

# **Tugas Besar 1 IF3170 Intelegensi Buatan**

## **“Minimax Algorithm and Alpha Beta Pruning in**

### **Adjacency Strategy Game”**

Disusun Oleh:

Fajar Maulana Herawan	13521080
Bagas Aryo Seto	13521081
Shidqi Indy Izhari	13521097
Muhammad Rizky Syaban	13521119



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

**2023**

## **Daftar Isi**

Struktur Program Permainan.....	3
Rencana Kelas/ Fungsi.....	8
Objective Function.....	8
Minimax Alpha-Beta Pruning.....	9
Local Search.....	10
Genetic Algorithm.....	11
Referensi.....	11

## Struktur Program Permainan

Struktur game ini dibuat berdasarkan repository github dan asisten lab gaib <https://github.com/GAIB20/adversarial-adjacency-strategy-game>. Main program berada di *class main* di dalam file Main.java. Di dalam *class* tersebut, terdapat *Metode start* yang berisi alur permainan. Berikut struktur lengkap dari program permainan yang berada dalam folder src.

Tabel 1 : Struktur Program

Nama File	Nama Class	Method/ Attribute	Keterangan
Main.java	Main	start	Metode yang diwarisi dari <i>class Application</i> . Metode ini akan dipanggil saat aplikasi JavaFX dimulai. Ia menerima satu parameter, yaitu objek Stage yang mewakili jendela utama aplikasi.
		main	Metode utama yang digunakan sebagai titik masuk utama aplikasi Java. Metode main ini akan dijalankan saat aplikasi dimulai.
InputFrameController.java	InputFrameController	isBotFirst	Atribut ini adalah objek CheckBox yang digunakan untuk memungkinkan pengguna memilih apakah bot akan bermain pertama atau tidak.
		player1	Atribut yang dianotasi dengan @FXML yang menghubungkan atribut dengan elemen TextField dalam file FXML yang memiliki ID yang sama yakni player1
		player2	Atribut yang dianotasi

			dengan @FXML yang menghubungkan atribut dengan elemen TextField dalam file FXML yang memiliki ID yang sama yakni player2
	numberOfRounds		atribut yang menghubungkan elemen ComboBox dalam file FXML dengan ID yang sama yakni jumlah ronde
	initialize		Metode ini dianotasi dengan @FXML dan digunakan untuk menginisialisasi controller setelah elemen-elemen yang sesuai dari FXML telah dihubungkan.
	reset		Metode ini dianotasi dengan @FXML dan digunakan untuk mengatur ulang isian dalam elemen-elemen input (player1, player2, numberOfRounds) ke nilai default (biasanya kosong).
	play		Metode ini menghandle tindakan saat tombol play ditekan
	isInputFieldValidated		Metode private yang digunakan untuk memvalidasi input yang dimasukkan oleh pengguna
OutPutFrameController.java	OutPutFrameController	gameBoard	Atribut yang dianotasi dengan @FXML yang menghubungkan atribut dengan elemen GridPane dalam file FXML yang

		berfungsi sebagai game board
	scoreBoard	Atribut yang dianotasi dengan @FXML yang menghubungkan atribut dengan elemen GridPane dalam file FXML yang berfungsi sebagai score board
	roundsLeftLabel	Atribut yang dianotasi dengan @FXML yang menghubungkan atribut dengan elemen Label dalam file FXML yang berfungsi sebagai label ronde yang tersisa
	playerXName	Atribut yang dianotasi dengan @FXML yang menghubungkan atribut dengan elemen Label dalam file FXML yang berfungsi sebagai label nama player X
	playerOName	Atribut yang dianotasi dengan @FXML yang menghubungkan atribut dengan elemen Label dalam file FXML yang berfungsi sebagai label nama player O
	playerXBoxPane	Atribut yang dianotasi dengan @FXML yang menghubungkan atribut dengan elemen HBox dalam file FXML yang berfungsi sebagai box pane X
	playerOBoxPane	Atribut yang dianotasi dengan @FXML yang

		menghubungkan atribut dengan elemen HBox dalam file FXML yang berfungsi sebagai box pane O
	playerXScoreLabel	Atribut yang dianotasi dengan @FXML yang menghubungkan atribut dengan elemen Label dalam file FXML yang berfungsi sebagai score X
	playerOScoreLabel	Atribut yang dianotasi dengan @FXML yang menghubungkan atribut dengan elemen Label dalam file FXML yang berfungsi sebagai score O
	playerXTurn	Atribut untuk status giliran player X
	playerXScore	Atribut untuk jumlah score X
	playerOScore	Atribut untuk jumlah score O
	roundsLeft	Atribut untuk jumlah ronde tersisa
	isBotFirst	Atribut boolean untuk status bot duluan
	bot	Atribut yang berfungsi sebagai bot yang bermain
	buttons	Atribut yang merepresentasikan papan permainan
	getInput	Metode untuk memasukan nama dan jumlah ronde
	initialize	Metode untuk membuat board permainan sebesar

			8x8 dan menginisialisasi X dan O di ujung sekaligus scoreboard
	selectedCoordinate		Metode untuk mengolah button yang diklik
	updateGameBoard		Metode untuk mengupdate isi board game
	setPlayerScore		Metode untuk menentukan skor player dengan perhitungan
	endOfGame		Metode untuk menentukan pemenang
	endGame		Metode untuk keluar dari permainan
	playNewGame		Metode untuk membuka kembali <i>InputFrame</i>
	moveBot		Metode untuk menjalankan bot
Bot.java	Bot	move	Metode untuk langkah dari bot

Selain kelas-kelas yang telah disebutkan, terdapat pula file XML InputFrame dan OutputFrame sebagai tampilan dari permainan.

## Objective Function

Poin pada permainan Adjacency Strategy Game yang memiliki ukuran papan 8x8 ini dihitung berdasarkan jumlah simbol yang dimiliki oleh seorang pemain. Maka, jumlah banyaknya simbol harus menjadi salah satu elemen fungsi objektif. Karena kedua pemain memiliki objektif yang sama, yaitu memaksimalkan jumlah simbol, maka dapat dibuat heuristik fungsi objektif yaitu jumlah simbol pemain 1 dikurang jumlah simbol pemain 2. Dengan menggunakan fungsi objektif tersebut, pemain 1 akan berusaha mendapatkan poin objektif tertinggi, sedangkan pemain 2 akan berusaha mendapatkan poin objektif terendah. Akan tetapi, perlu diingat peraturan permainan Adjacency Strategy Game yaitu apabila sebuah kotak kosong diisi, seluruh kotak di sekitar yang sudah terisi simbol musuh akan berubah menjadi simbol pemain. Jadi pada giliran maximizer, nilai fungsi objektif perlu dikurang jumlah kotak maksimum yang mungkin diubah oleh minimizer. Sedangkan pada giliran minimizer, nilai fungsi objektif perlu ditambah jumlah kotak maksimum yang mungkin diubah oleh maximizer.

$$f(r) = \begin{cases} s_1 - s_2 - 2 \times c_2 + 1, & \text{maximizer turn} \\ s_1 - s_2 + 2 \times c_1 + 1, & \text{minimizer turn} \\ s_1 - s_2, & \text{base} \end{cases}$$

Gambar 1 : Formula Perhitungan Fungsi Objektif

Dengan r merupakan ronde, s1 merupakan jumlah simbol pemain 1, s2 merupakan jumlah simbol pemain 2, c1 merupakan jumlah kotak maksimum yang mungkin diubah oleh maximizer, dan c2 merupakan jumlah kotak maksimum yang mungkin diubah oleh minimizer. Base merupakan basis permainan, yaitu ketika papan permainan penuh atau ronde permainan telah habis.

```
/**  
*  
* Return value of current state based on objective function
```

```

*  $S_1 - S_2 - 2 \times C_2 + 1$ , for maximizer turn
*  $S_1 - S_2 + 2 \times C_1 + 1$ , for minimizer turn
*  $S_1 - S_2$ , base ( terminal state )
* where base is there is nothing rounds left or board is full
* where  $S_1$  represents the number of symbols for our symbol,
*  $S_2$  represents the number of symbols for enemy symbol,
*  $C_1$  represents the maximum number of boxes that can be changed by the
maximizer,
* and  $C_2$  represents the maximum number of boxes that can be changed by the
minimizer
*
* @param turn , -1 : minimizer (enemy), 1 : maximizer (us), 0 : base
* @param map , map on that current state
* @return value of current stateAMO
*
*/
protected int ObjectiveFunction(int turn, Button[][] map) {
    if(turn == 1){
        return this.countSymbol(false, map) - this.countSymbol(true, map) -
               (2*calculateMaxChangeableBoxes(map, true)) + 1;
    } else if (turn == -1){
        return this.countSymbol(false, map) - this.countSymbol(true, map) +
               (2*calculateMaxChangeableBoxes(map, false))+1;
    } else if (turn == 0) {
        return this.countSymbol(false, map) - this.countSymbol(true, map);
    }
    return Integer.MIN_VALUE;
}

/**
*
* Return sum of symbol player
*
* @param isEnemy ,isEnemy: true (enemy), false (our)
* @param map, map in that current state
* @return sum of symbol player or bot
*
*/
protected int countSymbol(Boolean isEnemy ,Button[][] map){
    int sum = 0;
    for (int i = 0; i < ROW; i++){
        for (int j = 0; j < COL; j++) {

