 4 (2^2) elemente: $S = \underbrace{\{\neg P, \neg Q\}}_{s_1}, \underbrace{\{\neg P, Q\}}_{s_2}, \underbrace{\{P, \neg Q\}}_{s_3}, \underbrace{\{P, Q\}}_{s_4}$, stările s_1 și s_2 , respectiv s_3 și s_4 , stări în care Q sau $\neg Q$ sunt verificate fiind de fapt *neindentificabile*, deși funcția *percepție* a agentului este aceeași în ambele stări.

8.4.3. Agenți cu stări

În paragrafurile anterioare s-au prezentat modele de agenți a căror funcție de decizie (*acțiune*) asigura acțiuni unor *secvențe* de stări ale mediului sau unor secvențe de percepții.

În acest paragraf, se vor prezenta agenți care *mențin stările* – Figura 8.4.

Funcția de selecție a acțiunii va fi definită astfel:

$$\text{acțiune} : I \rightarrow A,$$

ca o asignare de acțiuni *stărilor interne* ale agentului. Va fi introdusă și o funcție adițională, funcția *următor*, care asignează *stări interne* unor perechi (*stare internă*, *percepție*)

$$\text{următor} : I \times P \rightarrow I.$$

ca o asignare de acțiuni *stărilor interne* ale agentului. Va fi introdusă și o funcție adițională, funcția *următor*, care asignează *stări interne* unor perechi (*stare internă*, *percepție*).

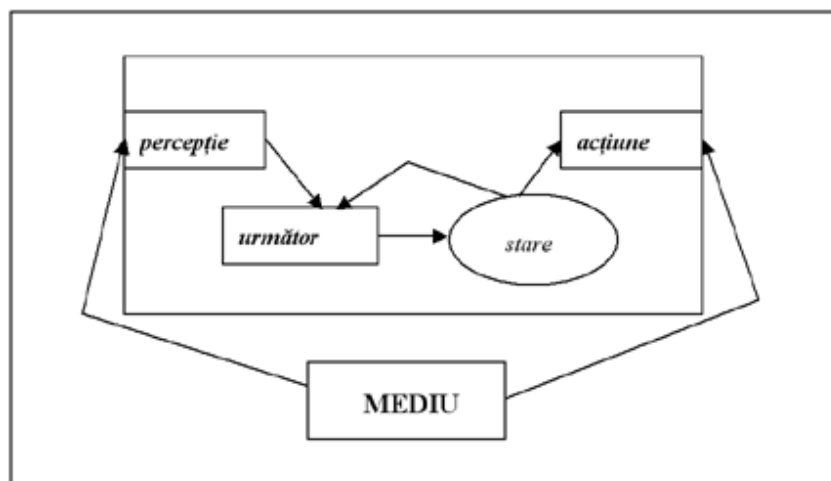


Figura 8.4: Agenți care mențin stările

Comportamentul unui agent bazat pe stări poate fi rezumat astfel:

- agentul pornește cu o stare internă inițială i_0 ;
- observă starea mediului s și generează un semnal (percepție) $\text{percepție}(s)$;
- starea internă a agentului va fi actualizată conform funcției *următor*, devenind $\text{următor}(i_0, \text{percepție}(s))$;
- acțiunea selectată de agent va fi $\text{acțiune}(\text{următor}(i_0, \text{percepție}(s)))$;
- acțiunea va fi executată, după care agentul reia ciclul.

De fapt, se poate observa că *agenții bazați pe stări* sunt *identici* ca putere expresivă cu *agenții standard* – orice agent bazat pe stări poate fi transformat într-un agent standard comportamental echivalent cu acesta.

8.5. Arhitecturi concrete pentru agenți inteligenți

În paragraful anterior, a fost prezentată noțiunea abstractă de *agent inteligent*. Au fost prezentate arhitecturi de agenți care mențin sau nu stările, dar nu s-a discutat despre cum ar trebui sau ar putea să arate aceste stări.

Deasemenea, s-a vorbit despre funcția abstractă *agent*, funcție care modelează procesul de aplicare a deciziilor unui agent, dar nu s-a prezentat modul în care o astfel de funcție ar trebui implementată.

În continuare, se vor prezenta aspecte legate de arhitectura *concretă* a unui agent. Vor fi considerate patru clase de bază:

- *agenți bazați pe logică (logic based agents)* – în care procesul de aplicare a deciziilor este realizat pe baza deducțiilor logice;
- *agenți reactivi (reactive agents)* – în care implementarea procesului de luare a deciziilor se bazează pe asignarea de acțiuni diverselor situații;
- *agenți tip opinie-cerere-intenție (belief-desire-intention)* – în care procesul de luare a deciziilor depinde de manipularea structurilor de date reprezentând opiniile, cererile, respectiv intențiile agentului;
- *arhitecturi stratificate (layered architectures)* – în care procesul de luare a deciziilor este realizat prin intermediul a diverse nivele soft, fiecare nivel ocupându-se de raționamentul despre mediu la diferite nivele de abstractizare.

8.5.1. Arhitecturi bazate pe logică

Modul “tradițional” de realizare a sistemelor artificiale inteligente (cunoscut sub denumirea de *Inteligență Artificială simbolică*) sugerează ca și comportamentul inteligent să fie generat în sistem prin furnizarea unei reprezentări *simbolice* a mediului și a comportamentului și o manipulare simbolică a acestei reprezentări.

Conform abordării tradiționale, reprezentarea simbolică se face sub formă de *formule logice*, iar manipularea simbolică corespunde *deducțiilor logice* sau *demonstrării teoremelor*. În această abordare, agentul poate fi considerat ca o *demonstrație a unei teoreme*. (dacă Φ este o teoremă care explică modul în care un agent inteligent trebuie să se comporte, sistemul va trebui să genereze de fapt o succesiune de pași – acțiuni- prin care să se *ajungă* la Φ - de fapt o “demonstrație” a lui Φ).

Va fi prezentat în continuare un model simplu de agent bazat pe logică, agent ce face parte din categoria așa numitor *agenți prudenți (deliberate agents)*. În astfel de arhitecturi, starea internă a agentului se presupune a fi o bază de date cu formule predicative aparținând logicii predicatelor de ordinul I (de exemplu: *temperatura(X,321)* sau *presiunea(X,28)*). Baza de date reprezintă de fapt *informația* pe care o are agentul despre mediu. De precizat faptul că aceste formule predicative pot sau nu fi valide, atâta timp cât ele reprezintă *percepția* agentului.

Fie L o mulțime de propoziții din logica de ordinul I, și fie $D = \wp(L)$ mulțimea *bazelor de date* din L (o mulțime de mulțimi de formule din L).

