UNIVERSITATEA DIN BUCUREȘTI

Facultatea de Matematică și Informatică

Lucrare de licență

Implementarea jocului "Backgammon" in limbajul Java

Îndrumător: *Lect.dr. Cidota Marina*

Absolvent: *Josu Vladimir*

-2010-

Cuprins

Prefata.....2
1. Teoria jocurilor.....4
1.1. Introducere în inteligența artificială.....4
1.1.1. Istoric.....5
1.1.2. Subdomenii ale inteligenței arfiticiale.....6
1.2. Introducere în teoria jocurilor.....8
1.2.1. Algoritmul Minimax.....11
1.2.2. Algoritmul Alpha-Beta.....14
2. Jocul Backgammon.....18
2.1. Introducere.....18
2.1.1. Istoria.....18
2.1.2. Iventar.....20
2.2. Regulament.....20
2.3. Variante ale jocului.....24
3. Dezvoltarea aplicației.....26
3.1. Tehnologii utilizate.....26
3.1.1. Java.....25
3.1.2. Pachetul de interfață Java Swing.....26
3.1.3. Fire de execuție.....27
3.1.4. NetBeans.....28
3.2. Implementarea algoritmului..... 31
3.3. Implementarea interfeței.....43
4. Manual de utilizare.....44
Bibliografie.....51

Prefață

Când s-a vorbit prima data de Inteligenta Artificiala (AI Artificial Intelligence) în 1956, totul parea o utopie, un vis prea frumos pentru a fi realizat, un stadiu al dezvoltarii considerat a fi greu de atins. în ultimii aproape 50 de ani, termenul a prins contur, devenind realitate, fiind în prezent folosit în toate stiintele care doresc sa se afirme. Initiatorul sau, prof. John McCarthy a prezentat noul concept în vara anului 1956 la întrunirea "Darthmouth Summer Research Project on Artificial Intelligence".

Termenul de Inteligenta Artificiala este întâlnit azi în numeroase publicatii tehnice, medicale, militare, stiintifice, de obicei, când vine vorba de aplicatii ce realizeaza performante de care numai omul era socotit capabil: recunoasterea si analiza vocii si a imaginilor, traduceri dintr-o limba în alta, diferite jocuri de inteligenta, luarea unor decizii complexe fara interventia unui operator uman etc. Initial, obiectivele Inteligentei Artificiale au fost foarte ambitioase: masina trebuia sa rezolve diferite probleme, sa învete din propria experienta si din evenimentele exterioare sistemului sau, sa efectueze rationamente, sa conceapa noi obiecte cu proprietati prestabilite.

Principalul scop al Inteligentei Artificiale este de a imita întrutotul creierul uman în modul în care acesta gândeste, raspunde si interactioneaza. în pofida nivelului atins de cercetatori, acest deziderat nu va fi atins foarte curând, creierul uman fiind înca o enigma, aproape imposibil de analizat matematic si/sau tradus în limbaj masina.

Indiferent de puterea lor de procesare, masinile nu vor înlocui, probabil niciodata, omul, cea mai inteligenta si puternica fiinta de pe Pamânt. Aceasta afirmatie este sprijinita de numeroase ratiuni. Cel mai important argument împotriva dezvoltarii masinii cu adevarat inteligente este cel al evolutiei. Masinile nu au parcurs rigorile de supravietuire timp de milioane de ani precum oamenii. Modul în care acestia interactioneaza, gândesc si se adapteaza sunt faze de dezvoltare ale intelectului, diferit fiecarui individ în parte. Acestui intelect i-au fost necesare milioane de ani sa evolueze, reprezentând, astfel, o etapa extrem de dificil de implementat în dezvoltarea masinii inteligente.

În informatică, în general, inteligenţa artificială e împărţită în două categorii:

- inteliligenţă artificială puternică (strong AI): prin aceasta se înţelege o intelegenţă artificială, de obicei bazată pe un computer, care chiar poate "gândi" şi este "conştientă de sine".

• inteligenţă artificială slabă (weak AI): o inteligenţă artificială care nu pretinde că poate gândi, putând însă rezolva o anumită clasă de probleme într-un mod mai mult sau mai puţin "inteligent", de exemplu cu ajutorul unui set de reguli.

Progresul în crearea unei inteligenţe artificiale puternice este mic. Aproape toate simulările inteligenţei se bazează pe reguli şi algoritmi obişnuiţi, existând un progres doar în domeniul celei slabe (de exemplu la recunoaşterea verbală şi a scrisului, la traducerea automată dintr-o limbă în alta sau şi la diferite jocuri, sah, backgammon).

Scopul lucrării de față este de a implementa in Java jocul Backgammon, cunoscut in română ca „Jocul de Table”.

Lucrarea cuprinde patru capitole, pe care le descriu pe scurt mai jos:

Primul capitolul începe cu o scurtă introducere in domeniul inteligenței artificiale, si continua cu descrierea detaliata a algoritmilor clasici folositi in teoria jocurilor: minimax și alpha-beta.

Capitolul 2 contine prezentarea generală a jocului Backgammon: istoria jocului, regulile, precum si câteva variante din diferite țari.

Capitolul 3 introduce tehnologiile utilizate pentru dezvoltarea aplicatiei, clasele și metodele cele mai importante, cu mici explicații asupra modului de implementare al acestora, precum si detalii despre interfata grafică.

Capitolul 4 pune la dispozitie un manual de utilizare a aplicației, fiind inserate capturi de ecran care explicitează modul de folosire.

1. Teoria jocurilor

1.1. Introducere în Inteligenta Artificială

Inteligenta Artificială poate fi definita ca simularea inteligentei umane procesata de masini, în special, de sisteme de computere. Acest domeniu a fost, în general, caracterizat de cercetari complexe în laboratoare si doar destul de recent a devenit parte a tehnologiei în aplicatii comerciale [3].

Inteligenta artificiala - stiinta si tehnologia crearii de masini intelectuale, mai cu seama crearea softurilor pe calculator. IA are legatura cu o sarcina asemanatoare utilizarii calculatoarelor pentru intelegerea intelectului uman, insa aceasta nu se limiteaza neaparat biologic la astfel de metode.

Câteva definiții

- Directie stiintifica, in limitele careia se pun si se rezolva diverse sarcini ale unei modelari de soft pentru operatiuni umane care in mod traditional sunt considerate intelectuale. Conform acestei definiţii primare, putem gândi inteligenţa artificială ca fiind acel domeniu al informaticii care se ocupă de automatizarea comportamentului inteligent.
- Știinta sub numele de AI intra in complexul stiintei de calculatoare, iar tehnologiile create pe baza acesteia conduc spre tehnologiile de informatie.Datoria acestei stiinte este reconstructia cu ajutorul sistemului de calculare si cu astfel de dispozitive artificiale de actiuni si gandiri rezonabile

Istoric

La început, crearea şi cercetarea inteligenţei artificiale s-a desfăşurat pe domeniul psihologiei, punându-se accent pe inteligenţa lingvistică, ca de exemplu la testul Turing. Acest test constă într-o conversaţie în limbaj uman natural cu o maşină (computer) care a fost programată special pentru acest test. Există un juriu uman care conversează cu acest computer, dar şi cu un om, prin câte un canal pur text (fără ca ei să se vadă sau să se audă). În cazul în care juriul nu poate să-şi dea seama care este computerul şi care omul, atunci inteligenţa artficială (programul calculatorului) a trecut testul.

Turing a prezis în 1950 că până în anul 2000 vor exista maşini (calculatoare) cu 109 bytes (1 GB) de memorie care vor putea "păcăli" 30% din juriile umane într-un test de 5 minute. Însă, în timp ce pe de-o parte tehnologia chiar a depăşit previziunile lui Turing, inteligenţa artificială este încă departe de a fi realizată.

Noile previziuni ale experţilor se bazează pe aşa-numita legea lui Moore ("numărul de tranzistori pe un circuit integrat se va dubla la fiecare 18 luni, prin urmare şi puterea de calcul"), "lege" care s-a îndeplinit pentru ultimii 30 de ani destul de bine, şi poate că va mai fi valabilă încă 5-10 ani. Pentru viitor se speră că noile tehnologii (cuantice, optice, holografice, nanotehnologiile ş.a.) vor permite menţinerea creşterii exponenţiale, astfel că în maximum 20 de ani computerele să depăşească puterea de procesare a creierului uman (vezi: Singularitate tehnologică). Unul dintre principalii susţinători ai acestei ipoteze, pe lângă Vernor Vinge, este cunoscutul expert Ray Kurzweil cu a sa celebră lege a întoarcerilor accelerate. Însă aceste consideraţii sunt în general de natură cantitativă, neglijând din păcate nenumăratele faţete calitative ale inteligenţei umane naturale.

Subdomenii ale inteligenţei artificiale

- jocurile (bazate pe căutarea efectuată într-un spaţiu de stări ale problemei)
- raţionamentul automat şi demonstrarea teoremelor (bazate pe rigoarea şi generalitatea logicii matematice. Este cea mai veche ramură a inteligenţei artificiale şi cea care înregistrează cel mai mare succes. Cercetarea în domeniul demonstrării automate a teoremelor a dus la formalizarea algoritmilor de căutare şi la dezvoltarea unor limbaje de reprezentare formală, cum ar fi calculul predicatelor şi limbajul pentru programare logică Prolog)
- sistemele expert (care pun în evidenţă importanţa cunoştinţelor specifice unui domeniu)
- înţelegerea limbajului natural şi modelarea semantică (caracteristica de bază a oricărui sistem de înţelegere a limbajului natural o constituie reprezentarea sensului propoziţiilor într-un anumit limbaj de reprezentare astfel încât aceasta să poată fi utilizată în prelucrări ulterioare)
- planificarea şi robotica (Planificarea presupune exempluistenţa unui robot capabil să exempluecute anumite acţiuni atomice, cum ar fi deplasarea într-o cameră plină cu obstacole)
- învăţarea automată (datorită căreia se realizează adaptarea la noi circumstanţe, precum şi detectarea şi exemplutrapolarea unor şabloane - “patterns”. Învăţarea se realizează, spre exemplu, prin intermediul aşa-numitelor reţele neurale sau neuronale. O asemenea reţea reprezintă un tip de sistem de inteligenţă artificială

modelat după neuronii -celulele nervoase- dintr-un sistem nervos biologic, în încercarea de a simula modul în care creierul prelucrează informaţiile, învaţă sau îşi aduce aminte)

Toate aceste subdomenii ale inteligenţei artificiale au anumite trăsături în comun, şi anume:

- O concentrare asupra problemelor care nu răspund la soluţii algoritmice; din această cauză tehnica de rezolvare a problemelor specifică inteligenţei artificiale este aceea de a se baza pe o căutare euristică.
- IA rezolvă probleme folosind şi informaţie inexempluactă, care lipseşte sau care nu este complet definită şi utilizează formalisme de reprezentare ce constituie pentru programator o compensaţie faţă de aceste probleme.
- IA foloseşte raţionamente asupra trăsăturilor calitative semnificative ale unei situaţii
- IA foloseşte, în rezolvarea problemelor, mari cantităţi de cunoştinţe specifice domeniului investigat.

Majoritatea tehnicilor din inteligenţa artificială folosesc pentru implementarea inteligenţei

- reprezentarea cunoştinţelor adresează problema reprezentării într-un limbaj formal, adică un limbaj adecvat pentru prelucrarea ulterioară de către un calculator, a întregii game de cunoştinţe necesare comportamentului inteligent;
- căutarea este o tehnică de rezolvare a problemelor care exempluplorează în mod sistematic un spaţiu de stări ale problemei, adică de stadii succesive şi alternative în procesul de rezolvare a acesteia (exemple de stări ale problemei pot fi configuraţiile diferite ale tablei de şah în cadrul unui joc sau paşii intermediari într-un proces de raţionament).

În IA exista două abordări majore complet diferite ale problematicii domeniului:

- abordarea simbolică;
- abordarea conecţionistă.

Într-o abordare simbolică cunoştinţele vor fi reprezentate în mod simbolic. O abordare total diferită -cea conecţionistă- este aceea care urmăreşte să construiască programe

inteligente folosind modele care fac un paralelism cu structura neuronilor din creierul uman.

1.2. Introducere în teoria jocurilor

Jocurile reprezintă o arie de aplicaţie interesantă pentru algoritmii euristici.

Jocurile de două persoane sunt în general complicate datorită existenţei unui oponent ostil şi imprevizibil. De aceea ele sunt interesante din punctul de vedere al dezvoltării euristicilor, dar aduc multe dificultăţi în dezvoltarea şi aplicarea algoritmilor de căutare.

Oponentul introduce incertitudinea, întrucât nu se ştie niciodată ce va face acesta la pasul următor. În esenţă, toate programele referitoare la jocuri trebuie să trateze aşa numita problemă de contingenţă. (Aici contingenţă are sensul de întâmplare).

Incertitudinea care intervine în cazul jocurilor nu este de aceeaşi natură cu cea introdusă, de pildă, prin aruncarea unui zar sau cu cea determinată de starea vremii. Oponentul va încerca, pe cât posibil, să facă mutarea cea mai puţin benignă, în timp ce zarul sau vremea sunt presupuse a nu lua în consideraţie scopurile agentului.

Complexitatea jocurilor introduce un tip de incertitudine complet nou. Astfel, incertitudinea se naşte nu datorită faptului că există informaţie care lipseşte, ci datorită faptului că jucătorul nu are timp să calculeze consecinţele exacte ale oricărei mutări. Din acest punct de vedere, jocurile se aseamănă infinit mai mult cu lumea reală decât problemele de căutare standard.

Întrucât, în cadrul unui joc, există, de regulă, limite de timp, jocurile penalizează ineficienţa extrem de sever. Astfel, dacă o implementare a căutării de tip A*, care este cu 10% mai puţin eficientă, este considerată satisfăcătoare, un program pentru jocul de şah care este cu 10% mai puţin eficient în folosirea timpului disponibil va duce la pierderea partidei. Din această cauză, studiul nostru se va concentra asupra tehnicilor de alegere a unei bune mutări atunci când timpul este limitat. Tehnica de “retezare” ne va permite să ignorăm porţiuni ale arborelui de căutare care nu pot avea nici un rol în stabilirea alegerii finale, iar funcţiile de evaluare euristice ne vor permite să aproximăm utilitatea reală a unei stări fără a executa o căutare completă.

