Inteligenţă artificială
Laboratorul 8

Jocuri. Algoritmul minimax

1. Obiectiv

Obiectivul acestui laborator este implementarea unui joc simplu între om și calculator, folosind algoritmul minimax.

2. Jocuri

Un *joc* reprezintă o succesiune de decizii (acțiuni) luate de părți ale căror interese sunt opuse. Jocurile pot fi clasificate după:

- *numărul de jucători:* 1, 2, *n* (*n* nu înseamnă că participă efectiv *n* jucători, ci că regulile permit împărțirea jucătorilor în *n* mulțimi disjuncte, astfel încât jucătorii din fiecare mulțime să aibă interese comune);
- *natura mutărilor:* există jocuri cu mutări libere (mutarea este aleasă conștient, dintr-o mulțime de acțiuni posibile, de exemplu șahul) și jocuri cu mutări aleatorii (mutarea este dictată de un factor aleatoriu – zaruri, cărți de joc, monede etc.). Pot exista jocuri cu ambele tipuri de mutări;
- *cantitatea de informație* de care dispune un jucător (un jucător de șah vede configurația completă a pieselor adversarului, pe când un jucător de cărți nu are acces la configurațiile celorlalți jucători).

Există două motive pentru care jocurile sunt un bun domeniu de explorat pentru inteligența artificială:

- Au o organizare clară, în care e foarte ușor de măsurat succesul sau eșecul;
- Nu necesită cantități mari de cunoștințe inițiale. Multe jocuri sunt concepute astfel încât să poată fi rezolvate prin metode de căutare din starea inițială până în poziția câștigătoare. Bineînțeles, acest lucru este adevărat numai pentru jocurile simple. Un contraexemplu este jocul de șah, care presupune un arbore de căutare cu factorul de ramificare 35, adică în fiecare moment există (în medie) 35 de mutări disponibile. Având în vedere că un jucător face (tot în medie) aproximativ 50 de mutări, putem trage concluzia că pentru un joc trebuie examinate $35^{2 \cdot 50} = 2{,}55 \cdot 10^{154}$ variante.

Din cauza numărului mare de posibilități, trebuie să existe euristici de căutare. În acest scop, se folosesc proceduri de generare și test. Procedurile de generare pot fi optimizate astfel încât să se

genereze numai mutări (sau căi) bune. De asemenea, procedurile de test trebuie să recunoască mutările bune, pentru ca acestea să fie explorate primele.

Există jocuri pentru care avem și alte tehnici, în afară de căutare. De exemplu, în șah („drosophila IA", după cum îl considera Alexander Kronod), deschiderile și finalurile sunt deseori memorate sub formă de modele în baze de date. Aici pot fi combinate tehnicile de căutare cu tehnici mai directe de genul celor menționate mai sus.

Când dimensiunea arborelui de căutare este prea mare, se utilizează o *funcție de evaluare statică*, care folosește informațiile disponibile la un moment dat pentru a evalua pozițiile care au probabilitate mare de a conduce la atingerea scopului – câștigarea jocului. În absența informațiilor complete, se alege euristic poziția cea mai promițătoare. De multe ori, este greu de găsit o funcție bună. De fapt, „inteligența" cu care joacă calculatorul depinde în mare măsură de această funcție de evaluare.

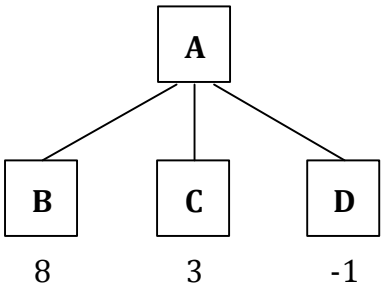
3. Algoritmul de căutare *minimax*

Minimax este un algoritm de căutare limitată în adâncime (*depth-first*, *depth-limited*). Se pleacă de la poziția curentă și se generează o mulțime de poziții următoare posibile. În acest scop, structura de date cel mai des folosită este arborele. Se aplică funcția de evaluare statică și se alege poziția cea mai bună.

