MINIMAX ALGORITHM AND ALPHA BETA PRUNING IN ADJACENCY STRATEGY GAME

Diajukan sebagai pemenuhan tugas besar I.

Oleh:

Kelompok 5

1. 13521100 - Alexander Jason
2. 13521116 - Juan Christopher Santoso
3. 13521139 - Nathania Calista Djunaedi
4. 13521162 - Antonio Natthan Krishna

Dosen Pengampu :

1. Dr. Nur Ulfa Maulidevi, S.T, M.Sc.
2. Dr. Eng. Ayu Purwarianti, S.T., M.T.

IF3170 - Inteligensi Buatan

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2023

Daftar Isi

Daftar Isi..... 2
I. Objective Function..... 3
II. Pengembangan Agent Menggunakan Algoritma Minimax Alpha Beta Pruning.....4
1. Deskripsi Algoritma..... 4
2. Fungsi Heuristik..... 6
3. Cuplikan Kode..... 7
III. Pengembangan Agent Menggunakan Algoritma Local Search..... 14
1. Deskripsi Algoritma..... 14
2. Fungsi Heuristik..... 16
3. Cuplikan Kode..... 18
4. Perbandingan Algoritma..... 25
IV. Pengembangan Agent Menggunakan Genetic Algorithm.....26
1. Deskripsi Algoritma..... 26
2. Fungsi Heuristik..... 28
3. Cuplikan Kode..... 28
V. Pengujian..... 39
1. Bot MiniMax vs Manusia..... 39
2. Bot Local Search vs Manusia..... 43
3. Bot Minimax vs Bot Local Search.....46
4. Bot Minimax vs Bot Genetic Algorithm..... 48
5. Bot Local Search vs Bot Genetic Algorithm..... 51
6. Rangkuman Pertandingan.....53
VI. Kontribusi Anggota..... 54
VII. Lampiran..... 55

I. *Objective Function*

Objective function merupakan sebuah fungsi yang akan mengembalikan nilai dari suatu state. Nilai ini akan dihitung saat state sudah berada dalam konfigurasi lengkap. Nilai dari suatu state didefinisikan sebagai berikut,

1. Nilai state pada awalnya bernilai 0.
2. Setiap kotak yang masih kosong bernilai 0.
3. Nilai state bertambah 1 untuk setiap kotak yang dimiliki oleh *agent*.
4. Nilai state berkurang 1 untuk setiap kotak yang dimiliki oleh lawan.

Nilai *state* positif menandakan bahwa *agent* berada dalam posisi yang lebih unggul dibandingkan lawan dan begitu sebaliknya. Hal ini membuat *agent* selalu berusaha untuk memaksimalkan nilai state, dimana lawan selalu berusaha untuk meminimalkan nilai state. Nilai state yang didapat dari *objective function* ini selanjutnya dapat digunakan untuk menentukan langkah yang akan diambil oleh *agent* untuk memenangkan permainan.

II. Pengembangan *Agent* Menggunakan Algoritma *Minimax Alpha Beta Pruning*

1. Deskripsi Algoritma

Algoritma *minimax* adalah algoritma yang memanfaatkan konsep rekursif atau algoritma *backtracking* dan digunakan untuk membuat keputusan atau dalam *game theory*. Algoritma ini menghasilkan *action* yang paling optimal untuk *player* dengan cara mengasumsikan bahwa musuh dari *player* bermain dengan optimal juga. Contoh penerapan algoritma *minimax* adalah dalam permainan *tic-tac-toe, go,* catur, dan lain - lain.

Dalam proses pencariannya, terdapat dua buah pemain yang diberi nama MAX dan MIN. Jika pemain MAX mendapat giliran, algoritma akan memilih *state* yang paling menguntungkan atau dengan kata lain, menghasilkan nilai *objective* tertinggi jika dibandingkan dengan *state - state* lainnya. Sebaliknya, jika pemain MIN mendapat giliran, algoritma akan memilih *state* yang paling merugikan atau dengan kata lain, menghasilkan nilai *objective* terkecil jika dibandingkan dengan *state - state* lainnya. Algoritma pencarian yang digunakan adalah *depth-first search*. Jadi, algoritma *minimax* akan melakukan proses pencarian sampai ke *terminal node* terlebih dahulu baru kemudian melakukan *backtracking* dengan menggunakan proses rekursif.

Pada permainan *adjacency strategy*, ada dua buah pemain yang akan saling berkompetisi untuk mendapatkan kotak terbanyak. Kedua buah pemain digambarkan dengan dua buah simbol, yaitu X dan O. Pada awal permainan, pemain dipersilakan untuk memilih simbol representasi pemain, algoritma yang ingin digunakan, jumlah ronde, dan menentukan pemain dengan giliran pertama. Pemain yang memilih *minimax algorithm* adalah pemain yang nantinya akan dianggap sebagai pemain MAX dalam algoritma minimax.

Algoritma *minimax* dimulai dengan membangkitkan *successor state* dari *current state* atau kondisi permainan pada saat ini. *Successor state* pada permainan ini adalah semua kotak yang masih kosong. Karena papan permainan berukuran 8x8 dengan 8 kotak sudah terisi pada awal permainan,maka kurang

lebih terdapat sekitar 56! *Tree* yang dapat dibangkitkan pada ronde pertama. Hal ini sangat memakan banyak waktu karena program harus menelusuri 56! *Tree* sebelum mengambil keputusan yang terbaik berdasarkan algoritmanya. Oleh karena itu, *successor state* yang dibangkitkan akan diseleksi lagi dengan 2 *heuristic function* agar memperkecil jumlah *tree* yang perlu dibangkitkan.

Setelah membatasi jumlah *successor state* yang dibangkitkan, selanjutnya algoritma *minimax* akan melakukan penelusuran *tree* sampai *terminal node*. Untuk kasus permainan *adjacency strategy,* kedalaman *tree* bergantung kepada jumlah ronde yang ingin dimainkan. Misalnya, kalau pemain memilih permainan dengan ronde sebanyak 28 ronde, kedalaman *tree* dapat mencapai 56 pada ronde awal karena ada 2 pemain yang harus jalan dalam 1 ronde. Sehingga, 1 ronde digambarkan dengan 2 *level tree.* Untuk mempercepat pencarian, penulis membatasi kedalaman *tree* yang dapat ditelusuri. Harapannya, dengan membatasi kedalaman *tree* yang dapat ditelusuri, algoritma *minimax* tetap dapat memberikan langkah paling optimal dan dalam waktu yang lebih singkat.

Ketika proses pencarian mencapai *terminal node* atau *depth* tertentu, algoritma akan memanggil *objective function* yang mengembalikan nilai *state* pada saat itu. Setelah mengetahui nilai *objective* dari *terminal node*, proses pencarian akan mulai melakukan *backtracking.* Pada setiap *node,* nilai yang dipilih akan bergantung pada giliran pemain. Jika sekarang merupakan giliran pemain MAX, program akan memilih *child node* dengan nilai *objective* terbesar. Sedangkan, jika sekarang merupakan giliran pemain MIN, program akan memilih *child node* dengan nilai *objective function* terkecil. Proses pencarian akan terus dilakukan sampai kembali ke *root node* atau *current state*.

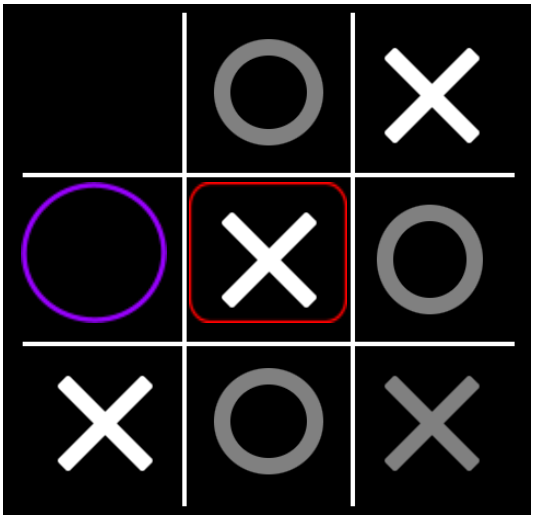
Untuk mengoptimalisasi proses pencarian, algoritma *minimax* dapat dikombinasikan dengan *alpha beta pruning.* Algoritma ini digunakan untuk memangkas (*pruning*) node - node yang dianggap tidak akan memberikan solusi optimal. Pada algoritma ini, terdapat dua variabel, yaitu *alpha* dan *beta. Alpha* menggambarkan nilai tertinggi sejauh ini yang diperoleh pemain MAX, sedangkan *beta* menggambarkan nilai terendah sejauh ini yang diperoleh pemain MIN. Pada awal proses pencarian, *alpha* akan diberikan nilai negatif tak

hingga dan *beta* akan diberikan nilai positif tak hingga. Ketika terjadi sebuah situasi dimana nilai *alpha* lebih besar sama dengan nilai *beta,* maka proses pencarian di *node* tersebut akan dipangkas.

2. Fungsi Heuristik

Dalam algoritma MiniMax, penulis menggunakan 2 *heuristic function*. Pendekatan *heuristic* pertama yang penulis gunakan dalam algoritma ini adalah kotak yang memiliki tetangga yang berisi simbol lawan. Penulis memilih *heuristic* ini dengan pertimbangan *agent* memprioritaskan kotak yang dapat menghasilkan poin tambah paling banyak. Implementasi *heuristic function dilakukan* dengan memberikan nilai +1 untuk setiap tetangga dari kotak yang akan dipilih yang tidak kosong dan merupakan kotak yang dimiliki lawan.

Penulis juga menggunakan pendekatan *heuristic function* kedua, dimana *heuristic* ini mencegah langkah yang dipilih bot untuk memberi lawan kesempatan untuk menghasilkan poin. Fungsi tersebut akan memeriksa apabila *successor* tersebut memiliki tetangga diagonal yang berisi elemen yang sama dengan elemen *agent*. Jika kondisi tersebut terpenuhi, fungsi akan memeriksa apakah tetangga tersebut memiliki petak kosong yang bersisian. Jika iya, maka *heuristic function* akan mengembalikan nilai 1. Agar lebih jelas, dapat dilihat gambar berikut.



Pada gambar tersebut, nilai *heuristic function* kedua dari langkah yang diberi kotak merah adalah 1. Hal tersebut diakibatkan karena kotak tersebut memiliki tetangga diagonal dengan elemen yang sama, serta terdapat kotak kosong yang diberi tanda ungu, yang dapat menjadi kesempatan bagi lawan untuk mendapat poin lebih. Kotak kosong tersebut memiliki tetangga yang bersisian dengan lebih dari 1 kotak berisi elemen *agent.*

Bobot untuk kedua *heuristic function* yang digunakan adalah sebagai berikut:

$$h(x) = 2 * f(x) - g(x)$$

Dimana f(x) adalah *heuristic function* pertama dan *g(x)* adalah *heuristic function* kedua. Fungsi pertama memiliki bobot yang lebih tinggi karena penulis memprioritaskan *heuristic* untuk mengambil poin yang lebih banyak. *Heuristic* kedua digunakan untuk melihat apakah langkah tersebut akan memiliki *trade-off* yang memuaskan atau tidak.