Ne vom referi la tehnici de joc corespunzătoare unor jocuri de două persoane cu informaţie completă, cum ar fi backgammon.

În cazul jocurilor interesante, arborii rezultaţi sunt mult prea complecşi pentru a se putea realiza o căutare exhaustivă, astfel încât sunt necesare abordări de o natură diferită. Una dintre metodele clasice se bazează pe „principiul minimax”, implementat în mod eficient sub forma Algoritmului Alpha-Beta (bazat pe aşa-numita tehnică de alpha-beta retezare).

- **O definire formală a jocurilor**

Tipul de jocuri la care ne vom referi în continuare este acela al jocurilor de două persoane cu informaţie perfectă sau completă. În astfel de jocuri există doi jucători care efectuează mutări în mod alternativ, ambii jucători dispunând de informaţia completă asupra situaţiei curente a jocului. (Prin aceasta, este exclus studiul majorităţii jocurilor de cărţi). Jocul se încheie atunci când este atinsă o poziţie calificată ca fiind "terminală" de către regulile jocului - spre exemplu, "mat" în jocul de şah. Aceleaşi reguli determină care este rezultatul jocului care s-a încheiat în această poziţie terminală. Un asemenea joc poate fi reprezentat printr-un arbore de joc în care nodurile corespund situaţiilor (stărilor), iar arcele corespund mutărilor. Situaţia iniţială a jocului este reprezentată de nodul rădăcină, iar frunzele arborelui corespund poziţiilor terminale.

Vom lua în consideraţie cazul general al unui joc cu doi jucători, pe care îi vom numi MAX şi respectiv MIN. MAX va face prima mutare, după care jucătorii vor efectua mutări pe rând, până când jocul ia sfârşit. La finalul jocului vor fi acordate puncte jucătorului câştigător (sau vor fi acordate anumite penalizări celui care a pierdut).

Un joc poate fi definit, în mod formal, ca fiind un anumit tip de problemă de căutare având următoarele componente:

- starea iniţială, care include poziţia de pe tabla de joc şi o indicaţie referitoare la cine face prima mutare;
- o mulţime de operatori, care definesc mişcările permise ("legale") unui jucător;
- un test terminal, care determină momentul în care jocul ia sfârşit;
- o funcţie de utilitate (numită şi funcţie de plată), care acordă o valoare numerică rezultatului unui joc; în cazul jocului de şah, spre exemplu, rezultatul poate fi câştig, pierdere sau remiză, situaţii care pot fi reprezentate prin valorile 1, -1 sau 0.

Dacă un joc ar reprezenta o problemă standard de căutare, atunci acţiunea jucătorului MAX ar consta din căutarea unei secvenţe de mutări care conduc la o stare terminală reprezentând o stare câştigătoare (conform funcţiei de utilitate) şi din efectuarea primei

mutări aparţinând acestei secvenţe. Acţiunea lui MAX interacţionează însă cu cea a jucătorului MIN. Prin urmare, MAX trebuie să găsească o strategie care va conduce la o stare terminală câştigătoare, indiferent de acţiunea lui MIN. Această strategie include mutarea corectă a lui MAX corespunzătoare fiecărei mutări posibile a lui MIN. În cele ce urmează, vom începe prin a arăta cum poate fi găsită strategia optimă (sau raţională), deşi în realitate nu vom dispune de timpul necesar pentru a o calcula.

1.2.1 Algoritmul Minimax

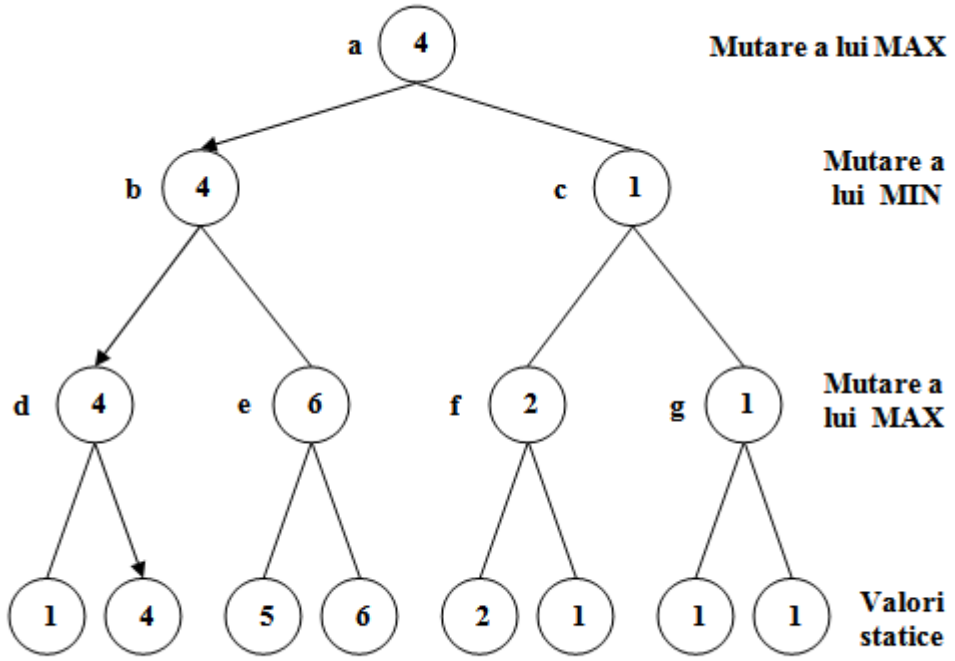
Oponenţii din cadrul jocului pe care îl vom trata prin aplicarea Algoritmului Minimax vor fi numiţi, în continuare, MIN şi respectiv MAX. MAX reprezintă jucătorul care încearcă să câştige sau să îşi maximizeze avantajul avut. MIN este oponentul care încearcă să minimizeze scorul lui MAX. Se presupune că MIN foloseşte aceeaşi informaţie şi încearcă întotdeauna să se mute la acea stare care este cea mai nefavorabilă lui MAX.

Algoritmul Minimax este conceput pentru a determina strategia optimă corespunzătoare lui MAX şi, în acest fel, pentru a decide care este cea mai bună primă mutare:

1. Generează întregul arbore de joc, până la stările terminale.
2. Aplică funcţia de utilitate fiecărei stări terminale pentru a obţine valoarea corespunzătoare stării.
3. Deplasează-te înapoi în arbore, de la nodurile-frunze spre nodul-rădăcină, determinând, corespunzător fiecărui nivel al arborelui, valorile care reprezintă utilitatea nodurilor aflate la acel nivel. Propagarea acestor valori la niveluri anterioare se face prin intermediul nodurilor- părinte succesive, conform următoarei reguli:
   - dacă starea-părinte este un nod de tip MAX, atribuie-i maximul dintre valorile avute de fiii săi;
   - dacă starea-părinte este un nod de tip MIN, atribuie-i minimul dintre valorile avute de fiii săi.
4. Ajuns în nodul-rădăcină, alege pentru MAX acea mutare care conduce la valoarea maximă.

❑ Observaţie: Decizia luată la pasul 4 al algoritmului se numeşte decizia minimax, întrucât ea maximizează utilitatea, în ipoteza că oponentul joacă perfect cu scopul de a o minimiza.

Un arbore de căutare cu valori minimax determinate conform Algoritmului Minimax este cel din figura următoare:



Valorile poziţiilor de la ultimul nivel sunt determinate de către funcţia de utilitate şi se numesc valori statice. Valorile minimax ale nodurilor interne sunt calculate în mod dinamic, în manieră bottom-up, nivel cu nivel, până când este atins nodul-rădăcină. Valoarea rezultată, corespunzătoare acestuia, este 4 şi, prin urmare, cea mai bună mutare a lui MAX din poziţia a este a-b. Cel mai bun răspuns al lui MIN este b-d. Această secvenţă a jocului poartă denumirea de variaţie principală. Ea defineşte jocul optim de tip minimax pentru ambele părţi. Se observă că valoarea poziţiilor de-a lungul variaţiei principale nu variază. Prin urmare, mutările corecte sunt cele care conservă valoarea jocului.

Dacă adâncimea maximă a arborelui este m şi dacă există b "mutări legale" la fiecare punct, atunci complexitatea de timp a Algoritmului Minimax este $O(b^m)$. Algoritmul reprezintă o căutare de tip depth-first (deşi aici este sugerată o implementare bazată pe recursivitate şi nu una care foloseşte o coadă de noduri), astfel încât cerinţele sale de spaţiu sunt numai liniare în m şi b.

În cazul jocurilor reale, cerinţele de timp ale algoritmului sunt total nepractice, dar acest algoritm stă la baza atât a unor metode mai realiste, cât şi a analizei matematice a jocurilor. Întrucât, pentru majoritatea jocurilor interesante, arborele de joc nu poate fi alcătuit în mod exhaustiv, au fost concepute diverse metode care se bazează pe căutarea efectuată numai într-o anumită porţiune a arborelui de joc. Printre acestea se numără şi tehnica Minimax, care, în majoritatea cazurilor, va căuta în arborele de joc numai până la o anumită adâncime, de obicei constând în numai câteva mutări. Ideea este de a evalua aceste poziţii terminale ale căutării, fără a mai căuta dincolo de ele, cu scopul de a face economie de timp. Aceste estimări se propagă apoi în sus de-a lungul arborelui, conform principiului Minimax. Mutarea care conduce de la poziţia iniţială, nodul-rădăcină, la cel mai promiţător succesor al său (conform acestor evaluări) este apoi efectuată în cadrul jocului.

Algoritmul general Minimax a fost amendat în două moduri: funcţia de utilitate a fost înlocuită cu o funcţie de evaluare, iar testul terminal a fost înlocuit de către un aşa-numit test de tăiere.

Cea mai directă abordare a problemei deţinerii controlului asupra cantităţii de căutare care se efectuează este aceea de a fixa o limită a adâncimii, astfel încât testul de tăiere să aibă succes pentru toate nodurile aflate la sau sub adâncimea d. Limita de adâncime va fi aleasă astfel încât cantitatea de timp folosită să nu depăşească ceea ce permit regulile jocului. O abordare mai robustă a acestei probleme este aceea care aplică „iterative deepening”. În acest caz, atunci când timpul expiră, programul întoarce mutarea selectată de către cea mai adâncă căutare completă.

- **Funcţii de evaluare**

  O funcţie de evaluare întoarce o estimaţie, realizată dintr-o poziţie dată, a utilităţii aşteptate a jocului. Ea are la bază evaluarea şanselor de câştigare a jocului de către fiecare dintre părţi, pe baza calculării caracteristicilor unei poziţii. Performanţa unui program referitor la jocuri este extrem de dependentă de calitatea funcţiei de evaluare utilizate.

  Funcţia de evaluare trebuie să îndeplinească anumite condiţii evidente: ea trebuie să concorde cu funcţia de utilitate în ceea ce priveşte stările terminale, calculele efectuate nu trebuie să dureze prea mult şi ea trebuie să reflecte în mod corect şansele efective de câştig. O valoare a funcţiei de evaluare acoperă mai multe poziţii diferite, grupate laolaltă într-o categorie de poziţii etichetată cu o anumită valoare. Spre exemplu, în jocul de şah, fiecare pion poate avea valoarea 1, un nebun poate avea

valoarea 3 şamd. În poziţia de deschidere evaluarea este 0 şi toate poziţiile până la prima captură vor avea aceeaşi evaluare. Dacă MAX reuşeşte să captureze un nebun fără a pierde o piesă, atunci poziţia rezultată va fi evaluată la valoarea 3. Toate poziţiile de acest fel ale lui MAX vor fi grupate într-o categorie etichetată cu "3". Funcţia de evaluare trebuie să reflecte şansa ca o poziţie aleasă la întâmplare dintr-o asemenea categorie să conducă la câştig (sau la pierdere sau la remiză) pe baza experienţei anterioare.

Funcţia de evaluare cel mai frecvent utilizată presupune că valoarea unei piese poate fi stabilită independent de celelalte piese existente pe tablă. Un asemenea tip de funcţie de evaluare se numeşte *funcţie liniară ponderată*, întrucât are o expresie de forma

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

unde valorile $w_i,\ i=\overline{1,n}$ reprezintă ponderile, iar $f_i,\ i=\overline{1,n}$ sunt caracteristicile unei anumite poziţii. În cazul jocului de şah, spre exemplu $w_i,\ i=\overline{1,n}$ ar putea fi valorile pieselor (1 pentru pion, 3 pentru nebun etc.), iar $f_i,\ i=\overline{1,n}$ ar reprezenta numărul pieselor de un anumit tip aflate pe tabla de şah.

În construirea formulei liniare trebuie mai întâi alese caracteristicile, operaţie urmată de ajustarea ponderilor până în momentul în care programul joacă suficient de bine. Această a doua operaţie poate fi automatizată punând programul să joace multe partide cu el însuşi, dar alegerea unor caracteristici adecvate nu a fost încă realizată în mod automat.

1.2.2 Algoritmul Alpha-Beta - o implementare eficientă a principiului Minimax

Tehnica pe care o vom examina, în cele ce urmează, este numită în literatura de specialitate alpha-beta prunning ("alpha-beta retezare"). Atunci când este aplicată unui arbore de tip minimax standard, ea va întoarce aceeaşi mutare pe care ar furniza-o şi Algoritmul Minimax, dar într-un timp mai scurt, întrucât realizează o retezare a unor ramuri ale arborelui care nu pot influenţa decizia finală.

- Principiul general al acestei tehnici constă în a considera un nod oarecare n al arborelui, astfel încât jucătorul poate alege să facă o mutare la acel nod. Dacă acelaşi jucător dispune de o alegere mai avantajoasă, m, fie la nivelul nodului părinte al lui n, fie în orice punct de decizie aflat mai sus în arbore, atunci n nu va fi niciodată atins în timpul jocului. Prin urmare, de îndată ce, în urma examinării unora dintre

descendenţii nodului n, ajungem să deţinem suficientă informaţie relativ la acesta, îl putem înlătura.

Ideea tehnicii de alpha-beta retezare este aceea de a găsi o mutare "suficient de bună", nu neapărat cea mai bună, dar suficient de bună pentru a se lua decizia corectă. Această idee poate fi formalizată prin introducerea a două limite, alpha şi beta, reprezentând limitări ale valorii de tip minimax corespunzătoare unui nod intern.