Se presupune că jocul este de sumă nulă, deci se poate folosi o singură funcție de evaluare pentru ambii jucători. Dacă $f(n) > 0$, poziția n este bună pentru calculator și rea pentru om, iar dacă $f(n) < 0$, poziția n este rea pentru calculator și bună pentru om. Funcția de evaluare este stabilită de proiectantul aplicației.

Pentru aplicarea algoritmului, se selectează o limită de adâncime și o funcție de evaluare. Se construiește arborele până la limita de adâncime. Se calculează funcția de evaluare pentru frunze şi se propagă evaluarea în sus, selectând minimele pe nivelul minimizant (decizia omului) și maximele pe nivelul maximizant (decizia calculatorului).

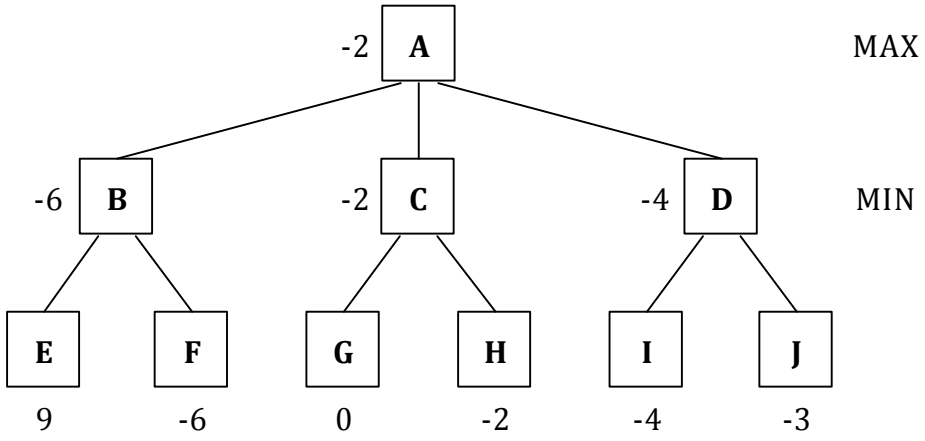
Să considerăm mai întâi căutarea pe un singur nivel (engl. "ply", pliu, cută, strat; în teoria jocurilor, această denumire sugerează numărul de niveluri în arbore care se studiază):



Primul nivel este un nivel maximizant, de aceea valoarea rădăcinii arborelui este maximul dintre valorile fiilor. Aici, valoarea lui *A* este 8, deoarece valoarea lui *B* (8) este valoarea maximă.

Procedura minimax presupune căutarea alternativă pe niveluri maximizante și niveluri minimizante. Dacă vom cerceta două niveluri, primul nivel va fi maximizant, iar al doilea minimizant.

Căutarea pe două niveluri este de forma:



În nodul *B*, vom avea valoarea minimă a fiilor săi (–6). La fel în *C* şi *D*. În *A*, vom prelua valoarea maximă dintre *B*, *C* şi *D*, adică –2.

Aceste valori numerice sunt date de funcția de evaluare statică. Deoarece aceasta este deseori imprecisă, e importantă căutarea pe cât mai multe niveluri, pentru a crește numărul de posibilități avute în vedere, și deci și șansele de a determina mutarea optimă.

Pseudocodul algoritmului minimax este următorul:

```
Minimax (board, depth)
  if depth = depthBound then
    return StaticEvaluation(board)
  else if IsMaximizingLevel(depth) then
    foreach child c of board
      compute Minimax(c, depth+1)
      return maximum over all children
  else if IsMinimizingLevel(depth) then
    foreach child c of board
      compute Minimax(c, depth+1)
      return minimum over all children

Initialize(depthBound)
Minimax(currentBoard, 0)
```