```

```

        if (isEnemy) {
            if (map[i][j].getText().equals(this.enemySymbol)) {
                sum++;
            }
        } else if (map[i][j].getText().equals(this.ourSymbol)) {
            sum++;
        }
    }
    return sum;
}

/**
*
* return the maximum number of boxes that can be changed by enemy or bot
* if enemy == true then it will return maximum number of our that can
change enemy symbol
*
* @param map, map on that state
* @param isEnemy ,isEnemy: true (enemy), false (our)
* @return maximum number of boxes that can be changed
*
*/
protected int calculateMaxChangeableBoxes(Button[][] map, Boolean isEnemy){
    int MaxChangeable = 0;
    for (int i = 0; i < ROW; i++){
        for (int j = 0; j < COL; j++) {
            int adj = 0; // adjacent from that position
            if (map[i][j].getText().equals("")){
                int startRow, endRow, startColumn, endColumn;

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

                // If selected button in Last row,
                // no subsequent/further row exists.
                if (i + 1 >= ROW)
                    endRow = i;
                else
                    endRow = i + 1;
                for (int k = startRow; k < endRow; k++)
                    for (int l = startColumn; l < endColumn; l++)
                        if (map[k][l].getText().equals(""))
                            adj++;
            }
            MaxChangeable = Math.max(MaxChangeable, adj);
        }
    }
    return MaxChangeable;
}

```

```

// Otherwise, the subsequent row exists for adjacency.
else
    endRow = i + 1;

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

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

// Search for adjacency for X's and O's or vice versa,
// and replace them.
for (int x = startRow; x <= endRow; x++) {
    adj = this.setAdjacency(x, j, adj, isEnemy, map);
}

for (int y = startColumn; y <= endColumn; y++) {
    adj = this.setAdjacency(i, y, adj, isEnemy, map);
}

if (MaxChangeable < adj) {
    MaxChangeable = adj;
}
}

return MaxChangeable;
}

```

## Minimax Alpha-Beta Pruning

Pencarian dengan Minimax Alpha-Beta Pruning akan dilakukan menggunakan pendekatan secara rekursif dengan fungsi `minimaxAB`. Fungsi ini akan menerima current state map, kedalaman pencarian `depth`, dan initial value dari `alpha` (*negative infinity*) dan `beta` (*positive infinity*). Simpul anak dibangkitkan dengan melakukan iterasi pada setiap kotak kosong dari `currentState`. Karena maksimal jumlah anak yang mungkin dibangkitkan pada permainan Adjacency Strategy Game dengan ukuran papan 8x8 adalah 56, maka jumlah simpul dapat bertambah dengan cepat secara faktorial. Jadi, diperlukan adanya batasan jumlah keturunan yang dibangkitkan hanya sebanyak `MaxDepth` kedalaman saja. Kedalaman berjumlah `MaxDepth` dipilih karena terdapat kemungkinan rugi pada simpul anak tetapi simpul keturunannya lebih menguntungkan. Selain itu, dibutuhkan pula *alpha-beta Pruning* yang memangkas anak atau keturunan yang diyakini tidak akan memberikan solusi terbaik. Berikut merupakan langkah-langkah pencarian menggunakan Minimax Alpha-Beta Pruning:

- Memeriksa apakah state papan permainan saat ini merupakan basis. Basis fungsi Minimax adalah ketika terminal state atau kedalaman maksimum tercapai. Terminal state ditandai ketika `roundsLeft = 1`. Jika state saat ini merupakan basis, maka evaluasi `state` akan dikembalikan sesuai fungsi objektif
- Jika state saat ini bukan basis, fungsi akan melakukan rekursi yang dibedakan berdasarkan kedalaman pembangkitan. Kedalaman pencarian `depth` selalu dimulai dari 0 sehingga jika kedalaman bernilai genap berarti kondisi pencarian dalam *maximum condition* sedangkan jika kedalaman pencarian bernilai ganjil maka kondisi pencarian sedang dalam *minimum condition*. Pencarian simpul anak akan memanggil kembali fungsi `minimax` dengan menambah parameter kedalaman.
- Apabila *maximum condition*, maka nilai `alpha` akan di-update jika *heuristic value* state yang sedang dibangkitkan lebih tinggi dari nilai `alpha`. Dan
- Sedangkan apabila *minimum condition*, maka nilai `beta` yang akan di-update jika `state` yang sedang dibangkitkan lebih rendah dari nilai `beta`.
- *Pruning* atau pemangkasan dilakukan setelah nilai `alpha` atau `beta` diperiksa. *Pruning* atau pemangkasan akan dilakukan jika terjadi kondisi dimana nilai `beta` lebih kecil atau sama dengan nilai `alpha`.

```

/**
 *
 * @param depth
 * @param map
 * @param alpha
 * @param beta
 * @return heuristic value
 */
private double miniMaxAB(int depth, Button[][] map, double alpha, double
beta){
    this.MaxDepth = Math.min(defaultMaxDepth, this.roundsLeft);

    // Leaf node
    if(depth == this.MaxDepth){
        try {
            //
            if (this.roundsLeft == depth) {
                return ObjectiveFunction(0, map);
            } else if (depth%2 == 1){
                // Minimum]
                return ObjectiveFunction(-1, map);
            }
            // Maximum
            return ObjectiveFunction(1, map);
        } catch (Exception e){
            System.out.println(e.getMessage());
        }
    }

    // Odd depth (Minimum condition) (player turn)
    if(depth % 2 == 1){
        // Generating its children
        outerloop:
        for (int i=0; i<map.length; i++){
            for (int j=0; j<map[i].length; j++){
                if (map[i][j].getText().isEmpty()) {
                    // update map
                    Button[][] childMap = getUpdatedState(map, i, j, true);
                    double eval = miniMaxAB(depth + 1, childMap,
                                           alpha, beta);
                    beta = Math.min(beta, eval);
                }
            }
        }
    }
}

```

```

        // pruning
        if (beta <= alpha) {
            break outerloop;
        }
    }
}

return beta;

// Even depth (Maximum condition) (bot turn)
} else{
    // Generating its children
    outerloop:
    for (int i=0; i<map.length; i++){
        for (int j=0; j<map[i].length; j++){
            if (map[i][j].getText().isEmpty()) {
                // update map
                Button[][] childMap = getUpdatedState(map, i, j, false);
                double eval = miniMaxAB(depth + 1, childMap,
                                         alpha, beta);
                if (eval > alpha) {
                    alpha = eval;
                    // assign selected coordinate
                    if (depth == 0) {
                        this.selected[0] = i;
                        this.selected[1] = j;
                    }
                }

                // pruning
                if (beta <= alpha) {
                    break outerloop;
                }
            }
        }
    }
}

return alpha;
}
}

```

## Local Search

Kami mengambil pencarian Local Search dengan pendekatan algoritma *hill climbing*.

Penerapan algoritma ini tidak menjamin langkah yang dipilih merupakan langkah yang terbaik yang menuju ke objektif. Hal itu disebabkan oleh algoritma *hill climbing* hanya mencari state tetangga yang dapat dituju dengan satu langkah. Oleh karena itu, algoritma ini memilih langkah selanjutnya yang merupakan langkah terbaik pada state tersebut. Langkah-langkah penerapan algoritma *hill climbing* pada permainan Adjacency Strategy Game ini sebagai berikut:

- Pertama, mengambil kondisi atau state permainan saat ini dan menggunakananya sebagai node root dalam pohon pencarian. Pohon pencarian ini akan mencerminkan semua kemungkinan langkah yang dapat diambil dalam permainan.
- Kedua, menghasilkan tetangga atau anak-anak dari node tersebut. Oleh karena itu, algoritma mencari semua kemungkinan penempatan bidak atau langkah-langkah strategis yang dapat diambil dari situ. Dengan demikian, hal tersebut menciptakan cabang-cabang baru dalam pohon pencarian.
- Ketiga, setiap node tetangga atau cabang yang dihasilkan akan dinilai atau dinilai menggunakan sebuah fungsi objective yang telah Anda tentukan. Fungsi ini berguna untuk memberikan skor numerik pada setiap langkah tersebut.
- Keempat, setelah semua node tetangga telah dinilai, akan dipilih node dengan skor tertinggi yang merupakan successor dari initial state dan memiliki skor lebih besar dari initial state sebagai langkah berikutnya.
- Kelima, Jika tidak ada skor yang lebih tinggi dari initial state maka pencarian selesai.

Algoritma ini sering gagal dalam mencapai *global maximum* tetapi tidak menutup kemungkinan dapat mencapai hal tersebut.

```
private int[] hillClimb() {  
    // Initialize variables to keep track of the highest score  
    // and corresponding move.  
    int maxScore = Integer.MIN_VALUE;  
    int bestI=-1, bestJ=-1;
```

```

Button[][] state = this.currentState;

// Loop through the current game state to evaluate all potential moves.
for (int i = 0; i < ROW; i++) {
    for (int j = 0; j < COL; j++) {
        // Check if the current cell is empty,
        // indicating a potential move.
        if (state[i][j].getText().equals("")) {
            // Generate a new game state after
            // making a move at the current cell.
            Button[][] child = getUpdatedState(state, i, j, false);

            // Evaluate the new game state using the ObjectiveFunction.
            int score = ObjectiveFunction(
                this.roundsLeft == 1 ? 0 : 1, child);

            // If the score is higher than the current maxScore,
            // update maxScore and move coordinates.
            if (score > maxScore) {
                maxScore = score;
                bestI = i;
                bestJ = j;
            }
        }
    }
}

// Return the best move's coordinates.
return new int[]{bestI, bestJ};
}

```

# Genetic Algorithm

Genetic Algorithm dapat diimplementasikan dalam permainan dengan cara sebagai berikut:

1. Inisialisasi Populasi Awal:

- Populasi awal dibentuk dengan membuat sejumlah individu, di mana setiap individu berisi serangkaian koordinat  $(x, y)$  yang merepresentasikan langkah-langkah dalam permainan.
- Setiap individu harus dihasilkan secara acak, tetapi harus mematuhi aturan bahwa langkah-langkah hanya boleh diambil pada kotak yang masih kosong di papan permainan.

