

Ministerul Educației, Culturii și Cercetări al Republicii Moldova

Universitatea Tehnică a Moldovei

Departamentul Informatică și Ingineria Sistemelor

**Lucrare de laborator nr.3**

La disciplina

Sisteme de inteligență artificială

A efectuat: st.gr. IA-182

Ulmanu Cristian

A verificat: Stratulat Ștefan

Chișinău 2020

Tema: **Sisteme de IA în jocuri**

**Sarcina:** Să se elaboreze un program care folosește un algoritm determinist (e.g. minimax, A\*, monte carlo) pentru a simula un jucător cu Ai într-un joc cu informații perfecte.

# **Mini-Max Algorithm in Intelligence Artificial**

* Algoritmul mini-max este un algoritm recursiv sau de backtracking care este utilizat în luarea deciziilor și teoria jocurilor. Oferă o mișcare optimă pentru jucător presupunând că și adversarul joacă optim.
* Algoritmul Mini-Max folosește recursivitatea pentru a căuta prin arborele jocului.
* Algoritmul Min-Max este folosit în principal pentru jocurile din AI. Cum ar fi șah, dame, tic-tac-toe, go și diverse jocuri de remorcare. Acest algoritm calculează decizia minimax pentru starea actuală.
* În acest algoritm joacă doi jucători, unul se numește MAX și altul se numește MIN.
* Ambii jucători luptă împotriva acestuia, pe măsură ce jucătorul adversar obține beneficiul minim în timp ce obține beneficiul maxim.
* Ambii jucători ai jocului sunt oponenți unul de celălalt, unde MAX va selecta valoarea maximizată și MIN va selecta valoarea minimizată.
* Algoritmul minimax efectuează un algoritm de căutare în profunzime pentru explorarea arborelui complet al jocului.
* Algoritmul minimax continuă până la nodul terminal al arborelui, apoi urmărește arborele ca recursivitate.

## **Funcționarea algoritmului Min-Max:**

* Funcționarea algoritmului minimax poate fi descrisă cu ușurință folosind un exemplu. Mai jos am luat un exemplu de joc-copac care reprezintă jocul cu doi jucători.
* În acest exemplu, există doi jucători, unul se numește Maximizer și altul se numește Minimizer.
* Maximizer va încerca să obțină scorul maxim posibil, iar Minimizer va încerca să obțină scorul minim posibil.
* Acest algoritm aplică DFS, deci în acest arbore de joc, trebuie să parcurgem frunzele până la nodurile terminale.
* La nodul terminal, valorile terminale sunt date, așa că vom compara acele valori și vom întoarce arborele până când apare starea inițială. Următorii pași principali implicați în rezolvarea arborelui de joc cu doi jucători:

**Pasul 1:** În primul pas, algoritmul generează întregul arbore de joc și aplică funcția de utilitate pentru a obține valorile de utilitate pentru stările terminale. În diagrama arborelui de mai jos, să luăm A este starea inițială a arborelui. Să presupunem că maximizatorul ia primul rând care are valoarea inițială în cel mai rău caz = - infinit, iar minimizatorul va lua următoarea tură care are valoarea inițială în cel mai rău caz = + infinit.



**Pasul 2:** Acum, mai întâi găsim valoarea utilităților pentru Maximizer, valoarea sa inițială este -∞, deci vom compara fiecare valoare în starea terminalului cu valoarea inițială a Maximizer și vom determina valorile nodurilor mai mari. Acesta va găsi maximul dintre toate.

* Pentru nodul D max (-1, - -∞) => max (-1,4) = 4
* Pentru nodul E max (2, -∞) => max (2, 6) = 6
* Pentru nodul F max (-3, -∞) => max (-3, -5) = -3
* Pentru nodul G max (0, -∞) = max (0, 7) = 7



**Pasul 3:** În pasul următor, este o rundă pentru minimizator, deci va compara valoarea tuturor nodurilor cu + ∞ și va găsi valorile nodului al treilea strat.

