UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

LUCRARE DE DIPLOMĂ

Teoria Jocurilor: Învățare versus Minimax

Conducător ştiinţific

Conf. Dr. Gabriela Şerban

Absolvent

Budurlean Adrian

2008

Cuprins

1. Inteligența Artificială 6

1.1 Ce este Inteligența Artificială? 6

1.2 Comportamentul uman. Testul Turing 7

1.3 Istoric al Inteligenței Artificiale 9

1.3.1 Antichitatea 9

1.3.2 Renaşterea 10

1.3.3 Primele calculatoare 10

1.3.4 Rețelele neuronale 11

1.3.5 Naşterea "Inteligenței Artificiale" 11

1.3.6 Anii ' 60 12

1.3.7 Generația a treia de calculatoare 12

1.3.8 Microprocesoarele. Întoarcerea rețelelor neuronale 13

1.3.9 Progrese recente 13

1.4 Prezentul în Inteligența Artificială 14

2. Teoria jocurilor 15

2.1 Jocurile şi Inteligența Artificială 15

2.2 Jocuri în forma extinsă şi jocuri în forma normalizată 15

2.3 Clasificarea jocurilor 17

2.4 Strategie şi funcție de câştig 18

2.4.1 Echilibrul Nash 18

2.5 Evoluția jocurilor 19

3. Metoda Minimax 21

3.1 Jocul ca problemă de căutare 21

3.1.1 Componentele unei probleme de căutare 21

3.2 Algoritmul Minimax 22

3.2.1 Tic-Tac-Toe 23

3.2.2 Paşii algoritmului Minimax 24
3.2.3 Proprietăți ale algoritmului Minimax 25
3.2.4 Funcția de evaluare 25
3.3 Minimax cu tăietură alfa-beta 26
3.3.1 Descrierea algoritmului 27
3.3.2 Proprietăți 30
3.4 Alternative la Minimax 30
4. Învățarea prin întărire 32
4.1 Introducere 32
4.2 Concepte generale 32
4.2.1 Învățarea bazată pe Diferențe Temporare 34
4.3 TD-Gammon 34
4.3.1 Prezentarea jocului de table 35
4.3.2 Modul de învățare 36
4.3.3 Rezultate 38
4.4 Concluzie 39
5. Aplicația „X – O” 41
5.1 Analiza 41
5.1.1 Importanța subiectului 41
5.1.2 Jocul Tic-Tac-Toe 41
5.1.2 Tic-Tac-Toe, o altă abordare 43
5.1.3 Reprezentarea învățării 46
5.2 Implementarea 47
5.2.1 Mediul de programare 47
5.2.2 Minimax 48
5.2.3 Învățare prin întărire şi rețea neuronală 49
5.3 Manual de utilizare 50
6. Concluzii 54
7. Bibliografie 55

INTRODUCERE

Jocurile sunt fascinante, iar scrierea de programe care să le joace este chiar şi mai fascinantă. Se poate spune că teoria jocurilor este pentru Inteligența Artificială cum este Grand Prix-ul pentru constructorii de motoare.

În lucrarea de față ne-am propus să facem o prezentare a învățării prin întărire ca alternativă la abordarea tradițională a rezolvării jocurilor prin metoda Minimax.

Minimax este considerat la scară largă ca fiind algoritmul de bază în rezolvarea jocurilor, însă vom vedea că are o sumedenie de deficiențe. În primul rând complexitatea lui este una exponențială ceea ce-l face greu de adaptat unui joc cu multe stări cum ar fi şahul sau tablele. Un alt dezavantaj este acela că Minimax pleacă mereu de la premisa că adversarul său face cea mai bună mutare posibilă, ceea ce în multe cazuri nu se întâmplă.

Alternativa pe care o propunem acestei abordări tradiționale este învățarea prin întărire bazată pe diferența între estimările unor sări ale jocului în diferite momente. Vom încerca să proiectăm un agent inteligent care cunoscând doar regulile de bază ale jocului, să învețe să joace la un nivel expert, fără a fi supravegheat de un om. Ne-am oprit la jocul Tic-Tac-Toe, unul tipic pentru abordarea Minimax, şi vom încerca să arătăm că învățarea este intr-adevăr o alternativă viabilă, cu cerințe minime.

Capitolul I, prezintă pe scurt domeniul Inteligenței Artificiale ca o ramură a ştiinței aflată în plină expansiune. Se încearcă o definire a termenului de Inteligență Artificială şi se face o scurtă incursiune în istoria acestui domeniu din perioada antichității şi până în zilele noastre.

Capitolul II, prezintă aspecte legate de teoria jocurilor, şi o integrare a acesteia în Inteligența Artificială. Sunt clasificate jocurile după mai multe criterii şi este făcută o prezentare a evoluției lor.

Capitolul III, descrie metoda de rezolvare a jocurilor Minimax. Se vorbeşte despre reprezentarea jocurilor ca probleme de căutare şi un studiu al complexității acestora. Apoi sunt

prezentate proprietățile şi paşii algoritmului Minimax, dar şi îmbunătățiri care pot fi aduse: tăietura alfa-beta.

Capitolul IV, prezintă concepte ale învățării prin întărire mai exact a învățării bazate pe diferențe temporare. Este descris programul lui Gerald Tesauro TD-Gammon şi se încearcă o analiză a modului în care acesta învață.

Capitolul V, face o prezentare amănunțită a aplicației „X-O". Este făcută mai întâi o analiză a topicului şi descriere a avantajelor învățării față de abordarea tradițională, iar apoi se face o trecere în revistă a modului în care au fost implementate aceste metode. Ultima parte a capitolului este un manual de utilizare al aplicației.

1. Inteligența Artificială

1.1 Ce este Inteligența Artificială?

Omenirea s-a autodenumit homo sapiens – omul înțelept – deoarece capacitățile noastre mentale sunt atât de importante în viața noastră de zi cu zi. Domeniul Inteligenței Artificiale, îşi propune să înțeleagă entitățile inteligente, şi unul dintre motivele studierii acestui domeniu este acela de a învăța mai multe despre noi înşine. Inteligența Artificială este una dintre cele mai noi discipline, a fost inițiată oficial in 1956, când numele a fost inventat. Deşi la vremea respectivă fusese subapreciată, odată cu genetica modernă, a fost deseori numită de către oamenii de ştiință din alte domenii ca "disciplina în care mi-ar plăcea să lucrez".

Inteligența Artificială are o mare varietate de subdomenii, de la arii generale ca şi percepția, pana la teme specifice cum sunt jocul de şah, demonstrarea teoremelor matematice, scrierea de poezii si diagnosticarea bolilor. De multe ori, cercetători din alte domenii, folosesc Inteligența Artificială pentru a sistematiza şi automatiza teme intelectuale la care au lucrat toată viața lor. Similar, cercetătorii Inteligenței Artificiale pot sa aplice metodele lor fiecărei arii din intelectul uman. Aşadar, avem motive sa spunem că, într-adevăr Inteligența Artificială este un domeniu universal.

Tocmai datorită complexității si universalității acestui domeniu, este extrem de greu sa formulăm o definiție completă a Inteligenței Artificiale. Totuşi, de-a lungul anilor s-au încercat diferite definiții. Acestea variază de-a lungul a două dimensiuni principale: procese cognitive şi comportament. Aceste definiții pot fi grupate în 4 categorii:

1. Sisteme care gândesc ca şi oamenii: "Inteligența Artificială este studiul modului în care calculatoarele pot fi făcute să gândească – maşinile care gândesc - " (1).

2. Sisteme care gândesc rațional: "Inteligența Artificială este studiul modalităților de a face capabil un calculator să perceapă, să raționeze şi să acționeze inteligent" (2).

3. Sisteme care acționează ca oamenii: “Inteligența Artificială este studiul modului în care putem determina calculatoarele să facă lucruri la care, în prezent, omul este mai bun” (3).

   “Arta de a crea calculatoare capabile să efectueze acțiuni ce necesită inteligență atunci când sunt executate de oameni” (4).

4. Sisteme care acționează rațional: “Inteligența Artificială este ramură a științei calculatoarelor ce se ocupă cu automatizarea comportamentului inteligent” (5).

1.2 Comportamentul uman. Testul Turing

Putem spune că dacă nu ar exista calculatoarele, astăzi nu am discuta despre Inteligența Artificială, deci, originea acestei ramuri a informaticii se ascunde prin anii construcției primelor calculatoare electronice. Atunci a fost momentul în care omul şi-a pus problema despre cât de puternice pot fi aceste maşini noi, capabile să efectueze calcule complicate? Pot fi făcute acestea să gândească? Pot fi ele înzestrate cu inteligență umană? Rămâne de văzut cum pot fi realizate aceste manifestări şi ceea ce este mai important *“Cum ştim dacă am reuşit sau nu?”*.

Pornind de la întrebarea aceasta, şi încercând să o reformuleze în termeni oarecum obiectivi, direct constatabili, matematicianul englez Alan Turing a imaginat testul care îi poartă numele şi care, odată satisfăcut de o maşină, ar dovedi “inteligența” ei. Turing a plecat de la o idee foarte naturala, aceea că dacă nu putem să definim în termeni exacți inteligența, însă spunem despre om că este inteligent, atunci am putea să spunem şi despre o maşină că este inteligentă în cazul în care s-ar comporta ca o ființa umană. Rămâne de văzut care aspecte ale comportamentului uman sunt într-adevăr relevante pentru inteligență.

Testul Turing porneşte de la un joc creat tot de Turing – jocul imitației – cu trei jucători: o maşină (A) şi doi oamenii (B) şi (C). A şi B nu se găsesc în aceeaşi cameră cu C. C nu ştie care

dintre ceilalți doi jucători este maşina, şi nu poate să îi vadă sau să le vorbească direct. Comunicarea se poate face în scris sau printr-un terminal. Scopul lui C este să deosebească maşina de om, pe baza răspunsurilor la orice fel de întrebări. Dacă C nu reuşeşte, atunci maşina poate fi considerată inteligentă.

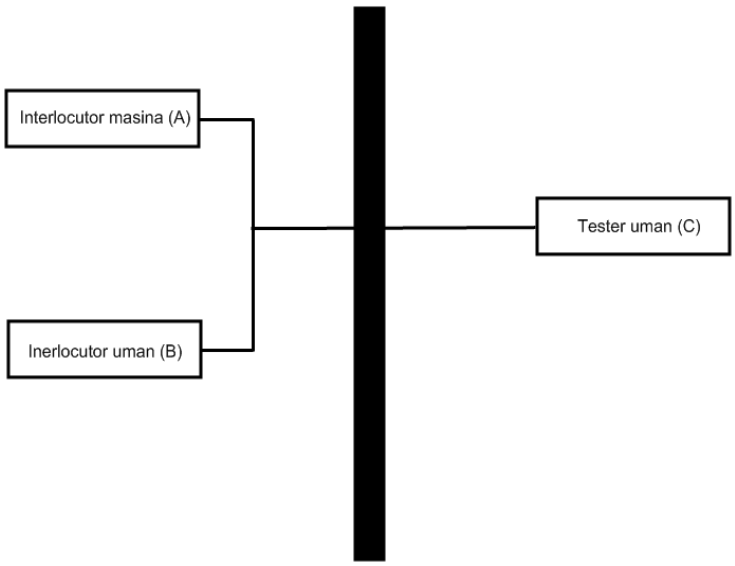


Figura 1-1 Testul Turing

La vremea respectivă nici un calculator nu a reuşit să treacă Testul Turing, si în 1950, anul în care a apărut primul articol despre acest test, autorul prezicea că în 50 de ani (aproximativ anul 2000), va fi posibil să existe o maşină care să joace jocul imitației atât de bine, încât şansa lui C de a identifica omul sa fie mai mică de 70%, după 5 minute de joc. Se pare că Turing s-a înşelat in privința acestei preziceri, mai mult, informaticienii din zilele noastre nu sunt la fel de optimiştii. Exista chiar, două tabere – unii care cred în posibilitatea de a satisface, cândva, Testul Turing, şi alții care sunt convinşi de contrariu.

Deşi a suscitat multe discuții de-a lungul anilor, Testul Turing reprezintă idealul pe termen lung al Inteligenței Artificiale ca ramura a informaticii. Turing considera că cel mai bun drum către realizarea unei maşini care să treacă testul său nu este programarea unui calculator dotat cu o mulțime fixa de cunoştințe, ci, mai degrabă, educarea unei maşini-copil, capabile să

învețe din experiență şi să folosească limbajul natural ca să-şi îmbogățească cunoştințele. Ea ar putea să-şi rezolve problemele proprii şi să-şi împlinească propriile sale planuri, dând dovadă de inteligență practică în viața de zi cu zi. De fapt, fiecare dintre aceste trăsături ideale s-au constituit în subdomenii ale inteligenței artificiale: învățarea automată, prelucrarea şi înțelegerea limbajului natural, achiziția de cunoştințe, construirea şi satisfacerea unor planuri (6).

1.3 Istoric al Inteligenței Artificiale

Deşi poate părea oarecum bizar, urme ale preocupărilor pentru inteligența artificială, se pot întâlni încă din cele mai vechi timpuri, când omul, încă nu descoperise ce înseamnă electricitatea, cu atât mai puțin putea sa-şi imagineze calculatoarele şi sistemele expert, care astăzi ne sunt la îndemână şi ne însoțesc la tot pasul.

1.3.1 Antichitatea

In Iliada lui Homer, in laboratorul lui Hefaistos, zeul focului, sunt prezentate primele „maşini inteligente". Acestea erau nişte dispozitive, care se roteau in jurul caierului de lână, ajutându-o astfel pe Afrodita, soția zeului, la tors.

Primul sistem expert, se consideră a fi un papirus egiptean, datând din mileniul III î.Hr., care consta in 48 de observații asupra diagnosticării şi tratării rănilor la cap. Cunoştințele erau organizate sub formă de reguli: dacă există un anumit simptom, atunci se recomandă un anumit tratament. Pe acelaşi principiu se bazează şi sistemele expert folosite astăzi în medicină, pentru depistarea cancerului şi a altor boli. Trebuie menționat că această realizare se bazează pe o altă invenție extraordinară a omului, scrisul, prin intermediul căruia obiectele fizice ale lumii reale pot fi reprezentate şi manipulate în mod simbolic (7).