Semnificaţia acestor limite este următoarea: alpha este valoarea minimă pe care este deja garantat că o va obţine MAX, iar beta este valoarea maximă pe care MAX poate spera să o atingă. Din punctul de vedere al jucătorului MIN, beta este valoarea cea mai nefavorabilă pentru MIN pe care acesta o va atinge. Prin urmare, valoarea efectivă care va fi găsită se află între alpha şi beta.

Valoarea alpha, asociată nodurilor de tip MAX, nu poate niciodată să descrească, iar valoarea beta, asociată nodurilor de tip MIN, nu poate niciodată să crească.

Dacă, spre exemplu, valoarea alpha a unui nod intern de tip MAX este 6, atunci MAX nu mai trebuie să ia în consideraţie nici o valoare internă mai mică sau egală cu 6 care este asociată oricărui nod de tip MIN situat sub el. Alpha este scorul cel mai prost pe care îl poate obţine MAX, presupunând că MIN joacă perfect. În mod similar, dacă MIN are valoarea beta 6, el nu mai trebuie să ia în consideraţie nici un nod de tip MAX situat sub el care are valoarea 6 sau o valoare mai mare decât acest număr.

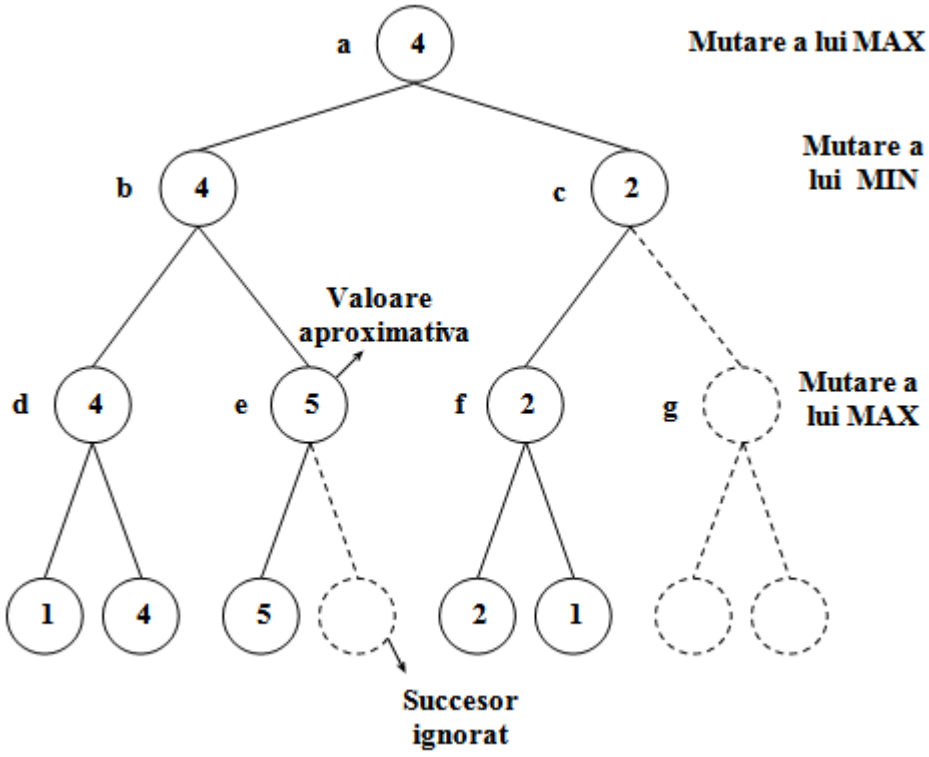
Cele două reguli pentru încheierea căutării, bazată pe valori alpha şi beta, pot fi formulate după cum urmează:

1. Căutarea poate fi oprită dedesubtul oricărui nod de tip MIN care are o valoare beta mai mică sau egală cu valoarea alpha a oricăruia dintre strămoşii săi de tip MAX.
2. Căutarea poate fi oprită dedesubtul oricărui nod de tip MAX care are o valoare alpha mai mare sau egală cu valoarea beta a oricăruia dintre strămoşii săi de tip MIN.

   Dacă, referitor la o poziţie, se arată că valoarea corespunzătoare ei se află în afara intervalului alpha-beta, atunci această informaţie este suficientă pentru a şti că poziţia respectivă nu se află de-a lungul variaţiei principale, chiar dacă nu este cunoscută valoarea exactă corespunzătoare ei. Cunoaşterea

valorii exacte a unei poziţii este necesară numai atunci când această valoare se află între alpha şi beta.

Figura de mai jos ilustrează acţiunea Algoritmului Alpha-Beta în cazul arborelui din de mai sus. Aşa cum se vede în figură, unele dintre valorile de tip minimax ale nodurilor interne sunt aproximative. Totuşi, aceste aproximări sunt suficiente pentru a se determina în mod exact valoarea rădăcinii. Se observă că Algoritmul Alpha-Beta reduce complexitatea căutării de la 8 evaluări statice la numai 5 evaluări de acest tip:



Corespunzător arborelui procesul de căutare decurge după cum urmează:

1. Începe din poziţia a.
2. Mutare la b.
3. Mutare la d.
4. Alege valoarea maximă a succesorilor lui d, ceea ce conduce la V(d) = 4.
5. Întoarce-te în nodul b şi execută o mutare de aici la e.

6. Ia în consideraţie primul succesor al lui e a cărui valoare este 5. În acest moment, MAX, a cărui mutare urmează, are garantată, aflându-se în poziţia e, cel puţin valoarea 5, indiferent care ar fi celelalte alternative plecând din e. Această informaţie este suficientă pentru ca MIN să realizeze că, la nodul b, alternativa e este inferioară alternativei d. Această concluzie poate fi trasă fără a cunoaşte valoarea exactă a lui e. Pe această bază, cel de-al doilea succesor al lui e poate fi neglijat, iar nodului e i se poate atribui valoarea aproximativă 5.

Căutarea de tip alpha-beta retează nodurile figurate în mod discontinuu. Ca rezultat, câteva dintre valorile intermediare nu sunt exacte (nodurile c, e), dar aproximările făcute sunt suficiente pentru a determina atât valoarea corespunzătoare rădăcinii, cât şi variaţia principală, în mod exact.

- **Consideraţii privitoare la eficienţă**

  Eficienţa Algoritmului Alpha-Beta depinde de ordinea în care sunt examinaţi succesorii. Este preferabil să fie examinaţi mai întâi succesorii despre care se crede că ar putea fi cei mai buni. În mod evident, acest lucru nu poate fi realizat în întregime. Dacă el ar fi posibil, funcţia care ordonează succesorii ar putea fi utilizată pentru a se juca un joc perfect. În ipoteza în care această ordonare ar putea fi realizată, s-a arătat că Algoritmul Alpha-Beta nu trebuie să examineze, pentru a alege cea mai bună mutare, decât O(bd/2) noduri, în loc de O(bd), ca în cazul Algoritmului Minimax. Aceasta arată că factorul de ramificare efectiv este în loc de b – în cazul jocului de şah 6, în loc de 35. Cu alte cuvinte, Algoritmul Alpha-Beta poate "privi înainte" la o adâncime dublă faţă de Algoritmul Minimax, pentru a găsi acelaşi cost.

  În cazul unei ordonări neprevăzute a stărilor în spaţiul de căutare, Algoritmul Alpha-Beta poate dubla adâncimea spaţiului de căutare (Nilsson 1980). Dacă există o anumită ordonare nefavorabilă a nodurilor, acest algoritm nu va căuta mai mult decât Algoritmul Minimax. Prin urmare, în cazul cel mai nefavorabil, Algoritmul Alpha-Beta nu va oferi nici un avantaj în comparaţie cu căutarea exhaustivă de tip minimax. În cazul unei ordonări favorabile însă, dacă notăm prin N numărul poziţiilor de căutare terminale evaluate în mod static de către Algoritmul Minimax, s-a arătat că, în cazul cel mai bun, adică atunci când mutarea cea mai puternică este prima luată în consideraţie, Algoritmul Alpha-Beta nu va evalua în mod static decât poziţii. În

practică, o funcţie de ordonare relativ simplă (cum ar fi încercarea mai întâi a capturilor, apoi a ameninţărilor, apoi a mutărilor înainte, apoi a celor înapoi) ne poate apropia suficient de mult de rezultatul obţinut în cazul cel mai favorabil.

2. Jocul Backgammon

2.1. Introducere

Backgammon (sau Table) este un joc de noroc şi strategie între două persoane, cu două zaruri si 30 de puluri, câte 15 pentru fiecare jucător, practicat pe o tablă specială de joc alcătuită din două rânduri de câte 12 săgeţi încrustate sau pictate. Jocul de table este printre cele mai vechi jocuri din lume, are diverse variante şi este practicat în toată lumea.

- **Scopul jocului** este ca fiecare jucător să aduca piesele lui in casa lui, si apoi sa le scoată de pe tabla de joc. Primul jucator care reuşeşte sa scoată din casa lui toate piesele care îi aparţin, in afara tablei de joc, va fi declarat câştigător.

- **Istoria**

Primele indicii ale unui joc asemănător vin din Egiptul antic unde se practica Senet, tot pe o tablă specială, mutările pieselor fiind decise prin mişcarea zarului. În Mesopotamia, un alt strămoş al tablelor se numea Jocul Regal al lui Ur. Iar în Iran, în oraşul Shahr-i Sokhta au fost descoperite semne ale unui joc asemănător, datând în jurul anului 3.000 î.H. În urma unor săpături s-au găsit zaruri şi 60 de piese [1]. Cel mai apropiat strămoş de actualul joc de table a fost găsit în oraşul iranian Jiroft unde s-a descoperit o masă de joc cu trei rânduri a câte 12 puncte, identică cu jocul roman „duodecim scripta".

În Roma antică se juca un joc intitulat Ludus duodecim scriptorum („Jocul celor 12 linii" sau „Jocul celor 12 semne") care avea loc tot pe o tablă cu trei rânduri a câte 12 puncte, iar piesele erau mutate pe toate cele trei rânduri, în funcţie de numărul dat prin rostogolirea zarului. O variantă ulterioară a

jocului a redus masa de joc la doar două rânduri şi a purtat numele Tabula (care înseamnă „tablă” sau „masă”), un joc similar celui modern, în care obiectivul era eliminarea tuturor pulurilor proprii înaintea adversarului. Erau folosite trei zaruri, iar pulurile adverse erau mutate în sens opus.

Burzoe prezintă conducătorilor indieni jocul nard

În secolul XI, poetul persan Ferdowsi îl declară pe Burzoe drept inventatorul jocului nard, undeva în secolul al VI-lea. Ferdowsi descrie o întâlnire între Burzoe şi conducătorii indieni care fac schimb de jocuri. Indienii îi prezintă fizicianului persan jocul de şah, iar acesta le arată jocul nard, practicat cu ajutorul unor zaruri create din fildeş şi arborele de tec.

Jocul de table propriu-zis a apărut prima dată în Europa în secolul XI, şi a fost repede adoptat de pasionaţii jocurilor de noroc. În 1254, Regele Franţei Ludovic al IX-lea a emis un decret prin care interzicea membrilor curţii să practice jocuri cu zaruri.[2] Cunoscut pentru pasiunea pentru şah, Alfonso al X-lea al Castiliei a descris în lucrările sale, Libro de los juegos şi El libro de ajedrez, dados e tablas regulile unor jocuri cu zaruri.

- **Inventar**
  - **Tabla**

    Tabla de joc pentru Table are 24 de triunghiuri in culori alternante numite puncte. Înaltimea unui punct poate atinge jumatatea inaltimii tablei. Punctele se noteaza de la 1 la 24. Numerotarea fiecarui jucator este diferita. Table este impărţită in 4 cadrane si fiecare cadran contine 6 puncte (triunghiuri). Fiecare jucator detine o zonă numita casă şi zone exterioare care sunt separate de o zonă ridicată numita bară (mijloc).

  - **Casa**

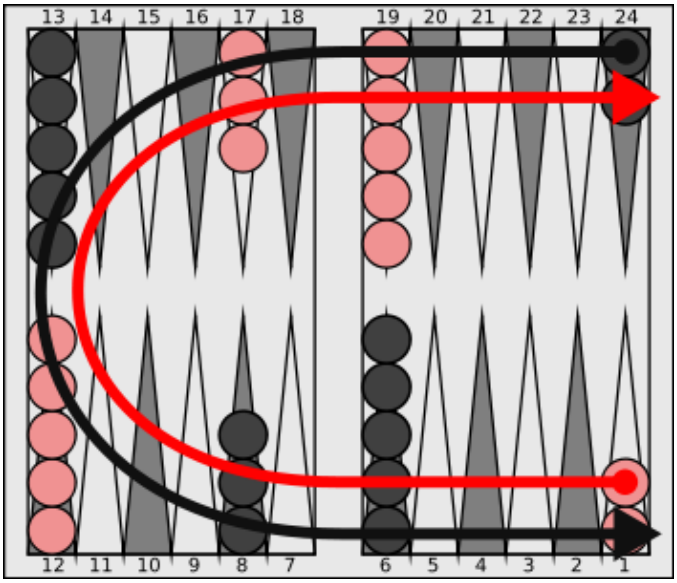
    Sase elemente consecutive in acelasi unghi al tablei se numeste casa jucatorului. Situarea casei depinde de reguli.

  - **Bara**

    Bara este secţiunea de mijloc a tablei de joc ce separa casele de zonele exterioare si odată ce o piesă ajunge pe ea (bară), rămâne afară din joc până ce poate intra pe un punct liber din casa adversarului prin aruncarea unui număr corespunzător la zar.

2.2. Regulament

- **Poziție de start**



Modul in care piesele sunt aranjate pe tablă la inceput se numeste pozitie de start. Două piese sunt pe pozitia 24 a fiecarui jucator, cinci pe pozitia a 13-a, trei pe pozitia a 8-a, si cinci pe pozitia 6 a fiecărui jucător. Direcţia de joc este din casa oponentului spre zona lui exterioară, spre zona proprie exterioară - până în casa proprie.

- **Startul jocului**

Fiecare jucator va lua câte un zar, și îl va arunca pe tabla de joc. Se analizează zarurile, jucătorul al cărui zar arată valoarea cea mai mare, va face prima mișcare. Dacă se întramplă să fie acelaș număr pe ambele zaruri, jucatorii arunca zarurile din nou, până când numerele vor fi diferite. În unele jocuri, jucătorii vor dubla automat miza în cazul în care amândoi nimeresc aceleaș numere.

