CAPITOLUL V

ALGORITMI DE CĂUTARE PENTRU AGENŢI

În acest capitol vor fi prezentaţi câţiva algoritmi de căutare care sunt utili în rezolvarea de probleme care implică unul sau mai mulţi agenţi. Termenul de căutare este prezent în mai multe tehnici de rezolvare a problemelor de IA. În probleme de căutare, secvenţa acţiunilor necesare pentru rezolvarea unei probleme nu poate fi cunoscută dinainte, ea poate doar să fie determinată printr–un proces de explorare a alternativelor şi eliminare a celor care nu satisfac cerinţele impuse de problemă. Din moment ce aproape toate probleme de IA necesită în rezolvarea lor o metodă de căutare, căutarea are un istoric lung în acest domeniu. Problemele de căutare sunt împărţite în trei mari categorii:

- **Probleme de căutare a unui drum** (**Path Finding Problems - PFP**)

  În acest tip de probleme obiectivul este transformarea unei configuraţii iniţiale date într–o configuraţie finală prin executarea de mutări permise. O astfel de problemă a primit denumirea de problemă de căutare a unui drum, deoarece rezolvarea unei asemenea probleme duce la determinare a unui drum (secvenţă de mutări), de la configuraţia iniţială la cea finală. Un exemplu ce descrie foarte bine tipul acesta de probleme este problema ***n-puzzle*** (secţiunea 5.2).

- **Probleme de satisfacere a constrângerilor (Constraint Satisfaction Problems - CSP)**

  O problemă de satisfacere a constrângerilor presupune găsirea unei configuraţii finale şi nu a unui drum către aceasta. Problemele de acest tip au primit denumirea de probleme de satisfacere a constrângerilor, deoarece acestea au ca scop determinarea unei configuraţii finale care îndeplineşte anumite condiţii (satisface anumite constrângeri). Un exemplu tipic este ***problema celor 8 dame***.

- **Jocuri cu doi jucători**

  Din moment ce din această categorie fac parte probleme în care apar doi agenţi competitivi rezolvarea acestora are o strânsă legătura cu IAD. În acestea scopul fiecărui agent este să câştige jocul. Un exemplu foarte cunoscut de astfel de problemă (joc) este ***şahul***.

Cei mai mulţi algoritmi din primele două categorii au fost iniţial proiectaţi doar pentru probleme rezolvate de un singur agent. Se pune întrebarea: *„Care dintre aceştia pot fi utilizaţi în probleme rezolvate de mai mulţi agenţi care cooperează?”*. Această întrebare se datorează faptului că, în general, se consideră că un agent are o capacitate limitată de a raţiona. Mai precis, capacitatea unui agent de a face calcule sau de a recunoaşte este, de obicei, limitată. Pentru un agent poate fi imposibil să aibă o imagine completă a unei probleme. Chiar dacă are o imagine completă a problemei un agent se poate lovi de probleme care ţin de volumul imens de calcule care sunt necesare, fapt care poate face o astfel de rezolvare o mare consumatoare de resurse sau chiar poate depăşi capacităţile agentului. De aceea, un agent va face doar un număr mic de calcule folosind doar informaţii locale şi apoi va lua decizia potrivită, în baza resurselor disponibile.

În cei mai mulţi algoritmi standard, fiecare pas este executat secvenţial, şi pentru fiecare pas este necesară toată informaţia existentă la nivel global. Pe de altă parte, o problemă de căutare o putem reprezenta cu ajutorul unui graf şi există algoritmi de căutare cu ajutorul cărora o problemă este rezolvată prin efectuarea de calcule locale

pentru fiecare nod al grafului. Ordinea acestor calcule este aleatoare sau are un grad mare de flexibilitate şi pot fi executate asincron şi concurent. Numim aceşti algoritmi *algoritmi asincroni de căutare*.

Atunci când o problemă este rezolvată utilizând mai mulţi agenţi cu capacităţi limitate, algoritmii asincroni de căutare sunt potriviţi din următoarele motive:

- Putem presupune că agenţii au capacităţi suficiente pentru a putea efectua calculele locale şi pentru a face faţă informaţiei necesare pentru efectuarea acestora.
- Dacă mai mulţi agenţi cooperează în rezolvarea unei probleme utilizând algoritmi asincroni de căutare, atunci ordinea de execuţie a acestora (fiecare agent are propriul fir de execuţie), poate fi aleatoare sau foarte flexibilă. Acest lucru este foarte important pentru că, în cazul în care aceştia ar trebui sincronizaţi, cantitatea de resurse necesare pentru a obţine un astfel de control poate fi foarte mare.

Importanţa rezolvării unei astfel de probleme prin combinarea unor calcule locale asincrone a fost pentru prima dată subliniată de Lesser, iar acestă idee a fost recunoscută pe scară largă în studii privind IAD.

5.1 Probleme de satisfacere a constrângerilor (Constraint Satisfaction Problems - CSP)

5.1.1 CSP în Inteligenţa Artificială tradiţională

Raţionamentul bazat pe constrângeri este o paradigmă foarte simplă şi puternică, cu ajutorul căreia pot fi formulate şi rezolvate multe probleme interesante.

O constrângere este o relaţie logică între mai multe variabile, fiecare luând valori din câte un domeniu precizat. Astfel, o constrângere indică restricţii asupra valorilor posibile pe care o variabilă le poate lua, datorită condiţionărilor dintre aceasta şi alte variabile. Constrângerile au o serie de proprietăţi remarcabile:

- constrângerile pot specifica informaţie parţială;
- constrângerile sunt non-direcţionale;
- constrângerile sunt declarative;
- constrângerile sunt aditive;
- constrângerile sunt rareori independente.

Programarea prin constrângeri reprezintă studiul sistemelor computaţionale bazate pe constrângeri. Ideea este de a rezolva probleme prin specificarea condiţiilor care trebuie satisfăcute de soluţie. Această abordare este, astfel, înrudită cu paradigma programării declarative, unde programul reprezintă o descriere a problemei, şi nu o reţetă de rezolvare a acesteia.

Problemele de **satisfacere a constrângerilor** (CSP) presupun:

- un set finit de variabile, fiecare având câte un domeniu finit de valori;
- o mulţime finită de constrângeri, fiecare introducând restricţii asupra valorilor pe care anumite variabile le pot lua simultan.

O soluţie a CSP reprezintă atribuirea câte unei valori fiecărei variabile, în aşa fel încât toate constrângerile existente să fie satisfăcute. În funcţie de problema de rezolvat, trebuie identificată fie o soluţie oarecare, fie o soluţie anume, fie toate soluţiile.

Spre deosebire de problema satisfacerii constrângerilor, problema **rezolvării constrângerilor** operează cu variabile având domenii infinite, constrângerile putând fi descrise prin ecuaţii matematice complicate.

Aplicaţiile practice ale problematicii satisfacerii constrângerilor sunt nelimitate, mergând de la probleme de puzzle simple (problema damelor, criptaritmetică) şi până la controlul traficului aerian, de metrou şi căi ferate.

5.1.1.1 Definiţia unei probleme de CSP

O **problemă de satisfacere a constrângerilor** (CSP) constă din:

- mulţime finită de *variabile* $X = \{x_1, x_2, ..., x_n\}$;
- pentru fiecare variabilă x_i o mulţime finită D_i de valori posibile (*domeniul* acesteia);
- mulţime finită de *constrângeri*, fiecare introducând restricţii asupra valorilor pe care anumite variabile le pot lua simultan

O **soluţie a unei CSP** este o atribuire câte unei valori fiecărei variabile (din cadrul domeniului său) astfel încât toate constrângerile existente să fie satisfăcute. Scopul este obţinerea:

- unei soluţii oarecare, fără o preferinţă anume;
- tuturor soluţiilor,
- unei soluţii optimale, sau cel puţin bune, fiind dată o funcţie criteriu definită pe (unele dintre) variabile.

Soluţiile unei CSP se pot obţine prin căutare sistematică prin toate valorile posibile. Metodele de căutare fie traversează spaţiul soluţiilor parţiale, fie explorează spaţiul atribuirilor complete de valori.

Majoritatea algoritmilor de satisfacere a constrângerilor se referă la probleme în care fiecare constrângere este fie unară, fie binară. Astfel de probleme sunt denumite **CSP binare**. O CSP binară poate fi reprezentată printr-un graf (sau reţea) de constrângeri, unde nodurile reprezintă variabile, iar arcurile reprezintă constrângeri între variabilele reprezentate de nodurile extreme. Acest graf se va numi **graf de consistenţă**. Deoarece constrângerile unare reprezintă condiţii puse asupra unei variabile independent celelalte, constrângerile unare se pot elimina prin reducerea domeniului variabilelor în cauză. Această metodă se numeşte *consistenţa nodurilor* (*node consistency*). Prin urmare, pentru simplificarea problemelor, constrângerile unare nu mai sunt reprezentate în cadrul grafelor de constrângeri.

Orice constrângeri *n*-are pot fi reduse la constrângeri binare. Pentru a ilustra aceast lucru, se introduce o variabilă suplimentară *u* care encapsulează cele *n* variabile originale, domeniul său fiind produsul cartezian al domeniilor variabilelor originale. O constrângere *n*-ară asupra variabilelor originale va fi înlocuită cu o constrângere unară asupra variabilei nou introduse. Practic, pentru fiecare constrângere *n*-ară se introduce câte o variabilă suplimentară cu domeniul modificat prin reducerea constrângerii *n*-are originale.

Rămâne ca variabilele suplimentare introduse să fie legate de variabilele originale. Există două modalităţi de a realiza acest lucru: cu păstrarea variabilelor originale, şi cu eliminarea variabilelor originale.

Dacă se doreşte păstrarea variabilelor originale, pentru fiecare variabilă originală x_i şi pentru fiecare variabilă suplimentară *u,* se introduce constrângerea $x_i=arg_i(u)$ (x_i este a *i*-a componentă a lui *u*). Dacă se doreşte eliminarea variabilelor originale, pentru oricare variabile suplimentare *u* şi *v*, se introduc constrângerile $arg_i(u)=arg_i(v)$ pentru fiecare valoare a lui *i*.

Prin urmare orice problemă de satisfacere a constrângerilor se poate reprezenta printr-o problemă binară echivalentă. Din acest motiv, faptul că întreaga literatură se concentrează asupra problemelor binare nu reprezintă o pierdere a generalităţii.

Algoritmii de căutare sistematică garantează găsirea unei soluţii, dacă aceasta există. Pe de altă parte, au dezavantajul unui consum de timp foarte ridicat. Cei mai simpli **algoritmi de căutare sistematică** pentru rezolvarea problemelor de satisfacere a constrângerilor sunt, aşa cum ne aşteptăm, Generează-şi-Testează şi Backtracking.

Algoritmul Backtracking are trei dezavantaje majore: (a) eşecul repetat datorită aceluiaşi motiv (întotdeauna se revine, necondiţionat, la ultima variabilă); (b) realizarea de efort redundant (un conflict, rămas valabil, nu se detectează în căutările viitoare); (c) detectarea conflictelor prea târziu (conflictul nu este detectat înainte să apară); (d) este exponenţial ca timp de execuţie.

Dezavantajul (a) este cunoscut sub numele de *trashing* şi este considerat dezavantajul major al paradigmei backtracking. Fără a intra în amănunte, vom spune doar că *trashing-ul* este cauzat de două aspecte: inconsistenţa nodurilor şi inconsistenţa arcelor grafului de consistenţă G al problemei CSP.

Al doilea mare dezavantaj al backtrackingului este complexitatea sa computaţională, care îl face impractic pentru probleme de dimensiuni mari (număr mare de variabile a problemei).

În paragraful următor vom da câteva tehnici de asigurare a consistenţei grafului G, tehnici care aplicate în rezolvarea problemei de CSP înaintea unei căutări de tip backtracking eficientizează căutarea, reducând *trashing-ul*.

5.1.1.2 Probleme supra-constrânse

Existenţa soluţiei unei probleme de satisfacerea constrângerilor presupune posibilitatea satisfacerii simultane a TUTUROR constrângerilor. În multe aplicaţii reale, acest lucru nu este posibil. Astfel de sisteme se numesc **supra-constrânse** (*over-constrained*). Pentru abordarea acestui tip de aplicaţii, problema satisfacerii constrângerilor a fost modificată în aşa fel încât generarea unei "soluţii" să fie, totuşi, posibilă. Dintre multiplele abordări, discutăm aici următoarele:

- CSP fuzzy;
- CSP probabiliste;
- CSP ponderate;
- ierarhii de constrângeri.

O **problemă de satisfacere a constrângerilor fuzzy** constă dintr-un set de variabile, fiecare cu domeniul ei şi dintr-o mulţime de constrângeri, care sunt constrângeri fuzzy: mulţimi fuzzy definite peste mulţimea tuturor combinaţiilor posibile de valori din domeniile variabilelor participante. Astfel, pentru constrângerea fuzzy *c*, vom nota cu *c*(*A*) gradul de apartenenţă a atriburii multiple *A* la *c*, adică măsura în care atribuirea A este bună (valoarea 0 însemnând constrângere clar încălcată şi atribuire interzisă, iar valoarea 1 însemnând constrângere clar respectată şi atribuire permisă). O

soluţie a unei astfel de probleme este un set de valori, câte una din domeniul fiecărei variabile, astfel încât expresia min{c(A) | c este o constrângere fuzzy} este maximizată peste toate atribuirile *A* posibile.

O **problemă de satisfacere a constrângerilor probabiliste** constă dintr-un set de variabile, fiecare cu domeniul ei şi dintr-o mulţime de constrângeri, fiecare constrângere *c* având o anumită probabilitate, $p(c)$, independentă de probabilităţile celorlate constrângeri, de a fi parte a problemei date. Definiţia soluţiei unei CSP probabiliste permite două abordări diferite. Astfel, soluţia este un set de valori, câte una din domeniul fiecărei variabile, astfel încât: (a) suma{produsul{$p(c)$ | c este constrângere din S} | S este o submulţime a mulţimii tuturor constrângerilor satisfăcute de A} este maximizată peste toate atribuirile *A* posibile, sau (b) produsul{1-$p(c)$ | c este o constrângere violată de A} este maximizată peste toate atribuirile *A* posibile.

O **problemă de satisfacere a constrângerilor ponderate** constă dintr-un set de variabile, fiecare cu domeniul ei şi dintr-o mulţime de constrângeri, fiecare constrângere *c* fiind definită cu ajutorul unei funcţii de cost, $w(c,A)$, care atribuie o valoare nenegativă fiecărui set de valori *A*. Acest cost este costul (sau penalitatea) asociat violării constrângerii. Scopul este de a obţine acea soluţie care conduce la o penalizare minimă. Astfel, soluţia este un set de valori, câte una din domeniul fiecărei variabile, astfel încât suma valorilor $w(c,A)$ pentru toate constrângerile *c* este minimizată peste toate atribuirile *A* posibile. Alternativ, se poate folosi produsul valorilor $w(c,A)$ pentru toate constrângerile *c* violate de atribuirea *A*.

