All three algorithms start by finding all of the open positions on the board.

Also, before selecting a position (after all calculations), the list of positions is shuffled and then sorted in each algorithm. This is to ensure that positions worth the same value are not always chosen in the same order.

**MiniMax**

This algorithm calculates the value of placing a stone on each possible position and selects the position with the greatest value.  
The value is calculated by means of recursion. For each available position the algorithm evaluates every possible outcome which could result from choosing that position. If the potential outcome is a loss then that position is given a value of -1. However, if the potential outcome is a win then that position is given the value of 1. It assumes that the opponent will make the best possible moves. It then selects the position with the highest value.

|  |  |  |  |
| --- | --- | --- | --- |
| **Performance of Minimax against other players on board size 3x3** | | | |
|  | Number wins as player 1 (out of 2) | Number wins as player 2 (out of 2) | Total number wins |
| Minimax vs Random Player | 2 | 0 | 2 |
| Minimax vs Monte Carlo Player | 2 | 0 | 2 |

2,2,1, 1,0,2, 1,0,2, 0,1,2,

1,1,2, 2,2,1, 2,2,0, 0,1,2,

1,1,2, 1,1,2, 1,1,0, 1,0,0,

**MonteCarlo**

For every possible position, the algorithm plays ‘random’ games to see what the outcome would be. A random game consists of repeatedly selecting a random position, out of those available, to represent each player’s turn until the game is either won or lost (or a draw). If the random game produced from selecting a particular position results in a win then the value of that position is increased by 1/(the number of moves required to win). If the random game results in a loss then the value of the position is decreased by 1/(the number of moves required to win).  
This process is repeated for 99% of the allocated response time. Then the position with the greatest value is selected.

|  |  |  |  |
| --- | --- | --- | --- |
| **Performance of MonteCarlo against other players on various board sizes** | | | |
|  | Number wins as player 1 (out of 4) | Number wins as player 2 (out of 4) | Total number wins |
| MonteCarlo vs Provided Random Player | 4 | 4 | 8 |
| MonteCarlo vs Provided MonteCarlo Player | 2 | 4 | 6 |
| MonteCarlo vs Provided Bandit Player | 3 | 2 | 5 |

0,0,0,1,2,0,0, 1,2,0,0,0,0,1,2,

0,0,0,1,2,0,0, 1,2,0,0,0,2,1,0,

0,0,0,1,2,0,0, 1,2,0,0,2,1,0,0,

0,0,0,1,0,0,0, 1,2,0,2,1,0,0,0,

0,0,1,2,0,0,0, 1,0,2,1,2,2,0,0,

0,1,2,0,0,0,0, 1,2,1,2,1,0,0,0,

1,2,0,0,0,0,0, 1,2,2,1,0,0,0,0,

1,0,0,0,0,0,0,0,

**Bandit**

The bandit algorithm follows a very similar procedure to the MonteCarlo algorithm. The major differences are that the calculated value for each position is not divided by the number of moves required to win/lose. The algorithm also keeps track of the number of times a random game is played from a particular position and the total number of random games played.  
Once a 99% of the allocated response time has elapsed, the algorithm then calculates the probability of winning for each position using the Bandit equation. The position with the highest probability is then selected.

|  |  |  |  |
| --- | --- | --- | --- |
| **Performance of Bandit against other players on various board sizes** | | | |
|  | Number wins as player 1 (out of 4) | Number wins as player 2 (out of 4) | Total number wins |
| Bandit vs Provided Random Player | 4 | 4 | 8 |
| Bandit vs Provided MonteCarlo Player | 2 | 2 | 4 |
| Bandit vs Provided Bandit Player | 2 | 2 | 4 |

0,0,0,0,0,0,2, 0,0,0,0,0,0,0,2,

0,0,0,0,0,2,1, 0,0,0,0,0,0,2,1,

0,0,0,0,2,1,0, 0,0,0,0,0,2,1,0,

0,0,1,2,1,0,0, 0,0,0,2,2,1,0,0,

0,1,2,1,0,0,0, 0,0,2,1,1,0,0,0,

2,2,0,0,0,0,0, 2,2,1,1,0,0,0,0,

1,0,0,0,0,0,0, 1,0,0,0,0,0,0,0,

0,0,0,0,0,0,0,0,

Find Position class at the end.

