

Laporan Tugas Besar 1 IF3170 Intelegensia Buatan
Semester I tahun 2023/2024

Minimax Algorithm and Alpha Beta Pruning in Adjacency Strategy Game



Anggota Kelompok:

Ulung Adi Putra	13521122
Muhammad Naufal Nalendra	13521152
Mohammad Rifqi Farhansyah	13521166
Satria Octavianus Nababan	13521168

**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2023**

Daftar Isi

Daftar Isi	2
BAB I	3
1.1. Penjelasan <i>Objective Function</i>	3
1.1.1. Fungsi <i>Objective</i> pada <i>Minimax Alpha Beta Pruning</i>	3
1.1.2 Fungsi <i>Objective</i> pada <i>Local Search (Hill Climbing Sideways)</i>	3
1.1.3 Fungsi <i>Objective</i> pada <i>Minimax Genetic Algorithm</i>	3
1.2. Algoritma <i>Minimax</i> dan <i>Alpha Beta Pruning</i>	4
1.3. Algoritma <i>Local Search</i>	11
1.4. Algoritma <i>Genetic</i>	15
BAB II	24
2.1. <i>Minimax</i> vs Manusia	24
2.2. <i>Local Search</i> vs Manusia	26
2.3. <i>Minimax</i> vs <i>Local Search</i>	27
2.4. <i>Minimax</i> vs <i>Genetic</i>	28
2.5. <i>Local Search</i> vs <i>Genetic</i>	29
BAB III	31
Referensi	32
Pranala Terkait	33

BAB I

Penjelasan Algoritma

1.1. Penjelasan *Objective Function*

1.1.1. Fungsi *Objective* pada *Minimax Alpha Beta Pruning*

Objective function yang kami implementasikan pada algoritma *Minimax Alpha Beta Pruning* ini adalah nilai heuristik yang mencerminkan keuntungan relatif dari suatu posisi dalam permainan. Tujuan algoritma ini adalah untuk mencari langkah terbaik yang akan menghasilkan nilai heuristik tertinggi bagi pemain yang dikendalikan oleh bot (pemain "O" atau "X" tergantung pada parameter yang diberikan). Ini dilakukan dengan menguraikan berbagai skenario dan memilih langkah terbaik berdasarkan perbandingan nilai heuristik dari setiap skenario. Dalam konteks permainan *Adjacency Strategy Game*, nilai heuristik ini mencerminkan seberapa baik posisi bot dalam mencapai tujuan permainan, yaitu memiliki marka sebanyak mungkin pada papan permainan.

Fungsi ``addHeuristicMove`` digunakan untuk menghitung nilai heuristik atau poin dari langkah-langkah yang mungkin dalam permainan. Langkah-langkah ini dihitung berdasarkan sejumlah faktor yang mencerminkan keuntungan potensial pemain. Faktor-faktor ini termasuk jumlah musuh yang berdekatan dan berbagai strategi permainan. Hasil perhitungan ini digunakan untuk mengevaluasi dan memilih langkah terbaik.

Pada algoritma *Minimax Alpha Beta Pruning* ini, pemilihan langkah dilakukan dengan meminimalkan atau memaksimalkan nilai heuristik berdasarkan peran pemain (pemain bot atau musuh). Selanjutnya, algoritma *minimax* digunakan untuk memilih langkah yang akan menghasilkan nilai heuristik terbaik untuk pemain bot.

1.1.2 Fungsi *Objective* pada *Local Search (Hill Climbing Sideways)*

Objective function yang kami implementasikan pada algoritma *Hill Climbing Sideways* adalah fungsi `"calculatePoint"`. Fungsi ini digunakan untuk menghitung nilai heuristik atau poin dari suatu solusi atau keadaan tertentu pada papan permainan.

Pada bagian awal algoritma, nilai heuristik dari solusi saat ini (*currentSolution*) dihitung dengan menggunakan fungsi `calculatePoint(currentSolution)`. Fungsi `calculatePoint` ini kemudian digunakan untuk menilai setiap langkah yang mungkin pada papan permainan, dan nilai-nilai tersebut digunakan dalam proses pengambilan keputusan untuk memutuskan langkah mana yang akan diambil selanjutnya. Objektif dari algoritma ini adalah untuk mencari langkah yang meningkatkan nilai heuristik solusi atau mempertahankan nilai tinggi dalam kasus di mana tidak ada langkah yang memperbaiki nilai saat ini.

1.1.3 Fungsi *Objective* pada *Minimax Genetic Algorithm*

Dalam *Genetic Algorithm*, fungsi objektif atau fitness function berperan sebagai ukuran kualitas dari solusi atau individu dalam populasi. Pada permainan *Adjacency Strategy Game*, tujuan dari fungsi objektif adalah untuk mengevaluasi seberapa baik suatu solusi (konfigurasi papan permainan) dalam mencapai target utama, yaitu memiliki sebanyak mungkin marka pada akhir permainan.

Dalam *Genetic Algorithm* yang kami implementasikan, *objective function* terdapat pada metode ``calculateFitness``. *Objective function* ini digunakan untuk mengevaluasi seberapa baik kromosom dalam

populasi (*gene pool*) untuk memahami sejauh mana mereka berhasil dalam permainan tertentu. Secara khusus, fungsi ini diimplementasikan untuk menilai kualitas langkah-langkah yang dihasilkan oleh kromosom.

Berikut adalah ringkasan tentang cara fungsi ini bekerja:

1. Input

- ``genePool``: Sebuah populasi kromosom yang dihasilkan secara acak.
- ``isForSelection``: Sebuah flag yang menentukan apakah evaluasi ini dilakukan untuk seleksi atau tidak.

2. Proses

- Untuk setiap kromosom dalam ``genePool``, nilai kecocokan dihitung berdasarkan seberapa baik langkah-langkah diwakili oleh kromosom tersebut dalam mencapai tujuan tertentu dalam permainan.
- Nilai ini dihitung menggunakan variabel ``temp`` yang awalnya diatur sebagai skor saat ini dalam permainan (``point``).
- Setiap kromosom merepresentasikan sejumlah langkah yang akan diambil oleh pemain O dan X secara bergantian.
- Selama iterasi melalui langkah-langkah dalam kromosom:
 - Jika indeks langkah adalah gen O (gen genap), simpan langkah tersebut di papan dan evaluasi skor.
 - Jika indeks langkah adalah gen X (gen ganjil), simpan langkah tersebut di papan dan evaluasi skor.
- Jika ``isForSelection`` adalah ``true``, maka jika skor (``temp``) kurang dari atau sama dengan 0, kromosom digantikan dengan kromosom acak baru.
- Jika ``isForSelection`` adalah ``false``, maka jika skor (``temp``) kurang dari atau sama dengan 0, skor diatur sebagai 9999 untuk menunjukkan bahwa kromosom ini tidak berhasil.

3. Output

- Sebuah ArrayList yang berisi nilai-nilai kecocokan (skor) untuk setiap kromosom dalam ``genePool``. Semakin tinggi skornya, semakin baik kromosom tersebut dalam konteks permainan tertentu.

Fungsi ini berfungsi sebagai metrik untuk mengevaluasi sejauh mana langkah-langkah yang dihasilkan oleh kromosom mendekati solusi optimal dalam permainan yang dimainkan. Tujuan akhir dari evolusi genetika adalah untuk menghasilkan kromosom yang memiliki nilai kecocokan (skor) yang lebih tinggi, yang secara teoritis mencerminkan strategi permainan yang lebih baik.