Ierarhiile de constrângeri sunt mulţimi de constrângeri, fiecare dintre acestea etichetate cu un indice, număr natural, corespunzând importanţei acordate. Astfel, ierarhia de constrângeri H este definită ca reuniune a mulţimilor H_0, H_1, H_2, ..., unde H_0 este mulţimea constrângerilor absolut necesare, H_1 este mulţimea constrângerilor importante, dar de un grad de importanţă ceva mai mic decât constrângerile din H_0, H_2 este mulţimea constrângerilor de un grad de importanţă ceva mai mic decât constrângerile din H_1, etc. Pentru determinarea soluţiei problemei de satisfacere a ierarhiei de constrângeri, se determină mai întâi soluţiile problemei CSP folosind doar constrângerile din H_0. Dacă acestea există, se determină acelea dintre soluţiile de mai sus care verifică şi constrângerile din H_1. Se continuă în acelaşi fel până când ne aflăm în imposibilitatea de a obţine o soluţie validă. Dacă s-a întâmplat acest lucru, atunci soluţia problemei este cea obţinută la pasul anterior. Dacă nu, înseamnă că problema nu este over-constrained, şi am obţinut o soluţie a problemei CSP originale.

5.1.1.3 Tehnici de consistenţă

Spre deosebire de căutare, care este nedeterministă, tehnicile de consistenţă sunt deterministe. Scopul acestora este de a elimina cât de multe arcuri din graful de constrângeri, dovedite într-un fel sau altul ca fiind *inconsistente*. În final, algoritmii de căutare vor opera într-un spaţiu de stări mai redus. Tehnicile de consistenţă cele mai utilizate sunt consistenţa de nod (NC, *Node Consistency*), consistenţa de arc (AC, *Arc Consistency*, cu mai multe variante) şi consistenţa de drum (PC, *Path Consistency*).

Consistenţa unui nod este cea mai simplă tehnică de consistenţă şi se reduce la eliminarea din domeniul fiecărei variabile a tuturor valorilor care violează constrângerile unare pentru acea variabilă.

subalgoritmul *NodeConsistency*(*G*) **este**
{Date: G - graful de consistenţă al problemei CSP}
{Rezultate: G - graful de consistenţă al problemei CSP, în care domeniile variabilelor au fost modificate pentru a asigura consistenţa nodurilor}
pentru fiecare V ∈ noduri(*G*) **execută**
pentru fiecare X în domeniul D al lui V **execută**
dacă există o constrângere unară pe V inconsistentă cu X
atunci
@şterge X din D
sf-dacă
sf-pentru
sf-pentru
sf-*NodeConsistency*

În urma aplicării acestui subalgoritm, graful de constrângeri va avea toate nod*urile consistente*, şi astfel toate constrângerile unare pot fi eliminate.

Un arc (V_i, V_j) se numeşte *consistent* dacă pentru fiecare valoare *x* din domeniul lui V_i există o valoare *y* în domeniul lui V_j astfel încât atribuirile V_i=*x* şi V_j=*y* sunt permise de constrângerea binară dintre V_i şi V_j. De remarcat că dacă arcul (V_i, V_j) este consistent, nu este neapărat necesar ca arcul (V_j, V_i) să fie consistent.

Subalgoritmul următor asigură consistenţa arcului (V_i, V_j): şterge din domeniul variabilei V_i toate acele valori pentru care condiţia de consistenţă de mai sus nu este verificată.

subalgoritmul *REVISE*(Vi,Vj) **este**
{Date: Vi, Vj - două variabile ale problemei CSP}
{Rezultate: domeniile variabilelor se modifică pentru a asigura consistenţa arcului (Vi, Vj)}
şterge ←fals
pentru fiecare X din domeniul Di al variabilei Vi **execută**
dacă nu există Y în domeniul Dj al variabilei Vj astfel încât (X,Y) este consistent
atunci
@şterge X din Di
şterge ←adevărat
sf-dacă
sf-pentru
REVISE ←şterge
sf-*REVISE*

Pentru a realiza consistenţa întregului graf, subagoritmul de consistenţă a arcelor *REVISE* se aplică în mod repetat, până când toate arcele grafului devin consistente.

Prima variantă a algoritmului de asigurare a consistenţei grafului, **AC-1**, apelează *REVISE* în mod repetat, pentru domeniul D_j, ori de câte ori *REVISE* a redus domeniul D_i pentru orice arc (V_j, V_i) din graful de constrângeri.

subalgoritmul *AC-1*(*G*) **este**
{Date: G - graful de consistenţă al problemei CSP}
{Rezultate: G - graful de consistenţă al problemei CSP, în care domeniile variabilelor au fost modificate pentru a asigura consistenţa tuturor arcelor}
 $Q \leftarrow \{(Vi,Vj) \mid (Vi,Vj) \in \text{arcuri}(G), i \neq j\}$;
 repetă
  schimbă← fals
  pentru fiecare (Vi,Vj) din Q **execută**
   s*chimbă* ← REVISE(Vi,Vj) sau *schimbă*
  sf-pentru
 până_când *schimbă* = fals
sf-*AC-1*

Următoarea variantă, denumită **AC-3**, optimizează funcţionarea algoritmului AC-1. Astfel, nu este necesar ca procesul de re-revizuire să acopere întregul graf de constrângeri. Pe de o parte, singurele arcuri afectate de reducerea domeniului variabilei V_k sunt arcurile (V_i, V_k), iar pe de alta, dacă reducerea domeniului s-a întâmplat la analiza arcului (V_k, V_m), re-revizuirea arcului (V_m, V_k) nu este necesară.

subalgoritmul *AC-3*(*G*) **este**
{Date: G - graful de consistenţă al problemei CSP}
{Rezultate: G - graful de consistenţă al problemei CSP, în care domeniile variabilelor au fost modificate pentru a asigura consistenţa tuturor arcelor}
 $Q \leftarrow \{(Vi,Vj) \mid (Vi,Vj) \in \text{arcuri}(G), i \neq j\}$
 cât-timp $Q \neq \emptyset$ **execută**
  @selectează şi şterge un arc oarecare (Vk, Vm) din Q
  dacă REVISE(Vk,Vm) **atunci**
   $Q \leftarrow Q \cup \{(Vi, Vk) \in \text{arcuri}(G), i \neq k, i \neq m\}$
  sf-dacă
 sf-cât-timp
sf-*AC3*

Optimizări de natura celor de mai sus sunt posibile în continuare, astfel încât există şi algoritmii AC-4, AC-5, AC-6, AC-7, etc.

Consistenţa de nod şi cea de arc sunt cazuri particulare ale unei noţiuni mai generale, numită k-consistenţă. Un graf este **k-consistent** dacă este adevărat că, dacă alegem valori oarecare ale *k*-1 variabile oarecare, care satisfac toate constrângerile dintre acestea, şi dacă alegem oricare a *k*-a variabilă, atunci există o valoare a acestei variabile care satisface toate constrângerile dintre aceste *k* variabile. De asemenea, un graf este **puternic k-consistent** dacă este *i*-consistent pentru toate valorile *i* cel mult egale cu *k*.

Consistenţa de nod este echivalentă cu puternic 1-consistenţa, şi consistenţa de arc este echivalentă cu puternic 2-consistenţa. Puternic 3-consistenţa se mai numeşte şi **consistenţă de drum** (*path consistency*).

Menţionăm faptul că asigurarea consistenţei arcelor în graful G nu este suficientă pentru pentru a găsi o soluţie a problemei CSP, ca urmare nu elimină necesitatea aplicării unei căutări de tip backtracking.

În urma aplicării unuia din din subalgoritmii de asigurare a consistenţei grafului G (AC-1, AC-3,...) sunt posibile trei situaţii:

- dacă domeniul unei variabile devine vid, atunci problema se numeşte **super-constrânsă** şi nu are soluţie;
- dacă toate domeniile variabilelor conţin o singură valoare, atunci combinaţia acestor valori furnizează o soluţie a problemei;
- dacă domeniile variabilelor conţin valori multiple, atunci nu se poate spune exact dacă problema are sau nu soluţie, ca urmare este necesară aplicarea unei metode de căutare pentru a găsi soluţia.

Există totuşi un rezultat care asigură faptul că o problemă CSP poate fi rezolvată fără backtracking.

Teoremă. *Dacă o problemă CSP binară are o structură de arbore (cu alte cuvinte graful de consistenţă este de fapt arbore) şi verifică condiţiile de consistenţă a nodurilor şi arcelor, atunci o soluţie a problemei se poate construi fără backtracking.*

5.1.1.4 Algoritmi de tip Backtracking în context CSP

În continuare descriem şi discutăm şase algoritmi de bază şi patru variante hibride ale algoritmului Backtracking. Aceştia variază prin modul în care decid asupra testelor de consistenţă pe care le realizează, ca şi prin modificarea domeniului variabilelor care urmează să fie instanţiate la paşii viitori. Algoritmii vor fi descrişi în limbajul C++.

5.1.1.4.1 Funcţii şi structuri de date folosite

Vom folosi următoarele variabile, constante şi proceduri. Pentru simplificare presupunem că toate variabilele au acelaşi domeniu de valori.

- *N* este o constantă care desemnează numărul variabilelor. Acestea vor fi numerotate 1, ..., N.
- *K* este o constantă care desemnează dimensiunea domeniului. Domeniul de valori este mulţimea {1, ..., K}.
- *v* este un tablou uni-dimensional de dimensiune N care conţine instanţierile curente ale variabilelor.
- Funcţia principală a fiecărui algoritm returnează numărul variabilei care este selectată ca punct de revenire în evoluţia de tip backtracking a algoritmului.
- Funcţia *consistent*(*curent*) returnează 1 dacă instanţierea curentă este consistentă cu instanţierile anterioare (sau, în funcţie de algoritm, viitoare) şi 0 în caz contrar.
- Funcţia *check*(*i, j*) returnează 1 dacă testul de consistenţă dintre v[i] şi v[j] este un succes şi 0 în caz contrar.
- Subalgoritmul *soluţie*() procesează soluţia memorată în tabloul v.
- Subalgoritmul *reuniune*(S_1, S_2) determină reuniunea a două mulţimi, S_1 şi S_2, cu returnarea rezultatului în S_1.
- Subalgoritmul *vid*(*S*) reiniţializează o mulţime S := Ø.
- Subalgoritmul *adaugă*(*x, S*) adaugă elementul x la mulţimea S.

- Subalgoritmul *şterge*(*x, S*) şterge elementul x din mulţimea S.
- Funcţia *max*(*S*) returnează elementul de valoare maximă din mulţimea S.

Procesul de căutare este demarat prin apelarea funcţiei principale cu parametrul 1 (reprezentând prima variabilă).

5.1.1.4.2 Backtracking cronologic (BT)

Algoritmul Backtracking cronologic este algoritmul de bază din această serie. Este algoritmul clasic, care va fi în continuare optimizat. Este un algoritm simplu, care revine de fiecare dată la ultima variabilă modificată.

```
int consistent(int curent) {
        for (int i=1; i<curent;i++) {
                if (check(curent, i) == 0) {
                        return 0;
                }
        }
        return 1;
}

int BackTracking(int curent) {
        if (curent>N) {
                solutie();
                return N;
        }
        for (int i=0; i<K; i++) {
                v[curent] = i;
                if (consistent(curent)) {
                        BackTracking(curent+1);
                }
        }
        return curent-1;
}
```

Când se încearcă instanţierea variabilei de pe poziţia *curent*, se verifică dacă noua valoare este consistentă cu instanţierile variabilelor anterioare. Dacă da, execuţia continuă, iar în caz contrar se abandonează, încercând următoarea valoare posibilă pentru variabila curentă. Când toate alternativele au fost epuizate, algoritmul revine la variabila anterioară, şi încearcă instanţierea ei cu o nouă valoare, ş.a.m.d. De fiecare dată când toate variabilele au fost instanţiate şi toate testele de consistenţă au fost verificate cu succes, algoritmul generează câte o soluţie.

Algoritmii pe care îi vom descrie în continuare fac parte din categoria algoritmilor de **Backtracking Inteligent**, ei fiind variante ale BT care înceracă să reducă *trashing-ul*, marea problemă a BT.

5.1.1.4.3 Backjumping (BJ)

Diferenţa dintre algoritmii Backtracking şi Backjumping apare în situaţia în care nu se poate realiza o instanţiere consistentă pentru variabila curentă. În această situaţie, Backtracking revine la variabila precedentă, în timp ce Backjumping revine la variabila cu indicele cel mai mare şi care a generat o inconsistenţă.

```
int consistent(int curent) {
        for (int i=1; i<curent;i++) {
                if (check(curent, i) == 0) {
                        max_check[curent] = max(max_check[curent], i);
                        return 0;
                }
        }
        max_check[curent] = curent-1;
        return 1;
}

int BackJumping(int curent) {
        if (curent>N) {
                solutie();
                return N;
        }
        max_check[curent] = 0;
        for (int i=0; i<K; i++) {
                int jump;
                v[curent] = i;
                if (consistent(curent)) {
                        jump = BackJumping(curent+1);
                        if (jump != curent) {
                                return jump;
                        }
                }
        }
        return max_check[curent];
}
```

Valoarea max_check[*i*] reprezintă indicele cel mai mare al unei variabile care a fost inconsistentă cu variabila x_i. Testele de consistenţă sunt realizate în ordinea indicilor: x_1, x_2, ..., x_{i-1}. În situaţia unui blocaj, algoritmul va reveni la variabila desemnată de max_check[i]. Totuşi, în restul situaţiilor, revenirea este cronologică.

5.1.1.4.4 Conflict-Directed Backjumping (CBJ)

Mecanismul de revenire la Conflict-Directed Backjumping este mai sofisticat decât la Backjumping. Fiecare variabilă x_i are asociată o mulţime, conf[i], care reprezintă

mulţimea variabilelor anterioare care sunt în conflict cu x_i. De fiecare dată când nu mai sunt valori de încercat pentru variabilă x_i, algoritmul revine la variabila desemnată de indicele maxim din conf[i], x_h. De asemenea, mulţimea conflictelor conf[i] este absorbită de mulţimea conf[h].

```
int consistent(int curent) {
        for (int i=1; i<curent;i++) {
                if (check(curent, i) == 0) {
                        adaugă(i, conf[curent]);
                        return 0;
                }
        }
        return 1;
}

int ConflictDirectedBackJumping(int curent) {
        if (curent>N) {
                solutie();
                return N;
        }
        vid(conf[curent]);
        for (int i=0; i<K; i++) {
                int jump;
                v[curent] = i;
                if (consistent(curent)) {
                        jump = ConflictDirectedBackJumping(curent+1);
                        if (jump != curent) {
                                return jump;
                        }
                }
        }
        int h = max(conf[curent]);
        reuniune(conf[h], conf[curent]);
        return h;
}
```

Datorită memorării mulţimii conflictelor cu variabilele anterioare, algoritmul permite reveniri în cascadă. Astfel, după prima revenire la variabila corespunzătoare celui mai recent conflict, este posibilă revenirea în contiuare, din conflict în conflict. Este adevărat că preţul plătit pentru aceasta este necesitatea memorării unor informaţii suplimentare.

