Introducere în domeniul Inteligenței Artificiale

Inteligența artificială este un domeniu nou, apărut după Cel de-al Doilea Război Mondial (1945), cu scopul de a construi mașini inteligente. Problema inteligenței și a modului de a gândi a fost discutată de mii de ani, fiind implicați filozofi, medici neurologi sau psihologi. Apariția calculatorului (precedata de aparitia mașinii Turing pentru decodarea codului nazist Enigma apoi ENIAC - primul calculator numeric ce putea rezolva o gamă mare de probleme numerice prin reprogramare) a facut posibilă încercarea de a implementa ideile existente, legate de inteligența artificială, pe aceste mașini (reprogramabile). În dezvoltarea domeniului Inteligență artificială, se pot identifica doua zone mari de activitate, prima fiind orientată către cercetări focalizate pe modelarea matematică și algoritmică a proceselor cognitive iar a doua pe ingineria, programarea și construcția sistemelor de calcul (centralizate sau decentralizate) pe care sa fie implementați algoritmii în vederea obținerii inteligentei artificiale.
De notat este faptul ca in aceeasi perioada (1945) apare Genetica și biologia moleculară, domenii aparent fără legătură cu Știința Calculatoarelor și Inteligența artificiala dar care se pot totuși alătura în ideea că Știința Calculatoarelor folosește setul binar [0,1] în programare, iar Natura scrie programe folosind setul quaternar [A,C,T,G], materializat prin baze nucleotidice care scriu ADN-ul. Acestea au fost si sunt in continuare domeniile in care specia umana investeste cele mai multe resurse umane dar si financiare.
Inteligența artificială inglobează zone multidisciplinare orientate spre rezolvarea unor probleme cum ar fi, explicarea procesului cognitiv, percepție, vedere, memorare, strategii de câștigare a unei competiții (un joc de șah), dobândirea capacității de a înțelege și răspunde la întrebări sau stabilire de diagnostice.
Pe parcursul acestui capitol se vor discuta următoarele aspecte introductive cu privire la Inteligenta Atrificială (IA):

- Ce este IA, o încercare de definire și câteva concepte conexe.
    - Definiții.
    - Noțiunea de Agent.
    - Testul Turing.
    - Oameni și roluri în IA
    - Domenii de cercetare în IA.
- La ce este bună IA și câteva exemple.
    - Paralelă cu “programe” care rulează în mintea umană
        - Comparație între creier și CPU
        - Exemple de “programe”. Importanța limbajului
    - Aplicații ale Inteligenței artificiale în viata de zi cu zi
    - Pericole ale IA asupra omenirii.
- Scurtă istorie a IA

De asemenea, pe parcursul cartii, se vor prezenta strategiile legate de rezolvarea unor probleme legate de IA, încercând, pe cât posibil, evitarea mecanismului matematic si programatic. Lucrul acesta nu este posibil întodeauna, și, ca urmare vor fi prezentate secvențe de pseudo-cod sau se va face referire la mecanisme matematice în mod minimalistic dar suficient de detaliat pentru a reține ideea principală.
În această carte se tratează următoarele zone din domeniul inteligentei artificiale:

- Sisteme ”Expert”
- Algoritmi de căutare. Căutare neinformată și informată
- Algoritmi evolutivi. Algoritmi genetici
- Algoritmi de clasificare. Vecinii cei mai apropiați (engleză: Nearest Neighbours). Rețele Neuronale Artificiale
- Logica Fuzzy (engleză: Fuzzy Logic)

Pentru ca cititorul să fie cât mai eficient în acoperirea materialului, se recomandă ca acesta să aibă totuși cunoștințe solide de programare, structuri de date (metode de memorare a datelor în calculator) și matematică.
Obiectivul principal al acestui material este transmiterea către cititor a unor informații elementare din domeniul vast al Inteligenței Artificiale, cu scopul de a construi o "fundație" pe care să se poată construi ulterior. După parcurgerea materialului, cititorul ar trebui să aibă o idee generală legată de tipul de strategie pe care trebuie să o folosească entitatea inteligentă construită pentru a își îndeplini funcția.

Ce este Inteligența Artificială ? Definiții.

Din punct de vedere lexical,
Această întrebare este destul de dificil de răspuns folosind o singură frază, o încercare fiind definirea cuvintelor "inteligență" și "artificial". Ca urmare, Dicționarul explicativ al limbii Române definește aceste două cuvinte după cum urmează:
Inteligență - Capacitatea de a înțelege ușor și bine, de a sesiza ceea ce este esențial, de a rezolva situații sau probleme noi pe baza experienței acumulate anterior; deșteptăciune. [1]
Artificial - Care imită un produs al naturii, care nu este natural; artificios; contrafăcut.[1]
Din definițiile celor două cuvinte se pot identifica unele din direcțiile de cercetare și aplicare ale IA, acestea fiind:

- Înțelegerea de către obiectul artificial (mașina, în cazul nostru) a unor situații (pe baza unor informații senzoriale: video, audio etc) – domeniu încă nerezolvat al IA
- Sesizarea esențialului unei probleme – zonă parțial dezvoltată și în curs de dezvoltare. Aici se face referire la algoritmi de clasificare nesupervizată (exemplu: Rețele neuronale artificiale profunde/adânci, engleză Deep Nets)
- Alegerea acțiunii optime în situații noi (cu scopul obținerii unui rezultat maximizat) – zonă parțial dezvoltată (exemplu: utilizarea algoritmilor de căutare pentru rezolvarea unor probleme)

În același timp, aceste idei (și nu numai) trebuie implementate într-un sistem real, cel mai la îndemână, la ora aceasta, fiind calculatorul numeric iar metoda de implementare a acestor idei este una programatică.
În acest context, Dicționarul explicativ al limbii Române definește
Inteligență artificială este un domeniu al informaticii care dezvoltă sisteme tehnice capabile să rezolve probleme dificile legate de inteligența umană [1]
De reținut: Inteligența Artificială este o ramură a informaticii (engleză: Computer Science).

Din punct de vedere al comunității internaționale a oamenilor de știință și cercetătorilor.
Inteligența Artificială se poate defini, conform P.H. Winston în [2], ca fiind studiul proceselor computaționale care permit percepție, gândire și acțiune.
În [4], Russell și Norvig definesc Inteligența Artificială drept studiul agenților care percep, gândesc și acționează. Se introduce astfel conceptul de AGENT:
Agentul este o entitate reală sau virtuală care își percepe mediul și acționează asupra lui.
Agentul inteligent (rațional) este un agent care acționează asupra mediului înconjurător pentru a își mări utilitatea.

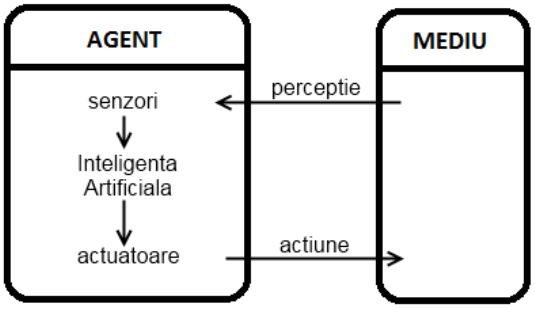

Fig 1.1 Diagrama conceptuală a unui agent inteligent conform Russell si Norvig

Un agent se poate materializa printr-o

- entitate reală – autoturism modern, robot, dronă etc
- entitate virtuală (un program de calculator) – filtru de spam (machine learning), route planner (căutare pe o harta), perosnaj dintr-un joc etc

Modul în care se face percepția, tipul de mediu și acțiunile posibile impun constrângeri asupra lucrurilor pe care un agent le poate face sau nu. De exemplu, există o mare diferență între o percepție perfectă a mediului sau o percepție cu "zgomot", automat și acțiunile vor fi distincte, respectiv vor fi produse prin tehnici diferite.
De reținut: apar constrângeri asupra construcției agentului din cauza mecanismului de percepție și din cauza modului de acțiune.
Conceptul de utilitate a unui agent se referă la modul și eficiența cu care acesta își îndeplinește o sarcină.
Utilitatea unui agent presupune existența unei sau mai multor metode de măsurare a eficienței acestuia. Practic se declară un scop al agentului, se măsoară performanța sa și se compară cu rezultatul dorit (ideal). Această idee se poate modela matematic printr-o funcție:

$$p = d - o \tag{1}$$

unde p=performanța, d=rezultatul dorit, o=rezultatul obținut

Exemplu:
Unui aspirator automat i se "atașează" o funcție de performanță, anume, cât de mult gunoi adună de pe podea în 8 ore; un al doilea criteriu ar fi curentul consumat, apoi zgomotul generat etc.
Important este și modul de construire al funcției de performanță, un agent inteligent ar putea să ia o găleată de gunoi, să o împrăștie și să o adune, rezultând astfel valori mari pentru performanță. De fapt se dorește o podea curată, lucru mai greu de măsurat decât cântărirea gunoiului adunat.
De reținut: Trebuie avută mare grijă în alegerea funcției de performanță.

În [4] Russell si Norvig prezintă patru abordări în care s-a dezvoltat Inteligența artificială. Acestea sunt ilustrate în Fig 2.

Sisteme care gândesc precum omul	Sisteme care gândesc rațional
Sisteme care acționează/se comportă precum omul	Sisteme care acționează/se comportă rațional

Fig.2 Abordări în Inteligența artificială

În Fig.2, rândul de sus se face referință la procesul de gândire/cognitiv, iar rândul de jos este orientat către acțiune și comportament. Coloana din stânga fece referire la comportament uman - imperfect, subiectiv, influențabil iar coloana din dreapta la un comportament rațional.
Pentru a putea programa o mașină să gândească precum omul trebuie în primul rând să se cunoască modul de gândire al unui om. Determinarea procesului de gandire umană este realizabilă prin două metode:

- Introspecție
- Teste psihologice

Rațiunea (modul de a gândi) a fost studiată încă din antichitate de către Pitagora (570-495 îc), Platon (427-347 îc) și Aristotel (384-322 îc), cel din urmă propune ideea de silogisme - structuri care produc întodeauna rezultate corecte, date fiind niște condiții inițiale corecte.

Exemplu de silogism: Ipoteza A: Socrate e un om.
Ipoteza B: Toți oamenii sunt muritori.
Deducție prin silogism, proces prin care rezultă o informație nouă:
Socrate e muritor.

Exemplu de silogism – mod de eșec:
A: Toți câinii sunt animale
B: Toate animalele au 4 picioare (ipoteză eronată)
Rezultă: Toți câinii au 4 picioare (rezultat corect dar produs din ipoteză greșită)

Condițiile prezentate mai sus sunt ideale, având certitudine (în sens statistic), dar ce se întâmplă cu situații care nu au certitudine de realizare, când apar probabilități. Pentru rezolvarea situațiilor de acest fel se propune utilizarea statisticii (ca disciplină matematică) și a logicii Bayesiene (numită după Thomas Bayes 1763). De asemenea, dacă se dorește modelarea unor concepte imprecise (nu improbabile) cum ar fi frig, cald, inalt, slab se poate folosi Logica Fuzzy propusă de Lotfi Zadeh (1965).
În cele mai sus prezentate se cristalizează câteva modele de gândire construite pe logică și matematică. În acest context o definiție a Inteligenței artificiale este:

Definiție: Inteligența artificială se concretizează prin algoritmi construiți pe baza unor modele logice sau matematice ale procesului de cogniție, percepție și acțiune.

"Brains are to intelligence as wings are to flight" este un dicton folosit în contextul Inteligenței Artificiale cu rolul de a sublinia ideea că nu este neapărat sa se reproducă zborul păsărilor pentru a produce un artifact capabil de zbor, în schimb, trebuie să se înțeleagă principiile care produc portanța unei aripi. În mod similar, nu trebuie reprodus un creier uman, trebuie înțelese principiile după care acesta funcționează.

Testul Turing

Testul Turing a fost propus de cercetatorul Alan Mathison Turing (1912-1954) în ideea de a stabili dacă o mașină manifestă inteligență asemănătoare sau echivalentă cu cea umană (inclusiv imperfecțiuni în logică sau memorie). Turing propune testul ca un joc, în lucrarea "Computing Machinery and Intelligence" din 1950 [5], pe când acesta era afiliat Universității din Manchester. Jocul presupune trei entități care răspund la întrebări:

A-calculator
B-om
C-om, Interogatorul

Acestea sunt separate de un perete, așa cum se arată în Fig.3, Interogatorului fiindu-i permis să discute cu entitățile A și B doar prin mesaje scrise. Turing sugerează ideea că, dacă interogatorul nu poate distinge între om-B sau calculator-A, atunci testul Turing este trecut cu succes.

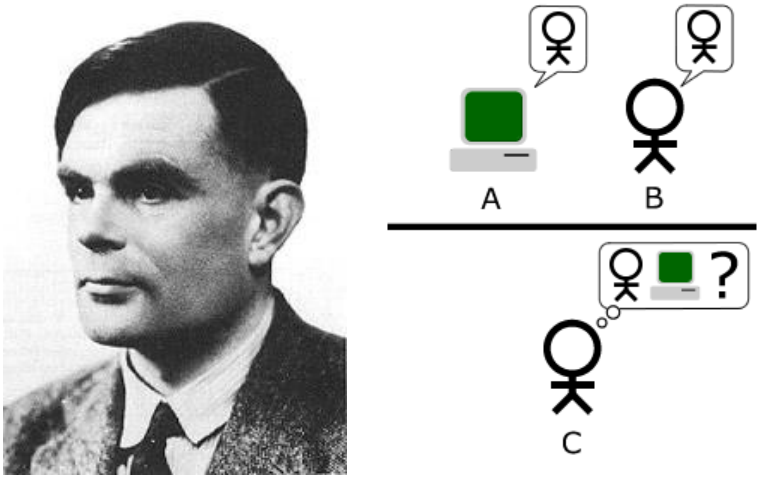

Fig.3 Poza lui Alan Turing (stanga); Schema Testului Turing (dreapta) [web 1]

Testul Turing poartă numele de "jocul imitației" (engleză: imitation game), pentru că un sistem de calcul trebuie să imite comportamentul unui om – inclusiv rațiunea imperfectă.
Exemple de intrebări pe care le poate pune interogatorul:
I: Care e culoarea părului tău? (un calculator nu are păr)
I: Într-o povestire apare textul "luminezi ca soarele" de ce nu se folosește "luna"? (în ideea că un calculator ar trebui să aibă o baze de cunoștințe memorată, din care să rezulte că soarele luminează mai puternic decât luna, ar trebui să înțeleagă subiectul discuției)

Testul Turing este unul dintre primele teste de verificare a inteligenței unei mașini. De la momentul publicării sale, până astăzi, a stârnit o mulțime de discuții, inclusiv filozofice, pe tema Inteligenței Artificiale. Din acest motiv, lucrarea lui Turing [5] rămâne o lucrare seminală pentru domeniul Inteligență Artificială.

Aptitudini pe care trebuie să le aibă un agent pentru a trece Testul Turing

1. Capacitate de procesare a limbajului natural (engleză: Natural Language Processing) – interpretarea mesajelor scrise, comunicarea răspunsurilor într-o limbă „umana"
2. Reprezentare internă a cunoașterii (engleză: Knowledge Representation) – pentru a memora informația necesară într-o discuție
3. Automatizarea procesului de gândire (Automated Reasoning) – pentru a utiliza informațiile memorate în răspunderea la noi întrebări și a trage concluzii noi care pot fi adăugate la baza de cunoștințe
4. Învățare (Machine Learning) – pentru a se adapta la situații noi și pentu a detecta tipare (fenomene repetitive)

Testul Turing Total

Presupune inclusiv interacțiunea fizică printr-o gaură într-un perete cu agentul inteligent. Agentului i se pot "înmâna" obiecte prin gaură, și acesta trebuie să răspundă întrebărilor legate de acestea.
Exemplu: Prin gaură se înmânează o minge. Agentul să poată spune culoare, mărime, duritate etc.

Testul Turing Total - aptitudini suplimentare necesare față de Testul Turing

1. Vedere Artificială (Computer Vision), miros, gust, tactil – pentru a percepe și clasifica obiectul
2. Robotică – pentru manipulare de obiecte

Testul Turing este o metodă clasică de determinare a nivelului de inteligență al unei mașini. Cu toate acestea există și alte teste care se pot efectua în vederea determinării gradului de inteligență al unui agent, de exemplu Camera chinezeasca – engleză: Chinese room etc.

Despre Alan Turing și rolul său în Știința Calculatoarelor.

Alan Turing (1912-1954) de origine Engleză a fost un cercetător în calculatoare, matematician, criptanalist. A avut un rol fundamental în dezvoltarea teoretică a științei calculatoarelor, formalizând ideea de algoritm și computație prin introducerea conceptului ce ulterior a rămas in istorie ca o Mașină Turing (engleză: Turing Machine). O Mașină Turing reprezinta modelul teoretic al unui calculator ce poate fi folosit în scop general, iar Turing este considerat părintele Științei Calculatoarelor și al Inteligenței Artificiale. Trebuie precizat că, până la Turing, toate mașinile, inclusiv calculatoarele erau concepute să îndeplinească o singură funcție. Turing vine cu propunerea, inițial teoretică, de a construi o mașină care să poată îndeplini mai multe funcții – prin reprogramare, de aici rezultă calculatorul de azi și ideea de Turing părinte al calculatoarelor.

Un alt rol fundamental pe care l-a avut Turing, a fost cel de criptanalist englez în timpul Celui De-al Doilea Război Mondial. Acesta a contribuit esențial la spargerea codului Enigma de cifrare a mesajelor radio militare naziste și, prin aceasta, la salvarea a aproximativ 14 milioane de vieți omenești și scurtarea cu aproximativ 2 ani a războiului [6]. Pentru a putea descifra codul Enigma, Turing a folosit cunștințele sale despre calculatoare și, împreună cu Gordon Welchman (matematician), au construit pentru Guvernul Englez un dispozitiv de decriptare electromecanic “The Bombe”, la Bletchley Park în perioada 1939-1944. Proiectul a fost bazat pe o mașină de decriptare poloneză “bomba kryptologiczna” dezvoltată până în 1939 de Marian Rejevski.

Mașina de decriptare dezvoltată de Turing a înglobat multe dintre ideile sale legate de calculatoare reprogramabile și a stat la baza construirii în 1944 a lui Colossus (UK) și în 1945 a lui ENIAC - Electronic Numerical Integrator And Computer în 1946, primul calculator electronic reprogramabil, la Universitatea Pennsylvania (SUA) de John Mauchly si J. Presper Eckert.

Turing moare în 1954, la 41 de ani, în condiții suspecte. Cauza morții a fost stabilită drept otravire cu cianuri. Trebuie menționat și faptul că Turing era condamnat pentru comportament indecent pe motiv că era homosexual. În 2009 Premierul Britanic Gordon Brown își cere scuze în numele Guvernului pentru tratamentele aplicate lui Turing, iar în 2013 Regina Elisabeta a Angliei îl scoate de sub acuzare post mortem. În 2016, Parlamentul Englez dezbate Legea Turing, care propune scoaterea de sub acuzare retroactiv a tuturor persoanelor care, de-a lungul istoriei, au fost condamnate că sunt homosexuali.

Date fiind realizările lui Turing în 41 de ani de viață, se poate doar specula ce ar fi putut realiza acesta dacă ar fi trăit mai mult.

Oameni și Roluri în Inteligența Artificială

În Inteligența Artificială, la fel ca în orice domeniu, există oameni care îndeplinesc diverse funcții. Aceste funcții pot fi împărțite în două categorii mari:

1. Cercetătorii în domeniul Inteligenței Artificiale au rolul de a studia fenomenele prin care se produce acest efect numit inteligență, pentru ca ulterior să creeze modele logice, matematice sau de altă natură pe baza cărora s-ar putea construi un algoritm sau agent
2. Inginerii în domeniul IA au rolul de a construi un agent inteligent și aplicații ale acestuia, în scopul rezolvării unor probleme reale, pe baza unor algoritmi dezvoltați de cercetători.

Domenii de cercetare în Inteligența Artificială

Inteligența Artificială este un domeniu vast, interdisciplinar care cuprinde multe zone de cercetare fundamentală. Câteva dintre acestea sunt:

Antropologie – studiul evoluției oamenilor din trecut până în prezent.

- Arheologie – studiul activității umane prin analiza urmelor recuperate
- Antropologie Lingvistică – studiul influenței limbajului asupra vieții sociale
- Antropologie fizică și biologică – studiul aspectelor biologice și comportamentale ale evoluției oamenilor, a primatelor înrudite și hominide dispărute
- Antropologie socio-culturală – studiul asemănărilor și diferențelor între diverse populații de oameni

Știința Calculatoarelor

- Algoritmi – un proces sau un set de reguli care trebuie urmărit în calcule sau alte operațiuni de rezolvare a unor probleme, în special efectuate ce calculator
- Știința Calculatoarelor – studiul teoriilor, modului de utilizare și proiectare a calculatorului

Matematică - Știință care se ocupă cu studiul mărimilor, al relațiilor cantitative și al formelor spațiale (cu ajutorul raționamentului deductiv)

Logică – știință axată pe principiile și criteriile folosite în procesul de deducție sau demonstrație

Filozofie – studiul naturii fundamentale a cunoașterii, realității și existenței

Psihologie – studiul comportamentului și minții omului acoperind toate aspectele legate de conștient, subconștient și procesul cognitiv. Este o disciplină academică și o știință socială care urmărește să înțeleagă individul sau grupul prin stabilirea unor principii general aplicate.

Biologie – știință a naturii preocupată cu studiul vieții și a organismelor vii, abordând structura, funcția, creșterea, evoluția, distribuția, identificarea și clasificarea acestora

Neuroștiință – știință medicală preocupată de sistemul nervos

Știință cognitivă – studiul gândirii, învățării și organizării mentale care se bazează pe aspecte de psihologie, lingvistică, filozofie și modelare computerizată.

Definitii din [1]

Aplicatii ce includ Inteligenta Artificiala

Paralelă cu "programe" care rulează în mintea umană

Comparație între creier și CPU

Mintea umană este un mecanism complex pe care nu îl înțelegem complet la ora actuală (2017), cu toate acestea, se pare că există ceva asemănător cu o subrutină sau procedură ce se întâmplă atunci când dorim să realizăm o sarcină. Aceste mici programe apelează la resursele fantastice (în sensul că o mașină creată de om, nici măcar nu se apropie de complexitatea organismului uman) pe care le pune la dispoziție organismul uman.

Exemplu de resurse ce stau la dispoziția sistemului cognitiv uman:

Senzori: cele 6 simțuri
- Vedere – ochi, retină, nervul optic
- Auditiv – timpan, oase ale sistemului auditiv, nervul vestibulo-cohlear
- Echilibru - organul de echilibru din urechea internă, nervul vestibulo-cohlear
- Tactil – fibre nervoase cu receptori pentru caldura, frig, mâncarime, durere, presiune
- Miros – senzori chimici din membrana mucoasei nazale, nervi olfactivi
- Gust -senzori chimici la nivel de limbă, nervi gustativi

Actuatoare: mușchii
- Striati – scheletici, aproximativ 650 (la care se adună mulți mușchi foarte mici)
- Netezi – organe, cateva mii
- Mixt – cardiac, unul singur

Procesorul: creierul și restul sistemului nervos
- Neuroni în creier: 10^{11}, dintre care $9*10^{10}$ în cerebel; 10^{10} în cortex (funcții de cogniție)
- Sinapse: 10^{15}

Dacă se face o comparație între un procesor Intel Core i7 6950X (2016) care e capabil de 317,900 MIPS la frecvența de 3.0 GHz [web 2] echivalentul a $3*10^{12}$ instrucțiuni pe secundă, s-ar ajunge la numărul de neuroni din creier, dacă aceștia ar executa o sinapsă pe secundă. Realitatea este alta, neuronii au 10^{15} sinapse, de unde rezultă un număr de 10^{26} "calcule" pe secundă. În plus, creierul uman procesează informația in paralel, pe când un CPU o face în serie. Concluzia este că fiecare om are în dotare un supercalculator. Comparația se poate extinde și la aspectul că, un calculator poate executa o operație de tipul 9345*2124 mult mai repede decât un creier uman, pe de altă parte, creierul uman nu are nici o problemă când trebuie să identifice un obiect într-o imagine.

Exemple de "programe" in creierul uman. Importanța limbajului

În lucrarea [7], lingvistul american Noam Chomsky subliniază faptul că limbajul de comunicare este o chestiune ce îl caracterizează pe homo sapiens. Interesant este că, primele urme ale lui Homo Sapiens pe Terra apar acum 200.000 de ani, dar dovezi ale utilizării sculelor, desene rupestre și probabil limbajul elementar – onomatopee (dar e destul de dificil de demonstrat) – ca mijloace de comunicare apar semnificativ întârziat, acum aproximativ 50.000 ani. Concluzia este, că lui Homo Sapiens i-a trebuit o perioadă lungă să dobândească această facultate, Chomsky fiid de părere că, parțial, această aptitudine este codificată în ADN-ul speciei.

Limbajul are un rol important, din mai multe puncte de vedere; el poate fi folosit să alerteze grupul, poate fi folosit în activarea unor procese cognitive în alte persoane sau poate fi folosit în a povesti diverse întâmplări, astfel rezultând procesul de învățare.

Învățarea/experiența se poate produce/câștiga prin trei metode principale:

1. Experiență proprie
2. Observații asupra altor persoane.
3. Povestiri.

La categoria povestiri intră inclusiv materialele scrise, în care autorul își exprimă în scris experiența și cunoștințele.

Exemplul 1. Un prim exemplu de program care rulează în mintea umană poate fi cel propus de Winston în [3], care propune următorul experiment, în care cititorul trebuie să conștientizeze procesul cognitiv:
Întrebare: Linia ecuatorului, peste câte țări trece în Africa? – vezi Fig. 4 cu sugestia să numărați.

Fig. 4. Harta Africii în zona Ecuatorului

Ce se întâmplă la nivel de proces cognitiv:

- Limbajul permite scriitorului să comande cititorului să ruleze um program
- Aparatul vizual al cititorului urmărește linia
- În minte, rulează un program de contorizare/numarare al țărilor de pe linie
- La sfârșit, cititorul poate enunța numărul contorizat folosind limbajul, număr (Răspuns) = 6.

Spre mirarea cititorului, acesta, tocmai a fost programat să execute o sarcină, în concluzie, programarea oamenilor se poate face prin limbaj.

Exemplul 2. Un alt exemplu de program care rulează în mintea umană poate fi următorul, și în timpul acestui experiment, cititorul este rugat să fie atent la procesul cognitiv ce se desfășoară în mintea sa:
Întrebare: Cât face 122+89 ? Cititorul este rugat să execute calculul (sigur, experimentul este mai spectaculos dacă nu se folosește un calculator)
Pentru a calcula rezultatul, cititorul aplelează la cunoștințe legate de aritmetică, și execută operații complicate, cum ar fi 2+9=11, 1 rămâne, 1 se adaugă în față etc., iar rezultatul final ar trebui să fie 211
Răspuns: 122+89=211

Experimentul nu se încheie aici, se propune o întrebare nouă:
Întrebare: Cât face 122+89 ? (in mod intentionat este vorba de aceleasi numere, nu este greșeală)

Răspuns: 211
Este puțin probabil ca cititorul să aplice același proces de calcul, laborios și consumator de timp, în schimb, acesta accesează direct zona de memorie pe termen scurt, sau se uită mai sus, și extrage răspunsul.
În concluzie, în faza inițială, se apelează un prim program care clasifică tipul de problemă - în aritmetică, după care, se apelează cunoștințe specifice aritmeticii și se calculează rezultatul după un algoritm. În faza a doua a experimentului, pentru că există recent o experiență de calcul a celor două numere, rezultatul se extrage direct din memorie.