- **Mutarea pieselor**

Fiecare jucator va mișca piesele conform celor 2 zaruri aruncate de către el. El va muta piesele in corcondanță cu numerele indicate de zaruri. Sa presupunem că zarurile indica 4-2, el va putea muta piesa 6 spatii sau o piesa 4 spatii si următoarea 2 spații. Trebuie sa luați in considerare că mutarea unei singure piese presupune defapt 2 mutari, fiecare mutare trebuie sa corespunda cu numarul indicat pe zar.

- **Dublele**

Daca pe ambele zaruri este indicat acelasi numar, de exemplu 4-4 sau 5-5, se va numi dubla, iar cel care a aruncat zarurile nimerind dubla va putea face 4 mutari in loc de doua. Deci daca va nimeri o dubla de 3, 3-3 va putea muta de 4 ori 3 spatii, indiferent ca muta o piesa sau 4.

Jucatorii arunca si joaca alternativ in timpul jocului, exceptie facand cazul in care unul dintre jucatori nu poate sa faca o miscare legala, insemnand ca va trebui sa renunte la mutarea sa in favoarea oponentului sau.

- **Marcarea spațiilor**

Un jucator poate marca un spatiu si sa-l detina daca are pozitionat in acel spatiu 2 sau mai multe piese, odata ce detine acel spatiu oponentul sau nu mai are voie sa se opreasca in acel spatiu indiferent ca face o mutare simpla folosinduse doar de unul dintre numerele aratate de zaruri, sau se foloseste de combinatia acestora si indiferent de numar, prima mutare pica pe un spatiu detinut de oponent.

- **Blocajul**

Un blocaj este atunci cand un jucator detine 6 spatii consecutive , astfel oponentul sau este prins dincolo de blocaj si nu poate trece de acesta deoarece cel mai mare numar pe zar este 6 si el detine 6 spatii.

- **Descoperire**

O singura piesa pe un spatiu este considerata descoperita, astfel daca in procesul de mutare al pieselor treceti sau mutarea se termina pe spatiul unde are oponentul piesa descoperita aceasta va fi inlaturata de pe tabla. De explu nimeriti 4-2 daca piesa se afla la 3 spatii distanta nu veti putea captura piesa chiar daca ati trecut peste ea, trebuie sa nimeriti spatiul in care se afla cu unul din numere, deci daca piesa se afla la 2,4 sau 6 (suma celor doua zaruri) spatii distanta va putea fi eliminata). Cel caruia i se inlatura piesa va trebui sa o readuca inapoi pe tabla. Acesta nu va putea face nici o miscare pana cand nu isi introduce piesa pe tabla. Piesa poate fi introdusa inapoi pe tabla in casa oponentului pe spatiul corespunzator numarului de pe zar, daca oponentul detine insa acel spatiu piesa nu va putea fi reintrodusa in joc.

- **Intrarea de pe bară in joc**

O piesă poate intra de pe bară in joc dacă la următoarea aruncare unul dintre numere corespunde cu un punct liber din casa adversarului si neocupat de 2 sau mai multe piese ale acestuia. Dacă nu poti intra pentru ca ambele puncte corespunzătoare zarurilor sunt ocupate, iti pierzi rândul (stai pe bară o tură). Dacă un jucător are una sau mai multe piese pe bară acestea trebuie sa intre toate în joc şi abia apoi alte piese vor putea fi mutate. Odată ce toate piesele de pe bară

au intrat in joc, orice numar neutilizat al aruncării va putea fi folosit pentru a face o mutare cu orice piesă personală.

- **Inchiderea tablei**

Un jucator care a reusit sa ocupe toate cele 6 spatii din propria casa cu cate cel putin 2 piese pe fiecare spatiu, va putea zice ca a inchis tabla. Daca oponentul sau are inca piese in bara nu le va putea reintroduce decat dupa ce va fi eliberat unul dintre spatii (va fi cel mult o piesa pe spatiu). Cel care a inchis tabla va continua sa runce cu zarurile si sa joace practic de unul singur pana se va elibera unul dintre spatii, astfel oponentul sau va avea sansa sa nimereasca numarul spatiului cu unul dintre zaruri.

- **Scoaterea pieselor de pe tabla de joc**

Scoaterea pieselor de pe tabla de joc este pasul final al acestui joc când iti vei scoate toate piesele din casă ,dar nu poţi incepe acest proces până când toate cele 15 piese personale au ajuns aici (în casă). După ce toţi "soldatii" tai sunt in casă poţi începe sa îi scoţi afară corespunzător numerelor ce cad la zaruri. Trebuie sa utilizezi intreg numărul dat , daca dai un 5 si nu ai nici o piesa pe poziţia 5 sau 6, trebuie sa scoţi o piesă afară de pe cea mai apropiată pozitie faţă de numărul dat (4, 3, 2, 1). Dacă dai un 5 şi nu ai nici o piesă pe pozitia 5 dar ai o piesă pe punctul 6, trebuie să muţi această piesă 5 puncte până pe punctul 1. Nu eşti obligat să scoţi o piesa afară daca ai altă mutare legală care poate fi utilă atunci când oponentul are piese pe bară sau inca detine un punct in casa ta. Dacă oponentul iţi loveşte o piesă in timp ce tu eşti in timpul procesului de scoatere afară a pieselor, trebuie sa introduci piesa in joc si sa o muţi de jur imprejurul tablei de joc până ce ajungi cu ea in casă si abia apoi poti continua procesul de scoatere afară a pieselor. Primul jucător care işi scoate afara toate cele 15 piese este câştigătorul jocului.

- **Probleme cu zarurile**

Se obisnuieste ca zarurile sa fie aruncate in partea dreapta a tablei de joc. Ambele zaruri trebuie sa ramana in interiorul acestei jumatati, daca se intampla ca unul din zaruri sa sara peste tabla, sau in partea celalta a tablei sau chiar sa se opreasca inclinat undeva pe colturi, se considera o problema cu zarurile astfel jucatorul va fi nevoit sa arunce din nou zarurile.

- **Dublarea**

  Jocule de Table este jucat la o miză per punct prestabilită de jucatori. Fiecare joc incepe de la un punct. În cursul jocului, un jucător care consideră că are un avantaj clar poate propune dublarea mizei. Poate face acest lucru numai la inceputul propriei ture şi înainte de a da cu zarul.

  Un jucator căruia i s-a oferit un joc dublu poate refuza, caz in care jocul se termină si el pierde doar un punct. Altfel, el trebuie sa accepte dublarea si jocul va continua cu o miză mărită. Un jucător care accepta dubla devine stăpânul zarurilor (sau al cubului de dublare) si numai el poate propune următoarea dublă (dublarea mizei de joc).

  Dublările consecutive in cadrul aceluiaşi joc se numesc redublări. Dacă un jucător refuză o redublare, el trebuie sa platească numărul de puncte ce reprezinta miza ce se dorea a fi dublată. Altfel, el devine noul stăpân al zarurilor şi jocul continuă cu o miză dublă fată de cea anterioară. Nu există nici o limită in ceea ce priveşte numărul de redublări ale mizei.

- **Regula Jacoby**

  Gammon-urile and backgamm-urile se pun ca fiind un singur joc dacă nici unul din jucători nu a oferit o dublă pe durata jocului. Această regulă măreste viteza de joc eliminând situaţiile în care un jucător evită dublarea jocului pentru a juca pentru un gammon. Regula Jacoby este folosită preponderent in jocul pe bani.

  Explicaţie:

  Gammon - Marţ

  Daca scoti toate piesele (15) inainte ca adversarul sa scoata vreuna, castigi un gammon, sa o partida dubla.

  Backgammon - Marţ tehnic

  Daca scoti toate 15 piesele inainte ca adversarul sa scoata vreuna de a lui si adversarul are cel putin o piesa pe bare sau in casa ta, castigi un backgammon, sau o partida tripla.

- **Regula Crawford**

  Dacă jucaţi un meci pana la un număr stabilit "n" si oponentul este înainte, daca ajunge la n-1 puncte corespunzător regulii Crawford nu i'i este permis utilizarea cubului de dublare in jocul următor (când el poate termina ajungând la n puncte).

- **Regula Holland**

În Jocurile post-Crawford cel ce a pierdut poate dubla numai după cel putin 2 aruncări de zar . Acest lucru este în avantajul liderului. Regula Holland este foarte rar folosită.

2.3. Variante ale jocului

- **În orientul mijlociu și asia centrală**

În Orientul Mijlociu şi Asia Centrală este practicat un joc asemănător cu tablele, intitulat ifranjiah (ceea ce înseamnă „Francii”). În Iran, denumirea jocului este takhte nard, iar în Israel şi în Orientul Mijlociu mai este cunoscut şi ca shesh besh (ceea ce înseamnă „şase şi cinci”).

Numele nardshir vine din două cuvinte persane: nard (Bloc de lemn) şi shir (Leu) care se referă la cele două tipuri de piese puse în joc.

În mai multe texte arabice se dezbate legalitatea şi moralitatea acestui joc. În secolul al VIII-lea şcolile musulmane de jurisprudenţă au anunţat interzicerea jocului, dar nu au reuşit oprirea creşterii popularităţii sale.

- **Mahbusa**

Mahbusa înseamnă „arestat”. Fiecare jucător începe cu 15 puluri dispuse pe cele 24 de puncte ale adversarului. Dacă piesa unuia dintre jucători ajunge pe acelaşi punct cu a unui adversar, este plasată peste a acestuia, care devine „arestată” şi nu mai poate fi mutată până când adversarul nu îşi mişcă propriul pul.

În unele ţări arabe este păstrat încă limbajul persan sau kurd la anunţarea cifrelor zarului, în locul limbajului arab modern

- **Narde**

În zona Podişului Iranului şi a Caucazului, mai ales în Iran, Armenia, Azerbaijan, Georgia şi Rusia este foarte răspândit jocul narde. Fiecare jucător posedă câte 15 puluri plasate în propria casă de 24 de puncte, diferenţa faţă de Mahbusa fiind interzicerea de a plasa o piesă proprie pe un culoar ocupat de o piesă a adversarului. Astfel, cea mai bună strategie de joc este crearea unor rânduri cât mai lungi de piese proprii pentru a-l împiedica pe adversar să se dezvolte.

Jocul este cunoscut sub numele 'Fevga' în Grecia şi 'Moultezim' în Turcia.

3. Dezvoltarea aplicatiei

3.1. Tehnologii utilizate

- **Java**

Java este un limbaj de programare de nivel înalt, dezvoltat de JavaSoft, companie în cadrul firmei Sun Microsystems. Dintre caracteristicile principale ale limbajului amintim:

- Simplitate: elimină supraîncărcarea operatorilor, moştenirea multiplă si toate "facilităţile" ce pot provoca scrierea unui cod confuz.
- Robustețe: elimină sursele frecvente de erori ce apar în programare prin eliminarea pointerilor, administrarea automată a memoriei si eliminarea fisurilor de memorie printr-o procedură de colectare a 'gunoiului' care rulează în fundal. Un program Java care a trecut de compilare are proprietatea ca la execuţia sa nu "crapă sistemul".
- complet orientat pe obiecte: elimina complet stilul de programare procedural
- usurinţă în ceea ce priveşte programarea în reţea
- securitate, este cel mai sigur limbaj de programare disponibil în acest moment, asigurând mecanisme stricte de securitate a programelor concretizate prin: verificarea dinamică a codului pentru detectarea secvenţelor periculoase, impunerea unor reguli stricte pentru rularea programelor lansate pe calculatoare aflate la distanţă, etc
- este neutru din punct de vedere arhitectural
- portabililtate:cu alte cuvinte Java este un limbaj independent de platforma de lucru, aceeaşi aplicaţie rulând, fără nici o modificare, pe sisteme diferite cum ar fi Windows, UNIX sau Macintosh, lucru care aduce economii substanţiale firmelor care dezvoltă aplicaţii pentru Internet;
- compilat si interpretat;
- asigura o performanţă ridicată a codului de octeţi
- permite programarea cu fire de execuţie (multitheaded)
- dinamicitate
- este modelat dupa C si C++, trecerea de la C, C++ la Java făcându-se foarte uşor.

- permite creearea unor documente Web îmbunătăţite cu animaţie şi multimedia.

Java - un limbaj compilat şi interpretat

În funcţie de modul de execuţie al programelor, limbajele de programare se împart în două categorii:

- interpretate: instrucţiunile sunt citite linie cu linie de un program numit interpretor şi traduse în instrucţiuni maşină; avantaj : simplitate; dezavantaje : viteza de execuţie redusă.
- compilate: codul sursă al programelor este transformat de compilator într-un cod ce poate fi executat direct de procesor; avantaj : execuție rapidă; dezavantaj : lipsa portabilității, codul compilat într-un format de nivel scăzut nu poate fi rulat decât pe platforma pe care a fost compilat.

Programele Java pot fi atât interpretate cât şi compilate.

Codul de octeţi este diferit de codul maşină. Codul maşină este reprezentat de o succesiune de 0 si 1; codurile de octeţi sunt seturi de instrucţiuni care seamănă cu codul scris în limbaj de asamblare. Codul maşină este executat direct de către procesor şi poate fi folosit numai pe platforma pe care a fost creat; codul de octeţi este interpretat de mediul Java si de aceea poate fi rulat pe orice platformă care foloseşte mediul de execuţie Java. Există 3 platforme Java furnizate de Sun Mycrosystems:

- Java Platform, Micro Edition (Java ME): pentru hardware cu resurse limitate, gen PDA sau telefoane mobile;
- Java Platform, Standard Edition (Java SE): pentru sisteme gen workstation, este ceea ce se găsește pe PC-uri
- Java Platform, Enterprise Edition (Java EE): pentru sisteme de calcul mari, eventual distribuite.

- **Pachetul de interfata Java Swing**

Crearea de interfeţe grafice constituie una din premisele absolut necesare în limbajele de programare moderne, dat fiind faptul că multe din aplicaţiile create au o destinaţie comercială. Asfel o interfaţa grafică poate contribui decisiv la succesul aplicaţiei pe piată.