5.1.1.4.5 Graph-Based Backjumping (GBJ)

Algoritmul Graph-Based Backjumping încearcă, de asemenea, o revenire mai mare de un nivel, şi anume la variabila cu indicele cel mai mare, conectată la variabila curentă în graful de constrângeri.

```
int consistent(int curent) {
        for (int i=1; i<curent;i++) {
                if (check(curent, i) == 0) {
                        return 0;
                }
        }
        return 1;
}

int GraphBasedBackJumping(int curent) {
        if (curent>N) {
                solutie();
                return N;
        }
        for (int i=0; i<K; i++) {
                int jump;
                v[curent] = i;
                if (consistent(curent)) {
                        jump = GraphBasedBackJumping(curent+1);
                        if (jump != curent) {
                                return jump;
                        }
                }
        }
        reuniune(P, părinţi(curent));
        int h=max(P);
        şterge(h, P);
        return h;
}
```

Funcţia părinţi(i) returnează mulţimea părintilor variabilei x_i, adică mulţimea variabilelor x_j conectate la x_i în graful de constrângeri, cu $j<i$. Avantajul major al acestei variante apare doar dacă graful de constrângeri are puţine arcuri. Pe de altă parte, costurile suplimentare sunt mai mici ca la varianta Conflict-Directed Backjumping, deoarece mulţimea părinţilor se generează o singură dată, la începutul execuţiei algoritmului.

5.1.1.4.6 Backmarking (BM)

Algoritmul Backmarking foloseşte o schemă de marcare cu scopul de a elimina anumite teste de consistenţă inutile. Algoritmul se bazează pe următoarele observaţii:

(a) Dacă la cel mai recent nod unde o anumită instanţiere a fost verificată, testul de consistenţă a eşuat faţă de o instanţiere anterioară care încă nu s-a schimbat, atunci acest test va eşua din nou (economii de tipul A – aceste verificări pot fi evitate);

(b) Dacă la cel mai recent nod unde o anumită instanţiere a fost verificată, testul de consistenţă a reuşit faţă de toate instanţierile anterioare care încă nu s-a schimbat, atunci acest test va reuşi din nou (economii de tipul B – trebuie să verificăm această instanţiere doar împreună cu instanţierile anterioare care s-au schimbat).

```
int consistent(int curent) {
        int oldmbl = mbl[curent];
        if (mcl[curent][v[curent]] < oldmbl) {
                return 0;
        }
        for (int i=oldmbl; i<curent;i++) {
                mcl[curent][v[curent]] = i;
                if (check(curent, i) == 0) {
                        return 0;
                }
        }
        return 1;
}

int BackMarking(int curent) {
        if (curent>N) {
                solutie();
                return N;
        }
        for (int i=0; i<K; i++) {
                v[curent] = i;
                if (consistent(curent)) {
                        BackMarking(curent+1);
                }
        }
        int h = curent-1;
        mbl[curent] =h;
        for (int i=h+1; i<=N; i++) {
                mbl[i] = min(mbl[i], h);
        }
        return h;
}
```

Vectorul mbl (de dimensiune n) reprezintă nivelul minim de backup, şi vectorul mcl (de dimensiune n x m) reprezintă nivelul maxim de control. Valoarea mbl[i] este indicele cel mai mic al unei variabile a cărei instanţieri s-a schimbat de când variabila x_i a fost instanţiată ultima oară. Valoarea mcl[i][j] este cel mai mare indice al unei variabile care a fost verificată împreună cu a j-a valoare a domeniului variabilei x_i.

Algoritmul Backmarking vizitează aceleaşi noduri ca şi Backtracking, cu aceleaşi reveniri nenecesare. Totuşi, la anumite noduri ar putea să nu realizeze nici un test de consistenţă.

5.1.1.4.7 Forward Checking (FC)

Spre deosebire de algoritmii anteriori, Forward Checking realizează testele de consistenţă cu variabile viitoare, nu cu cele din trecut. Astfel, în momentul în care variabila x_i este instanţiată, domeniile variabilelor x_j, cu $j>i$, sunt filtrate de toate valorile care pot genera inconsistenţe cu instanţierea curentă.

```
void restore(int i) {
        for (int j=i+1; j<=N; j++) {
                if (checking[i][j]) {
                        checking[i][j] = 0;
                        for (int a=0; a<K; a++) {
                                if (domains[j][a] == i) {
                                        domains[j][a] = 0;
                                }
                        }
                }
        }
}

int consistent(int curent) {
        for (int j=curent+1; j<=N; j++) {
                int old = 0;
                int del = 0;
                for (int a=0; a<K; a++) {
                        if (domains[j][a] == 0) {
                                old++;
                                v[j] = a;
                                if (check(current, j) == 0) {
                                        domains[j][a] = curent;
                                        del++;
                                }
                        }
                }
                if (del) {
                        checking[curent][j] = 1;
                }
```

```
                if (old – del == 0) {
                        return j;
                }
        }
        return 0;
}

int ForwardChecking(int curent) {
        if (curent>N) {
                solutie();
                return N;
        }
        for (int i=0; i<K; i++) {
                if (domains[curent][i]) {
                        continue;
                }
                v[curent] = i;
                int fail = consistent(curent);
                if (fail == 0) {
                        ForwardChecking(curent+1);
                }
                restore(curent);
        }
        return curent-1;
}
```

Algoritmul foloseşte vectorul întreg domains, de dimensiune N x K şi vectorul boolean checking, de dimensiune N x N. Astfel, domains[i][j] > 0 înseamnă că a j-a valoare a fost înlăturată din domeniul variabilei x_i, iar domains[i][j] = 0 înseamnă că valoarea este în domeniu. De asemenea, checking[i][j] = 1 dacă instanţierea curentă a variabilei x_i cauzează modificarea domeniului variabilei x_j, şi checking[i][j] = 0 în caz contrar.

Spre deosebire de semantica de până acum, de această dată funcţia consistent(i) returnează numărul variabilei anihilate de prezenta instanţiere, sau 0 în caz contrar. Procedura restore(i) reface modificările aduse de instanţierea variabilei x_i.

5.1.1.4.8 Hibrizi ai algoritmului Backmarking

Pentru optimizarea testelor şi a revenirilor, algoritmii descrişi mai sus pot fi combinaţi pentru a da naştere unor aşa-numiţi algoritmi hibrizi. Astfel, algoritmul Backmarking a dat naştere următoarelor variante hibride:

- Backmarking şi Backjumping (BM-BJ);
- Backmarking şi Conflict-Directed Backjumping (BM-CBJ);
- Backmarking şi Graph-Based Backjumping (BM-GBJ).

Aceşti algoritmi combină funcţionalitatea algoritmilor componenţi. În continuare vom descrie algoritmul BM-BJ, urmând ca celelalte să fie lăsate în seama cititorului.

```
int consistent(int curent) {
	int oldmbl = mbl[curent];
	if (mcl[curent][v[curent]] < oldmbl) {
		return 0;
	}
	for (int i=oldmbl; i<curent;i++) {
		mcl[curent][v[curent]] = i;
		if (check(curent, i) == 0) {
			max_check[curent] = max(max_check[curent], i);
			return 0;
		}
	}
	max_check[curent] = curent-1;
	return 1;
}

int BackMarkingBackJumping(int curent) {
	if (curent>N) {
		solutie();
		return N;
	}
	max_check[curent] = 0;
	for (int i=0; i<K; i++) {
		int jump;
		v[curent] = i;
		if (consistent(curent)) {
			jump = BackMarkingBackJumping(curent+1);
			if (jump != curent) {
				return jump;
			}
		}
	}
	int h = max_check[curent];
	mbl[curent] =h;
	for (int i=h+1; i<=N; i++) {
		mbl[i] = min(mbl[i], h);
	}
	return h;
}
```

5.1.1.4.9 Hibrizi ai algoritmului Forward Checking

Algoritmii hibrizi creaţi prin combinarea algoritmului Forward Checking sunt următorii:

- Forward Checking şi Backjumping (FC-BJ);

- Forward Checking şi Conflict-Directed Backjumping (FC-CBJ);
- Forward Checking şi Graph-Based Backjumping (FC-GBJ).

Aceşti algoritmi combină funcţionalitatea algoritmilor componenţi. În continuare vom descrie algoritmul FC-BJ, urmând ca celelalte să fie lăsate în seama cititorului.

```
int ForwardCheckingBackJumping(int curent) {
	if (curent>N) {
		solutie();
		return N;
	}
	max_check[curent] = 0;
	for (int i=0; i<K; i++) {
		if (domains[curent][i]) {
			continue;
		}
		int jump;
		v[curent] = i;
		int fail = consistent(curent);
		if (fail == 0) {
			max_check[curent] = curent-1;
			jump = ForwardCheckingBackJumping(curent+1);
			if (jump != curent) {
				return jump;
			}
		}
		restore(curent);
		if (fail) {
			for (int j=1; j<curent; j++) {
				if (checking[j][fail]) {
					max_check[curent] = max(max_check[curent], j);
				}
			}
		}
	}
	int h = max_check[curent];
	for (int j=1; j<=curent; j++) {
		if (checking[j][curent]) {
			h = max(h, j);
		}
	}
	for (int i=curent; i>=h; i--) {
		restore(i);
	}
	return h;
}
```

5.1.1.5 Propagarea constrângerilor

Nu este neapărat necesar ca algoritmii de căutare (de tip backtracking) şi tehnicile de asigurare a consistenţei să opereze independent. O variantă îmbunătăţită este ca aceşti algoritmi să fie combinaţi într-o tehnică hibridă: algoritmul de consistenţă este introdus în algoritmul de căutare. Astfel, oricând valoarea unei variabile este modificată, se aplică o tehnică de consistenţă pentru a reduce graful de constrângeri. Vom exemplifica utilizarea algoritmului AC-3 pe trei tehnici de căutare:

- **Backtracking**: consistenţa se verifică între variabila curentă şi variabilele anterioare. Subalgoritmul de mai jos este apelat de către algortimul părinte oricând o valoare nouă este atribuită variabilei curente, V_{curent}.

subalgoritmul *AC3_BT*(*G, curent*) **este**
{Date: G - graful de consistenţă al problemei CSP}
{ - *curent* - indicele variabilei curente}
{Rezultate: G - graful de consistenţă al problemei CSP, în care domeniile variabilelor au fost modificate pentru a asigura consistenţa tuturor arcelor}
Q ← {(Vi,V*curent*) | (Vi,V*curent*) ∈arcuri(*G*), i≠*curent*}
consistent ← adevărat
cât-timp (*Q* ≠∅) ∧ *consistent* **execută**
@selectează şi şterge un arc oarecare (Vk, Vm) din Q
consistent ← REVISE(Vk,Vm)
sf-cât-timp
AC3_BT←*consistent*
sf-*AC3_BT*

- **Forward Checking**: consistenţa se verifică între variabila curentă şi variabilele viitoare. Subalgoritmul de mai jos este apelat de către algortimul părinte oricând o valoare nouă este atribuită variabilei curente, V_{curent}.

subalgoritmul *AC3_FC*(*G, curent*) **este**
{Date: G - graful de consistenţă al problemei CSP}
{ - *curent* - indicele variabilei curente}
{Rezultate: G - graful de consistenţă al problemei CSP, în care domeniile variabilelor au fost modificate pentru a asigura consistenţa tuturor arcelor}
Q ← {(Vi,V*curent*) | (Vi,V*curent*) ∈arcuri(*G*), i>*curent*}
consistent ← adevărat
cât-timp (*Q* ≠∅) ∧ *consistent* **execută**
@selectează şi şterge un arc oarecare (Vk, Vm) din Q
consistent ← REVISE(Vk,Vm)
sf-cât-timp

AC3_FC←*consistent*
sf-*AC3_FC*

- **Look Ahead**: consistenţa se verifică între variabile viitoare. Subalgoritmul de mai jos este apelat de către algortimul părinte oricând o valoare nouă este atribuită variabilei curente, V_{curent}.

subalgoritmul *AC3_LA*(*G, curent*) **este**
{Date: G - graful de consistenţă al problemei CSP}
{ - *curent* - indicele variabilei curente}
{Rezultate: G - graful de consistenţă al problemei CSP, în care domeniile variabilelor au fost modificate pentru a asigura consistenţa tuturor arcelor}
Q ← {(Vi,V*curent*) | (Vi,V*curent*) ∈arcuri(*G*), i>*curent*}
consistent ← adevărat
cât-timp (*Q* ≠∅) ∧ *consistent* **execută**
@selectează şi şterge un arc oarecare (Vk, Vm) din Q
dacă REVISE(Vk,Vm) **atunci**
Q ← *Q* ∪{(Vi, Vk)∈arcuri(*G*), i≠k, i≠m, i>*curent* }
consistent ← REVISE(Vk,Vm)
sf-dacă
sf-cât-timp
AC3_LA←*consistent*
sf-*AC3_LA*

5.1.1.6 Euristici de ordonare

În general, în algoritmii de tip BT variabilele sunt instanţiate secvenţial, astfel încât instanţierile parţiale să fie consistente. Schimbarea ordinii în care variabilele sunt analizate, sau schimbarea ordinii în care valorile sunt atribuite variabilelor, poate mări în mod vizibil eficienţa algoritmilor de satisfacere a constrângerilor. Reordonările vor putea fi realizate static (înainte de începutul căutării) sau dinamic (variabila următoare este selectată pe baza unui proces de reordonare local, la momentul necesar).

În ceea ce priveşte reordonarea variabilelor, algoritmii de căutare folosiţi au un impact semnificativ. Iată câteva dintre euristicile mai importante:

- ***Fail First Principle*** - selectează variabila cu cele mai puţine valori posibile. În acest fel, riscul unei alegeri greşite este mai mic.
- ***Minimum Width Ordering*** - selectează variabila care depinde de cele mai puţine variabile instanţiate anterior.
- ***Minimum Conflict First*** - selectează variabila pentru care numărul de valori care nu sunt în conflict cu variabile este minim.

5.1.2 CSP în Inteligenţa Artificială Distribuită

În cele ce urmează vom prezenta problema satisfacerii constrângerilor prin prisma unui sistem multi-agent. Spre deosebire de problemele de inteligenţă artificială bazate pe agenţi *competitivi*, de exemplu, jocurile cu doi jucători, problemele de satisfacere a constrângerilor se bazează pe agenţi *cooperativi*: agenţii disponibili vor prelua părţi ale problemei şi vor coopera la rezolvarea acesteia. Operaţiile desfăşurate de fiecare dintre agenţi sunt independente de operaţiile desfăşurate de ceilalţi agenţi, acestea putând fi executate asincron şi concurent.

Presupunem următorul mecanism de distribuire a sarcinilor, de comunicare şi colaborare între agenţi:

- Fiecare agent corespunde unei variabile din graful de constrângeri asociat problemei de rezolvat.
- Agenţii comunică prin trimiterea de mesaje. Un agent poate trimite un mesaj numai către un alt agent cu identificator cunoscut.
- Între oricare doi agenţi, mesajele sunt primite în ordinea în care au fost transmise.
- Datorită întârzierilor în comunicarea dintre agenţi, nu se poate conta pe o anumită ordine de primire a mesajelor, cu excepţia situaţiei de mai sus.

În cele ce urmează vom identifica variabila V_i cu agentul corespunzător acesteia. Agenţii care au, în graful de constrângeri, legături directe cu agentul V_i se numesc vecini ai lui V_i. Agenţii cunosc identificatorii vecinilor lor.