Exemplul 3. Un al treilea exemplu de program poate fi invocat făcând următorul experiment mental, propus de Winston în [3]:
Sarcină: Imaginați-vă că fugiți pe stradă cu o găleată plină cu apă.
Întrebări: Ce se întâmplă cu apa din găleată? (vă rog să va opriți 2 secunde și să răspundeți)
Vă udați pe picioare ? (vă rog să va opriți 2 secunde și să răspundeți)
De fapt ați rulat în minte un simulator al realității, care vă spune că, probabil apa se varsă din găleată și vă udați pe picioare. Un alt aspect interesant este că, această informație nu se găsește pe google/internet, și probabil că nici nu ați avut vreodată o experiență să alergați pe stradă cu o găleată plină cu apă. Cu toate acestea, ați putut prezice rezultatele din acest scenariu. Deocamdata masinile nu pot realiza asemenea „preziceri".

Sarcină: Imaginați-vă că fugiți pe stradă cu o găleată plină cu monede în masă totală de 15kg.
Întrebare: Cât de bine puteți fugi? (vă rog să va opriți 2 secunde și să răspundeți)
Răspunsul, în acest caz va fi probabil, nu puteți fugi foarte bine, din cauza masei pe care o cărați. Cu toate că scenariul este și mai puțin probabil, simularea internă poate face prevederi suficient de precise legate de rezultatele acțiunilor pe care le facem.

De reținut: În mintea fiecărui om există un proces de simulare internă a realității, care ne ajută să prevedem rezultatele unor acțiuni pe care le intreprindem. De asemenea experiența rămâne memorată și ajută la finisarea procesului de simulare internă. Rolul limbajului este esențial pentru că activează programe complexe în mintea umană.

Exemplul 4. Numit Generează și testează, în care se presupune următorul scenariu, se dorește clasificarea unui animal văzut (de exemplu pe o câmpie) dar al cărui apartenență la o specie de animale nu se cunoaște. Procedura de clasificare poate fi relativ simplă, se caută într-un atlas zoologic tipul de animal care se potrivește cel mai mult animalui observat. Ideea de potrivire poate fi interpretată de un calculator prin multipli parametri, cum ar fi: forma cu 2,4,6 picioare, culoare, dispunerea picioarelor etc. Algoritmul de clasificare este reprezentat in Fig.5.

Fig. 5 Diagrama algoritmului Generează și testează

O variantă a algoritmului prezentat mai sus este dacă, problema se schimbă ușor, și se dorește identificarea unei acțiuni optimale. În acest caz se poate înlocui atlasul zoologic cu aparatul simulator din

mintea omului/ calculatorului unde se pot simula diverse acțiuni și se poate compara rezultatul acestora cu rezultatul dorit.

Aplicații ale Inteligenței artificiale în viata de zi cu zi

Inteligența artificială este o știință nou apărută, și ca în orice știință, partea de cercetare teoretică precede partea aplicativă, ca ordonare în timp. Cu toate acestea, lumea moderna nu poate fi concepută fără avantajele pe care ni le oferă Inteligența artificială.
În aceste condiții apare "efectul Inteligență artificială" (engleză: The AI effect), în care un observator minimizează efectele comportamentului unui program inteligent spunând că nu este vorba de inteligență adevărată e doar un efect. Cercetătorul în Inteligență artificială Rodney Brooks, membru al Academiei de Științe Australiene, spune într-un articol in revista Wired , "De fiecare dată când ne dăm seama cum funcționează ceva, lucrul acela își pierde din misticism; și continuăm prin a spune – Oh !, e doar o computație". În același context, filozoful suedez Nick Bostrom asociat Universității Oxford declară în [9] "o mulțime de aplicații ale Inteligenței artificiale au fost filtrate în aplicații generale, de multe ori fără să fie denumite măcar IA, pentru că, odată ce un lucru devine suficient de util și suficient de uzual, nu mai este etichetat Inteligență artificială".

După domeniile unde se utilizează Inteligența Artificială, se pot distinge:

Știința Calculatoarelor
Poate cea mai de succes aplicație este motorul de căutare Google. Acesta implementează tehnici de IA pentru a oferi utilizatorului ce caută (lucru care nu e trivial). Google intră în categoria mai mare a unor aplicații numite Web Crawler, niște programe care "bântuie" pe internet adunând date. Alte exemple de motoare de căutare sunt bing de la Microsoft sau Yahoo! Ale căror logo-uri sunt prezentate în Fig.6

Fig.6 Motoare de căutare

În [4] Russell și Norvig sublinează faptul că în laboratoarele de dezvoltare a sistemelor inteligente au apărut tehnologii precum time sharing, interactive interpreters, graphical user interfaces și mouse-ul pentru calculator, rapid development environments, structura de date numită linked list, automatic storage management, symbolic programming, functional programming, dynamic programming și object-oriented programming.
Filtrele de spam email (engleză: spam filter) trebuie să învețe continuu, pentru că spammerii se imbunătățesc continuu și dezvoltă noi strategii. Astfel nu se poate concepe un filtru de spam static eficient, ci doar unul adaptiv.

Medicină
Clinical decision support system – sisteme de asistare a medicilor prin propunerea unui diagnostic al unui pacient, bazat pe analize, medicul având rolul de a verifica și valida diagnosticul.
Interpretarea computerizată a imaginilor medicale – tomografii etc., Roboți companion, stabilirea automată de rețete de medicamente bazate pe diagnostic, crearea de medicamente.

Watson de la IBM, aparut in 2013, un program care este capabil sa discute in limbaj natural si sa puna diagnostice medicale.

Robotică – domeniu în care agenții învață să îndeplinească diverse sarcini, mai degrabă decât să fie programați

Asistent personal – aplicații precum Siri, Google Now, Cortana – Logo-urile respectivelor produse sunt ptezentate în Fig. 7.

Fig. 7 Logo pentru Siri, Google Now, Cortana personal assistant

Transport – autoturisme auto pilotate: Google Car – Fig.8 [web 3] sau Tesla

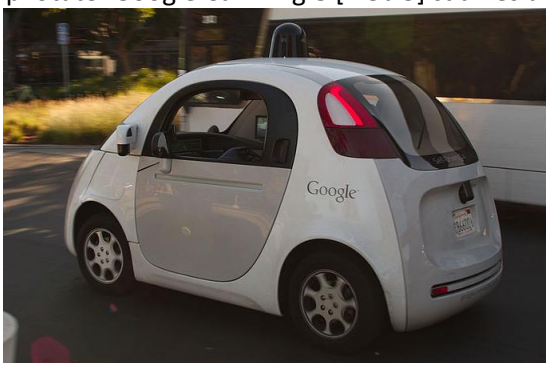

Fig.8 Google – autoturism auto-pilotat

Cutii de viteză automate inteligente – bazate pe logică Fuzzy în grupul Volkswagen, altă aplicație a inteligenței artificiale în domeniul transportului.

Industria jocurilor – fie că jucați împotriva "calculatorului", care este de fapt un agent, fie că sunteți într-o lume virtuală în care apar personaje (nu neapărat adversari), aceștia, de multe ori, manifestă un comportament inteligent și sunt numiți Video Game Bot – Fig.9. Bot provine de la prescurtarea lui roBOT.

Fig.9 Joc de șah și FIFA 13 (produs de EA Sports)

Jocurile de echipă sunt un subiect extrem de complex, pentru că există agenți colaborativi dar și agenți competitivi.

Finanțe
Aici, IA este folosită pentru a detecta posibilitățile de fraudă sau estimare și evaluare a prețului acțiunilor pe bursă.

Funcții pe care le poate rezolva Inteligența artificială:
Recunoaștere a caracterelor dintr-o poză (OCR - Optical character recognition) – permite prelucrarea automată a unei cantități mari de informații din poze (BigData). Încercați următorul experiment, faceți o poză unei pagini dintr-o carte, apoi instruiți un calculator să caute un cuvânt pe care voi (oameni fiind) îl puteți citi, probabil că nu se poate, puteți în schimb să folositi un software OCR care să transforme poza în litere și cifre, apoi în cuvinte care pot fi căutate. Astfel, se pot digitiza o milioanele de cărți care au apărut înainte de tehnologia OCR.
O altă aplicație a acestei tehnologii este prezentată în Fig. 10 – Identificarea numărului autoturismului. Acest lucru permite căutarea numărului într-o bază de date și extragerea de informații legate de proprietar.

Fig.10 Identificarea numărului autoturismului [web 4]

Identificarea scrisului de mână (Handwriting recognition) – aplicație ce permite scrierea directă, de mână, pe ecranul unui dispozitiv smart. Acest lucru este ilustrat în Fig. 11.

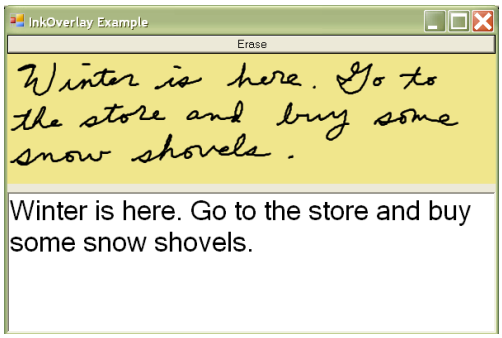

Fig. 11 Recunoașterea scrisului de mână

Recunoașterea vorbirii (Speech recognition) – aplicație ce permite unui om să spună ceva, iar calculatorul să scrie textul corespunzător. Acest lucru este foarte util în zona sistemelor de securitate și

urmărirea automatizată a convorbirilor, în ideea că, dacă apare un cuvânt clasificat drept cheie, sistemul poate emite un semnal de avertizare.
O alta aplicație a acestei funcții este în zona asistentului personal, omul poate să comande calculatorului să facă ceva, folosind limbajul natural (Natural Language Processing)

Traducerile de texte – automatizate, în care se ține cont de gramatica limbii sursă, respectiv țintă

Chatterbot – aplicații cu care se poate discuta prin mesaje scrise, în ideea de consultant/asistent

Creativitate artificială (Artificial creativity) – aplicație în stadiu incipient, prin care se dorește replicarea creativității, în special umane. Pentru aceasta se construiesc algoritmi și modele matematice ale proceselor psihologice din zona de creativitate.

Identificarea și recunoașterea unei fețe (Face identification and recognition) – permite detectarea de fețe în poze. Trebuie specificat că, prin identificarea unei fețe într-o poză, se dorește ca în poza respectivă, sistemul să spună dacă apare sau nu o față. Recunoașterea unei fețe se referă la faptul că, apare fața cetățeanului X în poză. În Fig. 12 se prezintă unele aspecte legate de algoritmii de identificare facială.

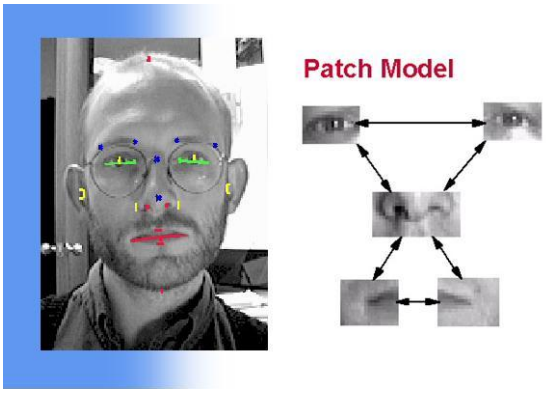

Fig. 12. Aspecte legate de identificare facială [web 5]

Vedere artificială (Computer vision, Machine Vision, Virtual reality and Image processing) – permite unui sistem de calcul să "vadă" ce se întâmplă în jurul său. Se folosește în multe zone cum ar fi, automatizarea autoturismului sau inspecție automată a componentelor din industria de manufactura. Aceste tehnologii sunt ilustrate în Fig. 13.

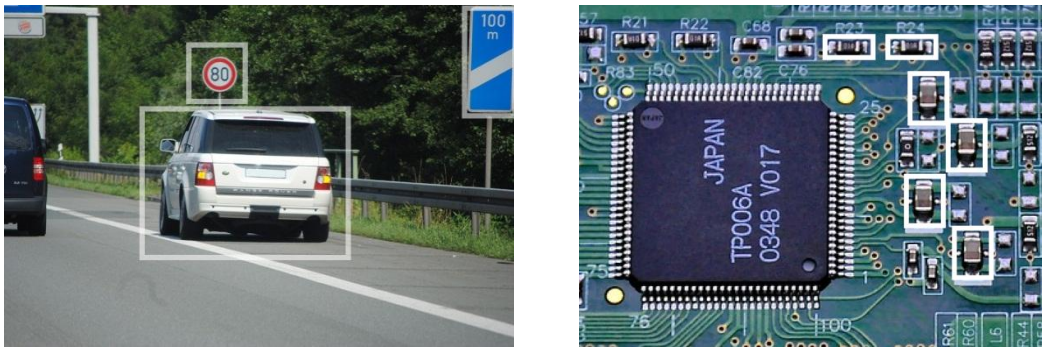

Fig.13. Identificare de obiecte – pe autostrada (stânga), electronice (dreapta) [web 6]

Planificare automată (Autonomous planning and scheduling) – folosită de exemplu în găsirea traseului optim pe o hartă memorată în calculator, planificarea mișcărilor într-un joc – șah, de exemplu, când Deep Blue l-a bătut pe Kasparov, Campionul Mondial de la acea vreme sau Logistică, in sensul eficientizării utilizării resurselor.

În cele de mai sus s-au prezentat doar unele dintre aplicațiile pe care le are acest domeniu vast, numit Inteligență Artificială.

Pericole ale IA asupra omenirii și a modului nostru de viață

La ora actuală acest subiect ține mai mult o speculație decat de un fapt real, dar având în vedere evolutia rapidă din domeniul Inteligenței Artificiale, aceste pericole se conturează tot mai clar.
Se discută despre două categorii de pericole:

1. IA conduce la introducerea automatizarii în tot mai multe locuri de muncă, lasând astfel mulți oameni fără serviciu. Există totusi dovezi că pe măsură ce unele locuri de muncă dispar, apar altele noi, insă *viteza de distrugere* a locurilor de muncă *este mai mare* decât *viteza de creare de locuri noi* de muncă. În sine, automatizarea nu este un lucru rău pentru că ușurează munca oamenilor, dar în acest proces, mulți oameni raman fără slujbă, tendință care duce la nevoia de schimbare a sistemului socio-politico-economic. Ca răspuns la aceste provocări se propune introducerea unui venit minim asigurat fiecarui om, indiferent dacă acesta are sau nu serviciu.

Exemple de servicii care sunt automatizate sau urmează a fi automatizate în viitorul apropiat: servicii de asamblare automatizată/robotizată, automobile care se auto-conduc sau asistenți personali (inclusiv pentru persoane cu disabilități sau în vârstă)

2. Posibilitatea ca Superinteligența unui Agent Inteligent care să fie îndreptată împotriva omenirii. În acest sens se introduce si termenul de Singularitate tehnologică (prezentat prima data de John von Neumann [10]) ca fiind momentul în timp când IA va depăși inteligența umană. Se estimează ca acest moment va fi atins undeva in 2040 conform Stuart Armstrong [11]. De asemenea, multe personalități din domeniul IT precum Bill Gates, Steve Jobs, Stuart Russell, Elon Musk alaturi de Stephen Hawhing (Astro-Fizician) și-au exprimat îngrijorarea cu privire la această posibilitate.

Fig. 14 Poza de profil a Microsoft Tay ChatterBot [web7] pe Twitter

Un exemplu, rudimentar totusi pentru o superinteligență, poate să fie Tay – un chatterbot (software care răspunde prin text unor mesaje text primite) pe care firma Microsoft l-a implementat pe platforma Twitter in 23 martie 2016. Proiectul Tay a fost o încercare de a produce un Agent Inteligent care să învețe să discute pe baza interacțiunii cu oamenii de pe platforma Twitter. După doar 16 ore Tay a fost dezactivat pe motiv că producea răspunsuri jignitoare la adresa interlocutorilor [web7]. De subliniat este faptul că Tay nu a fost programat ca sa producă răspunsuri jignitoare, dar prin interacțiunea cu oamenii, și în lipsa unor reguli de bun-simț implementate, acesta a învățat să se comporte jignitor. În mod similar, analizând comprtamentul uman, o superinteligență poate învăța să fie inclusiv violentă.
În altă ordine de idei, există multe jocuri în care oamenii se concurează cu agenți inteligenți și de multe ori oamenii pierd, de exemplu Șah sau GO, ca urmare există deja Agenți care surclasează omul în unele domenii.
Totuși beneficiile unei superinteligente sunt evidente, iar problemele prezentate ar trebui doar să crească gradul de atenție și precauție al cercetătorilor in dezvoltarea de noi sisteme cu IA.

Scurtă istorie a Inteligenței Artificiale

Studiul istoriei IA este important pentru că oferă informații despre modul cum a început să se dezvolte acest domeniu, modul cum au evoluat ideile si abordarile pe parcursul timpului și oferă o imagine de ansamblu – unde am fost și unde ne îndreptăm, putând astfel face previziuni despre viitor. Firul istoric al IA se desface în mai multe subdomenii, în continuare fiind prezentate cateva etape importante în evoluția domeniului în ordine cronologică. Sursele bibliografice folosite în acest domeniu sunt Russell si Norvig [4], McCorduck [12] și Wikipedia [web 8] (diverse articole).

Teorii (1850-1950)
Primele teorii legate de posibila utilizare a calculatorului în alte scopuri decăt cele computaționale au fost dezvoltate de Ada Lovelace, mathematician englez, care a lucrat împreună cu Charles Babbage la dezvoltarea primului calculator mecanic. Ea schițează ceva ce ulterior va fi cunoscut drept un algoritm de rezolvare numerică a unei probleme.

Începutul (1943-1956)
În perioada anilor 1920-1940 studii asupra neuronilor biologici arată că aceștia se bucură de o proprietate, anume, dacă stimulii depășesc un anumit prag, aceștia se activează și transmit un semnal mai departe pe directia dendrite-axon. Bazat pe aceste descoperiri, în 1943, doi cercetători Walter Pitts și Warren McCulloch propun un model al unui neuron artificial care, pus într-o rețea de neuroni artificiali asemanatori să se activeze (și să transmită un semnal mai departe) sau nu în funcție de semnalele primite de la neuronii invecinati, astfel rezultând o rețea neuronală. Ei au demonstrat că o asemenea rețea neuronală poate produce operații logice tip ȘI/SAU/ORI. De asemenea, în această perioadă se construiesc primele calculatoare electronice.

1950 – Testul Turing

1951 – Modelul neuronului artificial este transpus in realitate de Marvin Minsky, rezultând astfel SNARC, prima retea neuronală artificială

1956 –Conferința de la Dartmouth College in care *termenul de Inteligență Artificială a fost introdus în comunitatea științifică* de către John McCarthy. Conferința a fost patronată de Minsky, McCarthy, Claude Shannon (IBM) și Nathan Rochester (IBM).

Înflorirea Inteligenței Artificiale (1950 – 1970)
Perioada este marcată de un optimism exuberant, sustinut de diverse programe și concepte noi, care ilustrează potențialul domeniului. In continuare se prezinta cateva programe grupate dupa tehnicile de rezolvare a problemelor.

- Rezolvarea problemelor folosind paradigma - Gândirea ca un proces de căutare

  1950/1953 – Claude Shannon/Alan turing propun programe de jucat sah

  1955 – Newell si Simon prezintă programul LOGIC THEORIST care putea demonstra unele teoreme folosind procese de logică similar omului. Ulterior, în 1959 acestia dezvolta GPS-General Problem Solver, un program care să fie capabil să rezolve probleme generale in mod similar.

  1958 – John McCarty dezvolta mediul de programare LISP (*LIS*t *P*rocessor)

  1958 - Herbert Gelernter - Geometry Theorem Prover

  1961 – James Slagle – SAINT, un program de rezolvare simbolică a unor probleme de analiza matematica - integrale

  1971 – STRIPS (Stanford Research Institute Problem Solver), un program care controlează un robot mobil (SHAKEY), prin descompunerea unei sarcini în sub-sarcini

- Natural Language Processing – programe care inteleg comenzi la nivel de conversatie
  1956-1960 se dezvolta conceptul de Retele Semantice (engleza: Semantic Network) - Fig.15, folosind o reprezentare a legaturilor dintre cuvinte bazata pe teoria grafurilor.

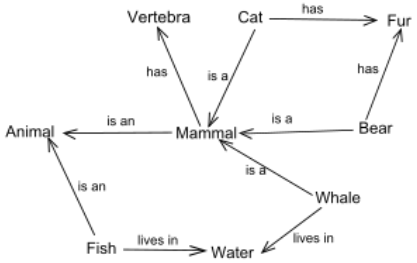

Fig.15 Retea Semantica

Codificarea in LISP a unei retele semantice se poate face asa cum se arata in exemplul urmator:

```
(defun *database* ()
((cat  (is-a Mammal)
       (has Fur)
       (size small))
 (bear (is-a Mammal)
       (movement quadruped))
 (Mammal  (is-an animal)
       (has Vertebra)
       (reproduction viviparous))))
```

1964 – Joseph Weizenbaum dezvolta ELIZA, un program in care utilizatorul putea avea o discutie text cu calculatorul. Programul functiona pe baza unor reguli de reinterpretare a mesajelor introduse de utilizator. Acesta program este primul din categoria chatterbot, variantele moderne fiind Siri, Cortana sau Google Now

- MicroWorlds – la inceputul anilor 1960, Marvin Minsky propune simplificarea realitatii prezentate unui program de calculator si reducerea acesteia la cateva aspecte esentiale – procedeu preluat din stiinte precum fizica, unde se folosesc modele simplificate pentru a intelege principiile de baza, de exemplu: se nejlijeaza forta de frecare, sau se folosesc corpuri solide rigide nedeformabile. Cel mai cunoscut program din aceasta categorie este SHRDLU BoxWord construit de Terry Winograd
- Retele neuronale
  1958 – Frank Rosenblatt introduce Perceptronul, un algoritm capabil sa invete sa clasifice doua tipuri de obiecte, bazat pe ideea de neuron artificial.
  In perioada 1969-1979 munca de cercetare in domeniul retelelor neuronale stagneaza pe motiv ca existau pareri cum ca teoriile conectiviste (retelele neuronale) nu ar fi calea catre IA, directia nefiind finantata in mod consistent, adversarii acestei terorii fiind Minsky sau Papert, ei sustinand teoriile logiciste introduse de McCarthy, conform carora tot ce se intampla in mintea unui om e bazat pe logica, mai putin pe conexiunile dintre neuroni.

<u>Prima criza in IA (1974-1980)</u> (engleză: A.I. Winter)
Aceasta apare pe motiv că cercetătorii din domeniu erau prea optimiști raportat la tehnologiile pe care le aveau la dispoziție. Criza se produce în principal datorită faptului că au fost finanțate foarte multe proiecte de cercetare dar rezultatele promise nu au apărut. Principalele cauze au fost:

- Capacitate limitată de calcul a calculatoarelor

- Durata de rezolvare a unor probleme de AI creste exponențial, odată cu creșterea liniară a complexității (numărul de parametri la intrare)
- Lipsa unor baze de date suficient de elaborate, respectiv a infrastructurii de stocare a datelor (pe vremea aceea nu a existat internet)

Efectele crizei s-au manifestat prin reducerea drastică a finanțării în domeniul IA și, ca urmare, mulți cercetători au părăsit domeniul, slăbind astfel capacitatea acestuia de a se dezvolta.

Renașterea (1980-1990)

În perioada 1970-1980 se dezvoltă o ramură a IA bazata pe reprezentarea cunoașterii folosind reguli de tip IF-THEN și silogisme. Acestea ulterior devin cunoscute ca Sisteme Expert (Rule Based Expert Systems, sau doar Expert Systems). Denumirea de Sisteme Expert provine din faptul că imită modul de lucru al unui expert dintr-un anumit domeniu.

1980 XCON – Sistem Expert care alcătuia automat lista de accesorii (cabluri, conectori etc) pentru calculatoarele produse și livrate de Digital Equipment Corporation inducând astfel economii de 40 Milioane USD anual în firmă. Acest sistem a fost printre primele mari succese financiare ale IA. Tehnologia folosită se bazeaza pe ideile implementate în urmatoarele programe dezvoltate ca platforme de cercetare:

1969 DENDRAL – Sistem Expert dezvoltat de Buchanan și Feigenbaum pentru identificarea structurii moleculare a unei substanțe chimice pe baza unei spectrograme.

1972 MYCIN – Sistem Expert care era capabil să prescrie o rețetă de medicamente pentru boli infecțioase ale sângelui bazat pe un set de analize ce erau introduse în sistem.

Acest succes financiar al lui XCON nu trece neobservat de competitori, iar pana in 1990 se dezvolta o întreagă industrie de producere a Sistemelor Expert valorând aproximativ 1 miliard de USD. Totodată se dezvoltă și platforme hadware tip workstation, optimizate să ruleze LISP – limbajul de programare cel mai eficient pentru construirea unui Sistem Expert

1981 – Guvernul Japonez initiaza un proiect in valoare de 850 mil USD – "Fifth Generation Computer" prin care dorește sa construiasca masini inteligente capabile să poarte o conversație, traducă un text, interpreteze o imagine și să gândească asemenea unui om.

1986 – apare o colectie de articole de cercetare intitulata „Parallel Distributed Processing" care produc o reorientare și relansare în domeniul conectivist (retele neuronale artificiale), aceasta a fost editata sub coordonarea lui Rumelhart, McClelland si Hinton.

A doua criză în Inteligența Artificială (1990)

Aceasta se referă la implozia industriei de Sisteme Expert, care folosea hardware specializat (optimizat pentru a rula LISP). Odată cu apariția Desktop PC-urilor IBM și Apple, mult mai ieftine și performante, întreaga industrie de productie a hardware-ului specializat pentru a rula Sisteme Expert a disparut, acestea fiind înlocuite de Desktop-uri. De asemenea, sisteme precum XCON au ajuns sa fie scump de intretinut, actualizat si nu erau capabile să învețe rapid cazuri noi. În paralel, proiectul japonez Fifth Generation Computer nu a livrat promisiunile ambițioase pe care și le-a propus si a fost oprit.

Evenimente recente și orientare către prelucrarea statistică a datelor (1990-prezent)

1990 – introducerea primului software de recunoaștere verbală (speech recognition) numit Dragon Dictate, preț de vanzare de 9000 USD

1996 – Sergey Brin și Larry Page dezvoltă PageRank, algoritmul folosit de catre motorul de cautare Google

1997 – Deep Blue, un supercomputer produs de IBM pe care ruleaza algoritmi de IA, îl bate la Șah pe Campionul Mondial Garry Kasparov

2005 – DARPA Grand Challange pentru vehicule autopilotate într-un traseu din deșert este câștigat de Universitatea Staford
2007 – DARPA Urban Challange pentru vehicule autopilotate în mediu urban este căștigat de un vehicul construit la Carnegie-Mellon University, acesta ține cont de trafic și semne de circulație.

Există și alte domenii unde IA și-a făcut simțită prezența, precum planificare automatizată (de exemplu producția dintr-o fabrică, misiuni spațiale, alocarea resurselor, utilizarea eficientă a resurselor unui PC atunci când acesta efectuează sarcini multitasking, identificarea unui traseu optim pe hartă), algoritmi bazați pe încredere (in sens statistic) și probabilități in apariția unui eveniment (belief networks, Bayesian networks, Hidden Markov Models) propuși de Judea Pearl.
De asemenea IA este asociată unor concepte precum BigData (prelucrarea unor cantități mari de date digitale care nu sunt structurate în baze de date), Machine Learning (mașini care învață să facă un lucru fară a fi programate sa facă acel lucru, avand doar algoritmi de învățare), Inteligența Artificială Generală (scopul fiind crearea unor mașini inteligente capabile să gândească precum omul)

<u>Concluzie</u>
După 70 de ani de existență a Inteligenței Artificiale, principala realizare este sedimentarea și conturarea clară a teoriilor care stau la baza dezvoltării acestui domeniu.
De-a lungul istoriei IA s-au conturat doua mari direcții de abordare teoretică a domeniului, conecționismul și logicismul, care la un moment dat se excludeau, dar istoria demonstrează că aceste teorii se completează.

