**Shako-proiect inteligenta artificiala**

**-Farcas Cristian Gabriel(234/1)**

**-Bizgan Gheorghe Alexandru(234/1)**

**1.Introducere**

**Shako** a fost inventat de Jean-Louis Cazaux

Site: [Odiseea șahului](http://history.chess.free.fr/) .

Shako este o variantă de șah jucată pe o tablă de 10x10.

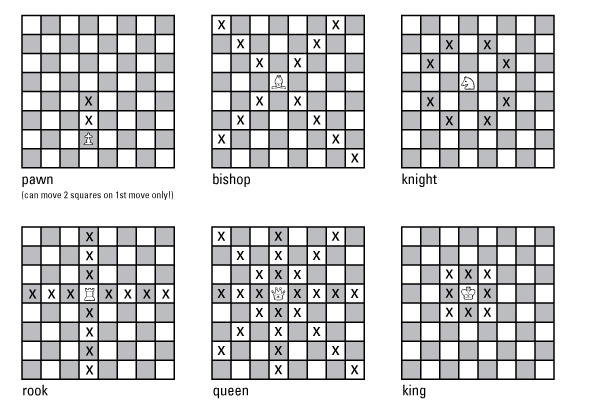
Fiecare jucător comandă 22 de piese care sunt setul standard cu încă două piese, tunuri și elefanți.



Aranjarea pieselor pe tablă:

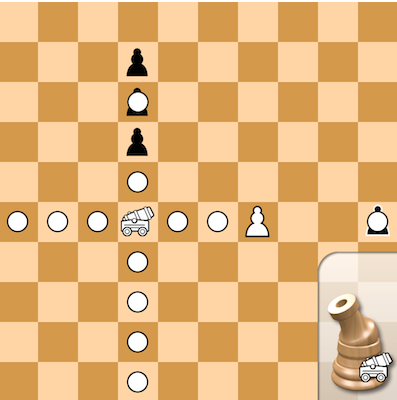
**White**:  
King f2; Queen e2; Rook b2, i2; Knight c2, h2; Bishop d2, g2; Elephant a2, j2; Cannon a1, j1, Pawn a3, b3, c3, d3, e3, f3, g3, h3, i3, j3, k3, l3.

**Black**:  
King f9; Queen e9; Rook b9, i9; Knight c9, h9; Bishop d9, g9; Elephant a9, j9; Cannon a10, j10; Pawn a8, b8, c8, d8, e8, f8, g8, h8, i8, j8, k8, l8



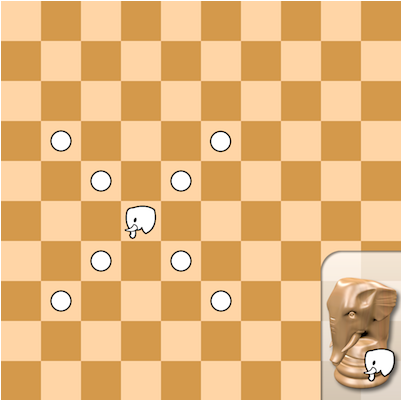
Suplimentar acestui set classic de piese Shako vine cu inca 2 piese noi Tun-ul (Cannon ) si Elefantul (Elephant)

**Tunul**



Când nu captează, se mișcă la fel ca Rook. Dar pentru a captura o piesă, trebuie să existe o piesă suplimentară în calea ei pe care tunul sări înainte de a ateriza pe piesa capturată. Această piesă intermediară poate aparține oricărui jucător. Tunul nu poate sări peste mai mult de o piesă pentru a-și face captarea și nu poate sări peste piese atunci când nu este capturat.

**Elefantul**



Elefantul calcă în diagonală unul sau două pătrate, sărind peste pătratul intermediar dacă este ocupat.

Toate celelalte piese se mișcă ca în șahul ortodox; de asemenea, roca este ca în șahul obișnuit. Pionii promovează pe al zecelea rând al tablei la Queen, Rook, Knight, Bishop, Elephant sau Cannon, la alegerea jucătorilor deținători.

Alte reguli sunt ca în șahul ortodox. Albul începe să joace.

**2.Implementarea Algoritimilor de Inteligenta Artificiala**

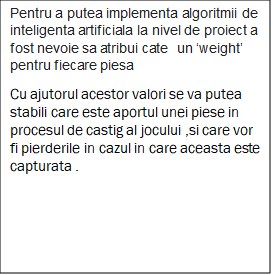
1.Jocul ca problema de cautare

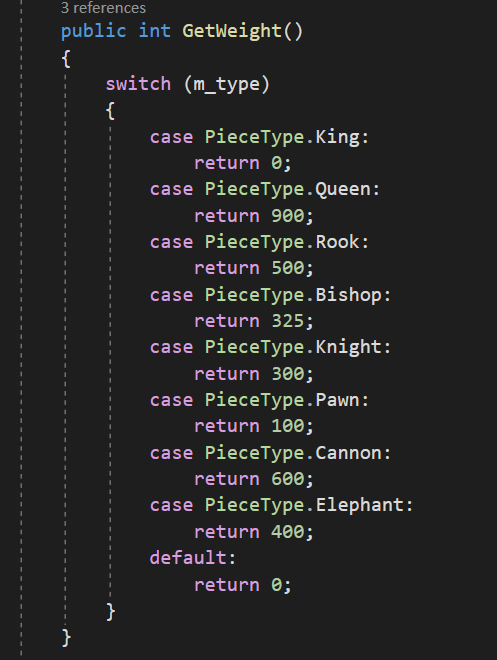
Pentru a putea fi implementat, un joc, trebuie codificat aşa încât să fie pe înţelesul calculatorului. De aceea un joc poate fi gândit ca un arbore a tuturor stărilor viitoare ale jocului.