3. Cuplikan Kode

1. Fungsi isValid

Proses	Memastikan apakah sebuah titik valid atau tidak
Keluaran	*boolean*

```
public boolean isValid(int x, int y){
        if(x >= 0 && x < 8 && y>= 0 && y < 8){
            return true;
        }
        return false;
    }
```

2. Fungsi calculateObjective

Proses	Menghitung *objective value* dari sebuah *state* permainan
Keluaran	int

```
public int calculateObjective(char[][] boardMap){
        int point = 0;
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                if (boardMap[i][j] == this.selfMark) {
                    point++;
                } else if (boardMap[i][j] == this.enemyMark)
{
                    point--;
                }
            }
        }
        return point;
    }
```

3. Fungsi searchAdjacent

Proses	Mengembalikan *ArrayList of PointValue* dari titik - titik yang bertetanggaan dengan lawan.
Keluaran	ArrayList<PointValue>

```
public ArrayList<PointValue> searchAdjacent(char[][]
boardMap, boolean isBot) {
        ArrayList<PointValue> pointValues = new
ArrayList<PointValue>();
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                int totalAdj =
searchTotalAdjacent(boardMap,isBot,new Point(i,j));
                int totalDia = searchDiagonal(boardMap,
isBot, new Point(i,j));
                int val = 2*totalAdj - totalDia ;
//                Memastikan bahwa boardMap[i][j] kosong dan
memiliki tetangga berupa musuh
                if (boardMap[i][j] == ' ' && totalAdj > 0) {
                    pointValues.add(new PointValue(i,j,val));
                }
```

```
            }
        }

pointValues.sort(Comparator.comparing(PointValue::getPointVal
ue).reversed());
        return pointValues;
    }
```

4. Fungsi searchTotalAdjacent

Proses	Mengembalikan jumlah tetangga yang dimiliki oleh sebuah titik
Keluaran	int

```
public int searchTotalAdjacent(char[][] boardMap,boolean
isBot ,Point point){
        char enemy = this.enemyMark;
        if(!isBot){
            enemy = this.selfMark;
        }
        int total = 0;
        int x = point.x;
        int y = point.y;
        // left
        if(isValid(x-1,y)&&boardMap[x-1][y] == enemy){
            total += 1;
        }
        // down
        if(isValid(x,y-1) && boardMap[x][y-1] == enemy){
            total += 1;
        }
        // right
        if(isValid(x+1,y) && boardMap[x+1][y] == enemy){
            total += 1;
        }
        // up
        if(isValid(x,y+1)&&boardMap[x][y+1] == enemy){
            total +=1;
        }
```

```
        return total;
    }
```

5. Fungsi *searchDiagonal*

Proses	Mencari jumlah diagonal sebuah *point* yang merupakan dirinya sendiri. Lalu memeriksa apakah titik di sebelah kiri / kanan / atas / bawah merupakan kotak kosong atau tidak. Kalau iya, maka akan return nilai 1.
Keluaran	int

```
public int searchDiagonal(char[][] boardMap,boolean isBot
,Point point){
        char self = this.selfMark;
        if(!isBot){
            self = this.enemyMark;
        }
        int total = 0;
        int x = point.x;
        int y = point.y;
        if(isValid(x-1,y-1)&&boardMap[x-1][y-1] == self){
            // left and down
            if((isValid(x-1,y)&&boardMap[x-1][y] == '
')||(isValid(x,y-1) && boardMap[x][y-1] == ' ')){
                total += 1;
            }
        }
        else if(isValid(x+1,y-1) && boardMap[x+1][y-1] ==
self){
            // right and down
            if((isValid(x+1,y)&&boardMap[x+1][y] == ' ') ||
(isValid(x,y-1) && boardMap[x][y-1] == ' ')){
                total += 1;
            }
        }
        else if(isValid(x+1,y+1) && boardMap[x+1][y+1] ==
```

```
self){
            // right and up
            if((isValid(x+1,y)&&boardMap[x+1][y] == ' ' )||(
isValid(x,y+1) && boardMap[x][y+1] == ' ')){
                total += 1;
            }
        }
        else if(isValid(x-1,y+1)&&boardMap[x-1][y+1] ==
self){
            // left and up
            if((isValid(x-1,y)&&boardMap[x-1][y] == '
')||(isValid(x,y+1) && boardMap[x][y+1] == ' ')){
                total += 1;
            }
        }
        return total;
    }
```

6. Fungsi *move*

Proses	Menggerakan bot
Keluaran	Array of integer

```
public int[] move(char[][] boardMap, int roundLeft, char
selfMark, char enemyMark) {
        this.root = new Tree(boardMap);
        this.selfMark = selfMark;
        this.enemyMark = enemyMark;
        this.maxDepth = 8;
        this.maxWidth = 0.5;
        this.startTime = System.currentTimeMillis();

        Point next = new Point();

        processTree(boardMap, true, 0, roundLeft * 2, -999,
999, next);
        return new int[] {next.x, next.y};
```

```
    }
```

7. Fungsi *processTree*

Proses	Melakukan algoritma *minimax* dengan *alpha beta pruning* untuk menentukan gerakan terbaik
Keluaran	int

```
public int processTree(char[][] boardMap, boolean isBot, int
depth, int leftround, int alpha, int beta, Point
selectedPoint) {
        // if processing time > 5s, then terminate program
        if(System.currentTimeMillis() - this.startTime
>5000){
            return calculateObjective(boardMap);
        }
        if (leftround > 0 && depth <= this.maxDepth) {
            // generate successor
            ArrayList<PointValue> pointValues =
searchAdjacent(boardMap, isBot);
            int maxIteration = (int)
Math.ceil(pointValues.size() * this.maxWidth);
            for(int z = 0; z < maxIteration;z++ ){
                PointValue pointValue = pointValues.get(z);
                char [][] state = new char[8][];
                for(int i = 0; i < 8; i++) {
                    state[i] = boardMap[i].clone();
                }
                changeState(state, isBot,
pointValue.getPoint().x, pointValue.getPoint().y);
                Point nextPath = new
Point(pointValue.getPoint());
                int val = processTree(state, !isBot, depth+1,
leftround-1, alpha, beta, nextPath);
                if (isBot && val > alpha) {
                    alpha = val;
                    if (depth == 0) {
```

```
selectedPoint.setLocation(pointValue.getPoint().x,pointValue.
getPoint().y);
                    }

                } else if (!isBot && val < beta) {
                    beta = val;
                    if (depth == 0) {

selectedPoint.setLocation(pointValue.getPoint().x,pointValue.
getPoint().y);
                    }
                }
            }
            return isBot ? alpha : beta;

        } else {
            return calculateObjective(boardMap);
        }
    }
```

8. Fungsi *changeState*

Proses	Memperbaharui kondisi *state* permainan setelah sebuah gerakan dilakukan
Keluaran	void

```
private void changeState(char[][] state, boolean isBot, int
x, int y) {
        state[x][y] = isBot ? this.selfMark : this.enemyMark;
        if(isValid(x-1,y)){
            state[x-1][y] = isBot && state[x-1][y] ==
this.enemyMark ? this.selfMark : !isBot && state[x-1][y] ==
this.selfMark ?this.enemyMark : state[x-1][y];
        }

        if(isValid(x,y-1)){
            state[x][y-1] = isBot && state[x][y-1] ==
this.enemyMark ? this.selfMark : !isBot && state[x][y-1] ==
```

```
this.selfMark ? this.enemyMark : state[x][y-1];
        }

        if(isValid(x+1,y)){
            state[x+1][y] = isBot && state[x+1][y] ==
this.enemyMark ? this.selfMark : !isBot && state[x+1][y] ==
this.selfMark ? this.enemyMark : state[x+1][y];
        }

        if(isValid(x,y+1)){
            state[x][y+1] = isBot && state[x][y+1] ==
this.enemyMark ? this.selfMark : !isBot && state[x][y+1] ==
this.selfMark ? this.enemyMark : state[x][y+1];
        }
    }
```

III. Pengembangan Agent Menggunakan Algoritma *Local Search*

1. Deskripsi Algoritma

Algoritma *local search* yang digunakan dalam pemenuhan Tugas Besar ini adalah algoritma *simulated annealing* dan *hill climbing with sideways move*. Kedua algoritma ini digunakan oleh penulis agar dapat dibandingkan performa antara keduanya, baik dari segi waktu maupun performa.

Algoritma *hill climbing with sideways move* dapat digunakan untuk mengatasi permasalahan yang muncul pada saat menggunakan algoritma *hill climbing steepest accent,* yaitu menghindari *stuck* pada *local optimum* dengan cara memperbolehkan munculnya *flat state.* Penerapan hal ini dilakukan dengan cara menghentikan proses pencarian hanya ketika *value* tertinggi dari seluruh *successor state,* atau biasa disebut dengan *neighbor state*, memiliki nilai *objective* yang lebih kecil dari *value* pada *current state.*

Pada awal proses pencarian, seluruh petak kosong akan dimasukkan ke dalam sebuah array dari petak, dengan tipe bentukan Array of (Array of int). Untuk setiap petak yang berada pada array tersebut akan dilakukan uji coba untuk dilihat nilai objektif yang dihasilkan bila petak tersebut dimasukkan ke dalam papan permainan. Selama pencarian, dilakukan juga uji terhadap *heuristics* yang telah ditetapkan. Petak yang menghasilkan nilai objektif tertinggi (*max value*) akan disimpan nilainya. Jika terjadi penemuan petak baru yang bernilai lebih tinggi atau sama dengan nilai petak tertinggi (*max value*), maka nilai petak tersebut akan menggantikan nilai *max value*. Bila seluruh iterasi telah selesai dijalankan, petak yang menghasilkan nilai *max value* akan dikembalikan sebagai solusi.

Algoritma berikutnya merupakan algoritma *simulated annealing*. Algoritma tersebut dipilih mengingat *simulated annealing* memiliki aspek *randomness* yang membuat algoritma ini dapat keluar dari *local maximum* dalam rangka mencari *global maximum*. Namun, alih-alih menggunakan *stochastic hill climbing* yang sepenuhnya mengandalkan aspek *randomness*, algoritma *simulated annealing*, masih mempertimbangkan perbandingan nilai objektif dari suatu *state* dengan *successor state*.

Algoritma *simulated annealing* dimulai dengan menentukan sebuah nilai variabel temperatur yang menjadi batasan bagi iterasi yang akan dilakukan. Nilai temperatur lantas akan terus menurun hingga iterasi dihentikan ketika variabel temperatur bernilai 0. Untuk setiap iterasi, dibangkitkan sebuah *random successor* dari petak kosong yang tertera pada papan permainan. Nilai objektif dari *random successor* lantas diujicobakan dan dibandingkan dengan nilai objektif dari *current state*. Seperti halnya algoritma *sideways move*, nilai *heuristics* juga dipertimbangkan untuk setiap *random successor* yang dibangkitkan. Setelah itu, nilai probabilitas untuk terjadi perpindahan akan dihasilkan berdasarkan nilai objektif dari *random successor*, nilai objektif dari *current state*, dan nilai variabel *temperatur* sebagai berikut:

1. Bila nilai objektif dari *random successor* (*new value*) lebih besar dibandingkan nilai objektif dari *current state* (*current value)*, maka mengembalikan 1 sebagai nilai probabilitas.
2. Bila nilai objektif dari *random successor* (*new value*) lebih kecil atau sama dengan nilai objektif dari *current state* (*current value)*, maka nilai probabilitasnya adalah

$$p = e^{\Delta E/t}$$, dengan

p : nilai probabilitas

e : bilangan *euler*

ΔE : selisih nilai *new value* dengan *current value*

t : nilai variabel temperatur

Setelah mendapatkan nilai probabilitas, akan dibangkitkan angka *random* yang berguna sebagai penanda perpindahan. Bila angka *random* tersebut bernilai lebih kecil dibanding nilai probabilitas, maka perpindahan akan terjadi dan nilai *new value* akan menggantikan nilai *current value*. Nilai *current value* adalah nilai yang dikembalikan ketika iterasi selesai dilakukan.

2. Fungsi Heuristik

Terdapat dua heuristik yang dipakai pada algoritma *local search* yang diterapkan, yaitu penghitungan adanya *mark* lawan pada petak yang bersebelahan dan penghitungan adanya potensi terjadinya perebutan kembali oleh lawan. Untuk menjelaskan *heuristics* kali ini, akan digunakan kondisi papan permainan sebagai berikut,

						O	O
						O	O
		O	O				
	X	O					
X							
O		X					
O	O	O					
X	O	X	X				

dengan pemain X sebagai pemain yang sedang melakukan pergerakan.

1. Penghitungan adanya *mark* lawan pada petak yang bersebelahan

   Dalam *adjacency strategy game*, tujuan utama dari pemain adalah untuk mendapatkan jumlah *mark* sebanyak-banyaknya, termasuk *mark* milik lawan. Maka dari itu, salah satu strategi yang seharusnya ditentukan adalah memastikan bahwa setiap gerakan dari *bot* adalah gerakan yang menguntungkan, yaitu mengambil petak lawan. Maka dari itu, ketika *successor* yang ditemukan adalah *successor* yang tidak memiliki *mark* lawan pada petak yang bersebelahan, *successor* tersebut dapat diabaikan.

						O	O
						O	O
		O	O				
	X	O					
X							
O	X	X					
O	O	O					
X	O	X	X				

						O	O
						O	O
		O	O				
	X	O					
X			X				
O		X					
O	O	O					
X	O	X	X				

2. Penghitungan adanya potensi perebutan kembali oleh lawan

   Ketika sebuah *mark* ditaruh pada papan permainan, hal tersebut membuka kemungkinan bagi lawan untuk mengambil *mark* lebih banyak

lagi. Seperti contohnya dalam hal ini, skenario urutan yang mungkin terjadi adalah sebagai berikut:

						O	O
						O	O
		O	O				
	X	O					
X							
O		X					
O	O	O					
X	O	X	X				

						O	O
						O	O
		O	O				
	X	O					
X							
X	X	X					
O	X	O					
X	O	X	X				

						O	O
						O	O
		O	O				
	O	O					
O	O						
X	O	X					
O	X	O					
X	O	X	X				

Sesaat pemain X menaruh *mark* nya, pemain O dapat berpotensi mengambil kembali *mark* tersebut karena bertetanggaan dengan lebih banyak *mark* X. Dengan kata lain, *mark* yang diletakkan oleh pemain X menjadi jembatan bagi pemain O untuk mendapatkan lebih banyak *mark*. Maka dari itu, dilakukan pengecekan *heuristics* bila terdapat petak tetangga dari petak tetangga yang memiliki *mark* yang sama dengan pemain, maka petak tersebut akan memiliki prioritas yang lebih rendah. Peletakan *mark* akan dilakukan pada posisi yang lebih tidak membahayakan.