1.2. Algoritma *Minimax* dan *Alpha Beta Pruning*

Algoritma Minimax dengan Alpha-Beta Pruning adalah salah satu teknik penting dalam kecerdasan buatan yang digunakan untuk pengambilan keputusan dalam permainan dan masalah penelusuran pohon yang memiliki banyak cabang seperti catur, permainan catur, dan banyak aplikasi lainnya yang melibatkan pengambilan keputusan berbasis pohon. Algoritma ini bertujuan untuk menentukan langkah terbaik dalam situasi permainan adversarial di mana dua pemain saling berlawanan. Algoritma Minimax membantu pemain untuk memaksimalkan hasil permainan dengan asumsi bahwa pemain lawan juga bermain secara optimal.

Inti dari algoritma Minimax adalah mengidentifikasi pilihan langkah yang menghasilkan nilai terbaik saat berada pada giliran pemain saat ini, dan pada giliran pemain berikutnya (lawan), memilih langkah yang menghasilkan nilai terburuk (karena lawan akan berusaha meminimalkan hasilnya). Proses ini terus berlanjut hingga pohon permainan dijelajahi hingga kedalaman tertentu.

Alpha-Beta Pruning adalah teknik optimasi yang digunakan untuk mengurangi jumlah simpul yang harus dieksplorasi dalam pohon permainan. Ini bekerja dengan mempertimbangkan alpha dan beta, yang merupakan dua nilai batasan yang menggambarkan rentang nilai yang mungkin. Pada giliran pemain saat ini, algoritma Minimax mencoba mengganti nilai alpha, sedangkan pada giliran pemain berikutnya, mencoba mengganti nilai beta. Ketika nilai beta lebih kecil atau sama dengan alpha di simpul tertentu dalam pohon, tidak perlu lagi mengeksplorasi simpul tersebut atau cabang pohon di atasnya karena pemain akan memilih cabang lain yang lebih baik.

Dengan menggabungkan Algoritma Minimax dengan Alpha-Beta Pruning, kita dapat secara signifikan mengurangi jumlah perhitungan yang diperlukan untuk menemukan langkah terbaik dalam permainan yang kompleks. Hal ini membuat algoritma ini sangat efisien dalam banyak aplikasi permainan, memungkinkan komputer untuk bermain catur, shogi, dan banyak permainan lainnya dengan tingkat kecerdasan tinggi. Selain itu, algoritma ini juga diterapkan dalam berbagai aplikasi dalam kecerdasan buatan yang melibatkan pengambilan keputusan berbasis pohon, seperti perencanaan robotik dan pemecahan masalah optimasi.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import javafx.scene.control.Button;

public class BotMinMax extends Bot{
    private static final int INFINITY = 9999;

    private Node node;

    private int ROW;
    private int COL;

    private Button[][] buttons;

    private String bot;
    private String opponent;

    public BotMinMax(int row, int col, Button[][] buttons, String bot) {
        this.ROW = row;
        this.COL = col;
        this.buttons = buttons;
        this.bot = bot;
        this.opponent = bot.equals("O") ? "X" : "O";
    }

    public int[] move(Button[][] buttons, int roundsLeft, int playerOScore, int playerXScore, boolean
    playerXTurn) {
```

```

this.node = new Node();
initializeBoard();

int depth = 4;
long startTime = System.currentTimeMillis();
long timeout = 5000;

int[] bestMove = randomMove(this.node);

for (int currentDepth = 1; currentDepth <= depth; currentDepth++) {
    long currentTime = System.currentTimeMillis();
    if (currentTime - startTime >= timeout) {
        break; // Exit the loop
    }

    Node tempNode = new Node(this.node); // Create a temporary node to avoid modifying the
current node
    minimax(tempNode, currentDepth, roundsLeft, -INFINITY, INFINITY, true);

    // Update the best move
    for (Node child : tempNode.getChildren()) {
        if (child.getValue() == tempNode.getValue()) {
            bestMove = child.getMove();
        }
    }
}

System.out.println(bestMove[0] + ", " + bestMove[1]);
return bestMove;
}

private void minimax(Node node, int depth, int roundsLeft, int alpha, int beta, boolean maxPlayer) {
    if(depth == 0 || roundsLeft == 0) {
        node.setTerminalValue(this.bot);
        return;
    }

    if(maxPlayer) {
        addBestMove(this.bot, node);

        if(node.getChildren().isEmpty()) {
            node.setTerminalValue(this.bot);
            return;
        }

        node.setValue(-INFINITY);
        for (Node child : node.getChildren()) {
            minimax(child, depth-1, roundsLeft-1, alpha, beta, false);
            node.setValue(Math.max(child.getValue(), node.getValue()));
            alpha = Math.max(child.getValue(), alpha);
            if(beta <= alpha) break;

```

```

    }
}

else {
    addBestMove(this.opponent, node);

    if(node.getChildren().isEmpty()) {
        node.setTerminalValue(this.bot);
        return;
    }

    node.setValue(INFINITY);
    for (Node child : node.getChildren()) {
        minimax(child, depth-1, roundsLeft-1, alpha, beta, true);
        node.setValue(Math.min(child.getValue(), node.getValue()));
        beta = Math.min(child.getValue(), beta);
        if(beta <= alpha) break;
    }
}
}

private void initializeBoard() {
    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {
            if (!buttons[i][j].getText().isEmpty()) {
                this.node.writeNodeBoard(i, j, buttons[i][j].getText());
            }
        }
    }
}

private void addBestMove(String botMarker, Node node) {
    int[] move = randomMove(node);

    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {
            if (node.getNodeBoard()[i][j].equals("")) {
                move = new int[] {i, j};
                Node child = new Node(node);
                child.writeNodeBoard(move[0], move[1], this.bot);
                child.setMove(move[0], move[1]);
                node.addChild(child);
            }
        }
    }
}

public int[] randomMove(Node node) {
    List<int[]> validMove = new ArrayList<>();
    Random rand = new Random();

    for(int i = 0; i < ROW; i++) {

```

```

        for(int j = 0; j < COL; j++) {
            if(node.getNodeBoard()[i][j].equals("")) {
                validMove.add(new int[]{i, j});
            }
        }
    }
    int rand_index = rand.nextInt(validMove.size());

    return validMove.get(rand_index);
}
}

```

```

import java.util.ArrayList;
import java.util.List;

public class Node {
    private static final int ROW = 8;
    private static final int COL = 8;

    private String[][] nodeBoard;

    private int[] move;

    private int value;

    private List<Node> children;

    public Node() {
        this.nodeBoard = new String[ROW][COL];
        this.value = 0;
        this.children = new ArrayList<>();
        this.move = new int[]{0, 0};

        for(int i = 0; i < ROW; i++) {
            for(int j = 0; j < COL; j++) {
                this.nodeBoard[i][j] = "";
            }
        }
    }

    public Node(Node parent) {
        this.nodeBoard = new String[ROW][COL];
        this.value = 0;
        this.children = new ArrayList<>();
        this.move = new int[]{0, 0};

        for(int i = 0; i < ROW; i++) {
            for(int j = 0; j < COL; j++) {
                this.nodeBoard[i][j] = parent.getNodeBoard()[i][j];
            }
        }
    }
}

```



```

    }

    // Setter
    public void writeNodeBoard(int row, int col, String botMarker) {
        String opponentMarker = botMarker.equals("O") ? "X" : "O";

        if(row != 0 && this.nodeBoard[row-1][col].equals(opponentMarker)) {
            this.nodeBoard[row-1][col] = botMarker;
        }

        if(row != ROW - 1 && this.nodeBoard[row+1][col].equals(opponentMarker)) {
            this.nodeBoard[row+1][col] = botMarker;
        }

        if(col != 0 && this.nodeBoard[row][col-1].equals(opponentMarker)) {
            this.nodeBoard[row][col-1] = botMarker;
        }

        if(col != COL - 1 && this.nodeBoard[row][col+1].equals(opponentMarker)) {
            this.nodeBoard[row][col+1] = botMarker;
        }

        this.nodeBoard[row][col] = botMarker;
    }

    public void setTerminalValue(String botMarker) {
        for(int i = 0; i < ROW; i++) {
            for(int j = 0; j < COL; j++) {
                if(this.nodeBoard[i][j].equals(botMarker)) {
                    this.value++;
                }
            }
        }
    }

    public void addChild(Node child) {
        this.children.add(child);
    }

    public void setValue(int value) {
        this.value = value;
    }

    public void setMove(int row, int col) {
        this.move[0] = row;
        this.move[1] = col;
    }