2. Evaluasi Fitness:

- Setiap individu dalam populasi dinilai berdasarkan *fitness value*. Evaluasi *fitness* dilakukan dengan menggunakan Objective function yang telah dijelaskan pada bagian sebelumnya.

3. Seleksi Parent:

- Individu yang akan menjadi orangtua untuk generasi berikutnya dipilih melalui metode turnamen. Di sini, beberapa individu secara acak dipilih untuk bersaing, dan yang memiliki kinerja terbaik akan menjadi orangtua.

4. Operasi Crossover:

- Proses crossover (rekombinasi) dilakukan dengan menggabungkan informasi dari dua orangtua untuk menghasilkan keturunan. Ini dilakukan dengan menukar bagian dari koordinat antara dua orangtua.

5. Operasi Mutasi:

- Terjadi proses mutasi, di mana beberapa individu dalam populasi mengalami perubahan acak dalam koordinat langkah-langkah mereka. Ini membantu mendorong variasi dalam populasi.

6. Generasi Populasi Baru:

- Populasi generasi baru dibentuk berdasarkan individu yang dihasilkan melalui crossover dan mutasi.

7. Kriteria Berhenti:

- Algoritma berjalan hingga kriteria berhenti tercapai, seperti mencapai solusi optimal atau mencapai batas generasi yang telah ditentukan.

```
/**
 *
 * perform return next move with genetic algorithm
 *
 * @param map ,map on that round
 * @return
 */
public int[] geneticMove(Button[][] map) {
    int populationSize = 28;
    int maxGenerations = 50;
    int tournamentSize = 10;

    // inisialisasi populasi awal
    List<List<int[]>> population = new ArrayList<>();
    while (population.size() < populationSize) {
        List<int[]> individual = new ArrayList<>();
        while (individual.isEmpty()) {
            int x = new Random().nextInt(8);
            int y = new Random().nextInt(8);
            if (map[x][y].getText().equals("")){
                int[] coordinates = {x, y};
                individual.add(coordinates);
            }
        }
        population.add(individual);
    }

    // algoritma genetika
    for (int generation = 0; generation < maxGenerations; generation++) {
        // evaluasi fitness setiap individu dalam populasi
        List<Integer> fitnessScores = new ArrayList<>();
        for (List<int[]> individual : population) {
            int fitness = evaluateFitness(individual, map);
            fitnessScores.add(fitness);
        }

        // cek kriteria berhenti
        if (Collections.max(fitnessScores) == 8 * 8
            && population.get(generation).size() == 1)
            break;
    }
}
```

```

        || generation == maxGenerations - 1) {
    List<int[]> bestIndividual = population.get(fitnessScores
                                                .indexOf(Collections.max(fitnessScores)));
    return bestIndividual.get(0);
}

// seleksi parent menggunakan metode turnamen
List<List<int[]>> parents = new ArrayList<>();
for (int i = 0; i < populationSize; i++) {
    List<Integer> tournament = getRandomSample(
                                populationSize, tournamentSize);
    List<Integer> tournamentFitness = new ArrayList<>();
    for (int j : tournament) {
        tournamentFitness.add(fitnessScores.get(j));
    }
    int winnerIndex = tournament.get(tournamentFitness.indexOf(
                                Collections.max(tournamentFitness)));
    parents.add(population.get(winnerIndex));
}

// operasi crossover
List<List<int[]>> offspring = new ArrayList<>();
for (int i = 0; i < populationSize - 1; i += 2) {
    int[] parent1 = parents.get(i).get(0);
    int[] parent2 = parents.get(i + 1).get(0);
    int[] child1 = crossover(parent1, parent2);
    int[] child2 = crossover(parent2, parent1);

    List<int[]> child1List = new ArrayList<>();
    List<int[]> child2List = new ArrayList<>();

    child1List.add(child1);
    child2List.add(child2);

    offspring.add(child1List);
    offspring.add(child2List);
}

// operasi mutasi
for (int i = 0; i < offspring.size(); i++) {

    int x = offspring.get(i).get(0)[0];
}

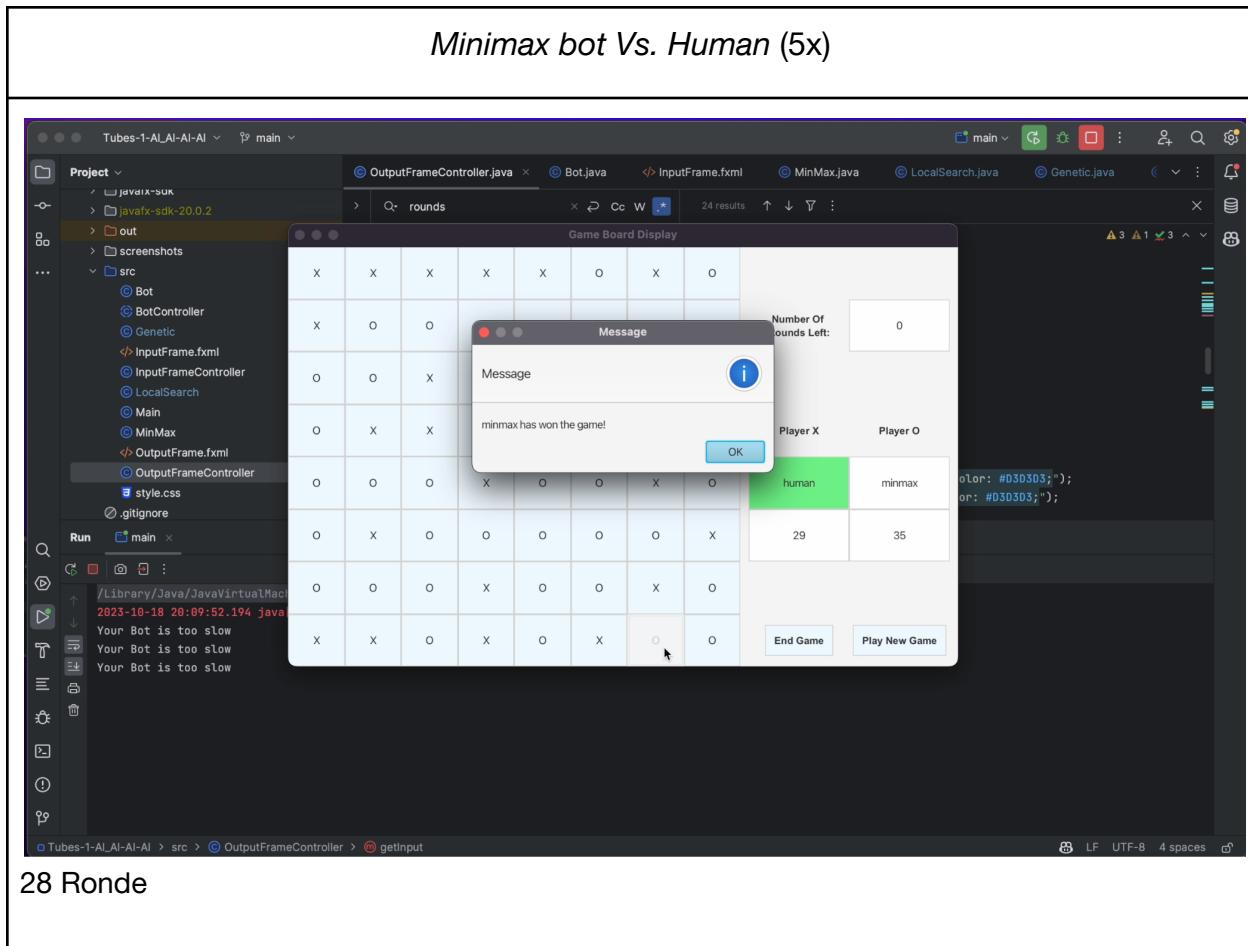
```

```
int y = offspring.get(i).get(0)[1];
while (!map[x][y].getText().equals("")){
    offspring.set(i, mutate(offspring.get(i)));
    x = offspring.get(i).get(0)[0];
    y = offspring.get(i).get(0)[1];
}

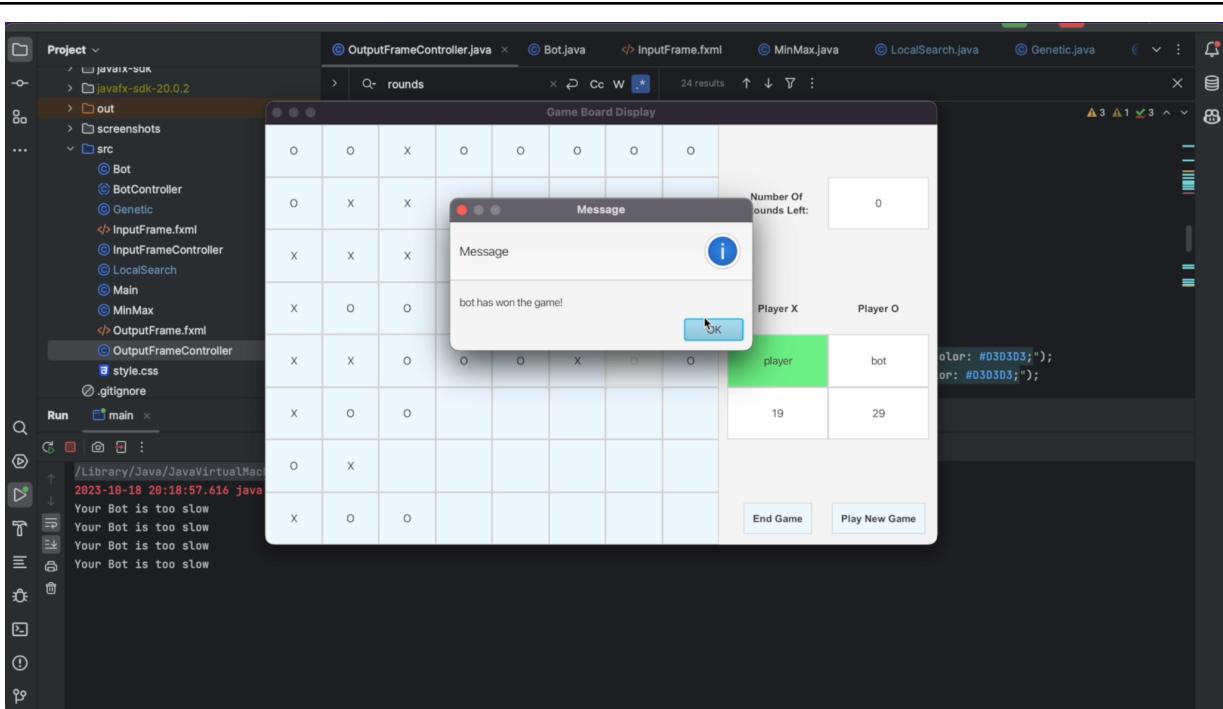
// generasi populasi baru
population = new ArrayList<>(offspring);
}

return null;
}
```

