

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

Semester I tahun 2023/2024



Kelompok 21:

Puti Nabilla Aidira (13521088)

Aulia Mey Diva Annandya (13521103)

Tabitha Permalla (13521111)

Althaaf Khasyi Atisomya (13521130)

Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung  
2023

## Daftar Isi

Daftar Isi	1
I. Objective Function	2
II. Proses Pencarian dengan Minimax dan Alpha-Beta Pruning	6
III. Proses Pencarian dengan Local Search	11
IV. Proses Pencarian dengan Genetic Algorithm	12
V. Hasil Pengujian	21
VI. Kontribusi Anggota	29
Referensi	30
Link Source Code	30

## I. Objective Function

Adjacency Strategy Game adalah permainan yang umumnya dimainkan oleh 2 orang. Tiap pemain biasanya direpresentasikan dengan simbol tertentu, dalam permainan di tugas besar ini, pemain direpresentasikan dengan marka 'O' dan 'X'. Setiap kali pemain meletakkan markanya di kotak kosong, maka apabila kotak-kotak tepat di atas, bawah, kiri, atau kanan dari kotak kosong tersebut terisi dengan marka lawan, maka isi dari kotak di sekitar itu akan diganti dengan marka pemain. Tujuan dari permainan adalah memiliki marka sebanyak mungkin di akhir permainan.

Poin dalam permainan ini dihitung berdasarkan jumlah kotak yang terisi oleh simbol pemain. Sehingga jumlah kotak yang diisi oleh simbol pemain dan juga jumlah kotak yang diisi oleh simbol lawan menjadi elemen-elemen penentu dari fungsi objektif.

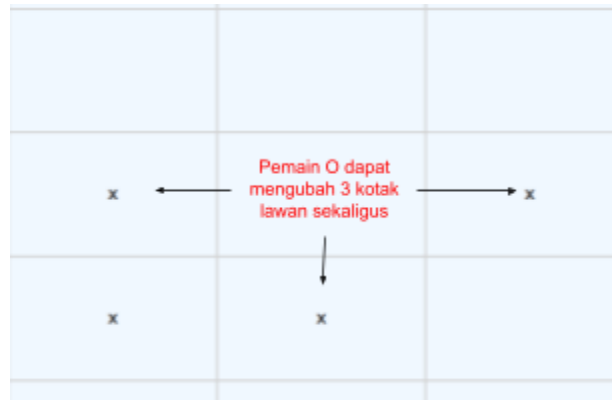
Lawan	Pemain
4	4

Selain itu, peluang suatu kotak dapat diubah oleh lawan juga harus masuk dalam pertimbangan. Misalnya, kotak yang berada di pinggir papan akan lebih sulit diubah dibandingkan dengan kotak di tengah papan. Kotak yang semua sisinya telah diisi juga pasti lebih menguntungkan daripada kotak yang sisi-sisinya masih kosong.



Di sisi lain, juga perlu dipertimbangkan pola letak yang dapat merugikan pemain itu sendiri. Sebagai contoh, letak simbol-simbol pemain pada papan sedemikian rupa sehingga pada

langkah selanjutnya lawan dapat menempatkan simbolnya untuk mengubah lebih dari satu simbol milik pemain.



Dengan mempertimbangkan faktor-faktor di atas dan asumsi bahwa bot selalu direpresentasikan oleh simbol 'O', kami merumuskan fungsi objektif sebagai berikut:

$$objective\ function = \sum_{i=1}^8 \sum_{j=1}^8 f(i, j)$$

Dengan

$$f(i, j) = k * a$$

Dimana nilai  $k$  sebagai berikut:

Nilai $k$	Isi grid (i,j)	Penjelasan
1	'O'	Kotak yang diisi 'O' (simbol bot) merupakan poin plus bagi bot
-1	'X'	Kotak yang diisi 'X' (simbol lawan) merupakan poin minus bagi bot
-0,5 (nilai dapat disesuaikan lagi setelah dilakukan eksperimen)	kosong	Kotak kosong dapat menjadi memberikan poin plus ataupun minus, tergantung pada isi kotak/grid di sekitarnya. Kotak kosong akan memberi poin plus jika di sekelilingnya terdapat banyak simbol lawan ('X'), karena memberi peluang untuk mengubah simbol. Akan tetapi berpotensi merugikan jika di sekelilingnya terdapat banyak simbol milik sendiri ('O').