Interfaţa in fond, asigură o interacţiune optimă dintre utilizator si aplicaţie si cum cele mai multe aplicaţii sunt comerciale ele trebuie să fie pe cât de performante in sine pe atât de bine trebuie sa interacţioneze cu utilizatorul. De multe ori rularea

aplicaţiei depinde de acţiuni întreprinse de un utilizator, ‚obiectele' grafice asteaptă să fie întreprinse acţiuni asupra lor pentru a lansa în execuţie o anumită secvenţa de cod.

Java pune la dispoziţia programatorului o gamă largă de elemente specifice elaborării si prelucrării interfeţelor grafice. Acestea sunt incluse in packetul AWT (Abstract Windowing Toolkit) si chiar înbunătăţite in Swing.

Atât Awt cât şi Swing asigură suportul Java pentru realizarea interfeţelor grafice prin intermediul unui set bogat de componente specifice ce au un mecanism robust de intrebuintare a evenimentelor generate de user, dar si suport pentru grafică şi instrumente de modelare a acesteia.

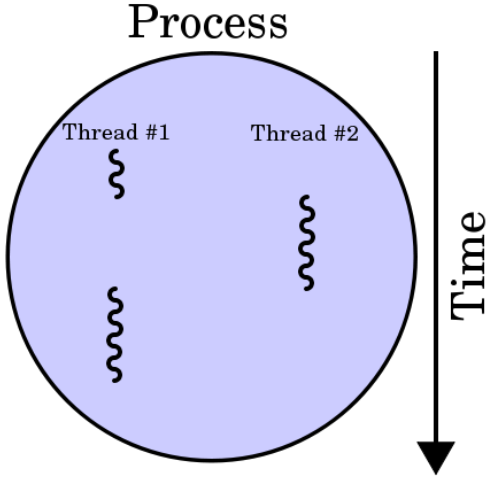
În plus Swing prezintă componente de un nivel mai inalt , de ex: vizualizările arborescente sau posibilitatea aplicarii unui sistem de customizare a design-ului prin plug-in+ul Look&Feel.

Swing oferă:

- portabilitatea applicaţiilor (si nu a applet-urilor)
- design-ul pur Java extinde raza de activităţi posibile pentru aceste componente
- suportă iconiţe si tool-tips
- flexibilitate crescută în manipularea design-ului componentelor in raport cu sistemul de operare
- posibilitatea modificării globale a anumitor caracteristici vizuale ale aplicaţiei.

- **Fire de executie**

Un fir de execuţie este o succesiune secvenţială de instrucţiuni care se execută în cadrul unui proces, sunt unităţi independente de execuţie a acestuia şi reprezintă o ale într-un program.



Un fir de execuţie are alocată o stivă în scopul inregistrării microprocesorului şi o intrare în lista de execuţie a componentei de alocare a execuţiilor. Firele de execuţie partajează toate resursele procesului, cum ar fi fişierele deschise sau memoria alocată

dinamic.

Fiecare fir de execuţie lucrează independent şi nu intră în contact cu alte fire de execuţie, decât în cazul în care acest lucru este realizat în mod specific de către programator.

Acest tip de programare, ce presupune existenţa mai multor fire de execuţie ce rulează în paralel, se numeşte programare concurentă. Cum avem un singur procesor şi mai multe thread de rulat, execuţia paralelă se simulează folosind diferite tehnici. În principiu se cunosc două asemenea tehnici care constituie de fapt politici de planificare a firelor:

- tehnica ce presupune că un fir de execuţie este lăsat să ruleze până când se termină sau până când ajunge în stare de aşteptare, moment în care este preluat un fir cu prioritate mai mare pregătit pentru executie.
- tehnica numită time-slicing ("felierea" timpului). Practic, fiecărui fir i se alocă un interval de timp în care poate ocupa procesorul. În felul acesta, fiecare fir ajunge, prin rotaţie, să ocupe câte puţin procesorul, iar utilizatorul are impresia că firele rulează în paralel.

Utilizarea firelor de execuţie este necesară în diverse aplicaţii. Firele de execuţie sunt utilizate în aplicaţii care necesită prelucrări de lungă durată, aplicaţii care urmăresc anumite evenimente şi aplicaţii de tip GUI care oferă posibilitatea întreruperii unei acţiuni.

De exemplu, încărcarea unui document este realizată de un fir de execuţie şi nu de firul principal, ceea ce permite utilizatorului interacţiunea cu fereastra care a lansat firul de execuţie.

- **NetBeans**

NetBeans este un proiect open-source, cu o bază de utilizatori foarte mare, o comunitate în creştere şi peste 100 de parteneri (în creştere!) din toată lumea. Sun Microsystems a fondat proiectul open source NetBeans în iunie 2000 şi continuă să fie principalul sponsor al proiectului.

Astăzi există două produse: NetBeans IDE şi platforma NetBeans.

NetBeans IDE este un mediu de dezvoltare - un instrument pentru programatori, pentru scrierea, compilarea, testarea, depanarea, proiectarea şi instalarea programelor. Este scris în Java - dar poate accepta orice limbaj de programare. De asemenea, există un număr imens de module pentru extinderea NetBeans IDE. NetBeans IDE este un produs gratuit, fără restricţii legate de modul de utilizare.

De asemenea, este disponibilă Platforma NetBeans; o bază modulară şi extensibilă, utilizată drept conector software pentru crearea aplicaţiilor desktop puternice. Partenerii ISV oferă Plugin-uri cu valoare adăugată, care se integrează uşor în platformă şi care pot fi utilizate, de asemenea, la dezvoltarea propriilor instrumente şi soluţii.

Ambele produse sunt open-source şi gratuite pentru uz comercial şi necomercial. Codul sursă este disponibil pentru reutilizare, conform Common Development and Distribution License (CDDL - Licenţa de distribuţie şi dezvoltare comună).

Cele mai utilizate componente ale Pantformei NetBeans sunt:

- **Containerul Modular Runtime**

  La fel cum serverele de aplicaţii cum ar fi GlassFish furnizează servicii de ciclu de viaţă a aplicaţiilor web, containerul Runtime NetBeans le furnizează aplicaţiilor Swing. Serverele aplicaţiilor înţeleg cum să compună module web, module EJB, şi aşa mai departe, într-o singură aplicaţie, la fel cum containerul runtime NetBeans înţelege cum să compună modulele NetBeans într-o singură aplicaţie Swing

  Modularitatea le permite dezvoltatorilor să îşi organizeze codul în module strict separate. Doar cele care au dependenţe explicit declarate pot folosi cod din pachetele expuse reciproc.

  Şi userii aplicaţiei finale beneficiază, deoarece aceştia pot instala module în aplicaţiile care rulează. Pe scurt, containerul Runtime NetBeans este un mediu de dezvoltare care înţelege ce este un modul, se ocupa de ciclul său de viaţă şi îl lasă să interacţioneze cu alte module din aceeaşi aplicaţie.

- **Sistemul de fişiere**

Când instalăm aplicaţii pe computer, installer-ul aplicaţiei copie fişiere şi foldere în sistemul de operare. Sistemul de fişiere NetBeans oferă funcţionalităţi similar aplicaţiilor Swing. Fiecare modul instalat într-o aplicaţie Swing poate instala fişiere

şi foldere în sistemul de fişiere al aplicaţiei, permitând celorlalte module din aplicaţie să le găsescă şi să le folosească.

De exemplu, dacă aveţi setări ale unui modul pe care doriţi să le puneţi la dispoziţia aplicaţiei, le puteţi înregistra în sistemul de fişiere. În acest fel, sistemul de fişiere NetBeans le permite modulelor din aplicaţie să comunice între ele.

- **Sistemul de ferestre**

Majoritatea aplicaţiilor au nevoie de mai mult de o fereastră. Codificarea unei interacţiuni bune între mai multe ferestre nu este atât de uşor pe cât pare. Din start, Platforma Netbeans oferă funcţionalităţi cum ar fi maximizare/minimizare, dock/undock, şi drag-and-drop pentru ferestrele din aplicaţie. Acest lucru este posibil deoarece ferestrele dintr-o aplicaţie fac parte din sistemul de ferestre Netbeans.

- **Managementul datelor**

În Platforma NetBeans, modelul JTree este complet diferit de modelul JList, cu toate ca acestea reprezinta aceleaşi date. Schimbarea între ele se poate face doar prin rescrierea modulului. API-ul NetBeans Nodes ne oferă un model generic pentru reprezentarea datelor. API-ul NetBeans Explorer & Property Sheet ne oferă numeroase componente Swing pentru afişarea nodurilor. Ca rezulatat, prin rescrierea unei singure linii de cod , putem schimba complet modul în care un model este reprezentat, fară a face modificări în modelul propriu-zis. În acest fel, putem prezenta date foarte rapid si eficient userilor.

3.2. Implementarea algoritmului

Pentru implementarea algoritmului s-au folosit urmatoarele clase:

- **Class Piece** extinde **JButton**

Contine:

- color : variabila de tipul enum colorType care poate fi: {nedefinita, white, black };
- icon: variabila de tipul ImageIcon,care contine textura piesei;
- iconW,iconB: variabila statica de tipul ImageIcon, cu care va fi instantiata variabila icon, in functie de tipul piesei
- int point: retine pozitia de pe tabla, pe care se afla piesa;
- MouseListener action: actiunea pentru mutarea piesei de pe o pozitie pe alta;
- constructor explicit care primeste ca parametru variabila color,de tip colorType, si in functie de ea instantiaza obiectul. Pentru a afisa piesele in forma rotunda am setat false cateva proprietati printre care:

setBorderPainted, setContentAreaFilled, setOpaque

```
public Piece(colorType color)
{
    this.color = color;
    switch (color)
    {   case white:
            icon = iconW;
            break;
        case black:
            icon = iconB;
            break;
    }
    setIcon(icon);
    setDisabledIcon(icon);
    setBorderPainted(false);
    setContentAreaFilled(false);
    setOpaque(false);
    setSize(icon.getIconWidth() - 1,icon.getIconHeight() - 5);
}
```

- **Class TPoint** extinde **Stack<Piece>**

Stiva de Piese, care se afla pe tabla.

- color: are aceeasi valoare cu piesele din stiva,in stiva pot fi numai un tip de piese(regula jocului). Daca stiva este goala, culoarea este nedefinita.
- location: retine coordonatele fizice de pe tabla; avem nevoie de coordonate cand adaugam o piesa in stiva.
- idx: retine indexul de pe tabla.
- maxPiece: o variabila statica de tipul int; avem nevoie pentru a nu depasi limitile grafice a tablei, la fel am suprascris functia add si pop din la clasa stack pentru a particulariza locatia fizica pentru fiecare piesa, si pentru a modifica pozitiile pieselor deja existente:

```
    @Override
    public boolean add(Piece p)
    {
        if(!isEmpty() && color != p.color)
            return false;
        else
            color = p.color;
        if(size() >= maxPiece)
        {
            int offSet = p.getHeight()/size();
            if(idx >12)
                offSet = -offSet;
            for (int i = 0; i < this.size(); i++) {
                Piece piece = this.elementAt(i);
                piece.setLocation(location.x, piece.getLocation().y +
offSet*i);
            }
        }
        else
        {
            if(idx <= 12)
            {
                location.y -= p.getHeight();
            }
            else
            {
                location.y += p.getHeight();
            }
        }
        p.setLocation(location);
        p.setPoint(idx);
        p.setVisible(true);
        if(!isEmpty())
            peek().setEnabled(false);
        return super.add(p);
    }
```

- **Class Table** extinde **Vector<TPoint>**
  - nrPoint: variabila statica de tipul int,care contine numarul punctelor de pe tabla;
  - tableH si tableW: variabile statice care contin dimensiunile grafice ale tablei;
  - midleGap, topGap: variabila statica care condine diferenta dintre dimensiunile grafice si virtuale ale tablei.
  - coordPlayerW,coordPlayerB: variabile de tip Point care retin locatia unde vor fi afisate zarurile pentru jucatori.
  - vInitPoz: vector de tipul int,care memoreaza pozitiile initiale ale pieselor pe tabla;
  - barW,barB: variabile de tipul stiva de tpoint in care vor fi plasate piesele eliminate de oponent;
  - outW,outB: variabile de tipul stiva de tpoint in care vor fi plasate piesele eliminate scoase din joc de player;
  - selectPiece,selectPoint: variabile de tipul int care vor retine piesa si punctul selectat de jucatorul curent, pe tabla, pentru a executa mutarea.
  - allInHouse: functie care primeste ca parametru o variabila de tip colorType, si returneaza true daca piesele de culoarea ceea sunt in zona tablei numita „casă”, false altfel;
  - existValidMove; functie care primeste ca parametru doua variabile , una de tip colorType si alta de tip int, si returneaza true, daca exista vreo mutare valida a pieselor de culoarea variabilei primite ca parametru, cu numarul de pozitii primite de la variabila de tip int. Aceasta functie este folosita dupa fiecare aruncare a zarurilor de un jucator.
  - initTable: functie fara parametri, este apelata la inceputul jocului pentru initializarea tablei de joc și setarea coordonatelor pieselor;
  - move: functie cu doi parametri de intrare, pozitia initiala, si pozitia finala; este apelata atunci cand un jucator a efectuat o mutarea a unei piese;
  - undoMove: functie cu doi parametri,care face exact inversul functiei move;
  - pushOut: functie apelata cand un jucator a scos o piesa de pe tabla;

- setPointCoord: functie apelata la inceputul jocului, care seteaza coordonatele punctelor de pe tabla;
- get: functia primeste ca parametru o variabila de tip int, si alta de tip colorType, si returneaza punctul cu numarul primit ca parametru, de pe tabla;
- validMove: functie apelata din functia move, care verifica daca mutarea pe care vrea sa o faca jucatorul este valida;
- validOut: la fel ca functia validMove, dar primeste ca parametru o singura variabila de tip int, si verifica daca piesa de pe pozitia respectiva poate fi scoasa afara ;

- **Class Dice** extinde **Jlabel** folosita pentru zaruri:
  - value:variabila de tipul int,retine valoarea zarului;
  - valid: variabila de tip boolean care este folosita pentru a sti daca un zar este valid,sau nu(un zar este valid daca a fost aruncat, dar nu a fost folosit pentru a face mutare)
  - side: vector static de tip ImageIcon in care vom retine texturele pentru cele 6 fețe ale zarului;
  - rand:variabila de tipul random, care este folosita pentru a genera o

```
side = new Vector<ImageIcon>(maxValue + 1);
side.setSize(maxValue+1);
for (int i = 1; i < side.size(); i++)
{
     ImageIcon img = new ImageIcon (Dice.class.getResource(
"resources/dice/dice"+i+".png"));
    side.set(i, img);
}
value = rand.nextInt(maxValue) + 1;
setIcon(side.get(value));
setLocation(p);
```

variabila aleatoare, astfel simulam aruncarea zarului.