5.1.2.1 Algoritmul de filtrare

Algoritmul de filtrare descris în continuare este varianta distribuită a algoritmului de asigurare a arc-consistenţei AC-1 descris în secţiunea 5.1.1. Acest algoritm foloseşte funcţia REVISE(V_i,V_j) descrisă în aceeaşi secţiune.

Fiecare agent începe prin a transmite vecinilor domeniul său de valori. De fiecare dată când un agent primeşte un mesaj de la un vecin, aplică procedura de revizuire a propriului domeniu corespunzător constrângerii cu acel vecin. Dacă în urma aplicării acestei proceduri, propriul domeniu se modifică, atunci noul domeniu este transmis tuturor vecinilor.

5.1.2.2. Algoritmul de consistenţă bazat pe hiper-rezoluţie

Algoritmul descris în continuare este varianta distribuită a algoritmului de asigurare a *k*-consistenţei. Acest algoritm foloseşte noţiunea de *set ilegal*, care este un *t*-uplu de atribuiri ilegale, în sensul că acestea violează cel puţin o constrângere între variabilele implicate.

Fiecare agent îşi reprezintă constrângerile ca seturi ilegale. Prin combinarea informaţiilor despre domeniul său şi seturile ilegale existente, agentul generează noi seturi ilegale. Orice set ilegal nou generat va fi transmis tuturor agentilor vecini. Dacă

agentul primeşte un set ilegal nou, îl va folosi în procesul de raţionare, pentru a genera noi seturi ilegale.

Un set ilegal se generează plecând de la seturi ilegale cunoscute şi de la domeniul unei variabile, folosind o regulă de deducţie denumită regula hiper-rezoluţiei, de genul următor:

$(V_1=x_{11} \vee V_1=x_{12} \vee V_1=x_{13} \vee ... \vee V_1=x_{1m})$
$\neg(V_1=x_{11} \wedge V_{i1}=x_{i1} \wedge ...)$,
$\neg (V_1=x_{12} \wedge V_{i2}=x_{i2} \wedge ...)$,
...
$\neg (V_1=x_{1m} \wedge V_{im}=x_{im} \wedge ...)$

$\neg (V_{i1}=x_{i1} \wedge ... \wedge V_{i2}=x_{i2} \wedge ... V_{im}=x_{im} \wedge ...)$

În cursul procesului de deducţie a unor noi seturi ilegale, se pot folosi o serie de proprietăţi interesante. Astfel, un superset al unui set ilegal este ilegal, la rândul său. Rezultă de aici că, dacă la un moment dat, se obţine setul ilegal {}, atunci problema este supra-constrânsă, deoarece orice set de atribuiri, fiind superset al mulţimii vide, este set ilegal.

5.1.2.3. Algoritmul Backtracking asincron

Algoritmul Backtracking asincron este varianta distribuită a algoritmului Backtracking, discutat în secţiunea 5.1.1. Metoda fixează o ordine de prioritate între agenţi. Astfel, agentul V_i are prioritate mai mare decât agentul V_j, cu $i<j$. Un agent va schimba valoarea atribuită variabilei sale oricând aceasta nu este consistentă cu valoarea comunicată de un agent cu prioritate mai mare. Orice schimbare de valoare este comunicată agenţilor cu prioritate mai mică. Când nu mai există valori care să fie testate, agentul generează un set ilegal pe care il transmite unui vecin cu prioritate mai mare, care îşi va schimba valoarea.

Fiecare agent păstrează valoarea curentă a celorlalţi agenţi, aşa cum este comunicată. Notăm mulţimea acestor atribuiri cu *info_local*. Totuşi, datorită modului de comunicaţie dintre agenţi, informaţia memorată poate fi deja perimată. Astfel, agentul nu poate solicita unui agent cu prioritate mai mare să îşi schimbe valoarea. Îi poate, însă, comunica setul ilegal pe care l-a generat, iar agentul va putea verifica dacă valoarea sa este încă ilegală, în care caz o va schimba.

Agenţii transmit două tipuri de mesaje: *bun*(V_i, x_i), care transmite valoarea curentă x_i a variabilei V_i, şi *ilegal*(set ilegal), care transmite un set ilegal. Notăm cu V_{curent} variabila curentă şi cu D_{curent} şi x_{curent} domeniul şi valoarea curentă ale acesteia.

când primeşti bun(Vi, xi) **execută**
    @adaugă (Vi, xi) la info_local
    @verifică_info_local
sf-execută

când primeşti ilegal(set ilegal) **execută**
    @memorează setul ilegal

pentru (Vk, xk) din setul ilegal, unde xk nu este vecin **execută**
@cere lui Vk să adauge V_curent la mulţimea vecinilor săi
@adaugă Vk la mulţimea vecinilor
@adaugă (Vk, xk) la info_local
sf-pentru
verifică_info_local
sf-execută

subalgoritmul *verifică_info_local*
când info_local şi x_curent nu sunt consistente, **execută**
dacă nu există în D_curent o valoare consistentă cu info_local
atunci
@generează un nou set ilegal folosind hiper-rezoluţia
@trimite setul ilegal la agentul cu prioritate minimă din acest set
când s-a găsit un set ilegal vid **execută**
@trimite mesaj către agenţi: nu există soluţie
@termină algoritmul
sf-execută
altfel
@selectează x din D_curent, consistent cu info_local
@x_curent← x
@trimite la vecini mesajul bun(V_curent, x_curent)
sf-dacă
sf-execută
sf-*verifică_info_local*

5.1.2.4. Căutare asincronă Weak-Commitment

Algoritmul de căutare asincronă Weak-Commitment generalizează algoritmul de Backtracking asincron prin introducerea posibilităţii modificării dinamice a priorităţii agenţilor, în acest fel fiind modificată ordinea în care agenţii vor reacţiona.

Prioritatea agenţilor se defineşte ca un număr întreg nenegativ, o valoare numerică mai mare corespunzând unei priorităţi mai mari. În caz de balotaj, ordinea este determinată de indicele numeric al variabilelor: indicele mai mic corespunde unei priorităţi mai mari. Valoarea de prioritate iniţială pentru toţi agenţii este 0. De asemenea, dacă nu există o valoare consistentă pentru V_i, valoarea de prioritate a acesteia este fixată la k+1, unde k este valoarea de prioritate maximă a agenţilor vecini.

când primeşti bun(Vi, xi, prioritate) **execută**
@adaugă (Vi, xi, prioritate) la info_local
@verifică_info_local
sf-execută

când primeşti ilegal(set ilegal) **execută**
@memorează setul ilegal

```
        pentru (Vk, xk) din setul ilegal, unde xk nu este vecin execută
                @cere lui Vk să adauge V_curent la mulţimea vecinilor săi
                @adaugă Vk la mulţimea vecinilor
                @adaugă (Vk, xk) la info_local
        sf-pentru
        verifică_info_local
sf-execută

subalgoritmul verifică_info_local
        când info_local şi x_curent nu sunt consistente execută
                dacă nu există în D_curent o valoare consistentă cu info_local
                atunci
                        @generează un nou set ilegal folosind hiper-rezoluţia
                        când s-a găsit un set ilegal vid execută
                                @trimite mesaj către agenţi: nu există soluţie
                                @termină algoritmul
                        sf-execută
                        când s-a găsit un set ilegal nou execută
                                @trimite setul ilegal la agenţii din set
                                prioritate_curent ←1 + p_max
                                //unde p_max este prioritatea maximă a vecinilor;
                                selectează_cea_mai_bună_valoare
                        sf-execută
                altfel
                        selectează_cea_mai_bună_valoare
                sf-dacă
        sf-execută
sf-verifică_info_local

subalgoritmul selectează_cea_mai_bună_valoare
        @selectează x din D_curent, consistent cu info_local şi
                x minimizează numărul constrângerilor violate cu agenţii cu
                prioritate mai mică;
        x_curent ← x;
        @trimite la vecini mesajul
                bun(V_curent, x_curent, prioritate_curent);
sf-selectează_cea_mai_bună_valoare
```

5.2 Probleme de căutare a unui drum (Path Finding Problems - PFP)

O problemă de căutare a unui drum (Path Finding Problem) poate fi definită în felul următor.

Definiţie

Se dau următoarele:

- o mulţime **N** de noduri, fiecare nod reprezentând o stare a problemei;

- o mulţime de legături directe între stări **L**, fiecare legătură directă (arc) reprezentând un operator disponibil unui agent care rezolvă problema;
- un nod unic ***si***, numit **stare iniţială**;
- o mulţime de noduri **G**, fiecare nod din această mulţime reprezentând o **stare finală**;
- pentru fiecare legătură se cunoaşte **costul** legăturii (reprezentând costul aplicării operatorului - se mai numeşte şi **distanţa** dintre noduri).

În acest context se cere să se afle un drum (în general, de cost total minim) de la starea iniţială la una din stările finale.

În cazul în care toate costurile asociate operatorilor sunt 1, problema se reduce la a afla numărul minim de operatori necesari pentru a ajunge din starea iniţială într-una din stările finale.

În contextul unei probleme de căutare a unui drum vom numi *vecinii unui nod i* acele noduri *j* cu proprietatea că (*i, j*) este arc.

Iată câteva exemple de probleme din această clasă de probleme:

- Problema **n-puzzle**: *Se dau $k=n^2-1$ căsuţe pe o tablă nxn. Una dintre căsuţele de pe tablă este goală. Sunt permise următoarele mutări: o căsuţă adiacentă căsuţei goale pe orizontală sau vericală îşi poate schimba locul cu aceasta. Se cere să se afle numărul minim de mutări pentru a transforma o configuraţie iniţială într-o configuraţie finală (ambele date).*
- Problema căutării într-un labirint (căutarea ieşirii dintr-un labirint, eventual cu obstacole).
- Probleme de planificare pentru roboţi.

Algoritmul clasic de rezolvare a problemelor din clasa PFP este algoritmul A*, pe care îl vom prezenta în secţiunea următoare. Algoritmul A* este de fapt un algoritm specific tehnicii **Branch and Bound**, fiind o generalizare a tehnicii de căutare **Best-First**.

5.2.1 PFP în Inteligenţa Artificială Tradiţională

Contextul în care vor fi descrise următoarele tehnici de căutare este contexul căutării unui drum într-un spaţiu de stări, aşa cum afost definită ăn secţiunea anterioară o problemă PFP.

5.2.1.1 Căutare Best-First

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

La fiecare pas al procesului de căutare selectăm nodul cel mai promiţător dintre cele pe care le-am generat până acum, prin aplicarea unei funcţii euristice pe toate aceste noduri. Apoi utilizăm regulile aplicabile pentru expandarea nodului ales şi generarea succesorilor săi. Dacă unul din succesori este soluţie, ne oprim. Dacă nu, toate aceste

noduri noi sunt adăugate la mulţimea de noduri generate până acum. Din nou, nodul cel mai promiţător este selectat şi procesul continuă. Ceea ce se întâmplă de obicei este că apare puţină căutare depth-first pe măsură ce drumul cel mai promiţător este explorat. Dar, dacă o soluţie nu este finalmente găsită, acel drum va începe să pară mai puţin promiţător decât altul care mai înainte a fost ignorat. În acel punct, drumul care a devenit mai promiţător este selectat şi analizat, dar drumul vechi nu este uitat. Căutarea poate reveni la acesta oricând toate celelalte drumuri devin mai proaste şi când acesta este din nou drumul cel mai promiţător.

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

Pentru implementarea procedurii de căutare Best-First vom avea nevoie de două liste de noduri:

- **OPEN –** noduri care au fost generate şi au ataşată valoarea funcţiei euristică, dar nu au fost încă examinate (nu le-au fost generaţi succesorii); această listă este o coadă cu priorităţi.
- **CLOSED –** noduri care deja au fost examinate; în cazul în care căutăm un arbore avem nevoie de această listă, pentru a evita generarea repetată a unor noduri.

Vom avea nevoie de o funcţie euristică care estimează meritele fiecărui nod pe care îl generăm. Această funcţie va permite algoritmului să caute drumul cel mai promiţător. Să o notăm cu f', ca să indicăm că este o aproximare a funcţiei f care întoarce valoarea corectă, exactă, a nodului. Pentru multe aplicaţii această funcţie o definim ca sumă a două componente, g şi h'.

Funcţia g este o măsură (nu o estimare) a costului trecerii din starea iniţială în starea curentă, iar funcţia h' este o estimare a costului adiţional de a obţine starea finală din starea curentă. Astfel funcţia combinată f' reprezintă o estimare a costului obţinerii stării finale din starea iniţială dea lungul drumului pe care ne aflăm. Să mai notăm că h' trebuie să fie o estimare a costului unui nod (cu cât mai bun este nodul, cu atât mai mică este valoarea funcţiei), nu a calităţii nodului (cu cât mai bun este nodul, cu atât mai mare este valoarea funcţiei). De asemenea, g trebuie să fie nenegativă, pentru a evita ciclurile din grafe, întrucât drumurile cu cicluri vor părea mai bune decât cele fără cicluri.

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

Algoritm: Căutare Best-First

1. Incepe cu OPEN ← {starea iniţială}
2. Până când se găseşte o stare finală, sau până când OPEN este vidă, execută:
   (a) Alege nodul cel mai bun din OPEN (nodul cu cel mai mic f').

(b) Generează-i succesorii.
(c) Pentru fiecare succesor execută:
i. Dacă nu a fost generat deja, evaluează-l, adaugă-l la OPEN şi înregistrează-i părinţii.
ii. Dacă a fost generat deja, schimbă-i părintele dacă acest drum nou este mai bun decât cel dinainte. În acest caz, actualizează costul obţinerii acestui nod (valoarea lui **g**) şi a oricărui nod succesor al acestuia.

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

5.2.1.2 Algoritmul A*

Algoritmul de căutare Best-First prezentat mai sus este o simplificare a unui algoritm numit *A**. Acest algoritm foloseşte funcţiile *f'*, *g* şi *h'* şi listele OPEN şi CLOSED descrise mai sus. Vom nota cu *si* starea iniţială a problemei.

Algoritm: A*

1. **Începe** cu OPEN ←{*si*}; *g*(*si*) ← 0; calculează *h'*(*si*); *f'*(*si*) ← *h'*(*si*) +0; CLOSED ← {}
2. **Până când se găseşte un nod final**, repetă următoarea procedură:
   - Dacă nu există noduri în OPEN, raportează eşec;
   - Dacă există noduri în OPEN, atunci:
     – BEST ← nodul din OPEN cu cel mai mic *f'*;
     – Scoate BEST din OPEN; plasează-l în CLOSED;
     – Verifică dacă BEST este stare finală;
     – Dacă DA, exit şi raportează soluţia (BEST sau drumul la el);
     – Dacă NU, generează succesorii lui BEST, dar nu pointa BEST la aceştia; Pentru fiecare astfel de SUCCesor execută următoarele:
     (a) Legătura părinte a lui SUCC să indice înapoi la BEST;
     (b) g(SUCC) ← g(BEST) + costul drumului de la BEST la SUCC;
     (c) **Dacă SUCC apare în OPEN**, atunci
     - Nodul din OPEN se numeşte OLD;
     - Elimină SUCC şi adaugă OLD la succesorii lui BEST;
     - Dacă drumul tocmai găsit până la SUCC este mai ieftin decât drumul cel mai bun până la OLD (*g*(SUCC) ≤ *g*(OLD)), atunci
       - Părintele lui OLD trebuie resetat la BEST;
       - Actualizează *g*(OLD) şi *f'*(OLD).