    // Getter
    public List<Node> getChildren() {
        return this.children;
    }

```

```

    }

    public int getValue() {
        return this.value;
    }

    public String[][] getNodeBoard() {
        return this.nodeBoard;
    }

    public int[] getMove() {
        return this.move;
    }
}

```

Berikut adalah penjelasan langkah-langkah dan konsep yang kelompok kami digunakan dalam algoritma *Minimax* dan *Alpha Beta Pruning* :

1. Inisialisasi
 - Inisialisasi objek MinimaxBot dengan parameter baris (ROW), kolom (COL), array tombol (buttons), dan jenis bot (bot).
 - Mengatur jenis lawan (opponent) berdasarkan jenis bot.
2. Fungsi *updateScore* : Digunakan untuk memperbarui skor bot.
3. Fungsi *move*:
 - Memulai pergerakan bot.
 - Inisialisasi papan permainan (*node*) dan mengisi papan berdasarkan tombol yang telah diatur.
 - Menentukan kedalaman pencarian (*depth*), waktu awal pencarian (*startTime*), dan batas waktu (*timeout*).
 - Memilih langkah awal secara acak menggunakan fungsi *randomMove*.
 - Melakukan iterasi untuk setiap kedalaman pencarian:
 - Membuat salinan node saat ini untuk menghindari modifikasi langsung.
 - Memanggil fungsi minimax untuk mencari nilai terbaik.
 - Memperbarui langkah terbaik berdasarkan nilai node anak yang memiliki nilai maksimal atau minimal (tergantung pada apakah bot adalah pemain maksimal atau minimal).
 - Mengembalikan langkah terbaik.
4. Fungsi *minimax* : Fungsi ini menerapkan algoritma *Minimax* dengan pemangkasan alpha-beta.
 - Jika kedalaman mencapai 0 atau ronde tersisa 0, menetapkan nilai terminal dan menghentikan pencarian.
 - Jika pemain adalah pemain maksimal (*maxPlayer*), maka bot (*this.bot*) melakukan gerakan dan mencari langkah terbaik yang maksimal.
 - Menambahkan gerakan berdasarkan heuristik menggunakan fungsi *addHeuristicMove*.
 - Jika tidak ada gerakan yang mungkin, menetapkan nilai terminal dan menghentikan pencarian.
 - Melakukan iterasi untuk setiap node anak dan memanggil rekursif fungsi *minimax*.
 - Memperbarui nilai node saat ini.
 - Memangkas cabang berdasarkan alpha-beta jika diperlukan.

- Jika pemain adalah pemain minimal, maka lawan (*this.opponent*) melakukan gerakan dan mencari langkah terbaik yang minimal. Konsepnya mirip dengan pemain maksimal, tetapi dengan nilai minimal dan pemangkasan alpha-beta yang berlaku.
5. Fungsi ***initializeBoard***: Inisialisasi papan node berdasarkan tombol yang telah diatur.
 6. Fungsi ***addBestMove***:
 - Menambahkan gerakan berdasarkan heuristik.
 - Menghitung jumlah musuh yang bersebelahan untuk setiap langkah yang mungkin dan memilih langkah dengan jumlah maksimum.
 7. Fungsi ***countAdjacentOpponentMarkers***: Menghitung jumlah marka musuh yang bersebelahan dengan suatu langkah.
 8. Fungsi ***randomMove***: Mengembalikan langkah acak dari langkah yang mungkin.

Algoritma ini berfokus pada pencarian heuristik dan langkah-langkah yang dapat memberikan keuntungan terbaik bagi bot. Penggunaan *Minimax* dengan pemangkasan alpha-beta membantu mengoptimalkan pencarian dan mengurangi jumlah cabang yang harus diperiksa.

1.3. Algoritma Local Search

Pada program Bot untuk aplikasi permainan Adjacency Strategy Game, telah diterapkan keempat algoritma *local search* yang diajarkan oleh mata kuliah IF 3170 – Inteligensia Buatan, yaitu: *steepest ascent hill climbing*, *sideways move hill climbing*, *stochastic hill climbing*, dan *simulated annealing*. Akan tetapi setelah melalui beberapa kali uji coba, kelompok kami memilih untuk menggunakan algoritma *sideways move hill climbing* sebagai solusi permasalahan permainan *adjacency strategy game*. *Sideways move hill climbing* memungkinkan bot untuk membuat keputusan terbaik dari instrumen-instrumen yang ada (*current state*, *successor*, dan *neighbor*) dengan tidak terlalu menyulitkan. *Steepest ascent hill climbing* tidak digunakan karena memungkinkan untuk terjadinya kondisi *stuck* pada local optima. Hal tersebut menyebabkan program akan mengalami *run time* yang cukup panjang, sehingga kurang begitu efektif untuk digunakan dalam menyelesaikan permasalahan ini. Sementara *stochastic hill climbing* tidak digunakan karena menggunakan semacam batas jumlah loop dalam melakukan komputasi. Batas loop tersebut membuat solusi yang dieksplorasi akan sangat sedikit dan sulit untuk mendapatkan solusi yang optimal. Sedangkan apabila menggunakan *simulated annealing*, diperlukan kalkulasi peluang $e^{\Delta E/T}$ serta pendefinisian nilai dari *T* (*temperature*) dan *cooling rate* untuk melakukan pemrosesan.

Namun, penerapan algoritma *sideways hill climbing* tetap menggunakan beberapa penyesuaian berdasarkan aturan permainan yang berlaku. Hal tersebut disebabkan oleh faktor ketidak mungkinan untuk tetap berada di suatu *state* yang sama terus menerus serta upaya optimalisasi untuk mendapatkan *highest value* sesuai yang diinginkan tanpa memerlukan waktu komputasi yang panjang.

```
public int[] hillClimbingSideWays(Button[][] buttons) {
    int[][] buttonValues = new int[buttons.length][buttons.length];
    Button[][] currentSolution = copyButton(buttons);
    int currentValue = calculatePoint(currentSolution);

    List<int[]> neighbors = new ArrayList<>();

    for (int i = 0; i < buttons.length; i++) {
        for (int j = 0; j < buttons[0].length; j++) {
            if (buttons[i][j].getText().equals("")) {
```

```

        Button[][] buttonTemp = updateGameBoard(i, j, currentSolution, false);
        buttonValues[i][j] = calculatePoint(buttonTemp);
        neighbors.add(new int[]{i, j});
    } else {
        buttonValues[i][j] = 0;
    }
}
}

List<int[]> bestMoves = new ArrayList<>();
int highestValue = currentValue;

for (int[] neighbor : neighbors) {
    int i = neighbor[0];
    int j = neighbor[1];
    if (buttonValues[i][j] >= highestValue) {
        if (buttonValues[i][j] > highestValue) {
            bestMoves.clear();
            highestValue = buttonValues[i][j];
        }
        bestMoves.add(new int[]{i, j});
    }
}

if (!bestMoves.isEmpty()) {
    int[] res = bestMoves.get((int) (Math.random() * bestMoves.size()));
    currentSolution = updateGameBoard(res[0], res[1], currentSolution, false);
    return res;
}