						O	O
						O	O
		O	O				
	X	O					
X							
O	X	X					
O	O	O					
X	O	X	X				

						O	O
						O	O
		O	O				
	X	O	X				
X							
O		X					
O	O	O					
X	O	X	X				

3. Cuplikan Kode

1. Fungsi *calculateObjective*

Proses	Menghitung nilai objektif dari suatu *state* permainan
Keluaran	int

```
public int calculateObjective(char[][] boardMap){
```

```
    int point = 0;
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (boardMap[i][j] == this.selfMark) {
                point++;
            } else if (boardMap[i][j] == this.enemyMark) {
                point--;
            }
        }
    }
    return point;
}
```

2. Fungsi *duplicateBoardAndInsert*

Proses	Menduplikasi papan permainan dan memasukkan suatu *mark* pada petak tertentu
Keluaran	Matrix of char

```
private char[][] duplicateBoardAndInsert(char[][] boardMap,
int[] pos){
    // Duplicate
    char[][] newMap = new char[8][8];
    for (int i = 0; i < 8; i++){
        for(int j = 0; j < 8; j++){
            newMap[i][j] = boardMap[i][j];
        }
    }

    // Insert
    newMap[pos[0]][pos[1]] = this.selfMark;
    if (pos[0] != 0 && newMap[pos[0]-1][pos[1]] ==
this.enemyMark){
        newMap[pos[0]-1][pos[1]] = this.selfMark;
    }
    if (pos[0] != 7 && newMap[pos[0]+1][pos[1]] ==
this.enemyMark){
        newMap[pos[0]+1][pos[1]] = this.selfMark;
```

```
    }
    if (pos[1] != 0 && newMap[pos[0]][pos[1]-1] ==
this.enemyMark){
        newMap[pos[0]][pos[1]-1] = this.selfMark;
    }
    if (pos[1] != 7 && newMap[pos[0]][pos[1]+1] ==
this.enemyMark){
        newMap[pos[0]][pos[1]+1] = this.selfMark;
    }
    return newMap;
}
```

3. Fungsi *generateRandom*

Proses	Membangkitkan *successor* secara random untuk algoritma *Simulated Annealing*
Keluaran	Array of int

```
private int[] generateRandom(char[][] boardMap){
    int[] currentMove = new int[]{(int) (Math.random()*8),
(int) (Math.random()*8)};
    while (boardMap[currentMove[0]][currentMove[1]] != ' '){
        currentMove = new int[]{(int) (Math.random()*8),
(int) (Math.random()*8)};
    }
    return currentMove;
}
```

4. Fungsi *moveProbability*

Proses	Menghitung probabilitas dari suatu successor untuk berpindah
Keluaran	double

```
private double moveProbability(double valueDiff, double t){
    if (valueDiff > 1){
        return 1;
    } else {
```

```
        return Math.exp( (double) (valueDiff-1)/  t);
    }
}
```

5. Fungsi *moveSuccess*

Proses	Menghitung apakah perpindahan berhasil terjadi berdasarkan nilai probabilitasnya
Keluaran	boolean

```
private boolean moveSuccess(double probability){
    Random rd = new Random();
    double res = rd.nextDouble();
    return res < probability;
}
```

6. Fungsi *countAdjacentOfMark*

Proses	Menghitung banyaknya tetangga dengan *mark* tertentu pada suatu petak
Keluaran	

```
private int countAdjacentOfMark(char[][] boardMap, int[] pos,
char mark){
    int count = 0;
    if (pos[0] != 0 && boardMap[pos[0]-1][pos[1]] == mark){
        count++;
    }
    if (pos[0] != 7 && boardMap[pos[0]+1][pos[1]] == mark){
        count++;
    }
    if (pos[1] != 0 && boardMap[pos[0]][pos[1]-1] == mark){
        count++;
    }
    if (pos[1] != 7 && boardMap[pos[0]][pos[1]+1] == mark){
        count++;
    }
```

```
    return count;
}
```

7. Fungsi *potentialTakenHeuristics*

Proses	Menghitung aspek *heuristics* untuk terjadinya pengambilan petak kembali oleh lawan
Keluaran	boolean

```
private boolean potentialTakenHeuristics(char[][] boardMap,
int[] tile){
    int count = 0;
    if (tile[0] != 0 && boardMap[tile[0]-1][tile[1]] == ' '){
        count += countAdjacentOfMark(boardMap, new int[]
{tile[0]-1, tile[1]},this.selfMark);
    }
    if (tile[0] != 7 && boardMap[tile[0]+1][tile[1]] == ' '){
        count += countAdjacentOfMark(boardMap, new int[]
{tile[0]+1, tile[1]},this.selfMark);
    }
    if (tile[1] != 0 && boardMap[tile[0]][tile[1]-1] == ' '){
        count += countAdjacentOfMark(boardMap, new int[]
{tile[0], tile[1]-1},this.selfMark);
    }
    if (tile[1] != 7 && boardMap[tile[0]][tile[1]+1] == ' '){
        count += countAdjacentOfMark(boardMap, new int[]
{tile[0], tile[1]+1},this.selfMark);
    }
    return count != 0;
}
```

8. Fungsi *getEmptyTiles*

Proses	Mengembalikan seluruh petak kosong pada *map* permainan
Keluaran	Array of (array of int)

```
private ArrayList<int[]> getEmptyTiles(char[][] boardMap){
```

```
    ArrayList<int[]> arr = new ArrayList<>();
    for (int i = 0; i < 8; i++){
        for (int j = 0; j < 8; j++){
            if (boardMap[i][j] == ' '){
                int[] tile = new int[2];
                tile[0] = i;
                tile[1] = j;
                arr.add(tile);
            }
        }
    }
    return arr;
}
```

9. Fungsi *move* (*Simulated Annealing*)

Proses	Menggerakan bot
Keluaran	Array of integer

```
public int[] move(char[][] boardMap, int roundLeft, char
selfMark, char enemyMark) {
    this.selfMark = selfMark;
    this.enemyMark = enemyMark;
    int[] current = null;
    double currentVal = calculateObjective(boardMap);
    double temperature = 10;
    while (temperature > 0){
        int[] newPos = generateRandom(boardMap);
        if (countAdjacentOfMark(boardMap, newPos,
this.enemyMark) > 0){
            char[][] newBoardMap =
duplicateBoardAndInsert(boardMap, newPos);
            double newVal = calculateObjective(newBoardMap);

            if (!potentialTakenHeuristics(boardMap, newPos))
{
                newVal +=0.5;
            }
```

```
            double prob = moveProbability(newVal-currentVal,
temperature);

            if (moveSuccess(prob)){
                System.out.println("SA construct: "+
newPos[0] + " " + newPos[1] + ", val: " + newVal);
                current = newPos;
                currentVal = newVal;
            }
        }
        temperature -= 0.1;
    }
    return current;
}
```

10. Fungsi *move* (*Sideways Move*)

Proses	Menggerakan bot
Keluaran	Array of integer

```
public int[] move(char[][] boardMap, int roundLeft, char
selfMark, char enemyMark) {
    this.selfMark = selfMark;
    this.enemyMark = enemyMark;

    ArrayList<int[]> emptyArr = getEmptyTiles(boardMap);

    double maxVal = calculateObjective(boardMap);
    int[] resTile = null;
    for (int[] tile: emptyArr) {
        if (countAdjacentOfMark(boardMap, tile,
this.enemyMark) > 0){
            char[][] newBoard =
duplicateBoardAndInsert(boardMap, tile);
            double currentVal = calculateObjective(newBoard);

            if (!potentialTakenHeuristics(boardMap, tile)){
                currentVal += 0.5;
            }
```

```
            if (currentVal >= maxVal){
                System.out.println("SM construct: " + tile[0]
+ " " + tile[1] + ", Val: " + currentVal);
                resTile = tile;
                maxVal = currentVal;
            }
        }
    }
    return resTile;
}
```

4. Perbandingan Algoritma

Algoritma *sideways move* dan algoritma *simulated annealing* akan diuji satu sama lain untuk membuktikan algoritma mana yang lebih optimal dalam menyelesaikan permasalahan *adjacency strategy game*. Pengujian akan dilakukan sebanyak 3 kali, dengan algoritma *sideways move* sebagai pemain X dan algoritma *simulated annealing* sebagai pemain O. Pemain yang mendapatkan giliran pertama akan diberikan secara bergantian, dimulai dari pemain O. Setiap pengujian akan dilakukan untuk permainan dengan jumlah secara berurutan 8 ronde,13 ronde, 18 ronde, 23 ronde, dan 28 ronde.

Uji	*Mark*	Pemain	Gerak Pertama	Jumlah Ronde	Pemenang
1	X	Sideways Move		8	Sideways Move
	O	Simulated Annealing	✔		
2	X	Sideways Move	✔	13	Simulated Annealing
	O	Simulated Annealing			
3	X	Sideways Move		18	Simulated Annealing
	O	Simulated Annealing	✔		

4	X	Sideways Move	✔	23	Sideways Move
	O	Simulated Annealing			
5	X	Sideways Move		28	Sideways Move
	O	Simulated Annealing	✔		

Dalam hal ini ditemukan fakta bahwa algoritma *sideways move* memenangkan pertandingan dibandingkan algoritma *simulated annealing* dengan skor 3-2. Untuk setiap pengujian yang dilakukan, program dijalankan 3 kali dan pemenang didapatkan dari pemenang terbanyak. Maka dari itu, dapat disimpulkan bahwa algoritma *sideways move* lebih unggul dibandingkan algoritma *simulated annealing*.

Beberapa faktor yang mungkin menyebabkan hal ini terjadi dikarenakan algoritma *simulated annealing* mengandung faktor *randomness* yang tidak bisa diprediksi. Di sisi lain algoritma *sideways move* memiliki hasil yang bersifat deterministik.

IV. Pengembangan *Agent* Menggunakan *Genetic Algorithm*

1. Deskripsi Algoritma

Algoritma genetik atau *genetic algorithm* adalah algoritma yang terinspirasi dari teori Charles Darwin mengenai seleksi natural. Algoritma ini menggambarkan proses dari evolusi natural dimana hanya individu yang paling baik (*fittest individual*) yang akan dipilih untuk melakukan reproduksi dan menghasilkan generasi yang memiliki nilai lebih baik dibandingkan dengan generasi sebelumnya.

Algoritma genetik terdiri atas proses pembangkitan populasi awal, pemilihan parent, penyilangan gen, mutasi gen, dan evaluasi hasil persilangan. Representasi genetika yang digunakan adalah sebagai berikut,

1. Gen direpresentasikan sebagai koordinat *cell* yang masih kosong.

2. Kromosom direpresentasikan urutan pengisian *cell*. Panjang kromosom ditentukan oleh jumlah ronde yang tersisa.

Kromosom inilah yang menjadi properti utama dalam algoritma genetika ini. Kromosom yang dimaksud disini adalah untaian DNA. *Parent* yang dipilih dan digunakan dalam algoritma ini berbentuk seperti kromosom. *Parent* akan disilangkan dan dimutasi seperti layaknya kromosom pada makhluk hidup. Persilangan akan terus dilakukan hingga didapatkan generasi baru yang memiliki *value* yang baik (dalam hal ini memenangkan permainan).

Berikut merupakan langkah langkah yang dilakukan dalam pencarian generasi baru yang digunakan dalam tugas ini,

1. Program akan menginisialisasi 2 *parent*. Masing masing *parent* akan dibangkitkan secara acak dengan ketentuan kromosom yang dibangkitkan merupakan kromosom yang valid (tidak ada 2 gen sama dalam 1 kromosom).
2. Kedua parent akan disilangkan dengan *crossover point* yang ditentukan secara acak.
3. Mutasi akan dilakukan dengan pemilihan *mutation point* secara acak yang akan digantikan dengan gen baru. Mutasi ini tetap memperhatikan dan menjaga kromosom agar tidak rusak setelah dimutasi.
4. Setelah persilangan, mungkin saja kromosom menjadi rusak (terdapat 2 gen sama dalam 1 kromosom). Oleh karena itu, mutasi dilakukan untuk memperbaiki kromosom yang rusak ini dengan mengganti semua gen ganda dengan nilai baru yang dihasilkan secara acak.
5. Generasi baru yang didapat akan dihitung nilainya dengan menggunakan *objective function* yang digunakan.
6. Kromosom yang akan diambil adalah kromosom yang yang memiliki nilai yang paling tinggi.
7. Apabila kromosom terbaik memiliki value yang bernilai negatif (yang berarti bot kalah dalam permainan). Akan dilakukan iterasi ulang hingga mendapatkan hasil yang sesuai dengan maksimal iterasi 5 kali.

8. Setelah itu, *next step* yang akan diambil oleh bot adalah nilai pertama dari kromosom terbaik yang didapatkan dari hasil persilangan.