     (d) **Dacă SUCC nu apare în OPEN, dar apare în CLOSED**, atunci
     - Nodul din CLOSED se numeşte OLD;
     - Adaugă OLD la lista succesorilor lui BEST;
     - Execută pasul 2(c)iii;

- Dacă am găsit un drum mai bun la OLD, îmbunătăţirea trebuie propagată la succesorii lui OLD, astfel:
    - OLD indică la succesorii săi; aceştia – la ai lor, ş.a.m.d.; fiecare ramură se termină fie cu un nod în OPEN, fie fără succesori;
    - fă o traversare Depth-First cu începere din OLD, cu schimbarea lui g şi f' ai nodurilor traversate, cu terminare la un nod fără succesori sau un nod până la care s-a găsit deja un drum cel puţin la fel de bun;
- condiţia "drum cel puţin la fel de bun" înseamnă următoarele:
    - dacă cel mai bun părinte al nodului este cel de unde venim, atunci continuă propagarea;
    - dacă nu, verifică dacă drumul nou actualizat este mai bun decât drumul memorat;
    - dacă este mai bun, resetează părintele şi continuă propagarea;
    - dacă nu, opreşte propagarea.

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

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

Iată în continuare câteva observaţii interesante despre acest algoritm.

Observaţia 1 priveşte rolul funcţiei g. Ne permite să alegem care nod să-l expandăm nu numai pe baza a cât de aproape este de starea finală, ci şi a cât de aproape este de starea iniţială.

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

Observaţia 2 priveşte rolul funcţiei h', estimarea funcţiei h, distanţa unui nod faţă de nodul final.

- Dacă h' este un estimator perfect al lui h, atunci **A*** va converge imediat la soluţie, fără căutare.
- Cu cât h' este mai bun, cu atât mai aproape vom fi de o căutare directă. Dacă, pe de altă parte, valoarea lui h' este întotdeauna zero, căutarea va fi controlată de g.
- Dacă valoarea lui g este de asemenea întotdeauna zero, strategia de căutare va fi aleatoare.

- Dacă valoarea lui g este întotdeauna 1, căutarea va fi Breadth-First. Toate nodurile de pe un nivel vor avea aceeaşi valoare a lui g, şi astfel vor avea valori ale lui f' mai mici decât nodurile de pe nivelul următor.
- Ce se întâmplă dacă h' nu este nici perfect, nici zero? Dacă putem garanta că h' nu supraestimează niciodată pe h, algoritmul **A*** garantează găsirea unui drum optimal către stare finală (determinat de g), dacă un astfel de drum există. Aceasta se poate vedea în câteva exemple la care vom reveni mai târziu.

Definiţie: Vom numi funcţia h' **funcţie euristică**.

Un exemplu de problemă a cărei rezolvare se face aplicând algoritmul A* este problema **n-puzzle**, pentru a cărei rezolvare, se poate folosi ca euristică (aplicată unei configuraţii C) distanţa Manhattan de la C până la configuraţia finală, calculată astfel: pentru fiecare căsuţă $c \in C$ care nu e goală şi care nu e aşezată corect (faţă de configuraţia finală) se adună distanţa Manhattan dintre poziţia lui c în configuraţia C şi poziţia lui c în configuraţia finală.

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

Definiţie: O euristică h' se numeşte **admisibilă** dacă nu supraestimează pe h.

Euristica anterioară (distanţa Manhattan până la starea finală) este o euristică **admisibilă**. În cazul în care nu se poate găsi o euristică bună, se poate satisface condiţia de admisibilitate a euristicii setând toate valorile lui h' la 0, ceea ce ne conduce la breadth-first, care e admisibil, dar ineficient.

Observaţia 3 priveşte relaţia dintre arbori şi grafe. Algoritmul **A*** a fost prezentat în forma sa cea mai completă, aplicat pe grafe. Poate fi simplificat să fie aplicat la arbori, prin aceea că nu se va mai verifica dacă un nod nou este deja în listele OPEN şi CLOSED. Aceasta face generarea nodurilor mai rapidă, dar poate duce la efectuarea de mai multe ori a unei operaţii de căutare, dacă nodurile sunt duplicate.

Teoremă. Dacă h' nu supraestimează pe h atunci **A*** este admisibil.

O altă observaţie referitoare la algoritmul **A*** este faptul că acesta este exponenţial atât ca şi timp de execuţie cât şi ca şi spaţiu de memorare.

5.2.2 PFP în Inteligenţa Artificială Distribuită

Una dintre întrebările justificate legate de clasa problemelor de căutare a unui drum este următoarea: *Cum poate fi legată formalizarea PFP de Inteligenţa Artificială Distribuită?*. Cu alte cuvinte, cum pot fi folosiţi mai mulţi agenţi în rezolvarea unei probleme de PFP?.

Răspunsul la această întrebare este simplu: e suficient să ne gândim la o problemă în care mai mulţi roboţi explorează un mediu necunoscut în scopul găsirii unei anumite locaţii şi o astfel de problemă poate fi formalizată ca o problemă de căutare a unui drum.

5.2.2.1 Programarea dinamică asincronă (Asyncronous Dynamic Programming - ADP)

Într–o problemă de căutare a unui drum principiul optimalităţii este valabil: ***„Dacă un drum este optim, atunci orice segment al acestuia este optim”***. Reciproc acesta nu este neapărat adevărat, adică dacă segmentele unui drum sunt optimale nu implică faptul că drumul este optimal.

Fie o problemă de căutare a unui drum. Să notăm distanţa minimă de la un nod *i* la un nod final cu $h^*(i)$. Din principiul optimalităţii, cel mai scurt drum de la nodul *i* la un nod final, printr–un nod vecin *j*, este dată de formula:

$$f^*(j) = k(i,j) + h^*(j),$$

unde $k(i,j)$ este distanţa dintre *i* şi *j* (valoarea arcului dintre cele două noduri).

Dacă *i* nu este un nod terminal, atunci are loc următoarea relaţie:

$$h^*(i) = min_j f^*(j).$$

Dacă $h^*(i)$ este cunoscut pentru fiecare nod *i*, atunci drumul optim poate fi determinat cu următoarea procedură:

- pentru fiecare nod *j*, vecin al nodului *i*, calculează $f^*(j) = k(i,j) + h^*(j)$, apoi deplasează-te la acel *j* pentru care $f^*(j)=min_j f^*(j)$.

Programarea dinamică asincronă calculează $h^*(i)$ prin repetarea calculelor locale pentru fiecare nod. Considerăm următorul model de interacţiune:

- pentru fiecare nod *i* există un proces (agent) corespunzător;
- fiecare proces *j* înregistrează valoarea $h(j)$, care este o estimare a valorii $h^*(j)$. Valoarea iniţială a lui $h(j)$ este arbitrară (ex.: ∞,0), cu excepţia nodurilor finale;
- pentru fiecare nod final *g*, $h(g)$=0;
- fiecare proces poate accesa valorile funcţiei *h* a nodurilor vecine (prin memorie partajată sau prin transmitere de mesaje)

În această situaţie fiecare agent *i* actualizează $h(i)$ după următoarea procedură:

- pentru fiecare nod *j* vecin al lui *i*, calculează: $f(j) = k(i,j) + h(j)$, unde $h(j)$ este distanţa curentă estimată de la nodul *j* la un nod final, iar $k(i,j)$ este costul arcului (i, j), apoi actualizează $h(j)$ astfel: $h(j)=min_j f(j)$.

În practică nu putem folosi algoritmul ADP pentru probleme PFP de dimensiuni mari (cu un număr mare de noduri). În cazul în care numărul de noduri este foarte mare, nu ne putem permite să avem procese (agenţi) pentru toate nodurile.

Oricum, ADP poate fi considerată ca fiind baza algoritmilor de rezolvare a problemelor de căutare a unui drum în sistemele multiagent. Ca exemple de algoritmi putem aminti: învăţarea în timp real A^*, algoritmul în timp real A^*, algoritmul de căutare

a ţintei mobile şi algoritmul bidirecţional de căutare în timp real, precum şi algoritmi multiagent de căutare în timp real. În aceşti algoritmi, în locul alocării unor procese pentrui toate nodurile, este introdus un control al execuţiei de către un număr rezonabil de procese - sau agenţi).

5.2.2.2 Algoritmul de învăţare în timp real A* (Learning Real Time A* - LRTA*)

Algoritmul care stă la baza Algoritmului de învăţare în timp real A* este Algoritmul A*, care este o variantă nedistribuită şi mai generală a celui ce urmează să fie prezentat. După cum am văzut în secţiunea 5.2.1.2, în cadrul acestuia sunt actualizate valorile pentru toate nodurile, ceea ce implică un volum mare de calcule inutile de vreme ce drumul va fi format doar din succesiunea câtorva dintre acestea. Prin efectuarea acestor calcule se garantează admisibilitatea algoritmului A*.

Algoritmul LRT A*

Acest algoritm are la bază algoritmul de programare dinamică asincronă (ADP) şi algoritmul A*.

Atunci când un singur agent rezolvă o problemă de căutare nu este întotdeauna posibil ca acesta să facă toate calculele locale pentru fiecare nod în parte. De exemplu, anumiţi agenţi autonomi s–ar putea să nu aibă timp suficient pentru planificare şi de accea vor trebui să îmbine planificarea cu execuţia. De aceea agenţii care acţionează independent trebuie să execute selectiv calcule doar pentru anumite noduri. Având în vedere această cerinţă, întrebarea care se pune este: ce nod să aleagă agentul? Răspunsul intuitiv la această întrebare este: să aleagă pentru actulaizare nodul în care se află.

Este cunoscut faptul că aria de percepţie a unui agent este una limitată, de aceea este uşor pentru agent să actualizeze valoarea funcţiei *h* a nodului curent, pentru ca apoi să poată trece pe poziţia următoare, şi anume a nodului vecin cel mai potrivit în atingerea scopului. Acest procedeu se repetă până când se ajunge într–o stare finală (la scopul propus). Aceasta este ideea de bază a algoritmului LRTA*.

Mai precis, în algoritmul LRTA* fiecare agent execută următoarea procedură (presupunem că nodul în care se află agentul este nodul *i*):

1. ***Calcul***
   Calculează $f(j) = k(i,j) + h(j)$, pentru fiecare vecin *j* al lui *i*
2. ***Actualizare***
   $h(i) = min_j f(j)$
3. ***Selectarea acţiunii***
   Agentul se deplasează la vecinul *j* care are cea mai mică valoare *f(j)*. Legăturile spre valorile *f(j)* minime sunt rupte aleator.

Notaţiile sunt aceleaşi cu cele din prezentarea algoritmului de programare dinamică asincronă. Ca şi în cazul algoritmului de programare dinamică asincronă, agentul înregistrează distanţa estimată *h(i)* pentru fiecare nod.

O caracteristică a acestui algoritm este aceea că agentul determină şi execută următoarea acţiune într–un timp constant. De aceea este numit algoritm de căutare *on–line, în timp real*. În acest algoritm valoarea lui *h* trebuie să fie una optimă, trebuie să nu supraevalueze valoarea reală *h**. Această caracteristică este descrisă prin condiţia $h(i) \leq h^*(i)$, care trebuie satisfăcută pentru fiecare nod *i*. Dacă valorile iniţiale satisfac acestă condiţie *h(i)* nu va fi mai mare decât valoarea reală $h^*(i)$, prin actualizare.

Ca şi în cazul algoritmului A*, vom numi funcţia care dă valorile iniţiale ale funcţiei *h* o *funcţie* ***euristică***. O funcţie euristică este numită ***admisibilă*** dacă nu supraestimează. Dacă nu găsim nici o funcţie euristică bună putem satisface condiţia setând toate estimările la 0.

În algoritmul de programare dinamică asincronă valorile iniţiale sunt arbitrare şi pot fi infinit. Ca urmare, întrebarea este: Ce anume face diferenţa între ADP şi LRTA*?

În algoritmul de programare dinamică asincronă actualizarea se face pentru fiecare nod. De aceea valoarea funcţiei *h* penru nodul respectiv va converge într–un final către valoarea reală indiferent de valoarea ei iniţială. Pe de altă parte, în cazul algoritmului LRTA* actualizarea se face doar pentru nodurile vizitate de agent. Astfel, dacă valoarea iniţială a funcţiei *h*, pentru nodul *i*, este mai mare decât valoare reală este posibil ca agentul să nu mai viziteze acest nod *i*, şi deci nodul *i* să nu mai fie luat în considerare.

În cazul unei probleme cu număr finit de noduri, în care costurile arcelor sunt pozitive şi în care există cel puţin un drum de la fiecare nod către un nod final, pornind cu estimări euristice admisibile şi nenegative, algoritmul LRTA* este ***complet***, adică va ajunge în final la un nod final.

Algoritmul este complet şi convergent, pentru probleme cu un număr finit de noduri. El nu este optimal deoarce el găseşte o estimare a soluţiei, dar nicodată nu supraestimează, ba mai mult prin repetări algoritmul *învaţă* să ajungă la soluţia optimă, dacă valorile estimate iniţial sunt admisibile.

O schiţă a demonstraţiei completitudinii acestui algoritm este descrisă în cele ce urmează. Fie $h^*(i)$, costul celui mai scurt drum de la nodul *i* la un nod final, şi fie *h(i)*, valoarea euristică a stării *i*. În primul rând, pentru fiecare stare *i*, are loc aproximarea $h(i) \approx h^*(i)$, deoarece această condiţie este adevărată la momentul iniţial când toate valorile lui *h* sunt admisibile, ceea ce înseamnă că acestea nu supraestimează costul real niciodată şi avem garanţia că această condiţie nu îşi va schimba valoarea de adevăr prin actualizare. Definim *eroarea euristică* la un anumit punct al algoritmului ca fiind suma diferenţelor $h^*(i)–h(i)$, calculate pentru fiecare stare *i*. Definim şi noţiunea de *imparitate euristică* ca fiind o cantitate pozitivă reprezentată de suma dintre eroarea euristică şi valoarea euristică a stării curente *i*. Este uşor de arătat că la fiecare pas al algoritmului această cantitate scade şi din moment ce nu poate fi negativă dacă aceasta ajunge la valoarea 0, atunci problema este rezolvată. Algoritmul deci se va termina, în final, cu succes.

Această demonstraţie poate fi uşor extinsă şi pentru cazul în care ţinta (nodul final) se mişcă.

În continuare vom demonstra convergenţa algoritmului. Se spune că valoarea *h(i)* este corectă dacă $h(i)=h^*(i)$. Dacă drumul este cel mai scurt, atunci la mutarea din starea *i* în starea vecină *j* dacă *h(j)* este corect, *h(i)* va fi corect după actualizare. Stiind că valorile funcţiei *h* sunt întotdeauna corecte pentru stările finale (ţintă) şi va fi ales la un moment dat drumul cel mai scurt, putem spune că *h(i)* converge către valoarea corectă $h^*(i)$.