O altă mare descoperire a antichității este logica silogistică a lui Aristotel, care datează din secolul IV î.Hr., şi este primul sistem formal de raționament deductiv. Astfel sunt consacrate cele trei "Principii ale gândirii":

- Al identității (A este A).
- Al noncontradicției (A nu poate fi simultan B şi non-B).
- Al terțului exclus (A este B sau non-B).

Aceste principii au avut o influență uriaşă asupra dezvoltării ştiințifice ulterioare a societății umane, deoarece noi cunoştințe puteau fi adăugate la fondul de cunoaştere existent prin demonstrarea de teoreme.

1.3.2 Renaşterea

În secolul al XVII-lea, Blaise Pascal, pune bazele primului calculator mecanic, capabil să efectueze adunări şi scăderi. Maşina a fost perfecționata apoi de Gottfried Wilhelm Leibnitz, astfel încât să realizeze înmulțiri şi împărțiri. Interesant că, acest calculator nu s-a bucurat de mare succes. Leibnitz a rămas cunoscut în special ca inventator al analizei matematice, deşi la vremea respectivă nu era la fel de celebru, fiind eclipsat de Newton.

1.3.3 Primele calculatoare

Primul calculator funcțional programabil, numit *Z3,* a fost produs în Germania, în 1941, de către Konrad Zuse. A fost distrus trei ani mai târziu în timpul celui de-al doilea război mondial si a fost reconstruit mai târziu ca piesă de muzeu. În 1948, la Universitatea Manchester din Marea Britanie s-a realizat *Mark I,* primul calculator valorificat din punct de vedere comercial. Acesta a fost şi primul calculator care a scris scrisori de dragoste. Atât Z3, cât şi Mark I se bazau pe tehnologia tubului vidat (a lămpii electronice).

1.3.4 Rețelele neuronale

În 1943, Warren McCulloch şi Walter Pitts au propus o arhitectură inteligentă bazată pe *rețele neuronale artificiale*, care încearcă să simuleze funcționarea creierului organismelor vii. Ca element distinctiv de logică simbolică explicită, în această abordare cunoştințele sunt înmagazinate de conexiunile dintre neuroni. Una dintre trăsăturile de baza ale acestor rețele neuronale, este capacitatea de a învăța.

Şase ani mai târziu, în 1949, Donald Hebb formulează o lege bazată pe învățarea neuronală: conexiunile dintre neuronii care se activează simultan se întăresc, iar cele dintre neuronii care se activează în contratimp se slăbesc. În 1958, Frank Rosenblatt propune un model practic al acestei paradigme, si anume *Perceptronul* (neuronul cu mai multe starturi).

1.3.5 Naşterea "Inteligenței Artificiale"

Momentul de naştere al Inteligenței Artificiale este legat de anul 1956, când la reuniunea de la Dartmouth College a fost prezentat programul de demonstrare a logicii propozițiilor "Logic Theorist" de către Newell, Shaw şi Simon. Acesta este considerat de mulți ca fiind primul program de Inteligență Artificială. Tot atunci s-au pus probleme legate de rezolvarea de jocuri, de activități care necesită raționament, de prelucrare a limbajului natural, de recunoaştere a modelelor ş.a. Deşi este chiar conferința la care s-a lansat termenul de *Inteligență Artificială*, la vremea ei, nu a fost considerată un succes. Totuşi în anii care au urmat au apărut tot mai multe laboratoare de cercetare în domeniul Inteligenței Artificiale.

În anul 1958, John McCarthy anunța limbajul LISP, destinat programării declarative. Acest limbaj, cu unele modificări se foloseşte şi azi, după 50 de ani.

1.3.6 Anii ‘ 60

Începutul anilor ‘60 aduce o noutate tehnologică, apariția *tranzistorului*, un evident progres, având în vedere că acesta înlocuia cam 40 de lămpi electronice. Tot în această perioadă apar şi primii roboți industriali.

În 1962, Lotfi Zahed (născut în Azerbaidjan) pune bazele *logicii fuzzy*, care contrazice principiul terțului exclus. Această teorie a fost întâmpinată cu scepticism şi chiar ostilitate, însă succesul aplicațiilor sale practice ulterioare este de necontestat.

Patru ani mai târziu, în 1966 Joseph Weizenbaum şi Kenneth Colby creează programul *ELIZA*, psihologul computerizat, care se va dovedi unul dintre cele mai celebre programe de Inteligență Artificială create vreodată. El conversează în limbaj natural, analizează frazele interlocutorului pentru a extrage cuvinte-cheie, şi apoi afişează o replică, dintr-o mulțime de variante predefinite. De cele mai multe ori conversația este una fluentă, iar unii „pacienți” chiar s-au ataşat emoțional de program, spre disperarea cercetătorilor.

1.3.7 Generația a treia de calculatoare

La sfârşitul anilor ’60 are loc un nou salt tehnologic, dând startul generației a treia de calculatoare, bazată pe *circuitele integrate* (chip-uri semiconductoare). De la inventarea acestora, numărul de tranzistoare de pe un chip s-a dublat la fiecare doi ani. În 1967, Richard Greenblatt inventează primul program de şah care învinge un jucător profesionist: MacHack.

În 1970, a luat ființa limbajul *Prolog*, ca rezultat al cercetărilor lui Alain Colmerauer, de la Universitatea din Marsilia, legate de formalizarea logică, pentru punerea bazelor unui limbaj de prelucrare a limbajului natural. Prolog este şi astăzi, cel mai utilizat limbaj de programare logică şi declarativă şi este intens utilizat în special la construirea sistemelor expert.

1.3.8 Microprocesoarele. Întoarcerea rețelelor neuronale

Începutul anilor ’70, aduce pe lângă, noțiunea de *algoritm genetic*, introdusă de John Holland, şi o nouă etapă tehnologică, cea a *microprocesoarelor*, care determină apariția generației a patra de calculatoare, care continuă până în prezent.

Cercetările în domeniul rețelelor neuronale sunt revigorate la mijlocul anilor ’80, când devine cunoscut algoritmul *delta generalizat* (back-propagation). Această metodă a fost propusă cu peste zece ani în urmă, însă a fost practic ignorată, deoarece presupunea un volum de calcule mult prea mare pentru perioada respectivă. Idee de baza este antrenarea percepronului multistrat, prin găsirea minimului funcției de eroare în raport cu ponderile conexiunilor.

1.3.9 Progrese recente

În anul 1986 vânzările de hardware şi software bazat pe Inteligență Artificială, în Statele Unite ale Americii au ajuns la 425 milioane de dolari. Sistemele expert erau foarte căutate datorită eficienței lor în diferite domenii. NASA propune un nou limbaj de programare declarativ numit *Clips.*

La începutul anilor ’90, Gerry Tesauro demonstrează că învățarea cu întărire este suficient de puternică, pentru a face ca un program care învață singur să învingă jucători profesionişti. Programul său, este un joc de table sugestiv denumit *TD-Gammon,* bazat pe teoria diferențelor temporale pentru calcularea recompenselor la diferite momente de timp. Asupra acestei idei fundamentale în teoria jocurilor, vom reveni într-unul din capitolele următoare.

1.4 Prezentul în Inteligența Artificială

În 1997, *Deep Blue,* un super-computer creat de firma IBM, îl învinge pe Garry Kasparov, campionul mondial de şah la acea vreme. Tot în acelaşi an, are loc primul campionat de fotbal cu roboți. Tehnici de Inteligență Artificială sunt folosite de programe de căutare pe Internet pentru extragerea de cunoştințe dar şi în domeniul militar, în operațiuni precum „Furtună în deşert".

În 2000, se construiesc deja animale de casă robotizate interactive, cu mare succes comercial, lideri în acest domeniu fiind firmele japoneze. Cynthia Breazeal coordonează proiectul *Kismet*, care încearcă construirea unui robot cu o față care exprimă sentimente. Se înregistrează un progres şi în sectorul maşinilor inteligente autonome. Robotul *Nomad* explorează regiuni îndepărtate din Antarctica în căutare de fragmente de meteoriți. Alți roboți transmit imagini de pe suprafața planetei Marte şi se cercetează posibilitățile de creştere a autonomiei vehiculelor spațiale (8).

2. Teoria jocurilor

2.1 Jocurile şi Inteligența Artificială

Jocurile sunt fascinante, iar scrierea de programe care să le joace este chiar şi mai fascinantă. Se poate spune că teoria jocurilor este pentru Inteligența Artificială cum este Grand Prix-ul pentru constructorii de motoare.

Dezvoltarea jocurilor şi a domeniului multimedia, în general, este în plină expansiune, o afacere de sute de milioane de dolari. La ora actuală nu se mai poate concepe un joc fără a avea în structură elemente de Inteligența Artificială. Implementată corect, aceasta garantează un produs bine vândut, deci profit şi satisfacție oferită jucătorilor (9).

Teoria jocurilor de strategie, poate fi, pe scurt, caracterizată ca aplicarea analizei matematice asupra modelelor abstracte de situații conflictuale. Primele astfel de modele în care a fost aplicată această teorie, au fost jocurile de societate ca şah, poker sau bridge. Apoi s-au aplicat aceleaşi modele în domenii ca economia, sociologia sau ştiințele politice. Teoria jocurilor este folosită, sau se află în strânsă legătură cu arii ca programare liniară, decizii statistice, managementul ştiinței, cercetare sau strategi militare, deşi, uneori corespondenta matematică a teoriei nu este valabilă.

2.2 Jocuri în forma extinsă şi jocuri în forma normalizată

Participanții la un joc sunt numiți jucători şi în general sunt 2, dar există şi jocuri cu mai mulți. Există de asemenea jocuri care implica elemente aleatoare numite şi şansă, care intervin atunci când se aruncă zarurile sau se extrage o carte. Toate aceste mutări sunt atribuite unui jucător „Şansă" – un jucător fictiv folosit doar pentru o bună abstractizare a jocului.

Spunem ca un joc este în *forma extinsă*, atunci când acesta este specificat de o listă de reguli. Pentru rezolvarea matematică a unui joc, avem însă nevoie de jocuri in *forma normalizată*. Pentru aceasta este nevoie sa cunoaştem o listă completă cu toate combinațiile de mutări legale posibile, pe care un jucător poate să le facă pentru fiecare situație care apare pe parcursul unui joc. Numărul acestor mutări poate fi astronomic, chiar şi pentru un joc simplu cum este Tic-Tac-Toe, deoarece aceasta listă de mutări este mult mai mare decât ceea ce noi numim *strategie*. Tocmai din acest motiv, aplicațiile teoriei jocurilor, chiar si pentru jocurile de societate, sunt mult limitate de dificultăți computaționale.

Să presupunem acum că avem un joc, cu *n* jucători, fără jucătorul Şansă, prezentat mai sus. Notăm cu $s_1, s_2, \ldots, s_n$ combinațiile de mutări posibile pentru toată durata jocului, specifice jucătorilor 1, 2, …, *n*. Fiind date acestea, jocul trebuie jucat intr-un mod unic si va rezulta un câştig unic. Fie $P_i(s_1, s_2, \ldots, s_n)$ valoarea câştigului jucătorului *i* pentru acest joc. Presupunem că $O_1, O_2, \ldots, O_k$ sunt venituri în urma unui eveniment aleator (de şansă), şi presupunem în continuare că probabilitățile de apariție a acestora sunt $p_1, p_2, \ldots, p_k$ cu $p_i > 0$ şi $p_1 + p_2 + \ldots + p_k = 1$. Probabilitatea matematică a evenimentului de şansă este suma $E = p_1O_1 + p_2O_2 + .. + p_kO_k$.

Dacă există aceste mutări aleatoare (de şansă) este uşor de intuit că un set de combinații de mutări posibile pentru fiecare jucător, nu va determina un venit unic al jocului, ci un set de posibile venituri. Aceste venituri se vor exclude reciproc, şi probabilitatea lor de apariție va depinde de mutările aleatoare asociate cu şansa lor de apariție. Prin urmare putem spune că $P_i(s_1, s_2, \ldots, s_n)$ este câştigul aşteptat pentru fiecare jucător $i = 1, 2, \ldots, n$.

Acum putem defini *forma normalizată* a jocului, ca lista tuturor câştigurilor aşteptate pentru fiecare jucător, pentru fiecare combinație posibilă de mutări legale, de-a lungul jocului. Pentru jocurile de 2 jucători, cea mai simplă metodă de stocare a acestora este folosind matrice bidimensionale (10).

2.3 Clasificarea jocurilor

După ce am vorbit mai sus despre jocuri în forma extinsă şi jocuri în forma normalizată, vom prezenta în continuare o clasificare a jocurilor în raport cu diverse criterii:

a. După modul în care jucătorii comunică intre ei:
   - *Jocuri cooperative*, în care jucătorii comunică intre ei înainte de luarea deciziilor şi pot face promisiuni, care vor fi respectate, înainte de alegerea strategiilor.
   - *Jocuri necooperative*. Jucătorii nu comunică intre ei înainte de luarea deciziilor.

b. În raport cu desfăşurarea în timp a jocurilor:
   - *Jocuri statice*; deciziile se iau simultan, după care jocul ia sfârşit.
   - *Jocuri dinamice*; deciziile jucătorilor sunt secvențiale, adică evoluează în timp.

c. După natura informației:
   - *Jocul în informație completă,* este tipul de joc în care toți jucătorii cunosc numărul celorlalți jucători, strategiile fiecăruia, funcțiile de câştig ale fiecăruia, precum şi regulile jocului.
   - *Jocul în informație incompletă,* este jocul în care cel puțin unul dintre jucători nu cunoaşte una sau mai multe funcții de câştig ale celorlalți jucători.

d. În raport cu tipul informației, în cazul jocurilor dinamice:
   - *Joc dinamic în informație perfectă*, este acel joc dinamic în care fiecare dintre jucători cunoaşte regulile, numărul jucătorilor, strategiile acestora, precum şi evoluția în timp a jocului (istoria jocului).

