

Laporan Tugas Besar I IF3170
Minimax Algorithm and Alpha Beta Pruning in Adjacency Strategy
Game



Disusun oleh:

Kelompok 19

Muhammad Hanan	13521041
Alex Sander	13521061
Ahmad Ghulam Ilham	13521118
Muhammad Abdul A. G.	13521128

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung
Jl. Ganesha 10, Bandung 40132
2023

1. Objective Function

a. Pada Minimax

Secara umum, fungsi objektif yang kami pilih adalah selisih skor antara bot dengan lawan



```
private int objectiveFunction(Button[][] buttons){
    int score = 0;
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (buttons[i][j].getText().equalsIgnoreCase("O")){
                score++;
            }
            else if (buttons[i][j].getText().equalsIgnoreCase("X")){
                score--;
            }
        }
    }
    return score;
}
```

Gambar 1.1 Cara Kerja Objective Function Secara Umum

Fungsi tersebut menerima input berupa matriks Button, sebuah kelas yang merupakan representasi sebuah kotak yang dapat diisi X atau O dalam permainan. Kemudian matriks tersebut diiterasiakan tiap elemennya untuk mengecek isi Button dan menambah skor +1 jika Button berisi O dan mengurangi skor -1 jika Button berisi X. Namun, ini juga tergantung dari konfigurasi bot yang menggunakan bidak X atau O. Inti dari fungsi objektif ini adalah menggunakan variabel skor untuk menghitung **banyaknya skor bot dikurangi skor lawan**, yang semakin positif nilai ini maka skor bot akan semakin tinggi daripada lawan, menandakan bot akan memenangi permainan.

Namun, dalam bot *Minmax*, fungsi ini diimplementasikan dengan cara yang berbeda yang lebih efisien dan agar bot tidak mengiterasi *button* setiap mengevaluasi state suatu langkah. Dalam method *move*, terdapat parameter skor kedua *player* yang digunakan untuk patokan perubahan skor di langkah-langkah yang dikalkulasi *bot*. Informasi *skor* ini masuk juga ke fungsi *min* dan *max*, dimana selisihnya digunakan untuk menghitung *minVal* / *maxVal*

```

int curPlayer0Score = player0Score;
int curPlayerXScore = playerXScore;

if (curDepth == maxDepth || curDepth == maxDepthCheck) {
    maxValues[0] = curPlayer0Score - curPlayerXScore;
    maxValues[1] = maxRow;
    maxValues[2] = maxCol;
    //=====

```

Gambar 1.2 Menghitung selisih skor di *method* Max di Minimaxbot

```

int curPlayer0Score = player0Score;
int curPlayerXScore = playerXScore;

if (curDepth == maxDepth || curDepth == maxDepthCheck) {
    minValues[0] = curPlayer0Score - curPlayerXScore;
    minValues[1] = minRow;
    minValues[2] = minCol;
    //=====

```

Gambar 1.3 Menghitung selisih skor di *method* Min di Minimaxbot

Perhitungan skor lanjut dari state *buttons* yang tadi, adalah dengan menghitung perubahan bidak yang terjadi setelah adanya peletakan suatu bidak, dihitung dengan perhitungan *updateGameBoard* yang mengembalikan nilai selisih yang didapat setelah merubah bidak lawan yang *adjacent* dengannya. Kemudian selisih tersebut dalam variabel *difference* digunakan untuk mengubah skor kedua pemain.

```

int difference = updateGameBoard(!isMaxingX, i, j, curButtons);
curPlayer0Score += (difference + 1);
curPlayerXScore -= difference;

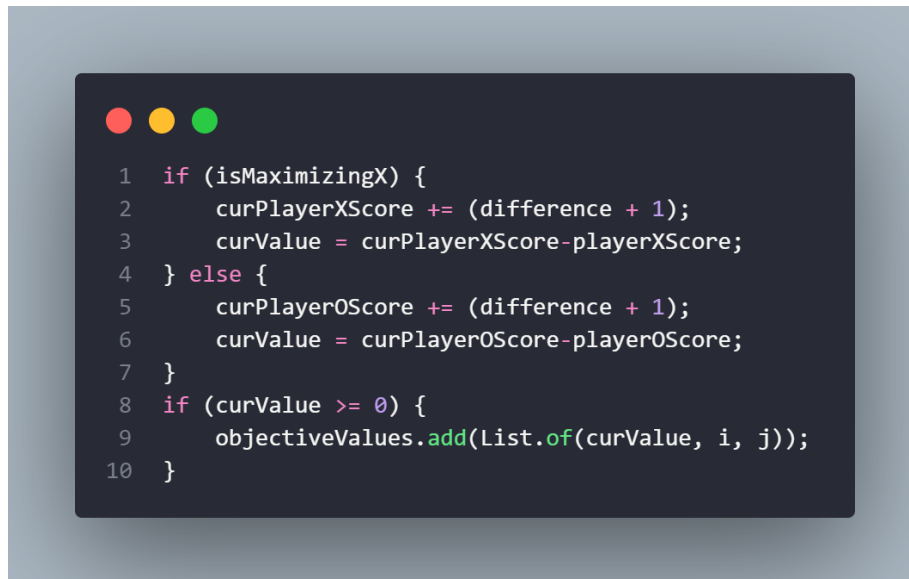
```

Gambar 1.4 Pemanggilan fungsi *updateGameBoard* dan Penggunaannya

Alasan pemilihannya dari pemilihan fungsi objektif ini yang sangat simpel adalah tujuan dari permainan ini hanyalah mendapatkan nilai skor yang lebih tinggi dari lawan. Kita tidak perlu memikirkan apakah pemilihan letak bidak strategis kedepannya (tidak harus memiliki skor yang lebih tinggi dari lawan) karena langkah pemikiran kedepan secara strategis tentu sudah dibayangkan/proyeksikan oleh fungsi minmax yang sudah berlangkah-langkah ke depan sehingga fungsi objektif ini tidak harus rumit.

b. Pada Local Search

Secara umum, fungsi objektif pada Local Search yang kami pilih adalah seperti berikut



```
1  if (isMaximizingX) {
2      curPlayerXScore += (difference + 1);
3      curValue = curPlayerXScore - playerXScore;
4  } else {
5      curPlayerOScore += (difference + 1);
6      curValue = curPlayerOScore - playerOScore;
7  }
8  if (curValue >= 0) {
9      objectiveValues.add(List.of(curValue, i, j));
10 }
```

Gambar 1.5. Objective Function untuk Local Search

Kode tersebut bertujuan untuk mendapatkan selisih skor maksimal antara skor sebelum bot jalan untuk menaruh simbol pada papan dengan skor setelah bot jalan. Karena menggunakan *Hill-Climbing with sideways move*, objective value hanya akan dipilih jika **selisih lebih besar atau sama** dengan skor sebelum bot jalan.

c. Pada Genetic

Secara umum, fungsi objektif pada Genetic Search yang kami pilih adalah seperti berikut

```

int playerXScore = 0;
int playerOScore = 0;
for (int i= 0; i < ROW; i++) {
    for (int j = 0; j < COL; j++) {
        if (curButtons[i][j].getText().equals("X")) {
            playerXScore++;
        } else if (curButtons[i][j].getText().equals("O")) {
            playerOScore++;
        }
    }
}
if (localIsMaximizingX) {
    return playerXScore-playerOScore;
}

return playerOScore-playerXScore;

```

Gambar 1.6. Objective Function untuk Genetic Search

Kode tersebut bertujuan untuk mendapatkan selisih antara skor Player X dengan Player O. Untuk mendapatkan skor tersebut, setiap sekuens aksi pada kromosom akan dilakukan pada gameBoard dan objective function hanya digunakan di pada aksi terakhir kromosom, sehingga didapat objective value untuk masing-masing kromosom.