5.2.2.3 Algoritmul în timp real A* (Real Time A* - RTA*)

Diferenţa esenţială dintre Algoritmul de învăţare în timp real A* şi Algoritmul în timp real A* este faptul că în actualizarea valorii funcţiei *h* nu se mai foloseşte valoarea minimă, ci se utilizează cea de–a doua valoare minimă. Funcţia folosită pentru a obţine cea de-a doua valoare minimă este numită *secondmin*.

Algoritmul RTA*este schiţat mai jos (*i* este starea curentă a agentului):

1. ***Calcul***
   Calculează $f(j) = k(i,j) + h(j)$, pentru fiecare vecin *j* al lui *i*
2. ***Actualizare***
   $h(i) = secondmin_j\ f(j)$
3. ***Selectarea acţiunii***
   Agentul se deplasează la vecinul *j* care are cea mai mică valoare *f(j)*. Legăturile spre valorile *f(j)* minime sunt rupte aleator.

Notaţiile sunt aceleaşi cu cele din prezentarea algoritmului de programare dinamică asincronă.

La fel ca LRTA*, în cazul unei probleme cu număr finit de noduri, cu legături pozitive, în care există cel puţin un drum de la fiecare nod către un nod final, pornind cu valori estimative admisibile şi nenegative, algoritmul RTA* este ***complet***, adică va ajunge în final la un nod ţintă.

De vreme ce este reţinută cea de–a doua valoare minimă, RTA* poate lua decizii locale optime în spaţiul arborelui problemei. Adică orice mutare făcută de RTA* este de–a lungul unui drum care estimează costul spre starea finală cu un număr minim de cunoştinţe bazate pe informaţiile deja obţinute. Cu toate acestea, acest rezultat nu poate fi extins pentru a acoperi şi grafele în care întâlnim cicluri.

O altă caracteristică este faptul că, deşi învaţă mai eficient decât algoritmul LRTA*, algoritmul RTA* poate supraestima valorile euristicii.

5.2.2.4 Algoritmul de căutare a ţintei mobile (Moving Target Search - MTS)

Acest algotim este o generalizare a LRTA* pentru cazul în care ţinta se poate mişca (este mobilă).

Algoritmii de căutare euristică presupun că starea finală este fixă, adică ţinta agentului este una care nu se modifică pe parcursul căutării. Spre deosebire de aceştia, în algoritmul MTS ţinta este una mobilă. În loc să caute drumul către o ţintă care staţionează, în cazul MTS agentul poate avea ca şi scop găsirea altui robot, care, de asemenea, se mişcă. Agentul ţintă poate coopera cu cel care vrea să ajungă la el, poate să aibă o mişcare independentă de cea a agentului care îl caută, sau poate să îl evite. Nu există nici o afirmaţie cu privire la faptul că agentul ţintă s–ar putea opri, dar scopul este atins în momentul în care poziţiile celor doi agenţi coincid. Pentru ca problema să aibă sens, în cazul în care agentul căutat nu cooperează cu cel care îl caută, primul amintit trebuie să se mişte mai încet decât cel de–al doilea. În caz contrar agentul ţintă îl poate evita pe celălalt

la nesfârşit chiar şi într–un spaţiu finit, doar fiind atent să nu ajungă într–un blocaj (*dead-end*).

În algoritmul MTS trebuie să se obţină şi să se reţină informaţii pentru fiecare poziţie a ţintei. Ca urmare, se reţine o matrice de valori euristice, reprezentând funcţia $h(x,y)$, pentru toate perechile de stări x şi y. Conceptual, toate valorile euristice sunt citite din această matrice, care este iniţializată cu valori returnate de funcţia statică de evaluare. Pe parcursul căutării, aceste valori euristice sunt actualizate pentru a le creşte acurateţea. În aplicarea acestui algoritm reţinem doar acele valori care diferă de cele statice. Astfel, chiar dacă matricea completă este de dimensiuni foarte mari, ea este o matrice rară.

Există două evenimente diferite care au loc pe parcursul acestui algoritm, acestea sunt: mişcarea agentului care rezolvă problema (caută) şi o deplasare a ţintei. Fiecare dintre aceste evenimente poate fi însoţit de actualizarea unei valori euristice. Se presupune că agentul şi ţinta se mişcă alternativ, şi nu pot traversa mai mult de o muchie la o deplasare. Agentul nu are nici o informaţie despre modul în care se deplasează ţinta şi nici nu poate prezice, măcar probabilistic, traiectoria acesteia. Scopul este atins când cei doi ocupă acelaşi nod.

În descrierea următoare avem, pe lângă notaţiile din prezentarea algoritmului de programare dinamică asincronă, notaţiile: x_i şi x_j reprezintă poziţia curentă, respectiv poziţia vecină a agentului, iar y_i şi y_j reprezintă poziţia curentă, respectiv cea vecină a ţintei. Pentru a simplifica problema presupunem că toate muchiile grafului au costul egal cu o unitate.

Următoarea procedură este executată când agentul care rezolvă problema se deplasează:

1. calculează $h(x_j,y_i)$ pentru fiecare vecin x_j al lui x_i;
2. actualizează valoarea lui $h(x_i,y_i)$, astfel:
$$h(x_i,y_i)= \max\{h(x_i,y_i), \min\{h(x_j,y_i) +1\}\}$$
3. deplasare pe poziţia vecină x_j, care are cea mai mică valoare pentru $h(x_j,y_i)$, atribuie lui x_i valoarea lui x_j.

Următoarea procedură este executată când ţinta se deplasează:

1. calculează $h(x_i, y_j)$ pentru noua poziţie y_j a ţintei
2. actualizează valoarea lui $h(x_i,y_i)$, astfel:
$$h(x_i,y_i)= \max\{h(x_i,y_i), h(x_i,y_j) +1\}$$
3. actualizează noul scop al agentului care rezolvă, ca fiind noua poziţie a ţintei atribuindu–i lui y_i valoarea lui y_j.

Un agent care execută algoritmul MTS are garanţia că îşi va atinge ţinta. Dovada acestei afirmaţii este faptul că MTS este o extensie a LRTA*. Se extinde o proprietate a acestuia din urmă, obţinându–se următoarea proprietate pentru MTS: într–un spaţiu finit, cu muchii de cost pozitiv şi egale cu o unitate, în care există cel puţin un drum de la fiecare nod la un nod final, pornind cu valori euristice iniţiale admisibile şi nenegative, acceptând deplasări în ambele direcţii de–a lungul unei muchii pentru toţi agenţii (ţinta şi cel care rezolvă problema), un agent care execută algoritmul MTS va ajunge la ţintă, dacă acesta periodic nu efectuează deplasări.

5.2.2.5 Algoritmul de căutare bidirecţională în timp real (Real time bidirectional search - RTBS)

Mişcarea ţintei le cere agenţilor să se adapteze schimbărilor scopurilor lor. Acest lucru ne permite să studiem diferite organizări ale agenţilor. Să presupunem că există doi agenţi într–un spaţiu complex al labirintului. Aceştia sunt proiectaţi să aibă ca scop principal întâlnirea lor. Fiecare dintre agenţi ştie întotdeauna poziţia sa curentă în labirint şi poate comunica cu celălalt robot. Astfel fiecare agent cunoaşte poziţia ţintei (la care trebuie să ajungă). Agenţii nu au o hartă a labirintului, dar se pot informa despre lumea înconjurătoare prin intermediul a diferiţi senzori. Pentru a avea o imagine cât mai completă a lumii este necesar ca agenţii să se poată deplasa, nu doar să îşi schimbe starea. În această situaţie dată care ar trebui să fie comportamentul agenţilor pentru ca aceştia să se întâlnească în mod eficient? Ar trebui să negocieze pentru a decide următoarea acţiune sau ar trebui să acţioneze independent? Este într–adevăr o organizare care utilizează doi agenţi superioară uneia care uitlizează unul singur? Acestea sunt doar câteva întrebări la care un proiectant al unor agenţi care rezolvă o problemă ar trebui să răspundă.

Spre deosebire de ceilalţi algoritmi prezentaţi până acum, algoritmul RTBS implică doi agenţi care încearcă să rezolve împreună o problemă. În algoritmul RTBS doi agenţi pornind din starea iniţială, respectiv starea ţintă se deplasează unul spre celălalt. Spre deosebire de varianta care nu implică o rezolvare în timp real, în algoritmul considerat, costul pentru obţinerea coordonării se aşteaptă să fie limitat, raportat la o perioadă limitată de timp. De vreme ce şi timpul pentru planificare este limitat eficienţa agenţilor poate fi mai redusă.

În executarea algoritmului RTBS, vor fi repetaţi paşii următoarei proceduri până când cei doi agenţi se întâlnesc în spaţiul de rezolvare a problemei:

1. ***strategie de control***
   Selecteză o deplasare înainte (pasul 2), sau o deplasare înapoi (pasul 3);
2. ***deplasare înainte(avans)***
   Agentul care se află în starea iniţială se deplasează spre agentul care se află în starea finală;
3. ***deplasare înapoi(revenire):***
   Agentul care se află în starea finală se deplasează spre agentul care se află în starea iniţială.

Algoritmii bazaţi pe RTBS pot fi împărţiţi în două categorii, în funcţie de autonomia agenţilor care rezolvă problema. Prima categorie se numeşte *RTBS centralizat*, din aceasta fac parte algoritmii în care este aleasă cea mai bună acţiune dintre toate acţiunile posibile ale celor doi agenţi. Cea de–a doua categorie se numeşte *RTBS descentralizat* şi ea fac parte algoritmi în care fiecare dintre cei doi agenţi ia decizii independent de deciziile celuilalt.

Pentru a explica mai bine mecanismul acestor tipuri de algoritmi aceştia vor fi prezentaţi ca metodă de rezolvare comparativ pentru n–puzzle. Analizăm din punct de vedere practic, dacă o abordare descentralizată este de preferat în locul uneia centralizate

În cazul problemei n–puzzle algoritmul de căutare unidirecţională în timp real utilizează un singur puzzle şi îmbină planificarea cu execuţia, alegând şi executând cea mai bună acţiune posibilă determinată în urma analizei tuturor acţiunilor posibile. Spre deosebire de acesta algoritmul RTBS utilizează două puzzle–uri, unul indicând iniţial starea de la care se porneşte iar celălat indică la momentul iniţial starea finală.

În RTBS centralizat se acţionează în aşa fel încât s–ar putea crede că un singur om rezolvă ambele puzzle–uri, pe când în RTS descentralizat se acţionează ca şi cum fiecare

puzzle este rezolvat de către un alt om. În RTBS centralizat stategia de control adoptată este alegera celei mai potrivite acţiuni dintre toate acţiunile posibile (şi cele ale primului şi cele ale celui de–al doilea agent, deplasări înainte şi înapoi) pentru a minimiza distanţa estimată până la starea finală. Se pot implementa doi algoritmi de RTBS centralizat bazaţi pe LRTA*, respectiv RTA*. În RTBS descentralizat strategia de control are ca sarcină doar alegerea uneia dintre alternativele: deplasare înainte sau deplasare înapoi. Astfel fiecare dintre agenţi ia decizii independente bazate pe informaţiile euristice proprii. În implementarea algoritmilor bazaţi pe RTBS descentralizat poate fi utilizat algotitmul MTS, atât pentru deplasarea înainte cât şi pentru deplasarea înapoi.

La o evaluare a rezultatelor practice (implementări pentru rezolvarea n–puzzle) se observă că în situaţii clare, în care funcţiile euristice returnează valori reale, RTBS dscentralizat are rezultate mai bune, iar în anumite situaţii particulare RTBS centralizat este mai eficient.

Pentru a observa diferenţa dintre o abordare unidirecţională şi una bidirecţională vom testa doi algoritmi, algoritmul bidirecţional de căutare în timp real (algoritmul RTBS) şi algoritmul unidirecţional de căutare întimp real, pentru două probleme din categorii diferite, n-puzzle şi problema labirintului. În cazul n–puzzle algoritmul RTBS ecte mai eficient decât varianta unidirecţională, dar în cazul labirintului varianta unidirecţională are rezultate mult mai bune decât algoritmul RTBS.

Rezultatele acestui test conduc la următoarea concluzie în căutarea bidirecţională: agenţii se deplasează într–un spaţiu combinat al problemei, ceea ce face ca erorile introduse de valori euristice greşite să crească exponenţial şi să scadă dramatic şansele de obţinere a unei soluţii optime. Aşadar, se poate spune că performanţaunei căutări în timp real nu este sensibilă atât la numărul de agenţi utilizaţi cât la topologia spaţiului de căutare şi în special la *depresia euristică*.

Deci revenind la problema iniţială în care doi genţi trebuie să se întâlnescă, adoptând strategia descentralizată, în care agenţii se deplasează independent unul spre celălalt, observăm că ne lovim de ineficienţa agenilor, pentru a rezolva acestă problemă adoptăm o nouă strategie, cea centralizată, în care se decide de comun acord care este agentul care se deplasează la pasul următor, dar nici în acest caz nu avem rezultate bune, agenţii necesitând coordonare.

Cea mai mare problemă în modul de abordare propus de acest algoritm este faptul că agenţii nu percep schimbările ce se produc în spaţiul lor de căutare.

5.2.2.6 Algoritmul de căutare multiagent în timp real (Real Time Multiagent Search - RTMS)

Chiar şi în cazul în care există doar doi agenţi RTBS nu este singura metodă de organizare a agenţilor care încearcă să rezolve problema. O altă metodă posibilă de organizare este să pornească ambii agenţi din starea iniţială şi să se îndrepte spre cea finală. Se observă că este mult mai natural să adoptăm spaţiul problemei propus în descrierea RTBS având această organizare.

Dacă există mai mulţi agenţi, cum pot aceştia să coopereze pentru a rezolva problema? Cheia proiectării acestor agenţi constă în selectarea unei organizări adecvate a agenţilor. Este evident că numărul organizărilor posibile este mare, deci vom porni cu cea mai simplă dintre ele: agenţii vor partaja acelaşi spaţiu de căutare cu o ţintă fixă comună.

Fiecare agent va executa algoritmul LRTA*, independent de ceilalţi, dar vor partaja valorile actualizate ale lui *h* (acest algoritm poartă numele de Multiagent LRTA*). În acest caz în momentul în care unul dintre agenţi atinge ţinta problema este rezolvată pentru toţi agenţii.

Pentru a stabili eficienţa acestui tip de organizare vom analiza următoarele două efecte:

- **Efectul de partajare a experienţei între agenţi.**

  Cum selecţia nodurilor pentru care se vor executa calcule este aleatoare în algoritmul de programare dinamică asincronă, această proprietate se regăseşte şi în algoritmul LRTA*. Ruperea aleatoare a legăturilor asigură faptul că, deşi pornesc din acelaşi nod iniţial şi partajează valorile funcţiei *h*, pe parcursul execuţiei algoritmului poziţiile curente ale agenţilor se vor dispersa.
- **Efectul luării deciziei independent.**

  Faptul că sunt utilizaţi mai mulţi agenţi pote fi un avantaj în rezolvarea de probleme în care decizii critice sunt necesare. Se poate afirma acest lucru deoarece în cazul utilizării unui singur agent, la momentul alegerii unei opţiuni dintre două posibile, probabilitatea ca acesta să ia decizia corectă este de 50%, pe când în cazul utilizării a doi agenţi probabilitatea ca decizia corectă să fie luată de unul dintre ei la momentul în care ajunge în acel punct al rezolvării problemei este de 75% (3/4, cazuri favorabile/cazuri posibile). Acest rezultat se poate genera pentru k agenţi obţinând o probabilitate de succes a acestora egală cu $1-1/2^k$.