  - maxValue: variabila statica de tipul int, care are valoare 6 pe tot parcursul jocului, folosita pentru a genera numere de la 1 la 6;
  - in constructor instantiem vectorul side astfel:
- **Class Player** extinde Thread

Clasa exdinde Thread,astfel avem posibilitatea sa pornim cate un thread pentru fiecare jucator, astfel facilitam functionarea fluida a aplicatiei;

- diceA,diceB: doua variabile de tipul Dice, (doua zaruri);
- color: variabila de tip colorType, care indica culoarea pieselor cu care joaca;
- wasRoll: variabila de tipul boolean, care ne indica daca jucatorul a aruncat zarurile;
- endTurn: variabila de tip boolean care ne indica daca a executat mutarea pieselor;
- gaveDouble: variabila de tipul boolean care indica daca jucatorul a dat dubla(la zaruri);
- pause: variabila de tip boolean care ne indica daca jucatoru se afla in asteptarea randului.
- pips: variabila de tip int,care retine numarul de pozitii cu care trebuie mutate toate piesele pentru a termina jocul;
- name: variabila de tipul string care retine numele jucatorului;
- haveValidMoves: returneaza true daca jucatorul mai are vreun zar si mutare valida;
- hideDice,show Dice: ascunde respectiv afiseaza zarurile si le seteaza ca fiind invalide, respectiv valide;
- roll: functia „arunca” zaurile;
- playerAction: functie care modifica un flag in clasa game(vezi class game)
- undoDice: functie care schimba statutul unui zar invalid, in valid; este apelata din functia undoMove din clasa Table;
- updatePips: recalculeaza valoarea retinuta de variabila pips;
- useDice: functie care schimba statutul unui zar valid, in invalid, daca a fosf facuta o mutare folosind acel zar;
- validDice: functie care primeste ca parametru o variabila de tip int, si returneaza true, daca jucatorul mai are zar nefolosit cu valoarea respectiva;
- run: functie suprascrisa din clasa threat, in care se ruleaza toate actiunile jucatorului;

```
    @Override
    public synchronized void run()
    {int sleepVal = 100;
        while(true) {
            if(!pause && !Game.instance.playerAction()){
            Game.instance.currentPlayerAction(true);
                if(Game.instance.stateWasChange) {
                    Game.instance.run();
                    sleepVal = 100;
                }else{
                    sleepVal = 50;
                    switch (Game.instance.currentState) {
                        case wasRoll:
                            if(Game.instance.gameHasBegin)
{Game.instance.setCurrentState(Game.gameState.selectMove);
                            }else{
Game.instance.setCurrentState(Game.gameState.firtstRoll);
                                diceB.setVisible(false); }
                            break;
                        case selectMove:
                            if(!haveValidMoves()){
                                Game.view.showMesage("No valid move");
                                Game.instance.changeCurrentPlayer();}
                            break;
                        case playerTurn:
                            if(endTurn) {
                                if(gotDouble) {
                                    endTurn = false;
                                    gotDouble = false;
                                    showDice();
Game.instance.setCurrentState(Game.gameState.selectMove);
                                }else{Game.instance.changeCurrentPlayer();
                    Game.instance.setCurrentState(Game.gameState.waitRoll); }
                                }else{
Game.instance.setCurrentState(Game.gameState.selectMove); }
                            break;
                        case moveOut:
                            if(endTurn)
                            {   if(gotDouble)
                                {   endTurn = false;
                                    gotDouble = false;
                                    showDice();
Game.instance.setCurrentState(Game.gameState.selectMove);
                                }else{Game.instance.changeCurrentPlayer();
Game.instance.setCurrentState(Game.gameState.waitRoll);
                                }}else if(pips != 0)
{Game.instance.setCurrentState(Game.gameState.selectMove); }
                            if(pips == 0) {
                                Game.view.showMesage("You win!"); }
                            break;
                    }
                }
                Game.instance.currentPlayerAction(false);
            }else{sleepVal = 1000;}
            try{ sleep(sleepVal);}
            catch (InterruptedException ex)
            {Logger.getLogger(iPlayer.class.getName()).log (Level.SEVERE, null,
ex);}
        }
    }
```

- **Class iPlayer** extinde Player

folosita pentru implementarea IA-ului;

in plus față de clasa player conține:

- turnes: variabila de tip vector de turn(vezi clasa turn);
- search: cauta prin mutarile posibile, si adauga in vectorul turnes prima mutare gasita care are costul mai mic decat costul ultimei mutari adaugate;
- move: functie care se apeleaza cand playerul trebuie sa faca o mutare;
- findMove: primeste ca parametru o variabila de tip Move, si verifica daca mutarea nu a fost deja generată;
- evaluateMove: calculeaza costul pentru o mutare;

```
public int evaluateMove(){
    int cost = 0;
    int hight = 3,medium = 2,low = 1,pointCost;
    for (int i = 1; i < Game.instance.table.size(); i++){
        TPoint point = Game.instance.table.get(i, color);
        if(point.color == color) {
            if(i < 7) {
                if(point.size() == 1)
                    pointCost = hight*(25 - i);
                else
                    pointCost = low*i;
            }
            else if(i < 18) {
                if(point.size() == 1)
                    pointCost = medium*(25 - i);
                else
                    pointCost = low * i;
            }
            else if(i <= 24) {
                if(point.size() == 1)
                    pointCost = low*(25 - i);
                else
                    pointCost = low * i;
            }else{
                pointCost = medium*i;
            }
            cost+= pointCost*point.size();
        }
    }
    return cost;
}
```

- in aceasta clasa avem la fel functia run care este suprascrisa si in care threadul ruleaza pana la sfarsiful jocului:

```
    @Override
    public void run()      {
        int sleepVal = 100;
        while(true){
            if(!pause && !Game.instance.playerAction()){
                Game.instance.currentPlayerAction(true);
                if(Game.instance.stateWasChange){
                    Game.instance.run();
                }else{
                    switch (Game.instance.currentState){
                        case waitRoll:
      Game.instance.setCurrentState(Game.gameState.wasRoll);
                            break;
                        case wasRoll:
                            if(Game.instance.gameHasBegin){
      Game.instance.setCurrentState(Game.gameState.selectMove);
                                playerAction();
                            }else{
      Game.instance.setCurrentState(Game.gameState.firtstRoll);
                                diceB.setVisible(false);
                            }break;
                        case selectMove:
                            if(haveValidMoves()){
                                move();
      Game.instance.setCurrentState(Game.gameState.playerTurn);
                            }else{
      Game.view.showMesage("No valid move");
                                hideDice();
      Game.instance.changeCurrentPlayer();
      Game.instance.setCurrentState(Game.gameState.waitRoll);
                            }break;
                        case playerTurn:
                            if(endTurn){
                                if(gotDouble){
                                    gotDouble = false;
                                    endTurn  = false;
                                    showDice();
      Game.instance.setCurrentState(Game.gameState.selectMove);
                                }else{
                                    Game.instance.changeCurrentPlayer();
      Game.instance.setCurrentState(Game.gameState.waitRoll);}
                            }else{
      Game.instance.setCurrentState(Game.gameState.selectMove);
                            }break;
                        case moveOut:
                            if(endTurn){
                                if(gotDouble){
                                    endTurn = false;
                                    gotDouble = false;
                                    showDice();
      Game.instance.setCurrentState(Game.gameState.selectMove);
                                }else{
                                    Game.instance.changeCurrentPlayer();
      Game.instance.setCurrentState(Game.gameState.waitRoll);}}
                            else if(pips != 0){
      Game.instance.setCurrentState(Game.gameState.selectMove);}
                            if(pips == 0){
      Game.view.showMesage( name + " win!");
      Game.instance.gameOver = true;}
                            break;
                        case showingMesage:
      Game.instance.setCurrentState(Game.gameState.hideMesage);
                            break;}}
      Game.instance.currentPlayerAction(false);
}
```

- **Class Move**
  - pozA,pozB: variabile de tipul int,care retin pozitiile intr-o mutare;
  - getDice: functie care returneaza diferenta pozitiilor, astfel obtinem valoarea zarului cu care a fost generata mutatia;
  - equal: functie care returneaza true daca obiectul de tip move, primit ca parametru este egal cu obiectul this;
- **Class Turn** exdinde **Stack<Move>**

Clasa ce contine o stiva cu elemente de tipul Move, astfel intr-un obiect de tipul Turn vom putea memora toate mutarile facut de un player la o singura aruncare a zarurilor. Clasa curenta mai contine urmatoarele variabile si functii:

- cost: variabila de tipul int, care va retine costul pentru murarile respective.
- equal: functie care primeste ca parametru un obiect de tipul clasei Turn, si returneaza true dacă variabila este egala cu obiectul this.

- **Class Game**

In clasa game vom instantia doua obiecte de tip Player, care sunt de fapt threaduri, si care vor interactiona prin aceasta clasa cu aplicatia.

Avem urmatoare le variabile si functii:

- instance: aceasta variabila statica va contine obiectul de tipul Game, care va fi instantiat la pornirea aplicatiei. Astfel vom putea accesa clasa game din orice alta clasa, in timpul rularii jocului;
- view: variabila statica de tipul BackgammonPCView cu ajutorul căreia vom putea accesa obiectele din interfata din orice alta clasa in timpul jocului;
- playerW,playerB: variabila statica de tipul Player, care va fi instantiata cu Player, sau iPlayer;
- threadSleep: variabila statica de tipul long care va fi folosita la oprirea temporara a threadurilor playerW si playerB;
- gameState: este o definita ca enumeratie, și poate avea valorile: {idle, firstRoll, waitRoll, wasRoll, waitMove, executeMove, moveOut, showMesage, showingMesage, hideMesage}
- setCurrentState: este o functie care primeste ca parametru o variabila de tipul gameState, si care incearca schimbarea starii curente a jocului daca ea nu a

fost deja schimbata, si daca schimbarea din starea curenta, intr-o stare noua este posibila

```
public synchronized boolean setCurrentState(gameState s){
    if(!stateWasChange && currentState != s){
        boolean canChange = false;
        switch(currentState){
            case firtstRoll:
                if(     s == gameState.waitRoll ||
                        s == gameState.showMesage)
                    canChange = true;
                break;
            case waitRoll:
                if(s == gameState.wasRoll ||
                        s == gameState.waitRoll ||
                        s == gameState.hideMesage)
                    canChange = true;
                break;
            case wasRoll:
                if(     s == gameState.waitMove ||
                        s == gameState.firtstRoll)
                    canChange = true;
                break;
            case waitMove:
                if(s == gameState.executeMove ||
                   s == gameState.moveOut ||
                   s == gameState.waitRoll)
                    canChange = true;
                break;
            case executeMove:
            case moveOut:
                if(s == gameState.waitRoll ||
                   s == gameState.showMesage ||
                   s == gameState.waitMove)
                    canChange = true;
                break;
            case showMesage:
                if(s == gameState.showingMesage)
                    canChange = true;
                break;
            case showingMesage:
                if(s == gameState.hideMesage)
                    canChange = true;
                break;
            case hideMesage:
                canChange = true;
                break;
        }
        if(canChange){
            stateWasChange = true;
            currentState = s;
        }
    }else{
        nextState(s);
    }
    return stateWasChange;
```

- run: este functia apelata de fiecare Thread, alternativ, atunci cand vine randul fiecaruia dintre playeri sa execute o actiune:

```
public synchronized void run(){
    stateWasChange = false;
    switch (currentState){
        case firtstRoll:
            if(playerW.wasRoll == playerB.wasRoll){
                if(playerW.diceA.value == playerB.diceA.value){
                    view.showMesage("Roll again");
                    changeCurrentPlayer();
                }else{
                    if(playerW.diceA.value > playerB.diceA.value){
                        setCurrentPlayer(Piece.colorType.white);
                        view.showMesage("First turn White");
                    }else{
                        setCurrentPlayer(Piece.colorType.black);
                        view.showMesage("First turn Black");
                    }gameHasBegin = true;
                }playerW.wasRoll = playerB.wasRoll = false;
            }else{
                changeCurrentPlayer();
                setCurrentState(gameState.waitRoll); }break;
        case waitRoll:
            view.RollButton.setEnabled(true);
            hideDice();break;
        case wasRoll:
            roll();
            view.RollButton.setEnabled(false); break;
        case executeMove:
            table.move();
            updatePipsValue();break;
        case moveOut:
            table.moveOut();
            updatePipsValue();break;
        case showMesage:
            view.MesageLabel.setVisible(true);
            if(gameOver){
                view.OkButton.setVisible(false);
                view.RollButton.setEnabled(false);
            }else{
                view.OkButton.setVisible(true);
                view.RollButton.setEnabled(false);
            }setCurrentState(gameState.showingMesage); break;
        case hideMesage:
            view.MesageLabel.setVisible(false);
            view.OkButton.setVisible(false);
            view.OkButton.setEnabled(false);
            setCurrentState(gameState.waitRoll);
            hideDice();break;
    }view.updateView();
    if(gameOver){ try {Thread.sleep(3000);
        } catch (InterruptedException ex) {
      Logger.getLogger(Game.class.getName()).log(Level.SEVERE, null, ex);
        }
        view.getApplication().exit(null); }
}
```

Functia run va fi apelata de fiecare data cand sa detectat o schimbare a starii curente a jocului.

Cand functia se apeleaza, valoarea care detecteaza schimbarea starii se modifica, cu false.