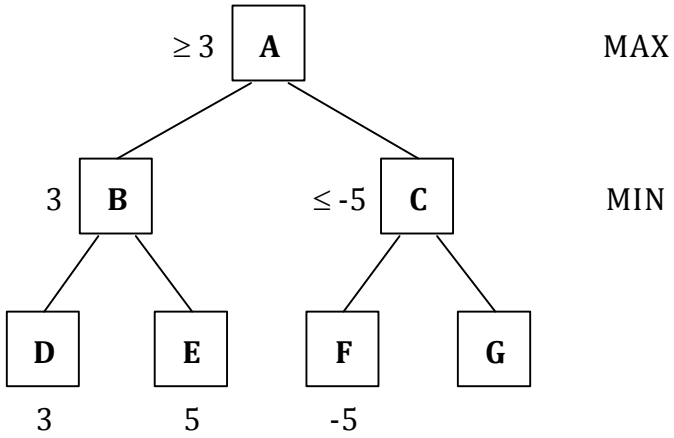
4. Retezarea alfa-beta

Algoritmul minimax determină un proces de căutare în adâncime. O cale de o anumită lungime este explorată şi apoi se aplică funcția de evaluare statică asupra pozițiilor jocului în ultimul pas al căii, iar valoarea calculată este trimisă la nivelul imediat superior pe cale şi aşa mai departe până se ajunge în rădăcina arborelui.

Eficiența procedurilor *depth-first* poate fi mărită prin tehnici *branch-and-bound*, în care pot fi abandonate mai repede soluțiile parțiale care sunt în mod evident inferioare altor soluții deja cunoscute. După cum minimax necesită gestionarea straturilor minimizante şi maximizante, şi strategia *branch-and-bound* trebuie modificată pentru a include două limite, una pentru fiecare

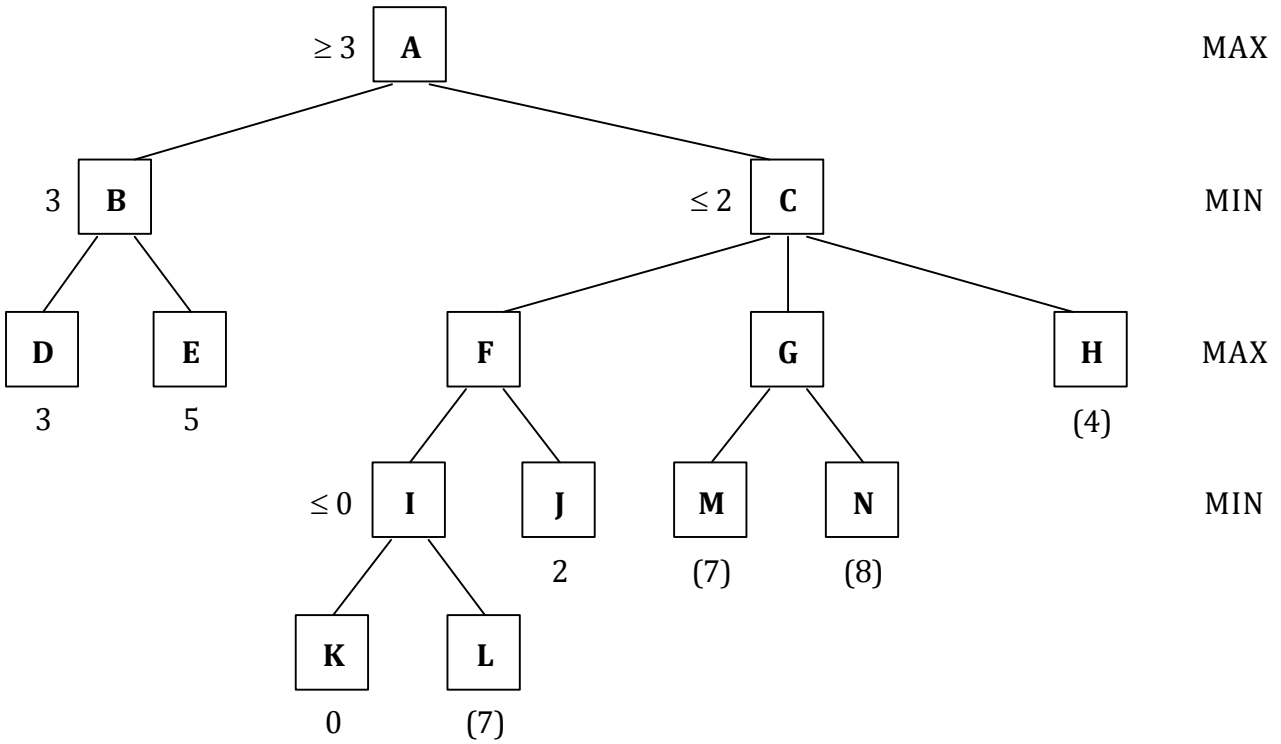
jucător. Această strategie modificată se numește *retezare alfa-beta* (engl. "alpha-beta pruning"). Ea necesită menținerea a două valori prag, una reprezentând limita inferioară a valorii unui nod care realizează o maximizare (*alfa*), iar cealaltă limita superioară a valorii unui nod care realizează o minimizare (*beta*). Retezarea alfa-beta este o optimizare a algoritmului minimax, care întrețese generarea arborelui cu propagarea valorilor. Unele valori obținute în arbore furnizează informații privind redundanța altor părți, care nu mai trebuie generate. În acest caz, se generează arborele în adâncime, de la stânga la dreapta și se propagă valorile finale ale nodurilor ca estimări inițiale pentru părinții lor. De fiecare dată când *alfa* ≥ *beta*, se oprește generarea nodurilor descendente.