- *Starea internă* a agentului va fi dată de un element din D . Vom nota Δ, Δ_1, \dots elementele lui D .
- procesul de *aplicare a deciziilor* agentului se va modela sub forma unui set de *reguli de deducție*, ρ (reguli de inferență logică). Se va nota $\Delta \xrightarrow{\rho} \Phi$ dacă formula Φ poate fi demonstrată din baza de date Δ folosind doar regulile de deducție din ρ .
- funcția de *percepție* rămâne neschimbată $\text{percepție} : S \rightarrow P$.
- funcția *următor* devine următor : $D \times P \rightarrow D$ - asignează unei perechi (bază de date, percepție) o nouă bază de date.
- funcția de selectare a acțiunii

$$\text{acțiune} : D \rightarrow A$$

va fi definită ca o *regulă de deducție*.

Definiția pseudo-cod a acestei funcții este prezentată în Figura 8.5.1.

Ideea este că programatorul agentului va reprezenta regulile de deducție ρ și baza de date Δ în așa manieră încât dacă o formulă $\text{Executa}(a)$ poate fi dedusă, a reprezentând o acțiune, atunci a este cea mai bună acțiune ce ar putea fi aleasă:

- dacă formula $\text{Executa}(a)$ poate fi demonstrată din baza de date, folosind regulile de deducție ρ , atunci a va fi returnată, ca fiind acțiunea ce va trebui executată;
- dacă nu, se caută o acțiune ce este *consistentă* cu regulile și baza de date, adică o acțiune $a \in A$ ce are proprietatea că $\neg \text{Executa}(a)$ nu poate fi dedusă din baza de date folosind regulile de deducție;
- dacă nu reușește nici pasul 2, atunci se returnează *null* (nici o acțiune nu a fost selectată), caz în care comportamentul agentului va fi determinat din regulile de deducție (“programul” agentului) și din baza sa de date curentă (reprezentând informația agentului despre mediul său).

Spre exemplu, problema unui *agent robotic care trebuie să facă curățenie într-o casă*. Robotul este echipat cu un senzor care îi va spune dacă este sau nu praf sau un aspirator în zona în care se află. În plus, robotul are în orice moment o orientare bine definită (*nord, sud, est* sau *vest*). Mai mult, pentru a putea aspira praful, agentul se poate deplasa înainte cu un “pas”, sau să se întoarcă spre dreapta cu 90° . Agentul se va deplasa prin cameră, care este împărțită într-un număr de dreptunghiuri egale (corespunzând unității de deplasare a agentului). Se va presupune că agentul nu face altceva decât să *curețe* – deci nu părăsește camera. Pentru simplificare, caroiul (camera) se presupune că este de 3×3 , iar agentul pornește întotdeauna din poziția (0,0), fiind cu fața spre *nord* (Figura 8.5.1).

Din specificațiile problemei, rezultă că:

funcția acțiune ($\Delta : D$): A
pentru fiecare $a \in A$ **execută**
 dacă $\Delta \xrightarrow{\rho} Executa(a)$ **atunci**
 returnează a
 sf-dacă
sf-pentru
pentru fiecare $a \in A$ **execută**
 dacă $\Delta \not\xrightarrow{\rho} \neg Executa(a)$ **atunci**
 returnează a
 sf-dacă
sf-pentru
returnează $null$
sf-funcție

Figura 8.5: Definiția pseudo-cod a funcției *acțiune*

- agentul poate primi un semnal (percepție) *praf* (cu semnificația că este praf sub el), respectiv *null* (informație nesemnificativă);
- agentul poate executa una din cele trei acțiuni posibile: *înainte*, *aspiră* sau *întoarce*.

Se vor utiliza trei predicate în vederea rezolvării acestei probleme și anume:

- $\hat{In}(x,y)$ - agentul este în poziția (x,y) ;
- $Praf(x,y)$ - este praf în poziția (x,y) ;
- $Orientare(d)$ - agentul e orientat în direcția d .

- vom nota $vechi(\Delta)$ informația “veche” din baza de date, pe care funcția *următor* ar trebui să o șteargă

$$vechi(\Delta) = \{P(t_1, t_2, \dots, t_n) \mid P \in \{In, Praf, Orientare\} \text{ si } P(t_1, t_2, \dots, t_n) \in \Delta\}$$

- e necesară o funcție *nou*, care indică mulțimea predicatelor noi ce trebuie adăugate în baza de date $nou : DxP \rightarrow D$, definită astfel:

$$nou(\Delta, P(t_1, t_2, \dots, t_n)) = \Delta_1 \quad ; \quad \Delta_1 = \Delta \cup \{P(t_1, t_2, \dots, t_n)\},$$

unde predicatul $P \in \{In, Praf, Orientare\}$, iar operația “ \cup ” aplicată bazei de date curente semnifică adăugarea unui nou fapt în baza de date.

Folosind notațiile anterioare, funcția *următor* va putea fi definită astfel:

$$urmator(\Delta, p) = (\Delta \setminus vechi(\Delta)) \cup nou(\Delta, p)$$

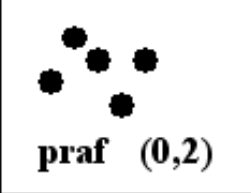
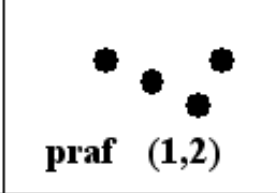
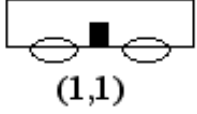
 praf (0,2)	 praf (1,2)	(2,2)
(0,1)	 (1,1)	(2,1)
(0,0)	(1,0)	(2,0)

Figura 8.6: Mediul agentului robotic

Urmează a fi specificate regulile ce definesc comportamentul agentului. Regulile de deducție sunt de forma $\phi(\dots) \rightarrow \psi(\dots)$, unde ϕ și ψ sunt formule care conțin predicate având ca argumente liste arbitrare de constante și variabile. Semnificația unei astfel de reguli este următoarea: dacă ϕ se “potrivește” cu baza de date a agentului, atunci ψ poate fi dedus, cu anumite variabile din ψ instanțiate. Cea mai importantă regulă de deducție se referă la scopul agentului

$$In(x, y) \wedge Praf(x, y) \rightarrow Executa(aspira) \quad (8.1)$$

Celelalte reguli furnizează algoritmul de deplasare a robotului în mediu.

$$In(0, 0) \wedge Orientare(nord) \wedge \neg Praf(0, 0) \rightarrow Executa(inainte) \quad (8.2)$$

$$In(0, 1) \wedge Orientare(nord) \wedge \neg Praf(0, 1) \rightarrow Executa(inainte) \quad (8.3)$$

$$In(0, 2) \wedge Orientare(nord) \wedge \neg Praf(0, 2) \rightarrow Executa(intoarce) \quad (8.4)$$

$$In(0, 2) \wedge Orientare(est) \rightarrow Executa(inainte) \quad (8.5)$$

De observat că regula prioritară este regula 1, iar reguli asemănătoare regulilor 2–5 trebuie descrise explicit pentru fiecare poziție a robotului. Toate aceste reguli, împreună cu funcția *următor* vor genera comportamentul specific al robotului.