Conectionismul – folosește ideea că o mașină poate învăța prin ajustarea ponderilor legăturilor dintre niște neuroni artificiali, asemănător creierului uman

Logicismul – comportamentul inteligent poate fi codificat prin logică, folosind reguli tip IF-THEN.

Treptat, toate teoriile apărute de-a lungul timpului in zona Inteligentei Artificiale au condus la dezvoltarea unei industrii ce astazi produce 8 miliarde USD anual.

Bibliografie

Autorul dorește să mulțumească Google și Wikipedia pentru modul în care au revoluționat cercetarea bibliografică

[1] Academia Română, Institutul de lingvistică "Iorgu Iordan" – Dicționarul explicativ al limbii române (ediția a II-a revizuită și adăugită), Editura Univers Enciclopedic Gold, 2009
[2] Winston, Patrick Henry. Artificial Intelligence. 3rd ed. Addison-Wesley, 1992. ISBN: 9780201533774.
[3] Patrick Winston. *6.034 Artificial Intelligence.* Fall 2010. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.
[4] Russell S., Norvig P. – Artificial Intelligence – A Modern Approach.Third Edition, Pearson-Prentice Hall, ISBN-13: 978-0136042594, 2009
[5] Turing, A. (1950), "Computing Machinery and Intelligence," *Mind*, 59 (236): 433–60.
[6] Copeland, Jack (18 June 2012). "Alan Turing: The codebreaker who saved 'millions of lives'". BBC News Technology. Retrieved 26 October 2014
[7] Bolhuis JJ, Tattersall I, Chomsky N, Berwick RC (2014) How Could Language Have Evolved? PLoS Biol 12(8): e1001934. doi:10.1371/journal.pbio.1001934; https://chomsky.info/20140826/, accersat la 04 ian 2017
[8] Brooks, Rodney, articol de Kahn, Jennifer (Martie 2002). "It's Alive". Wired (10.30). https://www.wired.com/2002/03/everywhere/ accesat 6 ian 2017.
[9] Bostrom, Nick - AI set to exceed human brain power articol CNN.com (iulie 26, 2006) http://edition.cnn.com/2006/TECH/science/07/24/ai.bostrom/ accesat 6 ian 2017
[10] Ulam, Stanislaw - "Tribute to John von Neumann". 64, #3, part 2. Bulletin of the American Mathematical Society: 5; (May 1958).
[11] Armstrong, Stuart - "How We're Predicting AI", from the 2012 Singularity Conference
[12] McCorduck, Pamela - Machines Who Think (2nd ed.), Natick, MA: A. K. Peters, Ltd., ISBN 1-56881-205-1, OCLC 52197627, 2004

[web 1] https://en.wikipedia.org/wiki/File:Turing_Test_version_3.png accesat in 2 ian 2017, Creative Commons, Public domain
[web 2] https://en.wikipedia.org/wiki/Instructions_per_second#MIPS, Intel InGaAs FTW (October 16, 2016). "Core i7-6950X Extreme Edition Deca Core benchmarks. About 6% IPC gains over Haswell-E". cpubenchmark, passmark, cpuboss,geekbench,cinebench
[web 3] Google self driving car; accesat la 07 ianuarie 2017
https://commons.wikimedia.org/wiki/File:Google_self_driving_car_at_the_Googleplex.jpg
[web 4] Automatic license plate recognition, accesat la 10 ianuarie 2017;
http://www.platerecognition.info/1103.htm
[web 5] IBM Research on Biometrics, accesat la 10 ianuarie 2017
http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=1926
[web 6] Imagini din domeniu public, accesat ianuarie 2017, pixabay.com
[web 7] https://en.wikipedia.org/wiki/Tay_(bot) accesat august 2017
[web 8] https://en.wikipedia.org/wiki/History_of_artificial_intelligence accesat august 2017

Reprezentari. Spatiul starilor

În acest capitol se prezinta despre diverse posibilități de reprezentare a cunoașterii în memoria internă a calculatorului, dar și tehnicile folosite în dezvoltarea unor programe inteligente care au avut un impact major in inteligența artificiala de-a lungul istoriei sale. De asemenea se prezinta aspecte legate de sistematizarea procesului de proiectare a unui agent inteligent.

Structuri de date folosite in reprezentarea informatiei

In acest subcapitol se prezinta o sistematizare a tipurilor de structuri de date folosite in programare pentru reprezentarea informatiei.

Structuri de date elementare (engleza: Primitive Data Types)

- Boolean, true sau false.
- Caracter.
- Floating-point, valori reale cu simpla precizie.
- Double, valori reale intr-un interval mai mare decat simpla precizie.
- Integer, valori intregi sau cu precizie fixa.
- String, o secventa de caractere.
- Referință (numit si reference, pointer sau handle), o valoare numerica ce face referinta la adresa unui obiect memorat într-o zona de memorie .

Structuri de date compuse (engleza: Compozite Data Types)

- Șir sau Vector (engleză: Array)
- Structura (înregistrare sau tuplu) (engleză: record, tuple, structure)

Exemplu (C++):

```
struct Account {
  int account_number;
  char *first_name;
  char *last_name;
  float balance;
};
```

Structuri de date abstracte (Abstract Data Types ADT)

Sunt structuri de date construite odata cu scrierea programului utilizând structuri elementare. Acest tip de structuri reflectă abordarea cu privire la implementarea algoritmului de catre programator. Nu sunt structuri formal definite in interiorul limbajului de programare asa cum sunt cele elementare, este sarcina programatorului sa le construiasca adaptat la cerintele problemei de rezolvat. Exista cateva tipare care s-au dezvoltat de-a lungul vremii, cum ar fi:

- Container sau clasa – este un depozit organizat de mai multe obiecte sau variabile
- Listă sau Listă conectată (linked list)
- Associative array (in care indexul poate fi diferit de un numar)
- Set (multime neorganizata de variabile, nu permite existenta dublurilor; întâlnită atunci când se folosește zona de Teoria Mulțimilor din Matematică)
- Multiset (sacoșă; engleză: Bag) (o multime neorganizata de variabile care permite existenta dublurilor)
- Stivă (engleză Stack) (elementele sunt adaugate/eliminate doar la un capăt al sirului: LIFO- Last In First Out)
- Coadă (engleză: Queue) (structura de date in care adaugarea se face la un capat al sirului, si eliminarea la celalalt capat; FIFO – First in, First out)
- Graf – structură de date construită pentru a implementa un graf neorientat sau orientat (asa cum este definit în Teoria Grafurilor din Matematică). In acest sens se utilizează conceptul de

listă inlănțuită (Linked List). Terminologia folosită pentru grafuri este identica cu cea folosită în Matematică. Se utilizează conceptul de noduri (puncte) conectate prin arce (linii), în fiecare nod putând fi memorată informație. Un exemplu de graf este prezentat in Fig.1

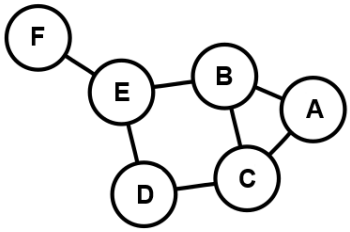

Fig.1. Exemplu de graf

- Arbore (engleză: Tree) – structura de date prin intermediul careia se pot modela structuri ierarhice. Arborii contin un nod numit rădăcină și subarbori numiți copii, reprezentați ca o mulțime de noduri înlănțuite. Un arbore este considerat un graf fără cicluri, graful din Fig.1 nefiind arbore din motiv că se poate forma ciclul ABC. Un exemplu de arbore este prezentat în Fig. 2

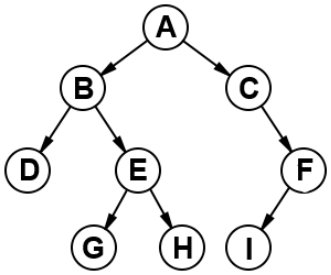

Fig.2 Exemplu de arbore

Terminologia folosită in descrierea arborilor este identică cu cea folosită în Matematică. Astfel putem distinge următoarele notiuni:

- Radacina (Root) – nodul din partea de sus a unui arbore.
- Copil (Child) – un nod conectat la un alt nod, atunci când ne deplasăm din Radăcină.
- Părinte (Parent) – opusul notiunii de copil.
- Frați (Siblings) – grup de noduri cu același părinte.
- Descendent – un nod ce poate fi atins repetând deplasarea de la părinte la copil
- Ascendent - un nod ce poate fi atins repetând deplasarea de la copil la părinte.
- Frunză (Leaf) – nod numit si nod extern, un nod fără copii
- Muchie, Ramură, Salt (Edge) – legătura între două noduri.
- Cale (Path) – secvență de noduri și muchii care conectează un nod cu un descendent
- Nivelul (Level) – unui nod este dat de numarul de conexiuni intre rădăcină și nodul curent, la care se adaugă cifra 1.
- Adâncimea arborelui (Depth) – numarul de muchii de la radacină la cea mai îndepărtată frunză
- Factor de ramificare (Branching Factor) – numărul maxim de copii pe care îi are un nod. Arborele din Fig. 2 are un factor de ramificare de 2

<u>Exemplu de construire a unei Linked List, Graf, Arbore</u>

Linked list este o structura de date ce faciliteaza in special memorarea unor structuri de tip lista, stiva, graf sau arbore.

In mod grafic, o linked list este prezentata in Fig. 3. Aici se pot distinge cele doua elemente caracteristice ale Linked List:

- zona de date, in cazul de fata datele memorate sunt 12, 99, 37
- zona de pointer catre urmatorul element din lista, in cazul de fata este vorba de sageata catre urmatorul element din lista

Structura realizata in conceptiunea linked list si prezentata in Fig. A1 este o structura de tip lista, in mod similar putandu-sa concepe grafuri sau arbori, adaugand elemente la zona de poiner.

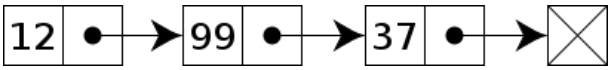

Fig. 3 Realizarea unei structuri tip lista utilizand conceptul linked list

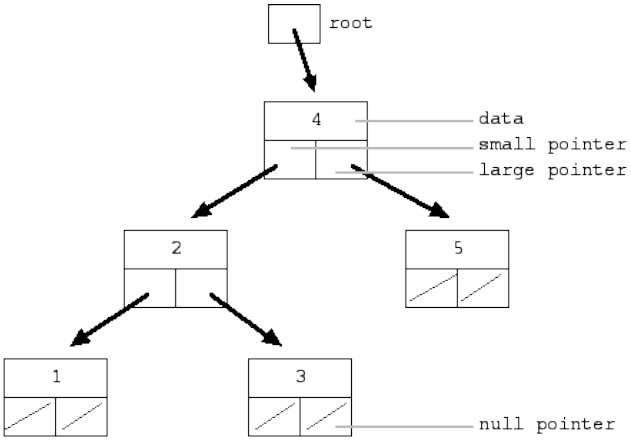

Fig. 4 Realizarea unei structuri tip arbore utilizand conceptul linked list
[http://cslibrary.stanford.edu/109/TreeListRecursion.html]

In Fig. 4 se reprezinta vizual o metoda de realizare a unei structuri de tip arbore utilizand conceptul “linked list”, modul in care se implementeaza structura tine de fiecare programator/designer in parte.

Metodele programatice de realizare a acestor structuri de date nu sunt in obiectivele acestei carti, ele facand parte dintr-un curs de programare, orientat pe un anumit tip de limbaj (C/C++, java, python etc)

Reprezentari interne ale cunoasterii si a spatiului starilor

In capitolul Introducere se prezinta obiectivul Inteligentei Artificiale ca fiind rezolvarea computerizata a unor probleme pentru care nu exista o procedura explicita de rezolvare codificata in calculator. În obținerea acestui deziderat este foarte important modul de reprezentare interna a cunostintelor de care este nevoie, pentru a ii permite sistemului de calcul sa identifice o solutie. Aceste cunostinte despre starea lumii trebuie trebuie sa fie codificate de asa maniera incat sa poata sa fie procesate prin intermediul unui sistem de calcul pe baza unui algoritm. Practic, o secventa potrivita de stari poate duce agentul din starea curenta in starea tinta. Toate starile pe care le poate avea o "lume" se numesc spatiul starilor.

Inainte de construirea unui agent inteligent, si a unei dezbateri cu privire la modul de reprezentare interna a cunoasterii, la nivel conceptual, trebuie sa existe o structura clara a functiilor pe care acesta trebuie sa le rezolve, o metoda de masurare a performantei agentului, care sunt elementele percepute si actiunile posibile din/asupra mediului, respectiv structura mediului in care agentul isi desfasoara activitatea. Se propune crearea unui tabel PAGE (Percepts, Actions, Goals, Environment)

Tipul de Agent	Percepts	Actions	Goals	Environment
Diagnoza mecanica	Simptome, rezultate teste, Raspunsuri ale soferului	Intrebari, Teste, Reparatii	Vehicul functional in parametri, Costuri minimizate	Vehicul, Atelier
Sistem de identificare a figurii	Pixeli de intensitate si culoare diferita	Identificarea fiecarei persoane din imagine	Identifiare corecta	Imagini de la aparatul foto
Robot Pick&Place	Pixeli de intensitate diferita	Preia obiecte, le alege si apoi le depune in recipiente	Obiecte depuse corect	Conveior cu obiecte
Sistem de control in centrala termica	Citire senzori gaz, presiune si temperatura	Deschide-inchide valve, ajusteaza temperatura	Mentinerea temperaturii si sigurantei	Centrala termica
Sofer automat	Semnal GPS, Senzori	Acceleratie, frana, rotirea volanului	Sosirea in siguranta la destinatie	Drumuri, trafic

Tabel 1. Tabel PAGE pentru diversi agenti

Folosind sistematizarea PAGE se poate trece la un caz concret. In [4] cap.3, Russell si Norvig propun un agent tip aspirator care "traieste" intr-un mediu (o lume) simplificat, care contine doar doua incaperi, in care poate exista sau nu praf, aspiratorul aflandu-se intr-una din cele doua incaperi. Toate opt starile posibile in care poate fi lumea (Environment) sunt reprezentate in Fig.2.5. Actiunile posibile ale agentului sunt Left, Right, Suck, iar Obiectivul este o lume curata – reprezentata prin starile [7,8]. Se disting doua categorii de probleme:

1. Senzorii agentului ii permit sa isi perceapa mediul si stie in ce stare se afla, aditional acesta stie rezultatul actiunilor sale, ca urmare, agentul poate sa isi determine (calculeze) precis starea dupa o serie de actiuni – concept denumit "Lume e accesibila" (The world is accessible). De exemplu, daca se

afla initial in starea {5}, acesta poate calcula ca secventa de actiuni {Right, Suck} il va duce in starea tinta (obiectiv). Acest tip de problema este denumita “single-state problem”, in sensul ca starea agentului in lume este determinata univoc.

2. Agentul nu stie starea lumii, in schimb stie care este rezultatul actiunilor sale. In exemplul aspiratorului, acesta poate fi in oricare din starile {1,2,3,4,5,6,7,8}, dar o actiune de tip {Right} ii va reduce din incertitudine, introducand ca stari posibile {2,4,6,8}, in final agentul poate sa ajunga la concluzia ca o secventa {Right, Suck, Left, Suck} il aduce in starea tinta. Acest tip de problema este denumita “multiple-state problem”, in sensul ca starea agentului in lume nu este determinata univoc.

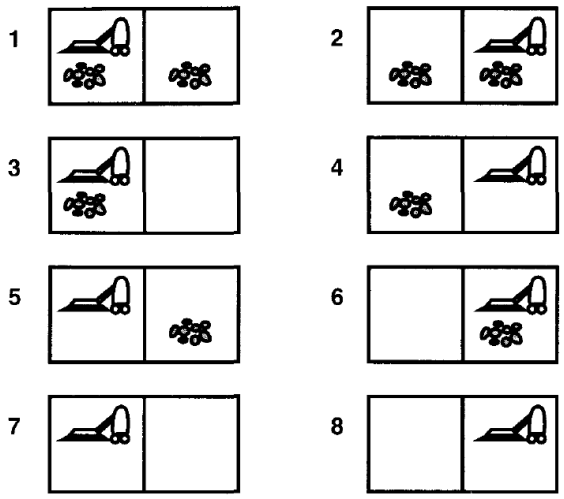

Fig. 2.5 Starile Posibile pentru Lumea Aspiratorului [4]

O posibilitate de codificare a lumii aspiratorului (corespunzatoare starii [4]) este folosirea unei matrici care sa primeasca valori, Fig. 2.6, iar actiunile sa induca modificari in matrice, de exemplu actiunea {Left} sa mute aspiratorul in Incaperea 1.

$$\begin{array}{r} \\ \text{Pozitie Aspirator} \\ \text{Praf} \end{array} \begin{array}{c} \begin{array}{cc} \text{Incaperi} & \\ 1 & 2 \end{array} \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{array}$$

Fig. 2.6 Codificare pentru Lumea Aspiratorului

Un alt exemplu de problema este: Un fermier doreste sa mute peste un rau o vulpe, o gasca si un sac de porumb. Din nefericire, barca sa este atat de mica, incat in ea incape doar fermierul si un obiect la o deplasare. In mod suplimentar, daca raman pe o parte a raului nesupravegheate de fermier, vulpea mananca gasca, si gasca mananca porumbul. Cum trebuie sa procedeze fermierul?

Daca se pune problema reprezentarii cunostintelor intr-un mod eficient, se observa ca limbajul verbal nu este reprezentarea potrivita pentru rezolvarea problemei. O alta reprezentare, potrivita pentru a rezolva problema, este cea folosita in Fig.2.7

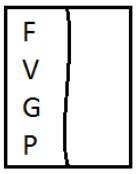

Fig. 2.7 Reprezentare a problemei Fermierului

Folosind aceasta reprezentare se pot genera toate starile in care poate fi lumea fermierului. Numarul acestora poate fi calculat tinand cont ca fiecare obiect (inclusiv fermierul) are 2 pozitii, sunt 4 obiecte, si se doreste calculul tuturor posibilitatilor: 2^4=16.

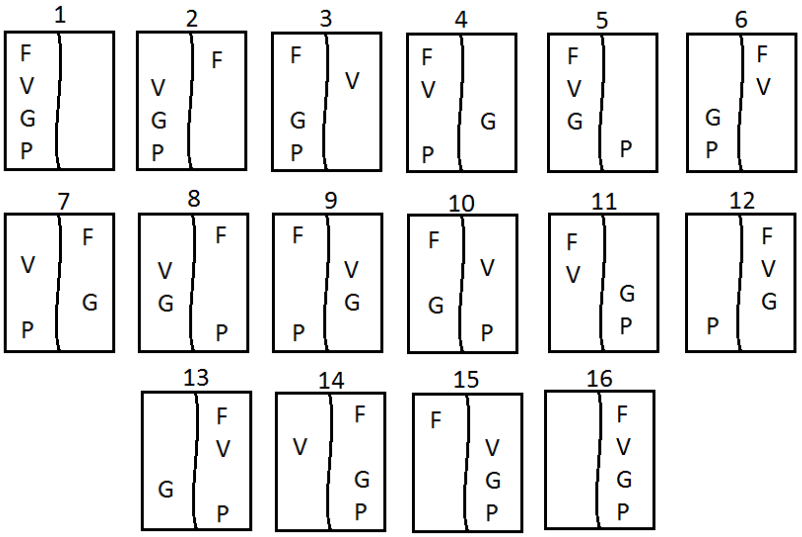

Fig. 2.8 Spatiul Starilor pentru problema Fermierului

Din cele 16 Stari, 6 sunt invalide {2,6,8,9,11,15} urmand ca cele ramase sa duca, eventual la solutie. Pentru a ajunge in starea tinta, agentul trebuie sa actioneze, producand astfel urmatorul graf de situatii si tranzitii intre starile lumii reprezentat in Fig. 2.9. tinand cont in acelasi timp de constrangeri.

Fig. 2.9 Graf de situatie si tranzitie intre starile lumii

O reprezentare interna in care se codifica folosind 0 pozitia unui obiect in stanga si 1 pozitia pe dreapta, ar putea fi una matriciala dupa cum se prezinta in Fig.2.10:

$$\begin{matrix} F \\ V \\ G \\ P \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Fig. 2.10 Reprezentare Matriciala a Problemei (starea {7})

Constrangerile impuse prin enuntul problemei intre obiecte si mediu se pot construi prin reguli tip IF-THEN dupa cum urmeaza:

a. Fermierul genereaza miscare si conduce barca, secventa de stari alese trebuie sa fie astfel incat doua stari consecutive nu pot avea aceeasi valoare pentru pozitia Fermierului (F)

b. Fermierul nu poate transporta decat pe sine si eventual un obiect aditional (Total 2 obiecte), constrangere care poate fi codificata folosind urmatoaea relatie

valoare absouta (Suma pe coloana(Stare curenta) – Suma pe coloana(Stare urmatoare)) <=2

c. Conflictele dintre obiecte pot fi modelate prin reguli tip IF in felul urmator, daca doua obiecte consecutive (V-G, G-P) au valoare diferita decat cea a Fermierului, atunci, starea este imposibila (fermierul e pe partea cealalta de rau)

De asemenea trebuie avut in vedere ca sa se evite intrarea intr-o stare care a fost deja adaugata arborelui, pentru a nu se produce bucle infinite.

Se poate observa din exemplul prezentat mai sus faptul ca folosind o asemenea arhitectura se poate construi un agent intelignet care sa se poata descurca in aceasta “lume” cu elemente si constrangeri reduse.

Arhitectura programului de integrare simbolica SAINT

SAINT [ref] (Symbolic Automatic INTegrator), un agent inteligent (program de calculator) a fost dezvoltat de catre James Slagle in 1961 ca Teza de Doctorat, cu scopul de a reproduce modul de gandire al unui student de anul I atunci cand trebuie sa rezolve probleme de integrare simbolica.

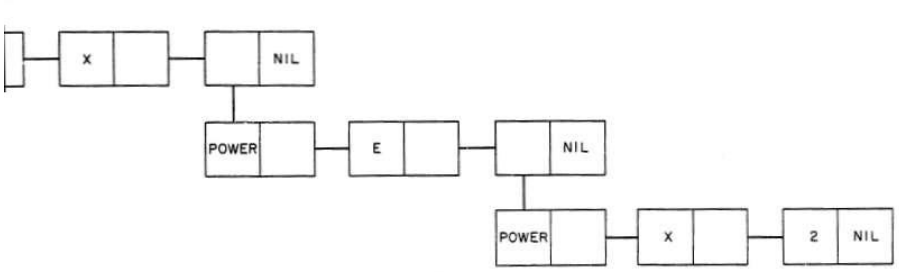

Fig. 2.11 Mod de memorare al unei expresii SAINT

SAINT a rezolvat 52 din 54 de probleme propuse cu o medie a timpului de computatie de 2 minute pentru o problema iar mediul de programare in care a fost scris SAINT este LISP (LISt Processor). Pentru scrierea unei functii matematice se foloseste sintaxa speciala descrisa de Slagle in Teza. Expresia $x \cdot e^{x^2}$ este introdusa in calculator sub forma (TIMES, X, (POWER, E, (POWER, X, 2))) si reprezentata in memorie asa cum se prezinta in Fig.2.11, unde fiecare bloc reprezinta o zona de memorie.

De asemenea SAINT a fost "inzestrat" cu unele proceduri cum ar fi operatii cu polinoame (adunare, scadere, inmultire, impartire), identificarea de tipare intr-o expresie (de exemplu sin(x), cos(x), power(x) …)

Pentru exemplificare Slagle propune integrala (1) spre a fi rezolvata de catre SAINT explicand in acelasi timp modul in care algoritmul functioneaza.

$$\int \frac{x^4}{\left(1-x^2\right)^{\frac{5}{4}}} dx \tag{1}$$

Rezultatul acestei integrale este prezentat in relatia (2)

$$\arcsin(x)+\frac{1}{3}\tan^3(\arcsin(x))-\tan(\arcsin(x)) \tag{2}$$

Algoritmul prin care SAINT abordeaza rezolvarea problemei initiale este una de descompunere in probleme mai simple, pana cand se ajunge la forma standard de integrala care este inlocuita in mod direct. Tabelul propus de Slagle foloseste 26 de integrale din care se folosesc doar cele prezentate in Tabelul 2. De notat este ca, constantele de integrare se adauga la finalul rularii algoritmului.

a.	$\int \frac{1}{x} dx = \ln x$
b.	$\int x^n dx = \frac{x^{n+1}}{n+1}$
c.	$\int \cos x dx = \sin x$

Tabel 2. Integrale standard folosite pentru rezolvarea exemplului prezentat de Slagle in [ref]

In general, problemele propuse nu se afla in forma standard, ca urmare este nevoie de manipularea acestora. Slagle propune un set de Transformari Sigure prezentate restrans in Tabelul 3.

1.	Semnul "-" scos in exteriorul integralei (pe motiv ca LISP codifica semnul in mod special)
2	Extragere de constante: $\int cf(x)dx = c\int f(x)dx$
3	Sumele de integrale descompuse: $\int \sum f_i(x)dx = \sum \int f_i(x)dx$
4	Diviziune polinomiala, cand se poate

De multe ori Transformarile sigure nu produc nici un efect asupra problemei, sau nu se pot aplica, pentru aceste cazuri Slagle propune efectuarea de incercari (cum ar fi schimbare de variabila, trecere la forma trigonometrica etc) numindu-le Transformari euristice. Cele mai importante asemenea Transformari euristice sunt prezentate in Tabelul 4.

A	O functie ce aplica lui x mai multe functii trigonometrice sa fie scrisa sub forma $f(\sin x, \cos x, \tan x, \cot x, \sec x, \csc x) = g(\sin x, \cos x) sau$ $g(\tan x, \csc x) sau\ g(\cot x, \sec x)$

B	Se noteaza $y = \tan x$; $\int f(\tan x)dx = \int \frac{f(y)}{1+y^2}dy$
C	Daca in expresie apare $1-x^2$ se noteaza $x = \sin y$ $1+x^2$ se noteaza $x = \tan y$

Tabel 4. Transformari Euristice

Algoritmul functioneaza prin aplicarea iterativa de transformari standard, sigure si euristice. Folosind acest proces, problema se descompune, iar pentru a tine evidenta tuturor elementelor se foloseste o structura arborescenta ca posibilitate de memorare interna a cunoasterii, prezentata in Fig.2.12

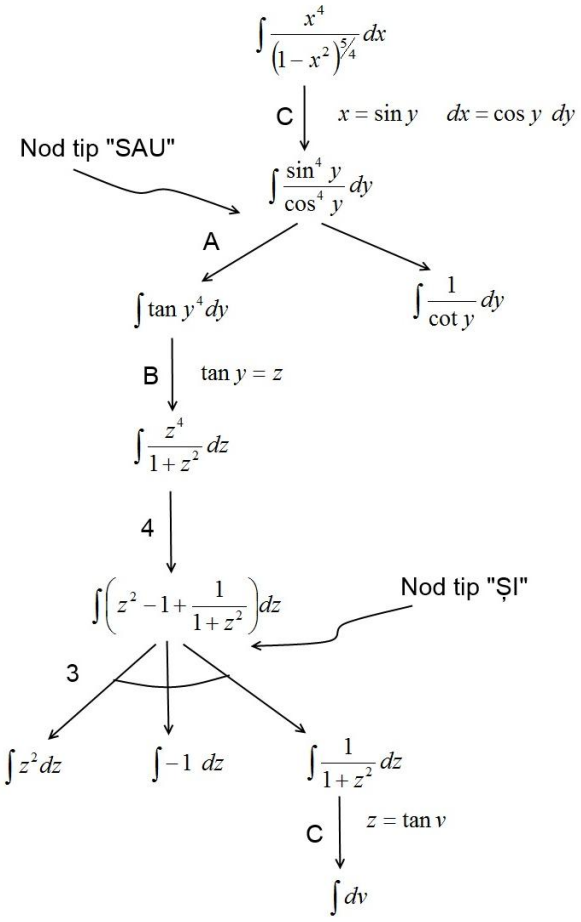

Fig.2.12 Arborele dezvoltat de SAINT in rezolvarea problemei propuse

In dezvoltarea produsa de algoritmul de solutionare al problemei se pot observa noduri de “SAU” si noduri tip “SI”. De asemenea, pentru obtinerea solutiei finale, se tine cont de toate transformarile de variabila efectuate.