		<p>Namun, poin plus dari kotak kosong tidak diperhitungkan dalam fungsi objektif karena tidak ada jaminan kotak itu akan tetap kosong pada pergerakan lawan selanjutnya. Justru, diharapkan, kotak kosong dengan banyak simbol lawan di sekitarnya diisi sesegera mungkin.</p> <p>Sehingga hanya kotak kosong yang berpotensi merugikan yang diperhitungkan dalam fungsi objektif. Akan tetapi, karena kotak kosong hanya 'berpotensi' merugikan, maka signifikansi nilai kali kotak kosong lebih kecil dibandingkan nilai kali kotak yang terisi 'X'.</p>
--	--	--

Dan nilai  $a$  sebagai berikut, dengan catatan nilai dapat dimodifikasi untuk menghasilkan rumus yang optimal setelah dilakukan eksperimen:

Nilai $a$	Isi grid (i,j)	Kondisi kotak di sekeliling grid (i,j)	Penjelasan
1.4	'O' atau 'X'	Tidak ada kotak kosong	<p>Semakin banyak kotak terisi di sekitar grid, maka semakin besar pengaruhnya. Semakin sedikit kotak terisi, semakin kecil juga pengaruhnya.</p> <p>Pada kasus grid (i,j) terisi, simbol yang mengisi grid/kotak tetangga/sekeliling tidak memiliki pengaruh.</p> <p>Pada kasus grid (i,j) kosong, hanya simbol 'O' yang masuk pertimbangan karena alasan yang sudah disebutkan di tabel sebelumnya</p>
	kosong	Ada 4 kotak terisi 'O'	
1.3	'O' atau 'X'	Ada 1 kotak kosong	
	kosong	Ada 3 kotak terisi 'O'	
1.2	'O' atau 'X'	Ada 2 kotak kosong	
	kosong	Ada 2 kotak terisi 'O'	
1.1	'O' atau 'X'	Ada 3 kotak kosong	
	kosong	Ada 1 kotak terisi 'O'	
1	'O' atau 'X'	Ada 4 kotak kosong	
0	kosong	Tidak ada kotak terisi 'O'	

Apabila disimpulkan, nilai fungsi objektif yang kami rumuskan ditentukan oleh semua kotak di papan permainan. Nilai tiap-tiap grid/kotak ditentukan oleh isi grid/kotak serta kondisi grid/kotak di sekitarnya (sisi atas, bawah, kiri, dan kanan).

## II. Proses Pencarian dengan Minimax dan Alpha-Beta *Pruning*

Sebelum melakukan pencarian dengan algoritma Minimax, permainan didefinisikan secara formal dengan elemen-elemen sebagai berikut:

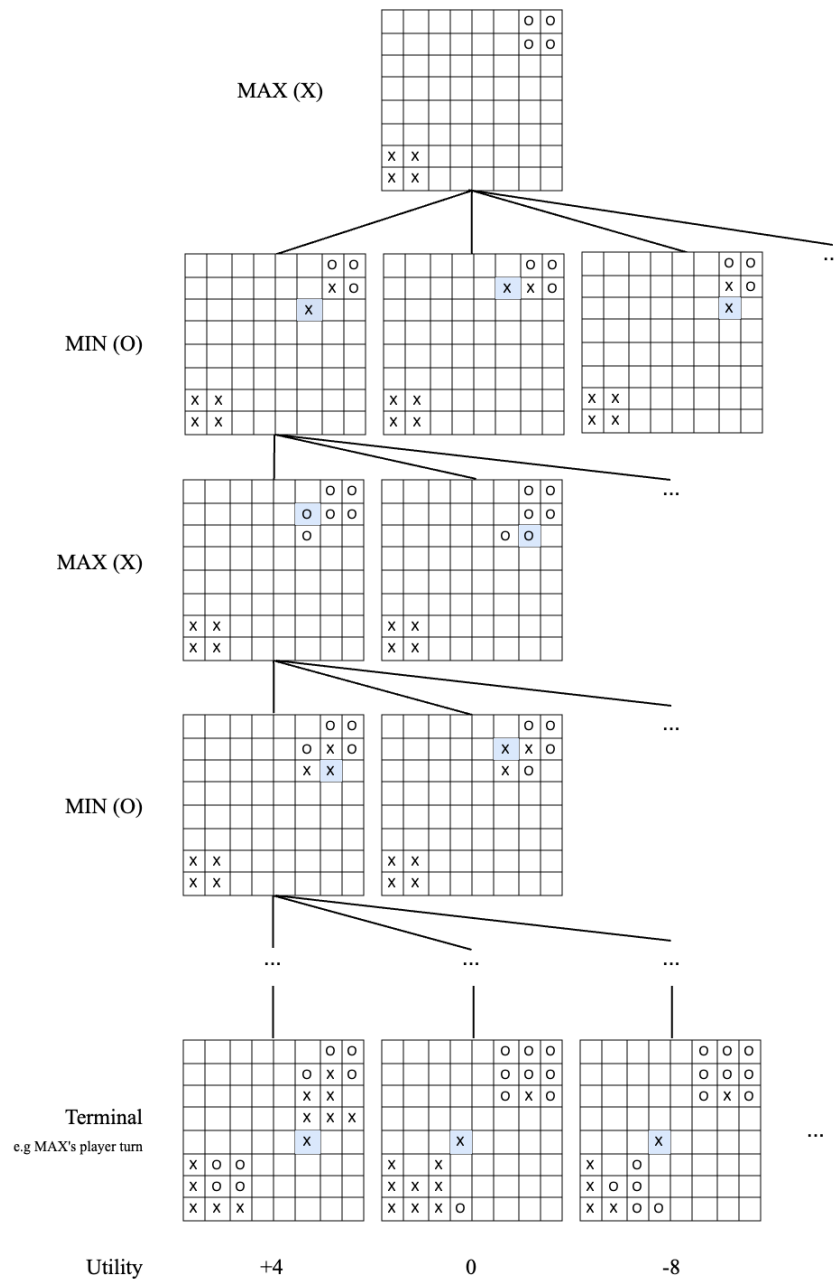
- a. *Initial State* ( $S_0$ ): Papan 8x8 dengan 4 'X' di pojok kiri bawah, 4 'O' di pojok kanan atas, dan sisanya kosong.
- b. *Player(s)*: Pemain yang mendapat giliran.
- c. *Actions(s)*: Menempatkan marka (X untuk pemain pertama dan O untuk pemain kedua) pada salah satu kotak kosong di papan permainan.
- d. *Result(s, a)*: *State* yang dihasilkan setelah pemain melakukan aksi penempatan marka X/O pada kotak kosong dan seluruh kotak *adjacent* yang berisi marka lawan berubah menjadi marka pemain.
- e. *Terminal-Test(s)*: *Terminal state* dicapai (*terminal-test* bernilai *true*) ketika sudah mencapai batas ronde yang ditetapkan atau waktu eksekusi melewati batas *timeout* (5 detik).
- f. *Utility(s, p)*: Nilai *utility* dihitung berdasarkan *objective function* (dijelaskan pada bagian I).