Ca și concluzie în abordarea *bazată pe logică*, procesul de aplicare a deciziilor este privit ca o *deducție*, partea de program a agentului (strategia de aplicare a deciziilor) este codificată

ca o teorie logică, iar procesul de selecție a acțiunii se reduce la o problemă de demonstrație. Abordarea logică este elegantă și are o semantică bine formalizată (logică).

Totuși, câteva dezavantaje ale acestei abordări ar fi:

- complexitatea computațională ce intervine în procesul de demonstrare a teoremelor ridică problema dacă agenții reprezentați astfel pot opera efectiv în medii supuse limitărilor de timp;
- procesul de aplicare a deciziilor în astfel de arhitecturi logice se bazează pe presupunerea că mediul nu își schimbă semnificativ structura în timpul procesului de decizie (o decizie care este corectă în momentul începerii procesului de decizie, va fi corectă și la sfârșitul acestuia);
- problema reprezentării și raționamentului în medii complexe și dinamice este deasemenea o problemă deschisă.

8.5.2. Arhitecturi reactive

Datorită problemelor ce apar în abordarea logico-simbolică, spre sfârșitul anilor '80, cercetătorii au început să investigheze alternative la paradigma logico-simbolică a Inteligenței Artificiale.

Aceste abordări noi, diferite de abordarea logico-simbolică, sunt prezentate ca fiind:

- *comportamentale* – un scop comun este dezvoltarea și combinarea comportamentelor individuale;
- *integrate* – o temă comună este cea a agenților situați (integrați) în anumite medii, nu a celor neintegrați în acestea;
- *reactive* – sistemele sunt deseori percepute ca reacționând pur și simplu la mediul lor, fără raționament asupra acestuia.

Cea mai bună arhitectură reactivă cunoscută până în prezent se consideră a fi *arhitectura "subsumption"* (AS), dezvoltată de Rodney Brooks.

Cele două caracteristici definitorii ale acestei arhitecturi sunt:

- procesul de aplicare a deciziilor agentului se realizează prin intermediul unei mulțimi de *comportamente ce îndeplinesc anumite sarcini*; fiecare comportament poate fi privit ca o funcție individuală de *acțiune* (definită anterior), funcție care asociază unor percepții (recepționate la intrarea în sistem) acțiuni ce urmează a fi executate; fiecare modul "comportamental" realizează anumite sarcini, și se presupune că modulele nu includ *reprezentări simbolice* și nici *raționament simbolic*. În multe implementări, aceste comportamente sunt implementate ca reguli de forma *situație* \rightarrow *acțiune*, reguli ce asignează direct acțiuni unor secvențe perceptuale;

- modulele sunt aranjate într-o *ierarhie* (*subsumption hierarchy*), comportamentele fiind dispuse pe *nivele*. Nivelele situate mai jos în ierarhie au o *prioritate* mai mare, cele situate mai sus reprezentând comportamente mai abstracte.

Funcția *percepție*, reprezentând abilitatea perceptuală a agentului, se presupune că rămâne neschimbată.

Funcția de decizie *acțiune* este realizată prin intermediul unei mulțimi de comportamente și a unei relații de *constrângere* existentă între nivele.

Un comportament va fi reprezentat sub forma unei perechi (c, a) , unde $c \subseteq P$ este o mulțime de percepții numită *condiție*, iar $a \in A$ este o acțiune.

Un comportament (c, a) se va spune că *este posibil* într-o stare $s \in S$ a mediului dacă și numai dacă $percepție(s) \in c$.

În plus, se va nota $Comportament = \{(c, a) | c \subseteq P \text{ și } a \in A\}$ mulțimea tuturor regulilor de comportament.

Unei mulțimi de reguli de comportament $R \subseteq Comportament$ i se va asocia o relație binară de *constrângere*, “ \prec ”, $\prec \subseteq R \times R$. Relația se presupune a fi o relație de ordine totală pe R (tranzitivă, nereflexivă și antisimetrică). Se va nota $b_1 \prec b_2$ dacă $(b_1, b_2) \in \prec$ și se va spune “ b_1 constrânge pe b_2 ” (b_1 se află în ierarhie mai jos decât b_2 , având deci prioritate față de b_2).

Definiția pseudo-cod a funcției *acțiune* este descrisă în Figura 8.5.2.

funcția *acțiune* ($p : P$): A

var *posibil* : $\wp(R)$

$posibil := \{(c, a) | (c, a) \in R \text{ și } p \in c\}$

pentru fiecare $(c, a) \in posibil$ **execută**

dacă $\neg(\exists(c', a') \in posibil \text{ a.î. } (c', a') \prec (c, a))$ **atunci**

returnează a

sf-dacă

sf-pentru

returnează *null*

sf-funcție

Figura 8.7: Definiția pseudo-cod a funcției *acțiune*

Una din problemele ce le ridică arhitectura bazată pe logică este complexitatea computațională, funcția *acțiune* a arhitecturii reactive având în cel mai rău caz o complexitate $O(n^2)$ (cum se poate deduce din algoritmul descris în Figura 8.5.2), n fiind maximul dintre numărul de comportamente și numărul de percepții.

Cu toate că arhitecturile reactive în general sunt relativ simple, elegante și cu o complexitate computațională mai redusă, există o serie de probleme nerezolvate pentru această categorie de agenți:

- dacă agenții nu utilizează modele ale mediului, atunci trebuie să aibă un număr de informații disponibile în mediul *local*, suficiente pentru a determina o soluție acceptabilă;

- e destul de dificil de văzut cum pot fi proiectate sistemele reactive pure astfel încât să *învețe* din experiență și să-și îmbunătățească performanțele în timp;
- caracteristica de bază a sistemelor pur reactive este interacțiunea dintre componentele comportamentale ale agentului, or în anumite situații astfel de relații sunt dificile sau poate imposibil de stabilit;
- în cazul în care numărul de nivele în arhitectura agentului crește, dinamica interacțiunilor între nivele devine extrem de complexă și greu de înțeles.

8.5.3. Arhitecturi tip Opinie-Cerere-Intenție

O soluție la problema de conceptualizare a sistemelor care sunt capabile de comportament rațional implică privirea agenților sub forma unor *sisteme intenționale*, al căror comportament poate fi prevăzut și explicat în termeni de *atitudini* ca opinii, cereri, intenții (modelul BDI Rao-Georgeff). În această abordare, agenții pot fi priviți sub forma unor sisteme raționale, capabile de comportament *orientat spre scop*.

