1. Cautarea de tip breadth-first. Prezentare generala si implementare in Prolog

Prezentare generala:

Strategia de cautare de tip BF extinde mai intai nodul radacina, apoi se extind toate nodurile generate de nodul radacina, apoi succesorii lor, si asa mai departe. Nodurile aflate la adancimea d in arbore sunt extinse inaintea nodurilor aflate la adancimea d+1. Spunem ca aceasta cautare este o cautare in latime.

Cautarea BF este un tip de cautare neinformata, adica nu se cunosc pasii sau costul drumului de la starea curenta la starea scop. Cautarea BF este completa si optima cu conditia ca costul drumului sa fie o functie descrescatoare de adancime a nodului.

Aceasta cautare este sistematica fiindca ia in consideratie toate drumurile de lungime 1, de lungime 2, samd. Daca exista o solutie, este sigur ca va fi gasita, iar daca exista mai multe, se va gasi intotdeauna solutia cea mai putin adanca.

Complexitate timp si spatiu: $O(b^d)$, unde b=nr de stari noi, iar d=lungimea drumului solutiei

Implementare in Prolog:

Pentru a programa in Prolog aceasta strategie este nevoie de o multime de noduri candidate, care vor forma o multime de drumuri candidate. Aceasta multime va fi reprezentata ca o lista de drmuri, iar in fiecare drum va fi o lista de noduri in ordine inversa. Capul listei fiind nodul cel mai recent generat, iar ultimul element este nodul de unde s-a inceput cautarea.

Cautarea este inceputa cu o multime de candidati avand un singur element: [[NodStart]].

Strategia BF este:

- daca primul drum contine un nod-scop pe post de cap, atunci aceasta este o solutie a problemei.

- altfel, inlatura primul drum din multimea de candidati, si genereaza toate extensiile de un pas ale acestui drum, care se adauga la sfarsitul multimii pe care se va face cautarea BF.

Cod:

```
%concat(+Lista1,+Lista2,-Listarez)
concat([],L,L).
concat([H|T],L,[H|T1]):-concat(T,L,T1).

%membru(+Element,+Lista)
membru(H,[H|_]).
membru(X,[_|T]):-membru(X,T).

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

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

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

`rezolva_b(+Start,- Sol)`:- este adevarat daca Sol este un drum (in ordine inversa) de la nodul Start la un nod scop.
`breadthfirst(+Listadrumuri,-DrumSolutie)`:- este adevarat daca un drum din multimea de noduri candidate Drumuri1 poate fi extins la o stare scop, rezultand Sol.
`concat(+Lista1,+Lista2,-Listarez)`:- este adevarat daca, atunci cand concatenam Drumuri cu DrumuriNoi rezulta Drumuri1.
`membru(+Element,+Lista)`:- este adevarat daca NodNou apartine listei [Nod|Drum].
`scop(Nod)`:- este adevarata daca Nod este scop al cautarii.
`s(Nod,NodNou)`:- este functia de succesiune si desemneaza faptul ca NodNou este succesor al Nod.

2. Cautarea de tip depth-first. Prezentare generala si implementare in Prolog

Prezentare generala:

Cautarea de tip depth-first este o caurate de tip neinformata. Ea extinde intotdeauna unul dintre nodurile alfate la nivelul cel mai adanc din arbore. Cautarea se intoarce inapoi si sunt extinse noduri de la adancimi mai mici doar daca a fost atins un nod care nu e scop si nu mai poate fi extins. Spunem ca aceasta este o cautare in adancime.

Necesitatile de memorie sunt foarte mici deoarece se memoreaza un singur drum de la radacina la nod-frunza, impreuna cu nodurile frate neextinse.

Dpdv al complexitatii spatiu, pentru un spatiu al starilor cu factor de ramificare b si m adancime maxima, vom memora doar bm noduri. Complexitatea timp este $O(b^m)$ inclusiv in cazul cel mai nefavorabil.

Dezavantaje:

- poate intra in ciclu infinit (rezolvabil prin adaugarea unui test de apartenta)
- poate gasi ca solutie un drum de lungime mai mare decat cel optim
- nu e nici complet nici optim

Avantaje:

- consum redus de memorie
- posibilitatea gasirii unei solutii fara a se explora o mare parte din spatiul de cautare

Implementare in Prolog:

Strategia depth-first este cea care se potriveste cel mai bine cu stilul de programare recursiv din Prolog. Strategia gasirii unui drum solutie de la un nod dat,N, pana la un nod scop este:

- daca N este un nod scop, atunci Sol=[N]

- altfel, daca exista un nod N1, succesor al lui N, astfel incat sa existe un drum , Sol1, de la N1 la un nod-scop , atunci Sol=[N|Sol1].

Cod fara mecanism de detectare a ciclurilor:

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

`rezolva_d(+Nod,-DrumSolutie)`:- este adevarat daca Sol este un drum de la Nod la un nod scop, insa poate rula la infinit in cazul ciclurilor

Cod cu mecanism de detectare a ciclurilor:

```
%membru(+Element,+Lista)
membru(H,[H|_]).
membru(X,[_|T]):-membru(X,T).

%rezolva_d(+Nod,-DrumSolutie)
rezolva_d(N,Sol):-depthfirst([],N,Sol).

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

`%rezolva_d(+Nod,-DrumSolutie)`:- este adevarat daca DrumSolutie este un drum de la Nod la un nod scop; va intoarce in DrumSolutie, drumul gasit intre Nod si un nod scop in ordine inversa.

`depthfirst(+Drum,+Nod, -Solutie)`:- este varianta imbunatatita a programului, avand un mecanism de detectare a ciclurilor; aici Nod este starea de la care trebuie sa pornim spre o stare scop, Drum este lista de noduri intre cel de start si Nod, iar Solutie este Drum extins via Nod spre un nod scop.

`membru(+Element,+Lista)`:- este adevarat daca Element apartine listei [Lista].