2. Fungsi Heuristik

Algoritma di atas masih memiliki kekurangan dimana *next step* yang diambil oleh bot cenderung acak. Oleh karena itu, heuristik tambahan diperlukan dimana akan dilakukan mutasi tambahan gen pertama pada kromosom dengan gen yang *adjacent* dengan *mark* lawan. Mutasi ini akan dilakukan setelah mutasi acak selesai. Heuristik ini bersifat opsional, dengan kata lain, apabila tidak terdapat lagi kandidat gen baru yang *adjacent* dengan *mark* lawan, heuristik ini tidak diterapkan untuk menjaga *randomness* dari untaian gen dibelakangnya.

3. Cuplikan Kode

1. Fungsi *initializeMap*

Proses	Digunakan untuk *copy* kondisi papan permainan pada saat awal - awal proses ke atribut *map*
Keluaran	void

```
public void initializeMap(char[][] boardMap) {
    this.map = new char[8][8];
    for (int i = 0; i < 8; i++) {
        System.arraycopy(boardMap[i], 0, this.map[i], 0, 8);
    }
}
```

2. Fungsi *getEmpty*

Proses	Mencari dan mengembalikan *point - point* yang masih kosong dalam papan permainan
Keluaran	Array of (Array of int)

```
public ArrayList<int[]> getEmpty() {
```

```
    ArrayList<int[]> point = new ArrayList<int[]>();
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (this.map[i][j] == ' ') {
                point.add(new int[] {i, j});
            }
        }
    }
    return point;
}
```

3. Fungsi *getParent*

Proses	Memilih *parent* yang akan digunakan untuk *crossover* dengan cara mengalikan bilangan *random* dengan ronde yang tersisa
Keluaran	Array of int

```
public int[] getParent(ArrayList<int[]> emptyCell, int
roundsLeft) {
    int[] parent = new int[roundsLeft];
    int length = emptyCell.size();

    ArrayList<Integer> idxleft = new ArrayList<>();
    for (int i = 0; i < length; i++) {
        idxleft.add(i);
    }
    for (int i = 0; i < roundsLeft; i++) {
        int idxselected = (int)
(Math.random()*idxleft.size());

        int selected = idxleft.get(idxselected);
        idxleft.remove(idxselected);
        parent[i] = selected;
    }
    return parent;
}
```

4. Fungsi *crossover*

Proses	Melakukan *crossover* antara *parent 1* dengan *parent 2*. *Crossover point* ditentukan dengan mengalikan bilangan *random* dengan jumlah ronde yang tersisa.
Keluaran	void

```
public void crossover(ArrayList<int[]> emptyCell, int
roundsLeft, int[] parent1, int[] parent2) {
    if (roundsLeft > 1) {
        int crossoverpoint = (int)
(Math.random()*roundsLeft);
        while (crossoverpoint == 0) {
            crossoverpoint = (int)
(Math.random()*roundsLeft);
        }

        ArrayList<Integer> buffer = new ArrayList<Integer>();

        for (int i = crossoverpoint; i < roundsLeft; i++) {
            buffer.add(parent1[i]);
            parent1[i] = parent2[i];
        }

        for (int i = crossoverpoint; i < roundsLeft; i++) {
            parent2[i] = buffer.get(i-crossoverpoint);
        }
    }
}
```

5. Fungsi *mutation*

Proses	Melakukan mutasi kepada rantai gen yang dihasilkan dengan memastikan integritas dari rantai gen tersebut (memenuhi heuristik dan tidak ada nilai yang sama pada rantai gen)

Keluaran	void

```
private void mutation(ArrayList<int[]> emptyCell, int
roundsLeft, int[] gene1, int[] gene2) {
    ArrayList<Integer> doublegene1 = searchDouble(gene1,
emptyCell.size());
    ArrayList<Integer> doublegene2 = searchDouble(gene2,
emptyCell.size());
    mutateGene(emptyCell.size(), gene1, doublegene1,
roundsLeft);
    mutateGene(emptyCell.size(), gene2, doublegene2,
roundsLeft);

    heuristicFirst(emptyCell, gene1);
    heuristicFirst(emptyCell, gene2);
}
```

6. Fungsi *searchDouble*

Proses	Mengembalikan index - index dalam *gene* yang memiliki *value* sama.
Keluaran	Array of int

```
private ArrayList<Integer> searchDouble(int[] gene, int
length) {
    ArrayList<Integer> valueDouble = new ArrayList<>();
    int[] dictionary = new int[length];
    for(int i = 0; i < length; i++) {
        dictionary[i] = -1;
    }

    for(int i = 0; i < gene.length; i++) {
        if (dictionary[gene[i]] == -1) {
            dictionary[gene[i]] = i;
        } else {
            valueDouble.add(i);
            if (valueDouble.indexOf(dictionary[gene[i]]) ==
-1) {
                valueDouble.add(dictionary[gene[i]]);
```

```
            }
        }
    }

    return valueDouble;
}
```

7. Fungsi *mutateGene*

Proses	Melakukan mutasi pada indeks - indeks tertentu. Indeks - indeks tempat dilakukannya *mutation* terdapat pada mutationpoint. *Value* yang dipilih untuk menggantikan *value* sebelumnya adalah angka *random*.
Keluaran	void

```
private void mutateGene(int length, int[] gene,
ArrayList<Integer> mutationpoint, int roundsLeft) {
    ArrayList<Integer> idxleft = new ArrayList<>();
    for (int i = 0; i < length; i++) {
        idxleft.add(i);
    }
    if (mutationpoint.isEmpty()) {
        int point = (int) (Math.random()*roundsLeft);
        System.out.println("Mutation point: " + point);
        int idxselected = (int)
(Math.random()*idxleft.size());
        int selected = idxleft.get(idxselected);
        idxleft.remove(idxselected);
        gene[point] = selected;
    } else {
        for (int i = 0; i < mutationpoint.size(); i++) {
            System.out.println("Mutation point: " +
mutationpoint.get(i));
            int idxselected = (int)
(Math.random()*idxleft.size());
            int selected = idxleft.get(idxselected);
            idxleft.remove(idxselected);
```

```
            gene[mutationpoint.get(i)] = selected;
        }
    }
}
```

8. Fungsi *heuristicsFirst*

Proses	Heuristik mutasi gen pertama pada kromosom dengan gen yang *adjacent* dengan mark lawan
Keluaran	void

```
private void heuristicFirst(ArrayList<int[]> emptyCell, int[]
gene) {
    ArrayList<Integer> point = new ArrayList<>();
    for (int i = 0; i< emptyCell.size(); i++) {
        if (isAdjacent(this.map, emptyCell.get(i)[0],
emptyCell.get(i)[1], true)) {
            boolean found = false;
            for (int j = 0; j < gene.length; j++) {
                if (gene[j] == i) {
                    found = true;
                    break;
                }
            }
            if (!found) {
                point.add(i);
            }
        }
    }

    if (!isAdjacent(this.map, emptyCell.get(gene[0])[0],
emptyCell.get(gene[0])[1], true) && !point.isEmpty()) {
        int idx = (int) (Math.random() * point.size());
        int selected = point.get(idx);
        gene[0] = selected;
    }
}
```

9. Fungsi *evaluate*

Proses	Melakukan evaluasi terhadap kondisi permainan dengan cara menghitung *objective value* nya.
Keluaran	int

```
private int evaluate(ArrayList<int[]> emptyCell, int[] gene)
{
    char [][] state = new char[8][8];
    for(int i = 0; i < 8; i++) {
        state[i] = this.map[i].clone();
    }

    boolean isBot = true;
    for (int idx : gene) {
        changeState(state, isBot, emptyCell.get(idx)[0],
emptyCell.get(idx)[1]);
        isBot = !isBot;
    }

    return calculateObjective(state);
}
```

10. Fungsi *changeState*

Proses	Memperbaharui kondisi *state* permainan setelah sebuah gerakan dilakukan
Keluaran	void

```
private void changeState(char[][] state, boolean isBot, int
x, int y) {
    state[x][y] = isBot ? this.selfMark : this.enemyMark;
    if(isValid(x-1,y)){
        state[x-1][y] = isBot && state[x-1][y] ==
this.enemyMark ? this.selfMark : !isBot && state[x-1][y] ==
this.selfMark ?this.enemyMark : state[x-1][y];
    }

    if(isValid(x,y-1)){
```

```
        state[x][y-1] = isBot && state[x][y-1] ==
this.enemyMark ? this.selfMark : !isBot && state[x][y-1] ==
this.selfMark ? this.enemyMark : state[x][y-1];
    }

    if(isValid(x+1,y)){
        state[x+1][y] = isBot && state[x+1][y] ==
this.enemyMark ? this.selfMark : !isBot && state[x+1][y] ==
this.selfMark ? this.enemyMark : state[x+1][y];
    }

    if(isValid(x,y+1)){
        state[x][y+1] = isBot && state[x][y+1] ==
this.enemyMark ? this.selfMark : !isBot && state[x][y+1] ==
this.selfMark ? this.enemyMark : state[x][y+1];
    }
}
```

11. Fungsi *isValid*

Proses	Memastikan apakah sebuah titik merupakan titik yang valid atau tidak dalam papan permainan
Keluaran	boolean

```
public boolean isValid(int x, int y){
    if(x >= 0 && x < 8 && y>= 0 && y < 8){
        return true;
    }
    return false;
}
```

12. Fungsi *isAdjacent*

Proses	Melakukan pengecekan apakah titik (x,y) mempunyai tetangga
Keluaran	boolean

```
public boolean isAdjacent(char[][] boardMap, int x, int y,
```

```
boolean isBot){
    if(isValid(x-1,y)){
        if (boardMap[x-1][y] == (isBot ? this.enemyMark :
this.selfMark)) {
            return true;
        }
    }

    if(isValid(x,y-1)){
        if (boardMap[x][y-1] == (isBot ? this.enemyMark :
this.selfMark)) {
            return true;
        }
    }

    if(isValid(x+1,y)){
        if (boardMap[x+1][y] == (isBot ? this.enemyMark :
this.selfMark)) {
            return true;
        }
    }

    if(isValid(x,y+1)){
        if (boardMap[x][y+1] == (isBot ? this.enemyMark :
this.selfMark)) {
            return true;
        }
    }
    return false;
};
```

13. Fungsi *calculateObjective*

Proses	Menghitung *objective value* dari sebuah *state*
Keluaran	int

```
public int calculateObjective(char[][] boardMap){
    int point = 0;
```

```
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (boardMap[i][j] == this.selfMark) {
                point++;
            } else if (boardMap[i][j] == this.enemyMark) {
                point--;
            }
        }
    }
    return point;
}
```

14. Fungsi *geneticAlgorithm*

Proses	Proses utama untuk melakukan *genetic algorithm.* Dalam fungsi ini, akan dilakukan proses pembangkitan *parent, crossover, mutation,* dan *evaluate* sampai memenuhi kondisi dimana urutan pada rantai gen mengakibatkan nilai objektif yang positif.
Keluaran	Array of int

```
public int[] geneticAlgorithm(ArrayList<int[]> emptyCell, int
roundsLeft) {
    int[] gene1, gene2, selectedgene = new int[0];
    boolean retry = true;
    int i = 0;
    while (retry) {
        gene1 = getParent(emptyCell, roundsLeft);
        gene2 = getParent(emptyCell, roundsLeft);
        printGeneStatus(emptyCell, roundsLeft, gene1, gene2);

        crossover(emptyCell, roundsLeft, gene1, gene2);
        System.out.println("After Crossover");
        printGeneStatus(emptyCell, roundsLeft, gene1, gene2);

        mutation(emptyCell, roundsLeft, gene1, gene2);
        System.out.println("After Mutation");
```

```
        printGeneStatus(emptyCell, roundsLeft, gene1, gene2);

        int point1 = evaluate(emptyCell, gene1);
        int point2 = evaluate(emptyCell, gene2);
        System.out.println("Point: " + point1 + " " +
point2);

        int maxpoint = Math.max(point1, point2);
        if (maxpoint < 0) {
            retry = true;
            i++;
            if (i == 10){
                selectedgene = point1 > point2 ? gene1 :
gene2;
                break;
            }
        } else {
            selectedgene = point1 > point2 ? gene1 : gene2;
            retry = false;
        }
    }

    return new int[] {emptyCell.get(selectedgene[0])[0],
emptyCell.get(selectedgene[0])[1]};
}
```

15. Fungsi *move*

Proses	Fungsi tempat *genetic algorithm* pertama kali diinisialisasi.
Keluaran	Array of int

```
public int[] move(char[][] boardMap, int roundLeft, char
selfMark, char enemyMark) {
    initializeMap(boardMap);
    ArrayList<int[]> emptyCell = this.getEmpty();
    this.selfMark = selfMark;
    this.enemyMark = enemyMark;
```