Aceste arhitecturi Opinie-Cerere-Intenție (OCI) – *Belief-Desire-Intention architectures* – își au rădăcinile în tradițiile filozofice despre înțelegerea *raționamentului practic* – procesul de a decide, în fiecare moment, ce acțiune să fie executată în vederea realizării scopurilor propuse.

Raționamentul practic implică două procese importante: a decide *care* scopuri se doresc a fi realizate și *cum* se vor realiza aceste scopuri. Primul proces este cunoscut sub numele de *deliberare*, iar al doilea de *raționament tip medii-capete (means-ends)*.

Procesul de decizie începe în general cu stabilirea *opțiunilor* disponibile. După generarea setului de alternative, urmează *alegerea* unor alternative din cele existente. Alternativele alese devin *intenții*, care de fapt vor determina acțiunile agentului. Intențiile apoi vor genera un feed-back în procesul de raționament al agentului.

În astfel de procese, un rol important îl joacă măsura în care intențiile agentului sunt sau nu *reconsiderate* din când în când. Dar reconsiderarea are un cost – atât ca timp cât și ca resurse computaționale. Din acest motiv, există următoarea dilemă:

- un agent care nu se oprește în a-și reconsidera des intențiile, riscă să încerce să-și realizeze intențiile chiar și după ce e evident că acestea nu pot fi realizate, sau nu mai există nici un interes în a le realiza;
- un agent care își reconsideră în mod *constant* intențiile, ar putea folosi timp insuficient în vederea realizării efective a acestora, riscând astfel să nu le realizeze niciodată.

Această dilemă este de fapt problema esențială de a alege între comportamentele (arhitecturile) *pro-active* (orientate spre scop) și cele *reactive* (conduse de întâmplări, posibilități).

O soluție la această dilemă pare a fi următoarea: în mediile statice, ce nu se schimbă, comportamentele pro-active, orientate spre scop sunt adecvate, pe când în mediile dinamice, abilitatea de a reacționa la schimbări prin modificarea intențiilor devine mai importantă.

Într-un model OCI, starea unui agent OCI poate fi descrisă prin:

- o mulțime de *opinii* despre lume;
- o mulțime de *scopuri* (*dorințe*) pe care agentul va încerca să le realizeze;
- o colecție de *planuri* ce descriu modul în care agentul își va realiza scopurile și cum va reacționa schimbărilor în opiniile pe care le-a avut;
- o structură de *intenții*, descriind cum agentul își realizează la un moment dat scopurile și cum reacționează curent la schimbările de opinii.

Procesul de raționament practic al unui agent OCI este ilustrat în Figura 8.5.3. După cum se ilustrează în această figură, există șapte componente de bază ale unui agent OCI:

- o mulțime de *opinii* curente, reprezentând informația pe care o are agentul despre mediul său curent;
- o *funcție de revizuire a opiniilor* - belief revision function (*fro*), care pe baza unei percepții și a opiniilor curente ale agentului, determină o nouă mulțime de opinii;
- o *funcție de generare a opțiunii* (*opțiuni*), care determină opțiunile disponibile ale agentului (dorințele acestuia), pe baza părerilor curente despre mediu și a *intențiilor* curente;
- o mulțime de *opțiuni* curente, reprezentând posibilele acțiuni disponibile;
- o *funcție de filtrare* (*filtrare*), reprezentând procesul de *deliberare* al agentului - determinarea intențiilor agentului pe baza opiniilor, cererilor și intențiilor curente ale acestuia;
- o mulțime de *intenții* curente – reprezentând focarul curent al agentului – acele stări pe care agentul s-a angajat să le atingă;
- o *funcție de selecție a acțiunii* (*execută*), care determină ce acțiune trebuie executată pe baza intențiilor curente.

În continuare, vom formaliza o arhitectură OCI.

Se va nota prin *Opi* – mulțimea posibilelor opinii, *Cer* – mulțimea cererilor posibile și *Int* – mulțimea intențiilor posibilele ale agentului. Deseori, elementele celor trei mulțimi sunt formule logice. În plus, trebuie definită și noțiunea de *consistență* a celor trei mulțimi, adică răspunsul la o întrebare de genul: în ce măsură intenția de a realiza x este consistentă cu cererea y . În cazul reprezentării sub formă de formule logice, problema consistenței se reduce de fapt la verificarea consistenței formulelor logice.

