ALGORITMI DE CĂUTARE (AO*) ÎN SISTEMELE DE ÎNVĂŢARE*

ing. Silviu Călinoiu

Universitatea "Politehnica" Bucureşti

Centrul de Cercetări Avansate în Învăţarea Automată, Prelucrarea Limbajului Natural şi Modelarea Conceptuală al Academiei Române

Rezumat: Articolul conţine o prezentare detaliată a algoritmilor de căutare de tip AO*. Aceşti algoritmi au fost bine studiaţi în literatură [1,6,7,8]. În lucrarea de faţă, ei sunt însă prezentaţi din perspectiva utilizării lor într-un sistem de învăţare automată. În timpul dezvoltării unui sistem de învăţare, multistrategică [12], au apărut câteva probleme legate de construirea arborilor de justificare (arbori and/or tipici pentru astfel de sisteme). Articolul prezintă soluţiile găsite pentru aceste probleme, cu accent pe modul de manipulare a mulţimilor de substituţii pentru variabile. Pentru completitudinea prezentării, în prima partea a articolului, vor fi prezentate rezultatele clasice, legate de algoritmii de tip AO*. În a doua parte, va fi prezentată problema construcţiei arborilor de justificare (arbori de tip and/or) împreună cu soluţiile găsite pentru rezolvarea ei.

Cuvinte cheie: căutare, problem-reduction, algoritm AO*, arbori and/or, arbori de justificare, învăţare automată.

1. Introducere

Multe sisteme de învăţare multistrategică conţin un motor de inferenţă deductiv. Este vorba, în primul rând de cele care au o componentă EBL (învăţare bazată pe explica]ii) [2,5]. Astfel de sisteme sunt folosite în mod tipic pentru rezolvarea de probleme sau pentru clasificarea de concepte. Ele încearcă, dacă primesc, de pildă, un exemplu pozitiv de concept, să construiaşcă o justificare a apartenenţei exemplului la conceptul respectiv folosind cunoştinţele din baza de cunoştinţe. Justificarea unui astfel de exemplu se face construind un arbore de justificare ce conţine într-o formă compactă demonstraţia. Arborele construit se numeşte arbore de demonstrare (*proof tree*) dacă, pentru construcţia sa, s-au folosit numai inferenţe deductive. În cazul în care s-au utilizat şi alte tipuri de inferenţe (abducţii, analogii etc.) arborele se numeşte *arbore de justificare plauzibil* (PJT) [10,11]. Trebuie subliniat că articolul de faţă nu îşi propune să prezinte motivaţiile acestor metode bazate pe explicare. În mod justificat, se poate pune întrebarea, ce se învaţă dintr-un arbore conţinând numai instanţe ale unor reguli deductive. În acest caz, problema reprezentată de rădăcina arborelui se află în închiderea deductivă a bazei de cunoştinţe, deci sistemul pur şi simplu aplică ceea ce ştie deja. Sistemele bazate pe explicare încearcă să facă generalizări ale acestor demonstraţii şi, în general, se produce o creştere a eficienţei sistemului. Cu alte cuvinte, se produc nişte transformări ce pot fi caracterizate drept învăţare [2,4,5,10,11].

În cele ce urmează, se va presupune că motorul de inferenţă (cel care construieşte arbori de justificare) are la dispoziţie o bază de cunoştinte conţinând *fapte* (fapte certe din domeniul teoriei) şi *reguli*. Regulile nu sunt neapărat de natură deductivă (certe), putând fi şi de natură analogică, abductivă, inductivă etc. Singura restricţie este aceea că regulile trebuie să aibă o formă clauzală (concluzie implicată de ipoteze). Restricţia îşi găseşte justificarea în modul de construcţie al arborilor de justificare, prezentat0 în cele ce urmează.

Un exemplul pozitiv de concept, primit de sistem ca date de intrare, are, în general, forma:

$$C \leftarrow F1 \wedge \ldots \wedge Fn$$

adică, expertul uman (cel care furnizează exemplele de concepte) consideră că faptele *F1,...,Fn*, care sunt adevărate, sunt motivul pentru care *C* adevărat. *C* este un fapt care afirmă apartenenţa la un concept. De exemplu, dacă *C* este *poluare (Copşa-Mică, mare)*, acest fapt afirmă apartenenţa entităţii *Copşa-Mică* la conceptul de "zonă cu poluare intensă" (*poluare (x, mare)*).

Sistemul încearcă să găsească justificarea exemplului de intrare, utilizând baza de cunoştinţe îmbogăţită cu faptele *F1,...,Fn*. Acesta este procesul în care se utilizează un algoritm de tip AO*. În esenţă, se construieşte printr-o căutare descendentă (top-down) un arbore de tip and/or conţinând justificarea pentru *C* (scopul iniţial). Arborele construit va avea drept frunze fapte din baza de cunoştinţe, iar nodurile interne vor conţine instanţe ale regulilor din baza de cunoştinţe (inferenţe) utilizate în procesul de demonstrare. Modul de construcţie al nodurilor interne este bazat pe următoarea idee. Dacă trebuie demonstrat că *C* este adevărat, atunci se caută reguli în baza de cunoştinţe ce au ca parte stângă (concluzie) pe *C*[1]. Fiecare regulă găsită va fi instanţiată şi va produce un nod de tip *and*. Nodul curent (cel care are scopul *C*) va fi un nod de tip *or* având drept fii nodurile create pentru fiecare regulă. În acest mod problema justificării lui *C* a fost transformată în problemele *H11,...,H1n,...,Hm1,...Hmp*. În lucrarea de faţă prin nod *or* se înţelege un nod care este justificat imediat ce unul din fiii săi este justificat, iar prin nod *and*, un nod care este justificat dacă toţi fiii săi sunt justificaţi. Figura 1.1 ilustrează

* Această cercetare a fost finanţată parţial de către Academia Română, parţial de către US National Research Council under the Twinning Program with Bulgaria and Roumania, 1995-1996, parţial de către Universitatea "Politehnica" Bucureşti şi parţial de către ECE prin PECO/92 contractul nr. CIPA3510OCT920044.

[1] Partea stângă a regulii nu trebuie să fie egală cu *C* ci să se "potrivească" cu *C*. Acest test se face încercând unficarea celor doi termeni (vezi secţiunea 5).

procesul de expandare al unui nod din cadrul procesului de construcţie al unui arbcre de justificare.