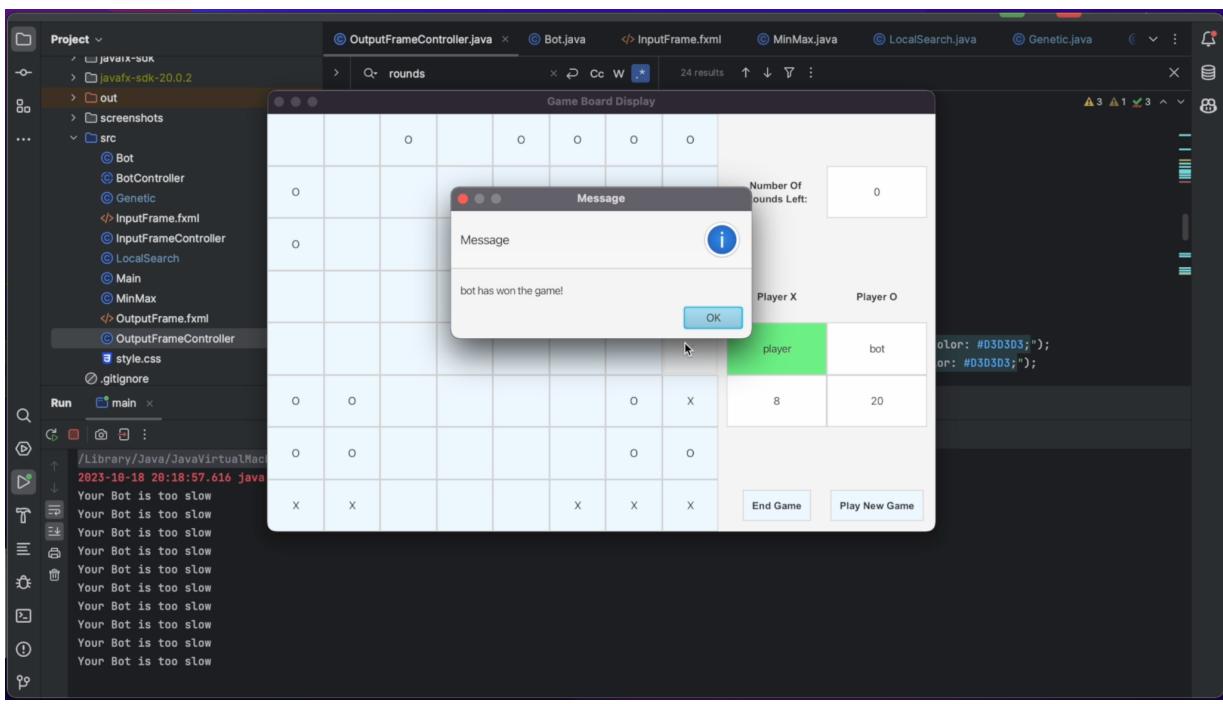
# Demo



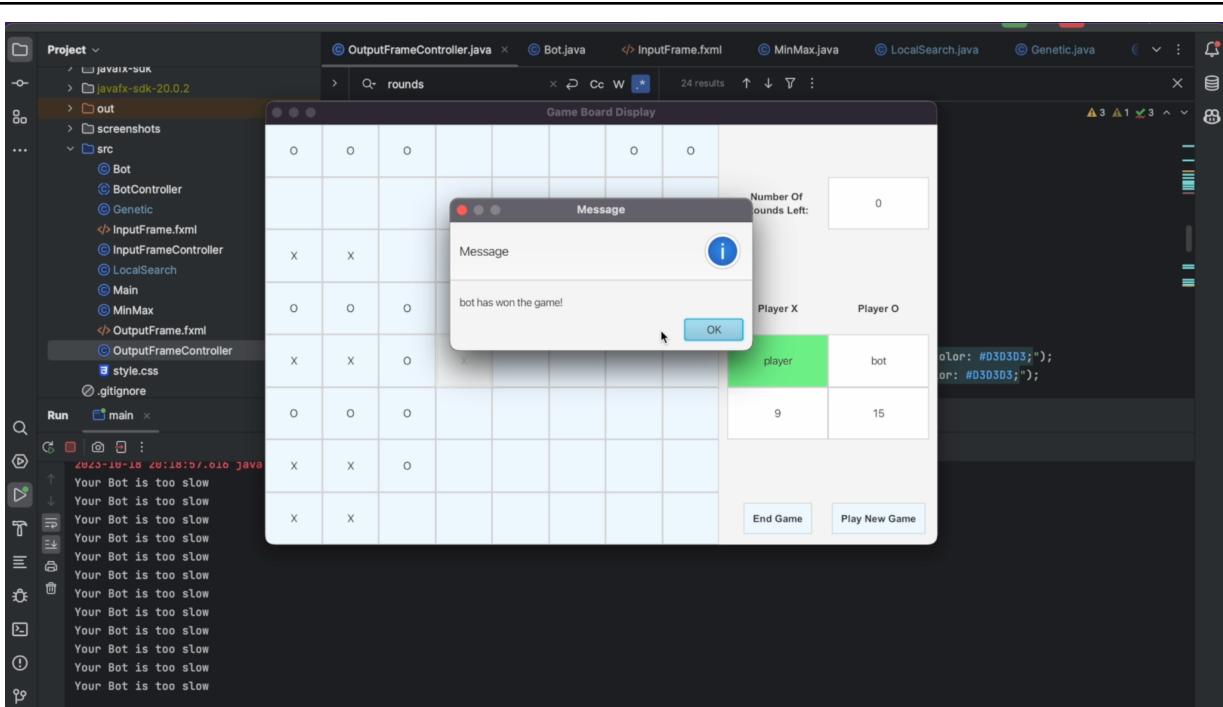
28 Ronde



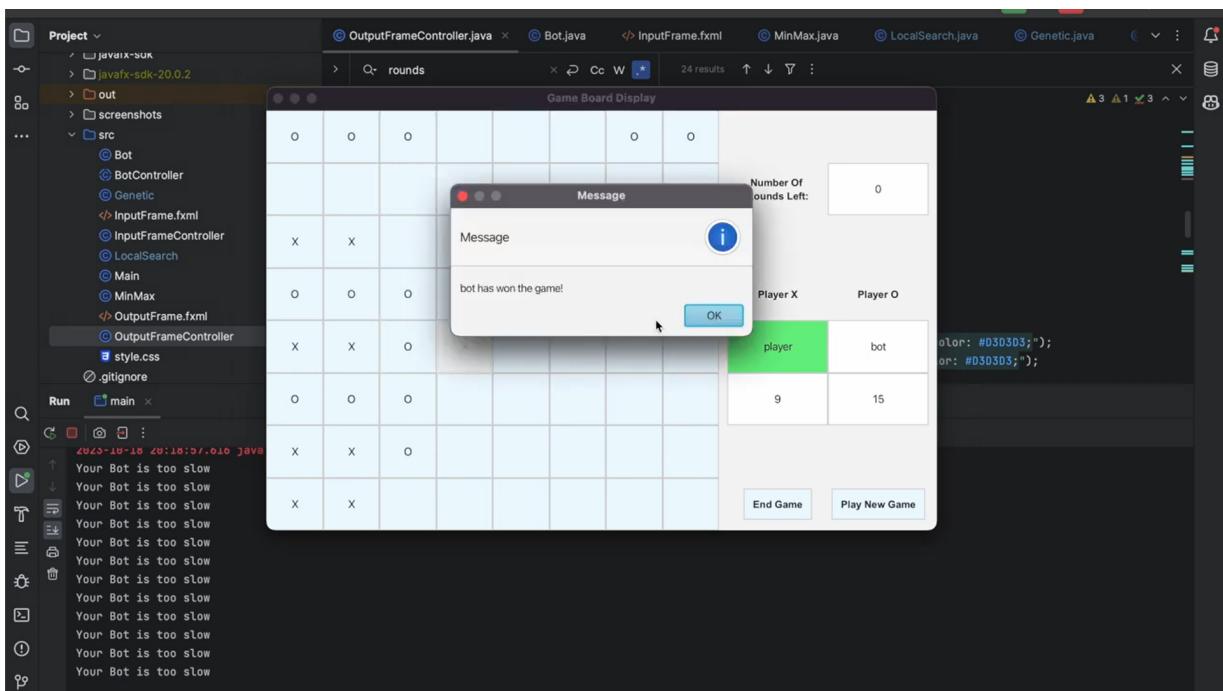
20 Ronde



14 Ronde



## 10 Ronde



## 8 Ronde

### Analisis

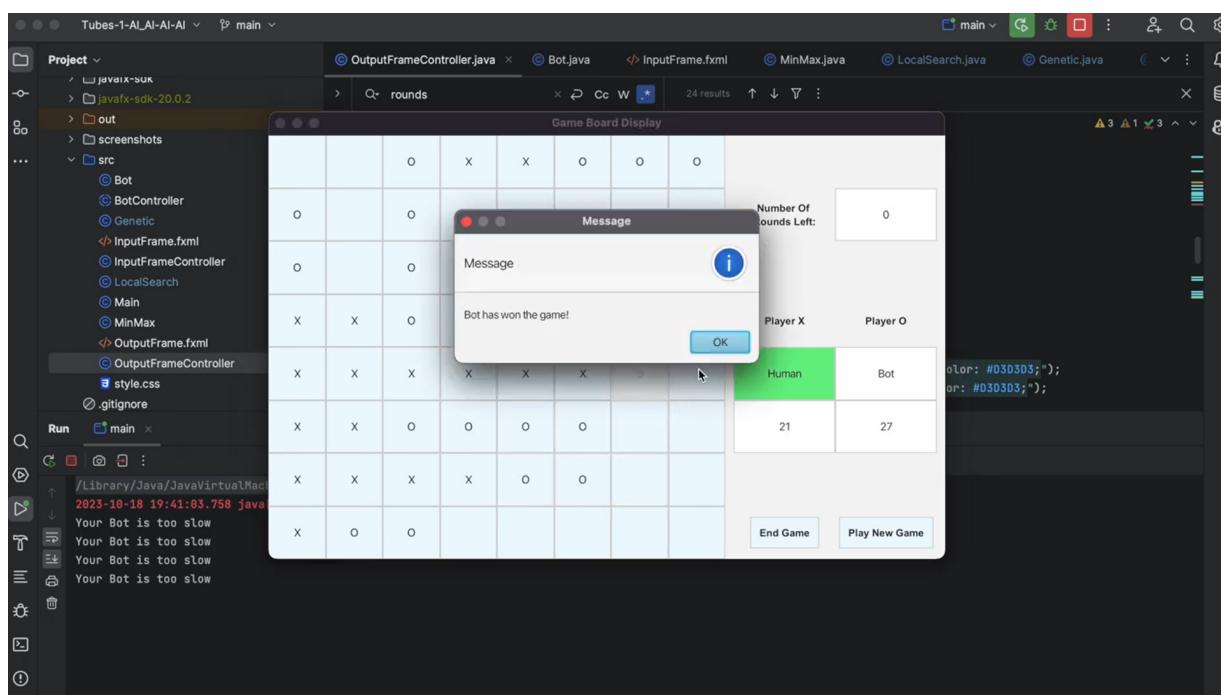
Pada awal permainan, Manusia diuntungkan atas *Minimax* bot dengan kedalaman pencarian maksimum (maxdepth) sebesar 5 yang terjadi karena adanya batasan waktu eksekusi

maksimal selama 5 detik. Pada tahap awal, jumlah langkah atau kemungkinan yang bisa diambil masih sangat banyak, sehingga bot membutuhkan waktu yang signifikan untuk mengevaluasi semua pilihan