Rezolvând o problemă cu ajutorul mai multor agenţi care acţioneză concurent poate creşte atât robusteţea cât şi eficienţa metodei de rezolvare. Există mai multe tehnici de creare a diferite organizări ale agenţilor. De exemplu se poate descompune scopul problemei în mai multe scopuri pentru diferiţi agenţi, sau atribuind diferiţilor agenţi diverşi operatori potriviţi pentru îndeplinirea scopului lor.

Deşi căutarea în timp real oferă o posibilitate atractivă de rezolvare a problemelor, în ceea ce priveşte resursele folosite, algoritmii dezvoltaţi până acum în acest domeniu au încă multe neajunsuri. În primul rând comportamentul agentului care rezolvă problema nu este destul de raţional pentru a fi însuşit de agenţi autonomi. Agentul prezentat în aceşti algoritmi tinde să execute acţiuni în plus înainte de a se concentra pe atingerea scopului, nu poate utiliza şi îmbunătăţi experienţele anterioare, nu se poate adapta unei schimbări dinamice a scopului şi nu poate coopera eficient cu alţi astfel de agenţi în rezolvarea diferitelor probleme.

Tehnicile de căutare în timp real furnizează o bază solidă pentru studierea ulterioară a problemei organizării în mediile multiagent dinamice şi nesigure.

5.3 Jocuri cu 2 jucători

Jocurile cu doi jucători se referă la situaţiile în care există 2 agenţi competitivi. ca urmare, legătura cu Inteligenţa Artificială Distribuită este evidentă, fiind vorba de sisteme multiagent în care agenţii sunt competitivi.

Un joc poate fi definit formal ca o problemă de **căutare** cu următoare componente:

- *starea iniţială*, care include poziţia tablei de joc şi o indicaţie referitor la care dintre jucători este primul la mutare;

- *o mulţime de operatori*, care definesc mutările legale pe care un jucător le poate face;
- un *test final (terminal)*, care determină când jocul este terminat. Stările jocului când acesta a ajuns la sfârşit se numesc **stări finale (terminale)**;

Secvenţa posibilelor mutări ale celor doi jucători poate fi descrisă sub forma unui arbore, aşa numitul **arbore de joc**. Este vorba de arborele descris în Figura 5.1 în care alternează două nivele: un nivel corespunzător jucătorului care începe jocul şi care va fi numit jucător **MAX** şi un nivel corespunzător oponentului său, pe care îl vom numi jucătorul **MIN**.

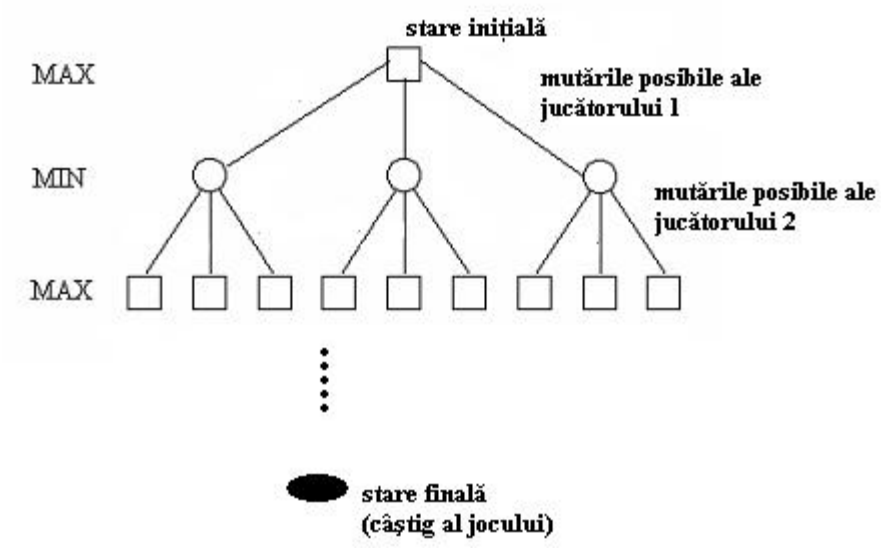

Figura 5.1 Arbore de joc

Din Figura 5.1 rezultă faptul că rezolvare jocului ar putea fi privită ca o problemă de căutare într-un spaţiu de stări. Dar complexitatea spaţiului stărilor este foarte mare pentru jocuri complexe, ca urmare o căutare directă în spaţiul stărilor ar fi practic imposibilă.

În continuare vom justifica complexitatea exponenţială a spaţiului stărilor unui arbore de joc. Presupunând că arborele ar fi complet, că numărul mutărilor posibile pentru jucători ar fi **b**, iar adîncimea la care s-ar câştiga jocul ar fi **d**, se poate simplu arăta că numărul total de noduri din acest arbore de joc ar fi $\mathbf{b^d}$, ca urmare complexitatea atât ca spaţiu de memorare cât şi ca timp de execuţie necesitat de o căutare în acest arbore de joc ar fi $\mathbf{O(b^d)}$.

Problema esenţială în teoria jocurilor este cea de luare a *deciziei* în efectuarea unei mutări. Prezenţa unui adversar face ca deciziile problemei să fie mai complicate ca şi alte probleme de căutare din spectrul Inteligenţei Artificiale. Adversarul introduce *incertitudinea*, pentru că nu ştim niciodată ceea ce ea sau el va urma să facă (în cele mai bune cazuri putem doar anticipa acest lucru).

Ceea ce face jocurile cu adevărat interesante este faptul că de obicei acestea sunt foarte greu de rezolvat. Şahul, de exemplu, are un *factor mediu de ramificare* (prin factor de ramificare se înţelege numărul de mutări care sunt posibile dintr-o anumită

poziţie respectiv numărul de arce care ies dintr-un nod în terminologia arborilor de joc) de aproximativ 35 iar uneori jocul ajunge până la 50 de mutări pentru fiecare jucător, deci arborele de căutare ar avea 35^{100} noduri (dintre care sunt doar 10^{40} sunt poziţii legale diferite). Tic-Tac-Toe (X şi O) este un joc plictisitor pentru adulţi pentru că este uşor de determinat mutarea bună. *Complexitatea* jocurilor introduce o nouă formă de incertitudine care apare nu datorită lipsei de informaţii ci pentru că timpul nu permite calcularea consecinţelor fiecărei mutări. În schimb, jucătorul mai trebuie să facă cea mai bună presupunere bazată pe experienţa anterioară, şi să acţioneze înainte ca el sa fie sigur de decizia pe care trebuie s-o ia. În această privinţa, jocurile sunt mai apropriate de lumea reală decât problemele de căutare standard ale Inteligenţei Artificiale.

Problema complexităţii apare în multe domenii ale Inteligenţei Artificiale, jocurile fiind o zonă de vârf în acest sens. Pentru că de obicei există limită de timp, jocurile penalizează foarte sever ineficienţa . Cercetările din domeniul jocurilor au mărit numărul de idei despre cum să se utilizeze cel mai bine timpul pentru luarea deciziilor bune în cazul în care luarea deciziilor optimale este imposibilă.

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

Pentru a îmbunătăţi eficienţa unui program de rezolvare a problemelor bazat pe căutare trebuie realizate două lucruri:

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

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

Desigur, căutarea nu este singura tehnică avută în vedere. În unele jocuri, în anumite momente sunt disponibile tehnici mai directe. De exemplu, în şah deschiderile şi închiderile sunt stilizate, şi deci este mai bine să fie jucate prin căutare într-o bază de date de modele memorate. Pentru a juca un joc, trebuie să combinăm tehnicile orientate pe căutare cu cele mai directe. Modalitatea ideală de a folosi o procedură de căutare pentru a găsi o soluţie a unei probleme este de a genera deplasări în spaţiul problemei până când se atinge o stare finală. Pentru jocuri complexe, deseori nu este posibil să căutăm până la identificarea unei stări câştigătoare, chiar dacă folosim un generator bun de mutări plauzibile. Adâncimea şi factorul de ramificare ale arborelui generat sunt prea mari. Totuşi este posibil să căutăm într-un arbore pe o anumită adâncime, de exemplu doar 10 sau 20 de mutări. Apoi, pentru a selecta cea mai bună mutare, stările generate trebuie comparate pentru a o descoperi pe cea mai avantajoasă. Aceasta se realizează cu o **funcţie de evaluare statică (FES)**, care foloseşte informaţiile disponibile pentru a evalua

stările individuale prin estimarea probabilităţii acestora de a conduce la câştigarea jocului.

Această funcţie este similară cu funcţia *h'* a algoritmului **A***. Funcţia de evaluare statică poate fi aplicată poziţiilor generate de mutările propuse. Din cauza dificultăţii construirii unei astfel de funcţii care să fie şi foarte corectă, este de preferat să generăm câteva nivele din arborele stărilor şi abia atunci să aplicăm funcţia.

Este foarte important modul în care construim funcţia de evaluare statică. De exemplu, pentru jocul de şah, o funcţie de evaluare statică foarte simplă se bazează pe avantajul pieselor: calculează suma valorilor pieselor albe (A), a pieselor negre (N) şi calculează raportul A/N. O variantă mai sofisticată este o combinaţie liniară a mai multor factori importanţi, printre care valoarea pieselor, posibilitatea de a avansa, controlul centrului, mobilitatea. Au fost construite şi funcţii neliniare. O posibilitate de a îmbunătăţi o funcţie bazată pe ponderarea unor factori este ca decizia în legătură cu ponderile factorilor să se bazeze pe experienţa din jocurile anterioare: ponderile factorilor care au dus la câştig vor creşte, cele ale factorilor care dus la înfrângere vor scădea.

Dar trebuie să luăm în calcul şi posibilitatea ca victoria noastră să fie cauzată de faptul că inamicul a jucat prost, nu că noi am fi jucat bine. Problema deciziei referitoare la care dintre o serie de acţiuni este responsabilă pentru un anumit rezultat se numeşte **problema atribuirii creditului** (*credit assignment*).

Am discutat despre două componente ale unui program de joc: un generator de mutări plauzibile bun şi o funcţie de evaluare statică bună. Pe lângă acestea avem nevoie şi de o procedură de căutare care permite analizarea a cât de multe mutări din spaţiul stărilor, înainte ca acestea să fie efectiv operate. Avem la dispoziţie mai multe strategii:

- Algoritmul **A***, pentru care funcţia *h'* este aplicată nodurilor terminale, şi valorile sunt propagate înapoi către vârful arborelui. Procedura nu este adecvată pentru jocuri deoarece jocurile sunt operate de două persoane adversare.
- Procedura **Minimax**, care se bazează pe alternanţa jucătorilor. În arborele stărilor, un nivel al stărilor generate de mutările unui jucător va fi urmat de un nivel al stărilor generate de mutările adversarului său. Trebuie să avem în vedere interesele opuse ale celor doi jucători.
- Algoritmul **B***, aplicabil atât arborilor standard de rezolvare a problemelor, cât şi arborilor de jocuri.

5.3.1 Funcţii de evaluare statice

O *funcţie de evaluare* returnează o estimare a utilităţii aşteptate a jocului din poziţia dată. O funcţie de evaluare statică bine aleasă trebuie să satisfacă următoarea condiţie esenţială pentru o stragie de joc bună: suma dintre estimarea statică a unei stări ***s*** din punct de vedere al jucătorului **MAX** şi estimarea statică a stării ***s*** din punct de vedere al jucătorului **MIN** trebuie să fie egală cu 0.

Jocul de şah

De secole, jucătorii de şah (la fel ca şi jucătorii altor jocuri) au dezvoltat multe căi de a judeca şansele de câştig din fiecare latură bazându-se pe calcularea poziţiei. De

exemplu în cărţile introductive despre şah dau o valoare materială fiecărei piese: fiecare pion valorează 1, nebunul valorează 3, tura 5 şi regina 9. Alte puncte ca şi "structura bună a pionului" şi "securitatea regelui" ar putea să valoreze jumătate de pion. Toate celelalte lucruri sunt egale, o latură care are un avantaj material a unui pion sau mai mult va câştiga probabil jocul, chiar şi un avantaj de 3 puncte este suficient pentru o apropiată victorie.

Este clar că performanţa programului de joc este dependentă de calitatea funcţiei ei de evaluare. Dacă este greşită, atunci programul va ajunge la poziţii care par bune, dar de fapt acestea sunt dezastroase.

Pentru jocul de şah o posibilă funcţie de evaluare este cea numită *funcţie liniară de cantitate* pentru că poate fi exprimată ca:

$$w_1f_1+ w_2f_2+.....+ w_nf_n$$

unde w_i ($1\le i\le n$) sunt cantităţi, iar f_i ($1\le i\le n$) sunt calităţile unei poziţii particulare.

w – va fi valoarea pieselor (1-pion, 3-nebun, etc.)

f – reprezintă numărul fiecărui fel de piese de pe tablă

Cele mai multe programe de jocuri folosesc o funcţie de evaluare liniară, deşi recent funcţiile neliniare au avut un mai mare grad de succes. În construirea formulei liniare trebuie să se aleagă calităţile şi apoi să se corecteze cantităţile până când programul joacă bine. Această ultimă sarcină poate fi automatizată printr-un program care să joace multe jocuri cu el însuşi dar până acum nimeni nu a avut o idee bună despre cum să se aleagă în mod automat calităţile.

Jocul TIC-TAC-TOE

Pentru jocul TIC-TAC-TOE, în scopul asigurării unei strategii de joc bune se pot folosi următoarele funcţii de evaluare statică (presupunem că jucătorul **MAX** joacă cu **X**):

- $FES(s, MAX) = \left(\sum_{i\in D_1} 1+\sum_{i\in D_2} 2\right) - \left(\sum_{i\in D_1'} 1+\sum_{i\in D_2'} 2\right)$ unde:
  - D_1 este mulţimea liniilor, coloanelor sau diagonalelor în care **MAX** are un **X** şi poate câştiga;
  - D_2 este mulţimea liniilor, coloanelor sau diagonalelor în care **MAX** are doi **X** şi poate câştiga;
  - $D_1^{'}$ este mulţimea liniilor, coloanelor sau diagonalelor în care **MIN** are un **0** şi poate câştiga;
  - $D_2^{'}$ este mulţimea liniilor, coloanelor sau diagonalelor în care **MIN** are doi **0** şi poate câştiga.
- $FES(s, MAX) =$ (numărul liniilor, coloanelor sau diagonalelor complete care sunt deschise pentru **MAX**)- (numărul liniilor, coloanelor sau diagonalelor complete care sunt deschise pentru **MIN**).
- $FES(s, MAX) = 3\cdot X_2 + X_1 - (3\cdot O_2 + O_1)$ unde:
  - X_n este numărul liniilor, coloanelor sau diagonalelor cu *n* **X**-uri şi fără **0**;
  - O_n este numărul liniilor, coloanelor sau diagonalelor cu *n* **0**-uri şi fără **X**.

5.3.2 Procedura de căutare MINIMAX

Abordarea minimax se bazează pe presupunerea că la orice moment, cei doi jucători vom alege mutarea optimă, altfel spus joacă fără greşeală.