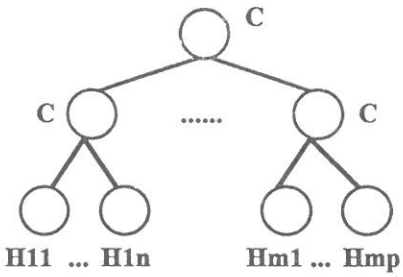


Figura.1.1. Procesul de expandare al unui nod din arborele de justificare. Nodurile *and* au legăturile către fii, desenate cu linii mai îngroşate

În ultimii ani, a apărut o nouă metodologie pentru abordarea problemei învăţării. Ea se bazează pe arborii de justificare plauzibili (metoda MTL-JT) [10,11,12]. Este unanim recunoscut că, pentru a face un salt calitativ semnificativ, sistemele de învăţare trebuie să utilizeze mai multe strategii de învăţare, încercând combinarea lor sinergistică. Metoda MTL-JT permite construcţia unui arbore de justificare ce conţine în nodurile interne, nu numai instanţe ale unor reguli deductive, ci şi ale altor tipuri de reguli. Aceste inferenţe nu vor avea plauzibilitate maximă (precum cele deductive) şi acest lucru ridică o problemă interesantă pentru algoritmii de construcţie, legată de funcţiile de cost ce trebuie utilizate [10].

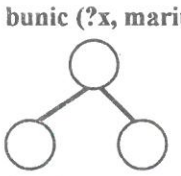
Problema cea mai dificilă în construirea arborilor de justificare plauzibili (arbori PJT) este legată de unificarea variabilelor şi de manipularea mulţimilor de substituţii pentru variabile, ce pot să apară. De exemplu, dacă scopul de rezolvat este,

bunic (?x, marius)

şi regula care are o parte stânga ce se potriveşte cu scopul este,

bunic (?x, ?y) ← părinte (?x, ?z) & părinte (?z, ?y)

nodul *and* , creat în arborele de justificare, va fi,



După crearea unui astfel de nod, algoritmul de construcţie PJT va încerca să demonstreze noile subscopuri. Când apar variabile, nu este însă destul să se demonstreze subscopurile fără a ţine seama de substituţiile obţinute pentru variabile în procesul de justificare. În cazul de faţă, soluţiile pentru cele două subscopuri trebuie să aibă o aceeaşi valoare pentru variabila *z*. Deci, trebuie ţinut seama de aceste substituţii pentru variabile. Există două momente de timp la care pot fi luate în consideraţie aceste substituţii. Primul ar fi când se trece la demonstrarea celui de-al doilea subscop după demonstrarea primului. În acest moment variabila *z* poate fi instanţiată cu valoarea găsită la primul subscop. Deşi această metodă funcţionează pentru algoritmi de căutare simpli (e.g., depth-first) ea este mult mai greu de implementat la algoritmi de tip AO* unde nu există nici o certitudine că fiul din stânga va fi demonstrat înaintea fiului din dreapta[2]. Al doilea moment ar fi după demonstrarea separată a tuturor subscopurilor. Se poate încerca la sfârşit "fuzionarea" mulţimilor de substituţii obţinute. Dacă cele două mulţimi sunt incompatibile (noţiunea va fi definită în secţiunea 5), atunci procesul de demonstrare al nodului respectiv eşuează, deşi separat s-au găsit soluţii pentru toţi fiii nodului.

Articolul este organizat după cum urmează. Secţiunea 2, *Arbori And/Or*, prezintă aspectele clasice ale teoriei căutării pentru arbori And/Or. Se definesc concepte de bază precum soluţia problemei, baza pentru soluţii etc. Secţiunea 3, *Strategia Best-First pentru arbori And/Or*, prezintă algoritmul generic de căutare în spaţiul problemei. În cadrul acestuia o seamă de noţiuni sunt păstrate la un nivel foarte general, cum ar fi calcularea criteriului de minim în evaluarea nodurilor, reuşindu-se astfel definirea unui algoritm generic, ce acoperă mai multe metode reale. Secţiunea 4, *Algoritmii AO şi AO**, prezintă variante de algoritmi best-first, având utilitate practică. Datorită modului în care se calculează bazele pentru soluţii şi a estimărilor pentru costuri, variantele pot fi implementate relativ eficient. Eficienţa se referă la procedurile de calcul pentru costurile utilizate în cadrul algoritmului, nu la complexitatea intrinsecă a problemei (explozia combinatorială a nodurilor din spaţiul problemei). Secţiunea 5, *Problema învăţării*, prezintă unele aspecte specifice sistemelor de învăţare, legate de

[2] Bineînţeles că, acest lucru poate fi controlat din funcţia de cost sau din mecanismul de expandare ai nodurilor, dar afirmaţia se referă la cazul general.

construcţia arborilor de justificare (de tip and/or). Este tratată problema substituţiilor de variabile, care pot să apară în urma instanţierii unor reguli din baza de cunoştinţe sau în urma unificării unor termeni cu fapte din baza de cunoştinţe. În sfârşit, secţiunea 6, *Concluzii*, prezintă câteva concluzii şi, mai ales, aspecte neelucidate complet şi care ar trebui studiate.

2. Arbori And/Or

În teoria căutării există două metode de formulare a problemelor, *state-space* şi *problem-reduction*. În metoda *state-space* (căutare în spaţiul stărilor) spaţiul de căutare ce trebuie explorat are forma unui graf, în timp ce în metoda *problem-reduction* (căutare în spaţiul problemei) are forma unui graf *and/or*. Forma specială a spaţiului de căutare în metoda *problem-reduction* implică şi algoritmi diferiţi de explorare.

Strategiile de căutare neinformate în spaţiul stărilor (breadth-first şi depth-first), pot fi uşor adaptate pentru explorarea grafurilor and/or. Principala diferenţă constă în definirea corectă a noţiunilor de *condiţie de terminare* şi *soluţie a problemei*. În căutarea în spaţiul stărilor, o soluţie este un drum în graful explorat (explicitat) de la nodul de start la un nod terminal şi poate fi complet caracterizată de proprietăţile unui singur nod (nodul terminal). Dacă nodul terminal, la care a ajuns procesul de căutare, satisface un anumit predicat (este sau nu soluţie) prin parcurgerea înapoi a drumului către rădăcină, obţinem soluţia completă.

La căutarea în spaţiul problemei (problem-reduction) lucrurile nu mai sunt deloc atât de simple. În acest caz, soluţiile nu sunt drumuri în arbori, ci chiar arbori. Pentru a verifica dacă a fost găsită o soluţie, trebuie văzut dacă o mulţime de noduri rezolvate (probleme elementare) induc un arbore soluţie având ca rădăcină scopul iniţial. Pentru a realiza acest lucru, de fiecare dată când se generează un nod, dacă se poate spune despre el că este rezolvabil sau nerezolvabil această informaţie este propagată înapoi (către predecesorii nodului) în porţiunea explicitată a grafului and/or. Dacă în acest proces de propagare însuşi nodul conţinând scopul iniţial (nodul rădăcină) este etichetat ca rezolvat, atunci o soluţie a fost obţinută. Pe de altă parte, dacă nodul rădăcină este etichetat ca nerezolvabil, atunci scopul iniţial nu are soluţie şi procesul de căutare se opreşte.