- *Joc dinamic în informație imperfectă,* este acel joc în care cel puțin unul dintre jucători nu cunoaşte istoria jocului, cunoscând însă celelalte elemente.

e. În raport cu structura câştigurilor:
   - *Jocul de sumă nula*, jocul în care suma câştigurilor este zero.
   - *Jocul de sumă nenulă*, jocul în care suma câştigurilor este diferită de zero.

f. După numărul de jucători:
   - *Jocuri cu doi jucători*
   - *Jocuri multi-persoană*
   - *Jocuri contra naturii* (11)

2.4 Strategie şi funcție de câştig

Numim *strategie* a unui jucător, o acțiune realizabila pe care jucătorul o poate alege în cadrul jocului. Mulțimea strategiilor tuturor jucătorilor formează mulțimea strategiilor jocului pe care o notăm: $S = S_1 \times S_2 \times \ldots \times S_n$, unde S_i este strategia fiecăruia din cei *n* jucători.

Funcție de câştig a jocului este funcția $u = (u_1, u_2, \ldots, u_n)$ formată din funcțiile de câştig ale fiecărui jucător. Dacă notăm funcția de câştig a fiecărui jucător cu u_i şi funcțiile de câştig ale celorlalți jucători cu u_{-i}, funcția de câştig a jocului va fi: $u : S \rightarrow \mathbf{R}$, $u = (u_i, u_{-i})$ (10).

2.4.1 Echilibrul Nash

Spunem că un joc merge spre un *Echilibru Nash* (nume preluat de la creatorul său John Nash), dacă strategiile ambilor jucători, îi vor duce pe aceştia la acelaşi câştig, ținând cont de strategia adversarului.

Echilibrul Nash mai poate fi definit şi ca o mulțime de strategii ($s_1^*, s_2^*, \ldots, s_n^*$) care respectă condiția:

$u_i (s_1^*, s_2^*, \ldots, s_i^*, \ldots, s_n^*) \geq u_i (s_1^*, s_2^*, \ldots, s_i^*, \ldots, s_n^*)$ pentru i = 1, n

sau

$u_i (s_i^*, s_{-i}^*) \geq u_i (s_i^*, s_{-i}^*)$ pentru i = 1, n (12).

2.5 Evoluția jocurilor

Teoria jocurilor este unul dintre primele domenii ale Inteligenței Artificiale, astfel prin 1769 apărea deja, prima fraudă odată cu apariția lui „Turk", un presupus joc de şah automat, despre care autorul spunea că ascunde un mic expert. Primele discuții serioase despre jocuri inteligente au avut loc prin 1846, când Charles Babbage propunea proiectarea unui motor analitic complet, care ar fi capabil să joace dame şi şah. Babbage a proiectat şi prima maşină care să joace Tic-Tac-Toe, însă nu a reuşit să o construiască. Prima maşină pentru jocuri a fost proiectată si realizată în 1890 de către Leonard Torres, şi era capabilă să joace sfârşitul unei partide de şah. Tot în aceeaşi perioadă, Ernest Zermelo venea cu ideea generării întregului arbore pentru jocul de şah, in vederea găsirii unei strategii perfecte.

Totuşi, aceste cazuri au fost relativ izolate, adevăratul început al jocurilor inteligente având loc la mijlocul secolului XX, odată cu apariția primelor calculatoare electronice programabile. Este perioada în care a fost prezentată teoria Minimax, de către von Neumann şi Morgenstern (1944). Pornind de la această idee s-au făcut mai multe încercări de realizare a unui program capabil să joace o partidă întreagă de şah, la un nivel satisfăcător.

În 1951, matematicianul englez Alan Turing, a scris primul program capabil să joace o partidă întreagă de şah. Publicat doar în 1953, programul nu a rulat pe un calculator, ci a fost testat printr-o simulare manuală împotriva unui oponent uman slab, care l-a învins. A urmat dezvoltarea căutării alfa-beta şi odată cu ea dezvoltarea de alte programe dintre care „MacHack6", primul care concura cu succes împotriva oamenilor. În 1982 Pearl a realizat o

analiză efectivă a factorului de ramificare şi complexităţii lui alfa-beta, arătând că este algoritmul de căutare optimal pentru jocuri.

De-a lungul anilor, programele pentru jocuri s-au îmbunătăţit gradual, ajungând ca în zilele noastre să concureze cu succes campionii mondiali. Cel mai elocvent exemplu este cel din 1997, în care campionul Garry Kasparov a fost întrecut de un super-computer, Deep Blue, creat de firma IBM. Obţinută în şase partide, această victorie a însemnat trecerea liniei Inteligenţei Artificiale dincolo de ce s-a realizat până în acel moment. Din studiile efectuate, a reieşit că în creierul uman se află aproximativ 100 miliarde de neuroni, fiecare capabil de 1000 de operaţii pe secundă. În jur de 30 de miliarde de neuroni formează "materia cenuşie", cea care gândeşte, restul de 70 de miliarde constituind "materia albă", cea care face legătura între neuronii din "materia cenuşie". În contrast, Deep Blue conţinea 480 de procesoare create special pentru jocul de şah, fiecare dintre ele fiind capabil să gândească aproximativ 2 milioane de poziţii pe secundă. Un computer actual poate realiza 10 la puterea 17 operaţii pe secundă (o operaţie în timpul în care lumina ar străbate un atom de hidrogen).

Este clar că viteza aparţine maşinii. Şi atunci ne punem întrebarea: “Ce atu mai are omul ?” Cel mai important pare a fi elementul surpriză, omul este imprevizibil, gândirea sa nu respectă întotdeauna un algoritm, aşa cum îl ştie maşina (9).

3. Metoda Minimax

3.1 Jocul ca problemă de căutare

Pentru a putea fi implementat, un joc, trebuie codificat aşa încât să fie pe înțelesul calculatorului. De aceea un joc poate fi gândit ca un arbore a tuturor stărilor viitoare ale jocului. Spre exemplu, în cazul jocului de şah, starea jocului poate fi definită ca aranjamentul pieselor pe tablă, precum şi cine este la rând să facă următoarea mutare

Starea inițială a jocului este rădăcina arborelui. În general, acest nod, are numeroşi fii, aceştia fiind toate mutările posibile, legale, pe care le poate face jucătorul al cărui rând este, şi aşa mai departe. Fiecare dintre aceste noduri reprezintă starea jocului după fiecare mutare a oponentului, şi are la rândul lui fii corespunzători celei de-a doua mutări a jucătorului curent şi aşa mai departe. Frunzele arborelui sunt stările finale ale jocului, din care nici o mutare nu mai poate fi făcută pentru că unul dintre jucători a câştigat, sau jocul s-a încheiat cu o remiză.

De fapt, arborele este un graf, deoarece pot fi mai multe mutări posibile dintr-o anumită stare a jocului, într-o altă stare particulară.

3.1.1 Componentele unei probleme de căutare

Un joc poate fi definit formal, ca o problemă de căutare cu următoarele componente:

- **Starea inițială,** include pozițiile pe tabla de joc şi cine este la mutare.
- **Mulțime de acțiuni,** care definesc mutările admise pe care le poate efectua un jucător.
- **Stare terminală,** care determină când se sfârşeşte jocul. Stările în care jocul se încheie se numesc **stări terminale.**

- **Funcție de utilitate,** care întoarce o valoare numerică pentru rezultatul jocului. De ex. în şah posibilitățile sunt 1 pentru victorie, 0 pentru egalitate şi -1 pentru înfrângere.

3.2 Algoritmul Minimax

Algoritmul Minimax este un algoritm de căutare într-un arbore. Acest algoritm urmăreşte selectarea celei mai bune mutări pentru calculator, într-un joc cu doi jucători. Mai întâi se construieşte arborele de joc cu toate mutările posibile pentru ambii jucători, după metoda descrisă mai sus.

Acest algoritm este denumit *Minimax,* deoarece calculatorul face mutarea care-i oferă câştigul maxim, în acelaşi timp asigurându-se că oponentul face mutarea cea mai defavorabilă calculatorului. Deoarece mutările alternează, şi algoritmul alternează maximizând şi minimizând nivelele arborelui în mod recursiv.

Întrucât, chiar şi în cazul unui joc simplu, numărul mutărilor posibile creşte exponențial, şi deoarece memoria unui calculator este limitată, de regulă, algoritmul Minimax face căutarea numai pe o adâncime fixă.

În continuare vom considera cazul unui joc simplu cu două persoane, pe care le vom numi sugestiv MAX şi MIN. Primul care mută primul este MAX, apoi mută pe rând, până la sfârşitul jocului, când unul este "premiat" iar celălalt penalizat.

MAX trebuie să găsească o strategie care să îl aducă la o stare terminală în care el este câştigătorul, indiferent de mutările pe care le face MIN. Strategia presupune că MAX face mutările corecte, indiferent de mutările lui MIN. Ideea este de a arăta cum se găseşte o strategie optimă, chiar dacă nu este timp suficient să o găsim (13).

3.2.1 Tic-Tac-Toe

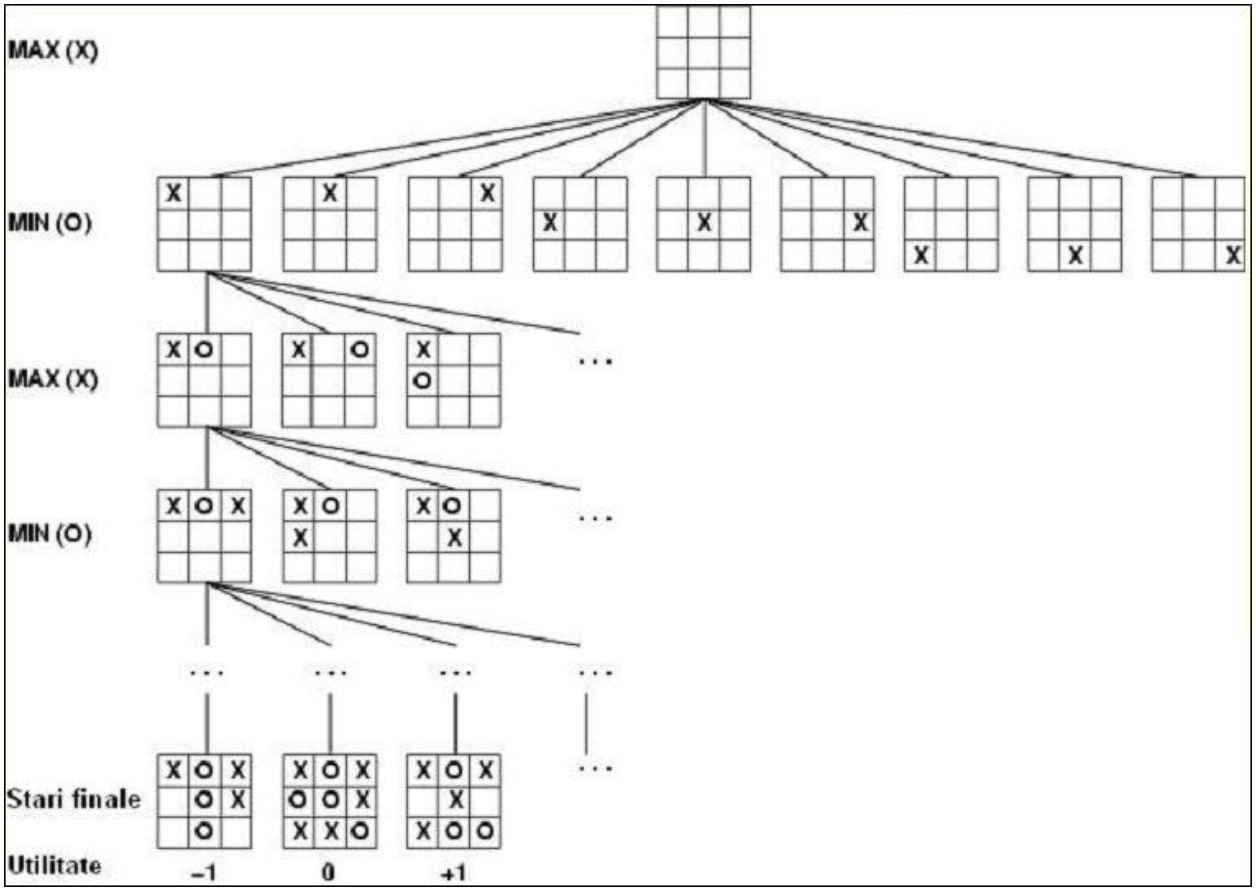


Figura 3-1 Arborele de căutare pentru Tic-Tac-Toe

În Figura 3-1 de mai sus este reprezentat jocul Tic-Tac-Toe(X şi 0), sub formă de arbore. De la starea inițială, MAX are posibilitatea de a alege din 9 mutări posibile. Jucătorii alternează punând X şi 0 până se ajunge la o stare terminală – stare în care un jucător are 3 elemente pe o linie, coloană sau diagonală, sau toate căsuțele au fost completate(egalitate). Numărul ataşat fiecărei frunze reprezintă utilitatea stării terminale din punctul de vedere a lui MAX. Valorile mari sunt considerate bune pentru MAX şi proste pentru MIN, reciproca fiind valabilă, de aici şi numele celor doi.

Sarcina lui MAX este să folosească acest arbore de căutare, şi ținând cont de utilitatea stărilor finale să determine cele mai bune mutări.

3.2.2 Paşii algoritmului Minimax

Algoritmul Minimax, care determină strategia optimă pentru jucătorul MAX, constă în 5 paşi:

1. Generează arborele de joc până la stările terminale
2. Aplică funcția de utilitate pentru fiecare stare terminală, pentru a îi determina valoarea.
3. Foloseşte utilitatea stărilor terminale pentru a determina utilitatea stărilor de la un nivel superior al arborelui.
4. Continuă evaluarea utilității nodurilor până ajunge la rădăcină.
5. Când se ajunge la rădăcină, MAX alege nodul de pe nivelul inferior cu valoarea cea mai mare.