tersebut. Keterbatasan waktu 5 detik membuatnya sulit bagi bot untuk mengevaluasi semua potensi langkah pada level pertama sehingga solusi yang didapatkan bukan solusi optimum. Sehingga semakin banyak ronde maka minimax semakin diuntungkan. Namun, kekurangan ini bisa diatasi dengan mengurangi kedalaman pencarian *minimax* bot. Selain itu, *Minimax* bot terbukti dapat mengalahkan player dengan tingkat kemenangan 100%.

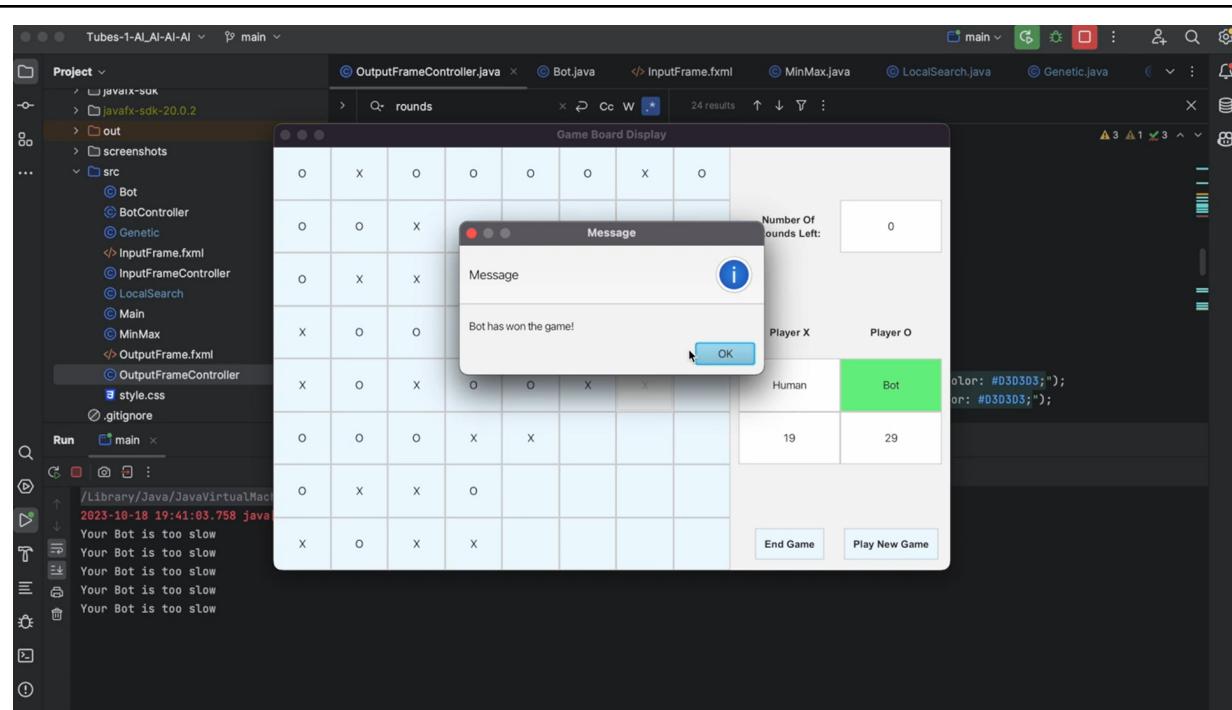
Pranala video : [Link](#)

<i>Minimax</i> bot	Win : 5	Lose : 0
Human	Win : 0	Lose : 5

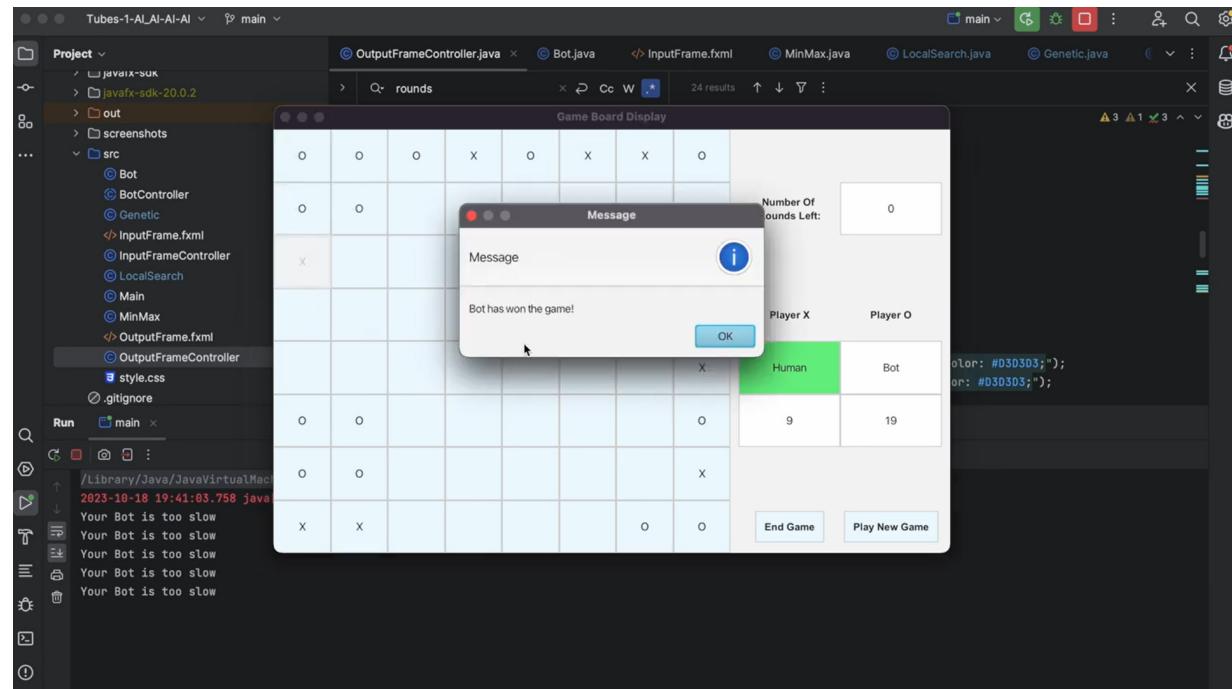
### Local-search bot Vs. Human (3x)