În exemplul următor, după ce examinăm nodul *F*, știm că adversarul (omul) are garantat un câștig de –5 sau mai puțin în *C*, de vreme ce adversarul este jucătorul minimizant. Dar mai știm de asemenea că noi (calculatorul) avem garantat un câștig de 3 sau mai mult în nodul *A*, care poate fi obținut dacă mutăm în *B*. Orice altă mutare care produce un câștig mai mic de 3 este mai proastă decât mutarea în *B* și poate fi ignorată. Deci, după ce am examinat doar nodul *F*, suntem siguri că mutarea în *C* este greșită (va fi mai mică sau egală cu –5), indiferent de câștigul în *G*. De aceea nu mai avem nevoie să explorăm nodul G.



Această metodă este utilă atunci când arborele de căutare este explorat pe mai multe niveluri. În acest caz, eliminând un nod pe un anumit nivel, eliminăm tot subarborele corespunzător acelui nod desfășurat pe nivelurile inferioare.

În exemplul următor, se prezintă o situație mai compexă, în care praguri *alfa* și *beta* nu sunt folosite doar pentru niveluri succesive în arbore. Trebuie remarcat în primul rând faptul că factorul de ramificare nu este constant, deoarece numărul de mutări posibile din configurații diferite nu este constant.



Dacă explorăm tot subarborele cu rădăcina *B*, descoperim că în *A* ne putem aștepta la un câștig de minim 3. Când această valoare *alfa* este pasată în jos către *F*, ea ne va permite să evităm explorarea lui *L*. După ce *K* este examinat, observăm că în *I* este garantat un câștig de 0, ceea ce înseamnă că în *F* este garantat un minim de 0. Dar această valoare este mai mică decât valoarea *alfa* = 3, astfel încât nu mai trebuie să evaluăm alte ramificații pornind din *I*. Jucătorul maximizant deja știe că nu trebuie să aleagă mutarea în *C* și apoi în *I*, de vreme ce prin această mutare câștigul rezultant nu va fi mai mare de 0 și un câștig de 3 se poate obține mutând în *B*. Apoi este examinat *J*, ceea ce determină o valoare de 2, care îi este atribuită lui *F*, de vreme ce este maximul dintre 0, în *I*, și 2, în *J*. Această valoare devine pragul *beta* în nodul *C*, indicând faptul că în *C* este garantat un câștig mai mic sau egal cu 2. În acest moment, *alfa* =3 și *beta* = 2, se îndeplinește din nou condiția *alfa* ≥ *beta* și în consecință, nu mai este necesară explorarea nodurilor *G* și *H*.