În cele ce urmează, se vor discuta aspecte legate de arbori and/or (nu grafuri and/or) şi se va presupune că un nod dintr-un astfel de arbore are următoarele câmpuri:

tip	specifică tipul nodului; valorea posibilă poate fi *and* sau *or*;
stare	specifică cunoştinţele curente, acumulate despre nod; nodul poate fi *rezolvat*, *nerezolvabil* sau *în-rezolvare*; ultimul caz este cel în care nu se ştie încă nimic despre nod; practic, înseamnă că nu se poate trage încă nici o concluzie despre rezolvabilitatea nodului;
cost	estimare a costului soluţiei pentru problema reprezentată de nod;
substituţii	câmp utilizat pentru a păstra legările de variabile, obţinute în urma proceselor de unificare; în cazul general, se poate face abstracţie de acest câmp;
succ	lista de noduri succesoare;
pred	lista de noduri predecesoare; în cazul arborilor and/or, un singur nod predecesor va fi în listă.

Procedura *etichetare* care propagă informaţia despre posibilitatea de rezolvare a nodurilor în arborele and/or şi care este esenţială în detectarea unei soluţii are următoarea formă:

```
etichetare (nod)

dacă nod este nod terminal atunci
  • nu se face nimic; la ieşirea din if se va
merge către predecesor
altfel dacă nod.tip = and atunci
  dacă toţi succesorii nodului nod sunt
rezolvaţi atunci
    nod.stare ← rezolvat
  altfel dacă cel puţin unul din succesorii
nodului nod este nerezolvabil, atunci
    nod.stare ← nerezolvabil
  altfel
    return false
altfel dacă nod.tip = or atunci
  dacă toţi succesorii nodului nod sunt
nerezolvabili atunci
    nod.stare ← nerezolvabil
  altfel dacă cel puţin unul din succesorii
nodului nod este rezolvat atunci
    nod.stare ← rezolvat
  altfel
    return false
```

dacă *nod* este nod rădăcină **atunci**
return true
altfel
etichetare (*nod*.pred)

În procedura de mai sus şi în cele ce urmează, se va numi nod *and* un nod care pentru a fi rezolvat trebuie să fie rezolvaţi toţi succesorii săi. Se va numi nod *or* un nod care este rezolvat dacă cel puţin unul dintre succesori este rezolvat. Procedura întoarce *true* dacă procesul de etichetare a reuşit să eticheteze rădăcina arborelui (rezolvat/nerezolvabil) şi *false* dacă procesul de etichetare s-a oprit undeva pe parcurs. Procedura ar putea conţine câte un apel recursiv pentru fiecare predecesor, pentru a funcţiona şi în cazul grafurilor and/or, dar în acest caz ar trebui luate precauţii suplimentare pentru a evita ciclurile posibile într-un graf.

Procedura *etichetare* este apelată când procesul de căutare a ajuns la un nod terminal care poate fi caracterizat ca *rezolvat* sau *nerezolvabil*. Nu ar avea nici un rost declanşarea procesului de propagare pentru un nod de tip *în-rezolvare* pentru că nu se poate aduce nici un aport de informaţie. Pentru nodurile *and* şi *or* dacă nu se poate spune nimic despre nod (*rezolvat* sau *nerezolvabil*) procesul de propagare se opreşte. Motivul opririi merită o scurtă justificare. Într-o prima instanţă s-ar putea crede că ar trebui propagată informaţia, pentru că un nod *or* poate fi etichetat *rezolvat* chiar dacă unul din succesori este *in-rezolvare* presupunând ca are totuşi un alt succesor etichetat *rezolvat*. Justificarea opririi propagarii este dată de faptul că procedura de propagare nu este apelată pe un arbore gata construit ce conţine numai noduri *în-rezolvare* ci este apelată incremental, la fiecare nou nod terminal găsit. În acest caz, dacă pentru un nod *or* s-ar fi găsit un nod succesor *rezolvat* oricum aceasta informaţie s-ar fi propagat şi deci propagarea în sus de pe un nod *in-rezolvare* nu ar aduce nici un aport informaţional. Trebuie subliniat că majoritatea sistemelor ce au un motor de inferenţă (acestea utilizează algoritmi de căutare pe arbori and/or) folosesc ipoteza *lumii închise* (closed world assumption) care face ca un nod terminal (care nu mai poate fi expandat) să fie sigur ori rezolvat ori nerezolvabil. Aceste sisteme nu pot opera cu cunoştinţe de genul "nu ştiu" sau "nu ştiu sigur".

Procedura *etichetare* ar putea fi scrisă top-down (parcurgând arborele and/or pornind de la rădăcină), dar acest lucru ar însemna traversarea întregului arbore pentru fiecare expandare a unui nod. Din cele prezentate mai sus rezultă că, acest lucru nu este necesar, procedura putând construi incremental informaţia *rezolvat/nerezolvabil*. Această caracteristică face ca procedura *etichetare* să fie "ieftină" (din punct de vedere al timpului de execuţie) şi, din fericire, majoritatea procedurilor ce operează asupra arborilor and/or pentru a calcula diferite informaţii au exact acelaşi şablon.

3. Strategia Best-First pentru arbori And/Or

După cum s-a văzut în secţiunea 2, în fiecare moment al procesului de căutare există mai multe noduri ce sunt candidate pentru expandare. Principiul metodei *best-first* este ca la fiecare moment de decizie (ce nod se expandează în continuare) să fie ales cel mai bun dintre *toţi* candidaţii posibili pentru explorare. Spre deosebire de reprezentarea în spaţiul stărilor, noţiunile de *candidat* şi *cel mai bun candidat* trebuie mult mai atent definite.

În principiu, un candidat reprezintă o mulţime de potenţiale soluţii. Cum în formularea *problem-reduction* soluţiile sunt arbori, candidaţii sunt mulţimi de arbori. În teoria căutării pentru arbori (grafuri) and/or, astfel de mulţimi se numesc *baze* (solution bases) şi o mare parte a efortului de căutare este orientat către identificarea lor.

Formal, fie G graful ce reprezintă porţiunea explicitată din spaţiul problemei. Este evident, din modul în care se construieşte, că G' este un graf conex conţinând nodul rădăcină şi având drept noduri de frontieră toate nodurile generate, dar neexpandate încă. Dintre acestea din urmă se va alege următorul nod de expandat. Orice subgraf al lui G' care poate fi potenţial extins la o soluţie trebuie privit ca un candidat pentru explorare. Se numeşte o bază B pentru G' un graf care îndeplineşte următoarele condiţii:

(1) B conţine nodul rădăcină

(2) dacă B conţine un nod *and,* atunci conţine şi toţi succesorii lui

(3) dacă B conţine un nod *or,* atunci conţine exact un succesor al acestuia

(4) nici un nod din B nu este etichetat *nerezolvabil*.

O procedură sistematică de enumerare a bazelor de la un moment dat este următoarea: se consideră că toate nodurile terminale (neexpandate încă) ale lui G sunt rezolvate (desigur cu execepţia celor deja etichetate ca nerezolvabile) şi apoi se determină mulţimea arborilor soluţie. Enumerarea bazelor este o operaţie costisitoare, dar, după cum se va vedea, ea se face tot într-o manieră incrementală, asemănătoare procedurii *etichetare*.