20 Ronde



## 14 Ronde



## 10 Ronde

### Analisis

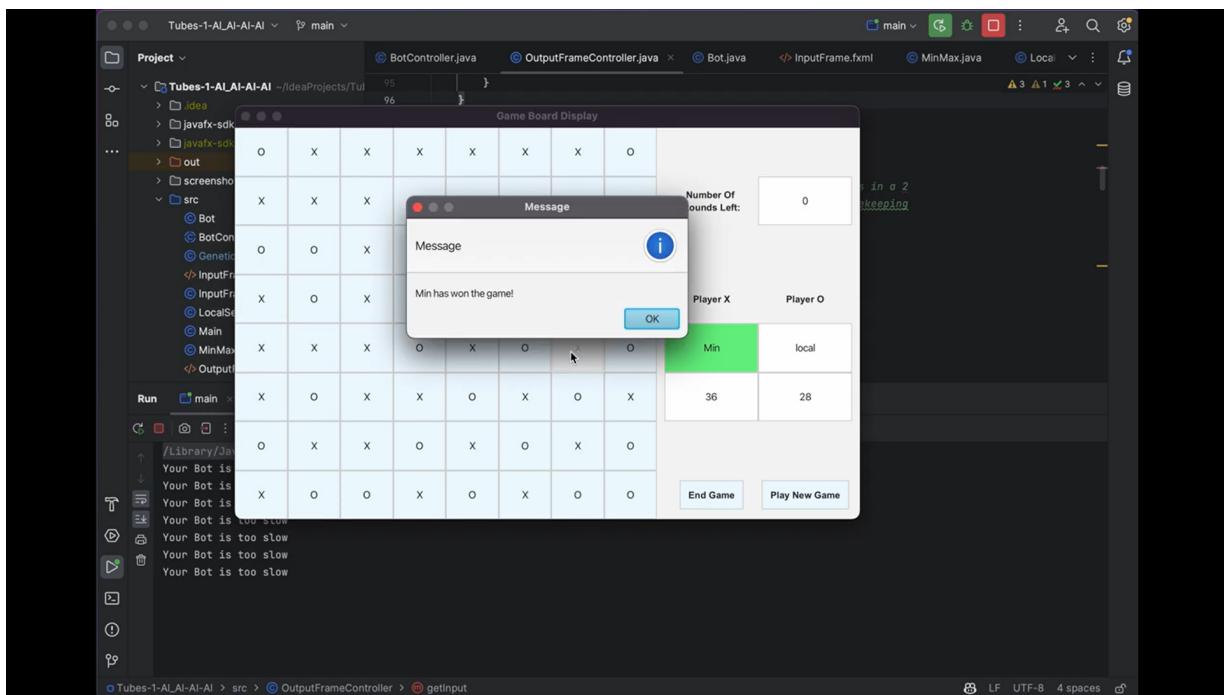
*Local-search bot* selalu mengambil langkah terbaik untuk setiap state permainan. Sehingga, walaupun terdapat kemungkinan bot untuk stuck atau berhenti di *local optimum*,

*Local-search bot* berhasil mengalahkan player dengan tingkat kemenangan 100% dikarenakan *human* yang tidak memiliki kemampuan yang cukup untuk mencari solusi optimum untuk setiap state.

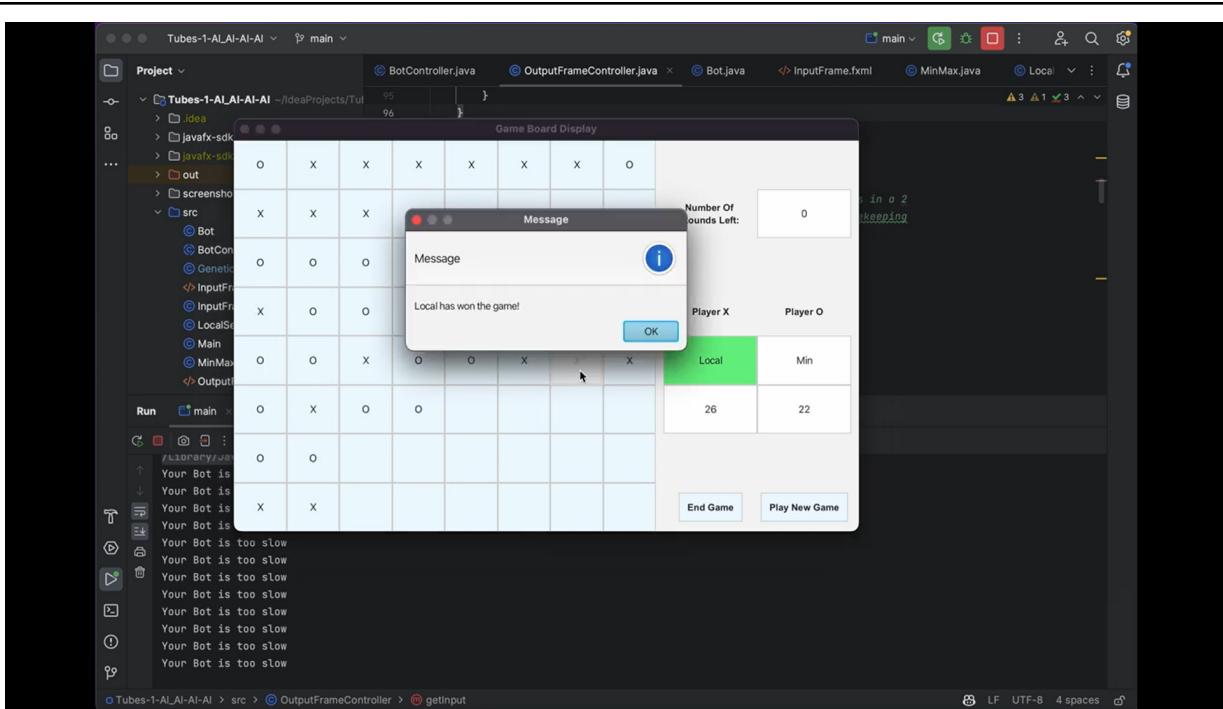
Link video : [Link](#)

<i>Local-search bot</i>	<i>Win : 3</i>	<i>Lose : 0</i>
<i>Human</i>	<i>Win : 0</i>	<i>Lose : 3</i>

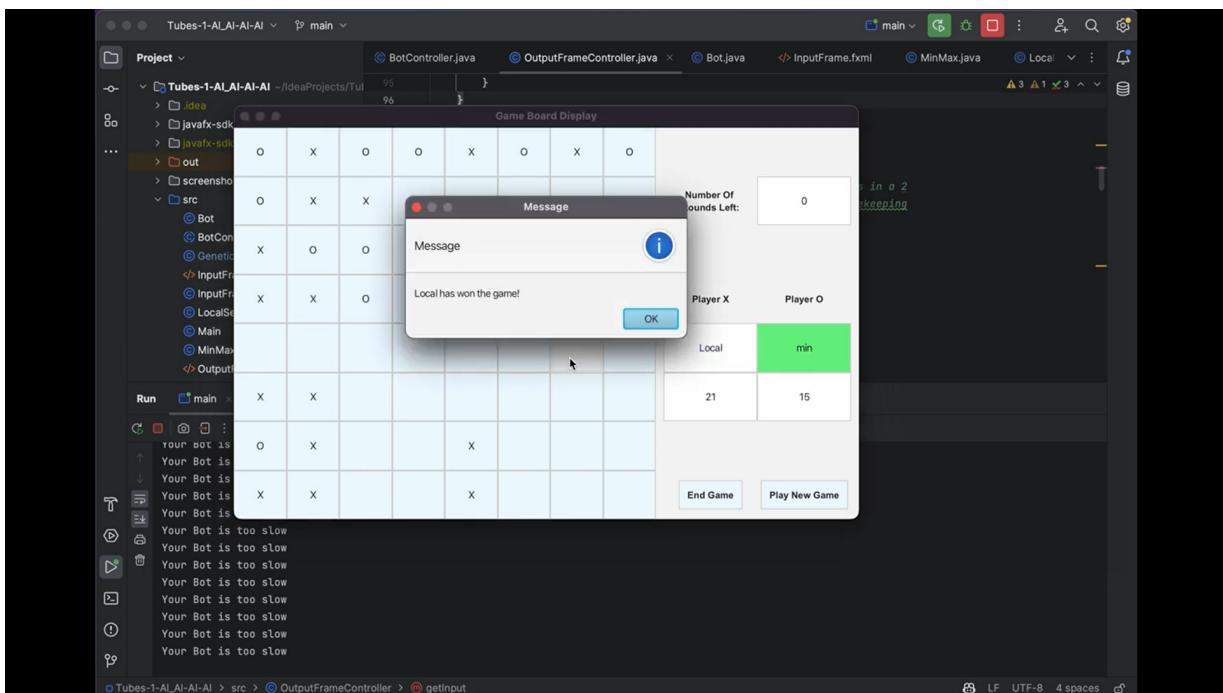
### *Minimax bot Vs. Local-search bot (3x)*