3. Cautarea in adancime iterativa. Prezentare generala si implementare in Prolog

Prezentare generala:

Cautarea in adancime iterativa este o cautare de tip neinformata. Reprezinta este o strategie care evita chestiunea stabilirii unei adancimi optime la care trebuie cautata solutia, prin testarea tuturor limitelor de adancime posibile: mai intai adancimea 0, apoi 1, etc. Acesti tip de cautare imbina beneficiile DF si BF:

- este optima si completa (BF)
- consuma cantitate mica de memorie (DF) deoarece cerinta de memorie este liniara

Ordinea extinderii este aceeasi ca la cautarea BF, numai ca anumite ramuri sunt extinse de mai multe ori. Strategia garanteaza gasirea nodului scop de la adancime minima, daca acesta exista. In general, este preferata atunci cand exista un spatiu al cautarii foarte mare, iar adancimea solutiei nu e cunoscuta.

Desi anumite noduri sunt extinse de mai multe ori, numarul total de noduri extinse nu e mai maire decat in cazul cautarii BF. Fie un arbore cu factor de ramificare b:

- nr noduri BF: $(b^{d+1}-1)/(b-1)$, d = adancimea la care se gaseste ultimul nod generat
- nr noduri DF: $(b^{j+1}-1)/(b-1)$, j = nivelul la care s-a oprit cautarea
- nr noduri ID: $(b^{d+2}-2d-bd+d+1)/(b-1)^2$, d = adancimea la care a ajuns cautarea (efectuand cautari DF complete, separate pt toate adancimile pana la d)

Complexitatea timp este $O(b^d)$, iar complexitatea spatiu O(bd).

Implementare in Prolog:

```
%membru(+Element,+Lista)
membru(H,[H|_]).
membru(X,[_|T]):-membru(X,T).

%cale(+Nod1,-Nod2,-Drum).
cale(Nod,Nod,[Nod]).
cale(NodInitial,NodUltim, [NodUltim|Drum]):-
cale(NodInitial,NodPenultim, Drum),
s(NodPenultim,NodUltim),
\+ membru(NodUltim,Drum).

%depthfirst_iterative_deepening(+NodStart, -Solutie)
depthfirst_iterative_deepening(NodStart, Sol):-
cale(NodStart,NodScop,Sol),scop(NodScop),!.
```

`cale(+Nod1,-Nod2,-Drum)` :- este adevarat daca Drum reprezinta o cale aciclica intre Nod1 si Nod2 in spatiul starilor; predicatul va genera toate drumurile aciclice posibile de lungime care creste cu cate o unitate si se va incheia atunci cand o cale se termina cu un NodScop.

`depthfirst_iterative_deepening(+NodStart, -Solutie)`:- este adevarat daca Solutie este un drum de la NodStart la un nod scop; va intoarce in Solutie, drumul gasit intre Nod si un nod scop in ordine inversa.

`membru(+Element,+Lista)`:- (mecanism de tratare al ciclurilor) este adevarat daca Element apartine listei [Lista].

4. Algoritmul A*. Textul algoritmului si admisibilitatea acestuia

Cautarea de tip bestfirst este o cautare informata sau cautare euristica. Ea difera de tehnica BF prin inaintarea in mod preferential de-a lungul unor noduri pe care informatia euristica le indica ca aflandu-se pe drumul cel mai bun catre scop.

Principiile pe care se bazeaza aceasta cautare sunt:

- se presupune existenta unei functii euristice de evaluare f-caciula, care depinde de fiecare problema, cu scopul de a ne ajuta sa decidem care nod trebuie extins la pasul urmator
- se extinde nosul cu cea mai mica valoare a lui f-caciula(n)
- procesul se incheie atunci cand urmatorul nod de extins este un nod-scop

Pentru a nu fi indusi in eroare de o euristica prea optimista, este necesar sa consideram si nevoia de a ne intoarce pentru a explora drumuri gasite anterior. De aceea ii vom adauga lui f-caciula un factor de adancime f-caciula(n) = h-caciula(n) + g-caciula(n), unde:

- g-caciula – estimatie a adancimii lui nin graf, reprezinta lungimea celui mai scurt drum de la nodul de start la n
- h-caciula – evaluare euristica a nodului n

Textul algoritmului A*:

Specificam familia functiilor f-caciula care vor fi folosite:

- h(n) = costul efectiv al drumului de cost minim intre nodul n si un nod scop, considerand toate nodurile scop posibile si toate drumurile posibile de la n la ele
- g(n) = costul unui drum de cost minim de la nodul de start n_0 la nodul n

⇨ f(n) = g(n) + h(n) – costul unui drum de cost minim de la nodul de start la un nod scop, drum ales dintre toate nodurile care trebuie sa treaca prin n

Consideram:

- h-caciula – factor euristic, estimatie a lui h(n)
- g-caciula – factor de adancime = costul drumului de cost minim pana la n

Algoritmul va folosi functia f-caciula = h-caciula + g-caciula.

1. Creeaza un graf de cautare G, constand numai din nodul inital n_0. Plaseaza n_0 intr-o lista numita OPEN.
2. Creeaza o lista numita CLOSED, care initial este vida.
3. Daca lista OPEN este vida, EXIT cu esec.
4. Selecteaza primul nod din lista OPEN, inlatura-l din OPEN si plaseaza-l in lista CLOSED. Numeste acest nod n.
5. Daca n este nod scop, opreste executia cu succes. Returneaza solutia obtinuta urmand un drum de-a lungul pointerilor de la n la n_0 in G.
6. Extinde nodul n, generand o multime, M, de succesori ai lui care nu sunt deja stramosi ai lui n in G. Instaleaza acesti membri ai lui M ca succesori ai lui n in G.
7. Stabileste un pointer catre n de la fiecare dintre membrii lui M care nu se gaseau deja in G (adica nu se afla nici in OPEN, nici in CLOSED). Adauga acesti membri si lui M listei OPEN. Pentru fiecare membru, m, al lui M, care se afla deja in OPEN/CLOSED, redirectioneaza pointerul sau catre n, daca cel mai bun drum la m gasit pana in acel moment trece prin n. Pentru fiecare membru al lui M care se afla deja in lista CLOSED, redirectioneaza pointerii fiecaruia dintre descendentii sai din G astfel inacat acestia sa

tinteasca inapoi de-a lungul celor mai bune drumuri pana la acesti descendenti, gasite pana in acel moment.

8. Reordoneaza lista OPEN in ordinea valorilor crescatoare ale functiei f-caciula.
9. Mergi la pasul 3.

Admisibilitatea algoritmului A*:

Conditii asupra grafurilor:

- Orice nod al grafului, daca admite succesori, are un numar finit de succesori.
- Toate arcele din graf au costuri mai mari decat o cantitate pozitiva, ε.