Acţiunea de alegere a *celui mai bun candidat* se formalizează ca un proces având două etape. În prima etapă, utilizând o funcţie de evaluare a grafurilor, F_g, se identifică cea mai bună bază. În

a doua etapă, în cadrul bazei alese, utilizând o funcţie de evaluare a nodurilor, F_n, se alege un nod pentru a fi expandat. Majoritatea cercetărilor în teoria căutării au fost direcţionate către studierea funcţiei F_g, cealaltă alegându-se într-o manieră ad-hoc.

În cele ce urmează, este prezentat algoritmul *best-first*. Trebuie subliniat că, deşi se foloseşte în mod curent denumirea de algoritm, de fapt este vorba de o clasă de algoritmi, deoarece o serie de aspecte sunt generice, putând fi instanţiate în multe feluri (e.g. funcţia F_g). În cadrul algoritmului sunt utilizate două structuri de date, *open* şi *closed* în care se păstrează nodurile generate şi, respectiv, nodurile expandate.

Algoritmul **best-first** (varianta 1)

1. Nodul de start (conţinând problema iniţială) se inserează în *open*.

2. Din graful explicitat G', se calculează cea mai promiţătoare (conform unui criteriu de optim) bază B, utilizând funcţia F_g şi informaţia euristică h. Informaţia h reprezint@ o estimare a costului unui nod şi este calculată pentru fiecare nod nou, generat la pasul (4).

3. Utilizând funcţia F_n, se alege un nod n, care se află şi în *open* şi în B'. Nodul n este extras din *open* şi inserat în *closed*.

4. Se expandează nodul n. Se inserează succesorii obţinuţi, atât în *open*, cât şi în G'. Pentru fiecare succesor n se calculează informatia h, adică un set de parametri ce caracterizează mulţimea arborilor soluţie pentru subproblema reprezentată de nodul n'.

5. Dacă există un succesor, n', al nodului n, care este nod terminal, atunci:

(a) se etichetează nodul ca *rezolvat* (dacă nodul reprezintă o problemă elementară) sau *nerezolvabil* (dacă nodul nu poate fi divizat mai departe în subprobleme);

(b) se propagă informaţia *rezolvat/nerezolvabil* spre rădăcina (procedura *etichetare*);

(c) dac@ nodul de start este etichetat *rezolvat*, atunci algoritmul se termină cu succes având B drept soluţie, iar dacă nodul de start este etichetat *nerezolvabil*, atunci algoritmul se termină cu eşec;

(d) se elimină din G nodurile ce nu mai pot influenţa etichetarea nodului de start (pasul este opţional fiind utilizat pentru a micşora consumul de memorie);

6. Se reia de la pasul (2).

Un arbore soluţie G_n pentru problema reprezentată de un nod n' este o mulţime de subarbori ai lui G, astfel încât să fie îndeplinite următoarele condiţii:

(i) n' este rădăcina lui G_n

(ii) n' poate fi etichetat *rezolvat* dacă se aplică procedura *etichetare* asupra lui G_n

Cum n' nu a fost încă expandat, G_n nu este accesibil. Informaţia h, ce trebuie calculată, va trebui să se bazeze pe cunoştinţe specifice domeniului problemei. Informaţia h are rolul de a caracteriza, atât dificultatea găsirii unei soluţii pentru problema reprezentată de nodul n' cât şi calitatea acestei soluţii. Există două principii fundamentale pentru algoritmii de căutare:

Principiul "small-is-quick". Acest principiu afirmă că, pentru a eficientiza procesul de căutare, trebuie utilizat un criteriu auxiliar de minimizare. Informaţia h trebuie aleasă astfel încât să dea o estimare suficientă pentru a calcula criteriul de minim.

Principiul "face-value". Acest principiu afirmă că, estimările imperfecte ale costului pentru nodurile de pe frontieră (generate, dar neexpandate), pot fi folosite în locul valorilor reale, putând astfel propaga estimările în sus către nodul rădăcină.

Pentru a exemplifica aceste principii, se presupune că se explorează un arbore and/or, din care porţiunea explicitată este prezentată în figura 3.1.

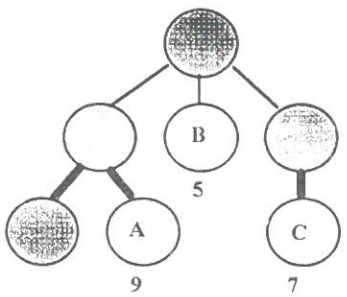


Figura 3.1. Arbore and/or. Nodurile *closed* sunt reprezentate prin cercuri închise la culoare, in timp ce nodurile *open* - prin cercuri deschise la culoare

Se presupune că se alege drept criteriu de minim numărul de arce din graful soluţie. În acest caz, cea mai bună soluţie este cea care are un număr minim de arce. Dacă informatia h, calculată pentru nodurile de pe frontiera A, B, C, este 9, 5, respectiv 7, atunci ,conform principiului *face-*

value, soluţia pentru nodul de start are costul estimat 6 (5, costul estimat pentru B, plus arcul ce leagă B de nodul de start). Celelalte soluţii au un cost mai ridicat. De exemplu, soluţia ce conţine nodul A are costul $12 = 3 + 9$.

Când se explorează grafuri (nu arbori), o problemă legată de evaluarea criteriului de minimizare poate să apară. Dacă un succesor al nodului curent de expandat se afla deja în *open*, adică costul estimat a fost deja propagat în sus, nu mai putem propaga pentru a doua oară această estimare pe noua cale spre rădăcina. Dacă am face aşa, atunci costul estimat pentru rădăcina nu ar mai fi corect, existând arce numărate de mai multe ori. Detaliul nu este important pentru probleme de satisfacere (satisficing problems) unde criteriul de minimizare este doar un instrument auxiliar în procesul de căutare, dar în cazul problemelor de optimizare, estimările trebuie propagate mult mai atent, acurateţea estimărilor fiind importantă.

Unul din rezultatele importante ale teoriei căutării este că, pentru a asigura găsirea unei soluţii optimale (în raport cu funcţia F_g), toate estimările de costuri trebuie să fie *optimiste* (mai mici decât costul real) [1,7,8]. În continuare, este prezentată o variantă îmbunătăţită a algoritmului best-first.

Algoritmul **best-first** (varianta 2)

1. Nodul de start (conţinând problema iniţială) se inserează în *open*.

2. Din graful explicitat G', se calculează cea mai promiţătoare (conform unui criteriu de optim) bază B, utilizând funcţia F_g şi informaţia euristica h. Informaţia h reprezintă o estimare a costului unui nod şi este calculată pentru fiecare nod nou generat la pasul (4). Dacă toate nodurile terminale din B pot fi etichetate *rezolvat,* algoritmul se termină cu B soluţie a problemei.

3. Utilizând funcţia F_n se alege un nod n care se află şi în *open* şi în B'. Nodul n este extras din *open* şi inserat în *closed*.