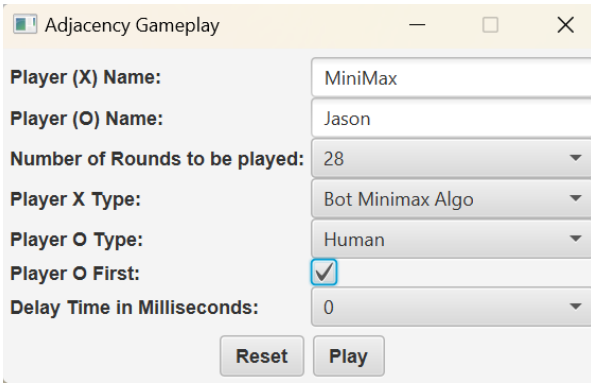
```
    return geneticAlgorithm(emptyCell, emptyCell.size() % 2
== 0 ? roundLeft * 2 : roundLeft * 2 - 1);
}
```

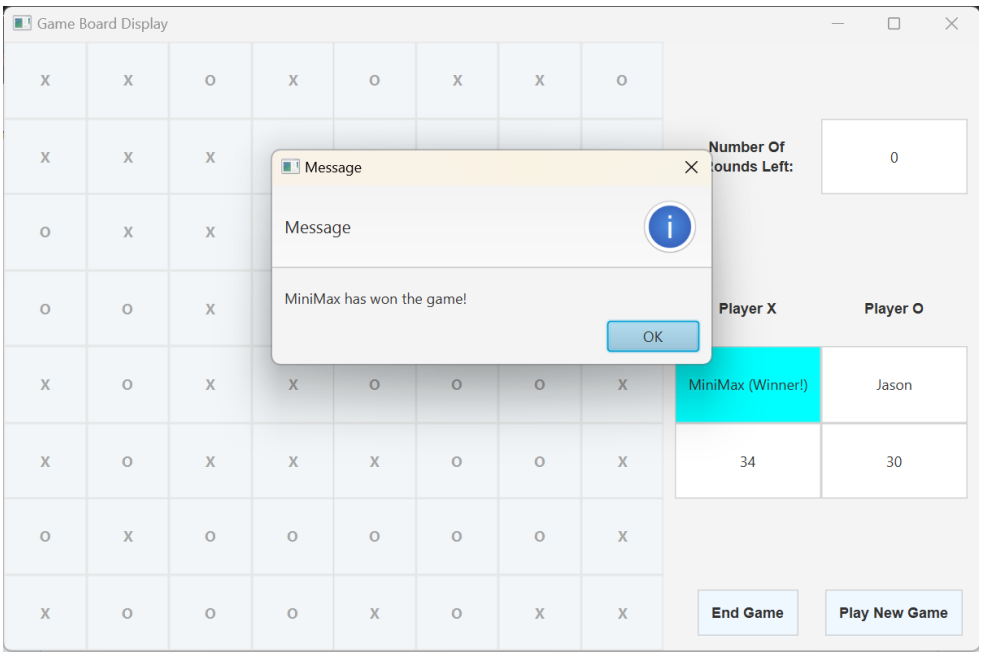
V. Pengujian

1. *Bot MiniMax* vs Manusia

1. Pengujian 1

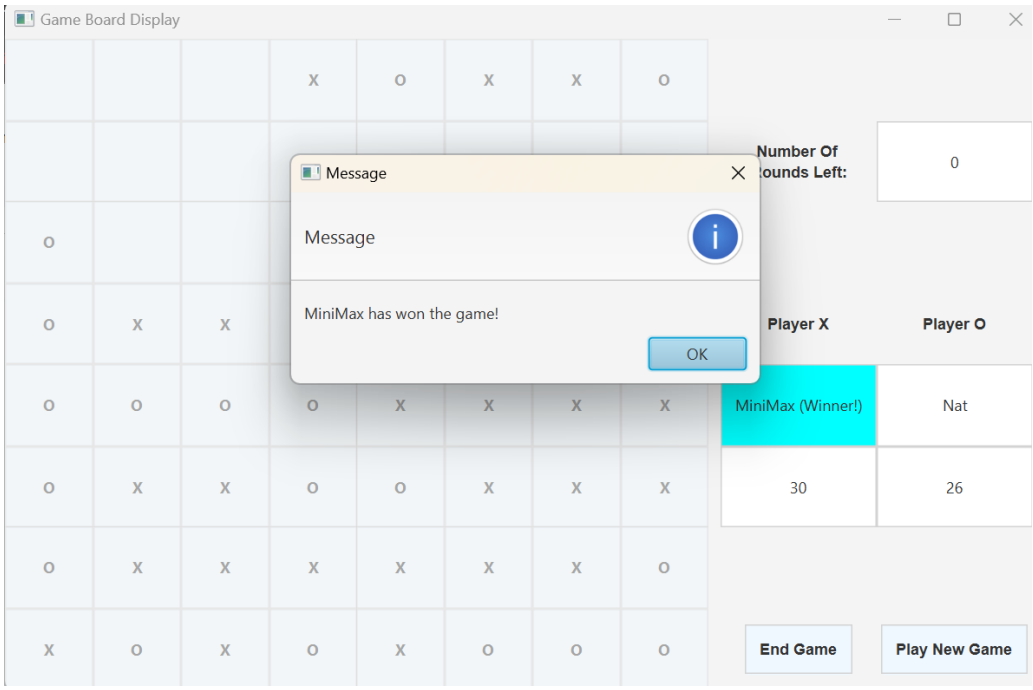
Player X	MiniMax	Player O	Jason
Jumlah ronde	28	Giliran Pertama	O
Pemenang		Minimax (X)	





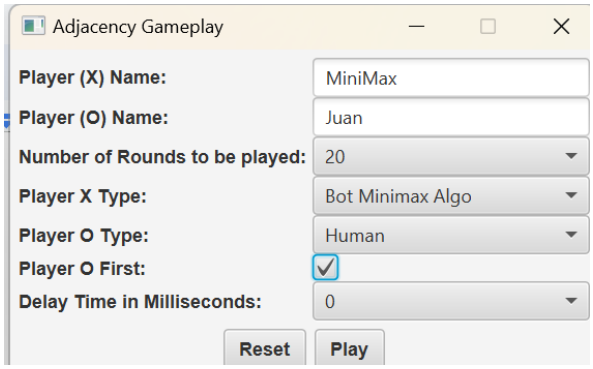
2. Pengujian 2

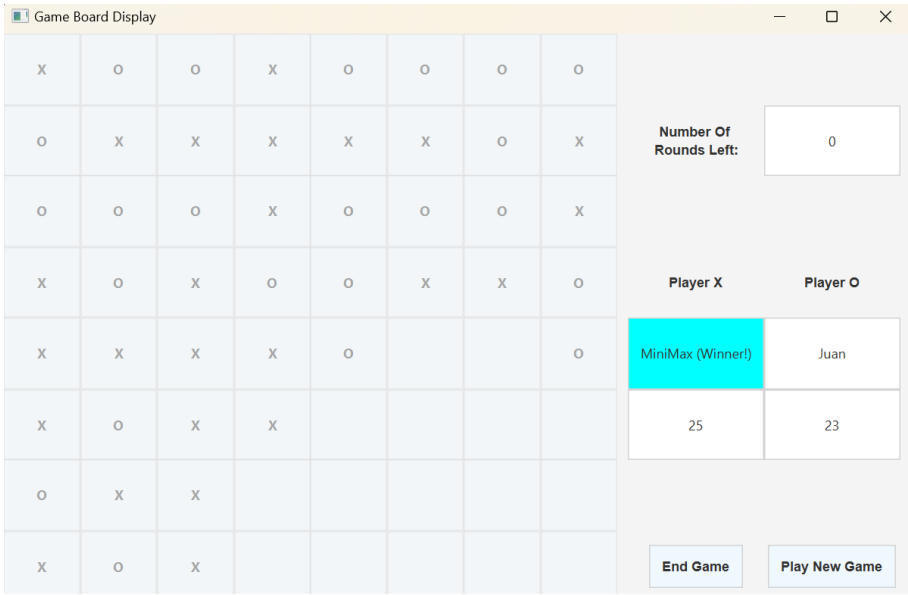
Player X	MiniMax	Player O	Nat
Jumlah ronde	24	Giliran Pertama	X
Pemenang		X	



3. Pengujian 3

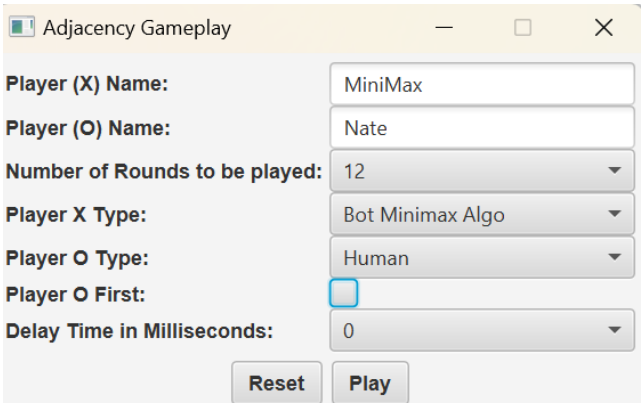
Player X	MiniMax	Player O	Juan
Jumlah ronde	20	Giliran Pertama	O
Pemenang		X	

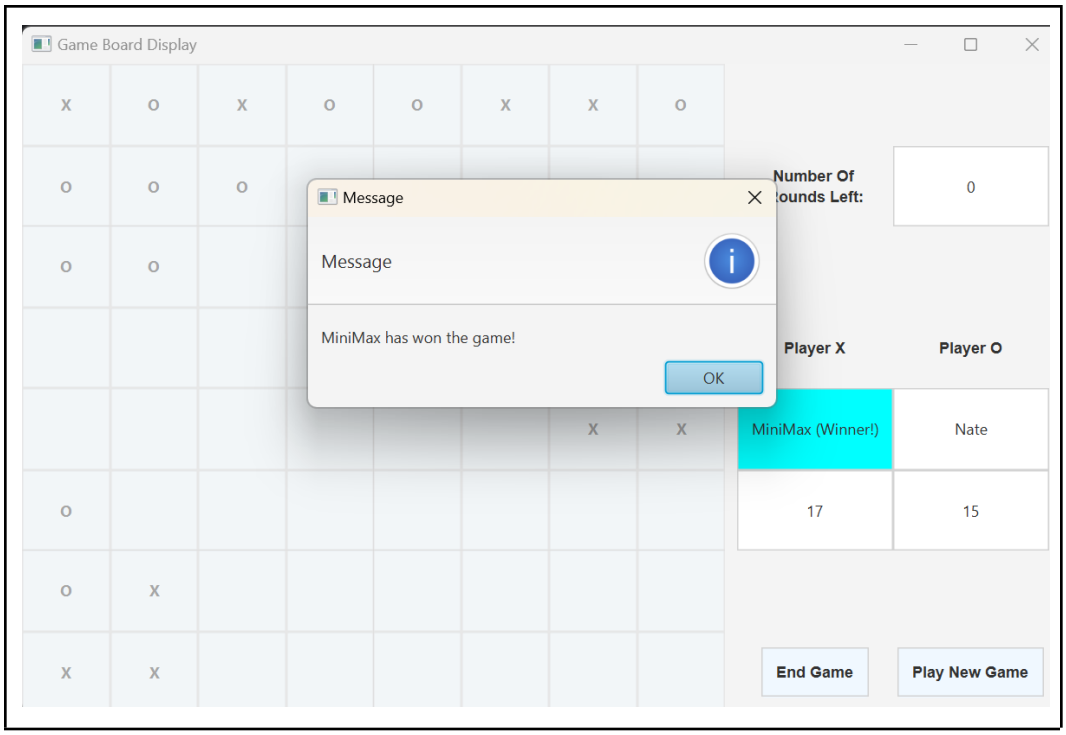




4. Pengujian 4

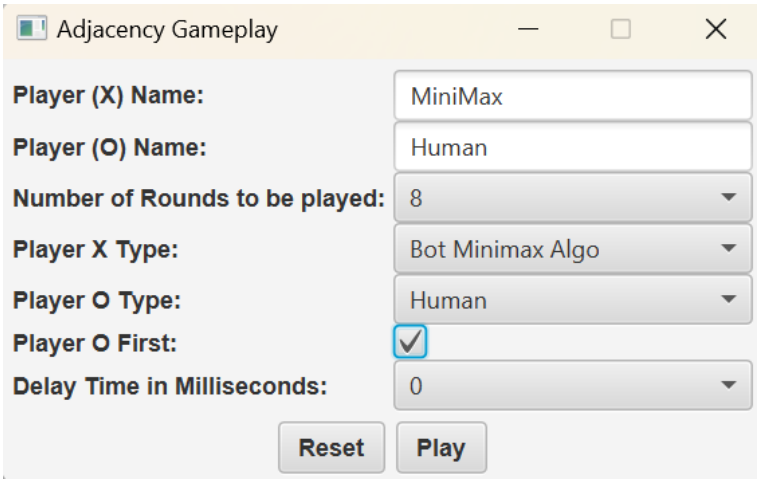
Player X	MiniMax	Player O	Nate
Jumlah ronde	12	Giliran Pertama	X
Pemenang		X	