Procedura de căutare Minimax este o procedură de căutare Depth-First limitată. O funcţie de evaluare va returna o valoare pentru fiecare nod, un nod favorabil jucătorului **MAX** având o valoare de evaluare mare, pe când un nod favorabil jucătorului **MIN** are o valoare de evaluare mică.

Ideea este să începem la poziţia curentă şi să utilizăm generatorul de mutări plauzibile pentru a genera mulţimea poziţiilor succesori. Acum putem aplica acestor poziţii funcţia de evaluare statică şi putem selecta poziţia cea mai promiţătoare. Valoarea acestei poziţii poate fi trecută ca valoare a poziţiei iniţiale, deoarece poziţia iniţială este la fel de bună ca cea mai bună poziţie produsă.

Vom presupune că funcţia de evaluare statică produce valori cu atât mai mari cu cât mai bună este poziţia pentru noi (jucătorul **MAX**). Astfel, în această fază scopul nostru este să **maximizăm** valoarea produsă de funcţie.

Dar cum ştim că funcţia de evaluare statică nu este absolut corectă, am dori să mergem cu căutarea mai departe. Dorim să vedem ce se întâmplă cu fiecare dintre poziţiile produse după ce adversarul va face o mutare. Pentru aceasta, în loc să aplicăm funcţia de evaluare statică poziţiilor generate de noi, vom aplica generatorul de mutări plauzibile pentru a genera câte o mulţime de mutări bune ale adversarului pentru fiecare poziţie generată de noi. Dacă dorim să ne oprim acum, după două mutări, putem aplica funcţia de evaluare statică. Dar acum trebuie să avem în vedere că adversarul decide mutarea pe care o va face şi valoarea care va fi transferată poziţiilor de pe nivelul superior. Scopul lui este să **minimizeze** valoarea funcţiei. Prin urmare, pentru fiecare mutare candidat a noastră, vom putea identifica o mutare candidat a adversarului. Concluzia care se impune este ca vom prefera să facem acea mutare pentru care mutarea candidat a adversarului (cea mai bună din punctul său de vedere) va produce poziţia cea mai bună din punctul nostru de vedere. La nivelul deciziei adversarului, întotdeauna a fost selectată şi trimisă spre nivelul superior valoarea minimă. La nivelul deciziei noastre, întotdeauna a fost selectată şi trimisă spre nivelul superior valoarea maximă. Figura 5.2 prezintă un exemplu de propagare a valorilor în arbore.

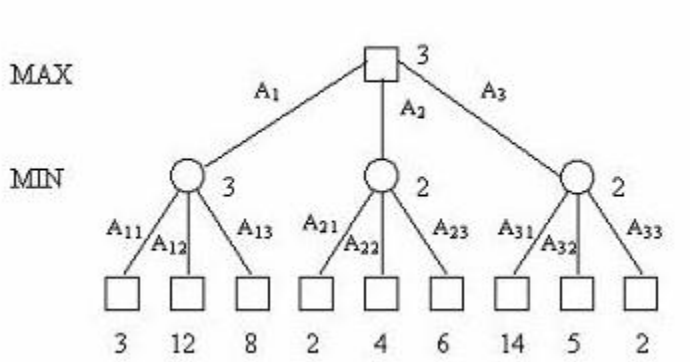

Figura 5.2. Arbore de joc generat cu algoritmul minimax

Ca urmare a celor descrise anterior, evaluarea fiecărui nod se poate face recursiv, în felul următor:

- valoarea unui nod **MAX** este egal cu valoarea maximă a fiilor săi;
- valoarea unui nod **MIN** este egală cu valoarea minimă a fiilor săi.

O problemă critică a procedurii Minimax este de a decide când să oprim recursivitatea şi să apelăm funcţia de evaluare statică. Dintre factorii care influenţează această decizie, amintim următorii:

- A câştigat una dintre părţi?
- Câte niveluri de mutări am explorat?
- Cât de promiţător este drumul curent?
- Cât de mult timp a rămas?
- Cât de stabilă este configuraţia?

În multe jocuri decizia de a opri explorarea arborelui de joc este legată de o adâncime fixată anterior, deşi această decizie ar putea conduce la strategii incorecte în multe situaţii. Ca urmare, decizia legată de oprirea expandării este mult mai complexă şi de multe ori se folosesc tehnici care îmbunătăţesc performanşa MINIMAX cum ar fi: căutarea secundară sau aşteptarea pasivă.

5.3.3 MINIMAX cu tăieturi alfa-beta

Un lucru bun în legătură cu tehnicile Depth-First este acela că eficienţa lor poate fi îmbunătăţită prin utilizarea tehnicilor branch-and-bound în care soluţii parţiale care sunt în mod clar mai slabe decât soluţii cunoscute pot fi abandonate mai repede. Dar, cum procedura tradiţională Depth-First a fost modificată pentru a lua în considerare pe ambii jucători, este de asemenea necesar să modificăm strategia Branch-and-Bound pentru a include două limite, pentru ambii jucători: o limită superioară şi una inferioară. Această strategie modificată se numeşte **trunchiere alfa-beta**.

Ideea tăieturilor este de a tăia (a nu explora) acele părţi ale arborelui de joc care nu influenţează valoarea (evaluarea) rădăcinii arborelui. Mai precis, următoarele valori sunt calculate şi actualizate:

- *valoarea* α - limita minimă a valorii de evaluare a unui nod **MAX**;
- *valoarea* β - limita maximă a valorii de evaluare a unui nod **MIN**.

Pe măsura explorării arborelui de joc şi vizitării nodurilor sale, pornind de la rădăcină şi efectuând o căutare Depth-First până la o anumită adâncime, valorile α şi β sunt aclualizate folosind următoarele reguli:

- valoarea α a unui nod **MAX** este maximul dintre valorile fiilor săi vizitaţi până în acel moment;
- valoarea β a unui nod **MIN** este minimul dintre valorile fiilor săi vizitaţi până în acel moment.

Se va putea tăia căutarea (expandarea) unei părţi din arborele de joc în momentul în care $\alpha \geq \beta$, altfel spus dacă apare una din situaţiile următoare:

- **tăietura α** - dacă valoarea β a unui nod **MIN** este mai mică sau egală cu cea mai mare valoare α a strămoşilor săi **MAX**, putem tăia partea din arborele de căutare de sub nodul **MIN**;
- **tăietura β** - dacă valoarea α a unui nod **MAX** este mai mare sau egală cu cea mai mică valoare β a strămoşilor săi **MIN**, putem tăia partea din arborele de căutare de sub nodul **MAX**.

În cele ce urmează vom schiţa procedura MINIMAX-α-β. La un anumit nivel ***niv*** în arborele de joc se efectuează următoarele operaţii:

1. Verifică dacă ***niv*** este nivelul de început, depăşeşte nivelul maxim admis sau este un nivel **MAX** sau **MIN**
   - 1a. Dacă este nivelul de început, atunci $\alpha \leftarrow -\infty$, $\beta \leftarrow \infty$
   - 1b. Dacă este nivelul maxim admis, atunci se calculează valoarea funcţiei de evaluare statice pentru nod şi se returnează rezultatul
   - 1c. Dacă este un nivel **MIN** atunci
     - 1c1. Până când toţi fiii sunt examinaţi cu MINIMAX sau până când $\alpha \geq \beta$
       - 1c1.1 $\beta \leftarrow$min{β, valoarea minimă raportată de MINIMAX pentru acest fiu}
       - 1c1.2 Foloseşte MINIMAX pentru următorul fiu al poziţiei curente, cu noile valori ale lui α şi β
     - 1c2. Returnează β
   - 1d. Dacă este un nivel **MAX** atunci
     - 1c1. Până când toţi fiii sunt examinaţi cu MINIMAX sau până când $\alpha \geq \beta$
       - 1c1.1 $\alpha \leftarrow$min{α, valoarea maximă raportată de MINIMAX pentru acest fiu}
       - 1c1.2 Foloseşte MINIMAX pentru următorul fiu al poziţiei curente, cu noile valori ale lui α şi β
     - 1c2. Returnează α

Menţionăm că varianta MINIMAX-α-β reduce rata exploziei combinatoriale, dar nu o previne.

Există cîteva îmbunătăţiri ale procedurii MINIMAX, ăn scopul îmbunătăţirii performanţei acesteia. Le vom enumera, fără a intra în amănunte:

- aşteptarea pasivă (*waiting for quiescence)*;
- căutarea secundară (*secondary searching*);
- folosirea unor arhive de mutări.

Alternative la tehnica MINIMAX pentru rezolvarea jocurilor sunt următoarele:

- Adâncirea Iterativă (*Iterative Deepening*);
- Adâncirea Iterativă de tip A* (*A* Iterative Deepening*);
- algoritmul B*.

5.3.4 Jocuri care includ şansa

În viaţa reală, pe lângă şah, mai sunt multe alte evenimente externe care ne pun în situaţii neplăcute. Multe jocuri reflectă acest lucru deoarece acestea implică elemente aleatoare (*elemente de şansă*) cum ar fi aruncatul zarurilor. Pe această cale, ne apropriem mai mult de realitate, şi este foarte interesant de văzut cum acest lucru afectează deciziile luate în procesul de căutare.

Jocul de table este un joc tipic care include elemente de şansă. Mutările legale ale unui jucător intr-un joc de table depind de aruncarea zarului. Un jucător ştie atât mutările legale ale lui cât şi ale adversarului dar nu ştie ce zar va arunca acesta din urmă şi deci ce mutare va face. Construirea un arbore complet de joc ca pentru Tic-Tac-Toe sau şah este deci imposibilă.

Un arbore pentru un jocurile includ un element şansa trebuie să conţină pe lângă nodurile MAX şi MIN *noduri de şansă*. Alternanţa la mutare a lui MAX şi MIN implică câte o aruncare cu zarul pentru fiecare. DICE poate fi considerat ca un jucător fictiv care apare la fiecare aruncare de zaruri şi face o mutare care este determinată de şansă. Un arbore schematic pentru table este reprezentat în Figura 5.3, nodurile de şansă fiind reprezentate prin triunghiuri. Ramurile care pornesc din fiecare nod de şansă reprezintă valorile care ar putea să apară la aruncatul zarurilor şi vor fi etichetate cu aceste valori şi cu şansa (probabilitatea) lor de apariţie. La aruncarea a două zaruri pot apare 36 moduri de distribuţie a valorilor inscripţionate pe feţele acestora dar, având în vedere faptul că 6-5 este la fel ca 5-6, există doar 21 de moduri diferite de distribuţie. Sunt şase duble (de la 1-1 până la 6-6 cu şansa de 1/36 să apară) şi alte 15 moduri distincte cu o şansă de 1/18.

Pentru a lua decizia corectă în asemenea jocuri normal trebuie aleasă din setul de mutări din setul de mutări $A_1,...,A_n$ mutarea care conduce la cea mai bună soluţie. Poziţiile nu au însă definită o valoare "minimax", cum în jocurile deterministe există utilitatea nodurilor terminale care indică jucătorului care este ceea mai bună mutare, deci valoarea nu poate fi calculată exact . Totuşi se poate calcula o medie (*valoare aşteptată*) care să reprezinte toate posibilităţile care pot să apară pe feţele zarurilor.

Valoarea aşteptată este destul uşor de calculat. Pentru nodurile terminale se poate folosi funcţia utilitate ca şi în jocurile deterministe. Un nivel mai sus în arbore se găsesc nodurile de şansă. În cazul în care considerăm nodul de şansă etichetat cu C atunci fie d_i o distribuţie posibilă a zarurilor şi $P(d_i)$ probabilitatea de a obţine această distribuţie. Pentru fiecare distribuţie posibilă, calculăm utilitatea celei mai bune mutări pentru MIN, şi se transmit la nivelul superior utilităţile Acestea sunt influenţate de şansă datorită faptului că o distribuţie particulară a fost obţinută. Notând cu $S(C,d_i)$ mulţimea tuturor poziţiilor generate prin mutările legale pentru distribuţia zarului $P(d_i)$ în poziţia C se poate calcula aşa numita *valoare "expectmax"* după formula:

$$expectmax(C) = \sum_i P(d_i) \cdot \max_{S \in S(C,d_i)} (utilitate(S))$$

Această formală calculează utilitatea aşteptată a poziţiei pentru cel mai bun joc. Mergând mai sus cu un nivel la nodurile lui MIN se poate aplica formula normală a lui minimax pentru că nodurilor de şansă le-au fost atribuite utilităţile. Când se ajunge din nou la nodurile de şansă (nodul B din Figura 5.3.) se procedează analog dar în poziţia B se calculează aşa numita *valoare "expectmin"* după formula:

$$expectmin(B) = \sum_i P(d_i) \cdot \min_{S \in S(C,d_i)} (utilitate(S))$$

Acest procedeu poate fi aplicat recursiv pentru toate nivelele arborelui cu excepţia rădăcinii unde distribuţia zarurilor este deja cunoscută. Procedeul este similar cu procedura MINIMAX descrisă pentru jocuri care nu includ şansa.

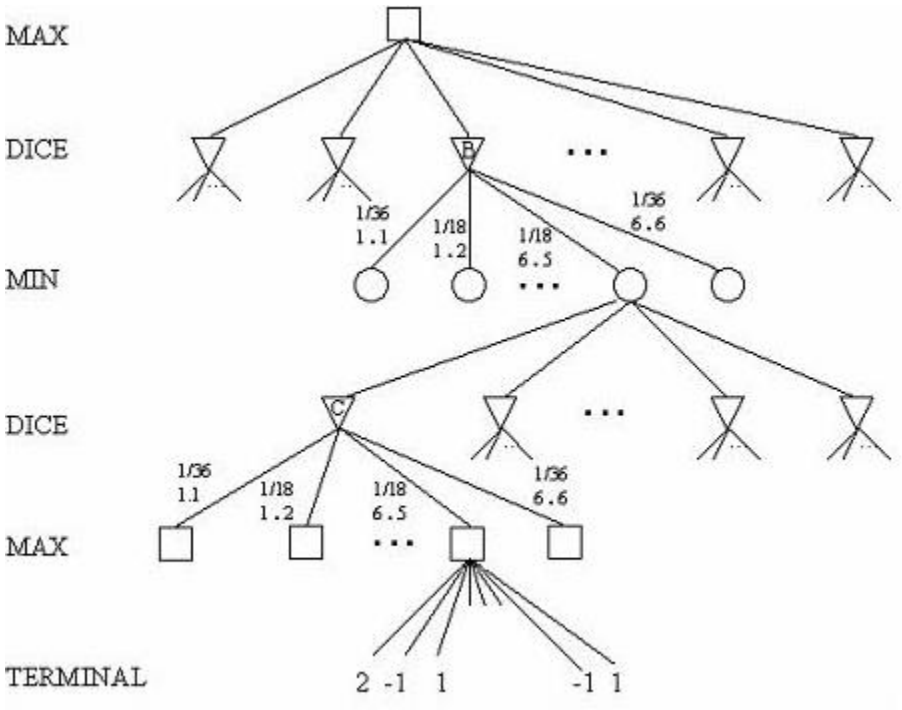

Figura 5.3. Arbore schematic pentru un joc de table