// If no strictly better moves are available, consider moving to a worse move with a probability
for (int[] neighbor : neighbors) {
    int i = neighbor[0];
    int j = neighbor[1];
    int delta = buttonValues[i][j] - currentValue;
    if (delta < 0 || Math.random() < Math.exp(-delta / 0.1)) {
        currentSolution = updateGameBoard(i, j, currentSolution, false);
        return new int[]{i, j};
    }
}

return new int[]{-1, -1}; // Return a sentinel value to indicate no valid moves.
}

private int calculatePoint(Button[][] buttons)
{
    int res = 0;
    for (int i = 0; i < buttons.length; i++)
    {
        for (int j = 0; j < buttons[0].length; j++)
        {

```

```

        if(buttons[i][j].getText().equals("O"))
        {
            res++;
        }
        else if (buttons[i][j].getText().equals("X"))
        {
            res--;
        }
    }
}
return res;
}

public Button[][] copyButton(Button[][] buttons)
{
    Button[][] copiedButton = new Button[8][8];
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            copiedButton[i][j] = new Button(buttons[i][j].getText());
        }
    }
    return copiedButton;
}

private Button[][] updateGameBoard(int i, int j, Button[][] buttonsExternal, boolean playerXTurn)
{
    Button[][] buttons = copyButton(buttonsExternal);
    // Value of indices to control the lower/upper bound of rows and columns
    // in order to change surrounding/adjacent X's and O's only on the game board.
    // Four boundaries: First & last row and first & last column.

    int startRow, endRow, startColumn, endColumn;

    if (i - 1 < 0)    // If clicked button in first row, no preceding row exists.
        startRow = i;
    else              // Otherwise, the preceding row exists for adjacency.
        startRow = i - 1;

    if (i + 1 >= 8) // If clicked button in last row, no subsequent/further row exists.
        endRow = i;
    else            // Otherwise, the subsequent row exists for adjacency.
        endRow = i + 1;

    if (j - 1 < 0)    // If clicked on first column, lower bound of the column has been reached.
        startColumn = j;
    else
        startColumn = j - 1;
}

```

```

        if (j + 1 >= 8) // If clicked on last column, upper bound of the column has been reached.
            endColumn = j;
        else
            endColumn = j + 1;

        // Search for adjacency for X's and O's or vice versa, and replace them.
        // Update scores for X's and O's accordingly.
        for (int x = startRow; x <= endRow; x++) {
//            System.out.println("x"+x);
            buttons = this.setPlayerScore(x, j, buttons, playerXTurn);
        }

        for (int y = startColumn; y <= endColumn; y++) {
//            System.out.println("y"+y);
            buttons = this.setPlayerScore(i, y, buttons, playerXTurn);
        }

        return buttons;
    }

    private Button[][] setPlayerScore(int i, int j, Button[][] buttonsExternal, boolean playerXTurn)
    {
        Button[][] buttonsCopy = copyButton(buttonsExternal);
        if (playerXTurn)
        {
            if (buttonsCopy[i][j].getText().equals("O"))
            {
                buttonsCopy[i][j].setText("X");
            }
        } else if (buttonsCopy[i][j].getText().equals("X"))
        {
            buttonsCopy[i][j].setText("O");
        }
        return buttonsCopy;
    }
}

```

Pada kode di atas, terdapat beberapa fungsi antara seperti copyButton yang digunakan untuk melakukan proses *duplicate* dari seluruh kondisi *board* permainan, updateGameBoard untuk mengubah value *board* disekitar *button* yang dipilih, setPlayerScore untuk mengubah value dari *board* pada *button* yang dipilih, calculatePoint untuk melakukan perhitungan poin kedua *player*. Seluruh fungsi antara tersebut digunakan untuk melakukan perhitungan oleh *bot*, sehingga dapat diperoleh hasil pemilihan *button* harus merubah kondisi *board* permainan. Sementara itu, fungsi hillClimbSideWays akan memanfaatkan seluruh fungsi antara tersebut untuk melakukan komputasi langkah yang hendak diambil. Pertama, program akan melakukan inisialisasi variabel-variabel yang diperlukan. Program kemudian akan membuat sebuah *array list neighbor* yang berisi *button* pada *board* dengan status belum diisi oleh *player* manapun. Setiap *button* tersebut akan dievaluasi nilainya. Program kemudian akan menentukan *successor* dengan nilai tertinggi yang dapat diraih. Apabila pada *highest value* diperoleh lebih dari satu *button* dengan nilai value yang sama, program akan memilih secara acak *button* yang hendak dipilih agar pergerakan yang dilakukan lebih

bervariasi dan tidak monoton. Pada program juga ditambahkan pergerakan *random* apabila tidak diperoleh *successor* yang sesuai.

Berikut adalah penjelasan langkah-langkah dan konsep yang kami digunakan dalam algoritma *Hill Climbing Sideways* :

1. Inisialisasi
 - Membuat matriks `buttonValues` untuk menyimpan nilai heuristik dari setiap langkah yang mungkin.
 - Membuat salinan solusi saat ini (`currentSolution`).
 - Menghitung nilai saat ini (`currentValue`) dengan fungsi `calculatePoint`.
2. Generasi Tetangga:
 - Untuk setiap sel yang kosong di papan permainan, menghasilkan tetangga dengan melakukan langkah ke sana dan menghitung nilai heuristiknya.
 - Menyimpan nilai heuristik setiap tetangga dalam matriks `buttonValues` dan daftar `neighbors`.
3. Seleksi Tetangga Terbaik:
 - Memilih tetangga dengan nilai heuristik tertinggi sebagai langkah terbaik.
 - Jika terdapat beberapa tetangga dengan nilai tertinggi, dipilih secara acak.
4. Pemilihan Langkah:
 - Jika terdapat tetangga dengan nilai heuristik yang lebih tinggi dari nilai saat ini, maka pemain bergerak ke salah satu dari tetangga tersebut secara acak.
 - Jika tidak, pertimbangkan kemungkinan bergerak ke tetangga yang memiliki nilai heuristik lebih rendah daripada nilai saat ini. Pemilihan ini dapat dilakukan dengan probabilitas tergantung pada seberapa jauh nilai heuristik yang lebih buruk (δ) dan parameter probabilitas ($\text{Math.exp}(-\delta / 0.1)$).
5. Penanganan Kesetaraan:
 - Jika terdapat lebih dari satu tetangga dengan nilai tertinggi, algoritma memilih satu dari mereka secara acak.
 - Penanganan Kemungkinan Bergerak ke Langkah Buruk:
 - Jika tidak ada langkah yang memperbaiki nilai saat ini, pertimbangkan kemungkinan bergerak ke tetangga yang memiliki nilai heuristik lebih rendah dengan probabilitas tertentu ($\text{Math.exp}(-\delta / 0.1)$).
6. Pengembalian Hasil:
 - Mengembalikan langkah yang dipilih dalam bentuk array `int`, misalnya `[i, j]`.
 - Jika tidak ada langkah yang valid, mengembalikan nilai sentinel `[-1, -1]`.

Algoritma ini mencoba untuk melakukan pergerakan ke langkah yang meningkatkan nilai heuristik (jika ada), atau kemungkinan bergerak ke langkah yang memiliki nilai heuristik lebih rendah dengan probabilitas tertentu. Penggunaan probabilitas ini memungkinkan algoritma untuk keluar dari minimum lokal dan menjelajahi solusi yang berpotensi lebih baik.

1.4. Algoritma *Genetic*

Algoritma genetik adalah algoritma pencarian yang terinspirasi oleh teori evolusi Charles Darwin tentang seleksi alam. Algoritma ini digunakan untuk menyelesaikan masalah optimasi dan pencarian yang kompleks. Algoritma genetik mencoba menemukan solusi terbaik untuk masalah dengan mengiterasi dan

mengubah populasi solusi potensial secara berulang. Proses dalam algoritma ini ada 3 tahap, yaitu *selection*, *crossover*, dan *mutation*. Pada awalnya akan dilakukan *generate* N buah gen secara random. Dari kumpulan gen tersebut akan dicari nilai *fitness function* untuk setiap gen yang ada. Nilai *fitness function* akan digunakan untuk menentukan gen mana yang akan disilangkan. Kemudian setelah dipilih gen untuk disilangkan, akan dilakukan tahap *crossover*. *Crossover* adalah tahap untuk menukar bagian dari *parent* sehingga menghasilkan dua individu baru. Setelah dilakukan *crossover*, akan dilakukan tahap *mutation*. *Mutation* adalah tahap untuk mengubah salah satu nilai pada individu baru dengan nilai acak untuk menciptakan keragaman populasi. Berikut adalah implementasi kode program untuk menyelesaikan persoalan ini:

```
import java.util.ArrayList;
import java.util.Arrays;
import javafx.scene.control.Button;

public class BotGenetic extends Bot{
    private Button[][] buttons;

    public BotGenetic(Button[][] buttons){
        this.buttons = buttons;
    }

    public int[] move(Button[][] buttons, int roundsLeft, int playerOScore, int playerXScore, boolean
    playerXTurn) {
        ArrayList<ArrayList<Integer>> unfilled = new ArrayList<>();

        for(int i= 0; i< 8; i++){
            for(int j= 0; j< 8; j++){
                if(buttons[i][j].getText().equals("")){
                    unfilled.add(new ArrayList<>(Arrays.asList(i, j)));
                }
            }
        }
        if(unfilled.size() < 10){
            int random = (int) (Math.random() * unfilled.size());
            return new int[]{unfilled.get(random).get(0), unfilled.get(random).get(1)};
        }
        ArrayList<ArrayList<Integer>> genePool = getGenePool(5, 20);
        ArrayList<Integer> fitness = calculateFitness(genePool, true);

        ArrayList<Integer> wheel = createWheel(fitness);

        ArrayList<ArrayList<Integer>> children = new ArrayList<>();
        for(int i = 0; i < 200; i++){
            ArrayList<Integer> parent1 = genePool.get(getOneParent(wheel));
            ArrayList<Integer> parent2 = genePool.get(getOneParent(wheel));

            int crossOverPoint = (int) (Math.random() * parent1.size());
            ArrayList<ArrayList<Integer>> crossOver = crossOver(parent1, parent2, crossOverPoint);
```



```

        int mutationPoint = (int) (Math.random() * parent1.size());
        mutation(crossOver.get(0), mutationPoint);
        mutation(crossOver.get(1), mutationPoint);
        children.add(crossOver.get(0));
        children.add(crossOver.get(1));
    }

    ArrayList<Integer> fitnessSolution = calculateFitness(children, false);

    int max = -99999;
    int indexMax = -1;
    for(int i = 0; i < fitnessSolution.size(); i++){
        if(fitnessSolution.get(i) > max){
            indexMax = i;
        }
    }

    int bestSolution = children.get(indexMax).get(0);
    int row = bestSolution / 8;
    int col = bestSolution % 8;

    return new int[]{row, col};
}

// public .
public ArrayList<Integer> getOneChromosome(int length){
    ArrayList<Integer> Chromosome = new ArrayList<>();
    for (int j = 0; j<length; j++){
        int tile = (int) (Math.random()*64);
        int row = tile / 8;
        int col = tile % 8;
        while(!this.buttons[row][col].getText().equals("") || Chromosome.contains(tile)){
            // System.out.println("masuk");
            tile = (int) (Math.random()*64);
            row = tile / 8;
            col = tile % 8;
        }
        Chromosome.add(tile);
    }
    return Chromosome;
}

public ArrayList<ArrayList<Integer>> getGenePool(int length, int N){
    //length = panjang chromosome atau level pohon
    //N = banyaknya random chromosome
    ArrayList<ArrayList<Integer>> GenePool = new ArrayList<>();
    for(int i = 0; i<N; i++){
        ArrayList<Integer> Chromosome = getOneChromosome(length);
        GenePool.add(Chromosome);
    }
}

```

```

// System.out.println(GenePool);
return GenePool;
}

public ArrayList<Integer> calculateFitness(ArrayList<ArrayList<Integer>> genePool, boolean
isForSelection) {
    ArrayList<Integer> fitness = new ArrayList<Integer>();

    int point = calculatePoint(buttons);
    for(int i=0; i<genePool.size(); i++){
        int temp = -1;
        while(temp <= 0){
            temp = point;
            Button[][] copiedButton = copyButton(buttons);
            for(int j = 0; j < genePool.get(i).size(); j++) {
                if(j%2 == 0){
                    int row = genePool.get(i).get(j) /8;
                    int column = genePool.get(i).get(j) %8;
                    copiedButton[row][column].setText("O");
                    temp ++;
                    if(isValidButton(row-1, column) && copiedButton[row-
1][column].getText().equals("X")){
                        copiedButton[row-1][column].setText("O");
                        temp += 2;
                    }
                    if(isValidButton(row+1, column) &&
copiedButton[row+1][column].getText().equals("X")){
                        copiedButton[row+1][column].setText("O");
                        temp += 2;
                    }
                    if( isValidButton(row, column-1) && copiedButton[row][column-
1].getText().equals("X")){
                        copiedButton[row][column-1].setText("O");
                        temp += 2;
                    }
                    if(isValidButton(row, column+1) &&
copiedButton[row][column+1].getText().equals("X")){
                        copiedButton[row][column+1].setText("O");
                        temp += 2;
                    }
                }else{
                    int row = genePool.get(i).get(j) /8;
                    int column = genePool.get(i).get(j) %8;
                    copiedButton[row][column].setText("X");
                    temp --;
                    if(isValidButton(row-1, column) && copiedButton[row-
1][column].getText().equals("O")){
                        copiedButton[row-1][column].setText("X");
                        temp -= 2;
                    }
                }
            }
        }
    }
}

```

```

        if(isValidButton(row+1, column) &&
copiedButton[row+1][column].getText().equals("O")){
            copiedButton[row+1][column].setText("X");
            temp -= 2;
        }
        if(isValidButton(row, column-1) && copiedButton[row][column-
1].getText().equals("O")){
            copiedButton[row][column-1].setText("X");
            temp -= 2;
        }
        if(isValidButton(row, column+1) &&
copiedButton[row][column+1].getText().equals("O")){
            copiedButton[row][column+1].setText("X");
            temp -= 2;
        }
    }
}
if(isForSelection){
    if(temp <= 0){
        genePool.set(i, getOneChromosome(genePool.get(0).size()));
    }
    }else{
        if(temp <= 0){
            temp = 9999;
        }
    }
}

// System.out.println(temp);
if(!isForSelection && temp == 9999){
    temp = 0;
}
fitness.add(temp);

}
// System.out.println("breakkk");
// System.out.println(fitness);
return fitness;
}

public ArrayList<Integer> createWheel(ArrayList<Integer> fitness){
    int sum = 0;
    for(int i = 0; i < fitness.size(); i++){
        sum += fitness.get(i);
    }

    ArrayList<Integer> wheel = new ArrayList<Integer>();
    int wheelCount = 0;
    for(int i = 0; i < fitness.size(); i++){

```