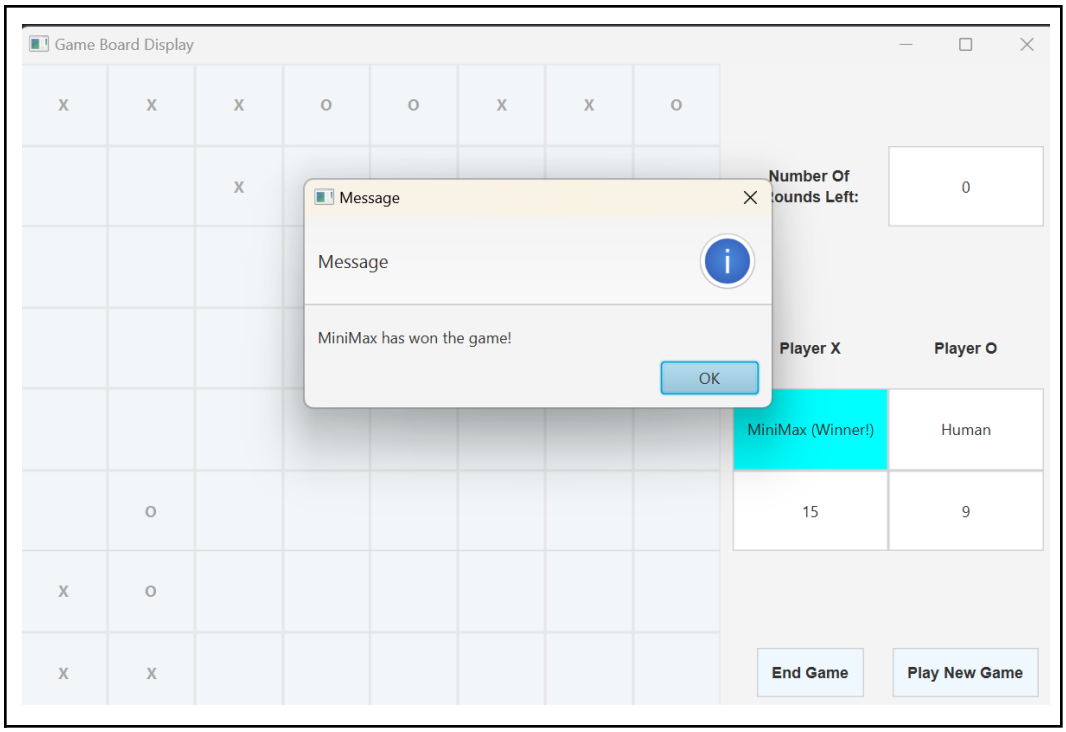




5. Pengujian 5

Player X	MiniMax	Player O	Human
Jumlah ronde	8	Giliran Pertama	O
Pemenang		X	

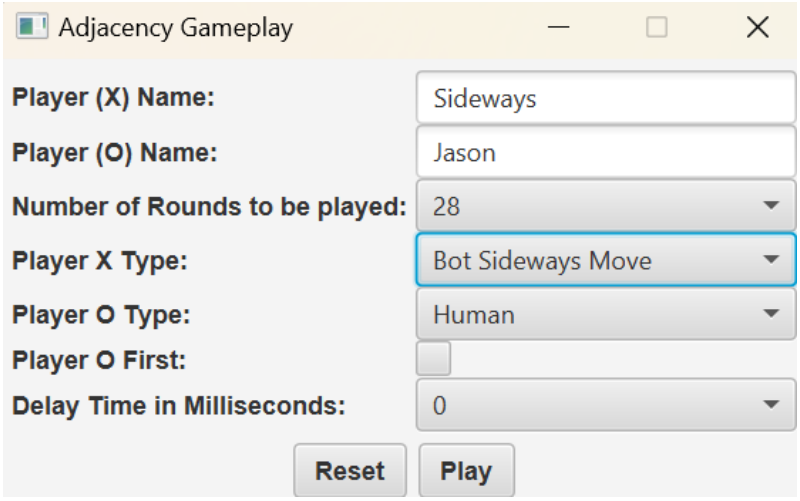


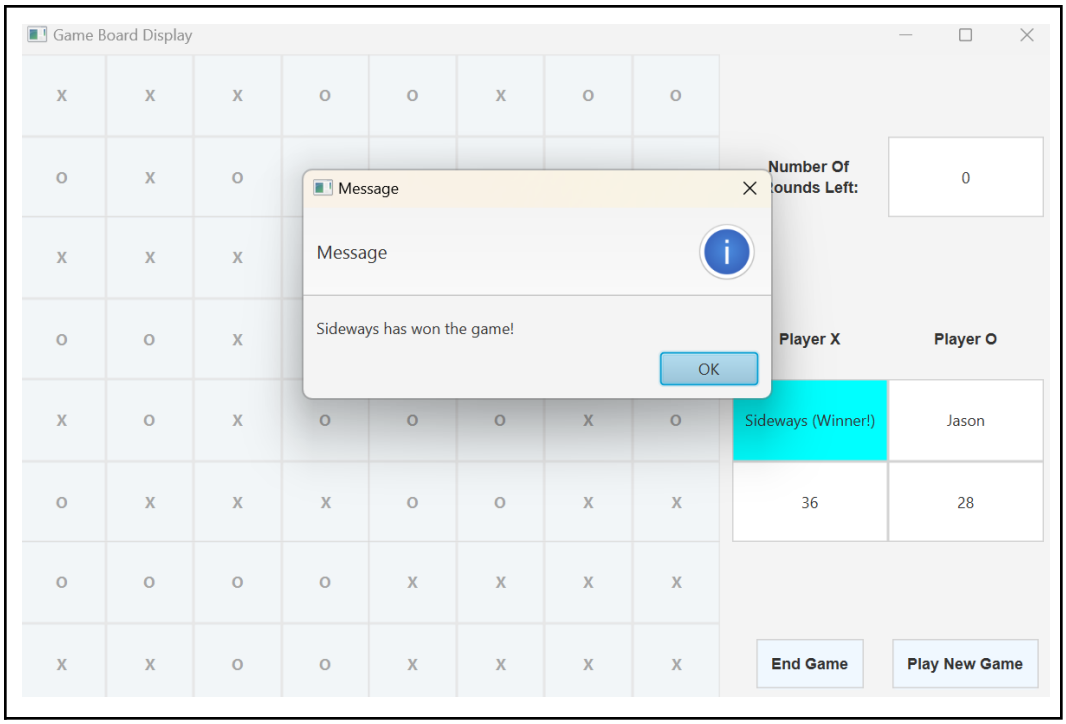


2. *Bot Local Search* vs Manusia

1. Pengujian 1

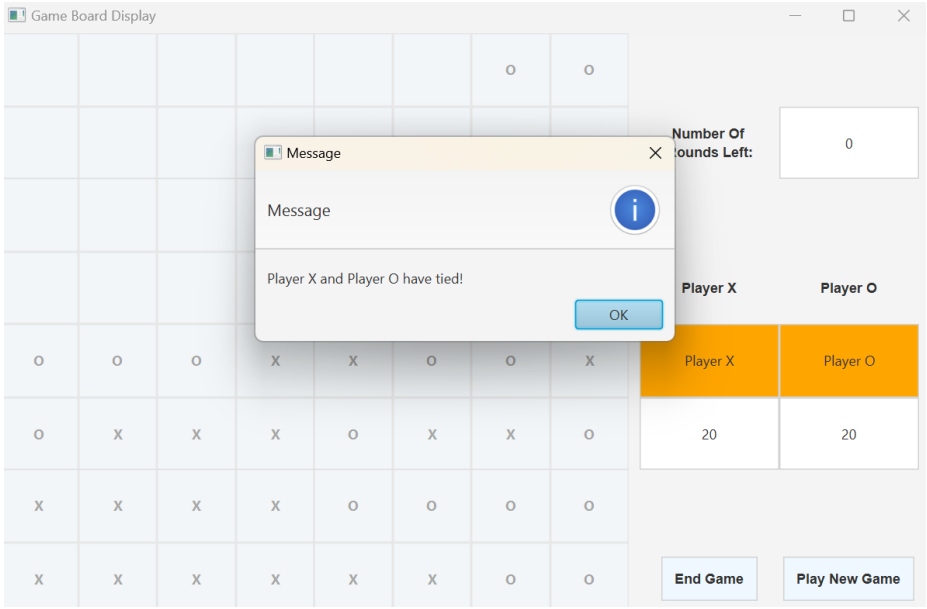
Player X	Sideways	Player O	Jason
Jumlah ronde	28	Giliran Pertama	O
Pemenang		X	