2. Pencarian Solusi dengan Minimax Alpha Beta Pruning Algorithm

Dalam algoritma *minimax* alpha-beta pruning, berlaku hal-hal berikut :

- Bot adalah pemain yang ingin mendapat objective function maximum (*maximizing player*), sedangkan “Player” adalah pemain yang ingin mendapat objective function minimum (*minimizing player*). Dimana informasi didapat dari mem-*passing* informasi variabel *isMaximizing*.
- Algoritma minimax alpha beta membatasi pencarian solusi sedalam 6 level (termasuk level 0). Dengan kata lain, pencarian dilakukan hingga langkah ke-3 dari bot. Hal tersebut dilakukan untuk meningkatkan kecepatan dari algoritma minimax alpha beta pruning yang diimplementasikan agar selalu dapat berjalan di bawah 5 detik. Berdasarkan beberapa uji coba, ketika ronde lebih besar daripada 3, algoritma memerlukan waktu yang sangat lama untuk menemukan solusi sehingga sangat sulit untuk menyelesaikan permainan dalam waktu yang wajar
- Pembatasan level pencarian solusi memiliki kelebihan dan kekurangan tersendiri. Dari sisi kelebihan, algoritma memiliki *feasibility* yang jauh lebih baik, terutama untuk permainan dengan jumlah ronde di atas 3. Meskipun begitu, kekurangan dari pembatasan level pencarian solusi adalah algoritma mungkin mendapat solusi langkah yang kurang optimal (bukan global maksima) pada permainan dengan jumlah ronde lebih dari 3 karena *game tree* yang tidak ditelusuri secara lengkap dan mendalam sehingga menaikan kemungkinan untuk memilih state yang mungkin menguntungkan di awal tetapi merugikan di langkah-langkah berikutnya..

Berikut adalah langkah-langkah pencarian solusi menggunakan algoritma minimax alpha beta pruning yang diimplementasikan:

1. Dari method move, tergantung dari bidak yang dipilih oleh *bot*, Bot akan memanggil *min* atau *max*

```
public int[] move(int roundsLeft, boolean isBotFirst, int player0Score, int playerXScore, Button[][] buttons, boolean isMaximizingX) {
    startTime = Instant.now();
    int maxDepth = roundsLeft * 2 - 1;

    int curDepth = 0;
    int[] maxValues;
    if (isMaximizingX == true) {
        maxValues = min(player0Score, playerXScore, curDepth, maxDepth, buttons, isBotFirst, isMaximizingX);
    } else {
        maxValues = max(player0Score, playerXScore, curDepth, maxDepth, buttons, isBotFirst, isMaximizingX);
    }
}
```

- a. `maxDepth` dihitung sebesar langkah yang bisa dilakukan oleh kedua player dari ronde yang tersisa.
 - b. `curDepth` adalah variabel untuk menginisialisasi kedalaman awal dalam proses menelusuri *tree* yang akan dibentuk.
 - c. Pada penjelasan langkah selanjutnya, diasumsikan *bot* memilih bidak O untuk bermain sehingga memanggil fungsi *max* yang pertama kali.
2. **Max** menginisialisasi nilai `maxVal` untuk mencari nilai children dari node state itu dan informasi kolom dan baris dalam button yang memaksimalkan value (dari fungsi objektif)

```
public int[] max(int playerOScore, int playerXScore, int curDepth, int maxDepth, Button[][] buttons, boolean isBotFirst, boolean isMaxingX) {
    int[] maxValues = new int[3];
    int maxVal = -64;
    int maxRow = -1;
    int maxCol = -1;

    int curPlayerOScore = playerOScore;
    int curPlayerXScore = playerXScore;

    if (curDepth == maxDepth || curDepth == maxDepthCheck) {
        maxValues[0] = curPlayerOScore - curPlayerXScore;
        maxValues[1] = maxRow;
        maxValues[2] = maxCol;
        //=====
        Instant endTime = Instant.now();
        Duration duration = Duration.between(startTime, endTime);
        System.out.println(duration.toMillis() + " - " + curDepth);
        //=====
        return maxValues;
    }
}
```

3. Dilakukan **iterasi** pada tiap elemen *button* untuk mencari button yang masih belum terisi bidak. Kemudian jika menemukan, akan dibuat state button baru yang merupakan hasil bayangan jika bidak diletakan di lokasi koordinat *button* ini. Kemudian dicarikan nilai perbedaan skor akibat dari peletakan bidak di lokasi tersebut dengan `updateGameBoard` dan menyesuaikan perubahan tersebut ke skor kedua player

```
curButtons[i][j].setText("O");

int difference = updateGameBoard(!isMaxingX, i, j, curButtons);
curPlayerOScore += (difference + 1);
curPlayerXScore -= difference;
```

4. Dari state pengandaian tersebut, state tersebut dan informasi yang diperlukan seperti skor dimasukan ke dalam fungsi **min** untuk dicari nilai *children* dari node state pengandaian

tersebut, dengan kedalaman bertambah satu dan skor yang sudah berubah akibat bidak tadi.

5. Di dalam **min** dilakukan inisialisasi nilai yang paling minimum jika dipilih peletakan bidak dari state tersebut.

```
int[] minValues = new int[3];
int minVal = 64;
int minRow = -1;
int minCol = -1;

int curPlayerOScore = playerOScore;
int curPlayerXScore = playerXScore;
```

6. Method **min** memilih suatu state baru dengan peletakan bidak baru yang akan dicari state *children* tersebut. Kemudian perubahan pada state tersebut dihitung skornya dengan method *updateGameBoard*

```
curButtons[i][j].setText("X");

int difference = updateGameBoard(isMaxingX, i, j, curButtons);
curPlayerOScore -= difference;
curPlayerXScore += (difference + 1);
```

7. Informasi dari *state* baru tersebut digunakan lagi untuk masuk ke method *max* lagi, dan siklus ini akan berlangsung hingga keadaan pruning terpenuhi atau dicapai kedalaman *maxDepth*.

```
int[] maxValues = max(curPlayerOScore, curPlayerXScore, curDepth: curDepth + 1, maxDepth, curButtons, isBotFirst, isMaxingX);
```

8. Saat fungsi **min** atau **max** dilakukan, setelah nilai max atau min Value dari *children states* nya didapatkan dilakukan perbandingan dengan nilai value yang paling tinggi/rendah didapat sejauh ini di iterasi pada kedalaman ini. .

```
if (maxValues[0] < minVal) {
    minVal = maxValues[0];
    minRow = i;
    minCol = j;
}
```


9. Selain itu, dicek juga nilai value yang didapat apakah memenuhi kondisi *pruning* pada *min* atau *max*. Pada *max*, jika nilai value dari *children state* yang dicek sudah lebih besar atau sama dengan dari nilai beta, maka akan dilakukan pruning karena sudah pasti nilai ini yang akan dipilih dan tidak perlu lagi menelusuri *children* yang lain.

```
if (maxVal >= beta) {  
    maxValues[0] = maxVal;  
    maxValues[1] = maxRow;  
    maxValues[2] = maxCol;  
    return maxValues;  
}
```

Dan nilai alpha diubah jika valuenya lebih besar dari nilai alpha dari *tree* pencarian ini.

```
if (maxVal > alpha) {  
    alpha = maxVal;  
}
```

10. Pada method *min*, dilakukan hal yang serupa dengan hal tersebut namun menggunakan variabel *beta*.

```
if (minVal <= alpha) {  
    minValues[0] = minVal;  
    minValues[1] = minRow;  
    minValues[2] = minCol;  
    return minValues;  
}  
if (minVal < beta) {  
    beta = minVal;  
}
```

11. Kemudian, pada kasus tidak dilakukan *pruning* maka akan dikembalikan nilai value yang diminimalkan/dimaksimalkan pada method *min/max* tersebut

```
minValues[0] = minVal;  
minValues[1] = minRow;  
minValues[2] = minCol;  
return minValues;
```

12. Nilai *value* yang dikembalikan oleh fungsi ***min/max*** akan dievaluasi pada state *parentnya* hingga mencapai ke state paling atas dan akan kembalikan koordinat *button* yang dipilih untuk digerakan pada state saat ini.

```
return new int[]{maxValues[1], maxValues[2]};
```

3. Alternatif Pencarian Solusi dengan Local Search

Kami menggunakan algoritma *Hill-Climbing with sideways move* dalam *Local Search* yang digunakan bot ini. Berikut adalah penjelasan dan screenshot algoritma *Local Search* yang diimplementasikan.