Conditii asupra lui h^:

- pentru toate nodurile n din graful de cautare, h^(n)=<h(n), altfel spus h^ nu supraestimeaza niciodata valoare efectiva a lui h.

Prin aceste conditii, A* garanteaza gasirea unui drum de cost minim.

Teorema 1: Atunci cand sunt indeplinite conditiile asupra grafurilor si asupra lui h^ si cu conditia sa existe un drum de cost finit de la nodul initial, n_0, la un nod scop, A* garanteaza gasirea unui drum de cost minim.

Demonstratia Teoremei 1:

*

Lema 1: Inainte de terminarea algoritmului A*, la fiecare pas, exista intotdeauna un nod, n*, in lista OPEN, cu urmatoarele proprietati:

- n* este pe un drum optim la scop
- A* a gasit un drum optim pana la n*
- $f^(n^*)=<f(n_0)$.

Demonstratia Lemei 1:

Demonstram prin inuductie. Pentru a demonstra ca la fiecare pas al lui A* concluziile lemei sunt valabile, este suficient sa demonstram:

1. sunt valabile la inceputul algoritmului
2. daca sunt valabile inainte de extinderea unui nod, vor fi valabile si dupa

1. Cazul de baza: la inceputul cautarii cand numai nodul n_0 a fost selectat pentru extindere, n_0 se afla in OPEN, este un drum optim la scop si A* a gasit acest drum. De asemenea, $f^(n_0)=<f(n_0)$ deoarece $f^(n_0) = h^(n_0) =< f(n_0)$. Astfel, nodul n_0, poate fi in acest stadiu, nodul n* al lemei.
2. Pasul de inductie: presupunem adevarate concluziile lemei la momentul in care au fost extinse m noduri (m>=0) si aratam ca ele sunt adevarate si pentru cand au fost extinse m+1 noduri.

Fie n* nodul din ipoteza si din lista OPEN, nod care se afla pe un drum optim gasit de A* dupa extinderea a m noduri.

Cazul I: Daca n* nu este selectat pentru extindere la pasul m+1, n* are aceleasi proprietati pe care le avea inainte, deci se demonstreaza pasul de inductie.

Cazul II: Daca n* este selectat pentru extindere, toti succesorii sai noi vor fi pusi in OPEN. Cel putin unul dintre ei, presupunem n_p, va fi pe un drum optim la un scop (deoarece, prin ipoteza, un drum optim trece prin n* si automat trebuie sa continue prin unul dintre succesorii sai). A* a gasit un nod optim la n_p, acel drum mai bun ar reprezenta si un nod mai bun catre scop, contrazicand ipoteza conform careia nu exista un drum mai bun catre scop decat cel gasit de A*

ca trecand prin n*. Deci in acest caz, permitem lui n_p sa die noul n* pentru pasul al (m+1)-lea. Pasul de inductie e partial demonstrat.
Demonstram acum f^(n_0)=<f(n_0) pentru toti pasii m inaintea terminarii algoritmului. Pentru orice nod, n*, aflat pe un drum optin gasit de A* avem:

- f^(n*) = g^(n*) + h^(n*) <=
- <= g(n*) + h(n*) <= (deoarece g^(n*) = g(n*) si h^(n*) =< h(n*))
- <= f(n*) <= (deoarece g(n*) + h(n*) = f(n*))
- <= f(n_0) (deoarece f(n*) = f(n_0), intrucat n* se afla pe un drum optim, completand demonstratia lemei).

*
Considerand Lema 1, vom arata ca algortimul A* se termina daca exista un scop accesibil si, mai mult, se termina prin gasirea unui drum optim la scop.

Aratam ca A* se termina: sa presupunem ca algoritmul nu se termina. In acest caz, A* continua sa extinda noduri la infinit si, la un moment dat, incepe sa extinda noduri la o adancime mai mare in arborele de cautare decat orice limitare finita a adancimii (s-a presupus ca graful in care se face cautarea are factor de ramificare finit). Intrucat costul fiecarui arc este mai mare decat $\varepsilon>0$, valorile lui g^ (prin urmare, ale lui f^) ale tuturor nodurilor din OPEN vor depasi, pana la urma pe f(n_0), contrazicand Lema 1.

Aratam ca A* se termina cu gasirea unui drum optim: A* se poate termina numai la pasul 3 (daca lista OPEN este vida) sau la pasul 5 (ajungand intr-un nod scop).

- O terminare la P3 poate interveni numai in cazul unor grafuri finite care nu contin niciun nod scop, iar teorema afirma ca A* gaseste un drum optim, numai daca exista.

Sa presupunem ca A* se termina prin gasirea unui nod scop care nu este optim, de exeplu prin gasirea unui n_{g2} cu f(n_{g2}) > f(n_0) si in conditiile in care exista un scop optim, n_{g1} != n_{g2}, cu f(n_{g1}) = f(n_0). Atunci cand are loc terminarea in nodul n_{g2}, f^(n_{g2}) >= f(n_{g2}) > f(n_0). Dar, chiar inainte ca A* sa il selecteze pe n_{g2}, conform Lemei 1, a existat un n* in OPEN aflat pe drumul optim, cu f^(n*) <= f(n_0). Prin urmare A*, nu l-ar fi selectat pe n_{g2}, deoarece A* selecteaza intotdeauna f^-valoarea cea mai mica.

Teorema a fost demonstrata.

<u>Definitia 1</u>: Orice algoritm care garanteaza gasirea unui drum optim la scop este un algoritm admisibil.

Prin urmare, A* este un algoritm admisibil. Prin extensie, vom spune ca orice functie h^ care nu supraestimeaza h este admisibila. (daca avem 2 versiuni de A* cu $h^_1 < h^_2$ vom spune ca A^*_2 este mai informat decat A^*_1).

<u>Teorema 2:</u> Daca algortimul A^*_2 este mai informat decat A^*_1, atunci la terminarea cautarii pe care cei doi algoritmi o efectueaza asupra oricarui graf, avand un drum de la n_0 la un nod scop, fiecare nod extins de catre A^*_2 este extins si de A^*_1.

5. Admisibilitatea si optimalitatea Algoritmului A*

Admisibilitatea algoritmului A*:

Conditii asupra grafurilor:

- Orice nod al grafului, daca admite succesori, are un numar finit de succesori.
- Toate arcele din graf au costuri mai mari decat o cantitate pozitiva, ε.