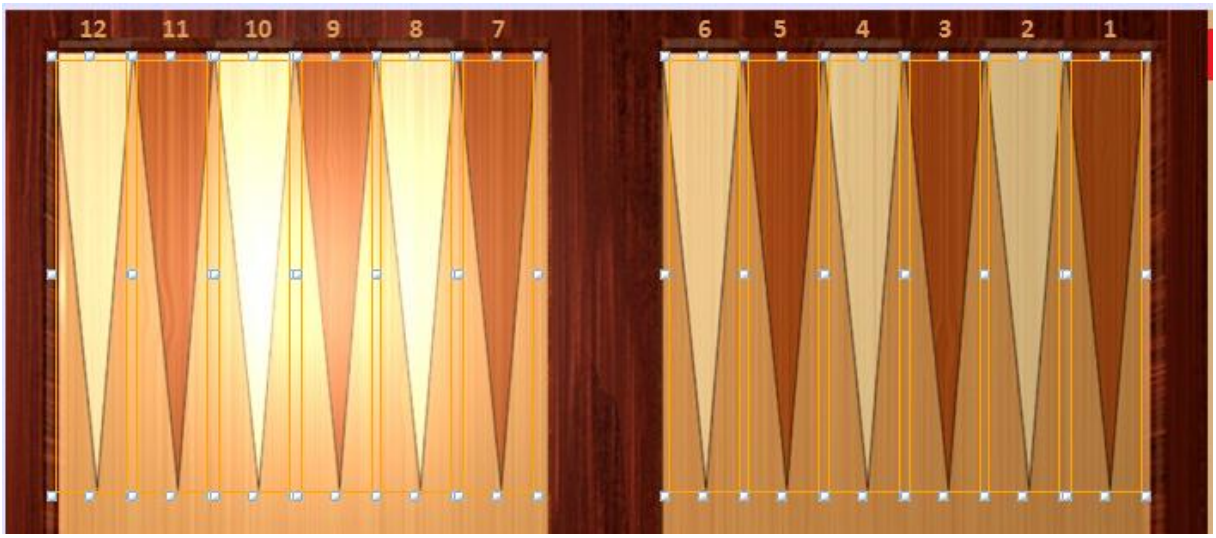
In functie de starea curenta se executa urmatoarele:

- firstRoll: se verifica daca ambii playeri au aruncat zarurile. Daca este adevarat, in cazul in numerele de la zarurile aruncate de playeri sunt egale, se afiseara un mesaj care informeaza jucatorii ca trebuie sa mai arunce inca cate o data zarurile, altfel o sa obtinem valoarea la zarul unu player mai mare decat valoare de la zarul celuilalt player, in functie de maxim vom alege cine face prima mutare.
- waitRoll: se seteaza vizibil butonul pentru aruncarea zarurilor, zarurile sunt ascunse;
- wasRoll: cand se ajunge la aceasta stare, se executa aruncarea zarurilor propriu zis;
- executeMove: se face mutarea, se recalculeaza valoare pips;
- moveOut: se face o mutare care scoate o piesa in afara tablei,se recalculeaza valoarea pips;
- showMesage: se face vizibil mesajul;
- showingMesage: nu executa nici o actiune;
- hideMesage: se ascunde mesajul si se seteaza valid buttonul pentru aruncarea zarurilor;

3.3. Implementarea interfeței

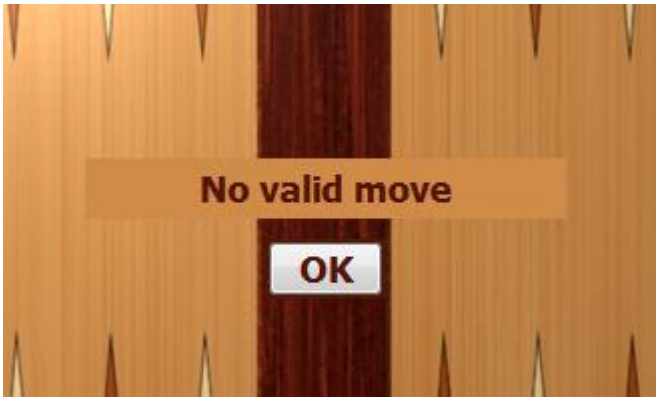
Pentru implementarea interfetei s-au folosit in mare parte controalele default puse la dispozitie de catre Netbeans cum ar fi : JFrame, JPanel, JButton, JLabel, JMenuBar etc. A fost insa nevoie de crearea unui buton nou, pentru a usura accesul pieselor pe tabla.

Astfel a fost creata clasa Piece care extinde JButton. Textura tablei a fost pusa pe un JLabel, peste care am adaugat cate un JPanel invizibil pentru fiecare punct de pe tabla. Fiecare JPanel are un raspuns pentru clickurile utilizatorului, astfel executam mutarile a pieselor pe tabla, ca raspuns in urma clikurilor, bineînțeles dacă mutarea este valida.

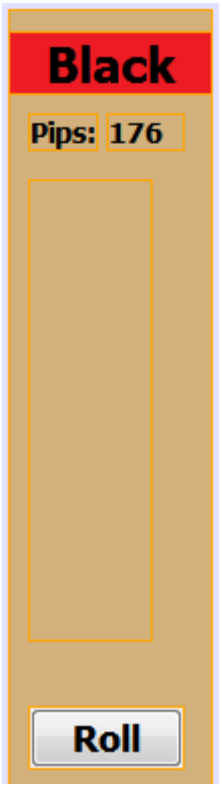


In mijlocul tablei avem doua controale:

- MesageLabel: de tip JLabel care este folosit pentru a informa jucatorii despre anumite avenimente din timpul jocului.
- OkButton:de tip JButton care ascunde mesajul de informare.



In partea laterala dreapa a tablei, avem un panou de informare a jucatorilor.



El este de asemenea creat cu un JPanel pe care avem adaugate mai multe detalii cum ar fi:

- RollButton: buton pentru aruncarea zarurilor;
- WhiteLabel,BlackLabel: sunt folosite pentru a afisa numele jucatorilor, dar si pentru a informa playerii in timpul jocului, al cui rand este. Lucrul acesta este afisat cu background rosu sau verde la acest JLabel, in functie de starea curenta a fiecarui player (roru = asteapta, verde = joaca).
- PointFinalWhite, PointFinalBlack: sunt doua controale invizibile de tip JPanel, exact ca cele folosite mai sus la tabla. Aceste controale sunt folosite cand jucatorul incepe sa scoata piese de pe tabla, dând click pe panou sau din dreapa, acesta va primi ca raspun comanda si va executa mutarea piesei de pe tabla.
- PipsWhiteVal,PipsBlackVal: sunt doua controale de tip JLabel folosite pentru afisarea valorii pips, pentru fiecare jucator.

4. Manual de utilizare

La pornirea aplicatiei apare interfața jocului. (Fig 3.1)

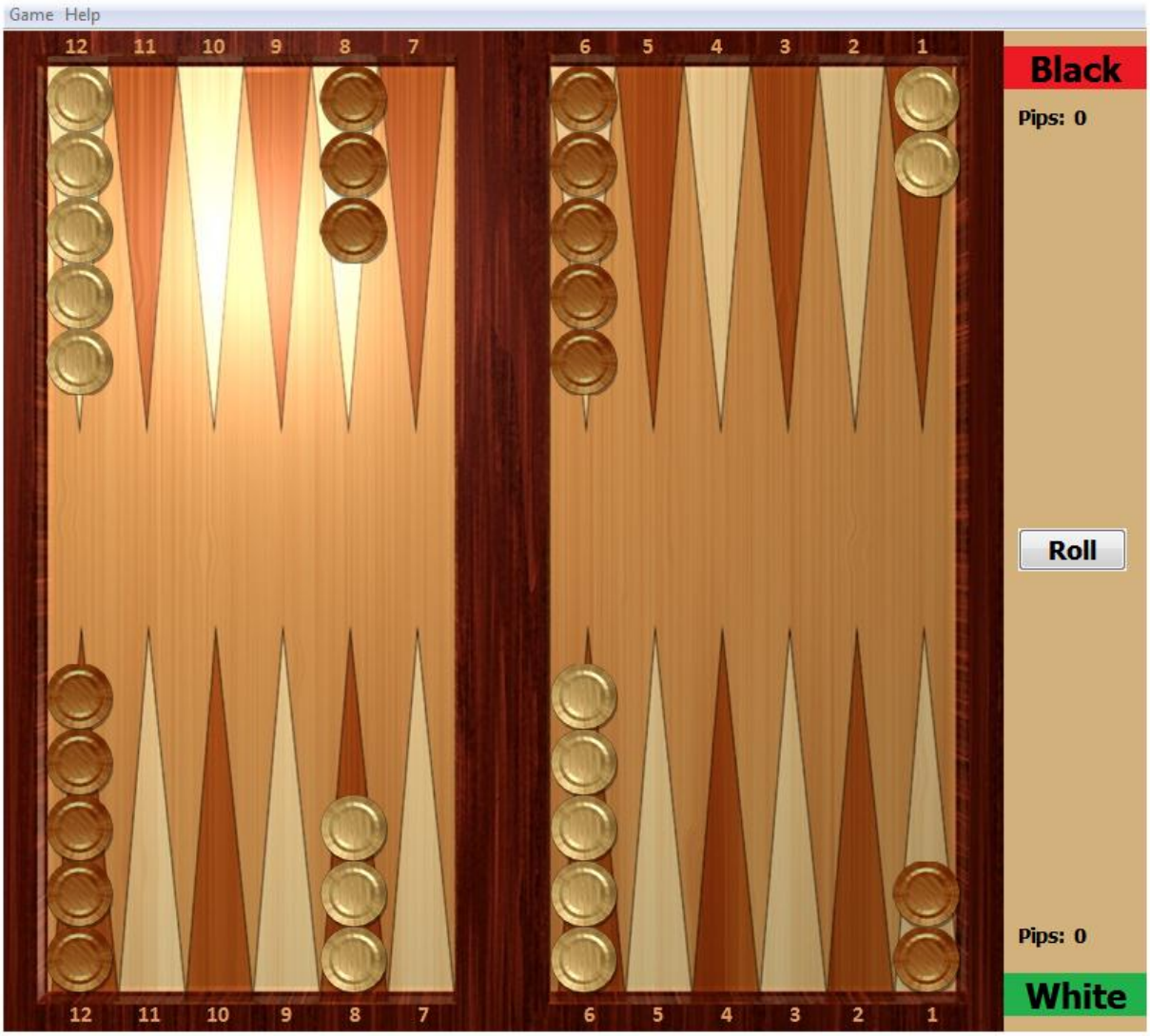


Fig3.1

In partea dreapta apare bara de informare asupra jocului. Ea contine in partea de sus si jos numele jucatorilor. In functie de culoarea acestui nume, jucatorul va sti daca este randul lui sau nu sa joace(daca este rosu,asteapta randul, daca este verde,joaca).

Lânga numele jucatorilor apare si valoarea Pips care informeaza jucatorul asupra numarului de care mai are nevoie pentru a termina jocul.

La mijlocul acestei bare gasim butonul Roll un urma actiunii caruia se executa aruncarea zarurilor de catre jucatorul curent.

Din bara de meniu sau cu ajutorul tastelor scurte F2, F3, utilizatorul poate alege modul de joc.(Fig.4.2)

Implicit este selectat modul de joc cu calculatorul. Dupa una din actiunile userului jocul se va restarta.

Fig.4.2

Initial jucatorul trebuie sa arunce un zar pentru a alege care din playeri va face prima mutare.

Pentru ca jucatorul sa arunce zarul trebuie apasat butonul Roll din bara din dreapta.

Dupa ce se arunca pentru prima data zarurile, apare un mesaj de informare. (Fig. 4.3)

Butonul Roll pe timpul afisarii mesajului de informare va fi dezactivat. El se va reactiva dupa ce userul va confirma mesajul.

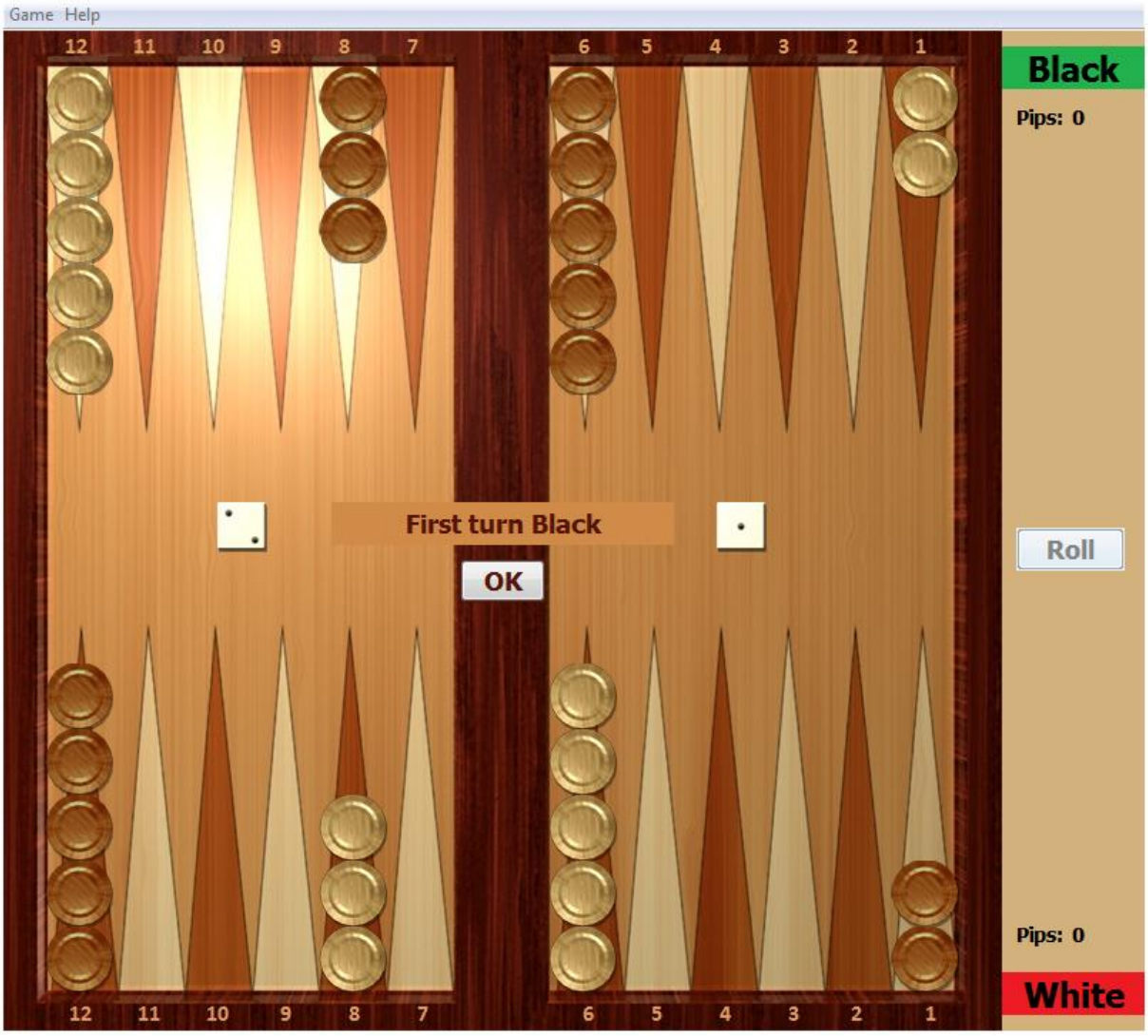


Fig.4.3

Dupa confirmarea mesajului prin apasarea butonului OK incepe jocul propriu zis.