4. Se expandează nodul n. Se inserează succesorii obţinuţi, atât în *open,* cât şi în G'. Pentru fiecare succesor n' se calculează informatia h, adică un set de parametri ce caracterizează mulţimea grafurilor soluţie pentru subproblema reprezentată de nodul n'.

5. Dacă există un succesor n', al nodului n, care este nod terminal, atunci:

(a) se etichetează nodul ca *rezolvat* (dacă nodul reprezintă o problemă elementară) sau *nerezolvabil* (dacă nodul nu poate fi divizat mai departe în subprobleme);

(b) se propagă informaţia *rezolvat/nerezolvabil* spre rădăcina (procedura *etichetare*);

(c) dacă nodul de start este etichetat *nerezolvabil,* atunci algoritmul se termină cu eşec;

(d) se elimină din G nodurile ce nu mai pot influenţa etichetarea nodului de start (pasul este opţional fiind utilizat pentru a micşora consumul de memorie);

6. Se reia de la pasul (2).

Varianta a doua a algoritmului conţine o modificare ce vizează obţinerea unei soluţii optimale. La pasul (4c), dacă nodul rădăcină a fost etichetat *rezolvat,* algoritmul nu se mai termină imediat. În acest fel, se oferă posibilitatea găsirii unei soluţii mai bune (din punctul de vedere al criteriului de optim) dacă aceasta există. De observat că, forma aceasta a algoritmului este în primul rând interesantă din punct de vedere teoretic, unificând diferitele variante reale posibile. Bineînţeles că, într-o implementare, o serie de aspecte nu mai pot fi atât de clar delimitate (mai ales din motive de eficienţă.

4. Algoritmi AO şi AO*

Algoritmul *best-first* prezentat în sectiunea anterioară, nu impune nici o condiţie asupra formei sau a naturii funcţiilor F_g şi F_n. Forma lor este restricţionată în practică, din motive de eficienţă, pentru a profita de structura recursivă a arborilor and/or. Pot fi identificate două caracteristici ce uşurează calcularea funcţiilor F_g şi F_n:

Calcule comune. Anumite calcule pentru estimarea unor costuri pot fi reutilizate la evaluarea diferiţilor candidaţi.

Actualizarea selectivă. După calcularea unei estimări de cost pentru un nod, numai "strămoşii" săi necesită actualizarea estimărilor de costuri. Toate celelalte noduri îşi păstrează valorile nealterate. Această caracteristică permite scrierea procedurilor de parcurgere ale arborelui and/or într-o manieră incrementală (vezi procedura *etichetare*).

Pentru un graf soluţie G, dacă notăm cu C_g costul lui G (valoarea proprietăţii, aleasă drept criteriu de minimizare), putem defini funcţiile de cost recursive ca mai jos.

Definitie. Funcţie de cost recursivă.

O funcţie de cost se numeşte recursivă dacă pentru fiecare nod n din graful G, are forma:

$$C_g(n) = F[E(n), C_g(n_1), \ldots, C_g(n_k)]$$

unde $n_1, \ldots, n_k$ sunt succesorii imediaţi ai nodului n, $E(n)$ caracterizează proprietăţile locale ale nodului n, iar F este o funcţie de combinare arbitrară (roll-back function).

Identificarea grafului *B*, baza cea mai "promiţătoare"

Dacă întregul spaţiu de căutare ar fi explorat (şi explicitat), un graf soluţie optim ar putea fi construit şi costul său, $C^*(s)$, ar putea fi calculat ca fiind minimul peste toate grafurile soluţie pentru nodul de start s. Procedura de calcul a acestui cost ideal este definită mai jos:

cost-minim (nod)

dacă *nod* este nod terminal rezolvat **atunci**
$C^*(n) = v(n)$, unde $v(n)$ este costul asociat soluţionării unei probleme primitive
altfel, dacă n este un nod terminal nerezolvabil (fără succesori posibili), **atunci**
$C^*(n) = +\infty$, unde prin $+\infty$ se formalizează ideea de nerezolvabil
altfel, dacă *nod*.tip = and, **atunci**
$C^*(n) = F[E(n), C^*(n_1), \ldots, C^*(n_k)]$
altfel, dacă *nod*.tip = or, **atunci**
$C^*(n) = \min F[E(n), C^*(n_i)], i = 1, \ldots, k$

Graful soluţie optim B, poate fi definit în funcţie de C^*, ca fiind un graf ce îndeplineşte următoarele condiţii:

(1) nodul de start s, este în B
(2) dacă un nod *and* este în B, atunci toţi succesorii săi sunt în B
(3) dacă un nod *or* este în B, atunci unul şi numai unul dintre succesorii săi se află în B, anume acel succesor n'', pentru care,

$$F[E(n), C^*(n')] = \min F[E(n), C^*(n_i)], i = 1..k$$

În momentul când se iau decizii asupra nodului de expandat, spaţiul de căutare nu este complet explicitat, deci va trebui să se folosească principiul *face-value* utilizând estimări asociate fiecărui nod de pe frontiera curentă. Se poate, pe baza estimărilor, să se calculeze care este cea mai promiţătoare soluţie, obţinând o definiţie constructivă pentru F_g şi pentru B. Funcţia care calculează costurile nodurilor bazându-se pe estimări este prezentată mai jos.

cost-minim-estimat (nod)

dacă *nod* este un nod din *open*, **atunci**
return $h(n)$
altfel, dacă n este un nod terminal nerezolvabil (fără succesori posibili), **atunci**
return $+\infty$, unde prin $+\infty$ se formalizează ideea de nerezolvabil
altfel, dacă *nod*.tip = and, **atunci**
return $F[E(n), C^*(n_1), \ldots, C^*(n_k)]$
altfel, dacă *nod*.tip = or, **atunci**
return $\min F[E(n), C^*(n_i)], i = 1, \ldots, k$

Cea mai promiţătoare soluţie, B, poate fi definită ca un graf cu următoarele proprietăţi:

(1) nodul de start este în B

(2) dacă un nod *and* este în B, atunci toţi succesorii săi sunt în B

(3) dacă un nod *or* este în B, atunci un singur succesor al său va fi în B, anume acela pentru care se obţine un cost estimat minim.

Practic, identificarea lui B se face împreună cu calculul costului estimat într-o manieră bottom-up, marcând pentru fiecare nod *or* succesorul cu costul estimat minim. În termenii costului estimat, funcţia F_g poate definită prin relaţia $F_g(s) = cost\text{-}minim\text{-}estimat(s)$.

Dacă algoritmul best-first, prezentat în secţiunea anterioară, utilizează o funcţie de cost recursivă, atunci acesta se numeşte *algoritm AO*. Dacă, în plus, testul de terminare este amânat până când graful soluţie este ales ca cea mai promiţătoare bază (varianta a 2-a a algoritmului best-first), atunci el se numeşte *algoritm AO**. Forma algoritmului AO* este prezentată în cele ce urmează:

Algoritmul **AO***

1. Se crează graful de căutare G (partea explicitată a spaţiului de căutare) ce conţine iniţial numai nodul de start s. Se inserează s în *open*. B conţine iniţial doar nodul de start.