- a. **Public int[] move (int roundsLeft, boolean isBotFirst, int playerXScore, Button[][] buttons, boolean isMaximizing):** Merupakan fungsi yang dipanggil oleh Bot untuk menentukan kotak yang dipilih. Bot akan mencari objective value dari semua gerakan yang valid. Sebelum menambahkan objective value tersebut ke array (sesuai dengan definisi hill climbing with sideways move), akan dicek terlebih dahulu apakah objective value memenuhi syarat ($\text{objectiveValue} \geq 0$), dengan kata lain objective function hanya ditambahkan ke array jika memiliki objective value yang lebih baik atau sama dengan objective value sekarang. Semua objective value yang ditambahkan ke array kemudian di-filter menjadi hanya yang memiliki objective value terbaik saja. Dari semua gerakan dengan objective value terbaik dipilih satu secara random. Gerakan yang terpilih tersebut akan di-return kepada Game untuk dijalankan.

```
1 public int[] move(int roundsLeft, boolean isBotFirst, int playerOScore, int playerXScore, Button[][] buttons,
2 boolean isMaximizingX) {
3
4     int curPlayerOScore = playerOScore;
5     int curPlayerXScore = playerXScore;
6     Button[][] curButtons = new Button[ROW][COL];
7     for (int i = 0; i < ROW; i++) {
8         for (int j = 0; j < COL; j++) {
9             curButtons[i][j] = new Button(buttons[i][j].getText()); // Create a new Button with the same properties
10        }
11    }
12
13    List<List<Integer>> objectiveValues = new ArrayList<>();
14    for (int i = 0; i < ROW; i++) {
15        for (int j = 0; j < COL; j++) {
16            if (curButtons[i][j].getText().equals("")) {
17                if (isMaximizingX) {
18                    curButtons[i][j].setText("X");
19                } else {
20                    curButtons[i][j].setText("O");
21                }
22                int difference = updateGameBoard(true, i, j, curButtons);
23                int curValue;
24                if (isMaximizingX) {
25                    curPlayerXScore += (difference + 1);
26                    curValue = curPlayerXScore - playerXScore;
27                } else {
28                    curPlayerOScore += (difference + 1);
29                    curValue = curPlayerOScore - playerOScore;
30                }
31                if (curValue >= 0) {
32                    objectiveValues.add(List.of(curValue, i, j));
33                }
34            }
35        }
36    }
37    restoreButtons(curButtons, buttons);
38    curPlayerOScore = playerOScore;
39    curPlayerXScore = playerXScore;
40    }
41
42    filterObjectiveValues(objectiveValues);
43    return getRandomBestValues(objectiveValues);
44 }
```

- b. **Public void restoreButtons(Button[][] curButtons, Button[][] backUpButtons):** Berfungsi untuk mengembalikan gameBoard ke keadaan awal setelah dilakukan simulasi gerakan pengecekan objectiveValue kotak pada fungsi move.

```
1 public void restoreButtons(Button[][] curButtons, Button[][] backUpButtons) {  
2     for (int i = 0; i < ROW; i++) {  
3         for (int j = 0; j < COL; j++) {  
4             curButtons[i][j] = new Button(backUpButtons[i][j].getText()); // Create a new Button with the same properties  
5         }  
6     }  
7 }
```

- c. **Public void filterObjectiveValues(List<List<Integer>> objectiveValues):** Prosedur yang akan dipanggil oleh fungsi move untuk melakukan filtering terhadap semua objectiveValue kotak pada array, sehingga hanya menyisakan kotak dengan objectiveValue terbaik saja.

```
1 public void filterObjectiveValues(List<List<Integer>> objectiveValues) {  
2     int maxLeftValue = objectiveValues.stream()  
3         .max(Comparator.comparing(triplet -> triplet.get(0)))  
4         .map(triplet -> triplet.get(0))  
5         .orElse(Integer.MIN_VALUE);  
6     objectiveValues.removeIf(triplet -> triplet.get(0) != maxLeftValue);  
7 }
```

- d. **Public int[] getRandomBestValues(List<List<integer>> objectiveValues):** Fungsi untuk memilih satu gerakan terbaik secara random setelah dilakukan filtering terhadap objectiveValues, sehingga dari beberapa objectiveValue terbaik hasil filtering hanya satu yang dipilih dan dikembalikan kepada fungsi move.

```

1  public int[] getRandomBestValues(List<List<Integer>> objectiveValues) {
2      Collections.shuffle(objectiveValues, new Random());
3      int[] bestValues = new int[3];
4      if (!objectiveValues.isEmpty()) {
5          List<Integer> randomTriplet = objectiveValues.get(0);
6          for (int i = 0; i < 3 && i < randomTriplet.size(); i++) {
7              bestValues[i] = randomTriplet.get(i);
8          }
9      }
10     // Return a subset of bestValues
11     return new int[]{bestValues[1], bestValues[2]};
12 }

```

e. Justifikasi pemilihan *Hill Climbing with Sideways Move* untuk *Local Search*

Justifikasi pemilihan algoritma adalah karena pemilihan gerakan selanjutnya berdasarkan objective function tertinggi, sehingga memaksimalkan kemungkinan bot untuk memenangkan permainan. Jika dibandingkan dengan algoritma *Stochastic Hill Climbing* dan *Simulated Annealing*, jumlah ronde dan kotak terbatas akan membatasi peluang kemenangan, mengingat kedua algoritma tersebut memilih gerakan secara random sehingga mungkin menguntungkan musuh. Selain itu, *Simulated Annealing* hanya optimal jika nilai T turun secara perlahan, padahal pada permainan ini hanya terdapat 56 kemungkinan gerakan. Oleh karena itu, penggunaan random untuk memilih gerakan selanjutnya dinilai tidak efektif.

Jika dibandingkan dengan *Steepest Hill Climbing*, terdapat kemungkinan bot akan menemukan shoulder sehingga menghasilkan gerakan yang tidak valid. Untuk menghindari error akibat gerakan tidak valid, digunakan algoritma *Hill Climbing with Sideways Move* untuk *Local Search*.

4. Strategi Genetic Algorithm

Genetic Algorithm yang diimplementasi pada bot *GeneticBot* mengikuti algoritma genetik pada umumnya, namun dengan modifikasi pada bagian evaluasi *fitness function value* dengan penggunaan local search. Pada “kromosom” yang digunakan sebagai populasi, terdiri atas kumpulan point yang akan dilakukan oleh bot, contohnya $[[1,2],[4,6]]$. Pada *fitness function*, akan disimulasikan aksi tiap point dalam kromosom tersebut dan mencari gerakan yang mungkin dilakukan oleh pemain lawan menggunakan local search. Kemudian hasil simulasinya akan diambil nilai *heuristic* sebagai *fitness value*-nya.

Genetic Algorithm memiliki kelebihan dari sisi *space complexity* dan *time complexity* dibandingkan dengan minimax alpha beta pruning, terutama dalam permainan yang membutuhkan penelusuran *game tree* dengan *branching factor* dan *depth* yang tinggi seperti pada Adjacency Strategy Game yang diberikan.