Selanjutnya, pencarian dengan Minimax Alpha-Beta *Pruning* dilakukan dengan langkah-langkah berikut:

- a. Pembentukan *Game Tree*

Dibentuk *game tree* dengan *game state (result)* sebagai simpul dan *moves (actions)* sebagai sisi (*edge*). Akar dari *game tree* merupakan *initial state*. Pada *game tree* pemain *max* dan *min* bermain bergantian sampai mencapai simpul daun yang merupakan *terminal state*. Pembangkitan *game tree* dilakukan secara *depth-first*. Pada permainan ini *game tree* memiliki *branching factor* = 56 dan *depth* = jumlah ronde. Dengan kata lain, *game tree* memiliki 56! simpul terminal jika *terminal-test* bernilai *true* ketika sudah tidak ada kotak kosong dan  $\frac{56!}{n!}$  jika *terminal-test* bernilai *true* ketika sudah mencapai ronde ke-*n*. Ukuran yang sangat besar membuat *game tree* sulit dibangkitkan. Sehingga, proses pencarian tidak *feasible* dilakukan tanpa *optimasi*. Selain itu, kompleksitas ruang dengan

optimasi pembangkitan aksi satu per satu adalah  $O(m)$  dengan  $m = \text{depth}$ , pada kasus terburuk  $m = 56$ . Berikut ilustrasi *game-tree* permainan (hanya diperlihatkan sebagian).



Gambar 4.1 *Game tree* permainan *Adjacency Strategy*



b. Perhitungan Nilai Minimax

Pada tiap simpul (*game state*) dilakukan perhitungan nilai Minimax. Perhitungan nilai dilakukan dengan melakukan *complete depth-first exploration* hingga mencapai nilai *utility* pada simpul terminal. Pada setiap *level tree* dipilih nilai maksimum atau minimum tergantung giliran pemain *min* dan *max*. Karena menggunakan *depth-first* kompleksitas perhitungan adalah  $O(b^m)$  dengan  $b = \text{branching factor}$  dan  $m = \text{depth}$ . Sehingga, pada kasus terburuk dan tanpa optimasi kompleksitas perhitungan adalah  $56^{56} \approx 7.9164325e + 97$ . Perhitungan nilai Minimax dapat dituliskan sebagai berikut.

```
MINIMAX(s) =  
{ UTILITY(s) if TERMINAL-TEST(s)  
{ maxa ∈ Actions(s) MINIMAX(RESULT(s, a)) if PLAYER(s) = MAX  
{ mina ∈ Actions(s) MINIMAX(RESULT(s, a)) if PLAYER(s) = MIN
```

c. Alpha-Beta *Pruning*

Teknik alpha-beta *pruning* dilakukan untuk mengoptimasi algoritma Minimax dengan mengurangi jumlah simpul (nodes) yang harus dianalisis dalam pencarian *game tree*. Alpha-beta *pruning* dilakukan dengan menyimpan nilai alpha dan beta pada tiap simpul dalam jalur pencarian. Alpha adalah nilai terbaik yang ditemukan pada jalur yang sedang dieksplorasi oleh pemain *max*. Beta adalah nilai terbaik yang ditemukan pada jalur yang sedang dieksplorasi oleh pemain *min*. Selanjutnya, *pruning* (pemotongan simpul) dilakukan ketika nilai suatu simpul yang sedang diperiksa lebih buruk dari nilai alpha atau beta saat ini berturut-turut untuk pemain *max* dan *min*.

d. *Timeout*

Batas waktu berpikir bot (*timeout*) adalah 5 detik. Pada implementasi algoritma, dilakukan pengecekan waktu sebagai bagian dari *terminal test* pada pemanggilan fungsi `minValue` dan `maxValue` (tahap pembentukan *game tree*).