2. Dacă *open* şi B nu au noduri în comun, algoritmul se termină cu B soluţie.

3. Se selectează din B utilizând o funcție F_n (selecție de noduri), un nod n ce se află și în *open*. Se extrage n din *open* și se inserează în *closed*.

4. Se expandează n, inserând toți succesorii săi în *open*. Aceștia se adaugă și în graful G (pointeri înapoi spre n).

5. Pentru fiecare succesor n', care nu este nod terminal, se calculează un cost estimat, $h(n')$. Dacă n' este un nod terminal cu costul $v(n')$, atunci $h(n') = v(n')$., iar, dacă n' este nod terminal *nerezolvabil*, atunci $h(n') = +\infty$.

Dacă n' se află deja în G' (nu se întâmplă în cazul arborilor and/or), atunci $h(n') = cost\text{-}minim\text{-}estimat\ (n')$.

6. Se propagă în sus costul estimat pentru fiecare succesor utilizând funcția $F[]$. În procesul de propagare, se marchează și cel mai bun succesor pentru fiecare nod *or*, pentru a putea identifica graful B.

7. Dacă costul estimat pentru rădăcină este $+\infty$, atunci algoritmul se termină cu eșec. Altfel, se șterg din G toate nodurile ce nu mai pot influența costul estimat pentru rădăcină.

8. Se reia de la pasul (2).

La pasul (6) algoritmul de mai sus utilizează o procedură care propagă estimările costurilor. Aceasta este identică din punct de vedere al formei cu procedura *etichetare*. În general, din motive de eficiență, toate activitățile de tipul "propagare în sus" se fac într-o aceeași funcție care actualizează și costurile estimate și informația rezolvat/nerezolvabil etc.

5. Problema învățării

Sistemele de învățare ce au o componentă de tip EBL sau care produc justificări plauzibile sunt foarte utile tocmai datorită posibilității de a oferi explicații. Ele au două moduri de operare. În modul de achiziție de cunoștințe și învățare se creează o bază de cunoștințe conținând o teorie a unui domeniu de expertiză. În al doilea mod, modul sistem expert, se face exploatarea bazei de cunoștințe construite, sistemul fiind însă în stare să producă justificări pentru toate deciziile luate. Chiar și în modul exploatare sistemul continuă să învețe. Se pare că această simbioză între metodele învățării automate și cele ale achiziției de cunoștințe reprezintă tendința cercetărilor din ultimii ani [9].

Contextul în cadrul sistemelor de învățare în care are loc procesul de construcție al arborilor de justificare a fost prezentat în secțiunea 1. Operatorii de expandare a nodurilor sunt regulile din baza de cunoștințe. Acești operatori sunt generici, conținând variabile, și pentru a fi utilizați este nevoie să fie instanțiați (anumite variabile să primească valori). Trebuie subliniat faptul că, problema utilizării regulilor de inferență, plauzibile poate fi decuplată de construcția arborelui de justificare. Detaliile legate de gradul de certitudine al regulilor pot fi ascunse în funcția de cost (ce face ordonarea globală a nodurilor de expandat) și funcția de expandare a unui nod (ce caută toate regulile utilizabile la expandarea unui nod). Fiind făcută această abstractizare, algoritmul AO* poate trata regulile pur sintactic (ipoteze care implică o concluzie).

În figura 5.1 este prezentată o bază de cunostințe foarte simplă cu ajutorul căreia vom exemplifica algoritmul de construcție al arborelui de justificare. Conform celor de mai sus, faptul că în exemplu nu sunt decât reguli deductive nu simplifică în nici un fel problema.

Fapte (axiome)

mama (maria, ion) tatăl (paul, ion)
mama (maria, ana) tatăl (paul, ana)
mama (elena, petre) tatăl (bogdan, petre)

Reguli de inferență

părinte (?x, ?y) ← tatăl (?x, ?y)
părinte (?x, ?y) ← mama (?x, ?y)
bunic (?x, ?y) ← părinte (?x, ?z), părinte (?z, ?y)

Figura 5.1. Exemplu de bază de cunoștințe

Să considerăm arborele ce demonstrează scopul, *părinte (?x, ana)*. Arborele care se construiește în procesul de demonstrație este prezentat în figura 5.2. Nodurile terminale în cadrul arborilor de justificare sunt noduri ce unifică cu un fapt.

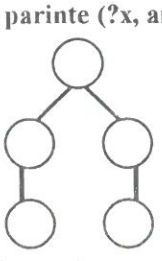


Figura 5.2. Arborele de justificare pentru scopul *parent (?x, ana)*.

Exemplul este foarte simplu, dar este ilustrativ pentru problemele care apar într-un sistem de învățare. Variabilele ce apar au fiecare atașate o mulțime de valori posibile (substituții).

Manipularea acestor valori reprezintă nucleul procesului de găsire al unei soluţii. Conceptual, se poate considera că arborele este format din procese cooperante (câte unul în fiecare nod). Există implementări în care efectiv arborele este un arbore and/or de corutine care îşi transmit mesaje reciproc [3]. Modelul corutinelor cooperante se dovedeşte a fi extrem de sugestiv. Mesajele ce se transmit sunt de cel puţin două tipuri. Mesaje "verticale" care merg de la fii către tată, şi mesaje "orizontale" care se transmit între fraţi. Orice algoritm AO*, într-o formă mai mult sau mai puţin explicită, realizează această transmitere de mesaje.

Pentru exemplul de mai sus, un răspuns al sistemului de genul "da, există un obiect aflat în relaţia *părinte* cu obiectul *ana*" este nesatisfăcător. Trebuie calculate şi valorile posibile pentru variabilele ce apar (în cazul de faţă *x*). Înainte de a prezenta amănuntele algoritmului trebuie definite câteva concepte ce vor fi utilizate. Vom numi *substituţie* o mulţime de perechi variabilă-valoare în care variabilele sunt toate diferite. De exemplu, *{?x/maria, ?y/ion}* este o substituţie. Un exemplu de *mulţime de substituţii*, noţiune care apare, de asemenea, în cadrul algoritmului, este *{{?x/maria}, {?x/ion}, {}}*.

Un nod terminal este un nod pentru care nu există o regulă din baza de cunoştinţe a cărei concluzie să unifice cu scopul nodului. În acest caz el este *rezolvat* dacă unifică cu un fapt din baza de cunoştinţe şi este *nerezolvabil* în caz contrar. Din procesul de unificare, dacă el se încheie cu succes va rezulta o mulţime de substituţii. De exemplu, pentru nodul terminal *tatăl (?x,ana)* rezultă mulţimea, *{{?x/paul}}*. Trebuie făcută distincţia între substituţia vidă, *{}*, şi eşecul unificării, notat cu *fail*. Pentru nodul terminal *tatăl (bogdan, petre)* substituţia rezultată este cea vidă în timp ce pentru *tatăl (paul, petre)* se obţine *fail*. Eşecul procesului de unificare în acest ultim exemplu este o consecinţă directă a ipotezei lumii închise. Este posibil ca, într-adevar, *paul* să fie tatăl lui *petre*, dar sistemul nu are cunoştinţă de acest fapt şi, în consecinţă, îl consideră fals.