Namun, kekurangan penerapan Genetic Algorithm pada Adjacency Strategy Game adalah tidak terjaminnya pilihan langkah optimum dan terbentuknya *illegal offspring* karena kemungkinan langkah yang sangat banyak pada node di level kedalaman yang sama. Hal tersebut menyebabkan algoritma perlu memperbaiki *illegal offspring* yang akan menghabiskan waktu komputasi.

Berikut adalah garis besar dan gambaran umum strategi pencarian langkah optimum pada Adjacency Strategy Game.

1. Mengenerasi populasi dasar secara acak.

```
int[][][] base = new int[POPULATION_COUNT][selectedDepth][];
Random random = new Random();

for (int i=0; i<POPULATION_COUNT;i++) {
    for (int j=0; j<selectedDepth;j++) {
        base[i][j] = new int[2];
        int selectednum = random.nextInt(availableMoves.length);
        base[i][j][0] = availableMoves[selectednum][0];
        base[i][j][1] = availableMoves[selectednum][1];
    }
}
```

2. Melakukan penilaian *fitness value* pada setiap kromosom

```

public int fitnessValue(int[][] arr) {
    Button[][] curButtons = new Button[ROW][COL];
    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {
            curButtons[i][j] = new Button(this.localButtons[i][j].getText()); // Create a new Button with the same properties
        }
    }
    int eval = 0;
    int difference = 0;
    for (int i = 0; i < selectedDepth; i++) {

        if (this.localIsMaximizingX) {
            curButtons[(arr[i][0])][(arr[i][1])].setText("X");
        } else {
            curButtons[(arr[i][0])][(arr[i][1])].setText("0");
        }

        difference = updateGameBoard( isBot: true, (arr[i][0]), (arr[i][1]), curButtons);
        int[] b = new int[2];
        b = this.local.move(this.localRoundsLeft, this.localIsBotFirst, this.localPlayer0Score, this.localPlayerXScore,
            curButtons, !(this.localIsMaximizingX));

        if (contains(getPartition(arr,i),b)) {
            return 0;
        }
    }
    if (this.localIsMaximizingX) {
        curButtons[(b[0])][(b[1])].setText("0");
    }
}

```

```

        if (this.localIsMaximizingX) {
            curButtons[(b[0])][(b[1])].setText("0");
        } else {
            curButtons[(b[0])][(b[1])].setText("X");
        }
    }
    difference = updateGameBoard( isBot: true, (b[0]), (b[1]), curButtons);
}

int playerXScore = 0;
int player0Score = 0;
for (int i = 0; i < ROW; i++) {
    for (int j = 0; j < COL; j++) {
        if (curButtons[i][j].getText().equals("X")) {
            playerXScore++;
        } else if (curButtons[i][j].getText().equals("0")) {
            player0Score++;
        }
    }
}

if (localIsMaximizingX) {
    return playerXScore-player0Score;
}

return player0Score-playerXScore;
}

```

3. Melakukan *selection* pada populasi dasar berdasarkan *fitness value*.

```

//Selection
int generation_count = 1;
int[] evalList = new int[POPULATION_COUNT];
evaluateList(base, evalList);
int[] sums = selectionList(evalList);
int totalsum = sumList(evalList);

int[][][] offspring = new int[POPULATION_COUNT][selectedDepth][];
for (int i = 0; i<POPULATION_COUNT;i++) {
    offspring[i] = getOffspring(base,sums,totalsum);
}

```

```

public void evaluateList(int[][][] arr, int[] list){
    for (int i = 0; i<arr.length;i++) {
        list[i] =fitnessValue(arr[i])+65;
    }
}

```

```

public int[][] getOffspring(int[][][] arr, int[] sumList, int sum) {
    int i = 0;
    Random random = new Random();
    int sel = random.nextInt(sum);
    boolean loop = true;
    while (sel>sumList[i] && loop) {
        if (i < sumList.length-1) {
            i++;
        } else {
            loop = false;
        }
    }
    return arr[i];
}

```

4. Melakukan *crossover* pada *offspring* atau populasi hasil seleksi.

```

for (int j = 0; j<Math.floorDiv(offspring.length,2);j++) {
    int[][][] crossed = cross(offspring[j],offspring[j+1],Math.floorDiv(selectedDepth,2));
    offspring[j] = crossed[0];
    offspring[j+1] = crossed[1];
}

```



```

public static int[][][] cross(int[][] arr1, int[][] arr2, int cpoint) {
    int[][] offspring1 = new int[arr1.length][];
    int[][] offspring2 = new int[arr2.length][];

    for (int i = 0; i < cpoint; i++) {
        offspring1[i] = arr1[i];
        offspring2[i] = arr2[i];
    }

    for (int i = cpoint; i < arr1.length; i++) {
        offspring1[i] = arr2[i];
        offspring2[i] = arr1[i];
    }

    int[][][] offspring = { offspring1, offspring2 };
    return offspring;
}

```

5. Melakukan mutasi pada populasi hasil *crossover*

```

for (int k = 0; k < offspring.length; k++) {
    if (random.nextInt( bound: 101) < MUTATION_CHANCE * 100) {
        int selectednum = random.nextInt(availableMoves.length);
        int mutation_point = random.nextInt(offspring[k].length);
        offspring[k][mutation_point][0] = availableMoves[selectednum][0];
        offspring[k][mutation_point][1] = availableMoves[selectednum][1];
    }
}

```

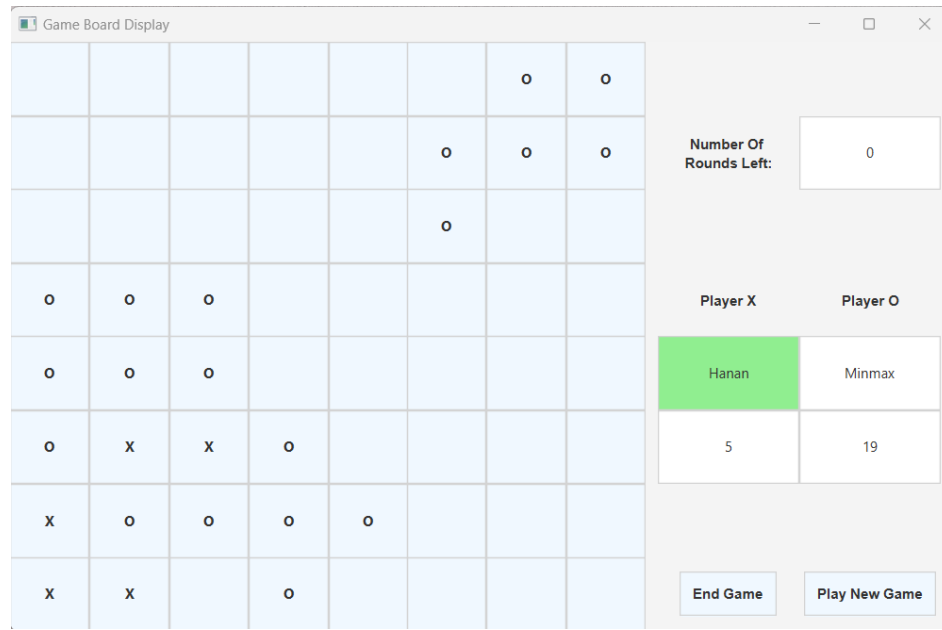
6. Ulangi langkah 3 sampai 5 hingga kriteria pemberhentian dipenuhi
7. Memilih kromosom dengan fitness value tertinggi

5. Hasil Pertandingan

Untuk video demonstrasi pertandingan dapat dilihat pada [LINK BERIKUT](#)