Berikut adalah *pseudocode* dari pencarian langkah menggunakan Minimax Alpha-Beta *Pruning* dengan *timeout*.

```
function miniMax(gameState) returns action
  bestValue =  $-\infty$ 
  alpha =  $-\infty$ 
  beta =  $+\infty$ 
  bestAction = getActions(gameState)[0]
  for each action in getActions(gameState):
    value = minValue(result(gameState, action, player), alpha, beta,
                      roundsLeft - 1)

    if (value > bestValue) then
      bestValue = value
      bestAction = action
    alpha = max(alpha, bestValue);
  return bestAction
```

```
function maxValue(gameState, alpha, beta, roundsLeft) returns utilityValue
  if (roundsLeft == 0 || time >= TIMEOUT) then
    return utility(gameState)
  value =  $-\infty$ 
  for each action in getActions(gameState):
    value = max(value, minValue(result(gameState, action, player),
                                alpha, beta, roundsLeft-1))

    if (value >= beta) then
      return value // pruning
    alpha = max(alpha, value)
  return value
```

```
function minValue(gameState, alpha, beta, roundsLeft) returns utilityValue
  if (roundsLeft == 0 || time >= TIMEOUT) then
    return utility(gameState)
  value =  $+\infty$ 
  for each action in getActions(gameState):
    value = min(value, maxValue(result(gameState, action, enemy),
                                alpha, beta, roundsLeft-1))

    if (value <= alpha) then
      return value // pruning
    beta = min(beta, value)
  return value
```

```
function getActions(gamestate) return actions
```

```
gamestate.filter(x, y -> gameState[x][y].equals(""))
```

```
function result (gameState, action, p) returns gameState  
  e = "O" if p is "X" else "X"  
  newState = create 8x8 empty array  
  copy gameState to newState  
  newState[action[0]][action[1]] = p  
  for box in neighbours(newState[action[0]][action[1]])  
    if box = e then  
      box = p  
  return newState
```

### III. Proses Pencarian dengan Local Search

Pencarian Local Search dapat dilakukan dengan algoritma Stochastic Hill Climbing (SHC). Algoritma Stochastic Hill Climbing merupakan variasi dari algoritma Hill Climbing yang dapat mengurangi kekurangan yang ada pada Hill Climbing biasa. Stochastic Hill Climbing memilih tetangga secara acak. Setelah tetangga terpilih, algoritma akan melakukan evaluasi dengan fungsi objektif dan membandingkannya dengan solusi saat ini. Proses ini diulang sampai kriteria berhenti terpenuhi. Berikut adalah properti SHC yang akan digunakan pada permainan:

- Current* adalah *state* dari board game saat ini.
- Fungsi *getRandomAction* yang akan memilih kotak yang belum memiliki marka secara acak.
- Neighbor* adalah suksesor dari *state* saat ini.
- Value* dari state yang didapat dari *objective function*.
- Kriteria berhenti algoritma, pencarian pada local search akan berhenti.

Algoritma pencarian dengan SHC diawali dengan menginisiasi *current* menggunakan fungsi *getRandomAction*. *Current* adalah koordinat kotak yang akan dipilih sebagai action selanjutnya. Selanjutnya akan dipilih *neighbor* dari *current* yang di-generate secara random. Nilai objektif dari *neighbor* akan dibandingkan dengan nilai objektif dari *current*. Apabila nilai objektif *neighbor* lebih besar dari *current*, *neighbor* tersebut dijadikan *current* yang baru. Proses ini akan dilakukan sampai kriteria berhenti terpenuhi. Kriteria berhenti yang digunakan pada pencarian adalah apabila proses *looping* telah berjalan lebih 64 iterasi. Kriteria berhenti ini dipilih agar proses pencarian tidak berjalan terlalu lama tetapi tetap menghasilkan hasil yang baik.

```
private int[] localSearch(){
    int[] current = getRandomAction();
    int nMax = 0;
    long startTime = System.nanoTime();
    while (nMax<=64 && System.nanoTime() - startTime < TIMEOUT){
        int[] neighbor = getRandomAction();
        if (objectiveFunction(neighbor) > objectiveFunction(current)){
            current = neighbor;
        }
        nMax++;
    }
    return current;
}
```

## IV. Proses Pencarian dengan Genetic Algorithm

Di dalam konteks permainan ini, penggunaan algoritma genetika atau *genetic algorithm* menjadi sebuah pendekatan yang menarik dalam mencari solusi optimal. Melalui proses iteratif yang terinspirasi dari evolusi biologis, *genetic algorithm* dapat mengoptimalkan strategi dan pergerakan dalam permainan dengan memanfaatkan konsep seleksi alami, reproduksi, dan mutasi. Berikut merupakan penggunaan *genetic algorithm* bekerja dalam permainan *Adjacency Strategy* ini dengan mengoptimalkan gerakan dan memenangkan permainan dengan pendekatan yang cerdas dan adaptif.