În Figura 3-2 de mai jos este prezentat un exemplu pentru stabilirea utilității din pasul 3 al algoritmului. Am luat în calcul un joc care se termină după numai două mutări (una MAX şi una MIN). Pentru MAX mutările posibile sunt A1, A2 şi A3, iar pentru MIN: A11, A12 etc., iar valorile stărilor terminale sunt între 2 şi 14.

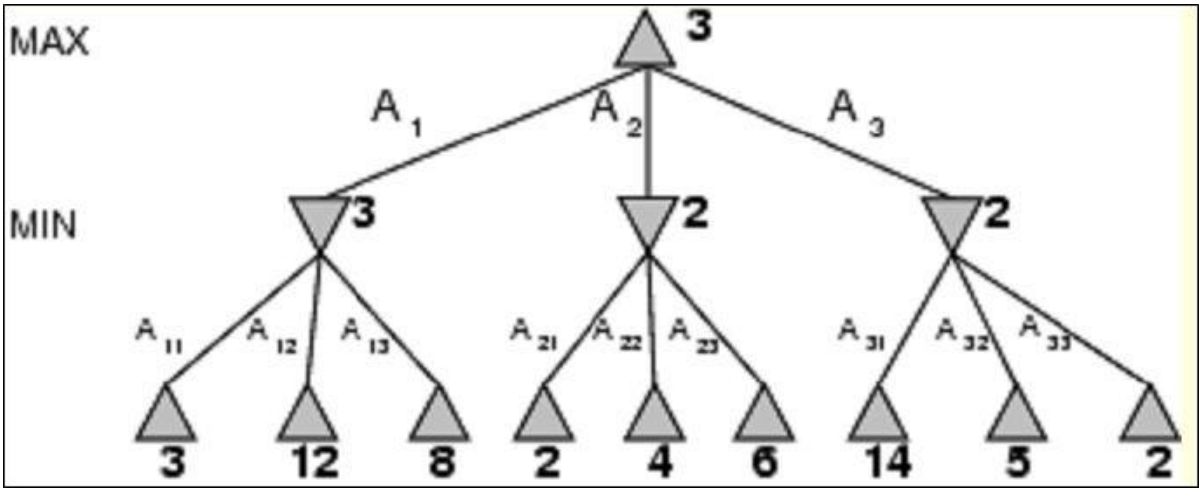


Figura 3-2 Aplicarea funcției de utilitate

Alegerea pe care o va face MIN pentru nodul de deasupra va fi valoarea cea mai mică, adică cea mai bună din punctul său de vedere. MAX va avea de ales între nodurile evaluate la 3, 2 şi 2, de unde evident va opta pentru valoarea maximă.

3.2.3 Proprietăți ale algoritmului Minimax

- Optimal ? Da (Împotriva unui adversar optimal).
- Complexitate temporara ? $O(b^m)$.
- Complexitate spațială ? O(bm).

  b – factorul de ramificare

  m – adâncimea arborelui

3.2.4 Funcția de evaluare

Este bine de ştiut că pentru un joc cum este şahul, factorul de ramificare este $b \approx 35$, iar adâncimea arborelui $m \approx 100$. Prin urmare algoritmul ar avea o complexitatea temporara uriaşa $O(35^{100})$, iar noi am plecat de la presupunerea că programul are timp să caute până la stările terminale, ceea ce este impractic.

Shannon propunea ca în loc să se meargă până la stările terminale şi să se folosească funcția de utilitate, căutarea să fie oprită mai devreme şi să se aplice o *funcție de evaluare* euristică noilor frunze ale arborelui.

Funcția de evaluare, întoarce o estimare a utilității aşteptate pentru joc dintr-o stare dată. Spre exemplu, în jocul de şah, această funcție se poate referi la valoarea materială a fiecărei piese: pion 1, nebun 3, tura 5, regina 9. Pe lângă acestea ar fi bine să ținem cont şi de alte aspecte cum ar fi aranjarea pionilor sau modul în care este protejat regele. Performanța unui program pentru jocuri depinde în mare măsură de funcția de evaluare aleasă. Dacă aceasta nu este bine definită, poate ghida programul spre stări care aparent sunt bune dar de fapt sunt dezastruoase.

Funcția de evaluare trebuie să corespundă cu funcția de utilitatea aplicată stărilor terminale şi în acelaşi timp nu trebuie să necesite foarte mult timp.

În concluzie, în elaborarea funcției de evaluare, trebuie făcut un compromis între acuratețea funcției şi costul de timp. De asemenea, această funcție trebuie să reflecte in mod foarte corect şansele reale de a câştiga.

Funcția de evaluare, bazată pe valoarea materială, în jocul de şah este liniară şi ponderată, şi poate fi dată ca:

$$Eval(s) = w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s)$$

Unde w_i este pondere, iar $f_i(s)$ este numărul de piese i, care sunt în plus față de adversar. Exemplu:

$w_2 = 5$ (pentru ture)

$f_2(s)$ = (numărul de ture ale lui MAX) – (numărul de ture ale lui MIN)

Cea mai simplă metodă de a controla cât de adânc să meargă căutarea este folosirea unei limite de căutare în adâncime. Reducerea se va face în acest caz la toate nodurile până la adâncimea *d*, ales în aşa fel încât să nu se depăşească timpul alocat pentru o mutare. O altă posibilitate, chiar mai bună, ar fi să considerăm căutarea iterativă în adâncime. Astfel programul porneşte o căutare completă în arbore, iar când nu mai este timp, întoarce mutarea cea mai bună până la adâncimea la care a ajuns (14).

3.3 Minimax cu tăietură alfa-beta

Marele handicap al algoritmului Minimax este complexitatea, care-l face să nu fie optim. Spre exemplu, pentru jocul de şah, un program bine scris poate căuta aproximativ 100 de poziții pe secundă. In turneele de şah, timpul pentru mutare este de 150 de secunde, deci programul ar putea căuta 150 000 de poziții. Cum factorul de ramificare este de 35, programul va fi capabil să

meargă în avans cu doar 3-4 mutări, ceea ce l-ar face să joace la un nivel de începător. Chiar şi un jucător mediu poate vedea 6-7 mutări înainte, ceea ce îl face pe program să fie uşor de bătut.

3.3.1 Descrierea algoritmului

Această problemă a complexității, prezentă mai sus, poate fi, din fericire, ajustată. Este posibil a se calcula decizia Minimax fără a vizita fiecare nod din arborele de căutare. Procesul constă în retezarea unor ramuri ale arborelui şi presupune neluarea în considerarea a acelor ramuri. Această tehnică de reducere a arborelui se numeşte **tăietură α – β.** Atunci când se aplică unui arbore, ea va întoarce aceeaşi mutare ca şi Minimax, însă elimină ramuri care nu pot influența decizia finală.

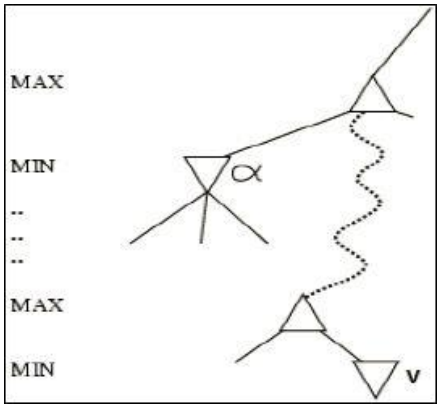


Figura 3-3 Tăietura alfa-beta

α este valoarea celei mai bune alegeri găsite până la momentul curent, la orice punct de-a lungul unui drum, pentru MAX. Daca *v* este mai prost decât *α*, MAX îl va evita, prin eliminarea acelei ramuri.

β este definit în mod similar pentru MIN, adică cea mai mică valoare găsită de-a lungul unui drum, pentru MIN.

Figurile 3-4, 3-5, 3-6, 3-7 şi 3-8 reprezintă o exemplificarea a modului în care tăietura α – β se aplică unui arbore.

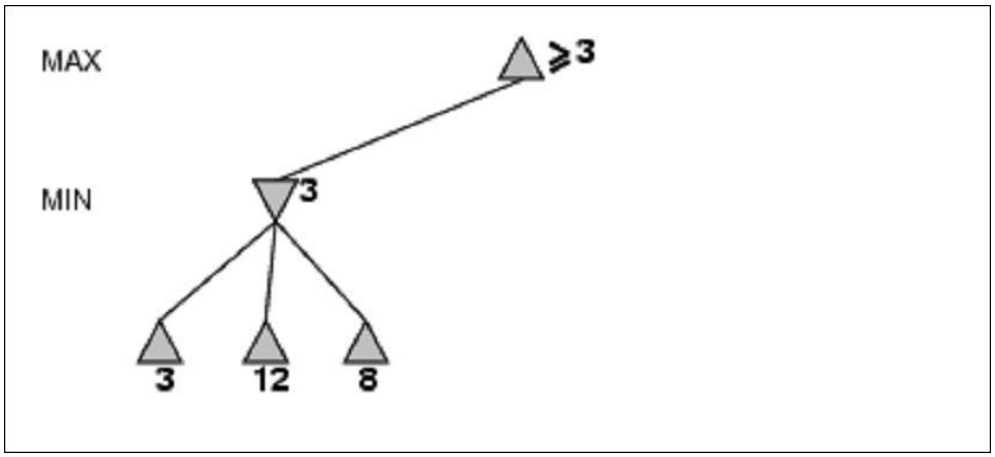


Figura 3-4 Aplicarea tăieturii alfa-beta

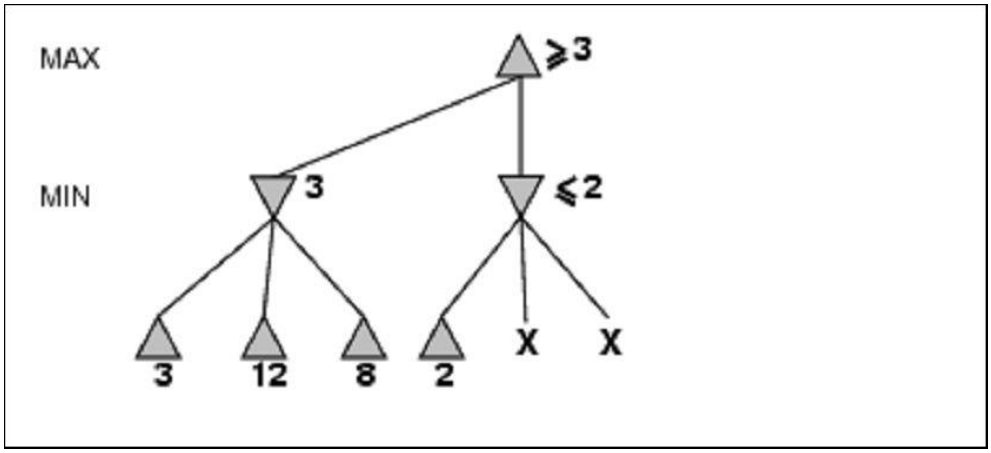


Figura 3-5 Aplicarea tăieturii alfa-beta

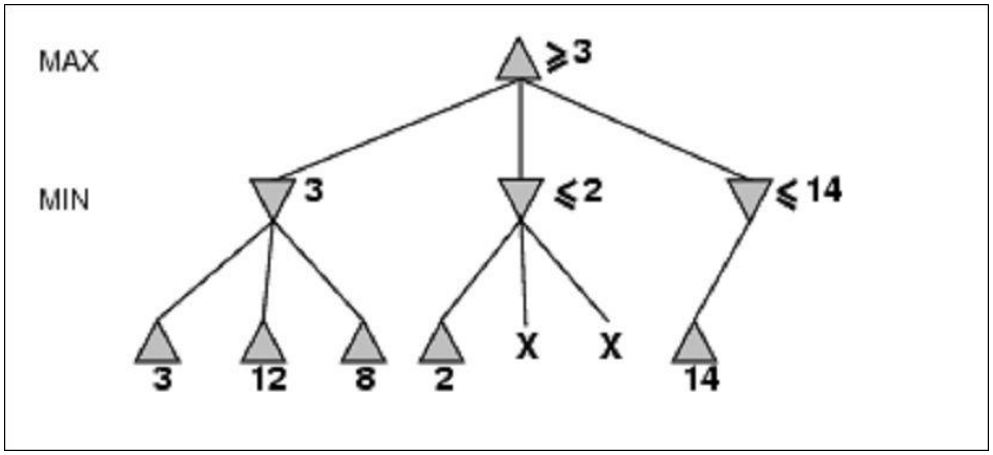


Figura 3-6 Aplicarea tăieturii alfa-beta

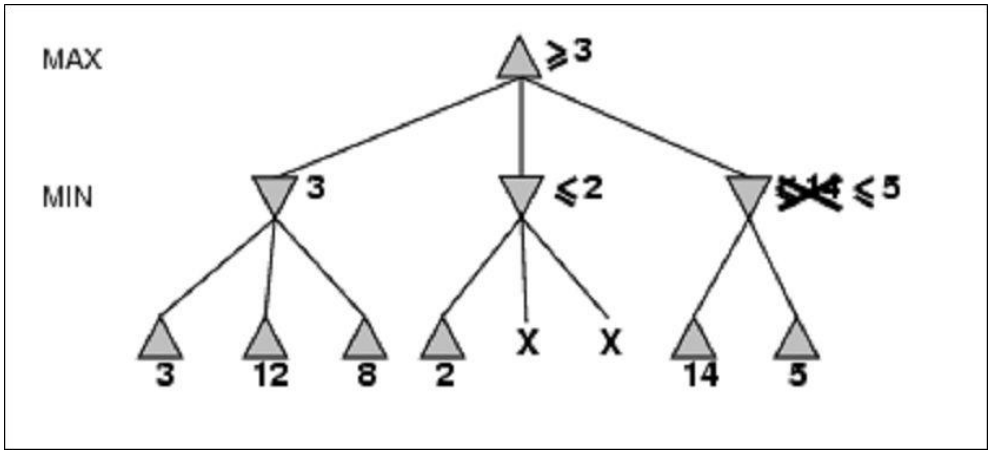


Figura 3-7 Aplicarea tăieturii alfa-beta

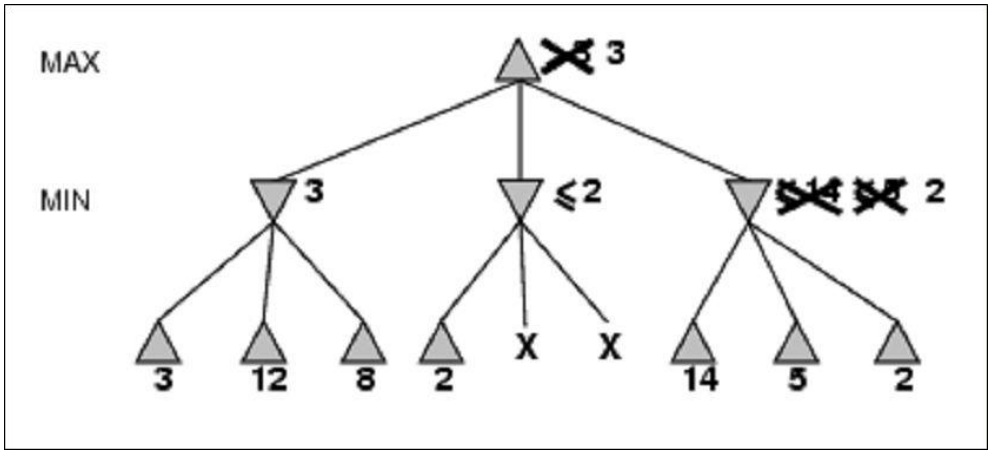


Figura 3-8

3.3.2 Proprietăți

În continuare vom prezenta câteva dintre trăsăturile de bază ale algoritmului Minimax cu tăietură $\alpha - \beta$:

- Reducerea nu afectează rezultatul final.
- O bună ordonare a mutărilor îmbunătățeşte algoritmul de reducere.
- Dacă succesori sunt puşi perfect în ordine (cei mai buni se află primii), atunci complexitatea temporară ar fi $O(b^{d/2})$, în loc de $O(b^d)$ cât are Minimax.
    - Deci $\alpha - \beta$, poate căuta de două ori mai mult decât Minimax.
    - Pentru jocul de şah, de exemplu, programul ar putea să se uite înainte 8 mutări în loc de 4.

- Tratând cu mare atenție calculele care afectează decizia, putem transforma un program de nivel începător în expert (14).

3.4 Alternative la Minimax

Minimax are câteva aspecte care, chiar cu îmbunătățirile descrise, sunt problematice. Spre exemplu, se bazează întotdeauna pe premisa că adversarul va selecta cea mai bună mutare. Această presupunere este acceptabil în situații de câştig, în care poate fi găsită o situație bună pentru noi. Dar în situațiile de înfrângere ar putea fi mai bine să ne asumăm riscul că adversarul va face o greşeală.

De exemplu, să presupunem că avem de ales între două mutări, ambele conducând la situații foarte proaste dacă adversarul joacă perfect, dar una dintre ele este mai puțin proastă decât cealaltă. Să presupunem în continuare, că mutarea mai puțin promițătoare ar putea conduce la o situație foarte bună pentru noi dacă adversarul ar face o singură greşeală. Deşi procedura Minimax ar selecta mutarea mai bună, ar trebui să alegem cealaltă mutare, care este probabil mai proastă, dar posibil mult mai bună. Pentru a putea lua astfel de decizii în cunoştință de cauză

trebuie să avem acces la modelul stilului de joc al adversarului, astfel încât să poată fi estimată posibilitatea apariției diferitelor greşeli. Dar o astfel de analiză este foarte greu de realizat (15).

4. Învățarea prin întărire

4.1 Introducere

Încă de la proiectul lui Shannon privind un algoritm pentru jocul de şah şi programul lui Samuel care învăța să joace dame, domeniul jocurilor pe table complexe de joc ca: GO, şah, dame, Othello sau table a fost pe scară largă privit ca un mediu ideal pentru testarea şi explorarea unei largi varietăți de concepte şi idei în Inteligența Artificială in general, şi în domeniul maşinilor auto-instruibile, în special. Aceste jocuri pe table ne oferă provocarea unei complexități uriaşe şi a unui rafinament desăvârşit necesar pentru a juca la un nivel de expert. În acelaşi timp, datele de intrare şi factorii de performanță sunt clar şi bine definite, iar mediul jocului este automatizat, astfel încât este uşor de simulat tabla de joc, regulile de joc şi determinarea câştigului, în momentul în care jocul se termină.

Putem spune că părintele *Învățării prin întărire(reinforcement learning)* este Gerald Tesauro care a inițiat această idee în programul numit TD-Gammon. Acest program are la bază o rețea neuronală care se autoinstruieşte să joace table, jucând împotriva ei înşişi şi învățând din rezultate. Deşi TD-Gammon şi-a surclasat toți adversarii în materie de table, nu acesta a fost motivul pentru care a fost dezvoltat. Tesauro a dorit, mai degrabă, să exploreze câteva noi idei şi abordări ale problemelor tradiționale, folosind învățarea prin întărire (16).

4.2 Concepte generale

Idee de bază a învățării prin întărire este următoarea: agentul care învață observă nişte date sau un model de intrare, şi produce un semnal de ieşire (numit şi acțiune sau semnal de control). Imediat, agentul primeşte de la mediu o recompensă sau un feed-back de întărire, care îi arată cât de bun sau cât de rău a fost semnalul de ieşire. Scopul învățării este de a genera cele mai bune acțiuni care duc la un câştig maxim. De multe ori această recompensă este întârziată, ea vine la sfârşitul unei lungi secvențe de intrări şi ieşiri. Astfel cel care învață, trebuie să figureze

modul în care distribuie câştigul şi penalizările pentru diferitele intrări, ieşirile conducând spre semnalul final de recompensă.

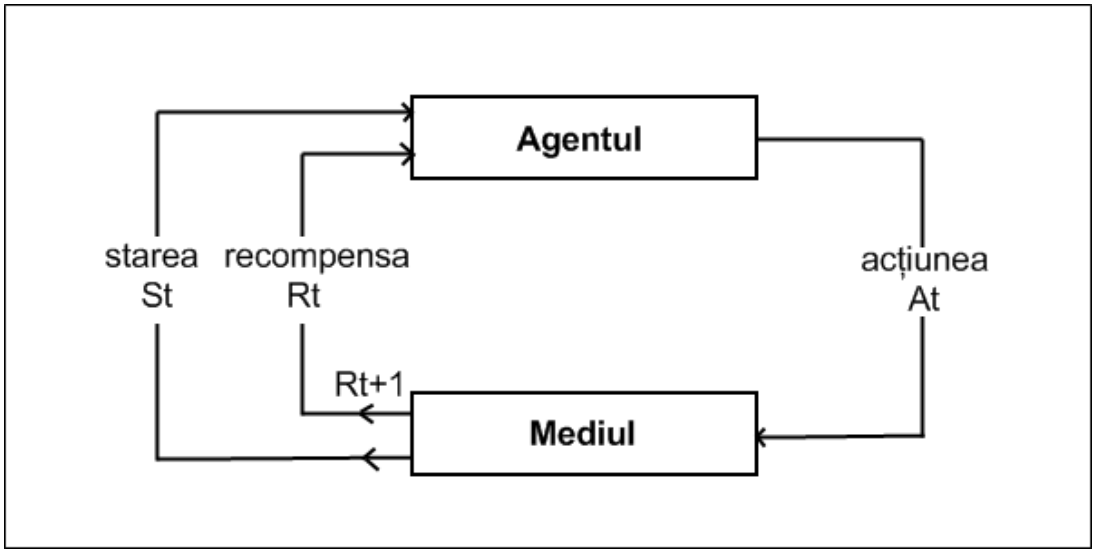


Figura 4-1 Interacțiunea dintre agent şi mediu

Învățarea prin întărire poate fi văzută ca un microcosmos pentru toate problemele de Inteligență Artificială. Un agent primeşte nişte percepții de la mediul în care se află, le marchează ca fiind utilități pozitive sau negative şi apoi se decide ce acțiune să execute (Figura 4-1).

La fiecare pas de timp *t*, agentul primeşte informații despre starea în care se află, $s_t \in S$, unde S este mulțimea stărilor posibile, şi pe baza acestei informații selectează acțiunea $a_t \in A(s_t)$ unde $A(s_t)$ este mulțimea de acțiuni disponibile din starea s_t. La următorul pas, în parte ca o consecință a acțiunii efectuate, agentul primeşte o recompensă numerică $r_{t+1} \in \mathbf{R}$, şi se află într-o nouă stare s_{t+1} (17).

Învățarea prin întărire a atras de-a lungul timpului un foarte mare interes datorită acestei paradigme a agentului inteligent care învață singur din propria experiență, fără ajutorul unui „profesor" inteligent. În contrast, pentru învățarea supervizată, este nevoie de un „profesor", care pentru fiecare set de date de intrare, sa-i explice agentului care este output-ul corect.

Din păcate, cu toată atenția care a fost acordată acestui tip de învățare de-a lungul anilor, până acum s-au realizat destul de puține succese practice în rezolvarea la scară largă a problemelor complexe din lumea reală. Una dintre deficiențe a fost aceea că în cazul învățării prin întărire cu întârziere, problema distribuirii câştigului temporal a rămas extrem de dificilă. O altă problemă ar fi aceea că cele mai multe abordării ale învățării prin întărire au fost limitate să învețe folosind tabele de căutare sau funcții liniare de evaluare, nici una dintre acestea ne fiind adecvate pentru cele mai multe clase de probleme din lumea reală.

4.2.1 Învățarea bazată pe Diferențe Temporare

În ultimii ani s-au inițiat mai multe concepte menite să întâmpine deficiențele descrise mai sus. Unul dintre acestea a fost prezentat de Sutton şi denumit „*Temporal Difference learning - bazată pe Diferențe Temporare"* (sau simplu TD). Ideea principală a metodei TD este învățarea pe baza diferenței între predicțiile temporale succesive. Cu alte cuvinte, scopul învățării este de a aduce predicția curentă a agentului, pentru datele de intrare curente, cât mai aproape de următoarea predicție, de la următorul pas.

Cea mai recentă astfel de metodă, este un algoritm conceput pentru a antrena rețele neuronale cu mai multe straturi, şi este denumit *TD(lambda)*. Acesta are la bază un semnal euristic de eroare pentru fiecare pas, calculat ca diferența dintre două predicții succesive, care deviază învățarea. Apoi, după ce o eroare de predicție a fost găsită la un anumit pas (de timp), se ia în calcul deprecierea exponențială pe care eroarea o produce în timp, şi estimările precedente, pentru stările precedente sunt corectate. Durata de timp a deprecierii exponențiale este ajustată de parametrul *lambda* (18).

4.3 TD-Gammon

Dacă în materie de şah, Deep Blue este considerat "naşul" maeştrilor, fiind primul program care a învins un campion mondial, în materie de table meritul îi revine programului *TD-*

Gammon. Prima versiune a acestui program a fost realizată în anul 1991 de către Gerald Tesauro, care câțiva ani mai târziu, a participat şi la proiectarea lui Deep Blue. TD-Gammon a fost conceput ca o cale de a explora capabilitatea unei rețele neuronale multistrat antrenată cu TD(lambda), de a învăța funcții complexe neliniare. A fost de asemenea conceput pentru a realiza o comparație detaliată între conceptul de învățare bazată pe diferențe temporare şi alternativa acestei idei: învățarea supervizată pe un set de exemple provenite de la maeştri .

4.3.1 Prezentarea jocului de table

Table este un joc antic, pentru doi jucători, care fascinează datorită faptului că implică şansa. Între două mutări fiecare jucător trebuie să arunce zarurile. Nu vom insista în cele ce urmează asupra detaliilor de joc, ci ne vom referi mai degrabă la problemele pe care le implică implementarea unui astfel de joc.

De-a lungul anilor s-a dovedit extrem de dificilă realizarea unui program care să joace table la nivel de expert. Dacă privim o reprezentare simplă doar a finalului de joc, este posibil să concepem un program care pe baza unei căutări să joace perfect. Oricum, această abordare nu poate fi adaptată la întregul joc, datorită numărului imens de stări posibile (aproximativ 10^{20}). Mai mult, forța brută a căutării în adâncime, care a dat rezultate foarte bune pentru jocuri ca şah, dame sau Othello, nu este mai este posibilă datorită factorului uriaş de ramificare, provocat de implicarea zarurilor. Pentru fiecare mutare, sunt 21 de combinații posibile de zaruri, fiecare dintre acestea putând fi jucată în aproximativ 20 de moduri, rezultând astfel o rată de ramificare de ordinul sutelor. Aceasta este mult mai mare decât la dame sau şah (în general pentru aceste jocuri factorul este 8-10 pentru dame şi 30-40 pentru şah), şi mult prea mare pentru a atinge o adâncime semnificativă chiar şi cu cele mai rapide super-computere existente.

În absența unei căutări în adâncime, programele de table trebuie să se bazeze pe o analiză a poziționării pieselor. Abordarea tipică a acestei probleme a constat în a lucra îndeaproape cu experți umani o lungă perioadă de timp, şi a încerca conceperea unei funcții de evaluare euristice care să imite cât se poate de bine poziționarea, cunoştințele şi judecata experților. Folosind această tehnică s-au obținut rezultate bune (programul Neurogammon al lui Tesauro), dar a

rămas şi un decalaj serios între modul de joc al celor mai buni oameni, şi abilitatea programatorilor de a încapsula judecata umană într-o funcție de evaluare.

O altă problemă este faptul că expertiza umană care este emulată, nu este infailibilă. În jocuri ca Othello sau table, în ultimii 20 de ani, s-a produs o schimbare substanțială asupra modului în care experții evaluează pozițiile. Referitor la multe convingeri ale trecutului, despre care experții aveau o părere unanimă, se crede astăzi că sunt cu siguranță greşite, şi ținând cont de aceasta, programatorii nu sunt ferm convinşi să accepte nici actualele păreri.