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

2.Componentele unei probleme de cautare

* Starea iniţială, include poziţiile pe tabla de joc şi cine este la mutare.
* Mulţime de acţiuni, care definesc mutările admise pe care le poate efectua un jucător.
* Stare terminală, care determină când se sfârşeşte jocul. Stările în care jocul se încheie se numesc stări terminale.
* Funcţie de utilitate, care întoarce o valoare numerică pentru rezultatul jocului. De ex. în şah posibilităţile sunt 1 pentru victorie, 0 pentru egalitate şi -1 pentru înfrângere.





Algoritm Implementat

* **MIN-MAX**

Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.Mini-Max algorithm uses recursion to search through the game-tree.Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.In this algorithm two players play the game, one is called MAX and other is called MIN.Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

**Implementare Proiect Shako:**

// Get the best move available to the player

public Move GetBestMove()

{

int alpha, beta;

int depth; // depth to which to do the search

TimeSpan ElapsedTime= new TimeSpan(1); // Total elpased time

Move BestMove=null; // The best move for the current position

// Initialize constants

const int MIN\_SCORE= -10000000; // Minimum limit of negative for integer

const int MAX\_SCORE= 10000000; // Maximum limit of positive integer

ArrayList TotalMoves=m\_Rules.GenerateAllLegalMoves(m\_Side); // Get all the legal moves for the current side

// Now we use the Iterative deepening technique to search the best move

// The idea is just simple, we will keep searching in the more and more depth

// as long as we don't time out.

// So, it means that when we have less moves, we can search more deeply and which means

// better chess game.

DateTime ThinkStartTime=DateTime.Now;

int MoveCounter;

Random RandGenerator= new Random();

// Game is near the end, or the current player is under check

if (m\_Rules.ChessBoard.GetSideCell(m\_Side.type).Count<=5 || TotalMoves.Count <= 5 )

m\_GameNearEnd = true;

// Game is near the end, or the Enemy player is under check

Side EnemySide;

if (m\_Side.isBlack())

EnemySide = m\_Rules.ChessGame.WhitePlayer.PlayerSide;

else

EnemySide = m\_Rules.ChessGame.BlackPlayer.PlayerSide;

if (m\_Rules.ChessBoard.GetSideCell(m\_Side.Enemy()).Count<=5 || m\_Rules.GenerateAllLegalMoves(EnemySide).Count <= 5 )

m\_GameNearEnd = true;

m\_TotalMovesAnalyzed=0; // Reset the total moves anazlye counter

for (depth = 1;; depth++) // Keep doing a depth search

{

alpha = MIN\_SCORE; // The famous Alpha & Beta are set to their initial values

beta = MAX\_SCORE; // at the start of each increasing search depth iteration

MoveCounter = 0; // Initialize the move counter variable

// Loop through all the legal moves and get the one with best score

foreach (Move move in TotalMoves)

{

MoveCounter++;

// Now to get the effect of this move; execute this move and analyze the board

m\_Rules.ExecuteMove(move);

move.Score = -AlphaBeta(m\_Rules.ChessGame.EnemyPlayer(m\_Side).PlayerSide,depth - 1, -beta, -alpha);

m\_TotalMovesAnalyzed++; // Increment move counter

m\_Rules.UndoMove(move); // undo the move

// If the score of the move we just tried is better than the score of the best move we had

// so far, at this depth, then make this the best move.

if (move.Score > alpha)

{

BestMove = move;

alpha = move.Score;

}

m\_Rules.ChessGame.NotifyComputerThinking(depth, MoveCounter, TotalMoves.Count,m\_TotalMovesAnalyzed, BestMove );

// Check if the user time has expired

ElapsedTime=DateTime.Now - ThinkStartTime;

if ( ElapsedTime.Ticks > (m\_MaxThinkTime.Ticks) ) // More than 75 percent time is available

break; // Force break the loop

}

// Check if the user time has expired

ElapsedTime=DateTime.Now - ThinkStartTime;

if ( ElapsedTime.Ticks > (m\_MaxThinkTime.Ticks\*0.25)) // More than 75 percent time is available

break; // Force break the loop

}

m\_Rules.ChessGame.NotifyComputerThinking(depth, MoveCounter, TotalMoves.Count,m\_TotalMovesAnalyzed, BestMove );

return BestMove;}

**3.Functionalitate**