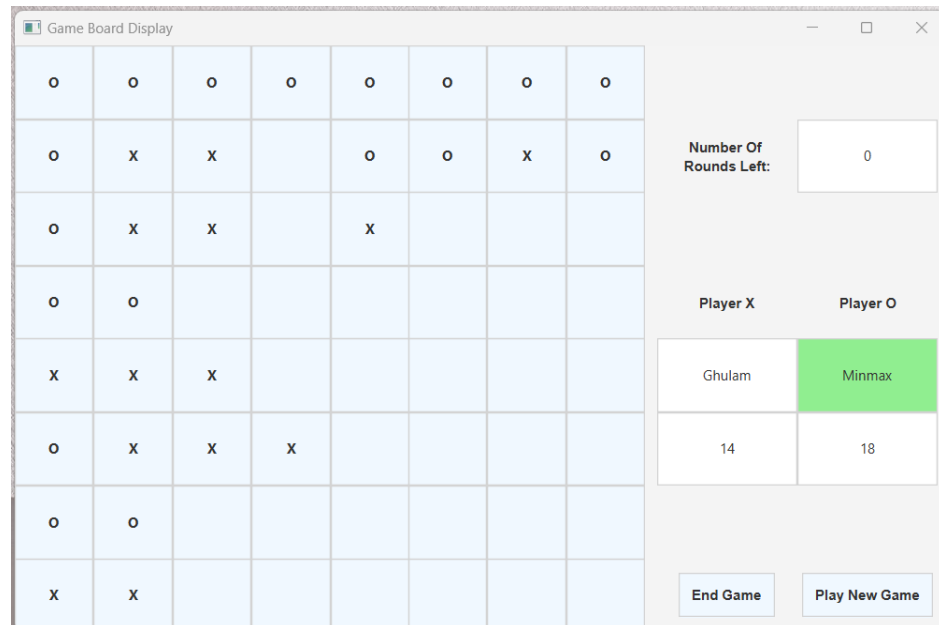
1. Minimax vs Manusia

a. 8 Ronde



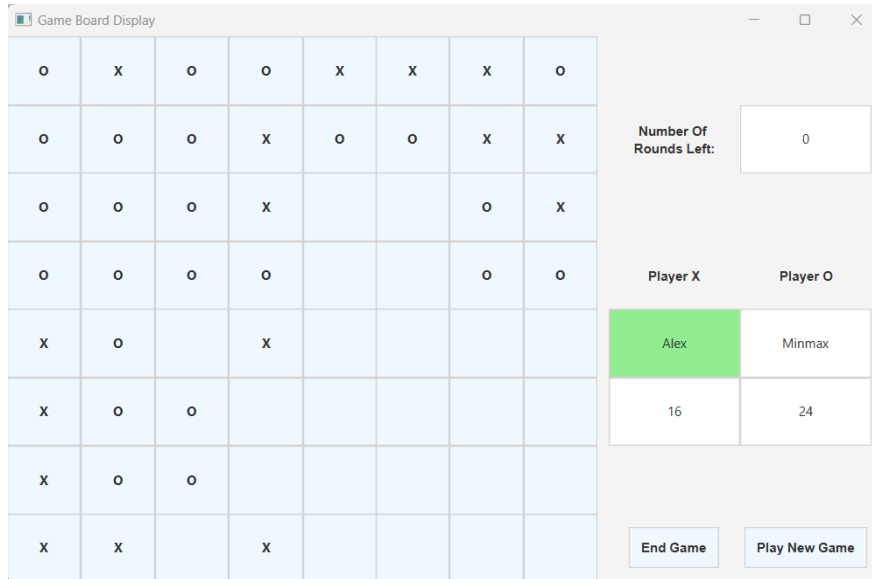
Gambar 5.1.1. Minimax vs Manusia 8 Ronde. Dimenangkan Minimax

b. 12 Ronde



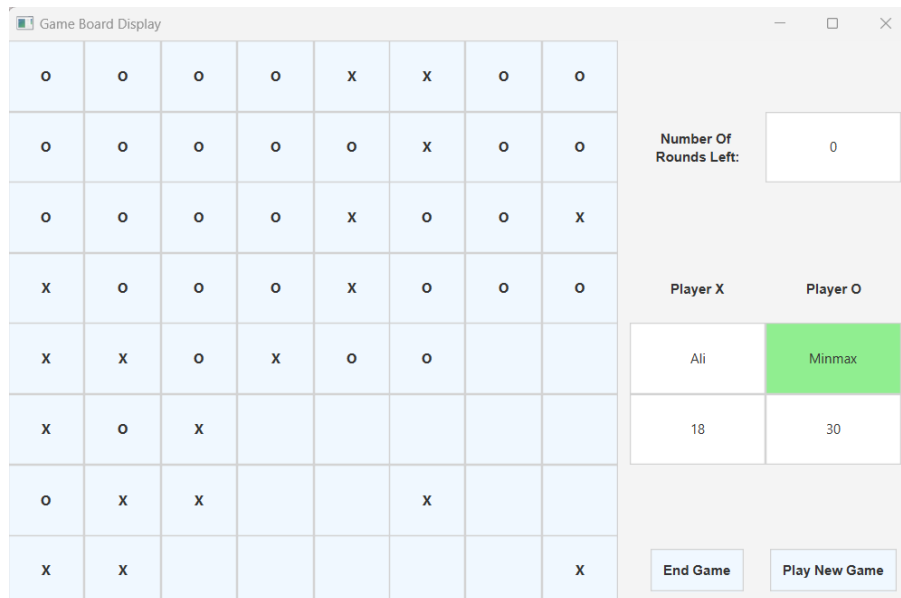
Gambar 5.1.2. Minimax vs Manusia 12 Ronde. Dimenangkan Minimax

c. 16 Ronde



Gambar 5.1.3. Minimax vs Manusia 16 Ronde. Dimenangkan Minimax

d. 20 Ronde



Gambar 5.1.4. Minimax vs Manusia 20 Ronde. Dimenangkan Minimax

e. 28 Ronde



Gambar 5.1.5. Minimax vs Manusia 28 Ronde. Dimenangkan Minimax

f. Persentase Kemenangan

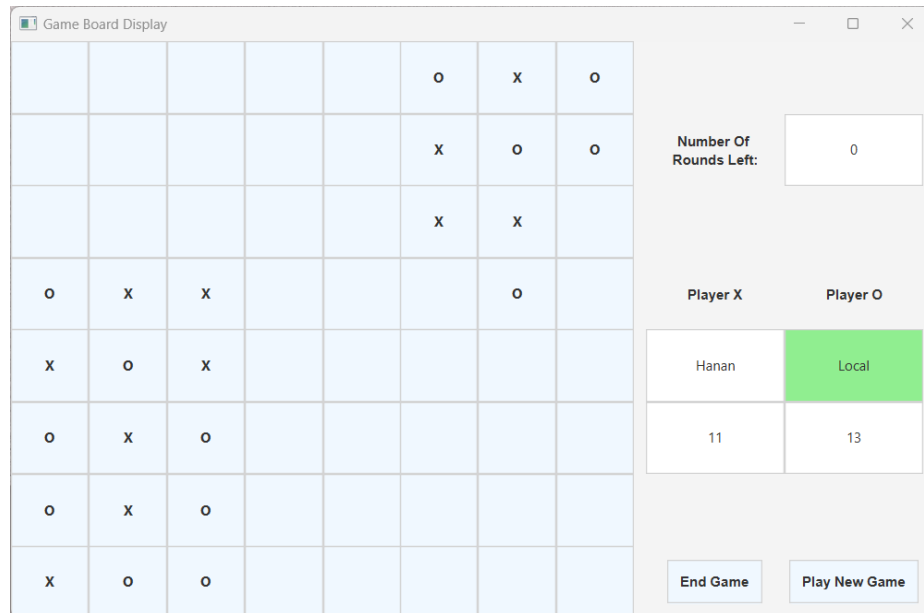
Tabel 5.1.1. Persentase Kemenangan

Player	Menang	Kalah	Persentase
Minimax	5	0	100%
Manusia	0	5	0%

Analisis hasil pertandingan Minimax vs Manusia: Hasil pertandingan sesuai harapan karena Minimax dapat mengetahui langkah terbaik ke depan. Dengan begitu, Minimax akan memilih kotak dengan kemungkinan menang terbesar jika dipilih pada ronde saat itu. Tetapi, hal yang perlu diperhatikan adalah karena kedalaman pencarian (*depth*) yang dibatasi pada setiap ronde, terdapat kemungkinan manusia dapat mengakali algoritma Minimax sehingga menjebak Minimax untuk mengisi kotak-kotak tertentu dan menyebabkan manusia untuk mendapat keuntungan skor yang besar pada ronde-ronde tertentu.