4.3.2 Modul de învățare

TD-Gammon reprezintă o abordare radical diferită asupra implementării unui program capabil de o sofisticată analiză a poziționării pieselor. Contrar încercării de a imita oamenii, TD-Gammon foloseşte propriu mod de joc, învățând din experiențele jocului împotriva lui înşişi. Deşi pare un evident handicap, lipsa tutelajului unui maestru uman, îi oferă libertate, în sensul că nu este influențat de anumite prejudecați umane, care pot fi greşite sau nesigure.

La baza lui TD-Gammon este o *rețea neuronală*, ilustrată în Figura 4-2, organizată ca o arhitectură standard de *perceptron multistrat (MLP)* (19). Această arhitectură, folosită de asemenea în algoritmul de propagare înapoi (backpropagation) pentru învățarea supervizată, poate fi privită ca un aproximator generic de funcții neliniare. Fiecare neuron din stratul de neuronal de intrare are o valoare de activare care reprezintă un stimul extern. Acesta trimite valoarea de activare fiecărui neuron din stratul ascuns cu care este conectat. Fiecare neuron al stratului ascuns calculează propria sa valoare de activare, depinzând de valorile de activare pe care le primeşte de la neuronii de intrare. Acest semnal este apoi trimis neuronilor din stratul de ieşire sau următorului strat ascuns, de neuroni, în cazul în care există mai multe straturi ascunse. Aceşti neuroni din stratul ascuns calculează valorile de activare în acelaşi fel şi le trimit mai departe vecinilor. În cele din urmă semnalul provenit de la neuronii de intrare se propagă prin rețea pentru a determina valorile de activare pentru toți neuronii de ieşire (17).

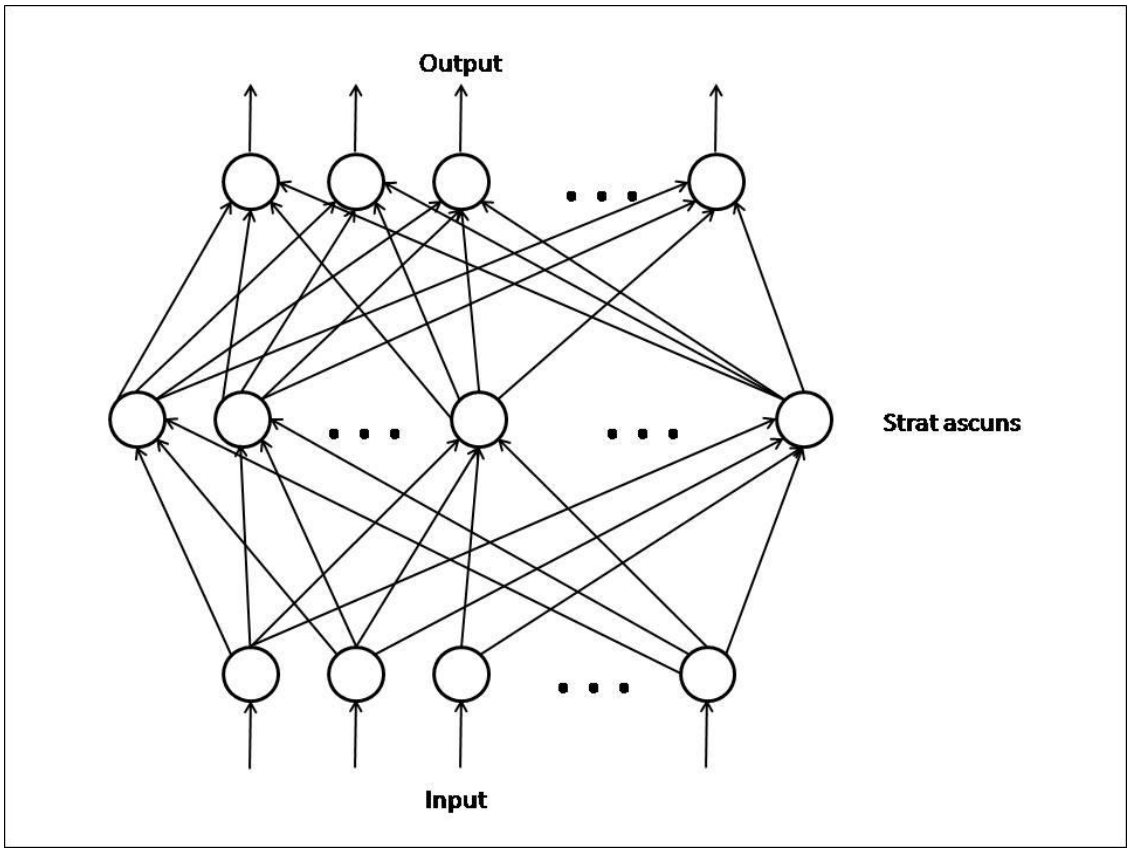


Figura 4-2 Structura rețelei neuronale cu trei straturi

Procedura de învățare pentru TD-Gammon este următoarea: rețeaua observă o secvență de poziții ale tablei de joc, începând cu poziția de start şi terminând cu poziția finală, în care unul dintre jucători şi-a "scos" toate piesele. Pozițiile sunt reprezentate ca vectori $x[1]$, $x[2]$, …, $x[f]$, date de intrare pentru rețeaua neuronală. La fiecare pas de timp din secvență corespunde unei mutări făcute de unul dintre jucători. Pentru fiecare vector de intrare $x[t]$ există un vector de ieşire $Y[t]$ care reprezintă o estimare a calității acestuia. Pentru acest joc, $Y[t]$ este un vector cu 4 componente corespunzător celor 4 posibile rezultate: câştigă Albul, câştigă Negrul, victorie normală sau marț.

La fiecare pas de timp este aplicat algoritmul TD(lambda) pentru a schimba ponderile rețelei, după următoarea formulă:

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

Alfa este o constantă cu valoare mică numită de obicei "coeficient de învățare", *w* este vectorul ponderilor care parametrizează rețeaua şi $\nabla_w Y_k$ este gradientul ieşirii rețelei.

Lambda este un parametru euristic menit să controleze cât de departe să meargă recalcularea estimărilor precedente, în urma găsirii unei erori la un anumit pas de timp. La sfârşitul fiecărui joc, se dă o recompensă finală *z* (în forma descrisă mai sus). Ecuația precedentă este folosită pentru schimbarea ponderilor, excepție făcând diferența ($z - Y[f]$) în locul diferenței ($Y[t+1] - Y[t]$) (16).

4.3.3 Rezultate

În primele experimente, datele de intrare introduse cuprindeau numai informații despre tabla de joc (numărul pieselor albe şi negre din fiecare locație) şi nu conțineau nici o informație adițională despre modul de joc cum ar fi tăria unui blocaj sau probabilitatea de a fi "scos".

În timpul antrenării, rețeaua neuronală este folosită pentru a selecta mutări pentru ambele părți (Alb şi Negru). La fiecare pas de-a lungul unei partide, rețeaua încearcă fiecare mutare posibilă. Va alege mutarea care aduce cel mai mare câştig pentru partea care trebuie să facă mutarea. Cu alte cuvinte, rețeaua învață din rezultatele jocului împotriva ei înşişi. Această idee este folosită chiar şi la începutul jocului când ponderile rețelei sunt luate aleator şi chiar şi strategia folosită este una aleatoare. Inițial această metodă pare să nu producă nici cea mai sensibilă învățare, pentru că strategia aleatoare este una extrem de rea, şi jocurile durează extrem de mult: cu mutări întâmplătoare de ambele părți, partidele au nevoie zeci sau chiar sute de mutări. În contrast, în mod normal o partidă are maximum 50-60 de mutări.

De-a lungul primelor câteva mii de jocuri de antrenament, rețeaua a învățat câteva strategii şi tactici elementare ca "scoaterea" adversarului sau practicarea unui joc sigur. Strategii

mai sofisticate au apărut după câteva zeci de mii de jocuri. Încurajator a fost faptul că odată cu creşterea numărului de jocuri de antrenament se îmbunătăţeşte si performanţa.

Cele mai bune rezultate au fost obținute cu o rețea neuronală cu 40 de neuroni în startul ascuns, care a fost antrenată 200 000 de partide. În aceste programul era capabil să joace foarte bine la un nivel intermediar, însă foarte interesant s-a dovedit a fi organizarea spațială a ponderilor rețelei neuronale cu valori pozitive şi negative (16).

Această primă versiune TD-Gammon 1.0 apărută în 1991, s-a dovedit a fi cel mai bun program de table conceput până la aceea dată, mai mult, era capabil să surclaseze toate programele concurente. Programul a mai fost îmbunătăţit în următorii ani, prin creşterea numărului de jocuri de învățare şi adăugarea unor strategii auxiliare, ajungând sa joace de la egal la egal cu maeştri umani.

Programul	**Jocuri de antrenament**	**Adversari**	**Rezultate**
TDG 1.0	300 000	Robertie, Davis, Magriel	+13 puncte / 51de jocuri (-0.25 ppj)
TDG 2.0	800 000	Goulding, Woolsey, Snellings, Russell, Sylvester	-7 puncte / 38 de jocuri (-0.18 ppj)
TDG 2.1	1 500 000	Robertie	-1 punct / 40 de jocuri (-0.02 ppj)

Tabel 4-1

4.4 Concluzie

Învățarea prin întărire este o nouă exprimare a problemelor de Inteligență Artificială. Un agent într-un mediu primeşte semnale, le ataşează unor utilități pozitive sau negative şi apoi se decide ce acțiune să aleagă. Sarcina de învățare poate varia în funcție de anumite situații:

- Mediul care poate fi accesibil şi inaccesibil.

- Agentul poate începe cu o anumită cunoaştere despre mediu sau va trebui să învețe singur.
- Recompensele pot fi recepționate doar în stări finale sau şi în stări intermediare.
- Recompensele pot fi valori numerice ale funcției de utilitatea sau pot fi doar sugestii („mutare bună”, „mutare rea”).
- Agentul poate să fie *pasiv*(doar observă mediul şi învață utilitatea stărilor) sau *activ*(acționează conform cu informației ce a învățat-o şi poate sugera explorarea unor părți necunoscute din mediu).

Sunt multe aplicații care aplică învățarea prin întărire, din afara domeniului jocurilor, cum ar fi controlul roboților sau strategii financiare. Totuşi pentru acest tip de aplicații pierdem un mare avantaj pe care îl avem în jocuri şi anume posibilitatea de a genera şi simula întregul mediu (17).

5. Aplicația „X – O”

5.1 Analiza

5.1.1 Importanța subiectului

Abilitatea de a juca jocuri este foarte importantă pentru managementul sistemelor informatice. Învățarea prin întărire te învață ce să faci – cum să asociezi situații spre acțiuni – astfel încât să maximizezi o recompensă numerică. “Elevului” nu i se spune ce acțiune să facă, precum în cele mai multe forme de instruire a maşinilor, dar în schimb, el trebuie să descopere care acțiuni îi aduc cea mai mare recompensă, încercându-le pe toate (18).

5.1.2 Jocul Tic-Tac-Toe

Aplicația „X – O”, propune o ilustrare a ideii generale de învățare prin întărire aplicată teoriei jocurilor, în contrast cu abordarea tradiționala: Minimax.

Pentru aceasta am folosit cunoscutul joc Tic-Tac-Toe (cunoscut si ca „X – O”). Doi jucători mută pe rând folosind o tabla de joc de dimensiune 3 x 3. Unul dintre ei joacă cu „X”, iar celălalt cu „O”, până când unul dintre ei câştigă, reuşind să plaseze trei semne de-ale sale pe o linie, coloană sau diagonală. Dacă nici un jucător nu reuşeşte aceasta şi toată tabla este completată, jocul se încheie la egalitate.

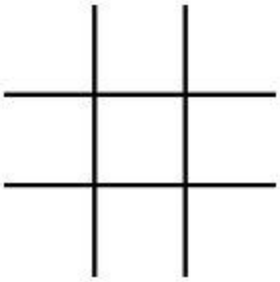


Figura 5-1 Tabla de joc

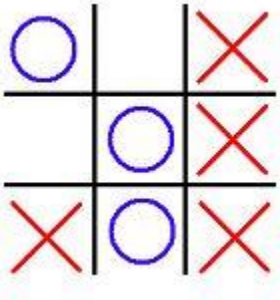


Figura 5-2 X a câştigat

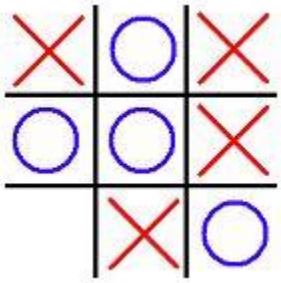


Figura 5-3 Remiză

5.1.2 Tic-Tac-Toe, o altă abordare

Un bun jucător va reuşi mereu să nu piardă la Tic-Tac-Toe, dar să presupunem că avem de a face cu un jucător imperfect, care uneori joacă greşit, permițându-ne astfel să câştigăm. Pentru moment să considerăm că victoriile şi remizele sunt la fel de rele pentru noi. Cum am putea construi un jucător care să găsească imperfecțiunile din jocul adversarului său şi astfel să învețe să-şi maximizeze şansele de a câştiga ?

În ciuda faptului că exemplul de față este o problemă foarte simplă, ea nu poate fi cu adevărat rezolvată într-un mod satisfăcător, folosind tehnicile tradiționale. De exemplu, clasicul Minimax, nu este corect în această situație, deoarece el presupune un anumit mod de joc din partea adversarului. De exemplu, un jucător Minimax, nu va atinge niciodată o stare a jocului din care ar putea să piardă, chiar dacă, de fapt, el câştigă mereu din acea stare, datorită jocului incorect al adversarului. Optimizările clasice pentru problemele cu decizii secvențiale, precum programarea dinamică, pot calcula o soluție optimă pentru orice adversar, dar pentru aceasta trebuie să cunoască o specificație completă despre adversar, incluzând probabilitatea cu care acesta face fiecare mutare din orice stare a tablei de joc. Presupunem în continuare că nu deținem aceste informații sau că nu sunt o prioritate pentru noi, aşa cum nu sunt o prioritate pentru majoritatea problemelor practice. Dar pe de altă parte, acest tip de informații pot fi obținute din experiență, în acest caz jucând multe partide împotriva adversarului din care sa-i învățăm comportamentul.