* Pentru nodul B = min (4,6) = 4
* Pentru nodul C = min (-3, 7) = -3



**Pasul 3:** Acum este rândul lui Maximizer și va alege din nou valoarea maximă a tuturor nodurilor și va găsi valoarea maximă pentru nodul rădăcină. În acest arbore de joc, există doar 4 straturi, prin urmare ajungem imediat la nodul rădăcină, dar în jocurile reale, vor exista mai mult de 4 straturi.

* Pentru nodul A max (4, -3) = 4



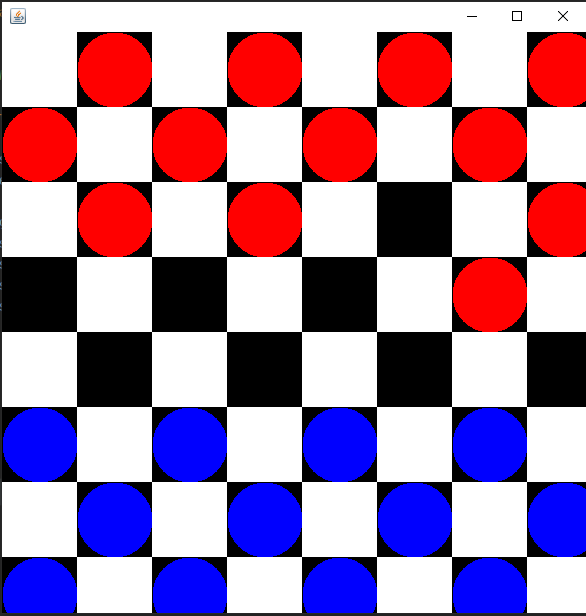
Acesta a fost fluxul de lucru complet al jocului minimax pentru doi jucători.

## **Proprietățile algoritmului Mini-Max:**

* **Complet** - Algoritmul Min-Max este complet. Cu siguranță va găsi o soluție (dacă există), în arborele de căutare finit.
* **Optimal-** Min-Max algoritm este optim în cazul în care ambele adversarii joacă optim.
* **Complexitatea timpului -** Pe măsură ce efectuează DFS pentru arborele jocului, astfel complexitatea timpului algoritmului Min-Max este **O (b m )** , unde b este factorul de ramificare al arborelui jocului, iar m este adâncimea maximă a arborelui.
* **Complexitatea spațială - Complexitatea** spațială a algoritmului Mini-max este, de asemenea, similară cu DFS, care este **O (bm)** .

**Partea practică. Componenta Min-Max**

package gameLogic;  
  
  
import java.util.ArrayList;  
  
  
public class aiMove {  
  
 private board.color color;  
 private board.color oppColor;  
 private Tree descisionTree;  
 private Move lastMove;  
 */\*\*  
 \* Creates a new aiMove with the color of its pieces  
 \** ***@param*** *color the color of the pieces the ai will move  
 \*/* public aiMove(board.color color) {  
 this.color = color;  
 if (color == board.color.*RED*) {  
 oppColor = board.color.*BLACK*;  
 } else {  
 oppColor = board.color.*RED*;  
 }  
 }  
  
 */\*\*  
 \* Picks its move based on all possible moves  
 \** ***@param*** *board the current state of the board  
 \** ***@return*** *the move picked by the ai  
 \*/* public Move getAIMove(board board) {  
 descisionTree = makeDescisionTree(board);  
 lastMove = pickMove();  
 return lastMove;  
 }  
  
 */\*\*  
 \** ***@return*** *the color of the ai  
 \*/* public board.color getColor() {  
 return color;  
 }  
  
 */\*\*  
 \* Creates a tree with a height of four that has all possible moves  
 \* for the next three moves of the game  
 \** ***@param*** *board the board that the tree will be based on  
 \** ***@return*** *a tree with all possible moves  
 \*/* private Tree makeDescisionTree(board board) {  
 Tree mainTree = new Tree(board, null, score(board));  
 ArrayList<Move> moves;  
 // Handles multiple jumps  
 if (board.isJumped()) {  
 moves = board.getJumps(lastMove.movRow, lastMove.movCol);  
 } else {  
 moves = board.getAllLegalMovesForColor(color);  
 }  
  