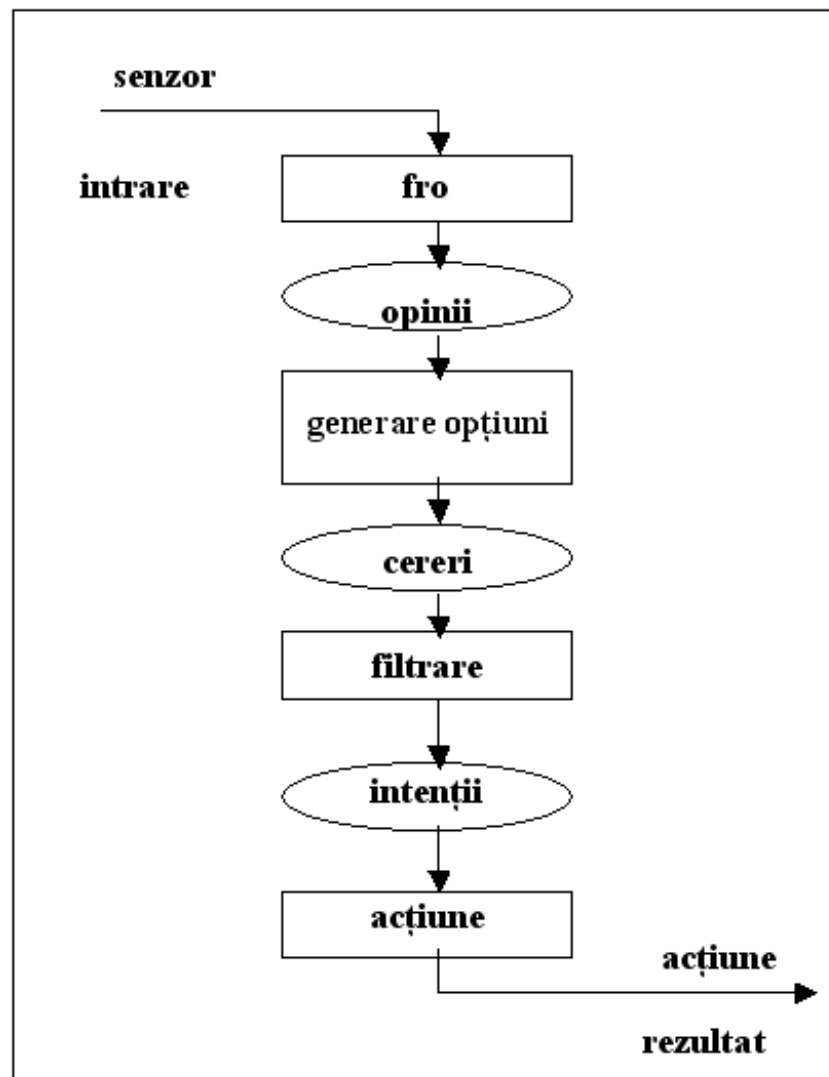


Figura 8.8: Forma generică a unei arhitecturi OCI

Starea unui agent OCI la un moment dat va fi reprezentată sub forma unui triplet (O, C, I) , unde $O \subseteq Opi$, $C \subseteq Cer$ și $I \subseteq Int$, iar funcția de revizuire a părerilor va fi $frp : \wp(Opi) \times P \rightarrow \wp(Opi)$, P fiind mulțimea percepțiilor.

Funcția de generare a opțiunilor se definește astfel: $optiuni : \wp(Opi) \times \wp(Int) \rightarrow \wp(Cer)$.

Funcția *opțiuni* are o serie de roluri:

- este responsabilă de raționamentul de tip *medii-capete* al agentului – procesul de a decide *cum* se vor realiza intențiile;
- odată ce agentul și-a format o intenție x , ulterior trebuie considerate toate opțiunile de a realiza x ; aceste opțiuni vor fi mai concrete decât x ; procesul de generare a opțiunilor unui agent OCI poate fi interpretat ca o elaborare recursivă a unui plan ierarhic, considerând în mod progresiv intenții din ce în ce mai specifice, până se va ajunge în final la acțiuni imediat executabile.

Procesul de deliberare al agentului OCI (a decide *ce* trebuie să facă) va fi reprezentat de funcția de filtrare, $filtrare : \wp(Op_i) \times \wp(Cer) \times \wp(Int) \rightarrow \wp(Int)$, funcție care actualizează intențiile agentului pe baza intențiilor avute anterior, a opiniilor și cererilor actuale.

Această funcție trebuie să aibă câteva roluri:

- trebuie să *omită* orice intenție care nu mai este realizabilă sau pentru a cărei realizare costul este mult mai mare decât cel preconizat;
- trebuie să *rețină* intențiile care nu sunt realizate, dar care sunt considerate totuși a fi avantajoase;
- trebuie să *adopte* noi intenții, fie pentru realizarea intențiilor existente, fie pentru a exploata noi oportunități.

De observat că funcția *filtrare* trebuie să satisfacă următoarea limitare:

$$\forall O \in \wp(Op_i), \forall C \in \wp(Cer), \forall I \in \wp(Int), \quad filtrare(O, C, I) \subseteq I \cup C,$$

adică intențiile curente sunt fie intenții avute anterior, fie opțiuni nou adoptate.

Funcția *execută* se presupune că returnează orice intenție executabilă – care corespunde unei acțiuni direct executabile.

$$executa : \wp(Int) \rightarrow A$$

Ca urmare, funcția *acțiune* a unui agent OCI, $actiune : P \rightarrow A$, va fi definită cu algoritmul pseudo-cod prezentat în Figura 8.5.3.

De remarcat că în reprezentarea intențiilor agentului s-ar putea ține cont de următoarele observații:

- să se asocieze fiecărei intenții o anumită *prioritate*, indicând importanța acesteia;
- reprezentarea intențiilor să se facă sub forma unei *stive*; o intenție să fie adăugată în stivă dacă este adoptată, sau scoasă din stivă dacă a fost realizată sau dacă e nerealizabilă.

În concluzie, arhitecturile OCI sunt arhitecturi de raționament practic, în care procesul de a decide ce trebuie făcut seamănă de fapt cu raționamentul practic pe care îl folosim în viața cotidiană.

Componentele de bază ale unei arhitecturi OCI sunt structurile de date reprezentând opinii, cereri și intenții ale agentului și funcțiile care reprezintă deliberarea și raționamentul.

Modelul OCI este atractiv deoarece:

- este *intuitiv*;

funcția *acțiune* ($p : P$): A

$$O := \text{fro}(O, p)$$

$$C := \text{optiuni}(C, I)$$

$$I := \text{filtrare}(O, C, I)$$

returnează *execută*(I)

sf-funcție

Figura 8.9: Definiția pseudo-cod a funcției *acțiune*

- furnizează o descompunere *funcțională* foarte clară.

În schimb, marea dificultate este maniera în care ar trebui implementate eficient funcțiile necesare.

Pentru cei mai mulți cercetători de Inteligență Artificială, ideea programării sistemelor informatice în termeni de noțiuni *mentale* ca opinii, cereri, intenții reprezintă componenta cheie a "calculului" bazat pe agenți. Conceptul a fost articulat de Yoav-Shoham, în propunerea de *programare orientată pe agenți*.

8.5.4. Arhitecturi stratificate

Datorită cerințelor ca un agent să fie capabil să aibă atât un comportament reactiv, cât și unul pro-activ, o descompunere evidentă ar necesita crearea de subsisteme separate care să trateze aceste două tipuri diferite de comportament.

Ideea conduce spre o clasă de arhitecturi în care diversele subsisteme sunt aranjate într-o ierarhie de *straturi* (*nivele*) ce interacționează între ele.

În general, într-o arhitectură stratificată ar trebui să existe minim două straturi, corespunzătoare celor două tipuri de comportament. Oricum, de o mare importanță este fluxul de control existent între nivele.

În principiu, există două tipuri de flux de control între nivelele unei arhitecturi stratificate (Figura 8.5.4):

- *Stratificare orizontală* (Figura 8.5.4.a) – nivelele soft sunt conectate direct la intrarea perceptuală (senzorul de intrare) și de acțiunea rezultat. De fapt, fiecare nivel acționează ca un agent, furnizând sugestii despre ce acțiune trebuie executată;
- *Stratificare verticală* (Figura 8.5.4.b și 8.5.4.c) - senzorul de intrare și acțiunea rezultat sunt fiecare tratate de cel mult un nivel.