Pentru a definitiva forma algoritmului, trebuie definit modul în care mulţimile de substituţii se propagă până la nodul pentru scopul iniţial. Dacă la nodul iniţial se ajunge cu *fail*, atunci problema nu are soluţie. Altfel, problema are soluţii şi mulţimea de substituţii oferă soluţiile posibile. Procedura de propagare substituţii este:

propagare-substituţii (nod)

dacă *nod* este un nod terminal, **atunci**
nod.stare ← *rezolvat* sau *nerezolvabil*
nod.substituţii ← *fail* sau rezultatul unificării cu fapte din KB

altfel, dacă *nod* este nod *and* ,**atunci**
dacă toţi succesorii nodului *nod* sunt *rezolvaţi*, **atunci**
nod.substituţii ← *fuzionare* (mulţimile de substituţii ale succesorilor)
dacă *nod*.substituţii ≠ *fail*, **atunci**
nod.stare ← *rezolvat*
altfel
return false
altfel dacă cel puţin un succesor este *nerezolvabil*, **atunci**
nod.stare ← *nerezolvabil*
nod.substituţii ← *fail*
altfel
return false

altfel, dacă *nod* este un nod *or*, **atunci**
dacă toţi succesorii nodului *nod* sunt *nerezolvabili* , **atunci**
nod.stare ← *nerezolvabil*
nod.substituţii ← *fail*
altfel, dacă cel puţin un succesor este *rezolvat*, **atunci**
nod.stare ← *rezolvat*
nod.substituţii ← *reuniune* (mulţimile de substituţii ale succesorilor)
altfel
return false

dacă *nod* este nod rădăcină, **atunci**
return true
altfel
propagare-substituţii (*nod*.pred)

Câteva explicaţii sunt necesare pentru a înţelege structura procedurii de mai sus. Procedura primeşte iniţial (la primul apel) ca parametru un nod terminal din arborele and/or parţial construit. Acest nod poate fi nerezolvabil, dacă nu s-a gasit nici un fapt în baza de cunoştinţe care să unifice cu scopul nodului, sau rezolvat caz în care în urma procesului de unificare există soluţii pentru scop sub forma unei mulţimi de substituţii. Trebuie subliniat că din modul în care se manipulează aceste substituţii rezultate în urma proceselor de unificare, se ia o decizie clară asupra naturii motorului de inferenţă. De exemplu, în implementările tipice de Prolog nu se lucrează cu mulţimi de substituţii , ci cu câte o substituţie o dată, existând mecanisme de revenire (backtrack) ce permit generarea incrementală a tuturor soluţiilor. Conceptual o asemenea abordare construieşte toţi arborii soluţie de tip *and* unul câte unul. În metoda prezentată aici şi în [12] se manipulează toate soluţiile deodată. Datorită acestui fapt soluţia obţinută este sub formă de arbore *and/or* ce păstrează într-o formă compactă toţi arborii soluţie de tip *and*. Nu este clar care metodă este mai eficientă. În orice caz, operaţiile de fuziune şi unificare de mulţimi de substituţii necesare în cadrul metodei doi sunt complexe şi în

cazul unor baze de cunoştinţe mari (conţinând multe fapte ce produc soluţii) este posibil să degradeze mult performanţele sistemului. Această direcţie de studiu va trebui neapărat cercetată. Cele două abordări devin echivalente dacă se doreşte o parcurgere exhaustivă a spaţiului problemei. Este exact ceea ce se întâmplă în [10,11,12] unde se doreşte cea mai bună soluţie posibilă, conform unui criteriu de minimizare. Dacă însă este satisfăcătoare obţinerea oricărei soluţii, atunci evident prima variantă este mult mai eficientă.

Pentru un nod *or* oricare dintre substituţii oferă o soluţie şi deci mulţimea rezultantă asociată nodului este reprezentată de reuniunea substituţiilor succesorilor. Trebuie remarcat că reuniunea se face într-o manieră incrementală. Nu este neapărat nevoie să aşteptăm soluţionarea tuturor succesorilor unui nod *or* pentru a calcula substituţii. De fapt operaţia corectă pentru un nod *or* în cadrul procedurii este,

nod.substituţii ←
reuniune (*nod*.substituţii, mulţimea de substituţii a noului succesor rezolvat).

In cazul nodurilor *and* trebuie însă aşteptat până ce toate nodurile devin rezolvate pentru a calcula mulţimea de substituţii. Operaţia poate fi făcută şi incremental, dar mulţimea de substituţii parţială oricum nu se poate propaga mai sus ca în cazul nodurilor *or*. Operaţia de fuziune modelează ideea că variabilele cu acelaşi nume din subscopuri diferite (succesoare ale aceluiaşi nod) să aibă valori egale. De exemplu, pentru substituţiile,

{?x/Mary, ?z/John} }i {?y/Peter, ?z/John}

prin fuziune, se obţine,

{?x/Mary, ?z/John, ?y/Peter}

în timp ce pentru,

{?x/Mary, ?z/John} }i {?y/Peter, ?z/Paul}

se obţine,

fail

Operaţia de fuziune se poate extinde la mulţimi de substituţii, fuzionând fiecare element dintr-o mulţime cu fiecare din cealaltă şi eliminând din noua mulţime rezultată *fail*-urile. De remarcat că, exceptând complexitatea intrinsecă a problemei explorării unui arbore and/or această operaţie de fuziune este cea mai costisitoare şi trebuie implementată foarte atent.

Reiese din cele prezentate ca în cazul arborilor de justificare, specifici sistemelor de învăţare pentru un nod *and* nu este destul ca toate subscopurile să fie rezolvate, mai trebuie studiată şi compatibilitatea soluţiilor găsite. Exceptând acest aspect un algoritm AO* de construcţie conţine exact aceleaşi etape prezentate în secţiunea 4, dar pe lângă toate celelalte informaţii (estimări de costuri, *rezolvat/nerezolvabil*) mai trebuie propagate în sus şi mulţimile de substituţii.

6. Concluzii

În articol au fost prezentaţi algoritmii clasici, utilizaţi în sistemele ce includ un motor de inferenţă, fie el deductiv sau plauzibil. Au fost, de asemenea, prezentate problemele care apar când se încearcă aplicarea algoritmilor generali în sistemele de învăţare. După cum s-a arătat, există mai multe aspecte care fac implementarea dificilă. În primul rând este vorba de limbajul de reprezentare (logică cu predicate de ordinul întâi) care face necesară gestionarea legărilor de variabile obţinute prin unificare. Această dificultate nu este caracteristică numai sistemelor de învăţare, ci tuturor sistemeler care îşi reprezintă cunoştinţele cu ajutorul logicii cu predicate de ordinul întâi (demonstratoare, planificatoare etc.). Un sistem care ar folosi ca limbaj de reprezentare a cunoştinţelor logica propoziţională ar fi mult mai uşor de implementat (nu există variabile,) dar şi mult mai puţin util. O a doua clasă de dificultăţi, este dată de definirea şi manipularea corectă a costurilor şi a inferenţelor plauzibile. De această dată sunt dificultăţi tipice sistemelor de învăţare, existând mai multe soluţii posibile.