### MinMax

@Override

public String MakeMove(String board)

{

*// Convert board string to board*

ArrayList<Position> emptySpots = new ArrayList<>();

BoardDataStructure temp = new BoardDataStructure(BoardSize);

StringTokenizer st = new StringTokenizer(board, ",");

for(int r =0; r < BoardSize; r++)

for(int c = 0; c < BoardSize; c++) {

int x = Integer.*parseInt*(st.nextToken());

temp.Board[c][r] = x;

*// Find empty positions*

if (x == BoardDataStructure.*Empty*)

emptySpots.add(new Position(c, r));

}

Position pos = MinMax(temp, emptySpots);

return pos.toString();

}

private Position MinMax(BoardDataStructure board, ArrayList<Position> emptySpots){

*// Perform MinMax algorithm*

int val = Integer.*MIN\_VALUE*;

int count = 0;

while (count < emptySpots.size()){

Position pos = emptySpots.remove(0);

board.Board[pos.getCol()][pos.getRow()] = Side;

val = Math.*max*(val, MinVal(board, emptySpots));

board.Board[pos.getCol()][pos.getRow()] = BoardDataStructure.*Empty*;

pos.setValue(val);

emptySpots.add(pos);

count++;

}

*// find Position that leads to the max result*

Collections.*shuffle*(emptySpots);

emptySpots.sort(Comparator.*comparingInt*(Position::getValue));

return emptySpots.get(emptySpots.size()-1);

}

private int MaxVal(BoardDataStructure board, ArrayList<Position> slots){

int val = board.CheckWinner();

if (val != BoardDataStructure.*Empty*)

return val == Side ? 1 : -1;

val = Integer.*MIN\_VALUE*;

int count = 0;

while (count < slots.size()){

Position pos = slots.remove(0);

board.Board[pos.getCol()][pos.getRow()] = Side;

val = Math.*max*(val, MinVal(board, slots));

board.Board[pos.getCol()][pos.getRow()] = BoardDataStructure.*Empty*;

slots.add(pos);

count++;

}

return val;

}

private int MinVal(BoardDataStructure board, ArrayList<Position> slots){

int val = board.CheckWinner();

if (val != BoardDataStructure.*Empty*)

return val == Side ? 1 : -1;

val = Integer.*MAX\_VALUE*;

int count = 0;

while (count < slots.size()){

Position pos = slots.remove(0);

board.Board[pos.getCol()][pos.getRow()] = Side == BoardDataStructure.*BlueMove* ? BoardDataStructure.*RedMove* : BoardDataStructure.*BlueMove*;

val = Math.*min*(val, MaxVal(board, slots));

board.Board[pos.getCol()][pos.getRow()] = BoardDataStructure.*Empty*;

slots.add(pos);

count++;

}

return val;

}

### MonteCarlo

private long startTime;

@Override

public String MakeMove(String board)

{

startTime = System.*currentTimeMillis*();

ArrayList<Position> availableSlots = new ArrayList<>();

*// Convert board string to board*

BoardDataStructure temp = new BoardDataStructure(BoardSize);

StringTokenizer st = new StringTokenizer(board, ",");

for(int r =0; r < BoardSize; r++)

for(int c = 0; c < BoardSize; c++) {

int x = Integer.*parseInt*(st.nextToken());

temp.Board[c][r] = x;

if (x == BoardDataStructure.*Empty*)

availableSlots.add(new Position(c, r, 0));

}

Position pos = monteCarlo(temp, availableSlots);

return pos.toString();

}

private Position monteCarlo(BoardDataStructure board, ArrayList<Position> slots){

while ((System.*currentTimeMillis*() - startTime) < ResponseTime\*1000\*0.99){

for (Position pos : slots){

board.Board[pos.getCol()][pos.getRow()] = Side;

int value = playRandomGame(board, slots);

pos.setValue(pos.getValue() + value);

board.Board[pos.getCol()][pos.getRow()] = BoardDataStructure.*Empty*;

}

}

Collections.*shuffle*(slots);

slots.sort(Comparator.*comparingInt*(Position::getValue));

return slots.get(slots.size()-1);