Komponen-komponen *Adjacency Strategy Game* dapat dipetakan menjadi komponen-komponen *Genetic Algorithm* sebagai berikut:

- a. *Gene*: Koordinat langkah yang mungkin diambil (baik oleh bot, ataupun oleh lawan)
- b. *Chromosome*: Kumpulan *gene* / kombinasi langkah, panjangnya sesuai dengan jumlah langkah yang tersisa (jumlah langkah sendiri + jumlah langkah lawan)
- c. *Population*: Kumpulan *chromosome* / kumpulan kombinasi langkah yang menjadi kandidat solusi

Langkah-langkah penyelesaiannya adalah sebagai berikut:

- a. Inisialisasi Populasi Awal ('initializePopulation')

Langkah ini memungkinkan penciptaan langkah-langkah awal atau individu-individu awal (kumpulan *state permainan*) yang akan mengalami proses evolusi. Dengan menghasilkan populasi awal, algoritma genetika memiliki dasar untuk memulai proses pencarian solusi yang lebih baik. Maka dari itu, dibuat sebuah fungsi '**initializePopulation**' yang bertanggung jawab untuk membuat populasi awal. Populasi ini terdiri dari sejumlah individu yang masing-masing berisi koordinat acak di papan permainan. Setiap individu merepresentasikan langkah bot dan musuh bergantian masing-masing sejumlah round yang tersisa sehingga panjang populasi sebesar `roundsLeft` dikalikan dengan dua. Pada bagian ini, koordinat yang akan mengisi masing-masing state tidak boleh sama dengan koordinat yang sudah terisi sebelumnya. Fungsi ini menerima parameter berupa `String[][] gameState` untuk mengetahui kondisi papan yang belum terisi maupun sudah terisi dan `int roundsLeft` untuk mengetahui jumlah *round* yang tersisa. Fungsi mengembalikan `int[][][]` yang merupakan *array of population*,

yang mana didalamnya terdapat *array of state*, dan di dalam setiap state terdapat *array of coordinates* (x,y).

```
private int[][][] initializePopulation(String[][] gameState, int
roundsLeft) {
    Set<String> usedCoordinates = new HashSet<>();
    for (int i = 0; i < gameState.length; i++) {
        for (int j = 0; j < gameState[i].length; j++) {
            if (gameState[i][j].equals("X") ||
gameState[i][j].equals("O")) {
                usedCoordinates.add(i + "," + j);
            }
        }
    }

    int lengthState = usedCoordinates.size() % 2 == 0 ?
roundsLeft * 2 : roundsLeft * 2 - 1;
    int[][][] population = new
int[POPULATION_SIZE][lengthState][2];
    int boardSize = 8;

    for (int k = 0; k < POPULATION_SIZE; k++) {
        for (int i = 0; i < lengthState; i++) {
            int x, y;
            String coordinate;
            int []coor = {-1,-1};
            do {
                x = (int) (Math.random() * boardSize);
                y = (int) (Math.random() * boardSize);
                coordinate = x + "," + y;
                coor[0] = x;
                coor[1] = y;
            } while (usedCoordinates.contains(coordinate) ||
(Arrays.asList(population[k]).contains(coor)));
            population[k][i][0] = x;
            population[k][i][1] = y;
        }
    }

    return population;
}
```

b. Seleksi ('selection')