Una din direcţiile viitoare de cercetare este crearea unei teorii unitare asupra utilizării cunoştintelor plauzibile (nesigure) în sistemele de învăţare. Există mai multe teorii asupra modului de propagare a gradelor de certitudine niciuna nefiind însă complet satisfăcătoare. De asemenea, ar trebui studiat care este cea mai bună abordare pentru construcţia unui arbore and/or, soluţia incrementală sau cea care construieşte în paralel toate soluţiile, din cel puţin două puncte de vedere: cel al unui sistem de învăţare şi cel al unui sistem deductiv general (tip Prolog).

Bibliografie

1. **BARR.,A., FEIGENBAUM, E.A.** (Eds.).: The Handbook of Artificial Intelligence. Vol. I. HeuristicPress, Stanford, California. William Kaufmann, Inc.,Los Altos, California, 1981.
2. **DeJONG, G., MOONEY, R.**: Explanation-Based Learning: an Alternative View. În: Machine Learning, Vol.1, 1986, pp.145-176.

3. **CĂLINOIU, S.**: Implementarea unui interpretor de PROLOG utilizând corutine. Lucrare de diplomă, Universitatea "Politehnica" Bucureşti, 1992.

4. **MINTON, S.**: Quantitative Results Concerning the Utility of Explanation-Based Learning. În: Artificial Inteligence, No.42, 1990, pp.363-392.

5. **MITCHELL, T.M., KELLER, T., KEDAR-CABELLI, S.**: Explanation - Based Generalization: a Unifying View. În: Machine Learning, Vol.1, 1986, pp.47-80.

6. **NILSON, N.J.** : Problem solving methods in Artificial Intelligence. New. York: McGraw-Hill, 1971.

7. **NORVIG, P.**: Paradigm of Artificial Intelligence Programming: Case studies in Common Lisp. Morgan Kaufmann Publishers, Inc., 1991.

8. **PEARL, J.** : HEURISTICS. Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley Publishing Company, 1984.

9. **TECUCI, G.:** Automating Knowledge Acquisition as Extending, Updating, and Improving a Knowledge Base. În: IEEE Transactions on Systems, Man, and Cybernetics, Vol.22, No.6, 1992.

10. **TECUCI, G.:** Plausible Justification Trees: A Framework for the Deep and Dynamic Integration of Learning Strategies. În: Machine Learning, Vol.11, pp.237-261, 1993.

11. **TECUCI, G.**: An Inference-Based Framework for Multistrategy Learning. In Michalski, R., Tecuci, G. (Eds.). Machine Learning. A Multistrategy Approach. Volume IV. Morgan Kaufmann Publishers, San Francisco, California, 1994.

12. **TECUCI, G., CĂLINOIU, S.,MARCU, D.**: Sistem de învăţare multistrategică, Raport de cercetare RR-05/1994, Centrul de Cercetări Avansate pentru Învăţare Automată, Prelucrarea Limbajului Natural şi Modelare Conceptuală, Academia Română, 1994.

INSTITUTUL DE CERCETĂRI ÎN INFORMATICĂ
B-dul Mareşal Averescu, Sect. 1, cod 71316, Bucureşti-Romania
ICI
RESEARCH INSTITUTE FOR INFORMATICS
8-10 Averescu Avenue, 71316, Bucharest 1, ROMANIA

TEACHER - SP v.1.

Descriere produs:

Autoinstruirea asistată de calculator în domeniul Metodologiei de realizare a proiectelor directoare de informatizare bazată pe facilităţile hypertext.

Domenii de aplicabilitate:

Însuşirea rapidă a cunoştinţelor metodologice de diferite grupări implicate în elaborarea unui proiect director: manageri, proiectanţi, utilizatori şi şefi de proiecte.

Configuraţie hardware-software:

- PC compatibil IBM-PC, memorie RAM - 2 Mb,
- monitor de preferinţă VGA color,

opţional:

- imprimantă,
- software: WINDOWS 3.1 şi HYPERRDOC 1.1 (produs ICI).

Punct de contact:

lab. Metodologii şi Instrumente asociate, Cristina Orăşanu, tel. 665.60.60 şi 665.70.15/212

Phone: +40-1-665.31.90
Telex: 11891 icpci r
Fax: +40-1-212.08.85

INSTITUTUL DE CERCETĂRI ÎN INFORMATICĂ
B-dul Mareşal Averescu, Sect. 1, cod 71316, Bucureşti-Romania
ICI
RESEARCH INSTITUTE FOR INFORMATICS
8-10 Averescu Avenue, 71316, Bucharest 1, ROMANIA

SISTEM DE PROGRAME PENTRU REŢELE NEURONALE

Descriere produs
Sistemul implementează următoarele arhitecturi de reţele neuronale: Hopfield, Bidirectional Access Memory (BAM), Multilayer Feed forward Perceptron (MLP) (Backpropagation), Counterpropagation, Self- Organization Map (Kohonen), Adaptive Resonance Theory (ART), Avalanche (Spatio-Temporal), Brain - State - in - a - Box, Learn Matrix, Boltzmann/Cauchy Machine

Sistemul prezintă următoarele caracteristici

- lucru cu masive oricât de mari de date
- alocarea dinamică a tuturor structurilor de date folosind inclusiv memorie extinsă şi la nevoie şi memorie externă
- lucru cu diverse structuri de fişiere de intrare (text, binar)
- configurare dinamică impusă de datele de intrare şi ieşire
- structuri şi proceduri comune de intrare/ieşire
- facilităţi de grafică, lucrul cu mouse-ul
- mesaje bilingve (engleză, română)
- interfaţă prietenoasă la utilizator (help, meniuri)

Domeniu de aplicabilitate
Orice domeniu de activitate dar cu prioritate în domenii cu următoarele tipuri de probleme: recunoaşterea formelor (vorbire, imagini, etc.), clasificări şi clusterizări (medicină, agricultură, ecologie, etc), previziuni şi prognoze (economic, finanţe, etc.), identificarea sistemelor şi dinamica proceselor (industrie etc.), optimizări.

Configuraţie hardware şi software

- Microcalculatoare compatibile IBM-PC cu sisteme de operare DOS şi cu minim de memorie internă de 2MB şi memorie externă de 40MB.

În prezent se pune la punct o variantă Windows.
Punct de contact ICI lab. 1.7. Dr. Ion Ciucă Tel: 665.60.60/221/267

Phone: +40-1-665.58.05
Telex: 11891 icpci r
Fax: +40-1-212.08.29