```

        double percentage = (fitness.get(i)/(sum + 1)) *100;
        wheelCount += percentage;
        wheel.add(wheelCount);
    }
    return wheel;
}

public int getOneParrent(ArrayList<Integer> wheel){
    double random = Math.random()*100;

    int index = 0;
    for(int i=0; i< wheel.size(); i++){
        if(random <= wheel.get(i)){
            index = i;
            break;
        }
    }
    return index;
}

public Button[][] copyButton(Button[][] buttons)
{
    Button[][] copiedButton = new Button[8][8];
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            copiedButton[i][j] = new Button(buttons[i][j].getText());
        }
    }
    return copiedButton;
}

private boolean isValidButton(int i, int j){
    return (i>=0 && i<8) && (j>=0 && j<8);
}

private int calculatePoint(Button[][] buttons)
{
    int res = 0;
    for (int i = 0; i < buttons.length ; i++)
    {
        for (int j = 0; j < buttons[0].length; j++)
        {
            if(buttons[i][j].getText().equals("O"))
            {
                res++;
            }
            else if (buttons[i][j].getText().equals("X"))
            {
                res--;
            }
        }
    }
}

```

```

    }

    }
}
return res;
}

```

```

public ArrayList<ArrayList<Integer>> crossover(ArrayList<Integer> parrent1, ArrayList<Integer>
parrent2, int crossoverPoint){
    ArrayList<ArrayList<Integer>> crossoverResult = new ArrayList<>();
    ArrayList<Integer> crossover1 = new ArrayList<>();
    ArrayList<Integer> crossover2 = new ArrayList<>();

    int length = parrent1.size();

    for (int i = 0; i<crossoverPoint; i++){
        int val = parrent1.get(i);
        int row = val / 8;
        int col = val % 8;
        while(!this.buttons[row][col].getText().equals("") || crossover1.contains(val)){
            val = (int) (Math.random() * 64);
            row = val / 8;
            col = val % 8;
        }
        crossover1.add(val);
    }
    for (int i = crossoverPoint; i<length; i++){
        int val = parrent2.get(i);
        int row = val / 8;
        int col = val % 8;
        while(!this.buttons[row][col].getText().equals("") || crossover1.contains(val)){
            val = (int) (Math.random() * 64);
            row = val / 8;
            col = val % 8;
        }
        crossover1.add(val);
    }

    for (int i = 0; i<crossoverPoint; i++){
        int val = parrent2.get(i);
        int row = val / 8;
        int col = val % 8;
        while(!this.buttons[row][col].getText().equals("") || crossover2.contains(val)){
            val = (int) (Math.random() * 64);
            row = val / 8;
            col = val % 8;
        }
        crossover2.add(val);
    }
    for (int i = crossoverPoint; i<length; i++){
        int val = parrent1.get(i);

```

```

        int row = val / 8;
        int col = val % 8;
        while(!this.buttons[row][col].getText().equals("") || crossOver2.contains(val)){
            val = (int) (Math.random() * 64);
            row = val / 8;
            col = val % 8;
        }
        crossOver2.add(val);
    }

    crossOverResult.add(crossOver1);
    crossOverResult.add(crossOver2);

    return crossOverResult;
}

public void mutation (ArrayList<Integer> child, int mutationPoint){
    int randomVal = (int) (Math.random() * 64);
    int row = randomVal / 8;
    int col = randomVal % 8;
    while(!this.buttons[row][col].getText().equals("") || child.contains(randomVal)){
        randomVal = (int) (Math.random() * 64);
        row = randomVal / 8;
        col = randomVal % 8;
    }