SAINT a dovedit ca este posibila construirea unor programe care pot sa rezolve unele categorii de probleme folosind o abordare de memorare simbolica a informatiei. Totodata SAINT face parte dintr-un grup mai larg de programe de prelucrare simbolica a informatiei scise la sfarsitul anilor 50 din dorinta de a construi masini inteligente.

BoxWorld si SHRDLU

SHRDLU este un program dezvoltat in cadrul MIT intre anii 1969-1970 de catre Terry Winograd pentru a studia posibilitatea omului de a interactiona prin limbaj natural cu calculatorul, in de fata folosind limba engleza.

SHRDLU trebuia sa accepte comenzi in limba engleza, sa le execute si sa dea explicatii legate de actiunile sale. Winograd impreuna cu colaboratorii sai au conceput o lume redusa, numita in comunitate micro-world, in care un brat robotic preia si depune cutii una peste alta. Acesta trebuie sa se descurce in situatii cand o cutie este acoperita, sau are forma piramidala si nu poate fi acoperita.

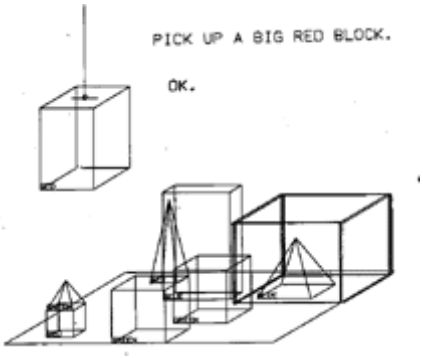

Fig. 2.13 Imagine a BoxWorld [http://hci.stanford.edu/winograd/shrdlu/]

In [Winston] cap.7 se propune o arhitectura simplificata a algoritmului dupa care BoxWorld functioneaza, aceasta fiind reprezentata in Fig.2.14

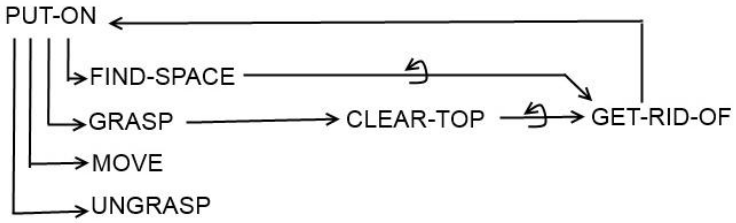

Fig.2.14 Procedurile utilizate de BoxWorld in planificarea miscarii [Winston]

Se observa in Fig.2.14 ca procedura GET-RID-OF apeleaza procedura PUT-ON fapt ce duce la aparitia recursivitatii si la dezvoltarea unui comportament complicat (cu toate ca structura programului este simpla). Similar strategiilor folosite in programele anterioare, acest program descompune o problema complicata in probleme mai simple. Se propune ca exemplu rularea programului pe o structura simpla, ilustrata in Fig.2.15

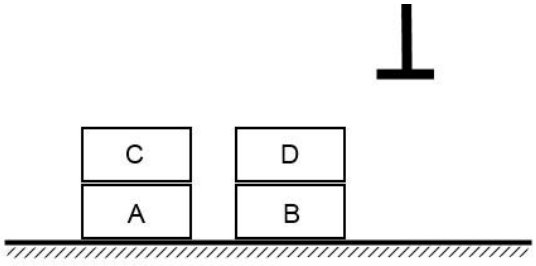

Fig. 2.15 Exemplu de aranjament BoxWorld

Pentru aranjamentul din Fig.2.15 se lanseaza comanda PUT-ON A B, ca urmare se aplica strategia de planificare a miscarilor si rezulta arborele de actiuni catre obiectiv, prezentat in Fig. 2.16.

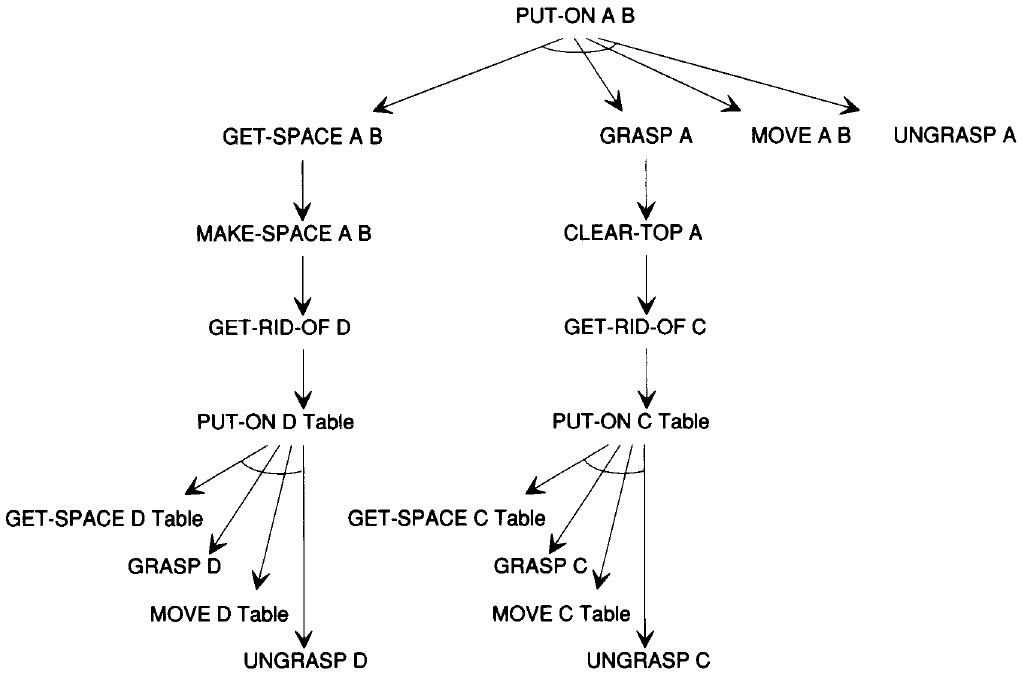

Fig.2.16 Arbore de actiuni catre obiectiv [Winston]

BoxWorld (ca subprogram al lui SHRDLU) are ca obiectiv planificarea miscarilor astfel incat sa se poata realiza obiectivul, insa SHRDLU a fost conceput pentru a fi capabil sa comunice folosind un limbaj natural cu utilizatorul si sa execute comenzile primite de la utilizator. Programul este capabil, inclusiv, sa raspunda la intrebari legate de comportamentul sau folosind urmatoarea strategie, de exemplu, pentru intrebarea:

I: WHY DID YOU CLEAR TOP OF A?
Raspunsul Agentului este sintetizat prin analiza procedurii din arbore situata in amontele procedurii invocate - CLEAR-TOP A, producand urmatorul mesaj: BECAUSE I WANTED TO GRASP A

Un alt tip de intrebare este
I: HOW DO YOU CLEAR THE TOP OF A?
Raspunsul este produs de Agent identificand procedura care urmeaza dupa comanda CLEAR-TOP A, rezulta astfel urmatorul text: I WANT TO GET RID OF C.

In felul acesta SHRDLU poate sa “discute” despre comportamentul sau intr-o maniera foarte apropiata de limbajul natural omenesc. Cu toate ca SHRDLU - BoxWorld este o lume restransa s-au putut demonstra unele strategii de obtinere a unui comportament natural din partea unui agent artificial.

Clasificarea tipurilor de medii in care poate evolua un agent

Din punct de vedere al mediului in care poate evolua un agent inteligent se poate distinge urmatoarea clasificare:

Observabilitatea mediului

- Medii complet observabile – senzorii ofera informatii complete despre starea mediului la orice moment in timp
- Medii partial observabile – senzorii ofera informatii incomplete (cu zgomot), sau nu au acces la portiuni din mediu
- Medii neobservabile – senzorii lipsesc sau nu au acces la informatiile din mediu

Numarul de angenti din mediu

- Agent singular, de exemplu situatia jocului de sudoku
- Multiagent, de exemplu jocul de sah este un mediu multiagent (este nevoie de doi agenti competitivi). In acest context al mediilor multiagent apar probleme legate de natura agentilor: competitivi, cooperativi si daca intre acestia exista comunicare.

Predictibilitatea mediului

- Deterministic, in sensul ca, daca se stie starea curenta a mediului, si se cunoaste actiunea ce urmeaza a fi efectuata atunci se va sti si starea urmatoare in care ajunge mediul (dupa efectuarea actiunii)
- Incert, daca mediul nu este complet observabil sau nu e deterministic
- Stohastic sau nondeterministic, atunci cand o actiune a agentului poate fi anticipata intr-o anumita masura masurata prin probabilitati (in sens statistic)

Mediu episodic sau secvential, in sensul ca unele evenimente se pot sau nu repeta

Mediu static sau dinamic. Daca mediul se modifica in timp ce agentul delibereaza, acest mediu se numeste dinamic, iar in caz contrar se numeste static

Mediu discret sau continuu, clasificare care se refera la modul in care este descrisa starea mediului. De exemplu, jocul de sah este un mediu discret, in care fiecare mutare produce o stare noua (nu exista jumatati de mutare), in schimb incercarea de a lovi o minge de tenis cu o racheta de tenis este un mediu continuu (numarul de variante in care ne este servita mingea este practic infinit)

Mediu cunoscut sau necunoscut proiectantului. Cateodata proiectantul nu este cunoscator al legilor ce guverneaza un anumit mediu, de exemplu, proiectantul nu stie complet legile fizicii, ca urmare, agentul trebuie sa invete sa se comporte in mediu inainte ca acesta sa ajunga la eficienta optima.

Aceasta clasificare a tipurilor de medii in care poate evolua un agent vine in ajutorul proiectantului avand in vedere ca unele strategii se aplica unor tipuri de medii, iar altele nu.

Concluzii Capitolul 2

- In rezolvarea computerizata a unei probleme, alegerea modului de reprezentare a datelor este esentiala, in acest sens s-au prezentat cateva exemple de aplicatii dezvoltate de-a lungul vremii.
- Pentru a sistematiza procesul de proiectare al unui agent inteligent se propune utilizarea unui tabel PAGE (Percepts, Actions, Goals, Environment), in acest fel, constrangerile asupra agentului devin evidente.
- Structurile de date folosite in reprezentare pot fi simple: variabile, matrici etc. dar si complexe: liste, stive, grafuri sau arbori. Pentru realizarea stucturilor complexe se propune folosirea variabilelor alocate dinamic – pointeri si crearea unei “linked list”.
- In multe situatii se pot construi proceduri care pot fi rulate in mod repetat si, in acelasi timp, ofera o arhitectura clara/limpede programului.
- Delimitarea intre starile in care se poate afla un agent care doreste sa rezolve o problema se poate face prin folosirea regulilor de tip IF-THEN. In capitolul despre Retele Neuronale, se vor prezenta metode specifice de clasificare a starilor, specifice Retelelor Neuronale.
- S-a prezentat o clasificare a tipurilor de medii in care poate sa evolueze un agent

Cunoastere. Rationament. Sisteme "Expert" bazate pe reguli.

Una dintre trasaturile caracteristice pe care trebuie sa le aiba un agent inteligent este capacitatea de a intelege mediul in care evoluează, dar și consecintele actiunilor sale. Omul utilizează în acest sens doua componente, prima este o baza de cunostinte despre mediu, iar a doua este un proces de rationament prin care se pot anticipa efectele actiunilor executate sau deduce informatii noi despre mediu, în concluzie, cele doua componente, baza de cunostinte si rationamentul trebuie ambele modelate pe calculator pentru a înzestra un agent inteligent cu capacități de înțelegere a mediului său.

O bază de cunoștințe (engleză: knowledge base), utilizată în zona inteligenței artificiale, este un depozit structurat sau nestructurat de informații folosit de către un sistem de calcul. Aceasta poate fi asemănată unei baze de date, care, în plus, este capabilă să memoreze reguli de relaționare între categorii de obiecte.

Rationament, din punctul de vedere al Dictionarului explicativ al limbii romane, se defineste ca o înlănțuire logică de judecăți, care duce la o concluzie. Substratul matematic al rationamentului poate fi scos in evidenta apeland la zona din matematica numita logica.

De-a lungul istoriei s-au dezvoltat mai multe tipuri de logica, fiecare cu idealizarile sale, astfel se pot distinge:

Tip de logica	Ontologie	Valori atribuite
Logica Propozitionala	Fapte	Adevarat/fals/necunoscut
Logica de rang unu	Fapte, Obiecte, Relatii	Adevarat/fals/necunoscut
Logica temporala	Fapte, Obiecte, Relatii, Timp	Adevarat/fals/necunoscut
Teoria probabilitatilor	Fapte	Probabilitati [0,1]
Logica fuzzy (neclara)	Grade de adevar	Valori [0,1]

Tabel 1. Tipuri de logica [RN]

Ontologia este o ramură a filosofiei care studiază ființa, respectiv, trăsăturile generale ale existenței. Acest termen este folosit datorita faptului ca se doreste modelarea printr-o metoda sau alta a lumii inconjuratoare.

Logica propozitionala presupune existenta unor propozitii care descriu starea lumii. Aceste propozitii pot avea valori de adevarat, fals sau necunoscut. Fiecare propozitie poate fi notata cu o litera (sau combinatii de litere cu indice), exemplificat în Tabelul 2.

Notatia propozitiei	Propozitia
C	Circul este minunat
I	Ioan iubeste circul
J	Ioan poarta vesta

Tabelul 2. Exemple de tipuri de propozitii

Fiecare propozitie este tratata ca un element indivizibil (atomic), si nu intereseaza elementele care alcatuiesc propozitia, precum subiecte, atribute sau actiuni. De asemenea se introduc operatori logici de compunere a doua sau mai multe propozitii, acestia sunt prezentati in Tabelul 3

Denumire	Simbol	Operator
Negatie	¬	NOT
Conjunctie	$\wedge$	"Și"
Disjunctie	$\vee$	"SAU"
Implicatie/Conditie	$\rightarrow$	daca …. atunci …. (in engleza: if …. then ….)
Echivalenta	$\leftrightarrow$	…. daca si numai daca …. (… echivalent ….)

Tabelul 3. Operatori folositi in Logica

Plecand de la propozitii simple, ca cele prezentate in Tabelul 2, se pot construi propozitii complexe folosind operatorii enumerati in Tabelul 3, de exemplu in Tabelul 4:

Reprezentare la nivel de cunoastere	Reprezentare Logica folosind notarea propozitiilor
Ioan iubeste circul ȘI Ioan poarta vesta	$I \wedge J$
Circul este minunat DACA SI NUMAI DACA Ioan iubeste circul	$C \leftrightarrow I$

Tabel 4. Exemple de propozitii compuse, formate din propozitii simple

Silogisme. Inferenta. Rationament computerizat in cadrul Logicii propozitionale

In paragrafele anterioare s-a descris modul in care se pot memora cunostintele in memoria calculatorului, acesta fiind primul pas pentru a obtine un agent inteligent. Pasul al doilea consta in a il inzestra pe agent cu capacitatea de a rationa. Pentru aceasta se foloseste operatorul logic de "Implicatie" "$\rightarrow$". De exemplu, se definesc doua propozitii:

La nivel lingvistic	La nivel de reprezentare logica-matematica
p1: Afara ploua p2: Ioan poarta vesta Se poate construi un proces rational folosind urmatoarea structura Daca Afara ploua atunci Ioan poarta vesta	p1: Afara ploua p2: Ioan poarta vesta p1 $\rightarrow$ p2 (p1 implică p2)

Prin inferenta sau silogism se intelege un tip de deducție în care o propoziție numită concluzie devine adevarata prin indeplinirea unei sau a mai multor propoziții numite premise (antecedent, ipoteza).

Daca, pentru demonstrarea unei propozitii sunt necesare mai multe procese de inferenta, se poate construi o structura arborescenta – un arbore de inferenta, asa cum se prezinta in Fig.1.

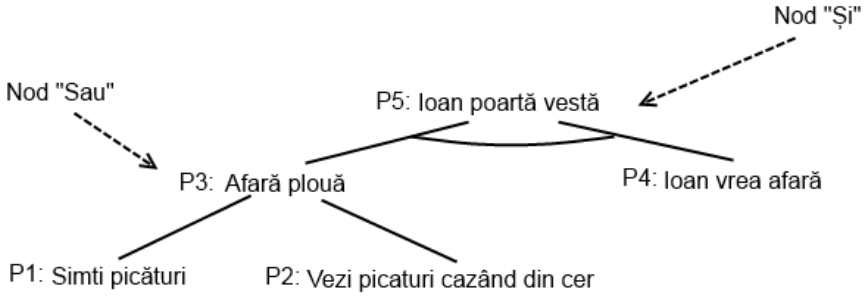

Fig.1 Arbore de inferenta

Din Fig.1 se observă nodurile tip "și" și nodurile tip "sau", acestea având ca efect faptul că P3 devine adevărată dacă P1 sau P2 e adevărat; în altă ordine de idei, P5 devine adevărat dacă P3 și P4 sunt adevărate în același timp.

Logica de rang unu (engleza: First-Order Logic) presupune descompunerea propozitiilor utilizate in "logica propozitionala" in elemente de tip OBIECT, PREDICAT (in sens de actiune asociata) și RELAȚIE. Pentru a intelege mai bine acest tip de Logica, se considera urmatorul exemplu in care se compara logica propozitionala cu logica de rang unu:

In logica propozitionlala:	sunt date exemplu doua propozitii [J,K] J: Ioan poarta vesta K: Marius poarta vesta
In logica de rang unu	este dat exemplu o propozitie [A] A: <cineva> poarta vesta <cineva> este o variabilă și poate lua valori din multimea [Ioan, Marius] In caz general: <OBIECT_tip_1><predicat><OBIECT_tip_2> <OBIECT_tip_1> poate fi [Ioan, Marius] <predicat> poate fi [poarta,arunca] <OBIECT_tip_2> poate fi [vesta, palarie]

Tabelul 4. Comparatie intre logica propozitionala si logica de rang unu

Din Tabelul 4, se poate observa descompunerea propozitiilor intr-o structura bazata pe obiecte si predicate. Structura enuntata anterior poate fi extinsa prin folosirea relatiilor dintre obiecte, cum ar fi, relatia tata(Ioan, Marius), coleg(Ioan, Marius), haine(vesta, sacou, pantalon, caciula) sau rotund(cerc, roata, soare).

Dat fiind aceasta structura bazata pe obiecte si relatii, se pot exprima proprietati pentru colectii intregi de obiecte, in detrimentul enumerarii obiectelor pe nume. In acest sens se introduce cuvantul cuantificator: cuantificator universal si cuantificator existential

Cuantificator universal (∀), este asemanator celui folosit in matematica si se citeste "Pentru toti …", sau mai simplu "Toti". De exemplu, folosind limbajul natural exprimam propozitia "Toti Studentii sunt Oameni", aceasta are un echivalent in Logica de rang unu:

$$\forall x \quad Student(x) \rightarrow Om(x) \tag{1}$$

expresia (1) tradusa in limbaj natural s-ar citi in felul urmator: „Pentru toti x, daca x este Student atunci x este Om".

Cuantificator existential (∃) echivalent cu cuvantul „exista" se poate folosi in exprimarea propozitiei „Ionica are un caiet in mana", echivalenta cu expresia formulata folosind Logica de rang unu:

$$\exists x \quad Caiet(x) \wedge InMana(x,Ionica) \tag{2}$$

altfel spus „Exista x astfel incat [(x sa fie caiet)=true] si [(x InMana lui Ionica)=true)]"

Asa cum „implicatia"este folosita in mod natural in cazul cuantificatorului universal, „conjunctia" se foloseste in cazul cuantificatorului existential.

Cuantificatori multipli se pot folosi de asemenea in scrierea unor expresii in Logica de rang unu. Daca se doreste exprimarea propozitiei „Toata lumea iubeste pe cineva", aceasta se poate scrie dupa cum urmeaza:

$$\forall x \ \exists y \quad Iubeste(x,y) \tag{3}$$

Pe de alta parte, daca se scrie

$$\exists x \ \forall y \quad Iubeste(x,y) \tag{4}$$

expresia (4) se interpreteaza ca „Exista cineva astfel incat sa iubeasca pe toata lumea", ca urmare, ordinea aparitiei Cuantificatorilor este esentiala in obtinerea rezultatului dorit.

Crearea si utilizarea unui „Sistem Expert"

Un „Sistem Expert" numit si „Sistem Expert Bazat pe Reguli" este un sistem computerizat (alcatuit din Hardware si Software) care imita procesul de decizie al unui expert uman. Aceste sisteme sunt proiectate pentru a emula procesul de rationament uman folosind reguli de tip IF-THEN. Primele „Sisteme Expert" au aparut la finele anilor 1970 dezvoltate de catre Edward Feigenbaum si sunt considerate ca fiind printre primele aplicații de succes ale Inteligenței Artificiale.

Cateva aplicatii dezvoltate in ideea „Sisteme Expert" sunt:

Dendral (1960) – considerat primul „Sistem Expert", a fost conceput pentru a propune o structura moleculara pentru unele substante analizate prin spectroscopie. Practic sistemului ii erau prezentate spectrogramele, iar acesta propunea o structura moleculara posibila pentru respectiva substanta. Acest proces era facut la vremea aceea de experti umani, tinand cont de un set de reguli.

MyCin (1970) – un program care emula modelul de decizie al unui medic specialist in boli infectioase precum meningita sau bacteriemie si recomanda antibiotice. MyCin propunea un tratament acceptabil in 69% dintre cazuri, fiind mai eficient decat medicii specialisti umani, iar pentru aceasta folosea un set de aproximativ 600 de reguli IF-THEN.

XCON (eXpert CONfigurator, 1978) – un program care asista personalul din vanzari al firmei DEC (Digital Equipment Corporation), vanzatorul cel mai mare de calculatoare/sisteme de calcul al acelor vremuri. Pe vremea aceea calculatoarele erau livrate pe componente, cu cabluri de legatura, conectori, adaptoare, software specific, rolul lui XCON fiind generarea automată a listelor de componente pentru o anumită configurație de calculator. Trebuie menționat că până atunci, listele de componente erau generate de către operatori umani.

Toate aceste „Sisteme expert" se construiesc în jurul a două componente de bază, un sistem de inferență (rationament) și o bază de cunoștinte (engleză: Knowledge Base) notată în continuare cu KB. Operațiile pe utilizatorul le poate impune sistemului, prin interfața om-calculator, se pot clasifica în două categorii:

- INTRODUCE (engleză: TELL) – comandă prin care se introduc informații noi in baza de cunoștințe sub formă de propoziții (Propoziție în sensul definit în Logica de Rang unu)
- INTREBARE (engleza: ASK) - în care se interogheaza „Sistemul Expert”, iar acesta afisează un răspuns.

De exemplu, în interacțiunea cu o bază de cunoștințe, se pot scrie următoarele comenzi pentru a adăuga informația: “Ioan este student”, „Mihai este om” și „Orice student este om”

01 TELL(KB, Student(Ioan))
02 TELL(KB, Om(Mihai))
03 TELL(KB, $\forall x$ Student(x) $\rightarrow$ Om(x))

De observat este faptul că în linia 01 și 02 se introduc în baza de cunoștințe AFIRMAȚII asupra situatiei în discuție, iar în linia 03 se introduce o REGULĂ prin care se stabileste că orice student este om. Aceste REGULI se memorează în baza de cunoștințe sub formă de declarații IF-THEN.

În final, după ce s-au adăugat informațiile, sistemul poate fi interogat dacă “Ioan este student”

ASK(KB, Student(Ioan)) iar rezultatul va fi True.

În continuare se vor folosi expresiile interogare (engleză: query) respectiv obiectiv (engleză: goal) pentru a face referire la toate acțiunile tip ASK asupra unui Sistem Expert.

Un alt tip de interogare poate sa fie “Este Ioan om?”

ASK(KB, Om(Ioan))

iar rezultatul ar trebui să fie și în acest caz True, chiar dacă această informație nu a fost memorată in mod explicit în baza de cunoștințe (KB). Pentru a obține acest rezultat, KB trebuie să fie capabilă să prelucreze expresia care face legătura între student și om. Procedeul prin care se poate obține un asemenea efect va fi detaliat în paragrafele următoare.

O bază de cunsștințe poate fi interogată și cu întrebări de forma “există cineva om?”

ASK_VARIABLE(KB, Om(x)) iar răspunsul în acest caz este {x/Mihai}, {x/Ioan}.

În acest caz ne interesează toate variabilele care sunt etichetate cu proprietatea “Om”.

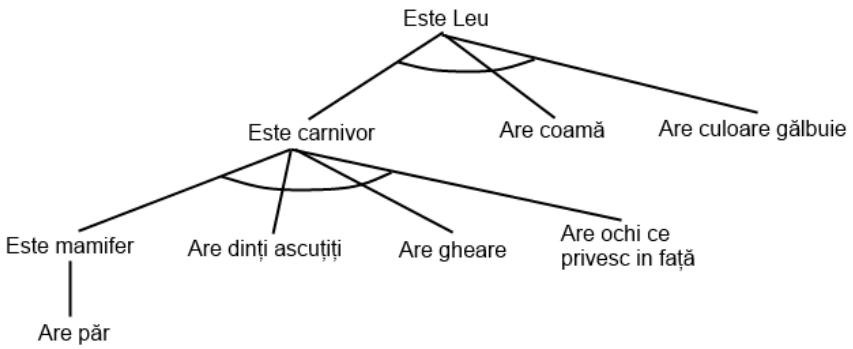

Fig. 2. Arbore de inferență pentru gradina zoologică

Pentru a ilustra modul de funcționare al unui Sistem Expert se propune ca exemplu un sistem de identificare a animalelor dintr-o grădină zoologică. De notat este faptul că în grădina zoologică există doar un număr limitat de animale, altfel spus DOMENIUL de valori este restrâns. Folosind informații ce pot fi extrase prin observare din mediul gradinii zoologice, se pot trage concluzii legate de identificarea unui element din domeniu și construi un arbore de inferență, așa cum se prezintă în Fig.2.