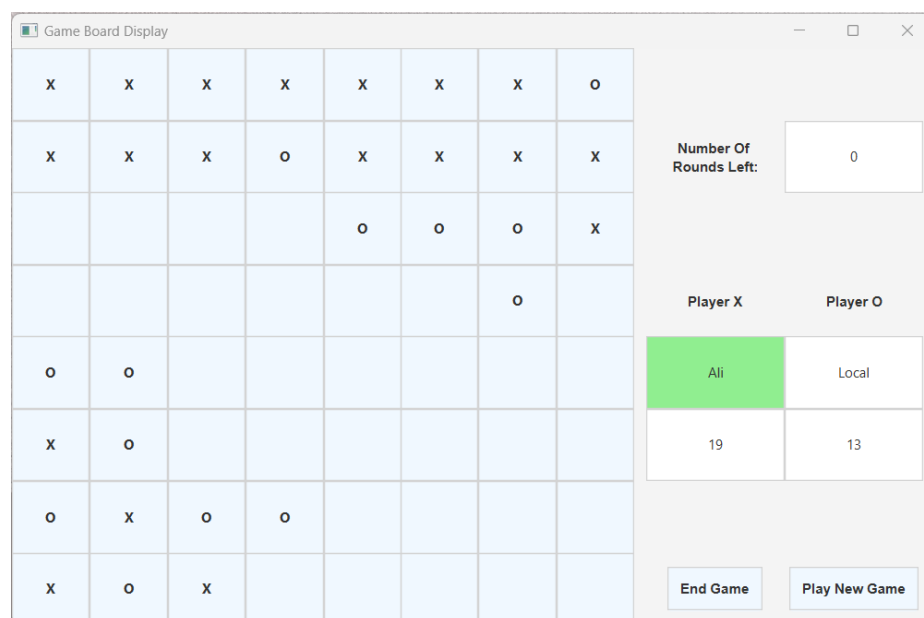
2. Local Search vs Manusia

a. 8 Ronde



Gambar 5.2.1. Local Search vs Manusia 8 Ronde. Dimenangkan Local Search

b. 12 Ronde



Gambar 5.2.2. Local Search vs Manusia 12 Ronde. Dimenangkan Manusia

c. 28 Ronde



Gambar 5.2.3. Local Search vs Manusia 28 Ronde. Dimenangkan Local Search

d. Persentase Kemenangan

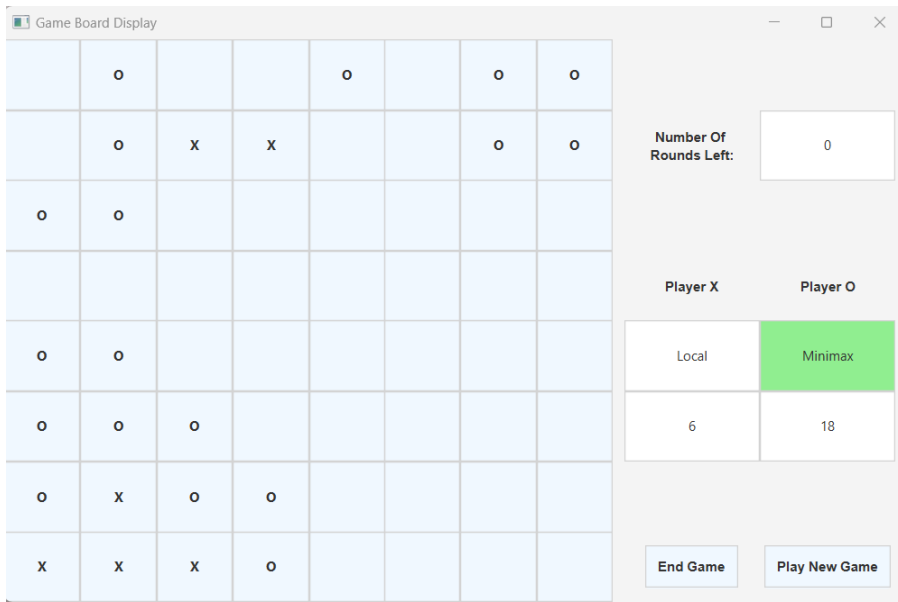
Tabel 5.2.1. Persentase Kemenangan

Player	Menang	Kalah	Persentase
Local Search	2	1	66,6%
Manusia	1	2	33,3%

Analisis hasil pertandingan Local Search vs Manusia: Hasil sesuai dengan yang diharapkan karena algoritma Local Search masih dapat memenangkan pertandingan meskipun hanya melakukan kalkulasi hanya pada ronde tersebut. Dibandingkan dengan Minimax, kemungkinan menang algoritma Local Search akan lebih sedikit, karena tidak dapat mempertimbangkan keadaan gameBoard pada giliran musuh dan giliran bot berikutnya.