Conditii asupra lui h^:

- pentru toate nodurile n din graful de cautare, h^(n)=<h(n), altfel spus h^ nu supraestimeaza niciodata valoare efectiva a lui h.

Prin aceste conditii, A* garanteaza gasirea unui drum de cost minim.

Teorema 1: Atunci cand sunt indeplinite conditiile asupra grafurilor si asupra lui h^ si cu conditia sa existe un drum de cost finit de la nodul initial, n0, la un nod scop, A* garanteaza gasirea unui drum de cost minim.

Demonstratia Teoremei 1:

*

Lema 1: Inainte de terminarea algoritmului A*, la fiecare pas, exista intotdeauna un nod, n*, in lista OPEN, cu urmatoarele proprietati:

- n* este pe un drum optim la scop
- A* a gasit un drum optim pana la n*
- f^(n*)=<f(n0).

Demonstratia Lemei 1:

Demonstram prin inuductie. Pentru a demonstra ca la fiecare pas al lui A* concluziile lemei sunt valabile, este suficient sa demonstram:

1. sunt valabile la inceputul algoritmului
2. daca sunt valabile inainte de extinderea unui nod, vor fi valabile si dupa

1. Cazul de baza: la inceputul cautarii cand numai nodul n0 a fost selectat pentru extindere, n0 se afla in OPEN, este un drum optim la scop si A* a gasit acest drum. De asemenea, f^(n0)=<f(n0) deoarece f^(n0) = h^(n0) =< f(n0). Astfel, nodul n0, poate fi in acest stadiu, nodul n* al lemei.
2. Pasul de inductie: presupunem adevarate concluziile lemei la momentul in care au fost extinse m noduri (m>=0) si aratam ca ele sunt adevarate si pentru cand au fost extinse m+1 noduri.

Fie n* nodul din ipoteza si din lista OPEN, nod care se afla pe un drum optim gasit de A* dupa extinderea a m noduri.

Cazul I: Daca n* nu este selectat pentru extindere la pasul m+1, n* are aceleasi proprietati pe care le avea inainte, deci se demonstreaza pasul de inductie.

Cazul II: Daca n* este selectat pentru extindere, toti succesorii sai noi vor fi pusi in OPEN. Cel putin unul dintre ei, presupunem np, va fi pe un drum optim la un scop (deoarece, prin ipoteza, un drum optim trece prin n* si automat trebuie sa continue prin unul dintre succesorii sai). A* a gasit un nod optim la np, acel drum mai bun ar reprezenta si un nod mai bun catre scop, contrazicand ipoteza conform careia nu exista un drum mai bun catre scop decat cel gasit de A* ca trecand prin n*. Deci in acest caz, permitem lui np sa die noul n* pentru pasul al (m+1)-lea. Pasul de inductie e partial demonstrat.

Demonstram acum f^(n0)=<f(n0) pentru toti pasii m inaintea terminarii algoritmului. Pentru orice nod, n*, aflat pe un drum optin gasit de A* avem:

- f^(n*) = g^(n*) + h^(n*) <=
- <= g(n*) + h(n*) <= (deoarece g^(n*) = g(n*) si h^(n*) =< h(n*))
- <= f(n*) <= (deoarece g(n*) + h(n*) = f(n*))
- <= f(n0) (deoarece f(n*) = f(n0), intrucat n* se afla pe un drum optim, completand demonstratia lemei).

*

Considerand Lema 1, vom arata ca algortimul A* se termina daca exista un scop accesibil si, mai mult, se termina prin gasirea unui drum optim la scop.

Aratam ca A* se termina: sa presupunem ca algoritmul nu se termina. In acest caz, A* continua sa extinda noduri la infinit si, la un moment dat, incepe sa extinda noduri la o adancime mai mare in arborele de cautare decat orice limitare finita a adancimii (s-a presupus ca graful in care se face cautarea are factor de ramificare finit). Intrucat costul fiecarui arc este mai mare decat ε>0, valorile lui g^ (prin urmare, ale lui f^) ale tuturor nodurilor din OPEN vor depasi, pana la urma pe f(n0), contrazicand Lema 1.

Aratam ca A* se termina cu gasirea unui drum optim: A* se poate termina numai la pasul 3 (daca lista OPEN este vida) sau la pasul 5 (ajungand intr-un nod scop).

- O terminare la P3 poate interveni numai in cazul unor grafuri finite care nu contin niciun nod scop, iar teorema afirma ca A* gaseste un drum optim, numai daca exista.

Sa presupunem ca A* se termina prin gasirea unui nod scop care nu este optim, de exeplu prin gasirea unui ng2 cu f(ng2) > f(n0) si in conditiile in care exista un scop optim, ng1 != ng2, cu f(ng1) = f(n0). Atunci cand are loc terminarea in nodul ng2, f^(ng2) >= f(ng2) > f(n0)¬¬¬. Dar, chiar inainte ca A* sa il selecteze pe ng2, conform Lemei 1, a existat un n* in OPEN aflat pe drumul optim, cu f^(n*) <= f(n0). Prin urmare A*, nu l-ar fi selectat pe ng2, deoarece A* selecteaza intotdeauna f^-valoarea cea mai mica.

Teorema a fost demonstrata.

Definitia 1: Orice algoritm care garanteaza gasirea unui drum optim la scop este un algoritm admisibil.

Prin urmare, A* este un algoritm admisibil. Prin extensie, vom spune ca orice functie h^ care nu supraestimeaza h este admisibila. (daca avem 2 versiuni de A* cu h^1 < h^2 vom spune ca A*2 este mai informat decat A*1).

Teorema 2: Daca algortimul A*2¬ este mai informat decat A*1, atunci la terminarea cautarii pe care cei doi algoritmi o efectueaza asupra oricarui graf, avand un drum de la n0 la un nod scop, fiecare nod extins de catre A*2 este extins si de A*1.

Optimalitatea algoritmului A*:

Fie G o stare scop optimala cu un cost al drumului f*. Fie G_2 o a doua stare scop, suboptimala, care are un cost al drumului $g(G_2) > f^*$.

Presupunem ca A* selecteaza din coada, pentru extindere, pe G_2. Intrucat G_2 este o stare scop, aceasta aleger ar incheia cautarea cu o solutie suboptimala. Vom arata ca nu e adevarat.