Sistemul Expert care produce arborele de inferență din Fig.2 este construit pe baza unor REGULI, după cum urmează:

R0: IF (?x are păr) THEN (?x este mamifer)
este echivalentul propoziției $\forall x$ ArePăr(x) $\rightarrow$ Mamifer(x)

R1: IF AND(?x este mamifer,
?x are dinți ascuțiți,
?x are gheare,
?x are ochi ce privesc in față) THEN (?x este carnivor)

R2: IF AND(?x este carnivor,
?x are coamă,
?x are culoare galbuie) THEN (?x este Leu)

Până aici s-au definit câteva reguli care permit unui obiect să devină “Leu”. Aceste reguli trebuie completate cu observații din mediu asupra obiectului. Odată făcute observațiile, ele vor fi introduse în baza de date sub forma unor propoziții și vor purta numele de AFIRMAȚII. În acest sens propunem un obiect numit „Simba”, ca urmare variabila ?x va primi valoarea “Simba”. Despre Simba putem extrage, prin observare folosind senzori și prelucrare de imagine, următoarele informații din mediul grădinii zoologice:

A0: Simba are păr
A1: Simba are dinți ascuțiți
A2: Simba are gheare
A3: Simba are ochi ce privesc în față
A4: Simba are coamă
A5: Simba are culoare gălbuie

Sarcina Sistemului Expert este să demonstreze propoziția: “Simba” este “Leu”. Pentru a realiza acest lucru, sistemul analizează fiecare regulă și o alege pe aceea care permite unui obiect să devină Leu, în cazul nostru este vorba despre R2, pentru că această regulă are consecința “?x este Leu”. De notat este faptul că ?x are valoare de variabilă și poate fi inlocuit prin orice obiect. În continuare se evaluează ipoteza prin care R2 se poate activa. În acest sens se observă că R2 se activează dacă “Simba e carnivor”, “Simba are coamă” și “Simba are culoare gălbuie”. **Dacă propozițiile “Simba are coamă” și “Simba are culoare gălbuie” sunt adevărate pentru că există afirmațiile A4 și A5, propoziția “Simba e carnivor” trebuie demonstrată.** Demonstrația se face in mod similar procesului prin care s-a început demonstrația că “Simba este Leu”. Procesul se repetă până când nu mai există propoziții de demonstrat sau propoziția obiectiv a fost demonstrată. O propoziție poate să fie demonstrată, primind astfel valoarea TRUE, sau nedemonstrată, primind valoarea FALSE. **Propoziția inițială “Simba este Leu” poate fi TRUE sau FALSE, în funcție de existența sau inexistența AFIRMAȚIILOR sau REGULILOR necesare pentru a o demonstra.**

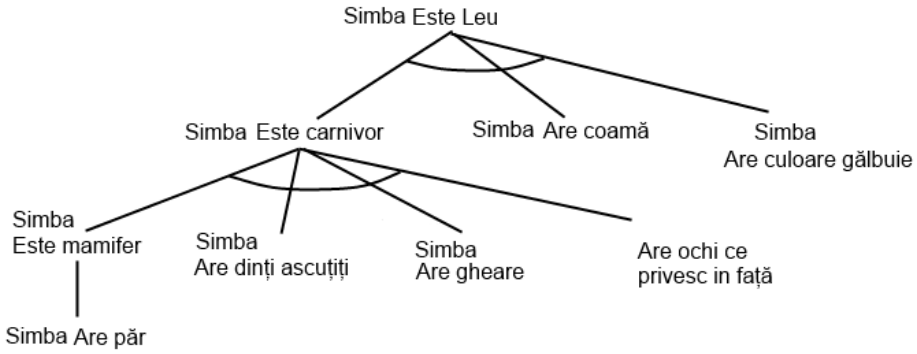

Fig. 3. Arbore de inferență pentru exemplul Simba este Leu

Procesul utilizat de un Sistem Expert pentru demonstrarea al unei afirmații pornind de la consecință la ipoteză se numește **Înlănțuire Înapoi** (engleză: BACKWARD CHAINING). În final, pentru demonstrarea propoziției "Simba este Leu" se construiește un arbore de inferență – Fig.3.

Un alt mod în care poate rula un Sistem Expert este **Înlănțuirea Înainte** (engleză: FORWARD CHAINING), adică abordarea regulilor de la ipoteză la consecință. Dacă un Sistem Expert rulează în acest mod, fiecare regulă care se poate executa va produce o afirmație nouă care va fi adăugată la baza de cunoștințe, procesul repetându-se până când nu mai există regula ce se pot executa. Pentru a sistematiza procesul descris mai sus se poate construi un tabel în care se introduce numărul iterației și propoziția adăugată la fiecare iterație – Tabelul 5.

Nr. iterație	Reguli ce se pot executa	Regula ce se execută	Afirmație nouă	Regulă ce expiră
1	R0	R0	A6: Simba este mamifer	R0-Simba
2	R0,R1	R1	A7: Simba este carnivor	R1-Simba
3	R0,R1,R2	R2	A8: Simba este Leu	R2-Simba

Tabel 5. Tabel obținut ca urmare a rulării a trei iterații de Înlănțuire Înainte

Se observă că la prima iterație, singura regulă ce poate fi executată este R0 iar afirmația nou obținută este „Simba este mamifer", de asemenea expiră regula R0 cu obiectul Simba. Putem să ne imaginăm că, în baza de cunoștințe există un al doilea obiect – Lăbuș, pentru care există afirmația „Lăbuș are păr" și, ca urmare, R0 s-ar putea executa și pentru Lăbuș, afirmația nouă fiind „Lăbuș este mamifer". Pentru a menține ordine în execuția regulilor și evitarea ciclurilor infinite, se folosește coloana „Regulă ce expiră".

Dacă se dorește stabilirea valorii de adevăr a unei propoziții cum ar fi „Simba este Leu" folosind Înlănțuirea Înainte, Sistemul Expert trebuie să ruleze până când se indeplinește una din condițiile următoare:

- se deduce propoziția căutată, caz în care propoziția de demonstrat va fi declarată True
- nu mai există reguli ce se pot executa, caz în care propoziția de demonstrat va fi declarată False

Structurarea cunoașterii

Pentru a pune ordine în procesul de identificare a cunoștințelor necesare unui Sistem Expert si a il face funcțional se propune un proces sistematic. Procesul se exemplifica printr-un agent de impachetat produse cumparate dintr-un magazin, facandu-se referire la ordinea de introducere a produselor în plasă [Winston]. Produsele ce urmează a fi împachetate sunt pâine, ouă, mazăre la conservă, mazăre înghețată, chipsuri de cartof, detergent.

Un om ar proceda în felul următor, ar pune prima dată mazărea la conservă, apoi pâinea, ouăle și chipsurile de cartofi, în ordinea fragilității. Pentru mazărea înghețată ar folosi o a doua plasă, iar pentru detergent ar folosi a treia plasă. Se observă că, pentru a fi eficient în procesul de împachetare este nevoie de cunoștințe specifice acestui proces. Pentru a extrage aceste cunoștințe, inginerul care dezvoltă Sistemul Expert e nevoit să discute cu persoanele care efectuează operatiile de impachetare, să le observe în acțiune și să pună întrebări pentru cazuri speciale, de exemplu:

Pentru detergent folositi tot timpul o plasă nouă ? Răspunsul putând fi, că, tot timpul, produsele nealimentare se introduc separat de cele alimentare.

Pentru mazăre înghețată folositi tot timpul plasa noua? Produsele inghetate se pun separat pentru ca sunt înghețate, iar gheața se poate topi și scurge. Cu această ocazie se poate introduce un nou atribut în baza de cunoștințe, anume “înghețat”, extinzându-se DOMENIUL de valori (în sensul Teorii mulțimilor din Matematică)

Se reține astfel câteva principii:

P1: Se poartă discuții și se observă experții, cei care execută fiecare operație.

P2: Se discută cazuri speciale, cum ar fi mazărea la conservă și mazărea înghețată

P3: Situatiile în care Sistemul greșește sau “crapă” (se oprește) sunt, in general cazuri în care este instruit să execute comenzi greșite, sau se ajunge intr-o situație neprevăzută, ca urmare sistemul se poate actualiza prin introducerea de informatii noi, care să acopere situația creată.

În general, procesul de structurare al cunoașterii se aplică de fiecare dată când se dorește eficientizarea unui proces, inclusiv prin automatizare. În limba engleză acest proces poartă numele de Knowledge Engineering.

Concluzii Capitolul 3

- o directie de dezvoltare a sistemelor cu inteligenta artificiala o prezinta sistemele bazate pe cunoastere. Aceste sisteme folosesc doua tipuri de resurse, o baza de cunostinte si un proces de rationament. Procesul de rationament utilizat se bazeza pe partea din matematica numita logica.
- Se prezinta aspecte legate de logica propozitionala (logica de rang zero), care apoi se dezvolta si se trece la logica de rang unu. In logica de rang unu se utilizeaza familii de obiecte si relatii intre acestea.
- Se descrie modul in care se poate costrui un sistem expert bazat pe cunostinte, pornind de la cum se poate construi baza de cunostinte dar si mecanismul de rationare.
- Se prezinta modul invers si direct de utilizare a unui Sistem Expert

Algoritmi de căutare utilizați ca metode de rezolvare a unor probleme. Agenți planificatori.

În acest capitol se prezintă câteva metode de rezolvare a unor probleme sau situații reale prin utilizarea algoritmilor de căutare. La nivel de corespondență cu inteligența umană, se pare că în creierul omului se desfășoară ceva asemănator cu un algoritm de căutare atunci când se dorește rezolvarea unei probleme, iar primii algoritmi de căutare (apăruți în secolul 19) au avut ca model, și au incercat să reproducă, procesul uman de căutare. Algoritmii de căutare pot oferi o secvență de acțiuni prin care agentul inteligent poate să treacă din starea curentă (de start) într-o stare dorită (țintă) de multe ori prin stări intermediare, și ca urmare, ar rezulta un agent care își poate **PLANIFICA** acțiunile pentru a obține rezultatul dorit.

Reprezentarea informației oferite agentului cu privire la spațiul de căutare se realizează în mod standard sub formă de graf (în sens matematic), dar acest lucru nu exclude alte posibilități dacă acestea se impun. S-a discutat în capitloul 2 despre metodele de reprezentare a unei asemenea structuri de date în memoria calculatorului. Rezultatul unui algoritm de căutare este, în general, o structură tip arbore (de asemenea discutată în capitolul 2) din care se poate extrage o secvență de acțiuni, numită cale, care să ducă la rezultat (țintă).

În general, algoritmii de căutare se pot împărți în două categorii, algoritmi de căutare

- neinformată, situație în care agentul nu are nici o informație din spațiul în care se desfășoară activitatea, știe doar stările în care a fost deja și stările imediate ce urmează să le dezvolte. Aceasta se traduce într-un spațiu de căutare (reprezentat printr-un graf) în care nu există nici un fel de informație
- informată, situație în care agentul inteligent are la dispoziție diverse tipuri de informații pe care le poate extrage din spațiul de căutare (reprezent printr-un graf).

Definirea problemei pentru un agent planificator

Un pas important pentru structurarea procesului de proiectare al unui agent planificator, informat sau neinformat, îl constituie definirea problemei pe care acesta trebuie să o rezolve. Astfel se introduc următoarele concepte folosind ca exemplu didactic un agent tip navigator GPS care caută un traseu între Timișoara și București pe harta României, Fig.1 :

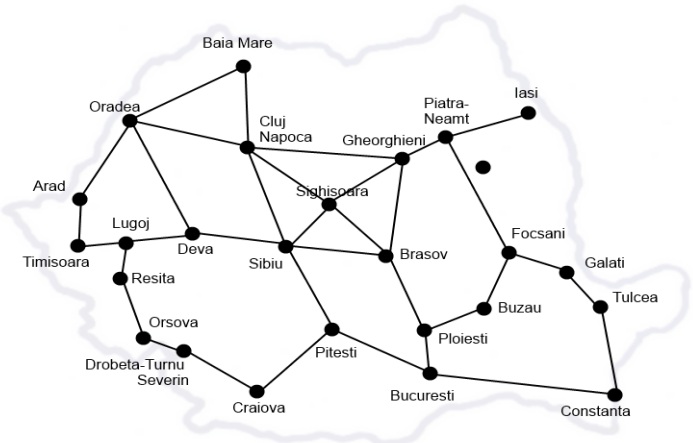

Fig 1. Harta României cu legături între orașe, simplificată, reprezentata sub formă de graf

- **starea inițială**, este starea (configurația) din care pleacă agentul, se notează în continuare cu s. Starea inițială pentru agentul nostru este Timișoara, notată folosind funcția *In*, de exemplu, *In(Timisoara)*
- descriere a mulțimii/setului/tipului de **actiuni** ce pot fi efectuate într-o stare s. De exemplu, instrucțiunea *ACTIUNI(s),* dacă *In(Timisoara)* produce mulțimea de acțiuni aplicabile *{Go(Arad), Go(Lugoj)}*
- **modelul de tranziție**, o descriere a rezultatului efectuării unei acțiuni. Instrucțiunea poate fi următoarea *REZULTAT(s,a)*, unde prin *s* se intelege starea curentă și prin a acțiunea ce se efectuează, iar rezultatul comenzii este o stare nouă s', de exemplu:

  REZULTAT(In(Timisoara), Go(Lugoj))=In(Lugoj)

  Puse laolalta, starea inițială, actiunile și modelul de tranziție definesc **spațiul stărilor** pentru problema noastră, adică, mulțimea tuturor stărilor ce pot fi obținute plecând de la starea inițială și efectuând toate acțiunile posibile. O **cale** în spațiul stărilor este definiă ca secvența de acțiuni ce leagă două stări (inclusiv neadiacente). O cale de a ajunge din Timișoara în București poate fi Timisoara, Lugoj, Deva, Sibiu, Pitești, București
- **starea țintă**, este starea pe care trebuie să o atingă agentul, în cazul nostru, starea țintă este *In(Bucuresti)*.
- **costul deplasării pe o cale**, este o funcție care atribuie o valoare numerică pentru o cale, în exemplul nostru poate fi vorba de numărul de kilometri parcurși. Costul deplasării pe o cale se poate calcula însumând **costul tuturor pașilor/acțiunilor** ce se efectuează atunci când se construiește calea. Se presupune că valoarea costului fiecărui pas este pozitivă. În problema noastră se utilizeaza cuvântul deplasare la nivel general, în loc, ar putea fi energie, litri de combustibil, număr de kilometri etc.

În final, **soluția** unei probleme o reprezintă **calea** ce leagă **starea inițială** de **starea țintă**. Calitatea soluției este măsurată prin costul căii, **soluția optimă** fiind calea cu cost minim. Pentru simplificarea problemei, se introduc câteva abstractizări, în cazul de față fiind vorba despre situația în care agentul se află doar într-un oraș sau altul, el nefiind niciodată pe drum.

Cu toate că s-a utilizat drept exemplu căutarea unei căi de conectare a unui oraș cu altul, algoritmii de căutare se folosesc într-un spectru mult mai larg, mai ales atunci când este necesară luarea unei decizii. Câteva exemple de probleme: Vacuum World [RN], 8 Puzzle, 8 Dame, Gasirea unui traseu într-o hartă, Turul orașelor, Problema agentului de vânzări (Traveling Salesperson Problem – TSP), VLSI Layout, Navigație a unui Robot mobil, Rezolvarea cubului lui Rubik.

Algoritmi de căutare a unei soluții. Elemente introductive

Algoritmii de căutare abordează spațiul de căutare, în cazul nostru harta României reprezentată ca un graf, plecând de la starea inițială = nodul inițial = nodul rădăcină, și produc o structură arborescentă Fig.2 din care se poate alege o secvență de acțiuni (numită și cale) care conduc la starea finală.

Primul principiu esențial în dezvoltarea unui arbore de căutare este că, într-o cale dezvoltată, să nu apară aceeași stare (același oraș, în cazul nostru) de două ori. Motivul pentru care se evită această situațe este că apar cicluri (bucle) infinite. Un exemplu ar fi umătorul, se presupune că se pleacă din

Timișoara și se dezvoltă următoarea cale Timisoara-Lugoj-Deva-Oradea-Arad-Timisoara, se observă că se produce o structură tip ciclu infinit care ar duce la blocarea algoritmului într-o buclă infinită, iar algoritmul nu ar găsi niciodată calea către București.

Al doilea prinicpiu este abordarea nodurilor copil "înainte", exemplificată și în Fig.2 unde căutarea pleacă din Timișoara, copii nodului Timișoara fiind Arad și Lugoj. În continuare se alege Lugoj, copii nodului Lugoj fiind Deva, Reșita și Timișoara (dacă s-ar analiza posibilitățile de deplasare din Lugoj, acesta fiind legat cu Deva, Reșita și Timișoara), dar pentru că se vine din Timișoara, acest nod (Timișoara) nu se adaugă la arbore pentru a nu crea buclă infinită.

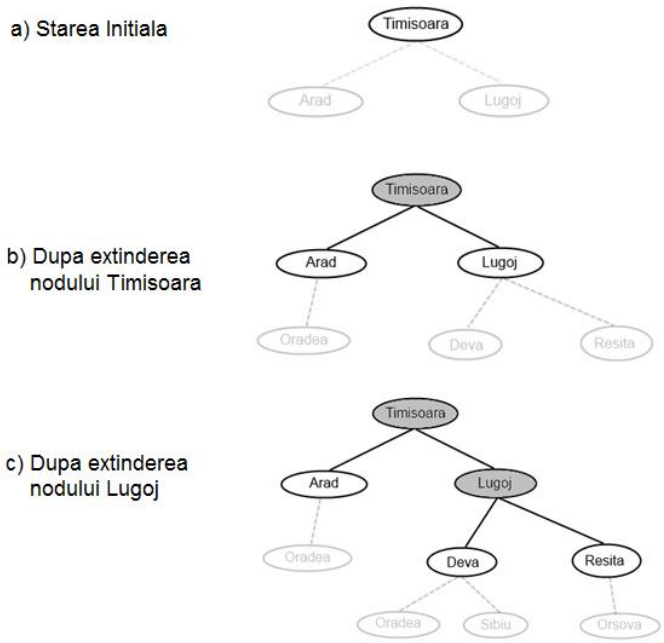

Fig.2 Exemplu de dezvoltare a arborelui de căutare

Atunci când se analizează performanțele unui algoritm de căutare se au în vedere următoarele aspecte:

- completitudine: Algoritmul e garantat să găsească o soluție atunci când există una;
- optimalitate: Algoritmul găsește soluția optimă (calea de cost minim);
- complexitate de timp: Timpul necesar găsirii unei soluții
- complexitate în spațiu: Memoria necesară în procesul de căutare

În continuare se vor prezenta cele două categorii mari de algoritmi de căutare, căutarea neinformată și căutarea informată.

Algoritmi de căutare neinformată

Aceste tipuri de algoritmi țin cont doar de forma spațiului unde se desfășoară căutarea, fără să țină cont de informații suplimentare ce pot apărea în spațiul de căutare. Dacă se folosește ca exemplu harta României, un algoritm neinformat tine cont doar de legaturile dintre orașe, reprezentate printr-o structură tip graf, dar nu ține cont de informații precum distanta dintre orașe, cost sau timp de deplasare.

Pentru a ilustra diferitele concepte de funcționare ale algoritmilor se utilizează graful din Fig. 3 [PHW online]. Acesta este un spațiu restrâns de căutare pentru a reprezenta principiile fundamentale ce stau la baza fiecărui algoritm, algoritmul în sine putând fi aplicabil oricărui spațiu, oricât de mare. Întodeauna se pleacă de la nodul de start S și se dorește găsirea unei căi până la nodul țintă G, dezvoltându-se un arbore de căutare din care se alege calea către țintă.

Ca o regulă, pentru a menține o structură ordonată a arborelui de căutare, nodurile se ordonează alfabetic pe fiecare nivel, dacă nu se specifică altfel.

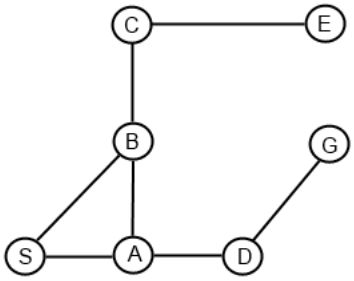

Fig. 3 Un spațiu de căutare reprezentat printr-un graf

1. Căutarea exhaustivă

Acest tip de căutare găsește toate căile posibile de la nodul de start S, fără a se focaliza pe găsirea unei anumite (distanta cea mai scurtă, lungă, număr minim de salturi etc.) căi de la start la țintă, respectând cele două condiții, să nu apară două aceleași noduri pe o ramură, respectiv căutarea se face înainte.

Arborele de căutare asociat acestui tip de căutare este prezentat în Fig. 4

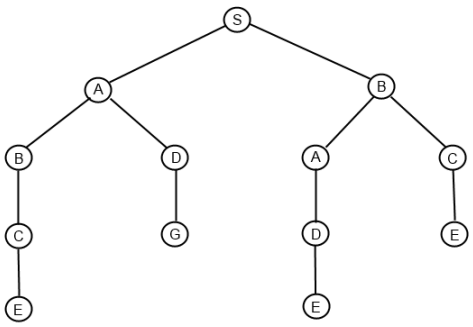

Fig.4 Arbore obținut prin căutare exhaustivă

Se observă că arborele de căutare găsește toate căile posibile plecând din S.

2. Căutarea în adâncime

Căutarea în adâncime (in engleză: Depth First Search, DFS) explorează, cât de mult se poate, o ramură din arborele de căutare. În cazul în care nu se mai poate înainta, se face un pas înapoi (backtrack) și se încearcă o altă cale. Pentru exemplificare se aplică algoritmul de căutare în adâncime pe graful din Fig.3, rezultatul fiind prezentat în Fig.5.

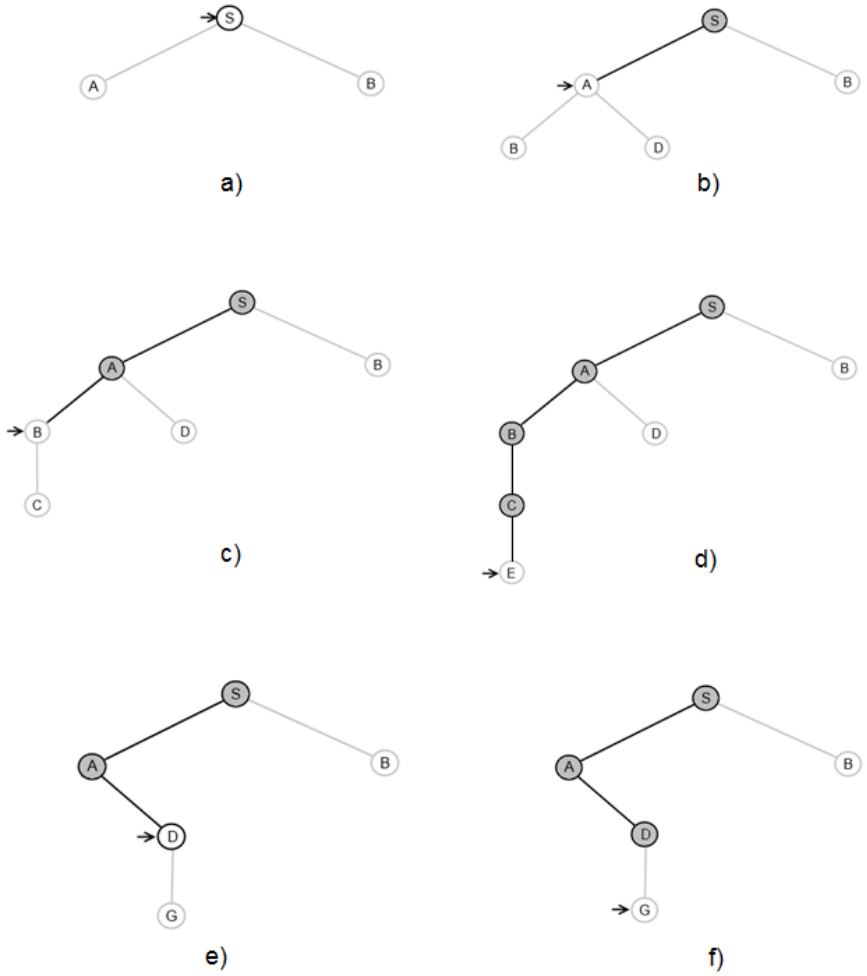

Fig. 5 Dezvoltarea arborelui de căutare în adâncime (→ reprezintă nodul curent analizat, fundal gri – nod trecut în dezvoltare, linie gri – noduri copil neexplorate)

Algoritmul funcționează în felul următor:

- se pleacă din nodul de S (de start), se identifică copii acestuia (Fig.5a) și se ordonează alfabetic.
- se alege ramura din stânga, în cazul acesta, conține nodul A (Fig.5b)
- se alege ramura din stânga, în cazul acesta, conține nodul B (Fig.5c)
- se alege ramura din stânga (de fapt există o singură ramură), în cazul acesta, conține nodul C apoi nodul E (Fig.5d)
- se dorește extinderea lui E, dar acesta nu are copii, ca urmare se face un pas înapoi până la primul nod neexplorat, anume D (Fig.5e). Acest pas se numește pas de BACKTRACKING

- se alege ramura din stânga (de fapt există o singură ramură), în cazul acesta, conține nodul G (Fig.5f)
- algoritmul se oprește pentru că s-a găsit nodul țintă

O particularitate a acestui algoritm o constituie pasul de întoarcere înapoi sau backtracking. Acest pas apare de fiecare dată când se dezvoltă la maxim o ramură, fără a se găsi soluția pe acea ramură, practic prin acest pas se continuă explorarea arborelui dar pe altă ramură, și algoritmul nu se blochează.

Pseudocodul pentru DFS este următorul:

```
01. Inițializare listă tip coadă
02. Dacă ținta este în prima cale din coadă, SARI la 05
03. Dacă se poate extinde prima cale din coadă, extinde și adaugă la coadă, la început,
        altfel șterge calea curentă
04. Sari la 02.
05. Afișează prima cale și termina programul
```

Urmărirea pas-cu-pas a pseudocodului produce urmatorul rezultat:

- inițializare coadă, coada va fi doar (S) – doar nodul de start
- extinde prima cale din coada–> coada nouă va fi (SA)(SB)
- extinde prima cale din coada –> coada nouă va fi (SAB)(SAD)(SB)
- extinde prima cale din coada –> coada nouă va fi (SABC)(SAD)(SB)
- extinde prima cale din coada –> coada nouă va fi (SABCE)(SAD)(SB)
- prima cale nu se mai poate extinde și nu conține nici ținta, deci se va șterge, coada va fi (SAD)(SB)
- extinde prima cale din coada – coada noua va fi (SADG)(SB)
- afișează prima cale – SADG – calea găsită prin DFS

În cazul căutării în adâncime, coada în care se adaugă nodurile este o coadă de tip LIFO – Last In First Out. Algoritmul nu produce rezultate optimale, nu este complet decât cu unele ajustări și pentru anumite spații de căutare, iar complexitatea în timp este relativ mare. Avantajul acestui algoritm este ca are complexitate în spațiu mică deoarece este nevoie să se memoreze doar calea curentă și nodurile de ramificație.

3. Căutarea pe nivel

Căutarea pe nivel (în engleză Breadth First Search – BFS) dezvoltă nodurile copil completând nivel după nivel în arborele de căutare. Practic, în spațiul de căutare, se pleacă din nodul start unde se explorează vecinii acestuia, formând primul nivel, apoi vecinii vecinilor, formând cel de-al doilea nivel și așa mai departe. Algoritmul continuă până se descoperă ținta sau se epuizează spațiul de căutare. O ilustrare a acestei abordări aplicată pe graful din Fig. 3 este prezentată în Fig.6.

Pseudocodul pentru DFS este următorul:

01. Inițializare listă tip coadă
02. Dacă ținta este în prima cale din coadă, sari la 05
03. Dacă se poate extinde prima cale din coadă, extinde și adaugă la coadă la sfârșit,
    altfel șterge calea curentă
04. Sari la 02.
05. Afișează prima cale

Urmărirea pas-cu-pas a pseudocodului produce urmatorul rezultat:

- inițializare coadă, coada va fi doar (S) – doar nodul de start
- extinde prima cale din coadă și adaugă la sfârșit –> coada nouă va fi (SA)(SB)
- extinde prima cale din coadă și adaugă la sfârșit –> coada nouă va fi (SB)(SAB)(SAD)
- extinde prima cale din coadă și adaugă la sfârșit –> coada nouă va fi (SAB)(SAD)(SBA)(SBC)
- extinde prima cale din coadă și adaugă la sfârșit –> coada nouă va fi (SAD)(SBA)(SBC)(SABC)
- extinde prima cale din coadă și adaugă la sfârșit –> coada nouă va fi (SBA)(SBC)(SABC)(SADG)
- afișează ultima cale din coadă – SADG – calea găsită prin BFS

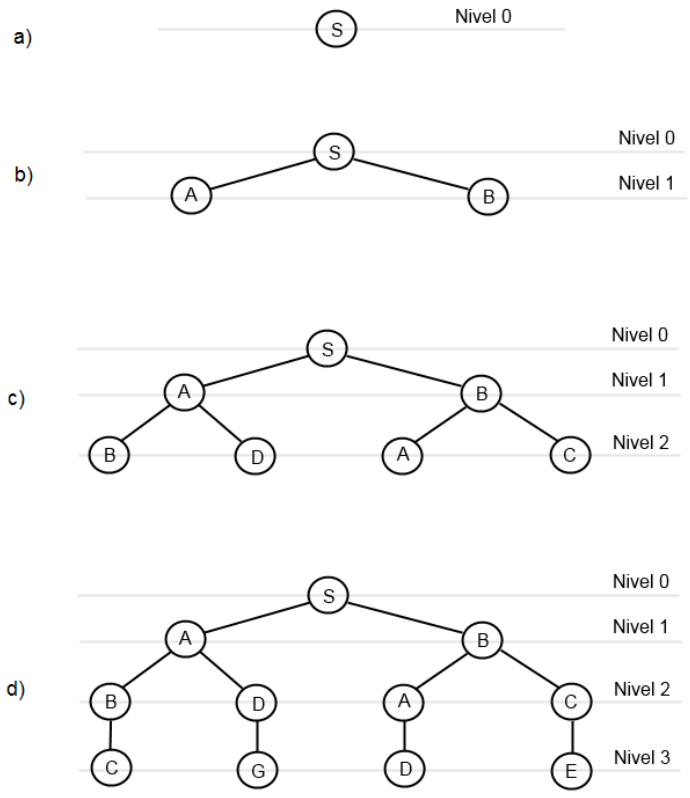

Fig. 6. Dezvoltarea arborelui de căutare pe nivel

În cazul căutării pe nivel, putem spune că algoritmul este complet dar nu generează în mod necesar o soluție optimă.