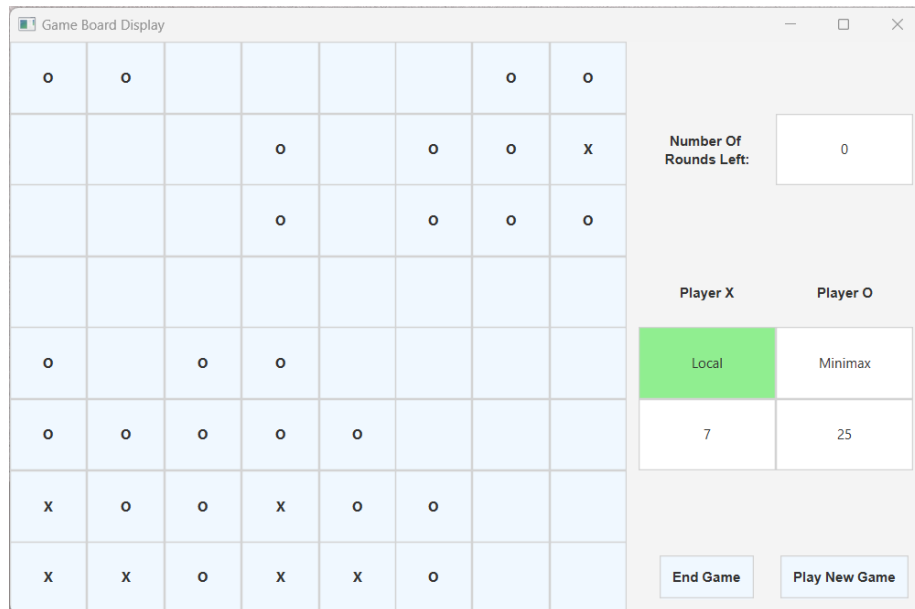
3. Minimax vs Local Search

a. 8 Ronde



Gambar 5.3.1. Minimax vs Local Search 8 Ronde. Dimenangkan Minimax

b. 12 Ronde



Gambar 5.3.2. Minimax vs Local Search 12 Ronde. Dimenangkan Minimax

c. 28 Ronde



Gambar 5.3.3. Minimax vs Local Search 28 Ronde. Dimenangkan Minimax

d. Persentase Kemenangan

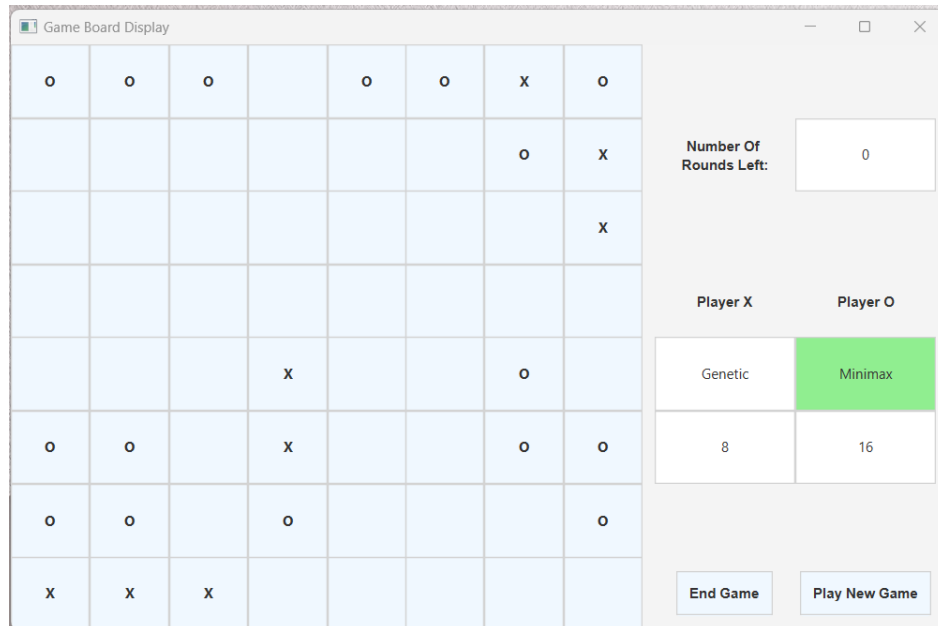
Tabel 5.3.1. Persentase Kemenangan

Player	Menang	Kalah	Persentase
Minimax	3	0	100%
Local Search	0	3	0%

Analisis hasil pertandingan Minimax vs Local Search: Hasil sesuai dengan yang diharapkan, yaitu Minimax mengalahkan Local Search pada pertandingan. Hal yang perlu menjadi perhatian adalah terdapat kecenderungan bagi algoritma Local Search untuk memiliki skor lebih tinggi pada ronde-ronde awal pertandingan, tetapi Minimax akan memiliki skor yang jauh lebih tinggi pada ronde-ronde akhir pertandingan. Terbukti berdasarkan hasil foto, perbedaan skor pada ronde yang sedikit lebih kecil dibandingkan dengan perbedaan skor pada jumlah ronde yang lebih banyak. Hal tersebut mungkin terjadi karena Local Search hanya mementingkan skor pada ronde saat itu juga, sementara Minimax memperhitungkan skor pada ronde-ronde selanjutnya, termasuk juga memprediksi gerakan lawan yang paling optimal.

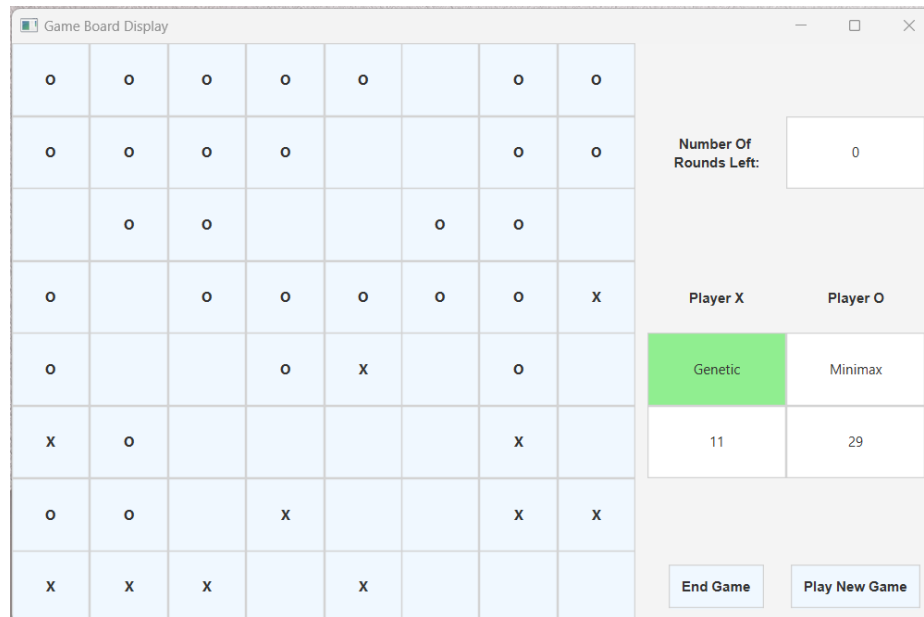
4. Minimax vs Genetic

a. 8 Ronde



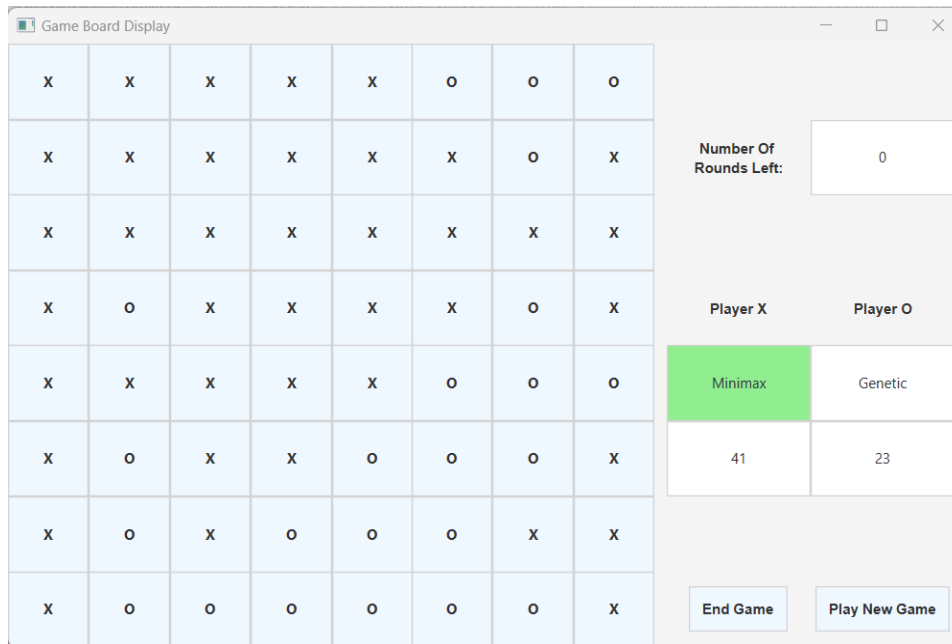
Gambar 5.4.1. Minimax vs Genetic 8 Ronde. Dimenangkan Minimax