Fie un nod n, care este, la pasul curent un nod frunza pe un drum optim la G (un asemenea nod sigur exista, daca nu ar exista, algoritmul ar fi returnat G) . Pentru acest nod, intrucat h^ este admisibila, trebuie sa avem :

$f^* >= f(n)$ (1).

Mai mult, daca n nu este ales pentru extindere in favoarea lui G_2, trebuie sa avem:

$f(n) >= f(G_2)$ (2).

Combinand (1) cu (2) obtinem:

$f^* >= f(G_2)$ (3).

Dar, cum G_2 e o stare scop, avem $h(G_2)=0$. Prin urmare:

$f(G_2) = h(G_2)$ (4).

Din (3) si (4) aratam ca: $f^* >= g(G_2)$.

Aceasta concluzie contrazice faptul ca G_2 este suboptimal. Ea arata ca A* nu alege niciodata spre extindere un scop suboptimal. A* intoarce o solutie numai dupa ce a selectat-o spre extindere, de aici rezulta optimalitatea algoritmului A*.

6. Implementarea in Prolog a cautarii de tip best-first

Vom imagina cautarea de tip best-first functionand in felul urmator: cautarea consta intr-un numar de subprocese concurente, fiecare explorand propriul subarbore. Subarborii au subarbori samd. Dintre toate aceste subprocese, doar unul este activ la un moment dat si anume cel care se ocupa de alternativa cea mai promitatoare (corespunzatoare celei mai mici f^-valori). Celelalte procese asteapta pana f^-valorile se schimba si o alta alternatica devine promitatoare, respectiv ia locul procesului activ.

In vederea implementarii in Prolog, vom extinde definitia lui f^ de la noduri la arbori:

- pentru un arbore cu un singur nod N, avem egalitate intre f^-valoarea sa si f^(N).
- pentru un arbore T cu radacina N si subarborii S_1, S_2, ... definim: $f^(T) = \min(\text{dupa } i)\ f^(S_i)$.

Vom reprezenta arborii de cautare in doua moduri:

- l(N,F/G) – corespunde unui arbore cu un singur nod N; G = g^(N) (costul drumului intre nodul de start si N) si F = G + h^(N).

- t(N,F/G,Subarb) – corespunde unui arbore cu subarbori nevizi; N este radacina, Subarb e lista subarborilor (ordonata crescator conform f^-valorii), G = g^(N), F este f^-valoarea actualizata a lui N, adica valoarea celui mai promitator succesor al lui N.

Recalculearea f^-valorilor e necesara pentru a permite programului sa recunoasca cel mai promitatoar subarbore, la fiecare nivel de cautare.

Cod:

```
%Predicatul bestfirst(Nod_initial,Solutie) este adevarat daca Solutie este un
%drum de la nodul Nod_initial la o stare scop.

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


%expandeaza(+Drum,+Arbore,+Limita,-A1,-Rezultat,-Solutie)
expandeaza(Drum,l(N,_),_,_, da,[N|Drum]):-scop(N).

%Caz 1: Daca N este nod scop, construim o cale solutie

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

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

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

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

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

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

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

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

```
%continua(+Drum,+Arb,+Limita,-Arb1,-Rez1,-Rez,-Sol)
continua(_,_,_,_,da,da,Sol).
continua(P,t(N,F/G,[A1|As]),Limita,Arb1,nu,Rez,Sol):-
        insereaza(A1,As,NAs),
        cea_mai_buna_f(NAs,F1),
        expandeaza(P,t(N,F1/G,NAs),Limita,Arb1,Rez,Sol).
continua(P,t(N,F/G,[_|As]),Limita,Arb1,imposibil,Rez,Sol):-
        cea_mai_buna_f(As,F1),
        expandeaza(P,t(N,F1/G,As),Limita,Arb1,Rez,Sol).

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

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

```
%insereaza(+Arb,+ListArb, -ListArbrez)
insereaza(A,As,[A|As]):-
        f(A,F),
        cea_mai_buna_f(As,F1),
        F=<F1,!.
insereaza(A,[A1|As],[A1|As1]):-insereaza(A,As,As1).

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

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

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

```
%cea_mai_buna_f(+Lista,-F)
cea_mai_buna_f([A|_],F):-f(A,F).
cea_mai_buna_f([],999999).
```

%In cazul unei liste de arbori vide, f^-valoarea este foarte mare

```
%membru(+Element,+Lista)
membru(H,[H|_]).
membru(X,[_|T]):-membru(X,T).
```

`expandeaza(+Drum,+Arbore,+Limita,-A1,-Rezultat,-Solutie)`:- predicat cheie; Drum reprezinta calea intre nodul de start al cautarii si Arbore; Arbore este arborele(subarborele) curent de cautare; Limita este f^-limita pentru expandarea lui Arbore, Rezultat este un indicator

si va lua una dintre valorile „da”, „nu”, „imposibil”; Solutie este o cale de la nodul de start (prin A1) catre un nod scop (in limita Limita), daca un astfel de nod scop exista. Exista trei posibiliati:

- Rezultat = da – Solutie va unifica cu o cale gasita expandand Arbore in limita Limita; A1 va ramane neinstantiat;
- Rezultat = nu – A1 va fi, de fapt, Arbore pana cand f^-valoarea sa depaseste Limita; Solutie ramane neinstantiat;
- Rezultat = imposibil – caz in care subarborele A1 si Solutie raman neinstantiate; acest caz indica explorarea lui Arbore ca pe o alternatica moarta; acest caz apare cand f^-valoarea lui Arbore <= Limita, dar nu mai exista succesori sau creeaza un ciclu.

`s(+Nod,-Nod1,-Cost)`:- este adevarat daca exista un arc intre Nod si Nod1

`scop(Nod)`:- este adevarat daca Nod este stare scop

`h(Nod,H)`:- H este o estimatie euristica a costului celui mai ieftin drum intre Nod si o stare scop.

7. Algoritmul Minimax. Textul algoritmului si implementarea in Prolog

Tipul de jocuri la care ne vom referi in continuare este acela al jocurilor de doua persoane cu informatie perfecta sau completa. Vom lua in considerare cazul general al unui joc cu doi jucatori, pe care ii vom numi MAX si MIN. MAX face prima mutare, dupa care cei doi vor efectua mutari pe rand pana la castigarea jocului.

Un joc poate fi definit ca fiind un anumit tip de problema de cautare avand urmatoarele componente:

- starea initiala – include pozitia pe tabla de joc, indicatie referitoare la primul care muta;
- o multime de operatori – definesc mutarile legale ale unui jucator
- un test terminal – determina sfarsitul de joc
- o functie de utilitate (de plata) – acorda o valoare numerica rezultatului unui joc (-1,0,1).

Astfel, actiunea lui MAX consta in cautarea unei secvente de mutari care duc la o stare terminala reprezentand o stare de castig.

In cadul algoritmului Minimax numim in continuare oponentii MIN si MAX. MAX reprezinta jucatorul care incearca sa castige sau sa isi maximizeze avantajul, iar MIN este oponentul care incearca sa minimizeze scorul lui MAX.

Algoritmul Minimax:

1. Genereaza intreg arborele de joc, pana la starile terminale.

2. Aplica functia de utilitate fiecarei stari terminale pentru a obtine valoarea corespunzatoare starii.
3. Deplaseaza-te inapoi in arbore, de la nodurile-frunza(valorile statice) spre nodul radacina, determinand corespunzator fiecarui nivel din arbore, valorile care reprezinta utilitatea nodurilor aflate la acel nivel. Propagarea acestor valori la niveluri anterioare se face prin intermediul nodurilor parinte succesive, conform regulilor:
    a. Daca starea parinte este MAX, atribuie-i maximul dintre valorile fiilor
    b. Daca starea parinte este MIN, atribuie-i minimul dintre valorile fiilor
4. Ajuns in nodul radacina, alege pentru MAX mutarea care conduce la valoarea maxima (decizia minimax).

Implementare in Prolog:

```
%minimax(+Poz, ? SuccBun, -Val)
minimax(Poz,SuccBun,Val):-
        %mutarile legale de la Poz produc ListaPoz
        mutari(Poz,ListaPoz),!,
        celmaibun(ListaPoz,SuccBun,Val);
        %Poz nu are succesori si este evaluat in mod static
        staticval(Poz,Val).

%celmaibun(+ListaPoz, -Succesor, -Valoare)
celmaibun([Poz],Poz,Val):-
        minimax(Poz,_,Val),!.

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

%Poz0 mai bun decat Poz21

%maibine(+Poz1,+Val1, +Poz2, +Val2, -PozRez, -ValRez )
maibine(Poz0,Val0,Poz1,Val1,Poz0,Val0):-
        %Min face o mutare la Poz0
        %Max prefera valoarea maxima
        mutare_min(Poz0),
        Val0>Val1,!
        ;
        %Max face mutare la Poz0
        %Min prefera valoarea mai mica
        mutare_max(Poz0),
        Val0<Val1,!.
%Altfel, Poz1 mai buna decat Poz0
maibine(Poz0,Val0,Poz1,Val1,Poz1,Val1).
```

`minimax(Poz,SuccBun,Val)`:- Val este valoarea minimax a unei Poz, iar SuccBun este cea mai buna pozitie succesor a lui Poz (mutarea care trebuie facuta pentru a obtine Val) ; cea mai buna mutare de la Poz conduce la pozitia SuccBun

`mutari(Poz,ListaPoz)`:- corespunde mutarilor legale ale jocului; ListaPoz este lista pozitiilor succesor legale ale lui Poz; esueaza daca Poz e o frunza

`celmaibun(ListaPoz,PozBun,ValBuna)`:- selecteaza cea mai buna pozitie PozBun dintr-o lista de pozitii candidate ListaPoz; ValBuna e valoarea lui PozBun

8. Algoritmul Alpha-Beta. Prezentare generala (cu un exemplu) si implementare in Prolog

Prezentare generala:

Ideea tehnicii de alpha-beta retezare este aceea de a gasi o mutare „suficient de buna", nu neaparat cea mai buna, dar suficient de buna pentru a lua decizia corecta. Practic, algoritmul alpha-beta este o versiune mai rapida a algoritmului Minimax, intrucat realizeaza o retzare a unor ramuri ale arborelui care nu pot influenta decizia finala. Aceasta idee poate fi formalizata prin introducerea a doua limite, apha si beta, repezentand limitari ale valorii de tip minimax corespunzatoare unui nod intern. Semnificatia acestor limite este:

- alpha – valoarea minima pe care este deja garantat ca o va obtine MAX
- beta – valoarea maxima pe care spera MAX sa o obtina

Pentru MIN, beta este valoarea cea mai nefavorabila pe care acesta o va atinge. Prin urmare, valoarea efectiva se va afla intre alpha si beta. De asemenea, valoarea beta asociata unui nod nu poate niciodata sa creasca, iar valoarea alpha asociata unui nod nu poate niciodata sa descreasca.

Cele doua reguli pentru incheierea cautarii, bazate pe alpha si beta:

- cautarea poate fi oprita dedesubtul oricarui nod de tip MIN care are o valoare beta mai mica sau egala cu valoarea alpha a oricaruia dintre stramosii sai de tip MAX
- cautarea poate fi oprita dedesubtul oricarui nod de tip MAX care are o valoarea alpha mai mare sau egala cu valoarea beta a oricaruia dintre stramosii de tip MIN.

Dpdv formal, putem defini o valoare de tip minimax a unui nod intern P, V(P, alpha, beta), ca fiind „suficient de buna" daca satisface urmatoarele cerinte:

- V(P, alpha, beta) < alpha, daca V(P) < alpha
- V(P, alpha, beta) = V(P), daca alpha <= V(P) <= beta
- V(P, alpha, beta) > beta, daca V(P) > beta, unde V(P) este valoarea nodului de tip minimax.

Implementare in Prolog:

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

%limitarebuna(+ListaPoz,+Alpha,+Beta,-PozBuna,-ValBuna)
limitarebuna([Poz|ListaPoz],Alpha,Beta,PozBuna,ValBuna):-
        alphabeta(Poz,Alpha,Beta,_,Val),
        destuldebun(ListaPoz,Alpha,Beta,Poz,Val,PozBuna,ValBuna).
%nu exista alt candidat
%destuldebun(+ListaPoz,+Alpha,+Beta,+Poz,+Val,-PozBuna,-ValBuna)
destuldebun([ ],_,_,Poz,Val,Poz,Val):-!.

destuldebun(_,Alpha,Beta,Poz,Val,Poz,Val):-
        %atingere limita superioara
        mutare_min(Poz),Val > Beta,!;
        %atinge limita inferioara
        mutare_max(Poz),Val < Alpha,!.

destuldebun(ListaPoz,Alpha,Beta,Poz,Val,PozBuna,ValBuna):-
        %rafinare limite
        limitenoi(Alpha,Beta,Poz,Val,AlphaNou,BetaNou),
        limitarebuna(ListaPoz,AlphaNou,BetaNou,Poz1,Val1),
```

```
        maibine(Poz,Val,Poz1,Val1,PozBuna,ValBuna).

%limitenoi(+Alpha,+Beta,+Poz,+Val,-AlphaNou,-BetaNou),
limitenoi(Alpha,Beta,Poz,Val,Val,Beta):-
        %crestere limita inferioara
        mutare_min(Poz),Val > Alpha,!.

limitenoi(Alpha,Beta,Poz,Val,Alpha,Val):-
        %descrestere limita superioara
        mutare_max(Poz),Val < Beta,!.
%altfel limitele nu se schimba
limitenoi(Alpha,Beta,_,_,Alpha,Beta).

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

`alphabeta(Poz,Alpha,Beta,PozBuna,Val)`:- PozBuna reprezinta un succesor „suficient de bun” al lui Poz, astfel incat Val satisface cerintele mai sus mentionate

`limitarebuna(ListaPoz,Alpha,Beta,PozBuna,Val)`:- gaseste in ListaPoz o pozitie suficient de buna, PozBuna, astfel incat vaoarea de tip minimax, Val, a lui PozBuna sa reprezinte o aproximatie suficient de buna relativ la Alpha, Beta.