Adancime	Noduri	Timp	Memorie
2	110	0.11 milisecunde	107 KB
4	11110	11 milisecunde	10.6 MB

6	10^6	1.1 secunde	1 GB
8	10^8	2 minute	103 GB
10	10^{10}	3 ore	10 TB
12	10^{12}	13 zile	1 petabyte
14	10^{14}	3.5 ani	99 petabyte
16	10^{16}	350 ani	10 exabyte

Tabel 1. Necesar de memorie și timp de procesare pentru căutarea pe nivel în funcție de adâncimea arborelui de căutare

Legat de complexitatea în timp și spațiu a algoritmului, se poate folosi un exemplu în care există un arbore uniform cu factor de ramificare constant b=10. În această situație, nodul rădăcină ar avea 10 noduri copil corespunzători nivelului 1, la nivelul 2, fiecare nod copil ar avea la rândul său 10 noduri copil situația ar corespunde la 100 de noduri terminale. Formula de calcul pentru numărul de noduri terminale se poate generaliza obținând expresia

$$Nr\_noduri\_term = b^d \tag{1}$$

unde b= factorul de ramificare și d=adâncimea arborelui. Pentru ilustrare se construiește Tabelul 1, datele fiind calculate pentru b=10, 1 milion de noduri calculate/secundă, 1000 byte/nod memorie

Optimizarea algoritmilor mai sus prezentați se poate face în două feluri:

1. introducerea unei **liste cu elemetele extinse** pentru ca acestea sa nu fie parcurse a doua oară. În Fig. 7 se observă arborele de căutare produs prin BFS dar și dublurile de noduri, care induc un consum suplimentar de resurse computaționale.

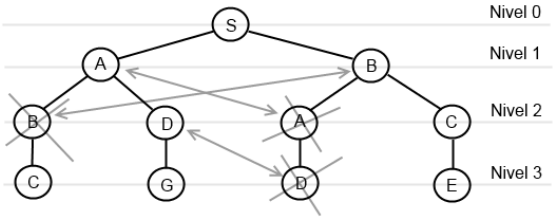

Fig.7. Căutare pe nivel și ilustrarea elementelor dublate

2. introducerea unui indicator care ne permite să **direcționăm căutarea către țintă**. Acest concept transformă algoritmul într-unul informat (tip Hill Climbing sau Beam) și va fi prezentat în capitolul destinat algoritmilor informați.

Algoritmi de căutare informată

Acest tip de algoritmi folosesc informații din spațiul de lucru pentru găsirea unei soluții. Dacă până acum, pe graful de situații nu a existat nici un fel de informație, de data aceasta, pentru a putea folosi un astfel de algoritm este nevoie de ornamentarea acestuia cu informații – Fig.8.

Tipul de informație ce poat să apară într-un graf este de două feluri:

1. **costul** de a trece dintr-un nod din graf în altul. Aceste informații se pot materializa într-o hartă prin distanța măsurată între două orașe. De exemplu, costul de a trece din S în A este 3, acesta este marcat pe arcul care unește cele două noduri. În mod similar se poate citi costul de deplasare între oricare alte două noduri. Pentru formularea matematică se introduce funcția de cost g().

2. **Informații (distanțe) euristice** sunt informații care permit direcționarea căutării către țintă. Aceste informații nu sunt extrem de precise, oferă doar o idee generală legată de poziția țintei. În exemplul cu harta României, acest tip de informații poate fi distanța aeriană între un oraș oarecare și orașul țintă. Distanțele euristice pot fi interpretate de asemena sub forma costului estimat de deplasare dintr-un nod în altul.

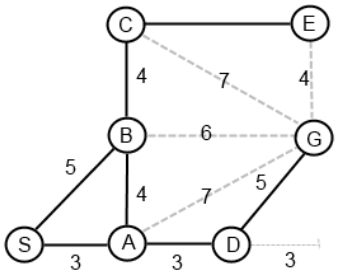

Fig.8 Spațiu de căutare cu informații [PHW online]

În Fig.8 informațiile euristice sunt reprezentate prin linie punctată. Aceste linii punctate nu conectează cele două noduri ci sunt folosite doar ca reprezentare a distanței euristice. De exemplu, între nodurile A și G nu există o cale directă dar distanța euristică (în linie aeriana) este 7. O altă metodă de reprezentare a distanței euristice între un nod și țintă de eemplu poate să fie o valoare trecută în interiorul fiecărui nod, metodă neilustrată aici.

Pentru o formulare concisă matematic se introduce o funcție h(n) numită funcție euristică, unde n reprezintă nodul față de care se dorește determinarea. De exemplu, fiind în nodul A, h(G) = 7.

Modul în care se utilizează această informație este, că se dorește alegerea nodurilor care sunt mai aproape de țintă și în felul acesta, se poate accelera procesul de căutare. Câteodată ideea de funcție euristică poate să și încurce procesul de căutare, un exemplu în acest sens este următorul, să zicem ca vă aflați în C și doriți să ajungeti în G, copii lui C sunt B și E, cu h(B)=6 respectiv h(E)=4, conform algoritmului se alege E pentru că este mai aproape de țintă, dar E este o fundătură pentru că nu este legat direct de G.

Un exemplu de funcție euristică folosit în graful din Fig.8 este bazat pe determinarea geometrică a distanței dintre cele două noduri, de exemplu:

- distanța euristică între B și G = lungimea lui AD la care se adaugă proiecția lui DG axa Ox, adică, BG=3+3=6
- distanța euristică între A și G = aplicăm Teorema lui Pitagora în ΔABG și obținem AG=7.07

Un al doilea exemplu este harta României (Fig.1), unde dacă se pleacă din Sighișoara către București, mecanismul euristic împiedică abordarea orașelor care ne îndepărtează de țintă.

În funcție de situație se pot aplica alte funcții de determinare a distanței euristice, specifice problemei respective. În cazul nostru, fiind vorba despre o hartă, se poate aplica geometria euclidiană.

Pentru două noduri între care există cale, se poate lua ca funcție euristică distanța dintre ele, de exemplu, între A și D, distanța euristică este 3 egală cu distanța dintre cele două noduri.

1. Algoritmul Hill Climbing

Acest algoritm folosește informația euristică pentru accelerarea căutării unei căi către țintă. Modul de operare al acestuia este asemănător cu cel al căutării în adâncime, doar că, de data aceasta se extinde prima dată nodul cel mai apropiat de țintă. Determinarea apropierii fiecărui nod de țintă se face folosind funcția euristică h(n), din acest motiv, algoritmul Hill Climbing este un tip de algoritm "lacom" (engleză: greedy) pentru că tinde să aleagă tot timpul nodul cel mai aproape de țintă.

Pseudocodul pentru Hill Climbing este următorul:

```
01. Inițializare listă tip coadă
02. Dacă ținta este în prima cale din coadă, sari la 05
03. Dacă se poate extinde prima cale din coadă, extinde și ordonează după h(n),
        altfel șterge calea curentă
04. Sari la 02.
05. Afișează prima cale
```

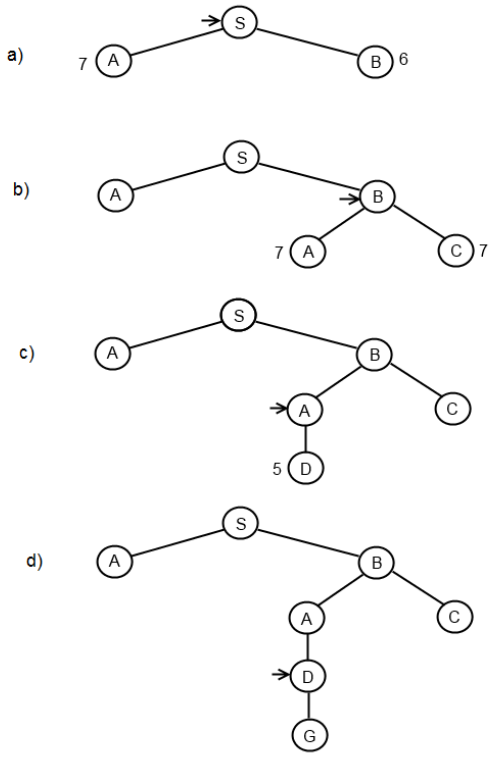

Fig. 9 Arbore de căutare obținut prin Hill Climbing

Până acum s-au utilizat algoritmi de căutare doar în spații cu stări discrete, reprezentate prin grafuri, Hill Climbing poate fi utilizat și în spații continue reprezentate prin diverse metode: funcții de 1, 2 ..n variabile, diagrame de contur etc. Problema se pune în felul următor: dat fiind un spațiu continuu, definit de una sau mai multe variabile, să se găsească valoarea maximă a funcției care produce acest spațiu. Un exemplu de asemenea situație este prezentat în Fig. 10 unde se folosește o funcție f() de o singură variabilă x, care produce spațiul/graficul din figură. Se dorește să se găsească maximul global al funcției folosind algoritmul de căutare Hill Climbing.

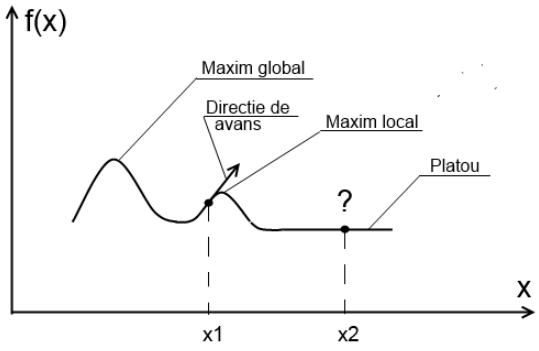

Fig. 10 Hill Climbing aplicat pe o funcție cu un parametru

Algoritmul funcționează în felul următor:

01. se inițializează o variabilă cu o valoare oarecare, să zicem x1

02. se calculează f(x1), f(x1+p), f(x1-p), unde p>0, este pasul cu care se face căutarea. Se utilizează p pentru a analiza care este direcția de creștere a funcției. Se observă că f(x1+p) > f(x1), ca urmare va trebui să efectuăm aceeași procedură pentru x1+p. Acest lucru se poate realiza programatic prin repetarea pasului 02 dacă x1=x1+p. Pasul 02 se repetă până când se ajunge în maxim, programatic, acest lucru se exprimă prin faptul că f(x1)>f(x1+p) și f(x1)>f(x1-p), adică, valoarea curentă al ui x1 produce valoarea cea mai mare pentru funcția f().

Problemele pe care le manisfestă Hill Climbing în spații continue sunt trei la număr: potențial de a gasi maxime locale în detrimentul maximelor globale, platouri și creste. În Fig. 10 se pot observa doar două dintre aceste probleme

- **maximul local** poate apare dacă se pornește căutarea din x1, se observă că algoritmul va avea tendința să se deplaseze către maximul local în detrimentul maximului global. O metodă de rezolvare a acestei probleme este utilizarea mai multor iterații de Hill Climbing cu valori dispersate pentru x1, acest procedeu fiind denumit Călire Simulată (engleză: Simulated annealing)
- **platourile** sunt zone din funcția de optimizat unde orice pas de căutare produce același rezultat, ca urmare este dificil de stabilit o direcție de avans către un optim (din moment ce totul este constant și noi dorim să găsim maximul)

Pentru a ilustra cea de-a treia problemă a lui Hill Climbing, **crestele**, este nevoie să folosim o funcție cu 2 parametri f(x,y). Pentru a fi mai intuitivi, să zicem că ne aflăm pe un deal, poziția noastră este dată de coordonatele (x,y), înățimea dealului fiind o funcție de (x,y). Putem concepe un algoritm care să ne ducă în vârful dealului prin pași pe direcția x sau y. Dacă se analizează Fig.11 (c), se observă că orice pas, fie pe direcția x, fie pe direcția y duce către valori mai mici ale înălțimii, ca urmare, algoritmul va considera că poziția curentă este cea de maxim, lucru care nu este adevărat, așa cum se observă și în figură.

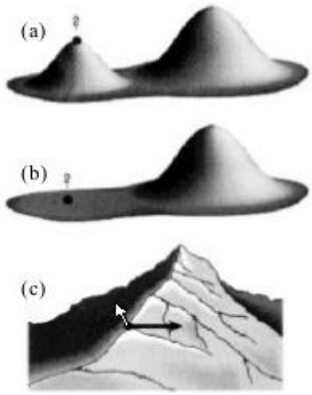

Fig. 11 Spații de căutare produse prin funcții de două variabile
[https://github.com/reSHARMA/Generic-Hill-Climber 05.08.2018]

În Fig.11 (a) și (b) se observă de asemenea problema maximului local și a platoului pentru spații descrise de funcții cu două variabile, dar algoritmul de căutare Hill Climbing funcționează și în spații multidimensionale, produse de funcții cu mai multe variabile.

2. Algoritmul de căutare Fascicul

La fel cum Hill Climbing este o versiune a căutării în adâncime care folosește informație euristică pentru a dirija căutarea către țintă, căutarea Fascicul (engleză: Beam) este o variantă a căutării pe nivel care utilizează informația euristică spre a dirija căutarea. În căutarea Fascicul se definește "lățimea fascicolului" w, ca fiind numărul de noduri ce vor fi dezvoltate în nivelul următor, nodurile alese spre a fi dezvoltate fiind cele w mai aproape de țintă.

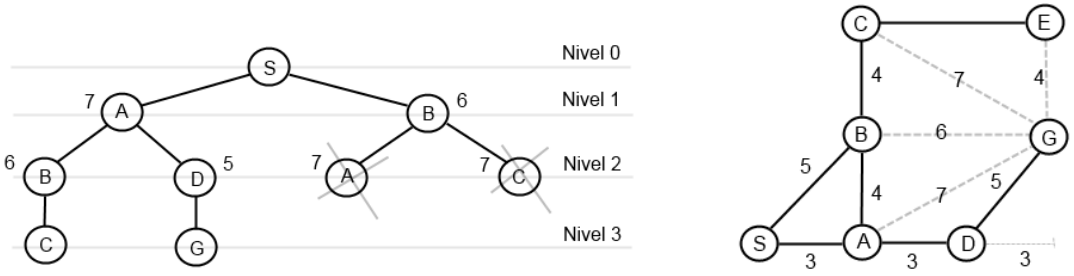

Fig. 12. Dezvoltarea arborelui de căutare pentru algoritmul de căutare Fascicul

În Fig. 12 se observă dezvoltarea arborelui de căutare folosind algoritmul Fascicul. Pentru acest caz s-a folisit o lățime a fascicolului w=2. Algoritmul funcționează în felul următor, se dezvoltă nivelul 0, ca urmare, pe nivelul 1 apar 2 noduri, după care se determina care sunt cele w noduri (în cazul nostru w=2) candidat pentru iterația următoare. Algoritmul se reia pentru nivelul 1, se dezvoltă nodurile de pe nivelul 1, rezultă 4 noduri pe nivelul 2, din care se aleg cele w noduri cele mai apropiate de țintă (cazul nostru B și D). Algoritmul se reia pentru nivelul 2, se dezvoltă nodurile, și se observă că pe nivelul 3 apare ținta.

Pseudocodul pentru căutarea Fascicul este următorul:

```
01. Inițializare listă tip coadă
02. Dacă ținta este în prima cale din coadă, sari la 05
03. Dacă se poate extinde prima cale din coadă,
        extinde, ordonează după h(n), alege cele mai bune w variante și scrie coada cu ele
        altfel șterge calea curentă
04. Sari la 02.
05. Afișează prima cale
```

Algoritmi de căutare a unei soluții optime. Branch and Bound. A*.

Aceste tipuri de algoritmi intră și ei în categoria algoritmilor de căutare informată pentru că utilizează informații din spațiul de căutare, dar sunt categorisiți într-o categorie proprie pentru că obiectivul lor este de a găsi calea optimă (minimă/maximă) către țintă. În acest sens se dorește minimizarea funcției (1)

$$f(x)=g(x)+h(x) \tag{1}$$

unde $f(x)$ - costul de a ajunge la țintă pe o rută ce include nodul x

$g(x)$ - costul acumulat până în nodul x

$h(x)$ - costul estimat euristic de a ajunge din nodul curent x până la țintă

1. Algoritmul Branch and Bound (B&B)

Acest algoritm optimizează funcția $f(x)$ calculând doar costul acumulat $g(x)$ de a ajunge în nodul x și considerând $h(x)$ ca fiind zero. Pentru o mai bună înțelegere se perzintă un exemplu bazat pe graful din Fig.13, unde se ignoră $h(x)$. Ca de obicei, se pleacă din S, algoritmul extinzând tot timpul cea mai scurtă cale descoperită analizând arborele în întregime, rezultă dezvoltarea arborelui de căutare cu secvența prezentată în Fig. 14.

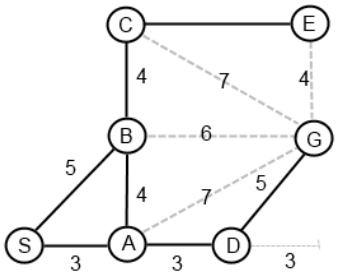

Fig.13 Spațiu de căutare cu informații [PHW online]

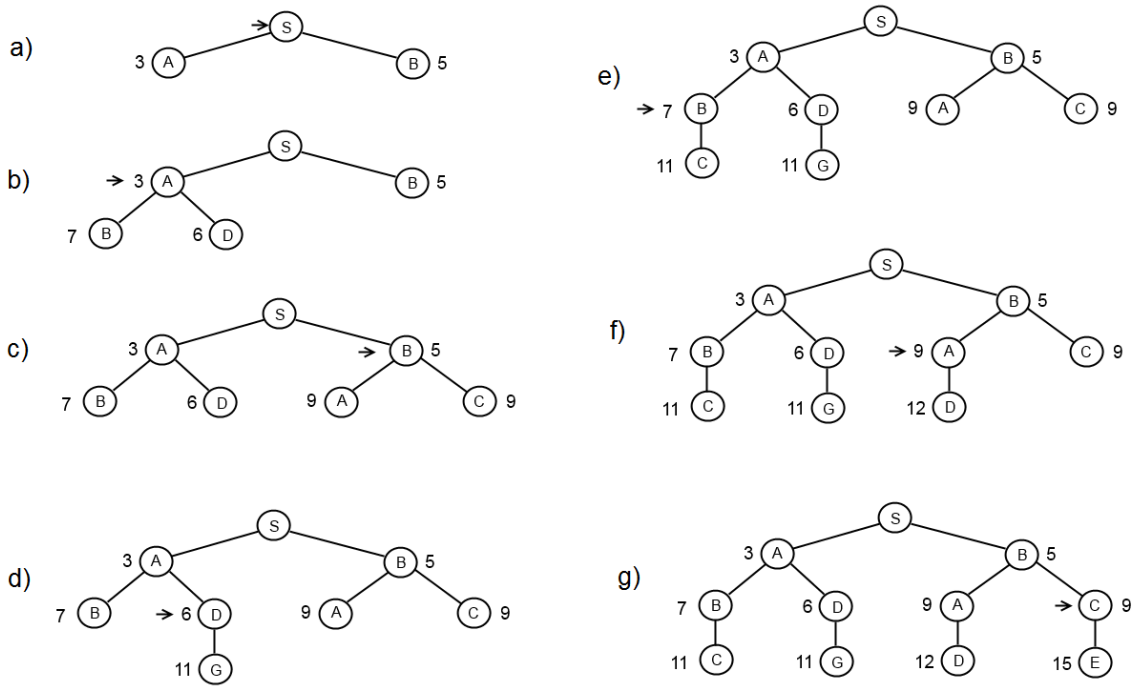

Fig. 14 Etapele parcurse în dezvoltarea arborelui de căutare folosind Branch and Bound

De notat este faptul că, în dreptul fiecărui nod din arborele de căutare prezentat în Fig.14 se notează distanța parcursă de agent urmărind calea respectivă, de exemplu distanța de la S la B pe calea SAB este 7. Modul de rulare al lui B&B este următorul:

- se pleacă din nodul S Fig 14a, se analizează arborele și se extinde cea mai scurtă rută: SA=3, rezultă Fig. 14b
- se analizează arborele în Fig. 14b și se extinde cea mai scurtă rută: SB=5, rezultă Fig. 14c
- se analizează arborele în Fig. 14c și se extinde cea mai scurtă rută: SAD=5, rezultă Fig. 14d
- se analizează arborele în Fig. 14d și se extinde cea mai scurtă rută: SAB=7, rezultă Fig. 14e, o observație la acest pas este că, algoritmul nu se oprește pentru că a găsit ținta, pentru că există alte căi neexplorate cu potențial de a ne aduce mai eficient la țintă, scopul algoritmului fiind găsirea soluției optime
- se analizează arborele în Fig. 14e și se extinde cea mai scurtă rută: SBA=9, rezultă Fig. 14f, o observație la a acest pas este că lungimea căii SBA=SBC=9, alegerea lui SBA se face din motiv alfabetic, dar în funcție de situație, se poate alege orice alt criteriu de departajare în cazul căilor de aceeași lungime
- se analizează arborele în Fig. 14f și se extinde cea mai scurtă rută: SBC=9, rezultă Fig. 14g
- se analizează arborele în Fig. 14g și se observă că SADG=11 este calea cea mai scurtă care duce la țintă, pentru toate celelalte am demonstrat ca sunt mai lungi.

Pseudocodul acestui algoritm

```
01 Inițializare coadă cu primul nod, S
02 Testează prima cale dacă conține nodul țintă, dacă da, afișează prima cale și oprește-te
03 Extinde prima cale și ordonează căile obținute în funcție de lungimea lor
04 Sari la 02
```

Dacă algoritmul B&B se completează cu listă a elementelor vizitate spre a nu fi extinse a doua oară, algoritmul rezultat se numește Dijkstra după numele savantului care l-a inventat în 1956.

2. Algoritmul A*

Algoritmul A* (citit A steluță, sau în engleză A star) este o variantă optimizată a lui Branch and Bound, obiectivul acestuia fiind de a determina calea optimă (minimă sau maximă) dintre două stări, similar lui B&B. Îmbunătățirea față de B&B a lui A* constă în faptul că, acesta, în mod suplimentar, implementează următoarele două idei:

1. Lista cu elemente extinse - ține cont de nodurile care au fost deja extinse prin crearea unei liste cu elemente extinse. Utilitatea acesteia fiind exemplificată prin cazul prezentat în Fig 14g, unde B&B găsește calea SA către nodul A, calea având cost 3, pentru ca, ulterior, să găsească calea SBA tot către nodul A dar de cost 9 și să o extindă în continuare, generând în mod evident căi mai lungi.
2. Distanța euristică $h(x)$ pentru a orienta căutarea către țintă

Similar lui B&B, scopul lui A* este minimizarea funcției $f(x)=g(x)+h(x)$, de data aceasta, termenul $h(x)$ fiind nenul.

Pentru a exemplifica modul de funcționare a lui A* se utilizează graful din Fig13, pașii rezultați în construirea arborelui de căutare fiind prezentați în Fig. 15.

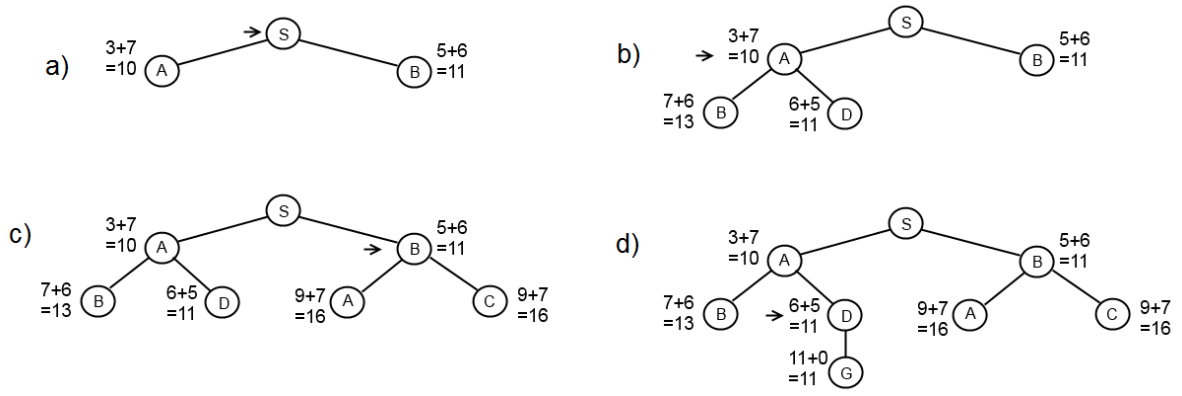

Fig. 15 Etapele parcurse în dezvoltarea arborelui de căutare folosind A*

Similar lui B&B, și în cazul lui A* apar câteva notații pe arborele de căutare în dreptul fiecărui nod, acestea fiind distanța cumulată parcursă până la nodul respectiv $g(x)$ la care se adaugă distanța euristică până la nodul țintă $h(x)$, în final, cele două valori se adună și se obține valoarea pentru funcția $f(x)$. Nodurile se dezvoltă în funcție de valoarea pe care o are $f(x)$ în fiecare nod, rezultând astfel arborele de căutare din Fig. 15. Modul de rulate al lui A* este următorul:

se pleacă din S, se analizează arborele

- se pleacă din nodul S Fig 15a, se analizează arborele și se extinde cea mai scurtă rută: SA=10, rezultă Fig. 15b, Lista cu elemente extinse: SA
- se analizează arborele în Fig. 14b și se extinde cea mai scurtă rută: SB=11, rezultă Fig. 15c, o observație la acest pas este că lungimea căii SB=SAD=11, alegerea lui SB se face din motiv alfabetic (B precede lui D), dar în funcție de situație, se poate alege orice alt criteriu de departajare în cazul căilor de aceeași lungime, Lista cu elemente extinse: SAB
- se analizează arborele în Fig. 14c și se extinde cea mai scurtă rută: SAD=11, rezultă Fig. 14d, , Lista cu elemente extinse: SABD
- se analizează arborele în Fig. 14d și se observă că SADG=11 este calea cea mai scurtă care duce la țintă, pentru toate celelalte am demonstrat ca sunt mai lungi

Pseudocodul lui A*

01 Inițializare coadă cu primul nod, S 02 Testează prima cale dacă conține nodul țintă, dacă da, afișează prima cale și oprește-te 03 Extinde cea mai scurtă cale în funcție de $f(x)=g(x)+h(x)$ dacă aceasta nu se termină într-un nod din lista cu elemente extinse și adaugă nodul la lista cu elemente extinse 04 Sari la 02

Distanța euristică admisibilă, distanță euristică consistentă și situații când A* nu funcționează

S-au văzut și din exemplul scurt prezentat, avanatajele lui A*. Cu toate acestea, A* poate să nu funcționeze corect dacă funcția euristică aleasă este necorespunzătoare, în acest fel distingându-se două categorii de funcții:

Euristică admisibilă, valorile produse de mecanismul de estimare trebuie să fie mai mici sau cel mult egale cu distanța pe cale până la țintă. De exemplu, în Fig.13, distanța euristică de la A la G este 7, iar distanța pe calea ADG este 8 satisfăcându-se astfel condiția de admisibilitate pentru această valoare euristică. În același timp trebuie reamintit faptul că Fig.13 reprezintă o hartă, reprezentarea este făcută la scară iar funcția euristică aleasă este distanța aeriană între nodul curent și nodul țintă rezultând astfel un spațiu euclidian (în acesta se poate aplica geometria euclidiană). Matematic, această legătură este reprezentată prin relația (2)

$$H(x,G) \le D(x,G) \tag{2}$$

unde H() este o funcție euristică ce calculează distanța dintre nodul x și ținta G, iar

D() este o funcție ce calculează calea dintre nodul x și ținta G

De multe ori însă, algoritmul de căutare nu rulează pe un spațiu euclidian, în aceste cazuri, algoritmul A* produce rezultate incorecte dacă se utilizează o euristică admisibilă fiind nevoie de o funcție euristică mai restrictivă. Un exemplu în acest sens este graful din Fig. 16, se observă că acesta nu este reprezentat la scară, și ca triunghiul SAB nu se poate forma dacă se iau în considerare dimensiunile geometrice ale laturilor, ca urmare acest spatiu nu este euclidian. În Fig.16 se notează pe arce distanța dintre noduri, iar în dreptul nodurilor distanța euristică admisibilă a acestora până la țintă.