b. 16 Ronde



Gambar 5.4.2. Minimax vs Genetic 16 Ronde. Dimenangkan Minimax

c. 28 Ronde



Gambar 5.4.3. Minimax vs Genetic 28 Ronde. Dimenangkan Minimax

d. Persentase Kemenangan

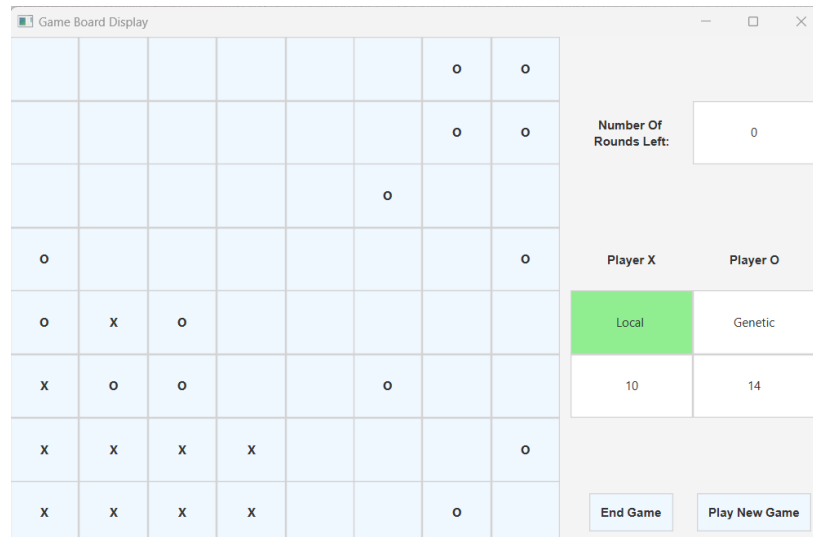
Tabel 5.4.1. Persentase Kemenangan

Player	Menang	Kalah	Persentase
Minimax	3	0	100%
Genetic	0	3	0%

Analisis hasil pertandingan Minimax vs Genetic: Hasil sesuai dengan yang diharapkan, yaitu Minimax memenangkan pertandingan. Hal tersebut karena Genetic Search melakukan gerakan terbaik secara *random*, sedangkan Minimax menentukan gerakan terbaik. Terlebih, untuk Genetic Search terdapat batasan waktu yang membatasi kemampuan pencarian populasi dan generasi yang lebih baik.

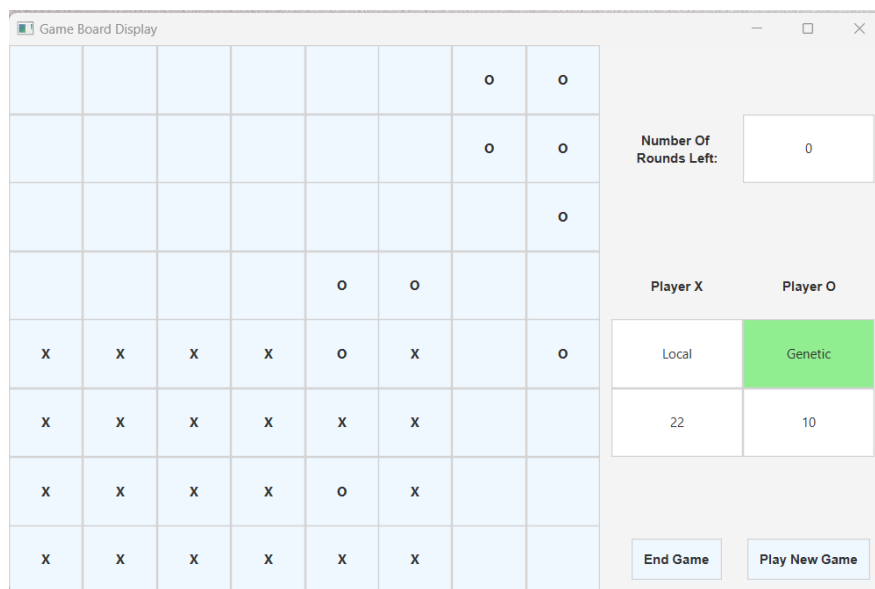
5. Local Search vs Genetic

a. 8 Ronde



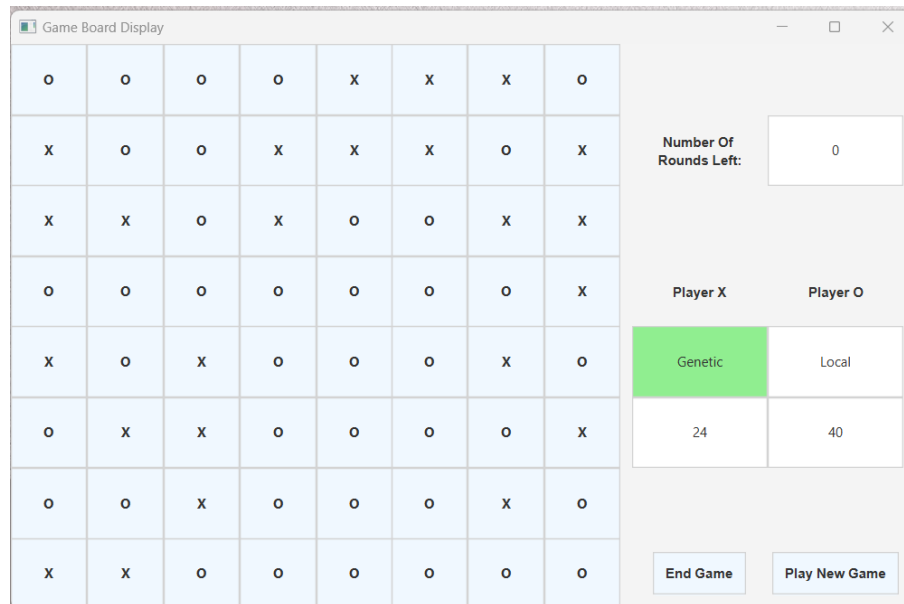
Gambar 5.5.1. Local Search vs Genetic 8 Ronde. Dimenangkan Genetic

b. 12 Ronde



Gambar 5.5.2. Local Search vs Genetic 12 Ronde. Dimenangkan Local Search

c. 28 Ronde



Gambar 5.5.3. Local Search vs Genetic 28 Ronde. Dimenangkan Local Search

d. Persentase Kemenangan

Tabel 5.5.1. Persentase Kemenangan

Player	Menang	Kalah	Persentase
Local Search	2	1	66,6%
Genetic	1	2	33,3%

Analisis hasil pertandingan Local Search dan Genetic Search: Hasil sesuai dengan yang diharapkan. Hal tersebut karena Genetic Search melakukan gerakan terbaik secara random, sedangkan Local Search melakukan gerakan terbaik setelah mengiterasi semua kemungkinan gerakan. Terlebih, untuk Genetic Search terdapat batasan waktu yang membatasi kemampuan pencarian populasi dan generasi yang lebih baik.

Pembagian Tugas

No	Nama	NIM	Pembagian Tugas
1.	Muhammad Hanan	13521041	Minimax
2.	Alex Sander	13521061	Genetic
3.	Ahmad Ghulam Ilham	13521118	Local Search
4.	Muhammad Abdul A. G.	13521128	Minimax

Referensi

- [Minimax with Alpha-Beta Pruning in Python \(stackabuse.com\)](https://stackabuse.com/minimax-alpha-beta-pruning-in-python/)
- [Minimax Algorithm in Game Theory | Set 1 \(Introduction\) - GeeksforGeeks](https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/)
- [Minimax Algorithm in Game Theory | Set 4 \(Alpha-Beta Pruning\) - GeeksforGeeks](https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/)
- [\(PDF\) Adversarial Search by Evolutionary Computation \(researchgate.net\)](https://researchgate.net/publication/312111111_PDF_Adversarial_Search_by_Evolutionary_Computation)
- [Algorithms Explained – minimax and alpha-beta pruning - YouTube](https://www.youtube.com/watch?v=73113333333)
- Modul 3: Beyond Classical Search, PPT Intelegensi Buatan [Beyond Classical Search](#)