Seleksi memungkinkan individu yang memiliki kinerja lebih baik (berdasarkan fungsi *fitness*) untuk dipilih sebagai orang tua dalam proses reproduksi sehingga karakteristik unggul dari generasi sebelumnya dapat diwariskan ke generasi berikutnya. Maka dari itu, dibuat sebuah fungsi '**selection**' yang bertanggung jawab untuk memilih individu yang akan menjadi orang tua generasi berikutnya berdasarkan nilai kecocokan (nilai *fitness function*) mereka. Proses ini dilakukan dengan menghitung probabilitas seleksi berdasarkan nilai *fitness* setiap individu. Kemudian, individu dipilih secara acak berdasarkan probabilitas yang telah dihitung. Setiap individu yang telah terpilih, akan dimasukkan ke dalam *array of individu*. Fungsi ini menerima parameter berupa `int[][][] population` yang akan dipilih secara acak berdasarkan nilai *fitness function*. Fungsi akan mengembalikan `int[][][]` yang merupakan *array of individu* yang terpilih secara acak

```
private int[][][] selection(int[][][] population) {
    float[] probabilities = new float[population.length];
    float sumFitness = 0;
    for (int i = 0; i < population.length; i++) {
        probabilities[i] = fitnessFunction(population[i]);
        sumFitness += probabilities[i];
    }
    for (int i = 0; i < probabilities.length; i++) {
        probabilities[i] /= sumFitness;
    }
    Arrays.sort(population, (a, b) ->
Float.compare(fitnessFunction(b), fitnessFunction(a)));
    int[][][] selectedPopulation = new
int[population.length][population[0].length][2];
    int index = 0;
    // random value as much as the population length
    for (int i = 0; i < population.length; i++){
        int selectedCount = 0;
        Random random = new Random();
        int randomValue = random.nextInt(101);
        // each random value check the probability
        for (int j = 0; j < probabilities.length; j++){
            if(randomValue >= selectedCount && randomValue <
```

```

selectedCount + probabilities[j] * 100){
    selectedPopulation[index] = population[j];
    index++;
    break;
} else {
    selectedCount += probabilities[j] * 100;
}
}
}

return Arrays.copyOf(selectedPopulation, population.length);
}

```

c. Reproduksi ('reproducePopulation')

Reproduksi memungkinkan individu-individu terpilih untuk meneruskan sifat-sifat unggul mereka ke generasi berikutnya. Dengan menerapkan teknik *crossover*, individu-orang tua dapat menghasilkan keturunan baru yang mewarisi kombinasi gen yang menguntungkan. Maka dari itu, dibuat fungsi '**reproducePopulation**' yang digunakan untuk menghasilkan keturunan baru dari individu-individu terpilih dari fungsi '**selection**'. Proses ini melibatkan penggunaan teknik dan fungsi '**crossover**', di mana beberapa bagian individu-individu dipertukarkan untuk menghasilkan keturunan baru. Fungsi ini menerima parameter berupa `int[][][] selectedPopulation` yang berisi hasil seleksi populasi dari fungsi '**selection**'. Fungsi ini mengembalikan `int[][][] offspringPopulation` yang merupakan populasi setelah dilakukan teknik *crossover*.

```

private int[][][] reproducePopulation(int[][][] selectedPopulation) {
    int[][][] offspringPopulation = new
int[selectedPopulation.length][selectedPopulation[0].length][2];

    // printPopulation(selectedPopulation);
    for (int i = 0; i < selectedPopulation.length; i += 2) {
        if (i + 1 < selectedPopulation.length) {
            // Get two parents from the selected population
            int[][] parent1 = selectedPopulation[i];
            int[][] parent2 = selectedPopulation[i + 1];

```



```

        // Apply crossover to create two children
        int[][][] children = crossOver(parent1, parent2);

        // Add the children to the offspring population
        offspringPopulation[i] = children[0];
        offspringPopulation[i + 1] = children[1];
    }
}

return offspringPopulation;
}

```

d. Mutasi ('mutatePopulation')