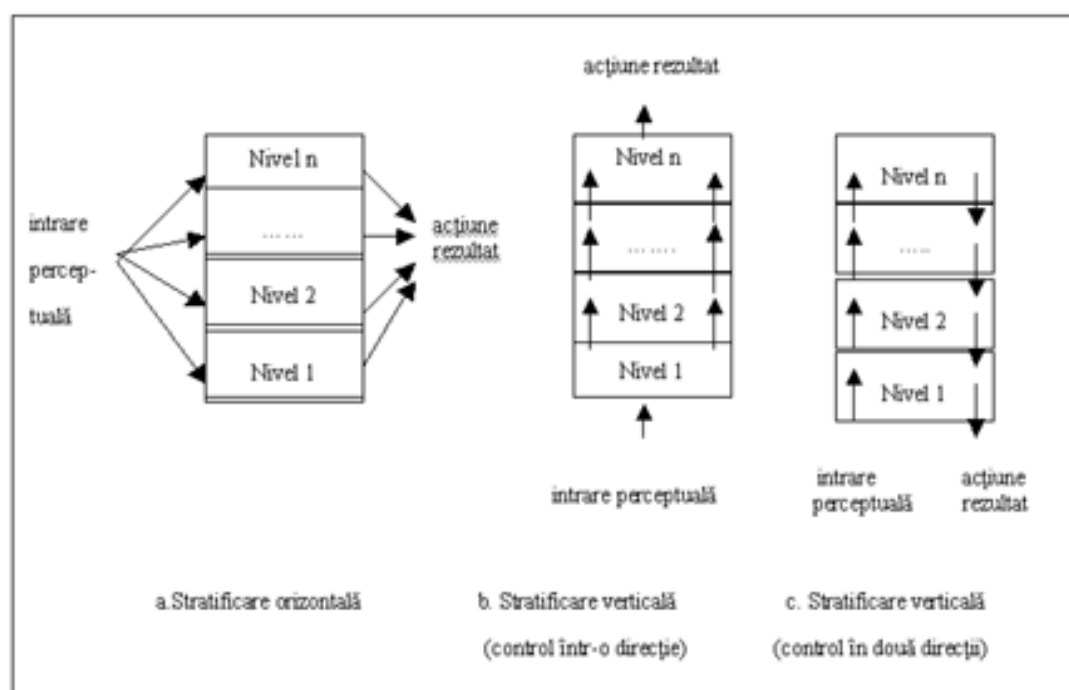


Figura 8.10: Arhitectură stratificată

8.6. Limbaje de programare pentru agenți

Pe măsură ce tehnologia agenților devine tot mai stabilă, ne putem aștepta să vedem cum o varietate de tehnici și tehnologii soft devin disponibile pentru proiectarea și construirea sistemelor bazate pe agenți. În acest capitol vom aduce în discuție două dintre cele mai cunoscute limbaje de programare pentru agenți.

8.6.1. Programare orientată pe agenți

Yoav Shoham propune “o nouă paradigmă de programare, bazată pe un alt punct de vedere asupra procesului de calcul”, pe care o numește *programare orientată pe agenți* – POA – (*agent-oriented programming*).

Idea de bază a POA este programarea directă a agenților, în termeni de *noțiuni mentale* (opinie, cerere, intenție), termeni pe care teoreticienii în domeniu le-au dezvoltat pentru reprezentarea proprietăților agenților.

POA poate fi privită ca un fel de programare “post-declarativă”. În programare declarativă (Prolog, spre exemplu), scopul este de a reduce importanța aspectelor de control: se stabilește un scop (*goal*) pe care sistemul își propune să-l realizeze, după care se lasă mecanismul intern să găsească ce trebuie făcut pentru realizarea scopului. Cu alte cuvinte, spre deosebire de programarea procedurală, ne focalizăm spre *ce* trebuie făcut, nu *cum* trebuie făcut.

Pe de altă parte, limbajele de programare declarativă sunt generate din principii matematice,

astfel încât activități ca: analiza, proiectarea, specificarea, implementarea, devin din ce în ce mai formale. Aceste limbaje oferă o metodă de analiză a corectitudinii programelor.

În POA, ca și în programarea declarativă, ideea este că ne stabilim scopurile și lăsăm în seama mecanismului de control al limbajului să realizeze scopul. Este, deci, destul de clar faptul că, pentru specificarea unor astfel de sisteme va trebui să folosim aserțiuni logice care pot fi formalizate printr-o logică a “cunoașterii”.

Prima implementare a paradigmei POA a fost limbajul de programare AGENT0. Într-un astfel de limbaj, un agent este specificat sub forma unei mulțimi de capacități (lucruri pe care poate să le facă), o mulțime de *opinii* inițiale (jucând rolul opiniilor dintr-o arhitectură OCI), o mulțime de *angajamente* (similare intențiilor dintr-o arhitectură OCI) și o mulțime de *reguli de angajament*. Regulile de angajament joacă rolul cel mai important, determinând modul în care agentul acționează.

Fiecare astfel de regulă conține o *condiție de mesaj*, o *condiție mentală* și o *acțiune*. Pentru a determina care regulă va fi aplicată la un moment dat, condiția de mesaj este potrivită cu mesajele pe care agentul le primește; condiția mentală este potrivită cu intenția agentului). Dacă regula se potrivește, agentul se *angajează* să execute acțiunea.

Acțiunile pot fi *private*, corespunzătoare unor proceduri interne sau *comunicative*, sub forma unor mesaje ce vor fi transmise. Mesajele pot fi de trei tipuri: “*tip cerere*”, “*tip refuz*”, pentru a executa sau a respinge acțiuni și mesaje “*informative*”, care transmit informații.

Un exemplu de *regulă de angajament* în AGENT0 ar putea fi următorul:

COMMIT

```
(
(agent, REQUEST, DO(time, action)
), ;; condiția de mesaj
(B,
[now, Friend agent] AND
CAN(self, action) AND
NOT[time, CMT(self, anyaction)]
), ;;condiția mentală
self,
DO(time, action)
)
```

Această regulă ar putea fi parafrazată în felul următor:

« **Dacă** recepționez un mesaj de la agent prin care îmi cere să execut o acțiune la un moment dat și eu cred că:

- agentul îmi este prieten;
- pot să execut acțiunea;

- la momentul respectiv, nu sunt angajat în executarea altei acțiuni

atunci mă angajez să execut acțiunea la momentul dat \gg .

Modul de operare al unui agent poate fi descris prin următoarea buclă (Figura 8.6.1):

- (1) Citește toate mesajele curente, actualizându-ți opiniile – și în consecință angajamentele – când e necesar;
- (2) Execută toate angajamentele pentru ciclul curent, acolo unde condiția de capabilitate a acțiunii asociate este satisfăcută;
- (3) Salt la (1).

De remarcat că acest limbaj este de fapt un *prototip*, nefiind conceput pentru realizarea de sisteme de producție pe scară largă.

8.6.2. “METATEM” Concurrent

Limbajul METATEM Concurrent a fost dezvoltat de Fisher și se bazează direct pe execuția formulelor logice. Un sistem în acest limbaj conține un număr de agenți ce se execută concurrent, fiecare agent fiind capabil să comunice cu ceilalți printr-o transmisie asincronă de mesaje.