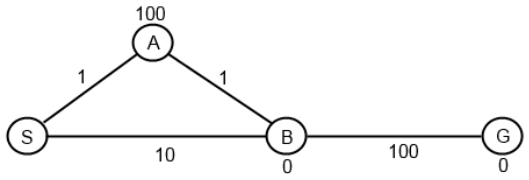

Fig.16 Spațiu ne-euclidian în care A* produce un rezultat greșit

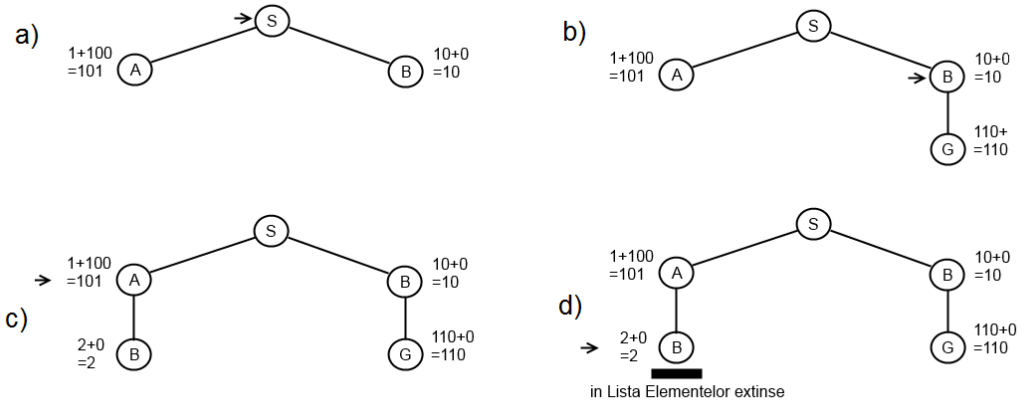

Fig. 17 Dezvoltarea arborelui de căutare pentru A* în spațiu ne-euclidean folosind euristică admisibilă

De notat este că se alege valoarea 0 drept valoare euristică admisibilă a unui nod până la țintă, lucru care nu contravine definiției unei valori euristice admisibile, aceasta fiind este orice valoare mai mică sau cel mult egală cu distanța pe cale, zero fiind întodeauna mai mic decât distanța pe cale. Acest

exemplu s-a ales în mod intenționat pentru a scoate în evidență cauzele pentru care A* eșuează în gasirea soluției optime.

Pentru graful din Fig.16, arborele de căutare A* aferent se dezvoltă în felul următor Fig.17:

- se extinde S, Fig.17a, S se adaugă la lista cu elemente extinse (lista:{S}), se analizează cea mai scurtă ramură din arbore, se alege calea SB=10 fiind cea mai scurtă cale
- se extinde B, Fig.17b, B se adaugă la lista cu elemente extinse (lista:{S,B}), se analizează cea mai scurtă ramură din arbore, se alege calea SA=101 fiind cea mai scurtă cale
- se extinde A, Fig.17c, A se adaugă la lista cu elemente extinse (lista: {S,B,A}), se analizează cea mai scurtă ramură din arbore, se alege calea SAB=2 fiind cea mai scurtă cale
- se extinde B, Fig.17d, B se adaugă la lista cu elemente extinse (lista: {S,B,A}), pas în care algoritmul realizează că B este deja în listă și nu poate fi adăugat, ca urmare, calea SAB se elimină, și rămâne calea SBG ca fiind cea mai scurtă cale.

Concluzia algoritmului A* rulat într-un spațiu ne-euclidean folosind distanță euristică admisibilă este că SBG=110 este calea cea mai scurtă, în realitate, calea cea mai scurtă fiind SABG=102, iar demostrația a fost făcută simulând rularea lui A* pe graf.

Euristică consistentă. Problema distanței euristice admisibile se poate soluționa utilizând o distanță euristică consistentă, aceasta respectând relația (3) în loc de relația (2).

$$\left|H(x,G)-H(y,G)\right| \le D(x,y) \tag{3}$$

Interpretarea relației (3) se face în felul următor, valoarea absoluta a diferenței între distanța euristică a unui nod x la țintă și a unui nod y la țintă trebuie să fie mai mică sau egală cu distanța efectivă între cele două noduri. În cazul grafului din Fig. 16 avem următoarea situație:

$H(A,G)=100$; $H(B,G)=0$; $D(A,B)=1$, înlocuind aceste valori în relația (3) se obține $100 \le 1$, deci distanța euristică folosită nu este consistentă.

În final putem concluziona că A* funcționează optimal pentru situațiile cu spații euclidiene și distanță euristică admisibilă, dar pentru spații ne-euclidiene este necesară utilizarea unei distanțe euristice consistente.

Căutare în condiții adversariale. Jocuri. MiniMax. AlfaBeta.

Căutarea în condiții adversariale apare ori de câte ori agentul inteligent se află în competiție cu alți agenți pentru a își atinge obiectivul. Din cauza prezenței mai multor agenți, mediul în care agentul își desfășoară activitatea este un mediu multiagent, aceste situații adversariale sunt denumite jocuri.

Conceptul de Teorie a jocurilor a fost introdus in 1944 de către John von Neumann și Oskar Morgenstern pentru a grupa și caracteriza comportamentul agenților care participă la un joc. Teoria jocurilor este utilizată răspândit în economie, are un substrat matematic și modelează și anticipează comportamentul agenților prezenți pe piață, acestea fiind numite Zero-Sum-Games. Aceste teorii pot fi utilizate și în descrierea unor jocuri de complexitate redusă, abordabile de către un agent inteligent.

Scopul acestui subcapitol este de a prezenta diverse metode prin care un agent inteligent ar putea deveni capabil să joace un joc și să îl câștige. Câteva exemple de jocuri ce pot fi abordate prin această metodă sunt șah, X și 0, table etc. Algoritmii prezentați se referă în special la jocuri deterministe, în sensul că se cunoaște rezultatul fiecărei acțiuni, jocuri cu mutări în ture, pentru care se cunoaște exact starea nouă ce se va obține.

1. MiniMax

Pentru exemplificarea modului de funcționare a algoritmului MiniMax și întroducerea terminologiei specifice se va utiliza jocul de șah. Un concept esențial ce trebuie introdus este ideea de valoare statică a tablei de joc, notată ***s***, aceasta reprezentând calitatea (nota) pe care o tabla de joc p are la un moment dat. În Fig. 18 sunt prezentate două situații pentru o tablă de șah obținute prin generarea unor mutări diferite. Dacă se analizează cele două situații din punct de vedere a eficienței mutărilor, se poate concluziona că Fig.18a este mult mai promițătoare pentru alb decât fig 18b, deci valoarea statică a tablei de joc în Fig.18a trebuie să fie mai mare decât pentru Fig.18b.

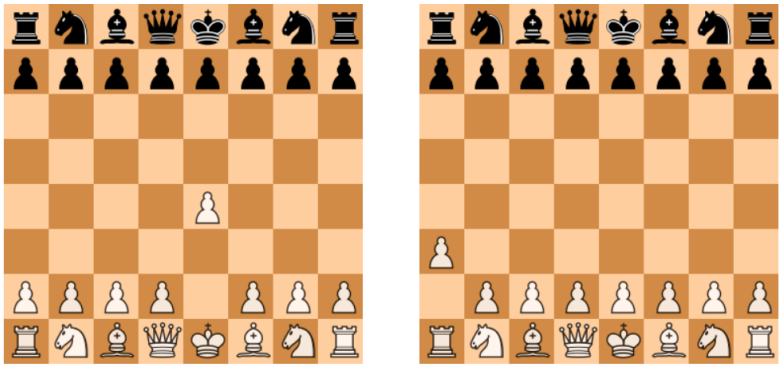

Fig.18 Situații de joc într-un meci de șah

O problemă în calcularea valorii statice constă în faptul că, agentul nu are aparat vizual, pentru a vizualiza tabla de șah și a conștientiza care poziție e mai bună, ca urmare, pentru calculul valorii statice se utilizează o funcție de evaluare f(), care poate avea forma prezentată în relația (4):

$$s = f(c_1, c_2, ..c_n) = k_1c_1 + k_2c_2 + ... + k_nc_n \quad (4)$$

unde s - reprezintă valoarea statică

f() – reprezintă funcția de evaluare, în cazul nostru este vorba de o combinație de criterii

c1, c2 ... cn – criteriile luate în considerare în determinarea calității unei table de joc

k1, k2 ... kn – coeficienții de ponderare a criteriilor

Câteva exemple de criterii utilizate în determinarea unei valori statice pentru o tabla de joc pot fi următoarele:

c1 – configurația tablei de joc (centru ocupat, centru atacat etc),

c2 – numărul de piese albe atacate,

c3 – numarul de piese negre atacate,

c4 – siguranța regelui etc.

Pasul următor în construirea unui agent inteligent care joacă șah este întegrarea acestui concept de evaluare statică într-un algoritm. Să presupunem că jocul de șah începe, iar agentul inteligent trebuie să execute o mutare, apare întrebarea, care mutare ar fi cea mai potrivită. Această situație este prezentată în Fig.19 rezultând un arbore de joc.

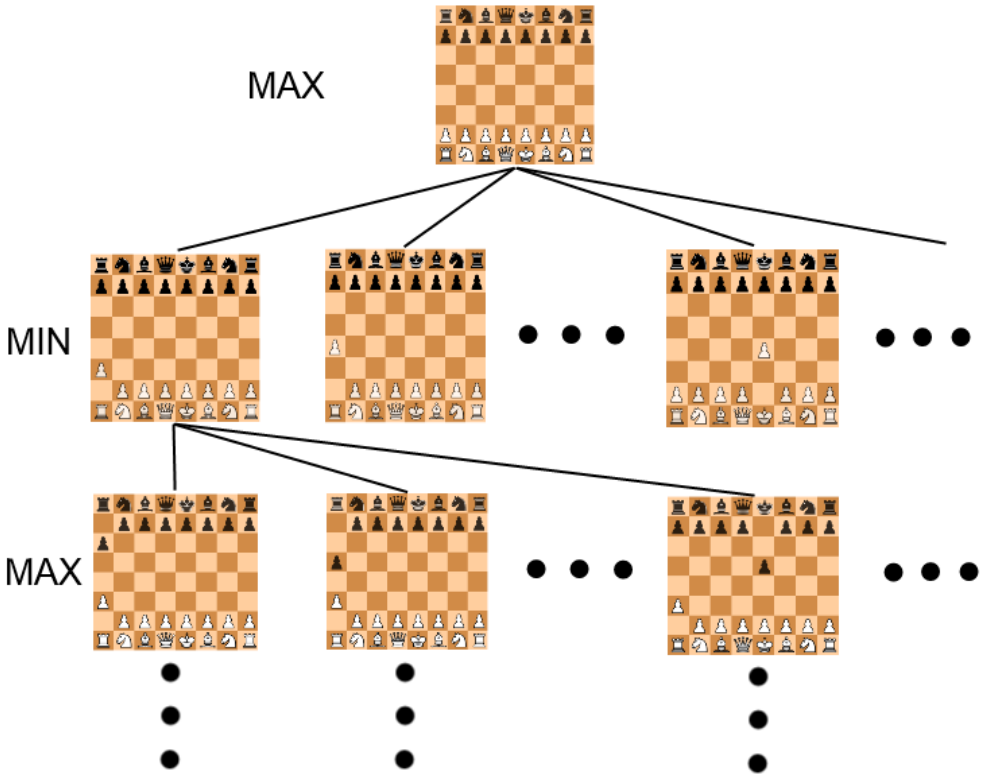

Fig.19 Dezvoltarea arborelui de joc pentru un joc de șah

În Fig. 19 se observă că se pleacă de la tabla de șah cu piesele ne-mutate. Se începe prin generarea tuturor mișcărilor posibile pentru alb - primul jucător - maximizator, apoi pentru fiecare mutare a albului se generează toate posibilitățile de răspuns ale negrului - minimizator, procesul fiind reluat

până când se obține un câștigător, după care se urmărește înapoi către rădăcină secvența de mutări care a dus la situația câștigătoare.

Cu toate că această metodă pare abordabilă, ea funcționează doar pentru unele jocuri extrem de simple cum ar fi X și 0, dar pentru jocul de șah nu este o variantă fezabilă pentru faptul că arborele de joc astfel dezvoltat este gigantic. Dacă considerăm că, la fiecare nivel există în medie 10 posibilități de mișcare (pion 1 careu avans, 2 careuri avans, cal, turn etc), acest parametru fiind numit factor de ramificare. De asemenea, să presupunem că jocul de șah are în medie 100 de alternații alb-negru, asta ar duce la un arbore cu 10^{100} noduri terminale, un arbore enorm care în realitate nu poate fi calculat cu mijloacele de calcul ce stau omenirii astazi la dispoziție. În concluzie, arborele de joc nu se dezvoltă până la stabilirea unui câștigător, ci un număr limitat de mutări. Utilizând această abordare, rezultă un arbore de joc cu număr limitat de niveluri, fiecare nod terminal al arborelui având o valoare statică. Un asemenea arbore de joc este prezentat în Fig. 20, unde arborele de joc s-a dezvoltat doar două niveluri, numerele înscrise în nodurile terminale ale arborelui reprezentând valorile statice ale tablei de joc. Arborele din Fig.20 fiind asociat secvenție de joc prezentate în Fig.19.

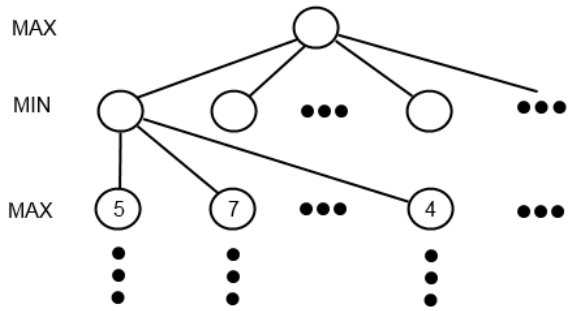

Fig. 20 Arbore de joc cu valori statice asociate

În secvența de joc, pornind de la o configurație a tablei de joc, asociată nodului rădăcină, primul agent inteligent alege mișcarea optimă, aceasta se reduce la identificarea tablei de joc de la nivelul inferior cu valoarea statică cea mai mare, de aici numele de agent maximizator al acestuia. După efectuarea mișcării urmează mișcarea celui de-al doilea agent, care dorește și el un optim, de data aceasta inversul maximizatorului, de aici denumirea celui de-al doilea agent de minimizator. Ca urmare, la fiecare nivel se alege valoarea minimă sau maximă în funcție de tipul nivelului. Folosind această abordare, toate valorile statice de la nodurile terminale trebuiesc propagate către rădăcină ținând cont de tipul fiecărui nivel.

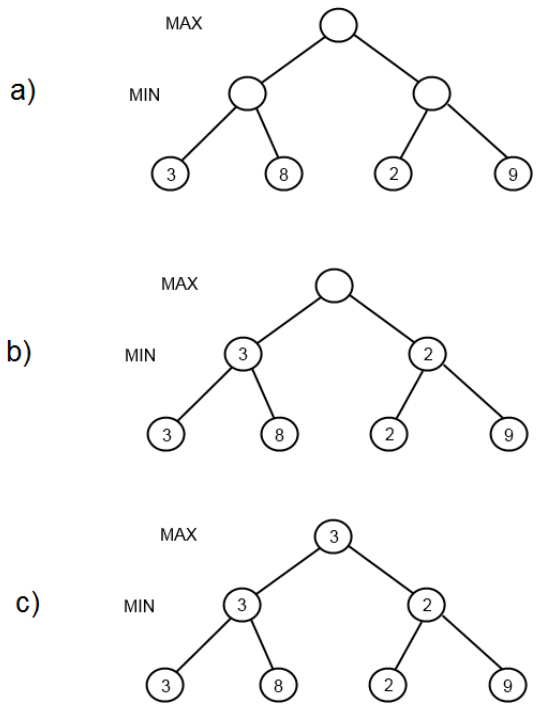

Fig. 21 Arbore de joc simplificat în scop educativ

În Fig. 21 se prezintă un arbore de joc simplificat în scop educativ, fiind vorba despre un joc cu factor de ramificare doi, iar nivelul până la care se face generarea de mutări este de asemenea doi. În jocul nostru, după secvențele de mutări se obțin valorile statice pentru nodurile terminale sunt prezentate în Fig. 21a ultimul nivel. Aceste valori trebuie propagate către rădăcină ținând cont de tipul fiecărui nivel (min/max). Valorile propagate către nodurile din nivelul MIN sunt valorile minime ale nodurilor copil, astfel rezultând Fig.21b. De exemplu, MIN(3,8) = 3. În mod similar se procedează și pentru pasul următor, diferența fiind în faptul că se alege maximul dintre nodurile copil, rezultând Fig. 21c, de exemplu MAX(3,2)=3.

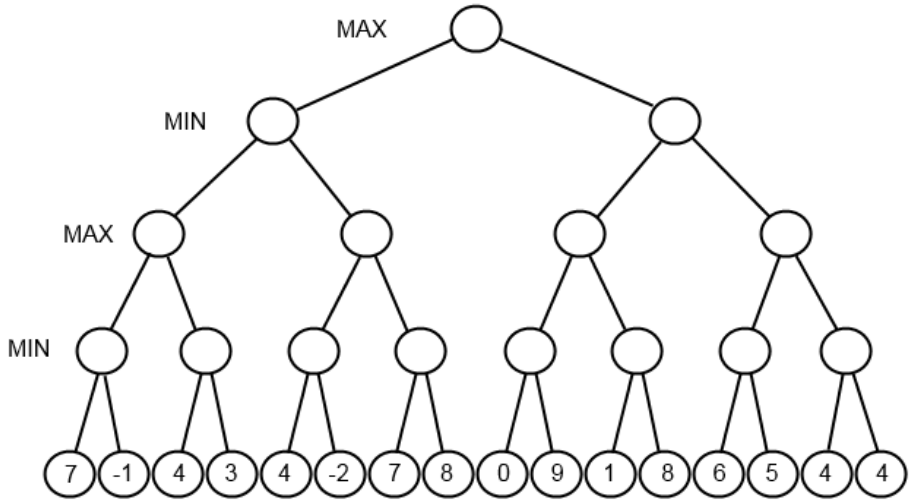

Fig. 22 Arbore de joc cu adâncime de 4

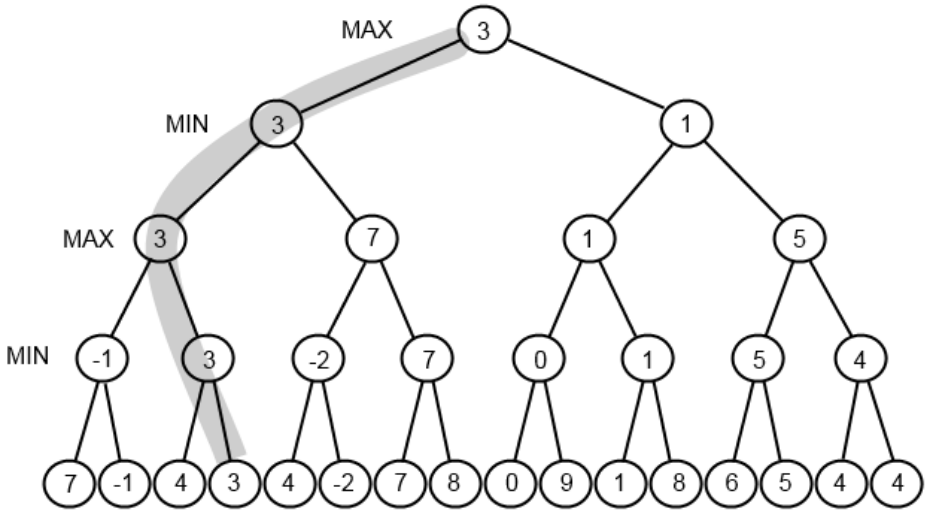

Fig. 23 Arbore de joc cu valorile propagate la rădăcină și marcaj către calea câștigătoare

Pentru a ilustra mai bine modul de funcționare al algoritmului MiniMax, în Fig. 22 se prezintă un arbore de joc cu factor de ramifiare 2 și adâncime 4 niveluri. Rezolvarea acestui arbore de joc este prezentată în Fig. 23 unde se poate observa și secvența de mișcări ce duce la rezultatul obținut de către maximizator în rădăcină.

O mare problemă a lui MiniMax este creșterea exponențială a numărului de noduri terminale într-un arbore de joc odată cu creșterea nivelurilor de adâncime. Eficiența algoritmului se poate optimiza utilizând o strategie numită Alfa-Beta.

2. Minimax cu Alfa-Beta

Algoritmul MiniMax devine ineficient odată cu creșterea în adâncime a arborelui de căutare pe motiv că numărul de noduri terminale ce trebuie evaluate crește exponențial. În continuare se prezintă strategia Alfa-Beta de optimizare a algoritmului MiniMax. Aceasta se bazează pe ideea că din arborele de căutare pot fi eliminate portiuni pe motiv că acestea nu vor fi selecționate niciodată pentru a fi urmate. Pentru introducere, conceptul se prezintă pe arborele din Fig. 21a, dezvoltarea algoritmului fiind prezentată în Fig. 22.

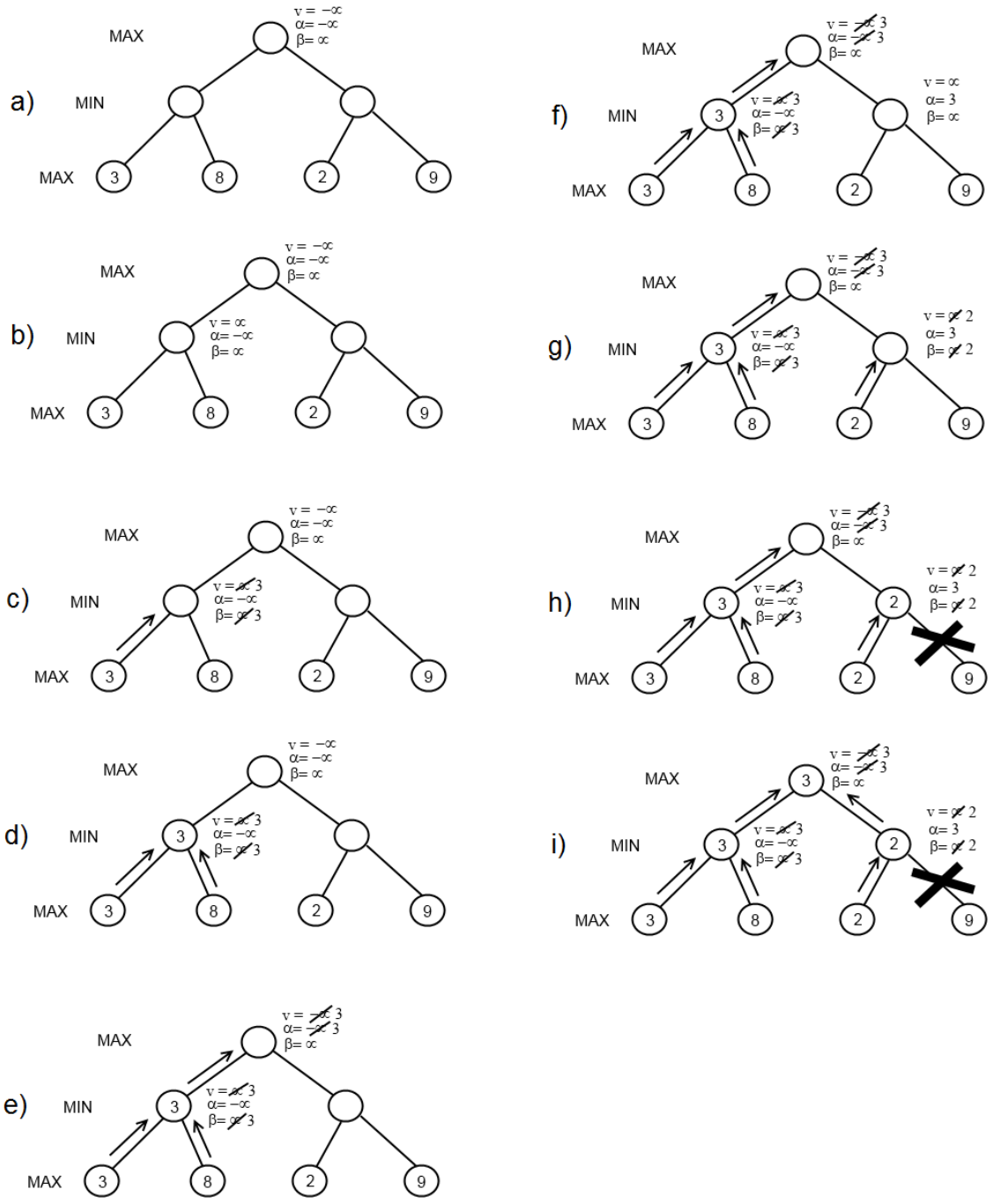

Fig. 22 Secvența de obținere a unui arbore de joc folosind optimizarea Alfa-Beta

Descrierea modului de funcționare. Se pleacă de la nodul rădăcină, acestuia se asociază 3 variabile:(v,α,β), având următoarele valori: (v=-∞, α=-∞, β=∞), acestea fiind reprezentate în Fig. 22a lângă nodul rădăcină. Prima variabilă "v" reprezintă valoarea nodului curent, în cazul inițializării, aceasta se inițializează cu -∞ dacă nodul este de MAX, respectiv cu ∞ pentru nod de MIN și reprezintă valoarea cea mai nefavorabilă pe care o poate lua nodul respectiv. Practic, pentru maximizator, cea mai nefavorabilă valoare este -∞ și invers pentru minimizator. Variabilele "α" și "β" reprezintă limitele între care se va focaliza căutarea în arborele de joc, "α" corespunde maximizatorului iar "β" minimizatorului.

În fig. 22b se inițializează nodul minimizator din stânga cu valorile (v=∞, α=-∞, β=∞). Se observă că variabila "v" ia valoarea ∞ corespunzătoare nodului minimizator iar valorile lui "α" și "β" sunt preluate de sus. Trebuie subliniat că arborele de joc se abordează într-o ordine de tip căutare în adâncime.