    child.set(mutationPoint, randomVal);
}
}

```

Berikut adalah penjelasan dan langkah langkah untuk menyelesaikan persoalan ini menggunakan algoritma genetik:

1. Inisialisasi :

- Menggenerate banyak populasi yang terdiri dari N buah kromosom dengan panjang *length*. Pada implementasi, kelompok kami menggunakan 20 buah kromosom dengan tiap kromosom memiliki panjang 5.
- Kromosom direpresentasikan dengan urutan langkah yang dipilih. Sebagai contoh : terdapat kromosom 1 2 3 4, artinya bot akan memilih tile dengan nomor 1, kemudian lawan akan memilih tile nomor 2, bot akan melanjutkan dengan memilih tile nomor 3 dan dilanjutkan lawan memilih tile nomor 4.

2. Evaluasi :

- Evaluasi dilakukan dengan cara menghitung *fitness function* untuk setiap kromosom. *Fitness function* pada implementasi ini dirumuskan sebagai, nilai akhir yang akan didapatkan setelah bot dan lawan melakukan urutan langkah sesuai pada kromosom.
- Kemudian akan dibuat sebuah *wheel* yang nantinya akan untuk memilih kromosom secara acak. *Wheel* dibuat dengan cara membagi nilai *fitness function* tiap kromosom dengan total *fitness function* semua kromosom. Sebagai contoh : terdapat 4 kromosom dengan nilai *fitness function* masing masing adalah 2, 4, 2, 2. Maka *Wheel*

yang didapat adalah : kromosom 1 dipilih saat nilai random 0 sampai 0.2, kromosom 1 dipilih saat nilai random 0.2 sampai 0.6, kromosom 3 dipilih saat nilai random 0.6 sampai 0.8, dan kromosom 4 dipilih saat nilai random 0.8 sampai 1. Setelah mendapatkan *wheel* akan dilakukan random untuk memilih parrent parrent untuk disilangkan.

3. Seleksi :

- Akan dilakukan beberapa random untuk menentukan pasangan parrent untuk disilangkan.

4. *Crossover* :

- Proses ini adalah proses menyuatuakan bagian awal parrent 1 dan bagian akhir parrent 2 untuk menciptakan anak 1. Kemudian akan disatukan juga bagian awal parrent 2 dan bagian akhir parrent 1. Pemilihan bagian awal dan akhir ditentukan oleh random *crossover point*, bagian awal adalah nilai dari indeks ke 0 hingga indeks ke crossover point. Sedangkan bagian akhir merupakan indeks ke crossover point hingga indeks terakhir.

5. *Mutation* :

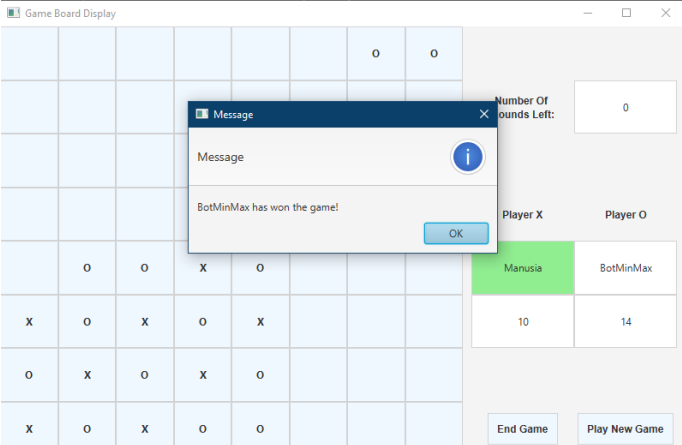
- Proses ini adalah proses untuk mengganti salah satu nilai pada individu yang dihasilkan dari *crossover*. Indeks yang akan diubah akan ditentukan secara random, dan nilai penggantinya juga akan didapatkan dengan cara random.

BAB II

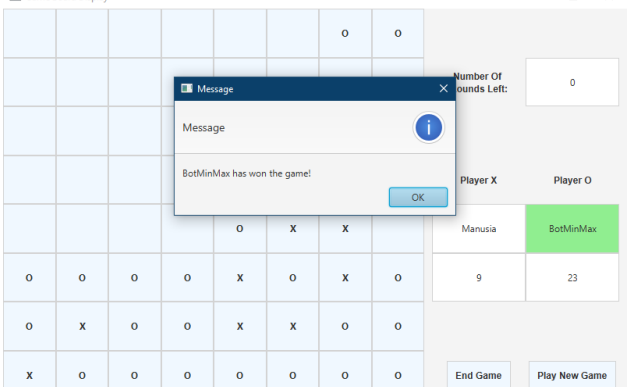
Hasil Pertandingan

2.1. Minimax vs Manusia

i. Percobaan 1 (Bot \neq *Run First*)

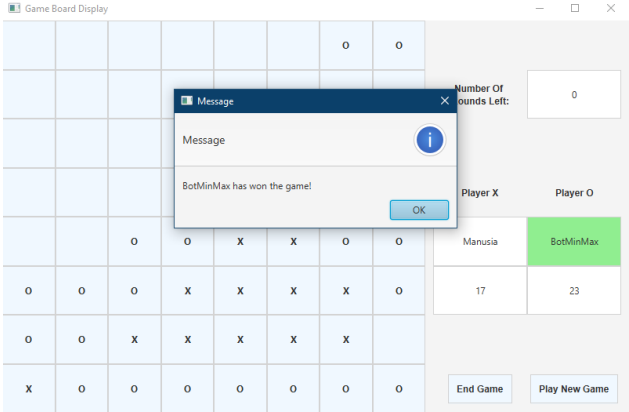
Ronde	Hasil	Pemenang
8		Bot

ii. Percobaan 2 (Bot = *Run First*)

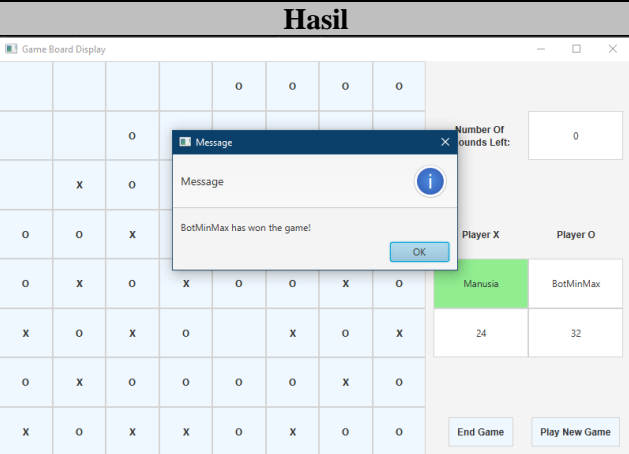
Ronde	Hasil	Pemenang
12		Bot

iii. Percobaan 3 (Bot = *Run First*)

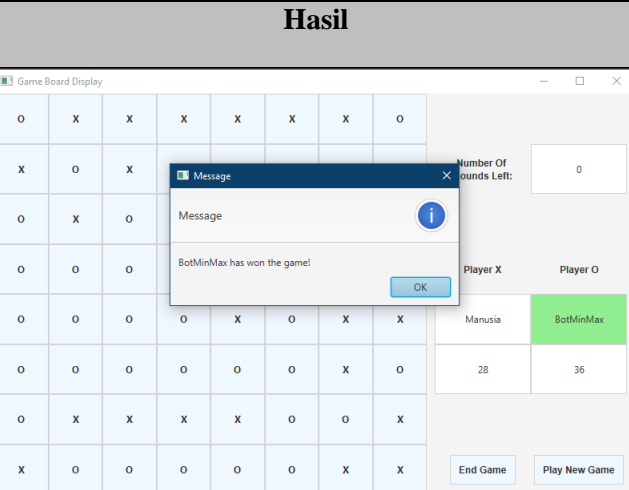
Ronde	Hasil	Pemenang
-------	-------	----------

16		Bot
----	--	-----

iv. Percobaan 4 (Bot \neq Run First)

Ronde	Hasil	Pemenang
24		Bot