Jucatorul care a avut valoarea zarului la prima aruncare mai mare va da zarul primul.

Dupa aruncare, zarurile vor aparea in partea stanga pentru playerul de sus-Black(Fig.4.4),

sau in partea dreapta pentru celalalt jucator.

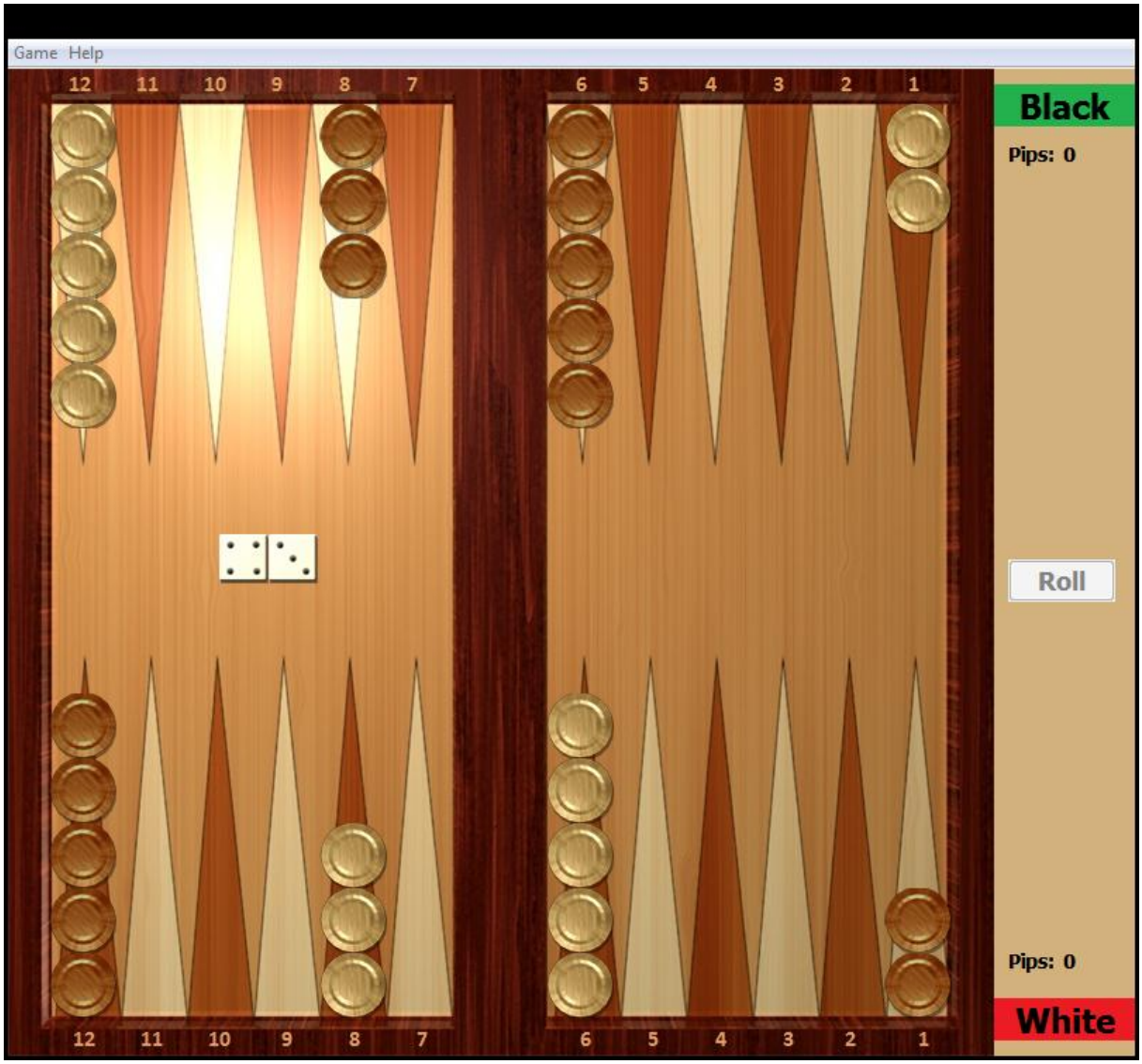


Fig.4.4

Dupa o mutare, zarul care a fost folosit pentru a executa acea mutare dispare, ramane afisat doar zarul nefolosit. Dupa ce userul a facut o mutare el poate sa faca urmatoarea mutare fara sa mai dea click din nou pe piesa din acelas punct. Ea ramane selectata atita timp cat jucatorul nu selecteaza o noua piesa. Asta ajuta la economisirea timpului cand avem duble de exemplu, si userul vrea sa mute 4 piese dintr-un punct in altul. In mod normal jucatorul trebuie sa dea 8 clickuri,diferite 2 cate 2 de pe un punct pe altul, pentru a face mutarile. Altfel e destul sa dea un click pe punctul din care vrea sa ia piesele, si inca 4 pe alt punct diferit de primul.

Cand o piesa este scoasa pe bara ea va aparea in mijlocul tablei de joc(Fig.4.5).

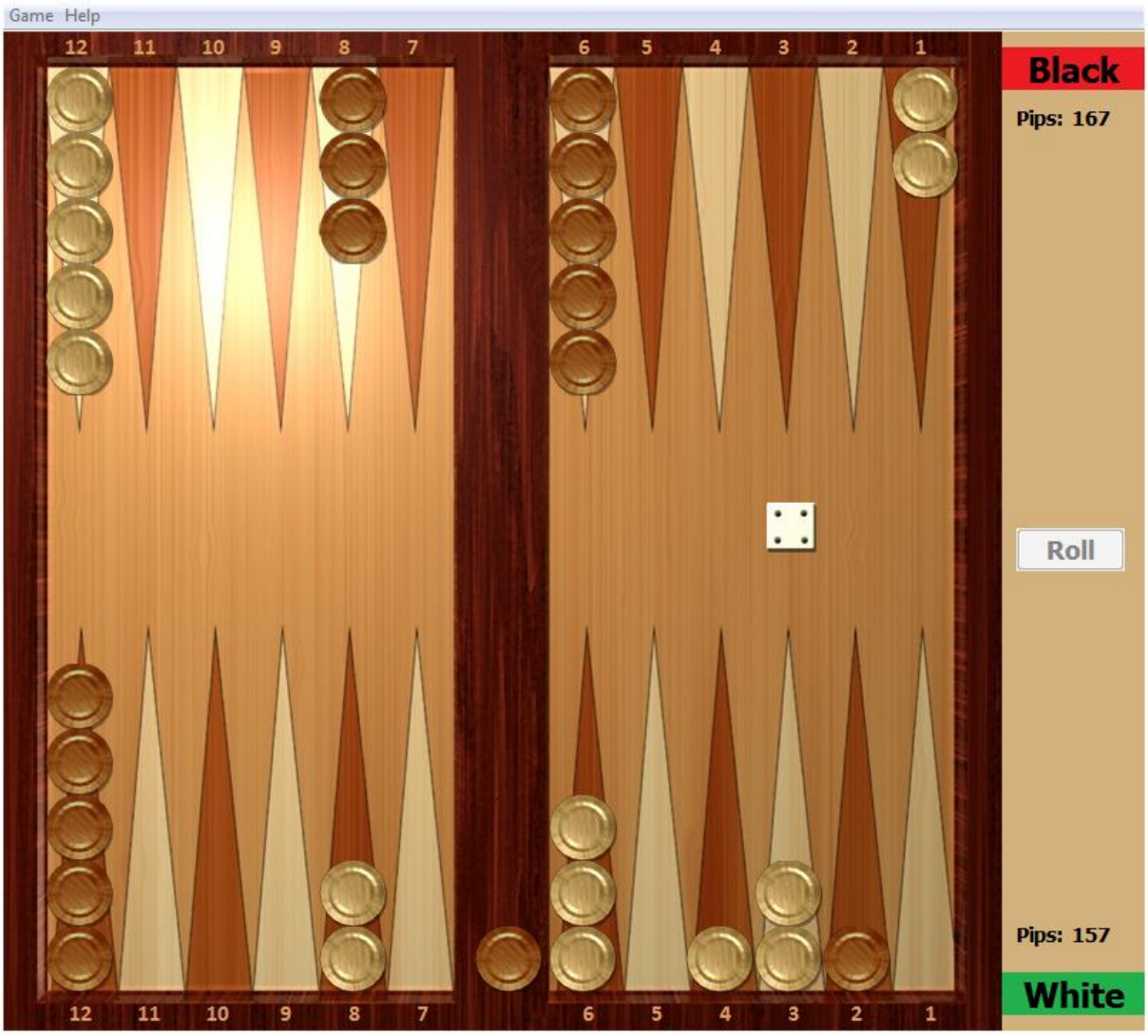
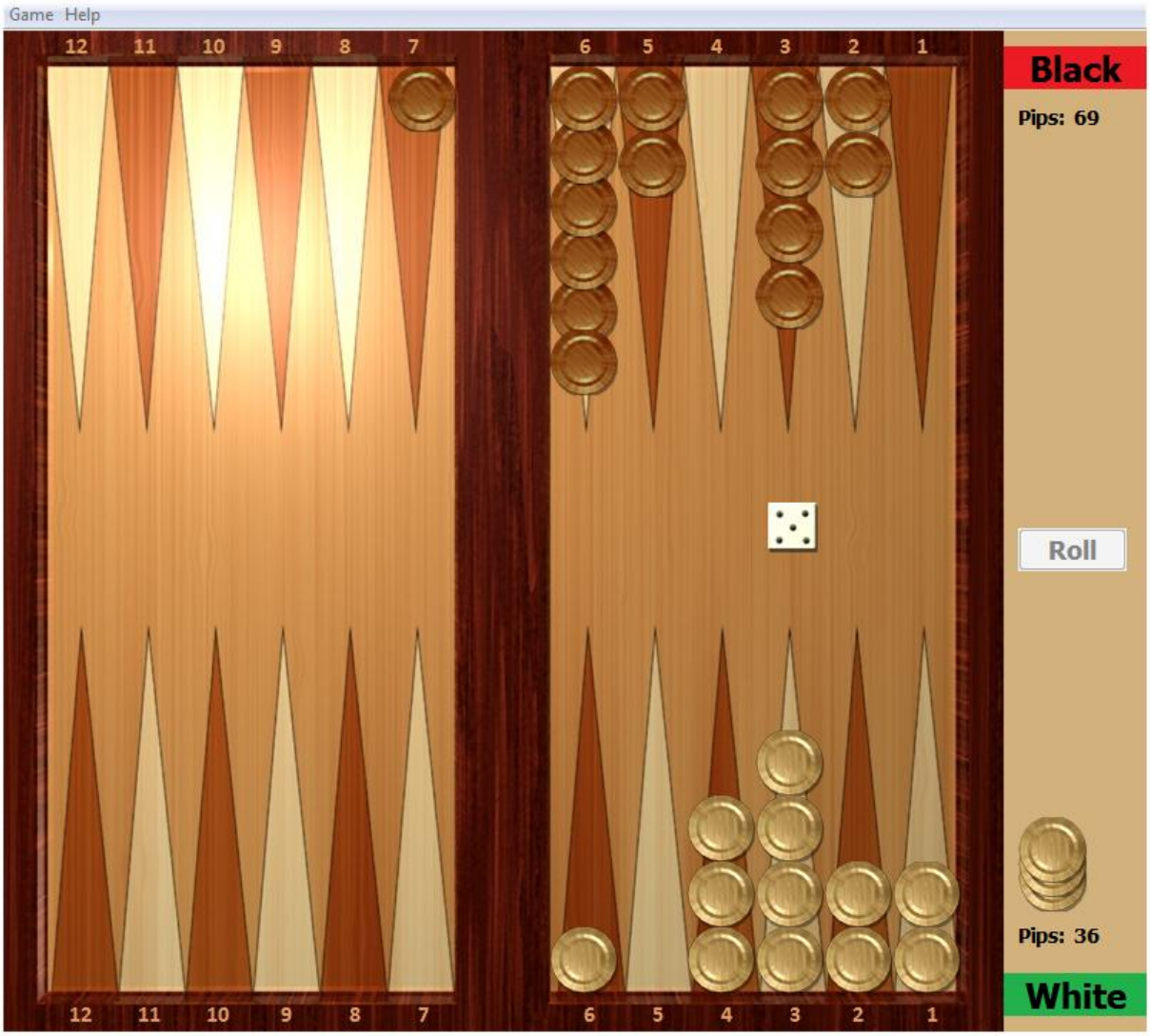


Fig.4.5

Cand un jucator are toate piesele in casa el poate sa scoata cate o piesa de pe tabla.

Pentru aceasta mutare jucatorul trebuie sa dea click pe piesa care doreste sa o selecteze, si apoi click pe bara de informatii.



Dupa ce unul din jucatori scoate toate piesele de pe tabla, apare un mesaj de felicitare, apoi aplicatia se restarteaza in același mod.

Bibliografie

1. **David M.Bourg,** *AI for Game Developers*,
   O'Reilly Media, 2004.
2. **Eckel Bruce,** *Thining in Java-3rdEdition*,
   Editura Prentice-Hall, 2002.
3. **Davidoviciu Adrian,** *Sisteme de inteligență artificială,*
   Editura Academiei Române, Bucuresti, 1991.
4. **Georgescu Horia,** *Introducere în universul JAVA*,
   Editura Tehnica, Bucuresti, 2002.
5. **Norvig Peter,** *Artificial Inteligence: A Modern Approach*,
   Prentice Hall, 2009.
6. http://java.sun.com/
7. http://netbeans.org/
8. http://wikipedia.org/
9. http://www.bkgm.com/
10. http://profs.info.uaic.ro/~acf/java/