O abordare evolutivă a acestei probleme se poate baza pe căutarea în spațiul tacticilor posibile, a uneia cu probabilitatea cea mai mare de a câştiga împotriva adversarului. În acest context, tactica este o regulă care îi spune jucătorului ce mutare să facă pentru fiecare stare a jocului. Pentru fiecare tactică analizată, o estimare a şanselor ei de victorie, poate fi obținută jucând un număr de partide împotriva adversarului. Această evaluare, ne va arăta apoi care tactică sau care tactici să fie analizate în continuare. Pentru a îmbunătății tactica, se pot aplica sute de metode de optimizare cum ar fi “Hill-climbing” sau un algoritmi genetici. Căutând direct în spațiul tacticilor, înseamnă că toate tacticile sunt propuse şi comparate pe baza unor evaluări scalare.

În continuare vom prezenta cum problema Tic-Tac-Toe poate fi abordată folosind învățarea prin întărire şi funcții de aproximare. Fiecărei stări posibile în care poate ajunge jocul, îi este atribuită o valoare reprezentând probabilitatea de a câştiga din acea poziție. Starea A este considerată mai bună decât starea B, dacă probabilitatea de a câştiga din A este mai mare decât cea din B. Presupunând că jucăm cu „X", atunci pentru toate stările în care acest semn apare de trei ori pe o linie, coloană sau diagonală (Figura 5-2), probabilitatea de a câştiga este 1, pentru că am câştigat deja. Similar pentru stările în care „O" apare de trei ori pe o linie, coloană sau diagonală probabilitatea de a câştiga este 0, pentru că nu mai putem câştiga din acea stare. Pentru început vom seta valorile inițiale ale stărilor cu 0.5(sau o valoare aleatoare), aceasta însemnând că estimarea de a câştiga din fiecare stare este de 50 %.

Aşadar, procesul de învățare se bazează pe recompensele primite la finalul fiecărui joc de antrenament. Pentru a selecta mutările pe care să le facem, vom examina fiecare dintre stările în care putem ajunge, pentru fiecare dintre mutările noastre posibile. În cele mai multe cazuri vom muta după metoda „greedy", adică selectând mutarea care ne duce în starea cu ceam mai mare probabilitate de câştig. Uneori, vom selecta mutări în mod aleatoriu; acestea se numesc *mutări de explorare* pentru că le folosim pentru a ajunge în stări în care, poate, altfel nu am ajunge niciodată.

În Figura 5-4 este reprezentată o succesiune de mutări. Liniile continue reprezintă mutările făcute în timpul jocului; liniile întrerupte reprezintă mutările pe care le-am analizat însă nu le-am făcut. Cea de-a doua mutare a noastră din figură este o mutare de explorare, pentru că a fost făcută deşi exista o mutare mai bună care ne-ar fi dus în starea $\mathbf{e}^*$, cotată mai bine. Mutările de explorare nu afectează învățarea aşa cum fac celelalte mutări, producând backup-uri precum cele reprezentate de liniile curbe în figură şi descrise în continuare.

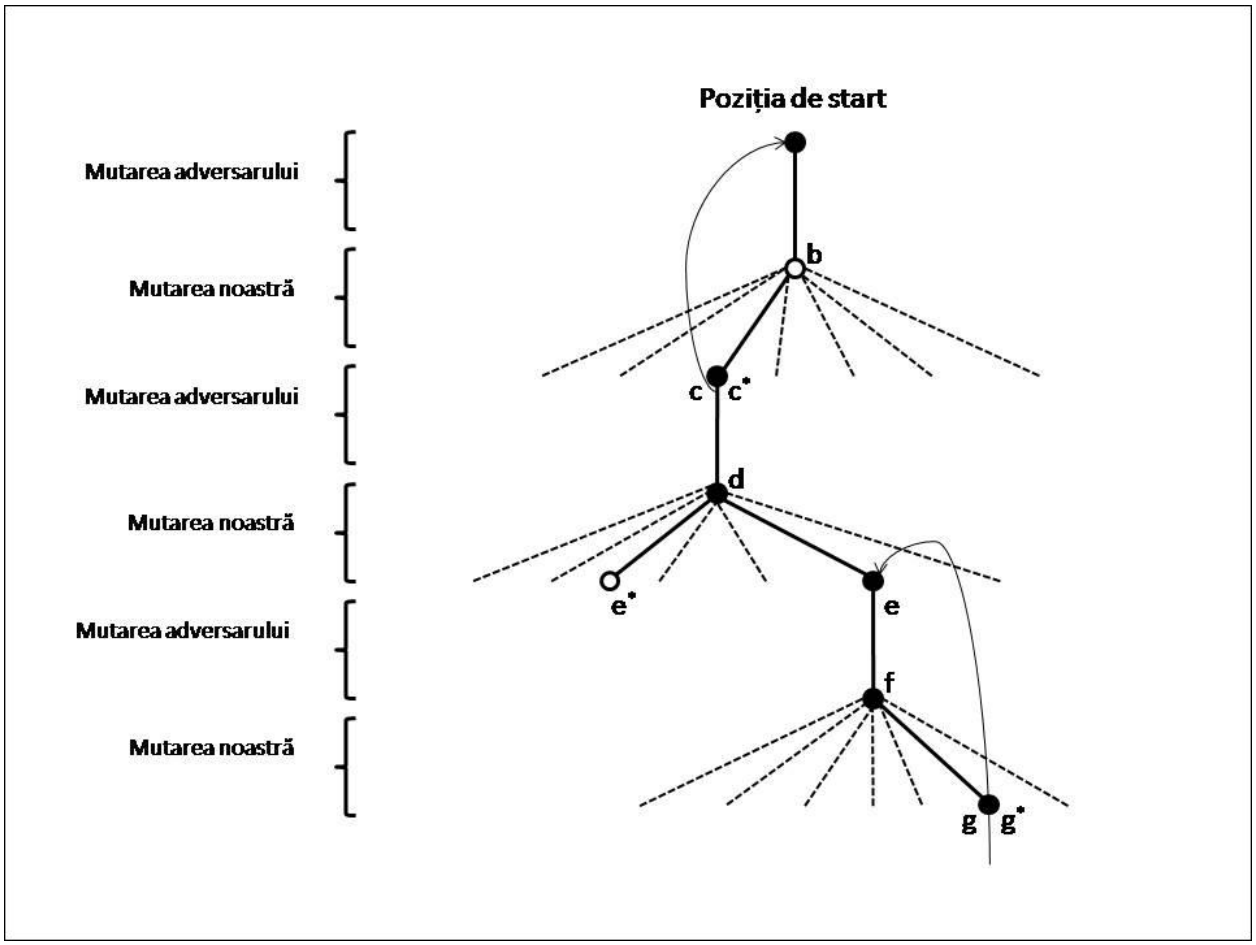


Figura 5-4 Secvență de mutări Tic-Tac-Toe

În timp ce jucăm, schimbăm valorile stărilor în care ne aflăm, încercând să facem nişte estimări cât mai precise asupra şanselor de câştig din acele poziții. Pentru aceasta, facem un "back up" al valorii stării după fiecare mutare (mai puțin cele de explorare), asupra stării precedente ca în Figura 5-4. Mai precis, valoarea curentă a stării precedente este ajustată să fie cât mai apropiată de valoarea stării următoare. Această operație se poate realiza schimbând valoarea stării anterioare cu o fracție din distanța spre starea curentă, ca în formula următoare:

$$\mathbf{P(s)} \leftarrow \mathbf{P(s)} + \boldsymbol{\alpha}[\mathbf{P(s')} - \mathbf{P(s)}]$$

unde **s'** este starea curentă, **s** este starea precedentă, **P(s)** este estimarea valorii stării **s**, iar $\boldsymbol{\alpha}$ este coeficientul de învățare.

Această regulă de „update” este un exemplu de învățare bazată pe diferențe temporare (4.2.1), pentru că schimbările se bazează pe diferența $[\mathbf{P}(\mathbf{s}') - \mathbf{P}(\mathbf{s})]$ dintre estimări făcute în două momente de timp diferite. Dacă coeficientul de învățare este redus corespunzător pe parcursul jocurilor, metoda converge spre valorile corecte ale fiecărei stări, producând astfel un joc optim din partea jucătorului nostru (18).

5.1.3 Reprezentarea învățării

În elaborarea acestei aplicații ne-am bazat pe ideile prezentate mai sus îmbinate cu o rețea neuronală după modelul lui Tesauro [] prezentat în (4.3.2). Această rețea neuronală folosește o arhitectură standard de perceptron cu straturi multiple, şi poate fi privită ca un generator de aproximări pentru funcții neliniare.

Datele de intrare pentru rețeaua neuronală sunt de forma unui vector cu 6 componente după modelul:

1. Numărul de linii, coloane sau diagonale pe care jucătorul are 1 X şi poate câştiga
2. Numărul de linii, coloane sau diagonale pe care jucătorul are 2 X şi poate câştiga
3. Numărul de linii, coloane sau diagonale pe care jucătorul are 3 X
4. - Numărul de linii, coloane sau diagonale pe care adversarul are 1 O şi poate câştiga
5. - Numărul de linii, coloane sau diagonale pe care adversarul are 2 O şi poate câştiga
6. - Numărul de linii, coloane sau diagonale pe care adversarul are 3 O

În timpul antrenamentului, rețeaua primeşte câte o succesiune de astfel de input-uri fiecare reprezentând câte o stare a jocului începând cu starea de start şi terminând cu cea finală când una dintre părți câştigă sau jocul se termină la egalitate. Rețeaua returnează un output care reprezintă o estimare a valorii input-ului primit. În funcție de rezultat se primeşte o recompensă: 1 pentru victorie, 0 pentru remiză şi -1 pentru înfrângere. Această recompensă este mai apoi redistribuită pentru a schimba valorile ponderilor dintre straturile rețelei (la începutul învățării au fost inițializate cu valori mici aleatoare). Modificarea ponderilor se face după fiecare pas prin aplicarea algoritmului TD(lambda), descris în 4.2.1, după formula:

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

Pentru jocurile de antrenament rețeaua neuronală joacă împotriva ei înşişi. Pentru o mutare joacă din perspectiva lui X, iar pentru următoarea din perspectiva lui O. În timpul acestor partide 10% dintre mutări sunt de explorare, iar coeficientul de învățare este setat la valoarea inițială α = 0.1.

După câteva mii de jocuri de antrenament, rețeaua ajunge la un nivel ridicat de joc, ajungând să fie învinsă numai în situații izolate. Este de asemenea interesant de văzut cum decurge învățarea după un număr mai mic de jocuri sau schimbând coeficientul de învățare.

5.2 Implementarea

5.2.1 Mediul de programare

Aplicația „X-O" este realizată în limbajul de programare **C#** şi platforma **.Net 2.0.** Interfața grafică a fost realizată folosind componentele grafice ale platformei, direct din namespace-ul *System.Windows.Forms.*

Tabla de joc este formată din 9 "căsuțe", fiecare dintre ele fiind implementată folosind clasa *Pătrat,* care moşteneşte *System.Windows.Forms.UserControl.* Metoda principală a acestei clase este *protected override void OnPaint(PaintEventArgs pea),* care suprascrie metoda *OnPaint,* fiind cea care desenează pe tabla de joc X sau O în urma unei mutări.

Aplicația are la bază doi algoritmi, folosiți în jocul Tic-Tac-Toe, pe de o parte algoritmul Minimax, iar pe de altă parte, învățarea prin întărire îmbinată cu o rețea neuronală.

5.2.2 Minimax

Datorită faptului că jocul are o complexitate relativ redusă, am reuşit implementarea unui algoritm care să poată căuta în adâncime până în stările terminale, astfel nemaiavând nevoie de o funcţie de evaluare. Pentru stările finale am folosit o funcţie de utilitate care oferă recompensa 1 pentru victorie, 0 pentru egalitate şi -1 pentru înfrângere. În continuare vom ilustra procedura recursivă, care găseşte cea mai bună mutare pentru jucătorul Minimax:

```
private MMove GetCeaMaiBunaMutare(MTabla tabla, int JucatorCurent)
        {
            MMove best = null;
            int MutariPosibile = tabla.PozitiiLibere;
            int i = r.Next(3); //incepem pe o pozitie aleatoare
            int j = r.Next(3);

            while (MutariPosibile > 0)
            {
                do
                {
                    if (i < 2)
                        i++;
                    else
                        if (j < 2)
                        {
                            i = 0;
                            j++;
                        }
                        else
                        {
                            i = 0;
                            j = 0;
                        }
                }
                while (tabla.Table[i][j] != 0);

                MMove noua = new MMove(i, j);
                MutariPosibile--;


                MTabla NewTabla = new MTabla(tabla);
                NewTabla.FaMutare(JucatorCurent, noua);
                NewTabla.VerificCastigator();

                if (NewTabla.Stare == 2)//joc in progres
                {
                    MMove     temp     =     GetCeaMaiBunaMutare(NewTabla,     -
JucatorCurent, alfa, beta);
                    noua.valoare = temp.valoare;
                }
                else  //joc in stare finala
```

```
                {
                    noua.valoare = NewTabla.Stare;
                }
                //daca e noua cea mai buna mutare de pana acum
                if (best == null || (JucatorCurent == pc && noua.valoare <
best.valoare) ||
                    (JucatorCurent == om && noua.valoare > best.valoare))
                {
                    best = noua;
                }
            }

            return best;
        }
```

5.2.3 Învățare prin întărire şi rețea neuronală

În elaborarea acestei părți a aplicației am folosit mai multe clase, dintre care, cele mai importante ar fi următoarele: *RețeaNeuronală, Joc, AgentInteligent, JucătorOm.* Tabla de joc a fost codificată intr-o matrice de 3 x 3 inițializată cu valoarea 0. Pe parcursul jocului se completează cu valori: -1 pentru X şi 1 pentru O.