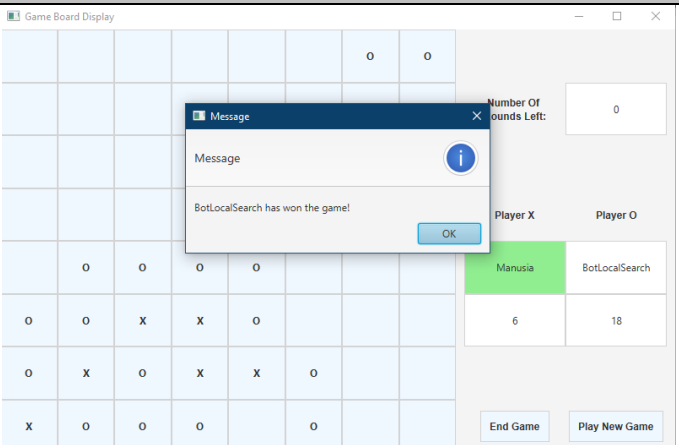
v. Percobaan 5 (Bot = Run First)

Ronde	Hasil	Pemenang
28		Bot

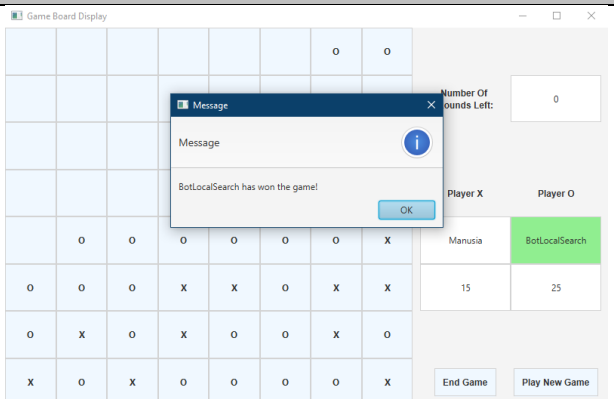
Bot	Win Status	Win Rate
MinMax	5/5	100%

2.2. Local Search vs Manusia

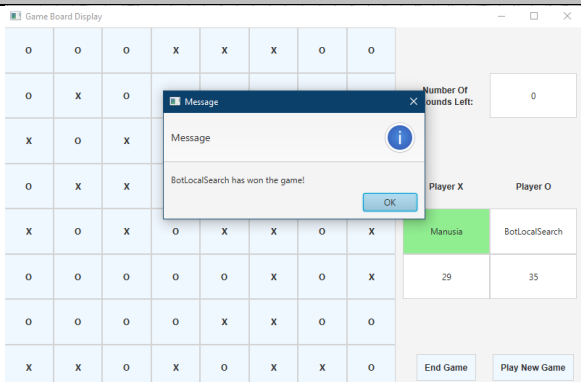
i. Percobaan 1 (Bot ≠ Run First)

Ronde	Hasil	Pemenang
8		Bot

ii. Percobaan 2 (Bot = Run First)

Ronde	Hasil	Pemenang
16		Bot

iii. Percobaan 3 (Bot ≠ Run First)

Ronde	Hasil	Pemenang
28		Bot

Bot	Win Status	Win Rate
-----	------------	----------

Local Search	3/3	100%
--------------	-----	------

2.3. Minimax vs Local Search

i. Percobaan 1 (Bot \neq Run First)

Ronde	Hasil	Pemenang
8		Local Search

ii. Percobaan 2 (Bot = Run First)

Ronde	Hasil	Pemenang
16		Draw

iii. Percobaan 3 (Bot \neq Run First)

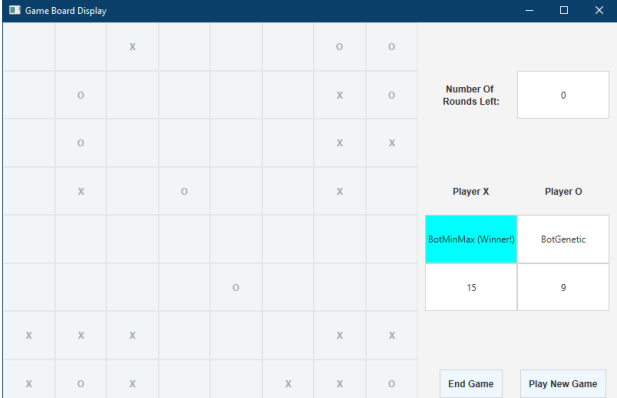
Ronde	Hasil	Pemenang
28		MinMax

Bot	Win Status	Win Rate
Minimax	1/3	33.333%

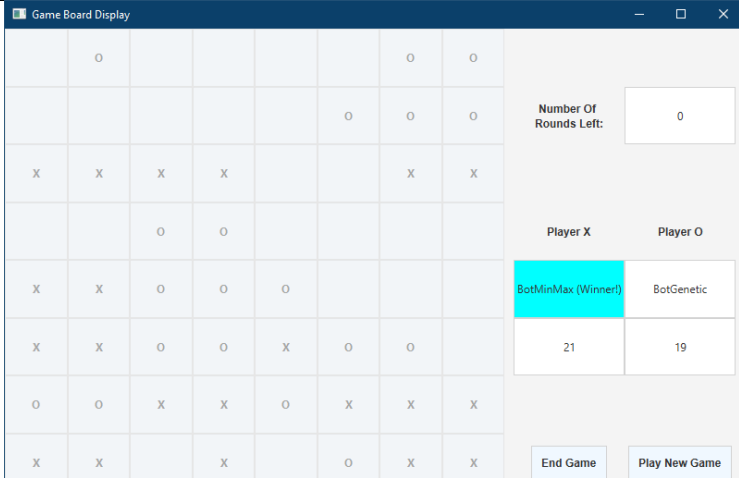
Bot	Win Status	Win Rate
Local Search	1/3	33.333%

2.4. Minimax vs Genetic

i. Percobaan 1 (Bot \neq Run First)

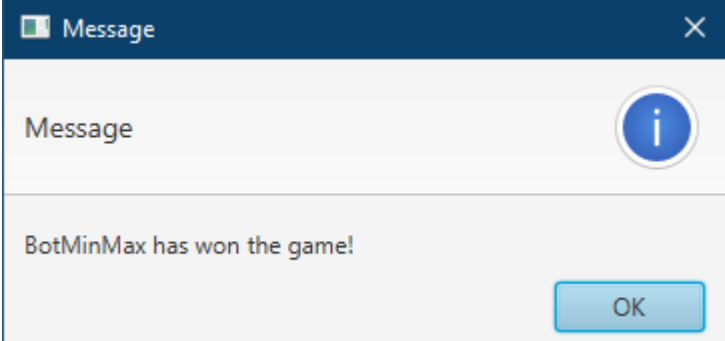
Ronde	Hasil	Pemenang
8		MinMax

ii. Percobaan 2 (Bot = Run First)

Ronde	Hasil	Pemenang
16		MinMax

iii. Percobaan 3 (Bot \neq Run First)

Ronde	Hasil	Pemenang
-------	-------	----------

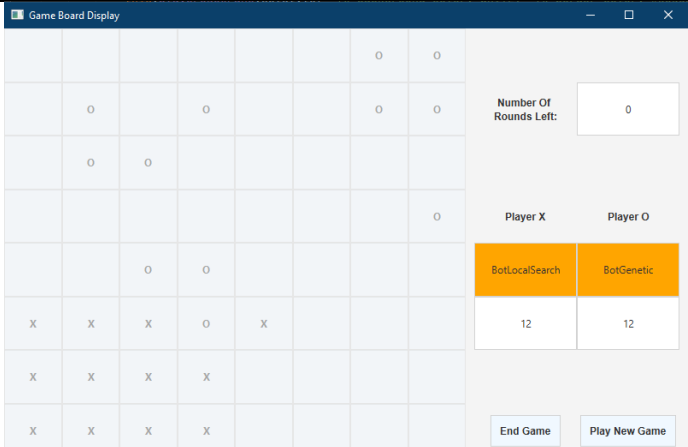
28		MinMax
----	--	--------

Bot	Win Status	Win Rate
Minmax	3/3	100%

Bot	Win Status	Win Rate
Genetic	0/3	0%

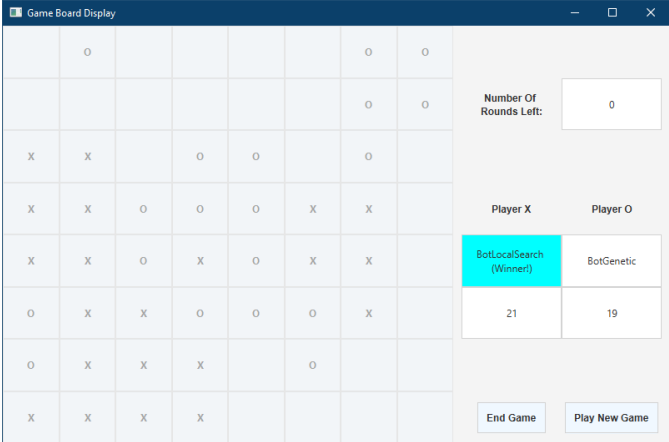
2.5. Local Search vs Genetic

- i. Percobaan 1 (Bot \neq Run First)

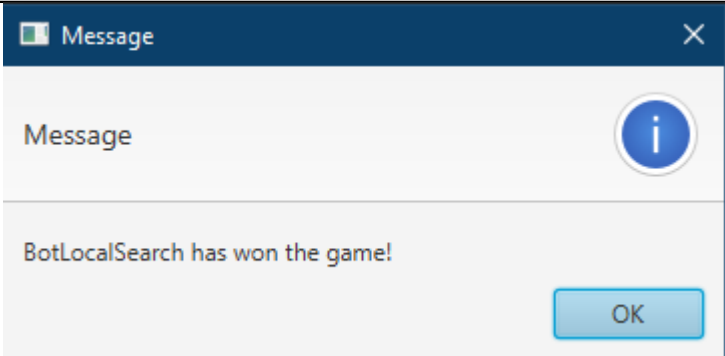
Ronde	Hasil	Pemenang
8		Draw

- ii. Percobaan 2 (Bot = Run First)

Ronde	Hasil	Pemenang
-------	-------	----------

16		Local Search
----	--	--------------

iii. Percobaan 3 (Bot \neq Run First)

Ronde	Hasil	Pemenang
28		LocalSearch

Bot	Win Status	Win Rate
Local Search	2/3	66.7%

Bot	Win Status	Win Rate
Genetic	0/3	0%

BAB III

Pembagian Tugas

No	Pekerjaan	Penanggung Jawab
1	<i>Objective Function</i>	13521122, 13521152
2	<i>Minimax and Alpha-Beta Pruning Algorithm</i>	13521152
3	<i>Local Search Algorithm</i>	13521166
4	<i>Genetic Algorithm</i>	13521122
5	Laporan	13521152, 13521166, 13521168
6	Repository	13521168

Referensi

- KK IF – Teknik Informatika – STEI ITB. 2023, “Diktat Modul 3: *Beyond Classical Search*”, https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804818592_IF3170_Materi03_Seg01_BeyondClassicalSearch_LocalSearch.pdf , diakses pada 16 Oktober 2023.
- KK IF – Teknik Informatika – STEI ITB. 2023, “Diktat Modul 4: *Adversarial Search*”, https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693969282598_IF3170_Materi04_AdversarialSearch.pdf , diakses pada 16 Maret 2023.
- Hong ,Tzung-Pei. “*Adversarial Search by Evolutionary Computation*”, https://www.researchgate.net/publication/220375025_Adversarial_Search_by_Evolutionary_Computation, diakses pada 16 Oktober 2023.

Pranala Terkait

Link Repository:

https://github.com/satrianababan/Tubes1_13521122.git