28 Ronde



20 Ronde



14 Ronde

### Analisis

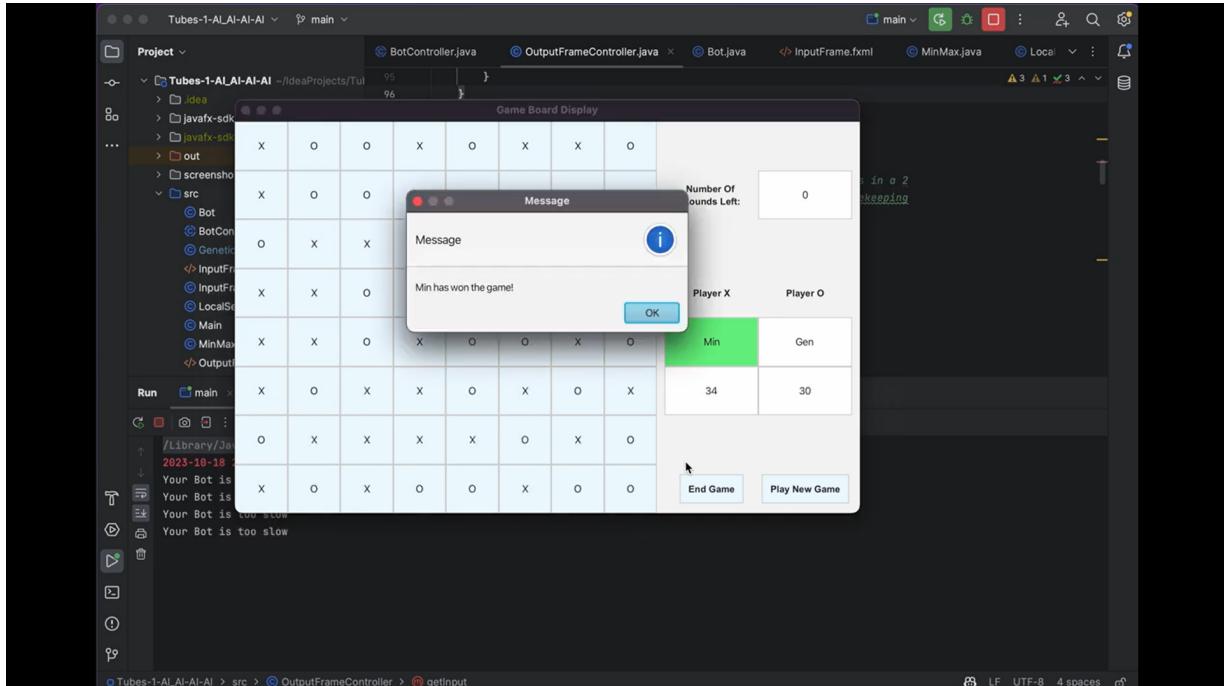
Minimax bot berhasil mengalahkan Local-search bot pada ronde permainan tinggi, akan tetapi Local-search bot unggul dari Minimax bot pada ronde permainan awal. Hal ini

disebabkan oleh banyaknya keturunan yang perlu digenerasi dan dievaluasi oleh *Minimax-bot* pada ronde awal sehingga bot mungkin melebihi batas waktu 5 detik dan melakukan *fallback* sehingga mengurangi tingkat efektivitas *Minimax-bot* pada tahap ini. Tetapi, ketika memasuki tahap pertengahan hingga akhir permainan, *Minimax-bot* tidak perlu lagi membangkitkan terlalu banyak keturunan dan dapat mencari solusi paling optimal dengan lebih efisien sehingga mengalahkan *Local-search bot*.

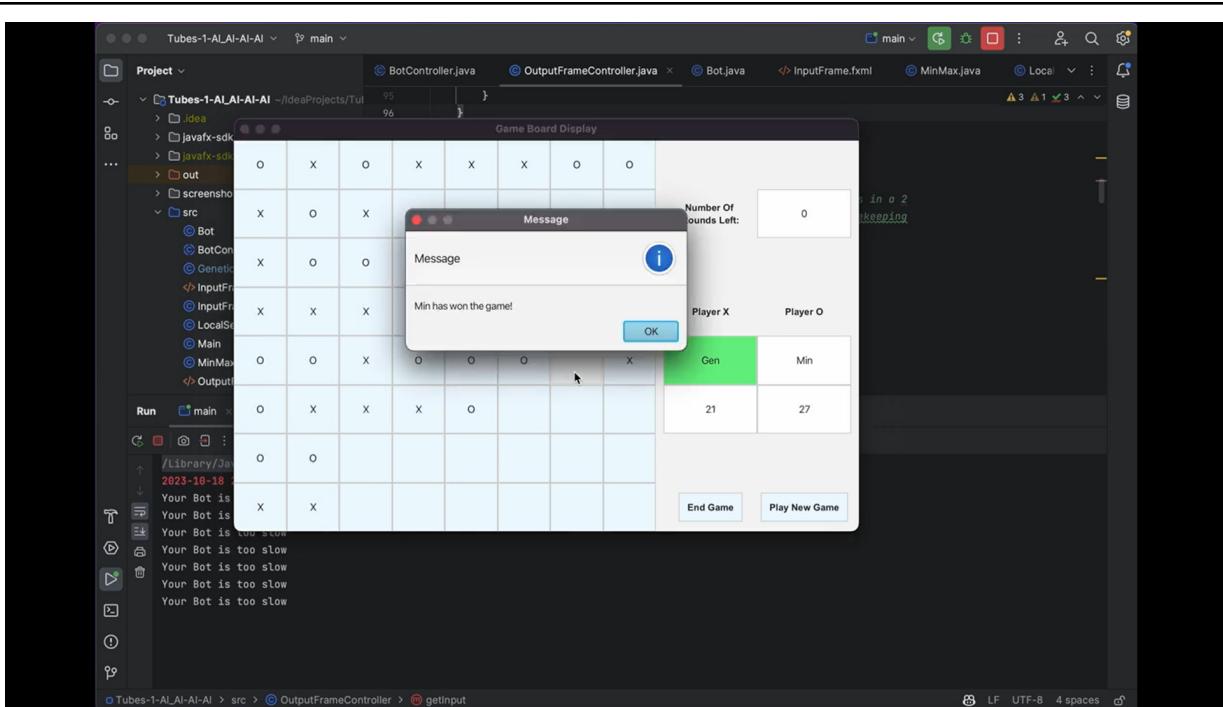
Link video : [Link](#)

<i>Minimax bot</i>	<i>Win : 1</i>	<i>Lose : 2</i>
<i>Local-search bot</i>	<i>Win : 2</i>	<i>Lose : 1</i>