Fiecare agent este programat dându-i-se o specificație *logico-temporală* a comportamentului care se dorește ca agentul să-l aibă.

Execuția programului agent corespunde construirii iterative a unui model logic pentru specificația temporală a agentului. E posibil să se demonstreze că procedura utilizată pentru execuția specificației unui agent este corectă arătând că: dacă e posibil să se satisfacă specificațiile, atunci agentul va realiza acest lucru.

Semantica logică a limbajului METATEM Concurrent este strâns legată de semantica logicii temporale, ceea ce înseamnă, printre altele, că specificarea și verificarea sistemelor METATEM Concurrent sunt propoziții logice.

Un agent program în METATEM Concurrent are forma $\bigwedge_i P_i \Rightarrow F_i$, P_i fiind o formulă logică temporală cu referire la prezent sau trecut, iar F_i fiind o formulă logică temporală cu referire la prezent sau viitor. Formulele $P_i \Rightarrow F_i$ se numesc *reguli*. Ideea de bază a execuției unui astfel de program se exprimă astfel:

în baza trecutului execută viitorul

În consecință, pentru fiecare regulă în parte se încearcă potriviri cu o serie de structuri *interne (istoric)*, iar dacă o potrivire este găsită, atunci regula se poate accepta. Dacă o regulă este acceptată, atunci anumite variabile din partea “viitor” sunt instanțiate, iar viitorul devine un *angajament* pe care, în consecință, agentul va încerca să-l îndeplinească. Îndeplinirea unui angajament înseamnă validarea unor predicate de către agent.

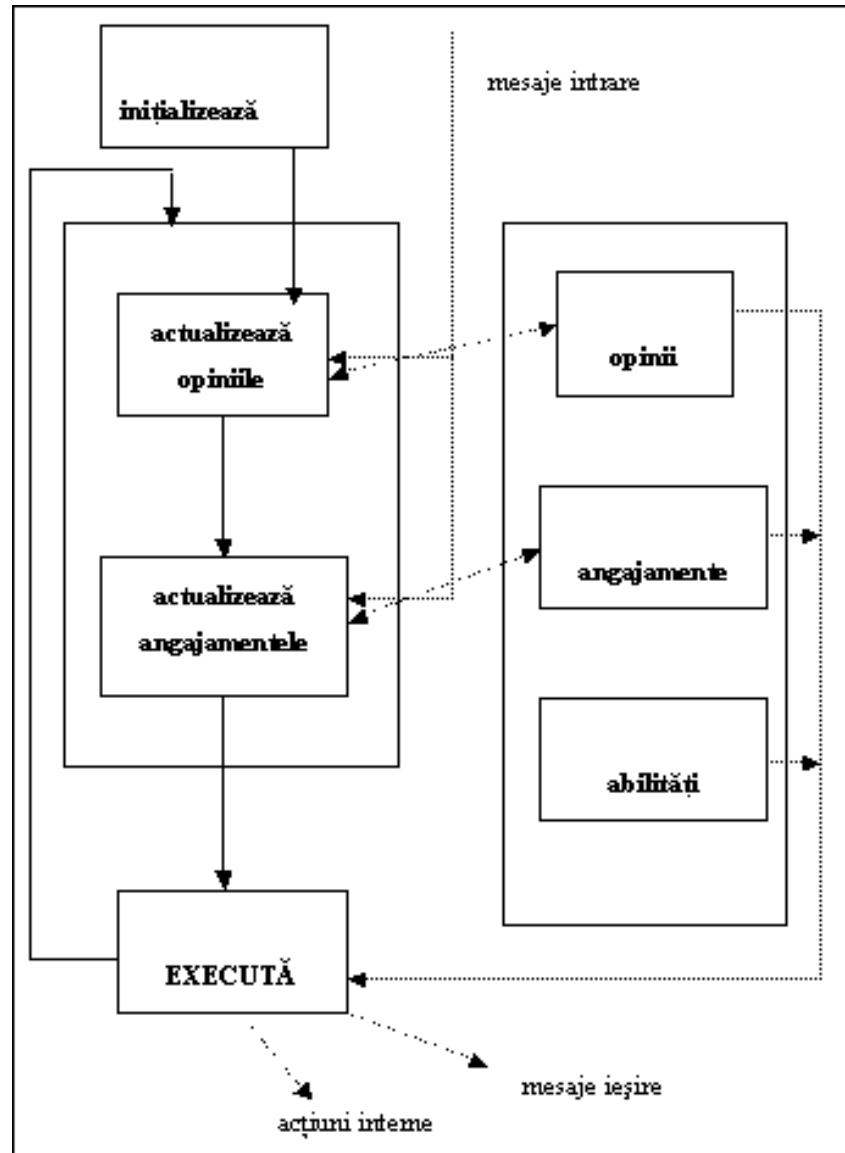


Figura 8.11: Fluxul de control în AGENT0

O definiție simplă a unui agent în limbajul METATEM Concurrent ar putea fi următoarea:

$cr(ask)[give]$:

$$\Theta ask(x) \Rightarrow \Diamond give(x)$$

$$(\neg ask(x) \Xi (give(x) \wedge \neg ask(x))) \Rightarrow \neg give(x)$$

$$give(x) \wedge give(y) \Rightarrow (x = y)$$

Agentul din exemplu este un controlor pentru o resursă care se înnoiește la infinit, dar care poate fi obținută de un singur agent la un moment dat. În exemplul anterior, operatorii Θ , \Diamond și Ξ sunt operatori ai logicii modale, a căror semnificație rezultă din interpretarea regulilor 1, 2 și 3. Agentul va trebui deci să forțeze excluziunea mutuală asupra resursei.

- prima linie a programului definește interfața agentului: numele acestuia este cr și acceptă două mesaje ask și $give$;

- următoarele trei linii reprezintă corpul programului
 - predicatul $ask(x)$ înseamnă că agentul x cere resursa;
 - predicatul $give(x)$ înseamnă că agentului x i s-a dat resursa;

Controlorul de resurse se presupune că este singurul agent capabil să “dea” resursa. Pe de altă parte, mai mulți agenți pot lansa simultan cereri pentru resursă. Cele trei reguli care definesc comportamentul agentului pot fi descrise astfel:

Regula 1. Dacă cineva a cerut resursa, atunci eventual i se dă resursa;

Regula 2. Nu se dă resursa până când cineva a cerut de când s-a dat ultima dată;

Regula 3. Nu se dă la mai multe persoane la un moment dat (dacă doi indivizi cer în același timp, atunci ei reprezintă aceeași persoană).

METATEM Concurrent este o bună ilustrare a modului în care o abordare bazată pe logică în programarea agenților funcționează, cu o logică destul de expresivă.