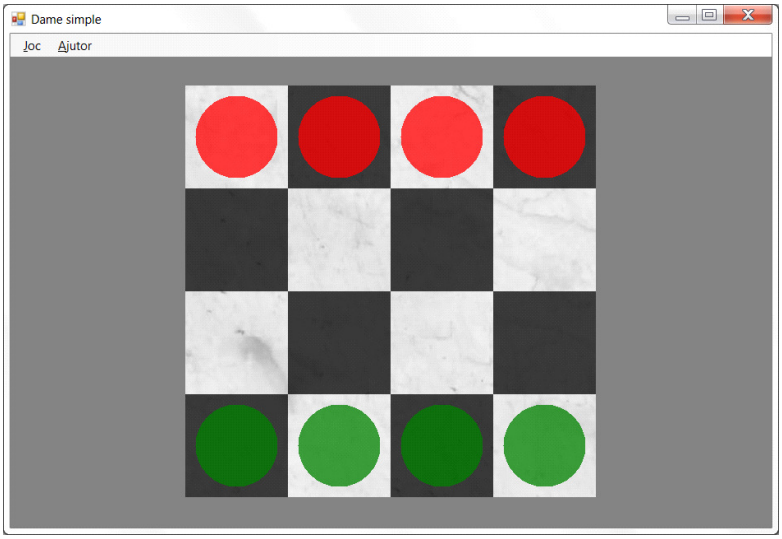
Nodurile care nu mai sunt expandate au valoarea funcției de evaluare notată între paranteze.

Pentru explicații suplimentare, consultați cursul 3.

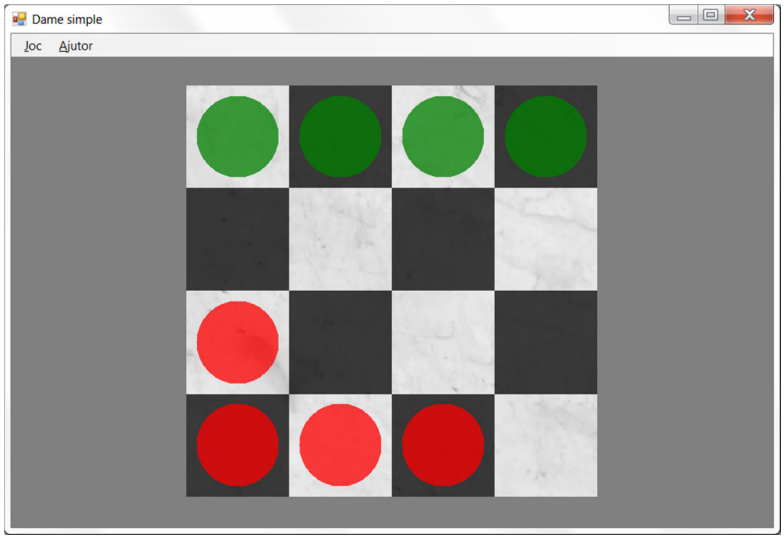
5. Aplicație

Realizați o variantă simplificată a unui joc de dame, cu următoarele reguli:

- există doi jucători;
- piesele sunt dispuse pe o tablă de 4x4 careuri;
- configurația (tabla) inițială este:



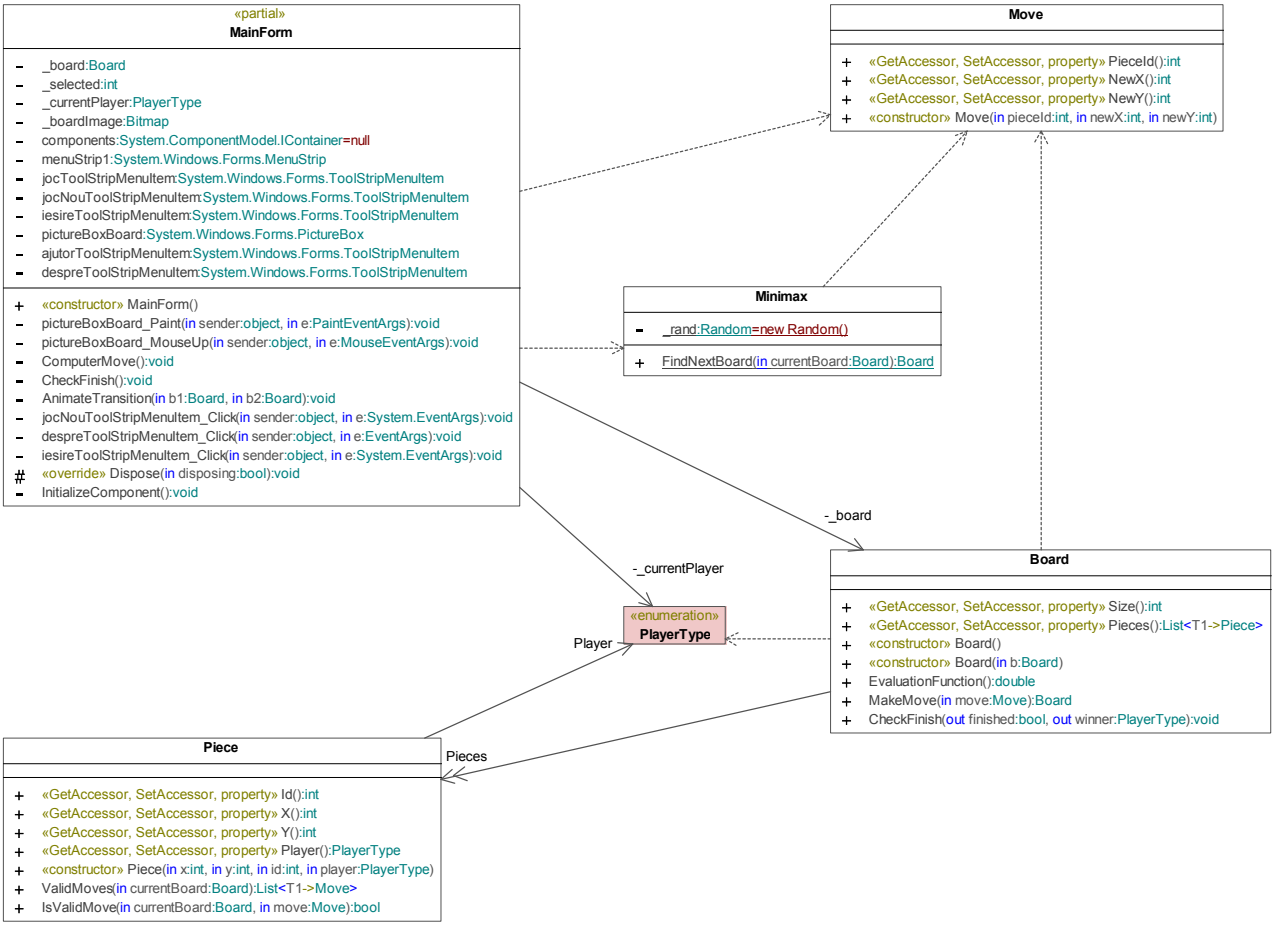
- fiecare jucător are ca scop să ajungă pe ultima linie cu toate piesele (în acest caz, a câștigat omul):



- mutări posibile: fiecare piesă poate fi mutată în orice direcție, într-un careu liber adiacent.