`limitenoi(Alpha,Beta,Poz,Val,AlphaNou,BetaNou)`:- defineste noul interval [AlphaNou,BetaNou], care este intotdeauna mai ingust sau cel putin egal cu [Alpha,Beta]

9. Reprezentarea cunostintelor cu reguli if-then. Interpretor pentru reguli in cazul inlantuirii inapoi (implementare in Prolog)

Reprezentarea cunostintelor cu reguli if-then:

Regulile if-then, numite si reguli de productie, constituie o forma naturala de exprimare a cunostintelor si au urmatoarele caracteristici suplimentare:

- Modularitate – fiecare regula defineste o cantitate de cunostinte relativ mica si independenta de celelalte
- Incrementabilitate – noi reguli pot fi adaugate bazei de cunostinte in mod relativ independent de celelalte

- Modificabilitate (consecinta a modularitatii) – regulile vechi pot fi modificate relativ independent de celelalte reguli
- Sustin transparenta sistemului – abilitatea sistemului de a explica deciziile si solutiile sale.

Regulile de productie faciliteaza generarea raspunsului pentru tipurile de intrebari:

- intrebare de tipul „Cum”: Cum ai ajuns la aceasta concluzie?
- intrebare de tipul „De ce”: De ce te intereseaza aceasta informatie?

Regulile de tip if-then adesea definesc relatii logice intre conceptele apartinand domeniului problemei. Relatiile pur logice pot fi caracterizate apartinand asa numitelor cunostinte categorice (cunostinte intotdeauna adevarate). In unele domenii insa, cum ar fi diagnosticarea in medicina, predomina cunostintele „moi” sau probabiliste (cunostintele empirice sunt valide numai pana intr-un anumit punct).

In astfel de cazuri, regulile pot fi modificate prin adaugarea la interpretarea lor logica a unei calificari de verosimilitate:

If conditie A then concluzie B cu certitudinea F

In cazul regulilor if-then exista doua modalitati de a rationa:

- inlantuire inapoi (backward chaining) – orientata catre scop (scop -> date)
- inlantuire inainte (forward chaining) – orientata catre date (date -> scop)

Interpretor pentru reguli in cazul inlantuirii inapoi (iP):

Pleaca de la o ipoteza si apoi parcurge in sensul inapoi reteaua de inferenta.

Principalul dezavantaj al acestei proceduri de inferenta consta in faptul ca utilizatorul trebuie sa enunte toate informatiile relevante de la inceput, sub forma de fapte.

Interpretare Prolog:

```
:- op(800,fx,if).
:- op(700,xfx,then).
:- op(300,xfy,or).
:- op(200,xfy,and).

este_adevarat(P):-
        fapta(P).

este_adevarat(P):-
        if Conditie then P,
        este_adevarat(Conditie).

este_adevarat(P1 and P2):-
        este_adevarat(P1),
        este_adevarat(P2).

este_adevarat(P1 or P2):-
        este_adevarat(P1) ;
        este_adevarat(P2).
```

`fapta(P):-` predicatul pentru enuntarea faptelor observate

`este_adevarat(P):-` interpretor pentru reguli, unde P este fie dat prin predicatul fapta(P), fie derivat

10. Reprezentarea cunostintelor cu reguli if-then. Interpretor pentru reguli in cazul inlantuirii inainte (implementare in Prolog)

Reprezentarea cunostintelor cu reguli if-then:

Regulile if-then, numite si reguli de productie, constituie o forma naturala de exprimare a cunostintelor si au urmatoarele caracteristici suplimentare:

- Modularitate – fiecare regula defineste o cantitate de cunostinte relativ mica si independenta de celelalte
- Incrementabilitate – noi reguli pot fi adaugate bazei de cunostinte in mod relativ independent de celelalte
- Modificabilitate (consecinta a modularitatii) – regulile vechi pot fi modificate relativ independent de celelalte reguli
- Sustin transparenta sistemului – abilitatea sistemului de a explica deciziile si solutiile sale.

Regulile de productie faciliteaza generarea raspunsului pentru tipurile de intrebari:

- intrebare de tipul „Cum”: Cum ai ajuns la aceasta concluzie?
- intrebare de tipul „De ce”: De ce te intereseaza aceasta informatie?

Regulile de tip if-then adesea definesc relatii logice intre conceptele apartinand domeniului problemei. Relatiile pur logice pot fi caracterizate apartinand asa numitelor cunostinte categorice (cunostinte intotdeauna adevarate). In unele domenii insa, cum ar fi diagnosticarea in medicina, predomina cunostintele „moi” sau probabiliste (cunostintele empirice sunt valide numai pana intr-un anumit punct).

In astfel de cazuri, regulile pot fi modificate prin adaugarea la interpretarea lor logica a unei calificari de verosimilitate:

If conditie A then concluzie B cu certitudinea F

In cazul regulilor if-then exista doua modalitati de a rationa:

- inlantuire inapoi (backward chaining)
- inlantuire inainte (forward chaining)

Interpretor pentru reguli in cazul inlantuirii inainte (implementare in Prolog):

Inlantuirea inainte nu incepe cu o ipoteza, ci face un rationament in directie opusa, de la partea cu if la partea cu then. Interpretorul presupune ca regulile sunt ca si inainte de forma:

if Conditie then Concluzie,

unde Concluzie poate fi o expresie de forma and/or. Interpretorul incepe cu ceea ce este cunoscut deja (relatia fapta), trage toate concluziile posibile si adauga aceste concluzii (folosind assert) relatiei fapta.

Cod:

```
:- op(800,fx,if).
:- op(700,xfx,then).
:- op(300,xfy,or).
:- op(200,xfy,and).

inainte:-
        fapta_noua_dedusa(P),              %o noua fapta
        !,
        write('Dedus:'), write(P), nl,
        assert(fapta(P)),
        inainte                            %continua
        ;
        write('Nu mai exista fapte!').     %toate faptele au fost deduse
```

```
fapta_noua_dedusa(Concl):-
        if Cond then Concl,             %o regula
        not fapta(Concl),               %concluzia regulii nu este o fapta inca
        fapta_compusa(Cond).            %conditia este adevarata?

fapta_compusa(Cond):-                   %fapta simpla
        fapta(Cond).
fapta_compusa(Cond1 and Cond2):-        %ambii sunt adevarati
        fapta_compusa(Cond1),
        fapta_compusa(Cond2).
fapta_compusa(Cond1 or Cond2):-         %doar unul dintre ei poate fi adevarat
        fapta_compusa(Cond1);
        fapta_compusa(Cond2).
```

11. Generarea explicatiilor si introducerea incertitudinii in sistemele expert. Implementare in Prolog

Generarea explicatiilor:

Una dintre caracteristicile regulilor de productie este sustinerea transparentei sistemului. Prin aceasta intelegem abilitatea sistemului de a explica deciziile si solutiile sale. Regulile de productie faciliteaza generarea raspunsului pentru urmatoarele intrebari:

- de tipul ”Cum”: Cum ai ajuns la aceasta concluzie?
- de tipul ”De ce”: De ce te intereseaza aceasta informatie?

In cazul intrebarilor de tipul ”Cum”, explicatia pe care programul o furnizeaza cu privire la modul in care a fost dedus raspunsul contituie un arbore de demonstratie a modului in care concluzia finala decurge din regulile si faptele aflate in baza de cunostinte.

Fie ”<=” un operator infixat. Atunci arborele de demonstratie poate fi reprezentat prin una din formele (in functie de necesitati):

- Daca P este o fapta, atunci arborele de demonstratie este P
- Daca P a fost dedus folosind o regula de forma if Cond then P, atunci arborele de demonstratie este P <= DemCond, unde DemCond este arborele de demonstratie a lui Cond
- Fie P1 si P2 propozitii ale caror arbori de demonstratie sunt Dem1 si Dem2. Daca P este de forma P1 and P2, atunci arborele de demonstratie corespunzator este Dem1 and Dem2. Daca P este de forma P1 or P2, atunci arborele de demonstratie este fie Dem1, fie Dem2.

Implementare in prolog:

```
:- op(800,fx,if).
:- op(700,xfx,then).
:- op(300,xfy,or).
:- op(200,xfy,and).
:- op(800,xfx,<=).
```

```
este_adevarat(P,P):-
        fapta(P).

este_adevarat(P, P <= DemCond):-
        if Conditie then P,
        este_adevarat(Conditie,DemCond).

este_adevarat(P1 and P2, Dem1 and Dem2):-
        este_adevarat(P1,Dem1),
        este_adevarat(P2,Dem2).

este_adevarat(P1 or P2,Dem):-
        este_adevarat(P1,Dem) ;
        este_adevarat(P2,Dem).
```

Introducerea incertitudinii:

Reprezentarea cunostintelor considerata pana acum pleaca de la presupunerea ca domeniile sunt categorice (raspunsul la orice intrebare e fie adevarat, fie fals, de unde si implicatiile categorice). Totusi, majoritatea domeniilor expert nu sunt categorice. Incertitudinea poate fi modelata prin atribuirea unei calificari, alta decat adevarat sau fals, majoritatii asertiunilor. Gradul de adevar poate fi exprimat prin intermediul unui numar real aflat intr-un anumit interval. Astfel de numere cunosc o varietate de denumiri precum factor de certitudine, masura a increderii sau certitudine subiectiva.

Fiecarei propozitii ii vom adauga un factor de certitudine:

Propozitie: FactorCertitudine

Vom defini o regula impreuna cu gradul de certitudine pana la care acea regula este valida:

if Conditie then Concluzie: Certitudine

Modul in care se combina certitudinile propozitiilor si ale regulilor: fie P1 si P2, propozitii, avand certitudinile c(P1) si c(P2). Definim:

- c(P1 and P2) = min(c(P1),c(P2))
- c(P1 or P2) = max(c(P1),c(P2))

Daca exista regula

if P1 then P2:C,

cu C reprezentand factorul de certitudine, atunci

c(P2) = c(P1)*C

Implementare in Prolog:

Se presupune ca estimatiile de certitudine corespunzatoare datelor observate sunt specificate de catre utilizator dat(P,Certitudine):

```
:- op(800,fx,if).
:- op(700,xfx,then).
:- op(300,xfy,or).
:- op(200,xfy,and).

%certitudine (Propozitie, Certitudine)
certitudine(P,Cert):-
        dat(P,Cert).
certitudine(Cond1 and Cond2, Cert):-
        certitudine(Cond1,Cert1),
        certitudine(Cond2,Cert2),
        minimum(Cert1,Cert2,Cert).
```

```
certitudine(Cond1 or Cond2, Cert):-
        certitudine(Cond1,Cert1),
        certitudine(Cond2,Cert2),
        maximum(Cert1,Cert2,Cert).
certitudine(P,Cert):-
        if Cond then P:C1,
        certitudine(Cond,C2),
        Cert is C1*C2.
```