8.7. Agenți inteligenți și Obiecte

Obiectele, după cum se știe, sunt definite ca fiind niște *entități* computaționale care *încapsulează* o serie de stări, sunt capabile să efectueze acțiuni, sau *metode* asupra acestor stări și să comunice prin transmiterea de mesaje.

În timp ce între *agenți inteligenți* și *obiecte* există similarități evidente, sunt și o serie de diferențe semnificative.

1. *Gradul de autonomie* al obiectelor și al agenților. După cum se știe, principala caracteristică a programării obiectuale este încapsularea – ideea că obiectele au control asupra stării lor interne. Datorită posibilității de a avea atribute (atât variabile de instanță cât și metode) **private** și **publice**, un obiect poate fi considerat ca exprimându-și autonomia asupra propriei stări: are control asupra acesteia. Dar un obiect nu-și poate exprima controlul asupra *comportamentului* său (spre exemplu, dacă o metodă **m** este disponibilă celorlalte obiecte, ea poate fi apelată de acestea ori de câte ori vor – odată ce un obiect își declară o metodă **publică**, nu mai poate avea controlul dacă acea metodă a fost sau nu executată).

Nu la fel este situația într-un sistem *multiagent*. Nu se poate admite faptul că agentul i va executa o acțiune (metodă) a doar pentru faptul că un alt agent j vrea acest lucru – a poate că nu este varianta optimă pentru i .

Astfel, dacă j cere agentului i să efectueze acțiunea a , i poate satisface sau nu cererea primită. Controlul deciziei despre executarea sau nu a unei acțiuni, este diferită în sistemele “cu obiecte” sau “cu agenți”. În cazul agenților, decizia este a agentului care

recepționează cererea, pe când în cazul obiectelor, decizia este a obiectului care lansează cererea. Această diferență între obiecte și agenți este sugestiv redată de [18]:

“Obiectele execută un lucru în mod liber, pe când agenții execută un lucru pentru bani.”

2. *Gradul de flexibilitate* al comportamentului. Modelul obiectual standard, nu prea are multe de spus despre cum să se construiască sisteme care să integreze un astfel de tip de comportament. Gradul de flexibilitate se referă, în esență, la trei lucruri:

- *reactivitate*: agenții inteligenți au abilitatea de a-și percepe mediul și să răspundă oportun la schimbările ce apar, în scopul atingerii obiectivelor pentru care au fost proiectați – așa numitul “*comportament orientat spre scop*”;
- *sunt activi*: agenții inteligenți sunt capabili să dea dovadă de comportament specific, luând inițiative în vederea atingerii obiectivelor;
- *abilitate socială*: agenții inteligenți sunt capabili să interacționeze cu alți agenți (posibili umani – poate fi cazul Sistemelor Expert), în vederea atingerii obiectivelor.

3. Se consideră că fiecare agent are propriul său *fir* (thread) de control – într-un model obiectual standard există un singur fir de control în sistem.

Recent a apărut noțiunea de *concurență* în programarea obiectuală, cu așa numitele *obiecte active*, care în esență sunt agenți, dar care nu dau dovadă, în mod necesar, de comportament autonom *flexibil*.

“Un obiect activ este unul care încorporează propriul său fir de control. Obiectele active sunt în general autonome, ceea ce înseamnă că pot da dovadă de un anumit comportament fără a fi acționate de un alt obiect. Obiectele pasive, în schimb, pot explicit să-și schimbe starea doar la o acțiune a unui alt obiect” (G. Booch, 1984).

În concluzie, există trei deosebiri fundamentale între punctul de vedere tradițional, asupra unui obiect, și punctul de vedere asupra unui agent:

- agenții încorporează mai puternic decât obiectele noțiunea de *autonomie*, iar în particular, pot decide pentru ei înșiși dacă să execute sau nu o acțiune solicitată de un alt agent;
- agenții sunt capabili de comportament *flexibil* (în cei trei termeni în care a fost definită flexibilitatea anterior), ceea ce lipsește modelului obiectual;
- un sistem multiagent e în mod inerent *multi-“thread”*, în care se consideră că fiecare agent are cel puțin un fir de control.

Bibliografie

- [1] AVRON BARR, EDWARD A. FEIGENBAUM, PAUL KOHEN, *The Handbook of Artificial Intelligence*, 3 vols., Morgan Kaufmann, Palo Alto, 1981
- [2] R.A. BROOKS, *A robot layered control system for a mobile robot*, IEEE Journal of Robotics and Automation, 2 (1), 1986, pp. 14–23
- [3] D.C. DENNETT, *The Intentional Stance*, The MIT Press: Cambridge. MA, 1987
- [4] RICHARD O. DUDA, PETER E. HART, *Pattern Recognition and Scene Analysis*, John Willey and Sons, New York, 1973
- [5] M. FISHER, *Concurrent MetateM, A language for modeling reactive systems*, Proceedings of Parallel Architectures and languages Europe (PARLE), Springer Verlag, 1993
- [6] IOAN GEORGESCU, *Elemente de Inteligență Artificială*, Editura Academiei RSR, București, 1985
- [7] EARL B. HUNT, *Artificial Intelligence*, Academic Press, New York, 1975
- [8] MIHAELA MALIȚA, MIRCEA MALIȚA, *Bazele Inteligenței Artificiale, vol. 1, Logici Propoziționale*, Editura Tehnică, București, 1987
- [9] D. MORLEY, *Semantics of Actions, Agents and Environments*, Doctoral Thesis, University of Melbourne, 1999
- [10] NILS J. NILSSON, *Principles of Artificial Intelligence*, Morgan Kaufmann, Palo Alto, 1980
- [11] A.S. RAO, M. GEORGEFF, *BDI Agents: from theory to practice*, Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, CA, 1995, pp. 312–319
- [12] ELAINE RICH, *Artificial Intelligence*, McGraw Hill, New York, 1983
- [13] ELAINE RICH, KEVIN KNIGHT, *Artificial Intelligence*, McGraw Hill, New York, 1991

-
- [14] J.S. RUSSELL, P. NORVIG, *Artificial Intelligence – A Modern Approach*, Prentice-Hall, Inc., New Jersey, 1995
 - [15] Y. SHOHAM, *Agent-oriented programming*, *Artificial Intelligence*, 60 (1), 1993, pp. 51–92
 - [16] PATRICK HENRY WINSTON, *Artificial Intelligence*, Addison Wesley, Reading, MA, 1984, 2nd ed.
 - [17] PATRICK HENRY WINSTON, *Lisp*, Addison Wesley, Reading, MA, 1984, 2nd ed.
 - [18] GERHARD WEISS, *Multiagent systems – A Modern Approach to Distributed Artificial Intelligence*, The MIT Press, Cambridge, Massachusetts, London, 1999