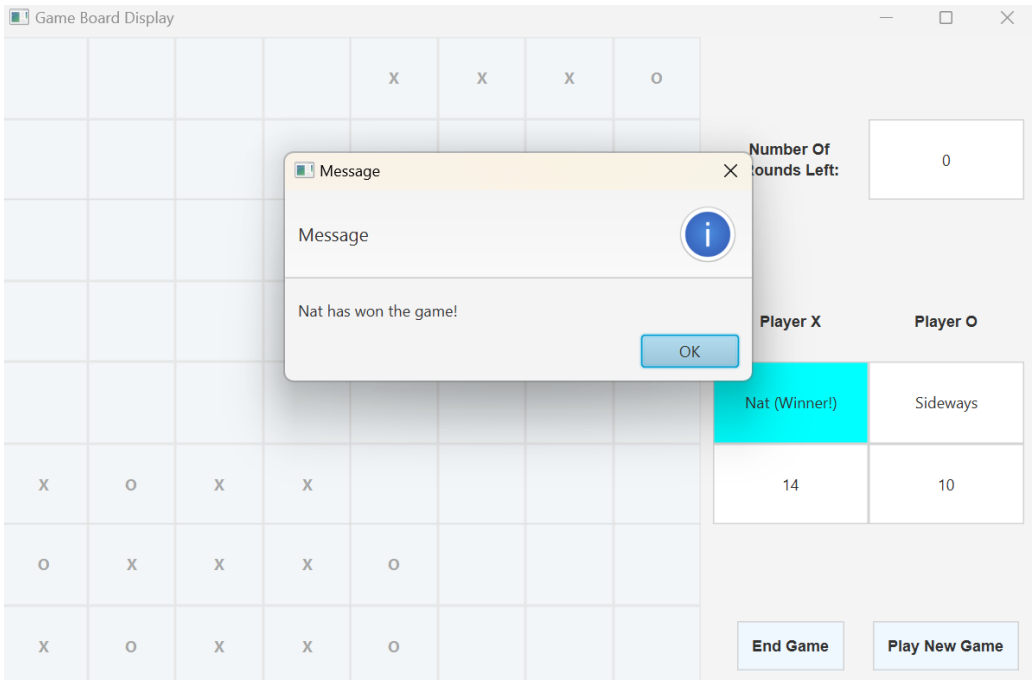
2. Pengujian 2

Player X	Sideways	Player O	Juan
Jumlah ronde	16	Giliran Pertama	O
Pemenang		Seri	



3. Pengujian 3

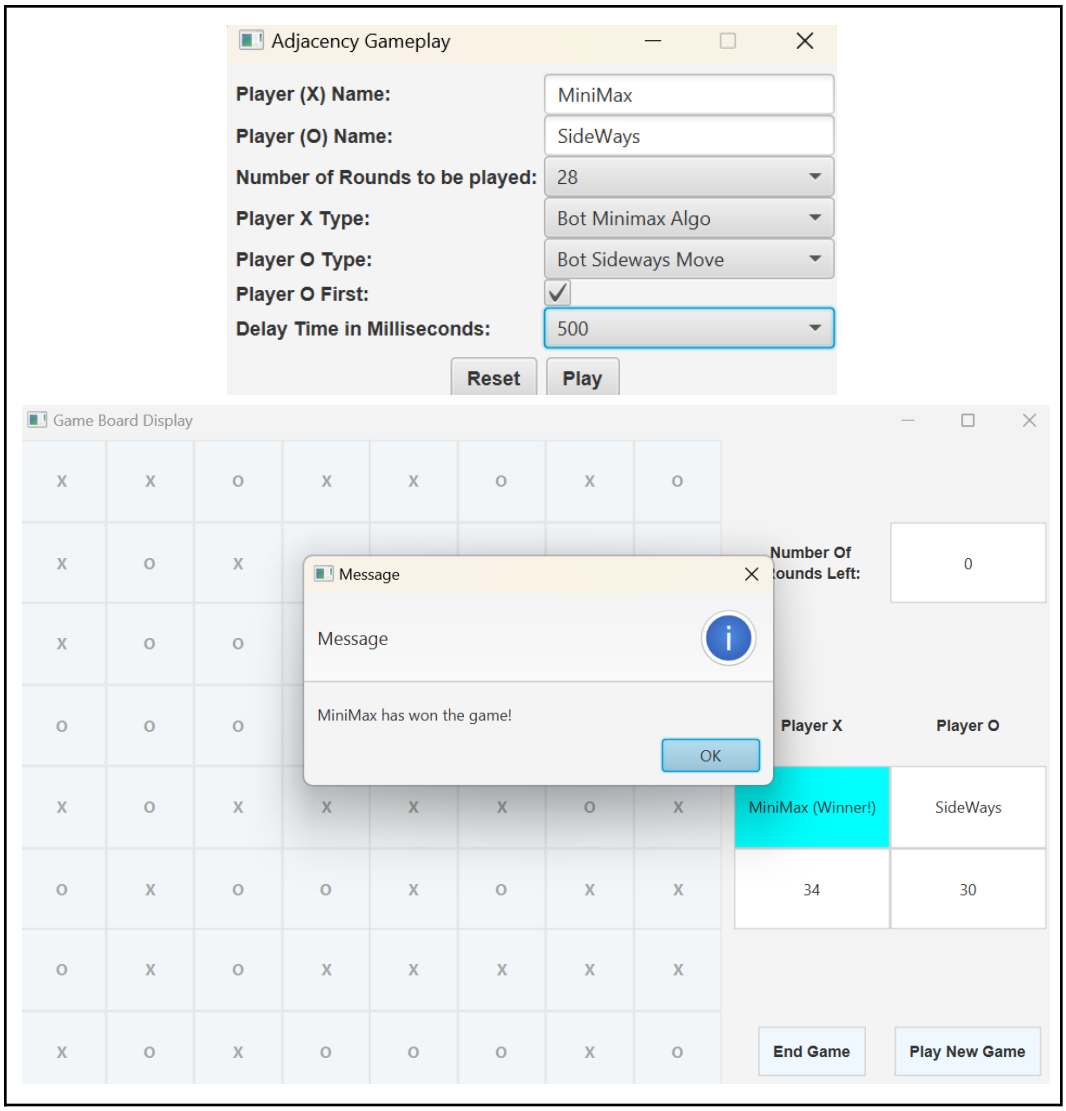
Player X	Nat	Player O	Sideways
Jumlah ronde	8	Giliran Pertama	O
Pemenang		X	



3. *Bot Minimax* vs *Bot Local Search*

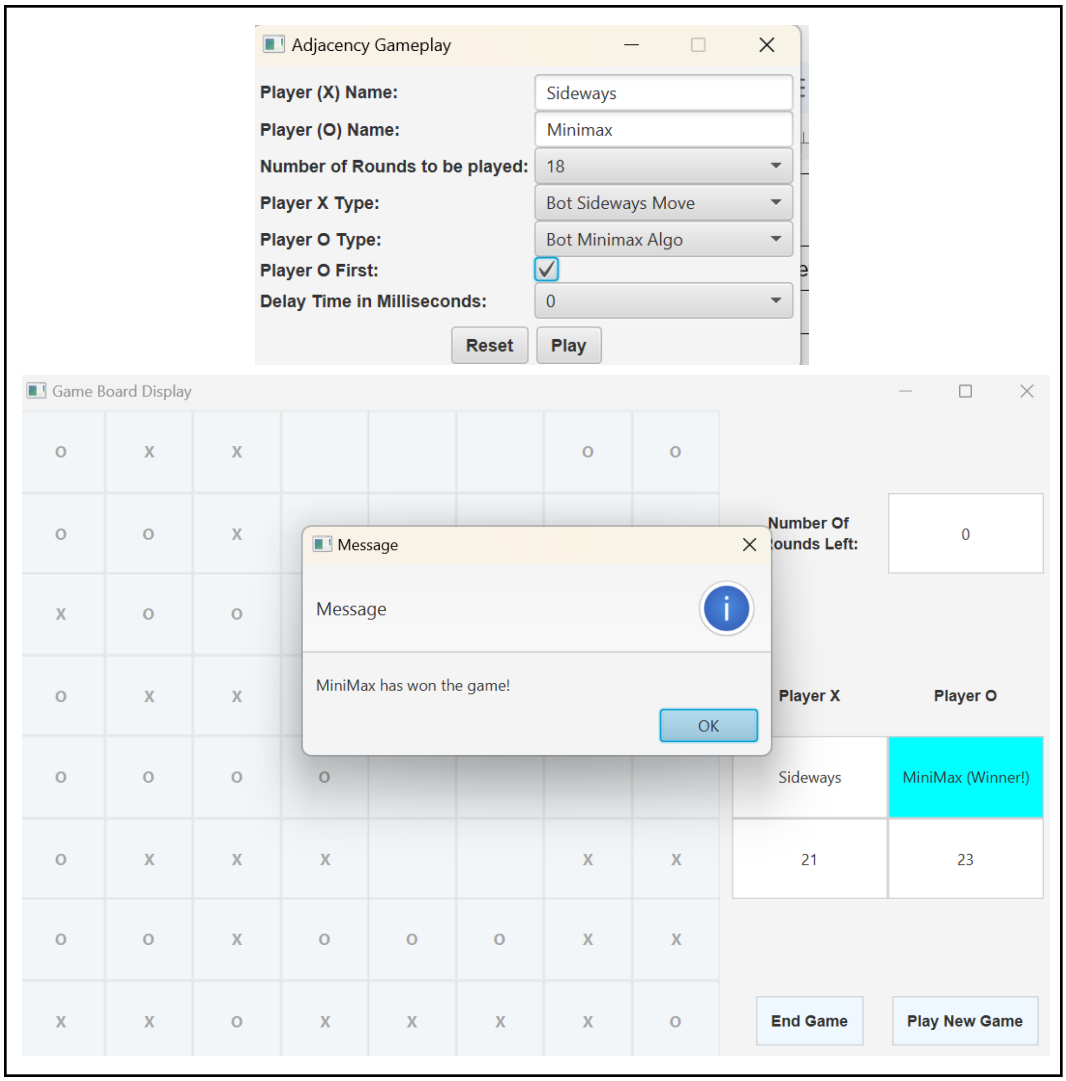
1. Pengujian 1

Player X	MiniMax	Player O	Sideways
Jumlah ronde	28	Giliran Pertama	O
Pemenang		X	



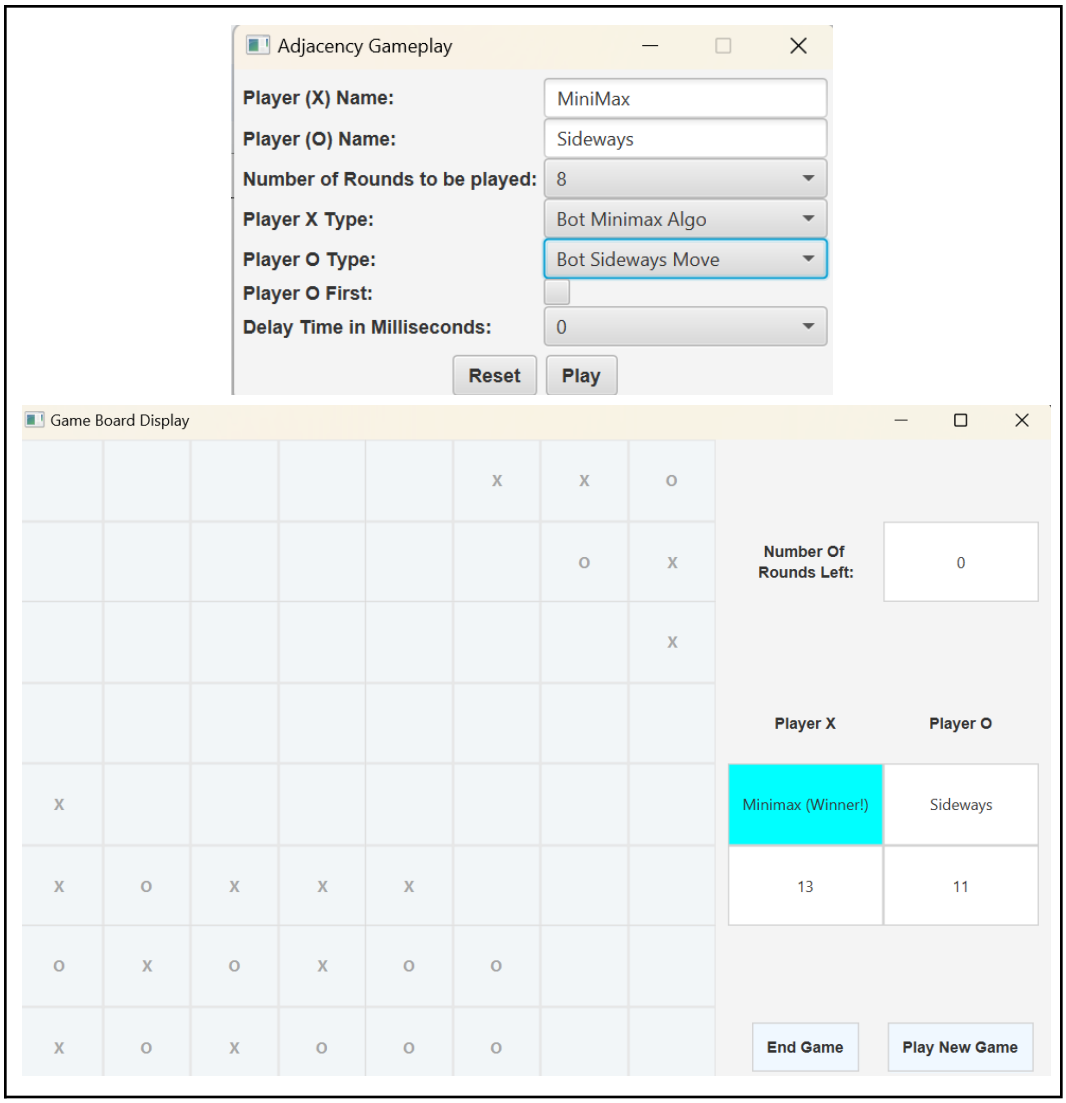
2. Pengujian 2

Player X	Sideways	Player O	MiniMax
Jumlah ronde	18	Giliran Pertama	O
Pemenang		X	



3. Pengujian 3

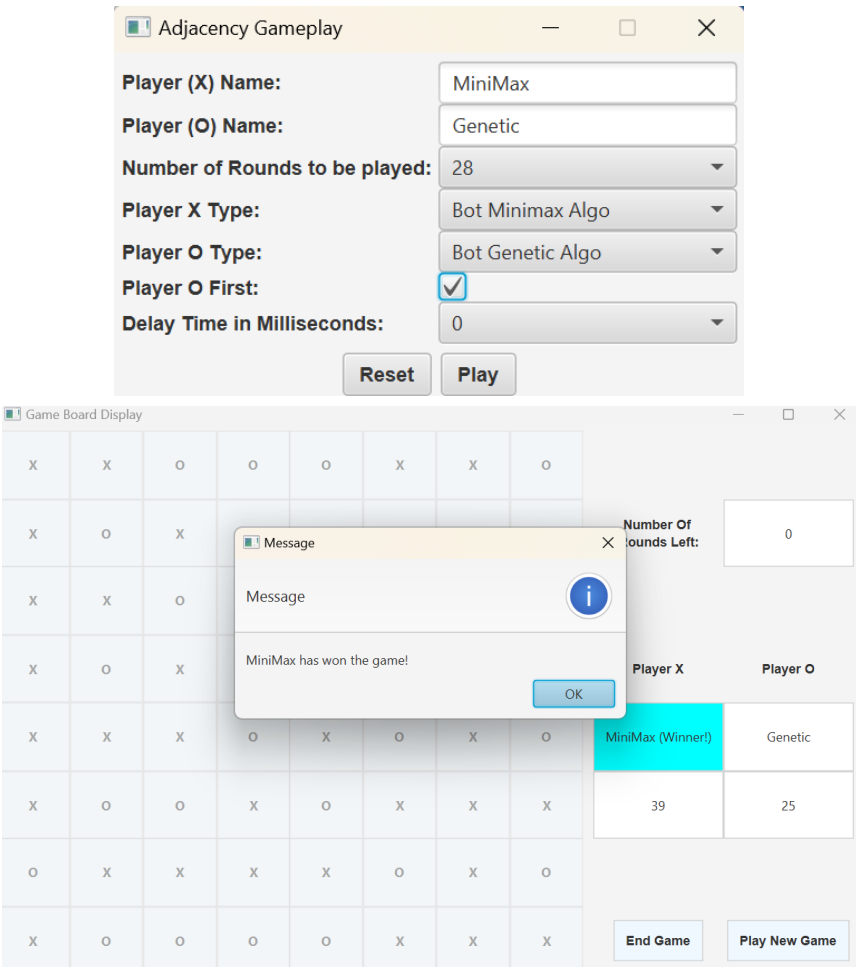
Player X	MiniMax	Player O	Sideways
Jumlah ronde	8	Giliran Pertama	X
Pemenang		X	