În Fig. 22c se observă că se ajunge la un nod terminal cu valoare statică, în cazul acesta se efectuează următoarele operații

v=MIN(valoarea curentă a lui v=∞, valoarea statică=3)=3

β=MIN(valoarea curentă a lui β=∞, valoarea curentă a lui v=3)=3

pentru că este un nod de minim, dacă era un nod de maxim, se folosea funcția MAX().

Se verifică dacă se îndeplinește condiția de tăiere pentru un nod de minim:

dacă (v ≤ α) atunci nu mai evalua restul ramurilor nodului curent (în engleză: α-cut)

Pentru situația noastră, v=3, α=-∞, condiția de mai sus nu se îndeplinește deci trebuie evaluată următoarea ramură.

În Fig.22d se evaluează următoarea ramură care este tot un nod terminal și se efectuează aceleași două operații

v=MIN(valoarea curentă a lui v=3, valoarea statică=8)=3

β=MIN(valoarea curentă a lui β=3, valoarea curentă a lui v=3)=3

Nodul curent nu mai are alți copii, deci valoarea sa rămâne 3.

În Fig. 22e valoarea nodului se propagă la părinte, unde se efectuează cele două operații, de data aceasta pentru un nod de maxim

v=MAX(valoarea curentă a lui v=-∞, valoarea statică=3)=3

α=MAX(valoarea curentă a lui α=-∞, valoarea curentă a lui v=3)=3

Se verifică dacă se îndeplinește condiția de tăiere pentru un nod de maxim:

dacă (v ≥ β) atunci nu mai evalua restul ramurilor nodului curent (în engleză: β-cut)

Pentru situația noastră, v=3, $\beta=\infty$, condiția de mai sus nu se îndeplinește deci trebuie evaluată următoarea ramură.

În fig. 22f se inițializează nodul minimizator din dreapta cu valorile (v=∞, α=3, $\beta=\infty$). Se observă că variabila "v" ia valoarea ∞ corespunzătoare nodului minimizator iar valorile lui "α" și "β" sunt preluate de sus.

În Fig. 22g se observă că se ajunge la un nod terminal al cărui valoare este propagată în sus și se efectuează cele două operații

v=MIN(valoarea curentă a lui v=∞, valoarea statică=2)=2

β=MIN(valoarea curentă a lui $\beta=\infty$, valoarea curentă a lui v=2)=2

Se verifică dacă se îndeplinește condiția de tăiere pentru un nod de minim:

dacă (v ≤ α) atunci nu mai evalua restul ramurilor nodului curent (în engleză: α-cut)

Pentru situația noastră, v=2, α=3, condiția de mai sus se îndeplinește deci următoarea ramură nu se mai evaluează producând economie de timp și memorie. Cititorul ar putea spune că acest lucru nu este adevărat, pentru că s-a muncit foarte mult pentru acest exemplu, dar să nu uităm că acesta este doar un exemplu demonstrativ al conceptului, și în realitate am avea situații cu arbori de ordinul 10^{10} unde o reducere cu 25% este semnificativă, însemnând 10^7 evaluări.

În Fig.22h se observă că ultima ramură nu este evaluată iar nodul părinte primește valoarea 2

În Fig. 22i valoarea nodului se propagă la părinte, unde se efectuează cele două operații, de data aceasta pentru un nod de maxim

v=MAX(valoarea curentă a lui v=3, valoarea statică=2)=3

α=MAX(valoarea curentă a lui α=3, valoarea curentă a lui v=3)=3

Nodul curent nu mai are copii, ca urmare valoarea lui rămâne v=3. De asemenea acest nod este și nodul rădăcină, v=3 fiind valoarea obținută de maximizator.

Dacă se analizează algoritmul MiniMax cu algoritmul MiniMax+AlfaBeta se obține același rezulta, avantajul lui Alfa-Beta fiind faptul că porțiuni din arborele de joc ajung să nu mai fie evaluate nici măcar din punct de vedere al valorii statice. Practic nu ne mai interesează ce valoare apare pe ramura tăiată, poate fi ∞ sau -∞, aceasta nu infșuențează rezultatul final.

Ca recapitulare cu privire la condițiile în care nu se face evaluarea, pentru noduri

MIN: dacă (v ≤ α) atunci nu mai evaluam restul ramurilor nodului curent (în engleză: α-cut)

MAX: dacă (v ≥ β) atunci nu mai evaluam restul ramurilor nodului curent (în engleză: β-cut)

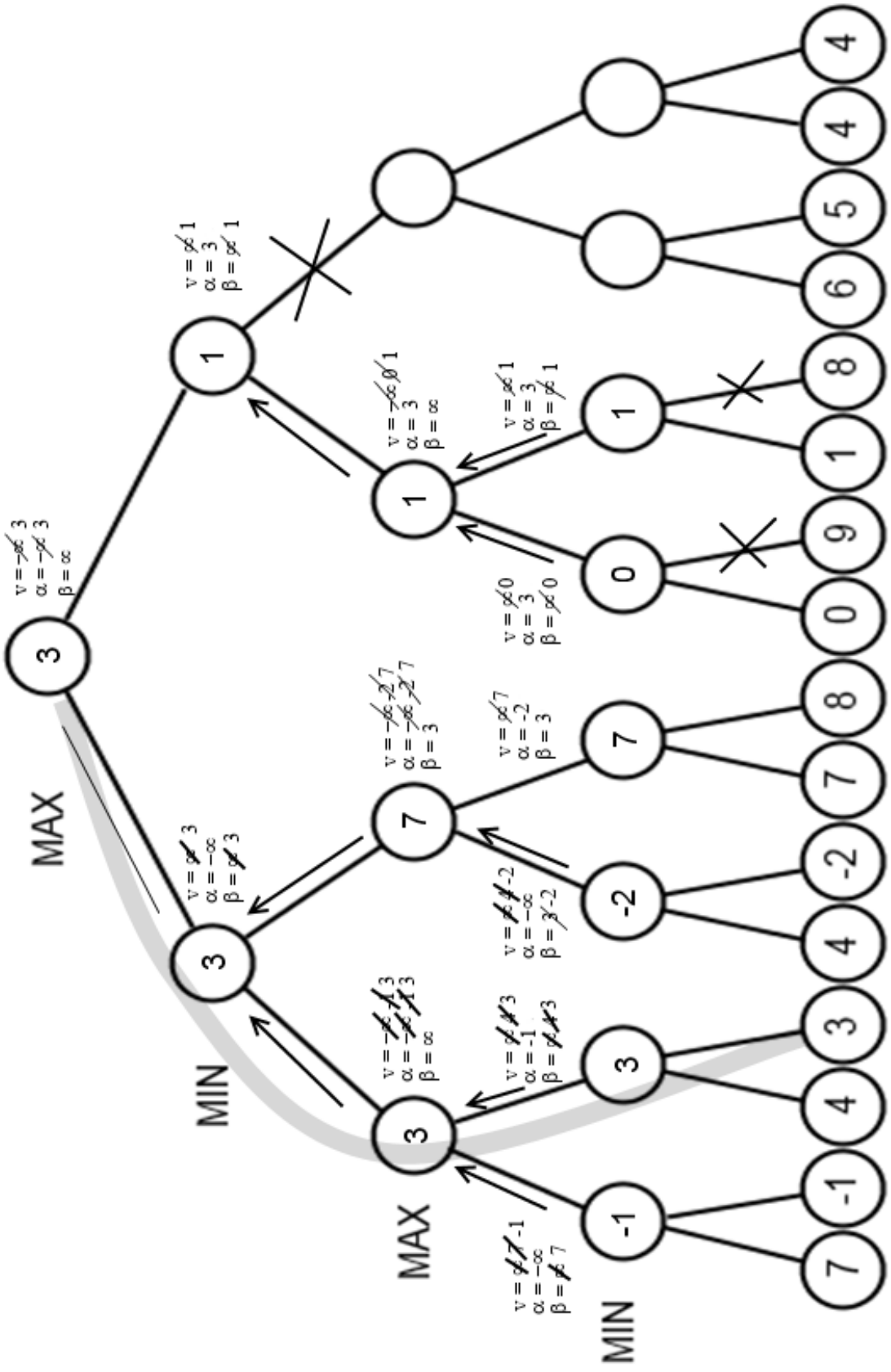

Fig. 23. Arbore de joc pe care se execută MINIMAX cu Alfa-Beta

În Fig. 23 se prezintă un arbore de joc cu adâncime de patru niveluri și cu factor de ramificare egal cu 2 asupra căruia se execută algoritmul MiniMax cu Alfa-Beta unde se pot urmări valorile pe care le obține fiecare nod, dar și ramurile ce nu sunt evaluate utilizând această metodă.

3. Progressive Deepening/Iterative Deepening

Algoritmii de tipul Progressive Deepening/Iterative Deepening, numiți în continuare IDA (Iterative Deepening Algorithm) se bazează pe o abordare de tip DFS (Depth First Search), dar spre deosebire de algoritmul DFS clasic, IDA nu dezvoltă o ramură din arborele de căutare până la capăt, ci doar o adâncime specificată. IDA începe căutarea și construirea arborelui de căutare de la adâncimea 0, apoi 1, 2 și așa mai departe, verificând dacă s-a găsit ținta întretimp.

Acest tip de algoritmi (IDA) se pot utiliza și în cadrul jocurilor oferind o soluție suboptimală la fiecare nivel dacă se utilizeaza MiniMax sau Minimax cu Alfa-Beta. Să presupunem că timpul de căutare pentru o soluție este limitat, iar computerul nu are timp să coboare un număr prea mare de niveluri în arborele de căutare, și în mod suplimentar, în funcție de complexitatea jocului nivelul de profunzime poate varia, în aceste situații algoritmii tip IDA pot oferi o secvență de mutări suboptimală dar eficientă dat fiind timpul limitat pentru calcularea mutării.

Concret, utilizând ideea IDA, la fiecare nivel din arborele de joc, algoritmul va avea pregătită o secvență de mutări. Să presupunem că algoritmul a ajuns să identifice o secvență de mutări la nivelul 3 și mai are timp să abordeze nivelul 4. Dacă timpul de căutare expiră pe când calculatorul caută o soluție pe nivelul 4, se poate oferi solutia gasită la nivelul 3. Dacă computerul finalizează nivelul 4 se poate reține această soliție și începe nivelul 5.

De asemenea, dacă la fiecare nivel se rearanjează nodurile descrescător/crescător se poate maximiza numărul de tăieri ce se pot executa, ca urmare se reduce spațiul de căutare și crește economia de timp.

Acest tip de algoritmi care pot oferi o soluție, chiar suboptimală dar suficient de bună, la orice moment dat se numesc algoritmi AnyTime.

În concluzie, s-au prezentat câțiva algoritmi de căutare în cadrul jocurilor, MiniMax ce poate oferi o secvență de mutări ce poate aduce agentul în câștig, MiniMax cu Alfa-Beta ca o variantă îmbunătățită și eficientizată a lui MiniMax, ce ne permite creșterea adâncimii de căutare, apoi se prezintă o variantă de algoritm care poate produce soluții oricând utilizând abordarea IDA.

Rezolvarea problemelor ce presupun satisfacerea unor constrângeri utilizând algoritmi de căutare (Constraint Satisfaction Problem)

Algoritmii de căutare pot apărea sub diverse forme, până acum am discutat despre algoritmi de căutare în spațiul stărilor a unei secvențe ce permite conectarea unei stări start cu o stare țintă sau despre algoritmi ce permit unui agent să aleaga o succesiune stări care, în final, să îi aducă câștig de cauză într-un joc, în continuare vom discuta despre algoritmi care sunt capabili să aleagă o stare ținând cont de constrângeri, acest tip de algoritmi intră în categoria CSP (Constraint Satisfaction Problems).

Fig. 24 Regiunile României [https://ro.m.wikipedia.org/wiki/Fi%C8%99ier:Romania_Regions.png]

Un exemplu de problemă este colorarea unei hărți precum cea din Fig. 24 utilizând doar 4 culori. Problema apare prima dată în 1852 în contextul dorinței de a imprima hărți colorate utilizând un număr minim de culori sub numele Teorema celor patru culori, dar este rezolvată pentru orice hartă (se face referință la o situație generalizată) abia în 1976 de către Kenneth Appel și Wofgang Haken utilizând un calculator și un algoritm de căutare și este prima Teoremă majoră demonstrată prin intermediul unui calculator. Teorema spune că orice suprafață plană împărțită în spații contigue, numită în continuare hartă, poate să fie colorată cu un număr minim de patru culori astfel încât nicicare două zone adiacente să nu aibe aceeași culoare.

Ținând cont de aceste constrângeri, sarcina agentului este de a colora harta României utilizând 4 culori. O reprezentare alternativă topologic a hărtii României din Fig.24 este prezentată în Fig. 25a unde se retine pozitia si vecinii fiecărei suprafețe și se poate observa utilizarea a doar 4 culori pentru colorarea acesteia.

Dat fiind faptul că se dorește rezolvarea computerizată a acestei probleme concrete dar și construirea unui algoritm generalizat de rezolvare a acestui tip de probleme, se propune o reprezentare topologică pe baza unei structuri de tip graf, ilustrată în Fig. 25b, aceasta purtând numele de rețea de constrângeri.

În această reprezentare,

- fiecare nod este abordat ca o variabilă V , are un nume și poate primi o valoare x
- există constrângeri unare numite in continuare C1 asupra unui nod
- fiecare arc reprezintă o constrângere binara numită în continuare C2 dintre două noduri.

Se introduce conceptul de Domeniu D ca fiind multimea de valori ce pot fi atribuite unui nod/variabile.

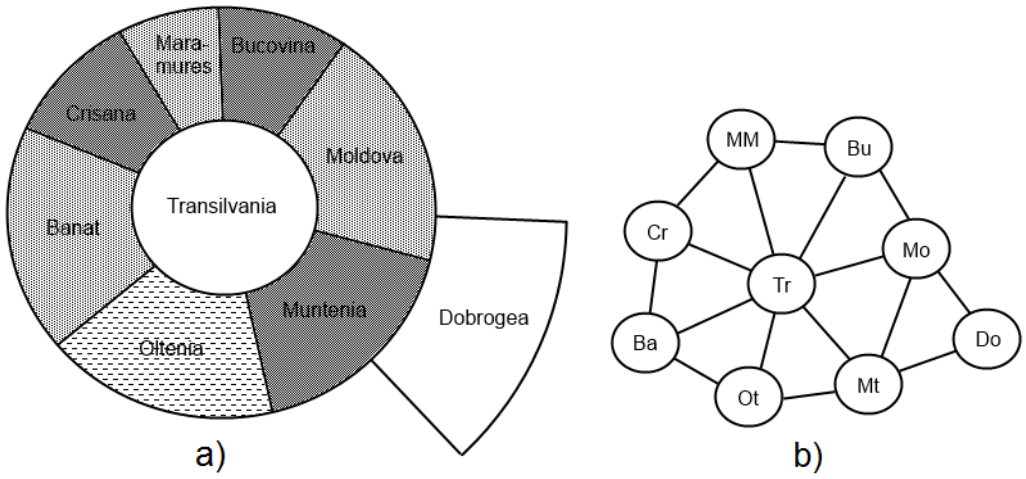

Fig. 25 Reprezentari topologice ale hărții României

Un exemplu de constrângere unară C1 ar fi ca un nod să poată primi doar valori pare din domeniul de valori D={0,1,2,3,4,5,6}, iar o constrângere binară C2 ar putea fi – nu aceeași culoare între două noduri.

1. Constraint Satisfaction Problem – Depth First Search (CSP-DFS)

Pentru cazul nostru,

- fiecare nod reprezintă o variabilă și poartă un nume V = {Ba, Cr, MM, Bu, Mo, Do, Mt, Ot, Tr},
- valorile pe care le poate primi o variabilă sunt x = {Yellow, Red, Green, Blue}
- constrângerile C1 nu există, iar C2 se materializează sub forma arcelor și reprezintă ideea de "nu au aceeași valoare", de exemplu (Ba,Tr) – nu aceeași culoare.

O reprezentare vizuală a modului de funcționare al algoritmului este ilustrată în Fig. 26. Structura obținută este un arbore de căutare de tip Căutare în Adâncime care pe fiecare nivel are valorile posibile ce le poate lua o variabilă.

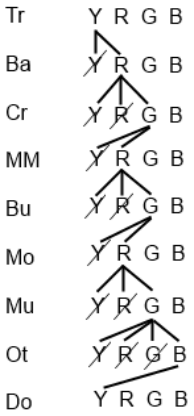

Fig. 26 Arborele de căutare al unei soluții de colorare a hărții, care să satisfacă toate constrângerile impuse de arce

Concret, prima variabilă abordată este Tr ce poate lua orice valoare {Yellow, Red, Green, Blue} se verifică constrângeri și pentru că acestea nu există, vom atribui Tr = Yellow.
Următoarea variabilă este `Ba`, se atribuie `Ba=Yellow`, se verifică constrângerile, se observă că Ba=Yellow intră în contradicție cu Tr=Yellow, ca urmare apare un pas de Backtrack și Ba ia următoarea valoare Ba=Red ce convine.
Următoarea variabilă este Cr,

se atribuie Cr=Yellow - contradicție cu Tr=Yellow și Backtrack;
Cr=Red – contradicție cu Ba=Red și Backtrack;
Cr=Green - convine.

Următoarea variabilă este MM,

se atribuie MM=Yellow - contradicție cu Tr=Yellow și Backtrack;
MM=Red - convine.

Următoarea variabilă este Mu,

se atribuie Bu=Yellow - contradicție cu Tr=Yellow și Backtrack;
Mu=Red – contradicție cu MM=Red și Backtrack;
Mu=Green - convine.

Următoarea variabilă este Ot,

se atribuie Ot=Yellow - contradicție cu Tr=Yellow și Backtrack;
Ot=Red – contradicție cu Ba=Red și Backtrack;
Ot=Green – contradictie cu Mu=Green și Backtrack;
Ot=Blue - convine.

Următoarea variabilă este Do,

se atribuie Do=Yellow - convine.

Acest algoritm bazându-se doar pe backtracking îl vom denumi CSP-DFS (Constraint Satisfaction Problem – Depth First Search), iar pentru a rezolva problema în acest mod se propune următorul pseudocod [https://www.ics.uci.edu/~dechter/publications/R69.pdf]

CSP-DFS
1. (Pas inainte) Dacă V_{curent} este ultima variabilă și are valoare, atunci toate celelalte variabile au valori acceptate, termină algoritmul și afișează aceste valori. Dacă nu, treci la următoarea variabilă, $V_{curent} = V_{curent\_next}$
2. (Alegera valorii) Alege valoarea $x \in D_{curent}$ pentru variabila V_{curent} astfel încât aceasta să nu fie în contradicție cu celelalte valorile celorlalte variabile, astfel:
a) Dacă $D_{curent} = \phi$ (nu mai sunt valori in Domeniu din care să alegem), mergi la 3.
b) Extrage următoarea valoare $x \in D_{curent}$ (în acest pas, valoarea extrasă e ștearsă din D_{curent})
c) Verifică dacă $V_{curent} = x$ satisface toate constrangerile impuse de variabilele $V_1..V_{curent-1}$, dacă nu, sari la a)
d) Atribuie $V_{curent} = x$ și sari la 1
3. (Backtrack) Dacă V_{curent} este prima variabilă, termina algoritmul și afișează “contradictie/inconsistent” altfel treci la variabila anterioară, $V_{curent} = V_{curent\_anterior}$ și sari la 2.

Algoritmul CSP-DFS poate găsi o soluție dacă aceasta există, doar că, în unele situații, aranjamentul variabilelor este neconvenabil și algoritmul rulează perioade îndelungate de timp. Pentru o problemă ușor mai complicată prezentată în Fig. 27, dacă ordinea de abordare a colorării suprafețelor este S1-S8 și se introduce o constrângere suplimentară astfel încât nici o culoare să nu fie utilizată în exces, s-ar întâmpla următoarea alocare:

S1-Yellow; S2-Red; S3-Green; S4-Blue; S5-Yellow; S6-Green; S7-Blue; S8 - ?

Se poate observa că S8 nu mai are culoare disponibilă în Domeniu, ca urmare algoritmul CSP-DFS trebuie să revină la atribuirea unei valori noi pentru S4 iar pentru a face acest lucru va încerca toate posibilitățile de alocare pentru S5-S7, algoritmul irosind timp și resurse în acest proces. Ca urmare se impune o strategie mai elaborată de găsire a unei soluții pentru această problemă, algoritmul optimizat pentru această situație fiind numit CSP-FC (Constraint Satisfaction Problem – Forward Checking).

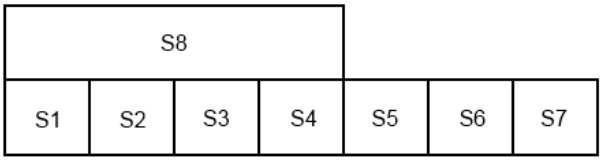

Fig. 27 Harta cu dispunere neconvenabilă a suprafețelor și constrângere suplimentară de utilizare echilibrată a culorilor

2. Constraint Satisfaction Problem – Forward Checking (CSP-FC)

Algoritmul CSP-FC este o optimizare a algoritmului CSP-DFS în sensul că de fiecare dată când se atribuie o valoare unei variabile/nod, această valoare este ștearsă din domeniul vecinilor. Dacă se constată că o variabilă are domeniul redus la mulțime vidă, algoritmul efectuează un pas de backtracking și încearcă alocarea unei valori noi pentru nodul anterior.

Pentru problema din Fig. 27 acest lucru se manifestă în felul următor, se atribuie valori în ordinea:

S1=Yellow $\Rightarrow$ $D_{S2}=\{Red, Green, Blue\}$; $D_{S8}=\{Red, Green, Blue\}$;
S2=Red $\Rightarrow$ $D_{S3}=\{Green, Blue, Yellow\}$; $D_{S8}=\{Green, Blue\}$;
S3=Green $\Rightarrow$ $D_{S4}=\{Blue, Yellow, Red\}$; $D_{S8}=\{Blue\}$;
S4=Blue $\Rightarrow$ $D_{S5}=\{Yellow, Red, Green\}$; $D_{S8}=\phi$

se observă că $D_{S8}=\phi$, ca urmare se revine și se atribuie o valoare nouă lui S4,

S4=Yellow $\Rightarrow$ $D_{S5}=\{Red, Green, Blue\}$; $D_{S8}=\{Blue\}$;
S5=Red $\Rightarrow$ $D_{S6}=\{Green, Blue, Yellow\}$; $D_{S8}=\{Blue\}$;
S6=Green $\Rightarrow$ $D_{S7}=\{Blue, Yellow, Red\}$; $D_{S8}=\{Blue\}$;
S7=Blue $\Rightarrow$ $D_{S8}=\{Blue\}$;
S8=Blue $\Rightarrow$ Terminare cu succes

Diferenta de abordare dintre algoritmul CSP-DFS și CSP-FC constă în faptul că CSP-FC își dă seama că S8 nu are valori în domeniu odată cu atribuirea unei valori pentru S4, spre deosebire de CSP-DFS care încearcă toate posibilitățile după care face backtracking până la modificarea culorii atribuite lui S4.

Algoritmul CSP-FC funcționează după următorul pseudocod:

CSP-FC
0. (Inițializare) Toate D_i vor fi inițial complete cu valorile pe care le poate primi variabilă V_i respectivă

1. (Pas inainte) Dacă V_{curent} este ultima variabilă și are valoare, atunci toate celelalte variabile au valori acceptate, termină algoritmul și afișează aceste valori. Dacă nu, treci la următoarea variabilă, $V_{curent} = V_{curent\_next}$
2. (Alegera valorii) Alege valoarea $x \in D_{curent}$ pentru variabila V_{curent} astfel încât aceasta să nu fie în contradicție cu celelalte valorile celorlalte variabile, astfel:
   a) Dacă $D_{curent} = \phi$ (nu mai sunt valori in Domeniu din care să alegem), mergi la 3.
   b) Extrage următoarea valoare $x \in D_{curent}$ (în acest pas, valoarea extrasă e ștearsă din D_{curent})
   c) Examinează următoarele variabile V_j, unde $j = curent \ldots n$. Elimină valorile x din D_j pentru situațiile de conflict între $V_{curent} = x$ și V_j. Dacă în acest proces se ajunge la vreun $D_j = \phi$, revino la valorile D_j înainte de începerea acestui pas (c) și mergi la a)
   d) Atribuie $V_{curent} = x$ și mergi la 1.
3. (Backtrack) Dacă V_{curent} este prima variabilă, termina algoritmul și afișează "contradictie/inconsistent" altfel treci la variabila anterioară, $V_{curent} = V_{curent\_anterior}$; resetează toate D la valorile înaintea atribuirii lui V_{curent}. Sari la 2.

3. Optimizări și concluzii

Aceste tipuri de algoritmi pot avea diverse aplicații, cum ar fi alocare computerizată de resurse și planificare automată, WSD (Word Sense Disambiguation) – determinarea automatizată a sensului unor cuvinte ce apar într-o propoziție sau rezolvarea unor jocuri tip puzzle: 8 Queens, Sudoku, Integrame etc. Pentru a li se crește eficiența algoritmii sus prezentați pot fi optimizați prin diferite metode, cele mai populare fiind

1. ordonarea nodurilor în ordine descrescătoare după numărul constrângerilor
2. extinderea căutării în cazul variabilelor care au domeniul redus la un singur element
3. pentru situațiile în care timpul de găsire al unei soluții este mult prea mare, se poate incepe de la o problemă mai relaxată, cu număr mare de resurse, pentru ca apoi să se reducă numărul resurselor până se obține un optim. De exemplu, colorarea hărții se poate face mai ușor utilizând 5 culori în loc de cele 4 impuse.

În cazul extinderii căutării în cazul variabilelor care au domeniul redus la un singur element algoritmul identifică variabilele cu $D = 1$ iar pentru acestea execută o căutare simulând alocarea acelei valori singulare variabilei și urmărind efectele. Dacă în acest proces de simulare se formează noi variabile cu domenii singulare, procesul se repetă.

Concluzii Capitolul 4

În acest capitol s-au prezentat diverse strategii ptin care un agent inteligent poate să identifice o serie de acțiuni care dacă sunt executate îl aduc în starea țintă, procesul de identificare a acțiunilor potrivite fiind numit **căutare**.

S-au prezentat câteva chestiuni de formalizare și struturare a oricărei probleme ce se dorește a fi rezolvată prin căutare, anume,

- problema se prezintă sub forma unor stări interconectate ce formează **spațiul stărilor**, această reprezentare fiind memorată în calculator sub forma unui graf
- pentru a rezolva problema se identifică **starea țintă**, **starea de start/plecare** și **starea curentă**
- ca urmare a rulării unui algoritm de căutare, în memoria calculatorului se produce o structură de tip **arbore**

Există în principal două categorii de algoritmi de căutare:

- algoritmi de **căutare neinformată**, în sensul că, pe graful ce reprezintă spațiul stărilor nu sunt prezente informații legate de costul de a trece dintr-o stare în alta. Din această categorie fac parte Căutarea în Adâncime (DFS Depth First Search) și Căutarea pe Nivel (BFS Breadth First Search)
- algoritmi de **căutare informată**, pe graf apar informații care pot dirija/optimiza calea găsită de algoritm. Din această categorie fac parte algoritmii Hill Climbing, Fascicul (Beam), Branch and Bound, A*. Informația găsită pe graf se poate împărți în două categorii:
    - **cost** de trecere dintr-un nod în altul
    - cost estimat de a ajunge dintr-un nod în altul, numim acest tip de informație, **distanță euristică**

Aplicații ale algoritmilor de căutare se pot găsi în jocuri, algoritmul MimiMax, MiniMax cu AlfaBeta, sau în rezolvarea unor probleme ce prezintă constrângeri.

Algoritmii de căutare reprezintă metode eficiente de rezolvare a unei game largi de probleme, dezavantajul major al acestora fiind consumul mare de resurse: timp, memorie și procesor, dar, cu toate acestea algoritmii de căutare sunt o componentă bine reprezentată în zona algoritmilor de Inteligență artificială.