}

private int playRandomGame(BoardDataStructure board, ArrayList<Position> slots){

ArrayList<Position> temp = (ArrayList<Position>) slots.clone();

int winner = BoardDataStructure.*Empty*;

int turn = Side;

int count = 0;

while(winner == BoardDataStructure.*Empty* && !temp.isEmpty()){

*// next player's turn*

turn = turn == BoardDataStructure.*BlueMove* ? BoardDataStructure.*RedMove* : BoardDataStructure.*BlueMove*;

*// make a random move*

int rand = r.nextInt(temp.size());

Position rando = temp.remove(rand);

board.Board[rando.getCol()][rando.getRow()] = turn;

*// check if there's a winner*

winner = board.CheckWinner();

*// count number of moves*

count++;

}

return winner == Side ? 1/count : -1/count;

}

### Bandit

private long startTime;

@Override

public String MakeMove(String board)

{

startTime = System.*currentTimeMillis*();

ArrayList<Position> availableSlots = new ArrayList<>();

*// Convert board string to board*

BoardDataStructure temp = new BoardDataStructure(BoardSize);

StringTokenizer st = new StringTokenizer(board, ",");

for(int r =0; r < BoardSize; r++)

for(int c = 0; c < BoardSize; c++) {

int x = Integer.*parseInt*(st.nextToken());

temp.Board[c][r] = x;

if (x == BoardDataStructure.*Empty*)

availableSlots.add(new Position(c, r, 0));

}

Position pos = bandit(temp, availableSlots);

return pos.toString();

}

private Position bandit(BoardDataStructure board, ArrayList<Position> slots){

*// play random games*

int pathsPlayed = 0; *// N*

while ((System.*currentTimeMillis*() - startTime) < ResponseTime\*1000\*0.99){

for (Position pos : slots){

board.Board[pos.getCol()][pos.getRow()] = Side;

int value = playRandomGame(board, slots);

pos.setValue(pos.getValue() + value); *// wi*

pos.setPlayCount(pos.getPlayCount() + 1); *// ni*

board.Board[pos.getCol()][pos.getRow()] = BoardDataStructure.*Empty*;

pathsPlayed++;

}

}

*// calculate selection probability for each position*

for (Position pos : slots)

pos.setSelectionProbability((pos.getValue()/(pos.getPlayCount()\*1.0) + 2\*(Math.*log*(pathsPlayed)/pos.getPlayCount())));

*// select node with greatest probability*

Collections.*shuffle*(slots); slots.sort(Comparator.*comparingDouble*(Position::getSelectionProbability));

return slots.get(slots.size()-1);

}

private int playRandomGame(BoardDataStructure board, ArrayList<Position> slots){

ArrayList<Position> temp = (ArrayList<Position>)slots.clone();

int winner = BoardDataStructure.*Empty*;

int turn = Side;

while(winner == BoardDataStructure.*Empty* && !temp.isEmpty()){

*// next player's turn*

turn = turn == BoardDataStructure.*BlueMove* ? BoardDataStructure.*RedMove* : BoardDataStructure.*BlueMove*;

*// make a random move*

int rand = r.nextInt(temp.size());

Position rando = temp.remove(rand);

board.Board[rando.getCol()][rando.getRow()] = turn;

*// check if there's a winner*

winner = board.CheckWinner();

*// count number of moves*

}

return winner == Side ? 1 : -1;

}

### Position

public class Position {

private int col;

private int row;

private int value;

private int playCount = 0; *// Number of times this node was played (for bandit method)*

private double selectionProbability = 0; *// For bandit method*

public Position(int col, int row) {

this.col = col;

this.row = row;

}

public Position(int col, int row, int value) {

this.col = col;

this.row = row;

this.value = value;

}

public int getCol() {

return col;

}

public void setCol(int col) {

this.col = col;

}

public int getRow() {

return row;

}

public void setRow(int row) {

this.row = row;

}

public int getValue() {

return value;

}

public void setValue(int value) {

this.value = value;

}

public int getPlayCount() {

return playCount;

}

public void setPlayCount(int playCount) {

this.playCount = playCount;

}

public double getSelectionProbability() {

return selectionProbability;

}

public void setSelectionProbability(double selectionProbability) {

this.selectionProbability = selectionProbability;

}

@Override

public String toString() {

return col + "," + row;

}

}