4. *Bot Minimax* vs *Bot Genetic Algorithm*

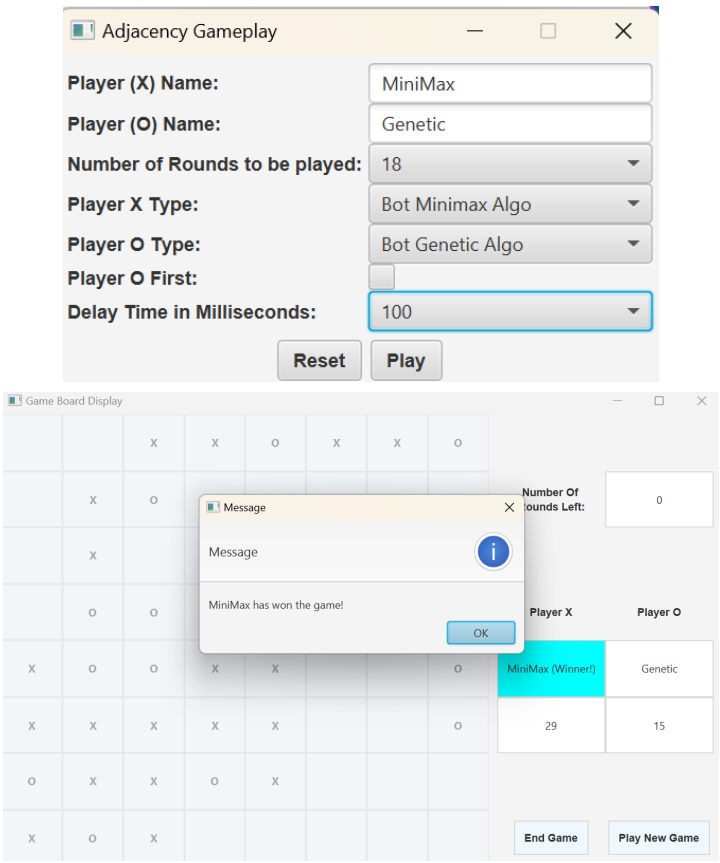
1. Pengujian 1

Player X	Minimax	Player O	Genetic
Jumlah ronde	28	Giliran Pertama	O
Pemenang		X	



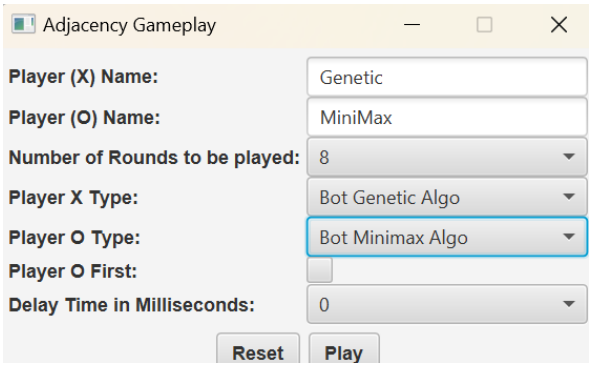
2. Pengujian 2

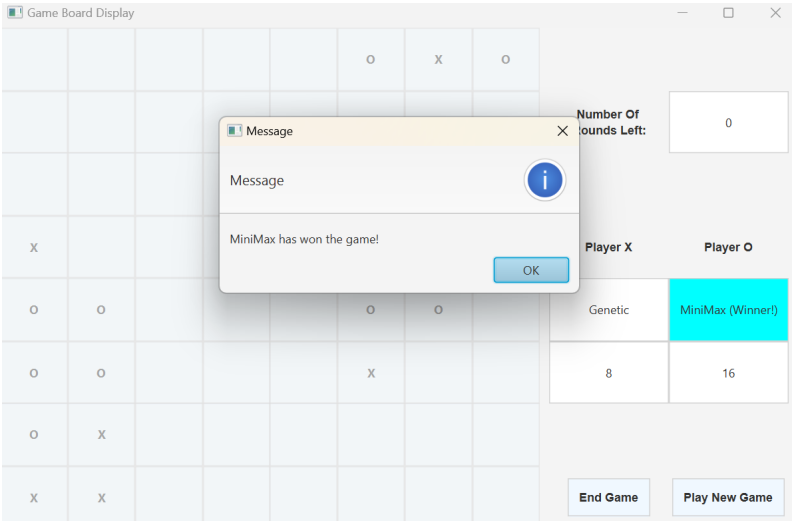
Player X	Minimax	Player O	Genetic
Jumlah ronde	18	Giliran Pertama	X
Pemenang		X	



3. Pengujian 3

Player X	Genetic	Player O	MiniMax
Jumlah ronde	8	Giliran Pertama	X
Pemenang		O	

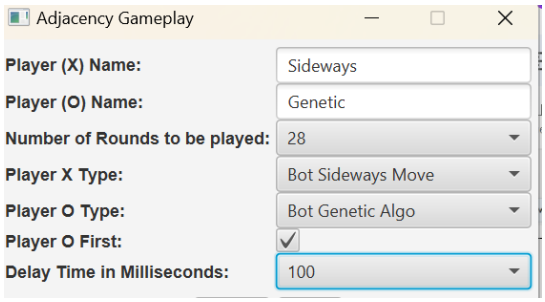


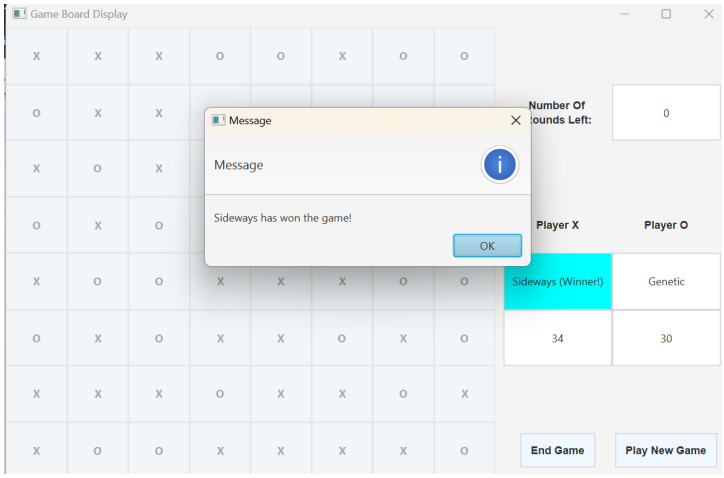


5. *Bot Local Search* vs *Bot Genetic Algorithm*

1. Pengujian 1

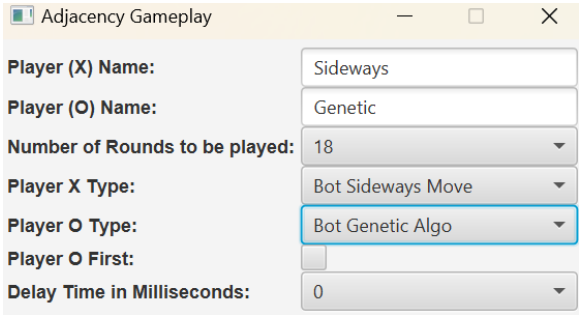
Player X	Sideways	Player O	Genetic
Jumlah ronde	28	Giliran Pertama	O
Pemenang		X	

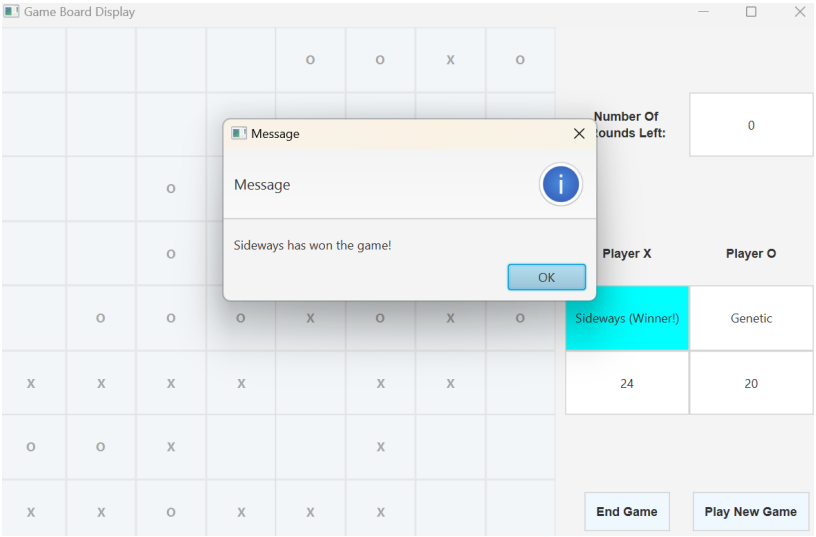




2. Pengujian 2

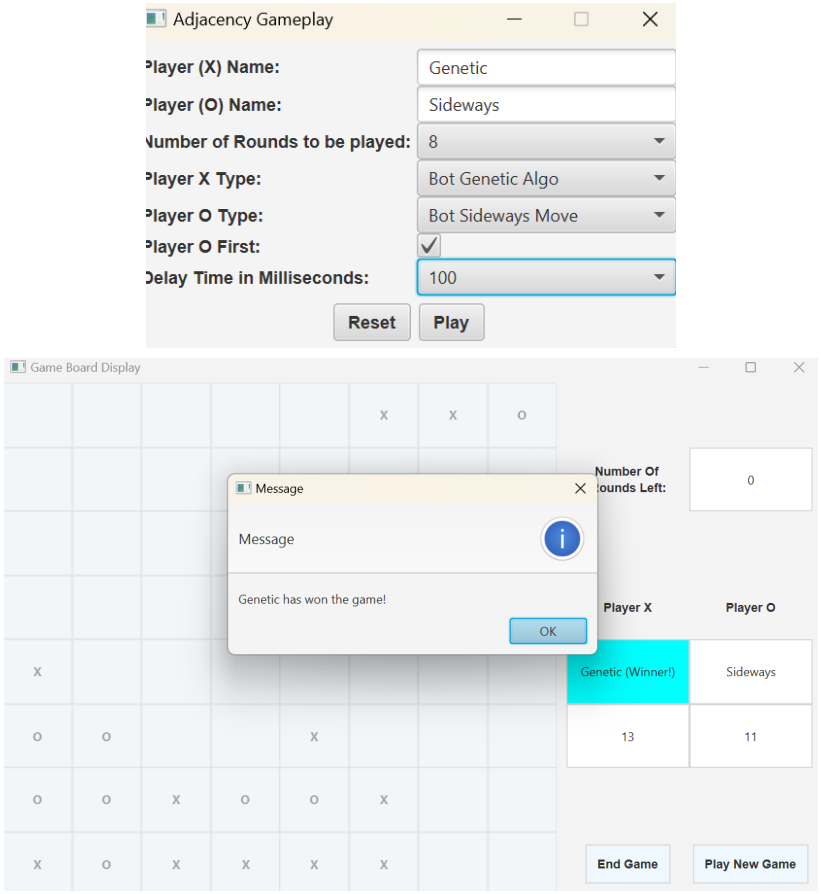
Player X	Sideways	Player O	Genetic
Jumlah ronde	18	Giliran Pertama	X
Pemenang		X	





3. Pengujian 3

Player X	Genetic	Player O	Sideways
Jumlah ronde	8	Giliran Pertama	O
Pemenang		X	



6. Rangkuman Pertandingan

1. Berdasarkan Jenis Pertandingan

Pemain 1	Pemain 2	Jumlah Pertandingan	% Menang Pemain 1	% Menang Pemain 2
Bot Minimax	Manusia	5	100%	0%
Bot Local	Manusia	3 (1 seri)	33.3%	33.3%

Search				
Bot Minimax	*Bot Local Search*	3	100%	0%
Bot Minimax	*Bot Genetic Algorithm*	3	100%	0%
Bot Local Search	*Bot Genetic Algorithm*	3	66.7%	33.3%

2. Berdasarkan Keseluruhan

Agent	Jumlah Pertandingan	Jumlah Menang	% Menang Agent
Bot Minimax	11	11	100%
Bot Local Search	9	4	44.4% (1 seri)
Bot Genetic Algorithm	6	1	16.6%

VI. Kontribusi Anggota

Berikut adalah tabel rincian mengenai pendistribusian kerja oleh anggota kelompok:

No	NIM Anggota	Nama Anggota	Deskripsi
1	13521100	Alexander Jason	Algoritma Minimax Alpha Beta Pruning, Heuristik, Laporan
2	13521116	Juan Christopher Santoso	Algoritma Local Search, Heuristik, Penyesuaian Mode Permainan, Laporan
3	13521139	Nathania Calista Djunaedi	Algoritma Minimax Alpha Beta Pruning, Heuristik, Laporan
4	13521162	Antonio Natthan Krishna	Algoritma Genetic, Algoritma Minimax Alpha Beta Pruning, Laporan

VII. Lampiran

Repository tempat penyimpanan program dapat diakses melalui **link** berikut.