Vom prezenta în continuare procedura care stă la baza antrenării rețelei neuronale:

```
public void trainNet(int Player)
        {
            int[][] table = new int[3][];
            for (int i = 0; i < 3; i++)
                table[i] = new int[3];

            int[][] BestTable = new int[3][];
            for (int i = 0; i < 3; i++)
                BestTable[i] = new int[3];

            double Recompensa = 0.0;
            int MoveNum = 0;
            bool MuatreDeExplorare;

            while (true)
            {
                MuatreDeExplorare = getBestTable(table, BestTable, Player,
true);
                copyTable(BestTable, table);
                if (win(table)) //jocul s-a terminat cu victorie
                {
                    Recompensa = Player * -1;
                }
                else
```

```
                {
                    Recompensa = 0.0;
                }
                if (!MuatreDeExplorare)  //daca nu e mutare de explorare
                {
                    ReteaN.updateWeights(0.0, (double[])prepareInputVector(-
1, table).Clone());
                }
                if (Recompensa != 0)
                {
                    break;
                }
                else if (MoveNum == 8)
                {
                    break;
                }
                MoveNum++;
                Player *= -1; // schimba jucatorul
            }

            ReteaN.finishUpdateWeights(reward); //face update la ponderile
retelei
            NrJocuri++;
              }
```

5.3 Manual de utilizare

Pentru rularea aplicația X-O avem nevoie de un computer echipat cu un sistem de operare Microsoft Windows şi .Net Framework 2.0. La lansarea în execuție apare tabla de joc, şi implicit este pornit un joc împotriva lui Minimax (Figura 5-5). Apoi jucătorul (uman) poate da click pe una dintre căsuțele libere pentru a o marca. Imediat calculatorul va răspunde cu o mutare şi cei doi îşi succed mutările până când jocul se încheie, cu remiză sau victoria unuia dintre ei. În Figura 5-6 este ilustrată tabla de joc după terminarea partidei, în acest caz fiind egalitate între jucătorul uman şi Minimax. Titlul inițial al ferestrei a fost “OM vs PC(Minimax)”, iar la încheierea partidei se va schimba în funcție de rezultat. În cazul de față va deveni “Remiză !” (Figura 5-6).

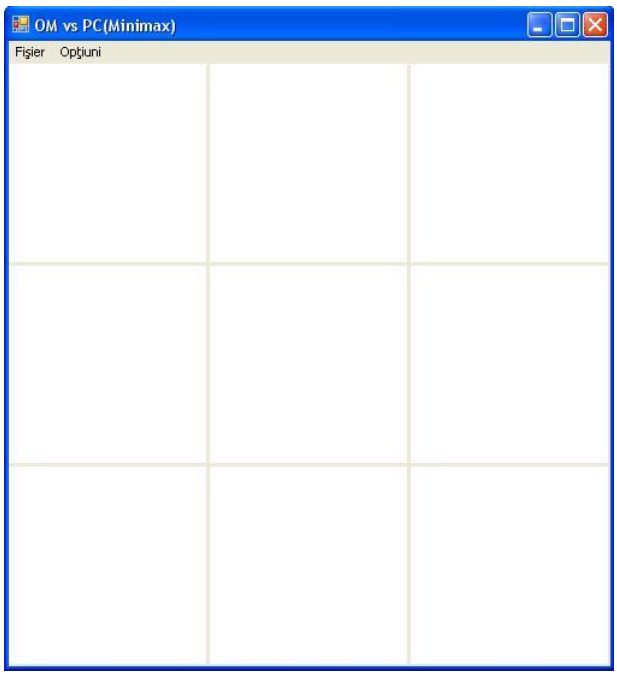


Figura 5-5 Tablă de joc goală

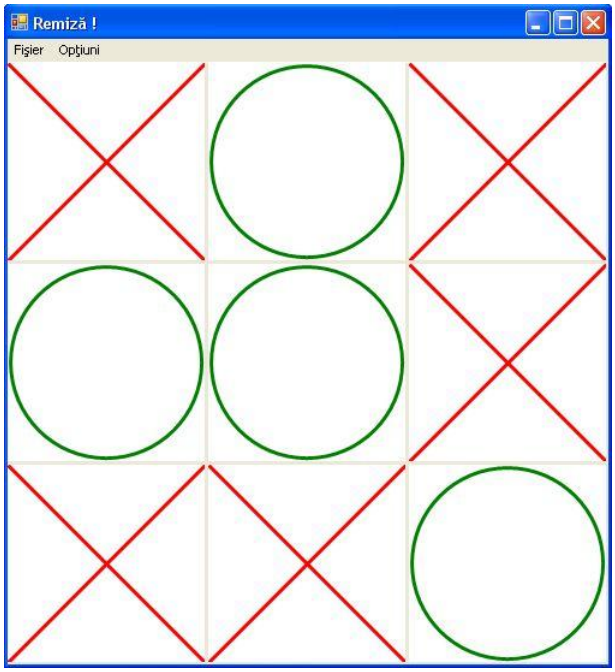


Figura 5-6 Joc terminat la egalitate

Meniul aplicației are următoarele componente:

1. **Fişier**, cu următoarele opțiuni (Figura 5-7):
    a. **Joc nou,** cu opțiunile de a începe un joc nou împotriva lui Minimax, sau împotriva rețelei neuronale instruită cu TD. În cazul în care este selectată cea de-a doua opțiune, va mai apărea o fereastră de unde se pot selecta detalii despre modul de învățare: valoarea coeficientului *alfa* şi numărul de jocuri de antrenament. Opțiunea de joc nou poate fi accesată în orice moment, chiar dacă un joc se află deja în desfăşurare.
    b. **Ieşire,** se produce închiderea aplicației, fără a mai fi solicitat nici un mesaj de confirmare. Se poate apela şi prin combinația de taste *CTRL+I.*

2. **Opțiuni,** cu următoarele componente (Figura 5-8):
    a. **Omul mută primul,** setează ca jucătorul care face prima mutare să fie omul.
    b. **PC mută primul,** ca jucătorul care face prima mutare să fie computerul.
    Opțiunile **a** şi **b** autoexclud în mod reciproc.
    c. **Omul e cu X,** jucătorul uman este cel care foloseşte X.
    d. **PC e cu X,** computerul este cel acre foloseşte X.
    Opțiunile **c** şi **d** autoexclud în mod reciproc.

    Acest sub meniu este funcționabil numai după terminarea jocului în curs. Modificările făcute în timpul unui joc nu sunt luate în calcul.

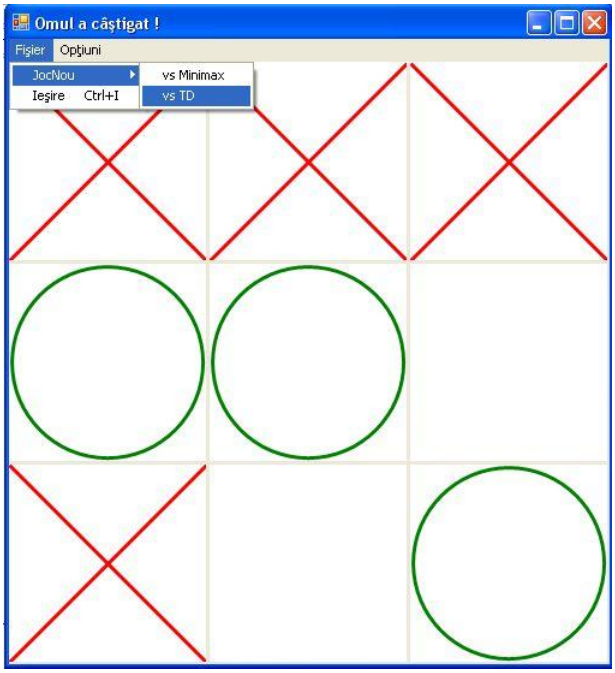


Figura 5-7 Meniu – Fişier

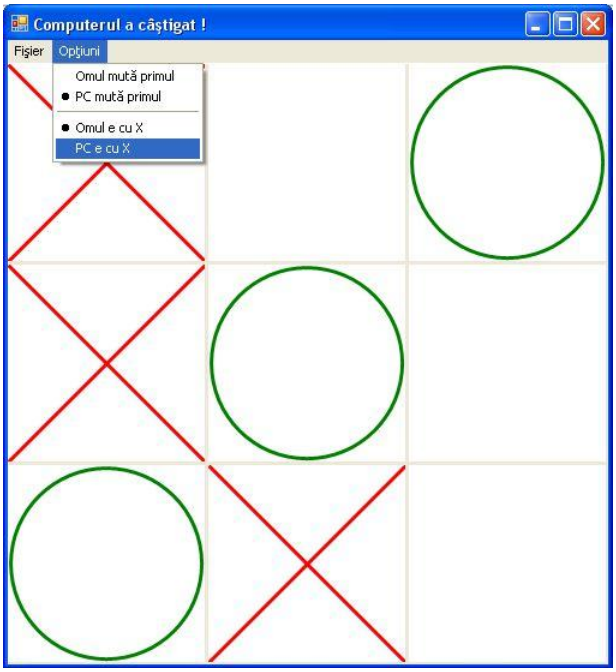


Figura 5-8 Meniu – Opțiuni

6. Concluzii

În lucrarea de față, ne-am propus să prezentăm învățarea prin întărire, mai exact învățarea bazată pe diferențe temporare, ca o alternativă la abordarea tradiționala a jocurilor, folosind metoda Minimax.

Mai întâi am prezentat abordarea tradițională a jocului Tic-Tac-Toe, folosind metoda Minimax. Jucătorul inteligent rezultat joacă într-adevăr perfect acest joc (nu pierde niciodată), dar foloseşte o căutare în adâncime costisitoare din punct de vedere al complexității. Pentru un joc mai complex cum ar fi tablele sau şahul, folosind Minimax nu am reuşi să construim un jucător inteligent imbatabil, ci cel mult unul de nivel mediu.

În paralel, am reuşit realizarea unui sistem având la bază o rețea neuronală care învață să joace Tic-Tac-Toe, fără a avea nici un fel de cunoştințe inițiale. Rețeaua se autoinstruieşte jucând un set de jocuri împotriva ei înşişi şi acordându-şi recompense după fiecare partidă. Este practic o învățare din interacțiunea cu mediul, reprezentat în acest caz de oponent. După câteva mii de jocuri de antrenament ajunge să joace la un nivel foarte bun, fiind un adversar veritabil pentru jucătorul uman.

În jocurile de instruire, am pus rețeaua să joace împotriva ei înşişi, dar ar fi interesant de văzut cum decurge învățarea dacă partenerul de antrenament este jucătorul Minimax. De asemenea, păstrându-i structura, acest sistem ar putea fi extins să joace table sau orice alt joc care se desfăşoară pe o tablă.

Exemplul prezentat este destul de simplu şi dă impresia că învățarea prin întărire este limitată, ceea ce este departe de a fi adevărat. Ea poate fi aplicată nu numai în jocuri ci şi într-o mulțime de alte domenii ale Inteligenței Artificiale, în rezolvarea multor probleme din lumea reală pentru care încă nu avem soluții concrete.

7. Bibliografie

1. **Haugeland, John.** *Artificial Intelligence: The Very Idea.* s.l. : MIT Press, 1989.

2. **Winston, Patrick Henry.** *Artificial intelligence (3rd ed.).* Boston : Addison-Wesley Longman Publishing Co., Inc , 1992.

3. **Rich, Elaine şi Knight, Kevin.** *Artificial intelligence 2nd ed.* New York : McGraw-Hill, 1991.

4. **Kurzweil, Ray.** *The Age of Intelligent Machines.* Cambridge : MIT Press, 1990.

5. **Luger, G.F şi Stubbliefeld, W.A.** *rtificial Intelligence : Structures and Strategies for Complex Problem Solving.* Redwood City : s.n., 1993.

6. *Inteligența Artificială.* **Vasilescu, Raluca.** 1996.

7. **Dumitrescu, Dan.** *Principiile Inteligenței Artificiale.* Cluj-Napoca : Ed. Albastră, 1999.

8. **Gâlea, Dan şi Leon, Florin.** Curs de Inteligență Artificială: Universitatea Tehnică "Gh. Asachi" Iaşi. [Interactiv] http://eureka.cs.tuiasi.ro/~fleon/curs_ia.htm.

9. **Ghionea, Ionuț.** Inteligența Artificială. [Interactiv] http://www.catia.ro/articole/ai/ai.htm.

10. **Wiens, Elmer G.** Operations Research - Game Theory . [Interactiv] http://www.egwald.ca/operationsresearch/gameintroduction.php.

11. **Weiss, Adrienne J.** *Educational board game* . s.l. : Buchanan Ingersoll Professional Corp. , 1998.

12. **Rosenthal, R. W.** *A class of games possessing pure-strategy, Nash equilibrium.* s.l. : International Jounal of Game Theory 2, 1973.

13. *Minimax reference point approach and its application for multiobjective optimisation* . **Yang, Jian-Bo.** 3, Manchester : European Journal of Operational Research Manchester School of Management, 2000, Vol. 126.

14. **Stoean, Cătălin.** Teoria Jocurilor. [Interactiv] http://inf.ucv.ro/~cstoean.

15. **Pop, Horia F. şi Şerban, Gabriela.** *Inteligența Artificială.* Cluj-Napoca : UBB Cluj-Napoca, 2004.

16. *Temporal Difference Learning and TD-Gammon.* **Tesauro, Gerald.** 3, s.l. : Communications of the ACM, 1995, Vol. 38.

17. **Şerban, Gabriela.** *Sisteme multiagent în Inteligența Artificială distribuită.* Cluj-Napoca : Editura Risoprint, 2006.

18. **Sutton, Richard şi Barto, Andrew.** *Reinforcement Learning: An Introduction.* s.l. : MIT Press, 1998.

19. **Dumitrescu, Dan.** *Rețele neuronale. Teorii şi aplicații.* Bucureşti : Teora, 1996.