Florin Leon, Inteligenta artificiala - Laborator, http://florinleon.byethost24.com/lab_ia.htm

Indicații. Se dă un prototip de aplicație, în care sunt implementate toate clasele necesare pentru rezolvare și interfața grafică cu utilizatorul. Diagrama UML de clase a aplicației complete este următoarea:

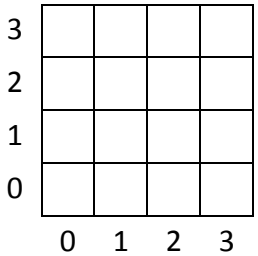


Trebuie implementate doar următoarele metode:

Metoda EvaluationFunction() din clasa Board

Calculează funcția de evaluare statică pentru configurația (tabla) curentă. Pentru un joc oarecare, programatorul trebuie să imagineze funcția de evaluare. Pot exista mai multe funcții posibile pentru un joc. Cu cât funcția aceasta este mai „inteligentă”, cu atât calculatorul va juca mai bine. De exemplu, în cazul nostru, o funcție de evaluare poate lua în calcul diferență dintre cât au avansat piesele calculatorului și cât au avansat piesele omului. Cu cât funcția este mai mare, cu atât poziția este mai bună pentru calculator, adică aceasta este mai aproape de configurația câștigătoare.

În program, coordonatele *x* și *y* ale pieselor sunt considerate ca mai jos:



Să presupunem că alegem funcția:

$$F = \sum_c dy_c - \sum_o dy_o ,$$

unde dy_c reprezintă avansul pieselor calculatorului, iar dy_o, ale omului.

Funcția va fi 0 când niciun jucător nu are vreun avantaj, de exemplu:

3				
2	C	C	O	O
1	O	O	C	C
0				
	0	1	2	3

Aici, primele două piese ale calculatorului au avansat o căsută pe axa *y*, de la 3 la 2, iar ultimele două au avansat două căsuțe, de la 3 la 1. Analog pentru om, de la 0 la 1, respectiv de la 0 la 2:

$$F = (1+1+2+2) - (1+1+2+2) = 0.$$

Funcția va fi mare când calculatorul are un avantaj, de exemplu:

3				
2				
1	O	O	O	O
0	C	C	C	C
	0	1	2	3

$$F = (3+3+3+3) - (1+1+1+1) = 8.$$

Funcția va fi mică când omul are un avantaj, de exemplu:

3	O	O	O	O
2	C	C	C	C
1				
0				
	0	1	2	3

$$F = (1+1+1+1) - (3+3+3+3) = -8.$$

Se poate verifica faptul că această funcție poate fi scrisă mai compact, luând în calcul direct coordonatele *y* ale pieselor și nu cât au avansat acestea:

$$F = 12 - \sum_c y_c - \sum_o y_o \ ,$$

unde y_c reprezintă coordonatele *y* ale pieselor calculatorului, iar y_o, ale omului.

Am putea alege și o funcție mai simplă, care să ia în considerare numai piesele calculatorului, însă în acest caz calculatorul ar încerca doar să avanseze și nu ar putea încerca să blocheze avansarea omului. De asemenea, se pot imagina și funcții mai complexe, care să ia în calcul pozițiile libere de pe linia scop, pozițiile relative ale pieselor adversarului etc.

Metoda ValidMoves(Board currentBoard) din clasa Piece

Returnează lista tuturor mutărilor permise pentru piesa curentă în configurația currentBoard. Folosește metoda IsValidMove.

Metoda IsValidMove(Board currentBoard, Move move) din clasa Piece

Testează dacă o mutare este permisă în configurația curentă. O mutare este invalidă dacă: nu mută nicio piesă, nu mută într-un careu adiacent ci sare peste mai multe careuri, mută în afara tablei sau mută într-un careu ocupat de altă piesă. Această metodă validează atât mutările calculatorului (în metoda ValidMoves din clasa Piece) cât și mutările omului (în evenimentul pictureBoxBoard_MouseUp din clasa MainForm).

Metoda FindNextBoard(Board currentBoard) din clasa Minimax

Aplică algoritmul minimax pe un singur nivel. Primul nivel corespunde mutării calculatorului, deci este maximizant. Transformă mutările valide ale tuturor pieselor din currentBoard într-o listă de configurații, folosind metoda MakeMove din clasa Board. Apoi selectează din această listă configurația cu funcția de evaluare statică maximă. Dacă există mai multe configurații cu funcția de evaluare maximă, se alege aleatoriu una dintre ele.

În cazul general, descris în laborator, algoritmul ar trebui să lucreze pe un arbore.