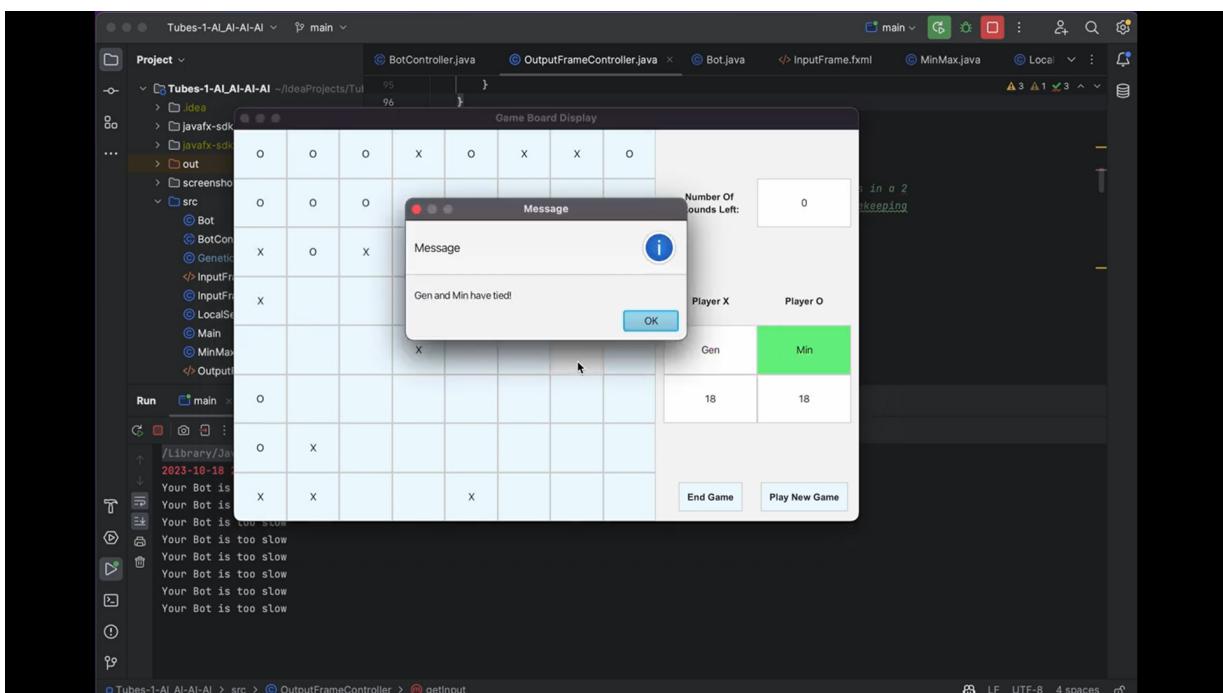
### *Minimax bot Vs. GA bot (3x)*



28 Ronde



20 Ronde



14 Ronde

### Analisis

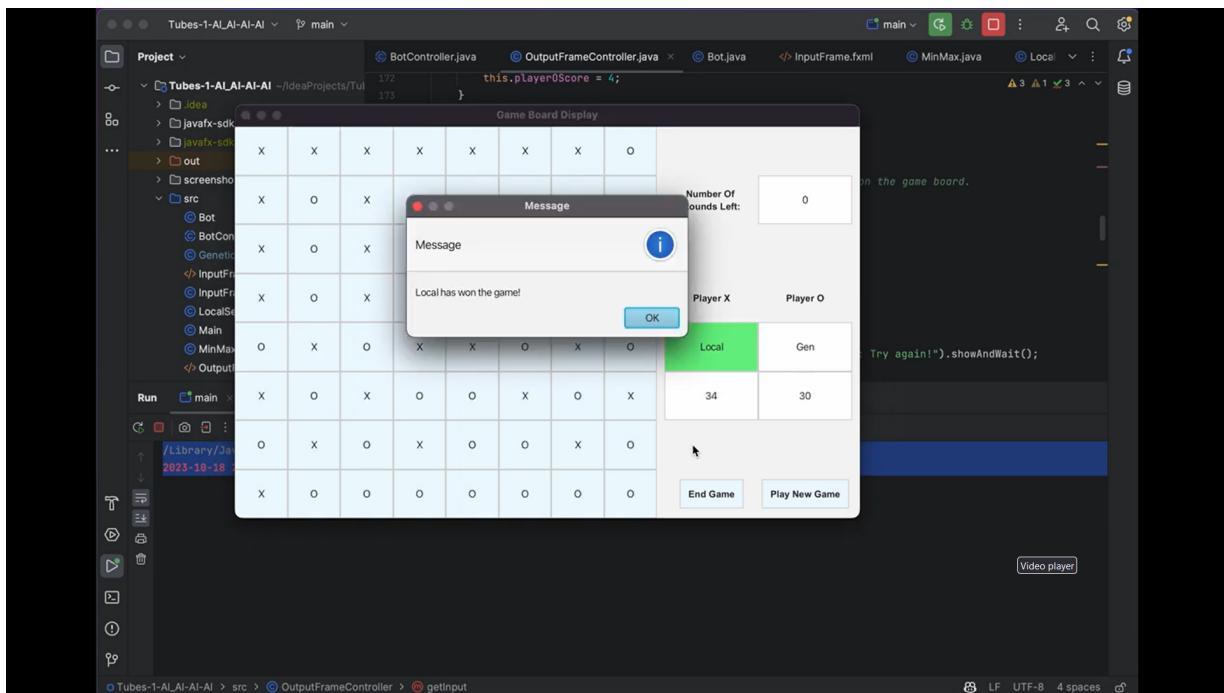
Sama seperti sebelumnya, *Minimax bot* berhasil mengalahkan *GA bot* pada ronde permainan tinggi, akan tetapi *GA bot* unggul dari *Minimax bot* pada ronde permainan awal. Hal ini juga

disebabkan oleh kurangnya tingkat efisien *Minimax-bot* pada awal permainan sehingga *fallback* mungkin terjadi.

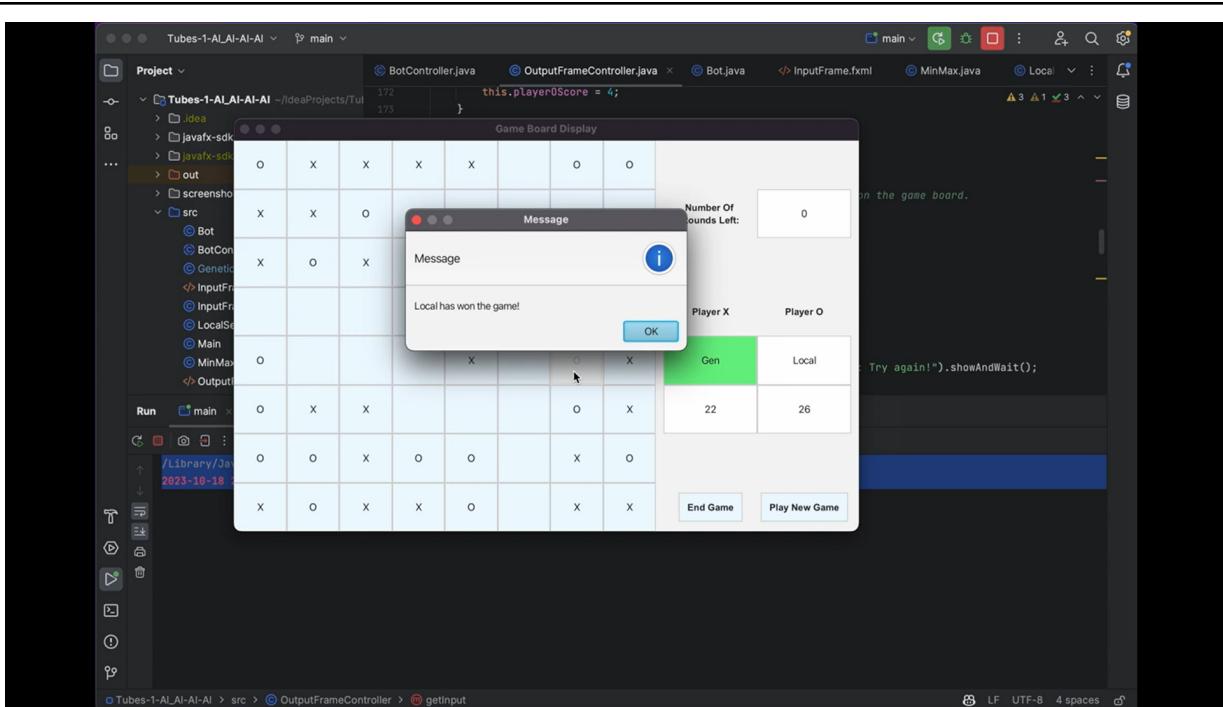
Link video : [Link](#)

<i>Minimax bot</i>	<i>Win : 2</i>	<i>Lose : 1</i>
<i>GA bot</i>	<i>Win : 1</i>	<i>Lose : 2</i>

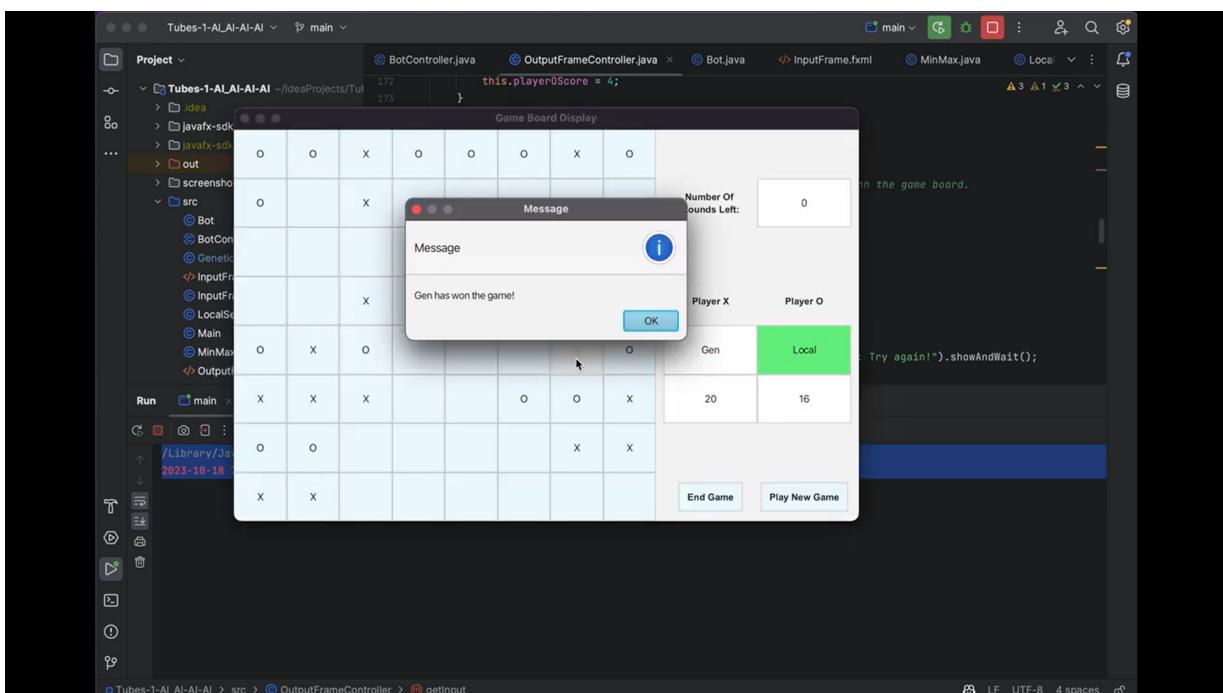
### *Local-search bot Vs. GA bot (3x)*



28 Ronde



20 Ronde



14 Ronde

### Analisis

Walaupun Local-search bot unggul dari GA bot dengan 2 kemenangan dibanding 1, tetapi poin kedua bot tidak terpaut jauh. Hal ini disebabkan karena sifat kedua bot pencarian yang

mirip yaitu mencari solusi terbaik untuk setiap state permainan.

Link video : [Link](#)

<i>Local-search bot</i>	<i>Win : 2</i>	<i>Lose : 1</i>
<i>GA bot</i>	<i>Win : 1</i>	<i>Lose : 2</i>

	<i>Human (skill issue)</i>	<i>Minimax Bot</i>	<i>Local-search Bot</i>	<i>GA Bot</i>
<i>winrate</i>	0%	80%	77%	33%

## Kontribusi

NIM	Nama	Tugas
13521080	Fajar Maulana H	BotController.java, InputFrameController.java, OutputFrameController.java, Genetic.java
13521081	Bagas Aryo Seto	LocalSearch.java, Genetic.java
13521097	Shidqi Indy Izhari	Genetic.java, Minmax.java
13521119	Muhammad Rizky Sya'ban	Minmax.java, Bot.java

## Lampiran

[Link repository](#)

## Referensi

Russel, Stuart J. dan Norvig, Peter. (2016). Artificial Intelligence: A Modern Approach 3th Edition. Edinburgh: Pearson Education Ltd.

<https://blog.paperspace.com/tic-tac-toe-genetic-algorithm-part-1/>