 for (Move move : moves) {  
 // Make second row  
 board temp = copyBoard(board);  
 temp.movePiece(move);  
 temp.handleJump(move);  
 Tree firstLayer = new Tree(temp, move, score(temp));  
 ArrayList<Move> secondMoves = temp.getAllLegalMovesForColor(oppColor);  
  
 for (Move sMove : secondMoves) {  
 // Make third row  
 board temp2 = copyBoard(temp);  
 temp2.movePiece(sMove);  
 temp2.handleJump(sMove);  
 Tree secondLayer = new Tree(temp2, sMove, score(temp2));  
 ArrayList<Move> thirdMoves = temp2.getAllLegalMovesForColor(color);  
  
 for (Move tMove : thirdMoves) {  
 // Make fourth row  
 board temp3 = copyBoard(temp2);  
 temp3.movePiece(tMove);  
 temp3.handleJump(tMove);  
  
 secondLayer.addChild(new Tree(temp3, tMove, score(temp3)));  
 }  
  
 firstLayer.addChild(secondLayer);  
 }  
 mainTree.addChild(firstLayer);  
 }  
  
 return mainTree;  
 }  
  
 */\*\*  
 \* Picks the move based on minimax  
 \** ***@return*** *the move that was selected  
 \*/* private Move pickMove() {  
 int max = -13;  
 int index = 0;  
 for (int i = 0; i < descisionTree.getNumChildren(); i++) {  
 Tree child = descisionTree.getChild(i);  
 int smin = 13;  
 // Find the max leaf  
 for (Tree sChild : child.getChildren()) {  
 int tMax = -13;  
 for (Tree tchild : sChild.getChildren()) {  
 if (tchild.getScore() >= tMax) {  
 tMax = tchild.getScore();  
 }  
 }  
 sChild.setScore(tMax);  
 // Find the min on the third level  
 if (sChild.getScore() <= smin) {  
 smin = sChild.getScore();  
 }  
 }  
 child.setScore(smin);  
 // Find the max on the second layer and save the index  
 if (child.getScore() >= max) {  
 max = child.getScore();  
 index = i;  
 }  
 }  
 return descisionTree.getChild(index).getMove();  
 }  
  
 */\*\*  
 \* Scores the given board based on a weighted system  
 \** ***@param*** *board the board that will be scored  
 \** ***@return*** *the score of the given board  
 \*/* private int score(board board) {  
 if (color == gameLogic.board.color.*RED*) {  
 return board.getRedWeightedScore() - board.getBlackWeightedScore();  
 } else {  
 return board.getRedWeightedScore() - board.getRedWeightedScore();  
 }  
 }  
  
 */\*\*  
 \* Creates a new board with the same information as the given board  
 \** ***@param*** *board the board that will be copied  
 \** ***@return*** *a copy of the given board  
 \*/* private board copyBoard(board board) {  
 gameLogic.board.color[][] color = new board.color[8][8];  
 for (int row = 0; row < 8; row++) {  
 for (int col = 0; col < 8; col++) {  
 color[row][col] = board.getInfoAtPosition(row, col);  
 }  
 }  
 return new board(color, board.getNumRed(), board.getNumBlack(), board.getNumRedKing(), board.getNumBlackKing());  
 }  
  
  
}



**Bibliografie:**

1. L. G. Komartsova, A. V. Maksimov "Neurocomputatoare", MSTU im. N. Bauman, 2004, [ISBN 5-7038-2554-7](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/5703825547)
2. <https://www.javatpoint.com/mini-max-algorithm-in-ai#:~:text=Mini%2Dmax%20algorithm%20is%20a,decision%2Dmaking%20and%20game%20theory.&text=Mini%2DMax%20algorithm%20uses%20recursion,and%20various%20tow%2Dplayers%20game>.
3. [Provincial Healthcare Index 2013](http://www.fraserinstitute.org/uploadedFiles/fraser-ca/Content/research-news/research/publications/provincial-healthcare-index-2013.pdf) (Bacchus Barua, Fraser Institute, January 2013 -see page 25-)
4. Michael Maschler, [*Eilon Solan*](https://en.wikipedia.org/wiki/Eilon_Solan) & Shmuel Zamir (2013). Game Theory. [*Cambridge University Press*](https://en.wikipedia.org/wiki/Cambridge_University_Press). pp. 176–180.