Mutasi perlu dilakukan karena memperkenalkan variasi yang diperlukan dalam populasi meskipun reproduksi dapat mentransfer sifat-sifat yang menguntungkan. Hal ini memungkinkan pencarian ruang solusi yang lebih luas, mencegah populasi terjebak pada *local minimum*, dan meningkatkan keberagaman populasi. Maka dari itu, dibuat sebuah fungsi '**mutatePopulation**' yang bertanggung jawab untuk memperkenalkan variasi ke dalam populasi dengan mengubah beberapa individu secara acak. Pada bagian ini, beberapa koordinat individu dipilih secara acak dan diganti dengan koordinat acak baru, asalkan koordinat baru tersebut tidak berada pada tempat papan yang sudah terisi sebelumnya. Fungsi ini menerima parameter berupa `int[][][] population` yang merupakan populasi dari hasil '**reproducePopulation**' yang nantinya populasi tersebut akan dilakukan mutasi pada indeks tertentu. Fungsi ini mengembalikan `int[][][] mutatedPopulation` yang berisi populasi yang sudah termutasi.

```

private int[][][] mutatePopulation(int[][][] population) {
    int[][][] mutatedPopulation = new int[population.length][][];
    Random random = new Random();
    Set<String> usedCoordinates = new HashSet<>();
    for (int i = 0; i < gameState.length; i++) {
        for (int j = 0; j < gameState[i].length; j++) {
            if (gameState[i][j].equals("X") ||
gameState[i][j].equals("O")) {
                usedCoordinates.add(i + "," + j);
            }
        }
    }
}

```

```

    }
}
for (int i = 0; i < population.length; i++) {
    int[][] currentState = population[i];
    int mutationIndex = random.nextInt(currentState.length);
    int[][] newState = new int[currentState.length][2];
    for (int j = 0; j < currentState.length; j++) {
        if (j == mutationIndex) {
            // if the index is mutated
            int newX, newY;
            String coordinate;
            int[] coor = {-1,-1};
            do {
                newX = random.nextInt(8);
                newY = random.nextInt(8);
                coordinate = newX + "," + newY;
                coor[0] = newX;
                coor[1] = newY;
            } while (usedCoordinates.contains(coordinate) ||
(Array.asList(newState).contains(coor)));
            newState[j][0] = newX;
            newState[j][1] = newY;
        } else {
            // the other state that not mutated
            newState[j][0] = currentState[j][0];
            newState[j][1] = currentState[j][1];
        }
    }
    mutatedPopulation[i] = newState;
}
return mutatedPopulation;
}

```

e. Fungsi Fitness ('fitnessFunction')

*Fitness function* digunakan untuk memberikan penilaian terhadap seberapa baik tiap-tiap solusi/*chromosome* di dalam populasi. *Fitness function* dihitung dengan mencari selisih dari jumlah mark pemain dengan jumlah mark lawan pada akhir permainan jika seluruh langkah di dalam *chromosome* diterapkan pada permainan. Akan tetapi, karena ada kemungkinan selisih ini bernilai negatif, maka *fitness function* dirumuskan sebagai berikut:

$$F = \text{jumlah marka pemain} - \text{jumlah marka lawan} + 32^*$$

*\*F ditambahkan 32 untuk menghindari nilai F negatif.*

```
private float fitnessFunction(int[][] actions){
    String[][] tempState = new String[8][8];
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            tempState[i][j] = gameState[i][j];
        }
    }
    int n = 0;
    for (int[] action : actions) {
        int x = action[0];
        int y = action[1];
        String player, enemy;
        if (n%2 == 0) {
            player = "O";
            enemy = "X";
        } else {
            player = "X";
            enemy = "O";
        }
        tempState[x][y] = player;
        if (x!=0 && gameState[x-1][y] == enemy){
            tempState[x-1][y] = player;
        }
        if (x!=7 && gameState[x+1][y] == enemy){
            tempState[x+1][y] = player;
        }
        if (y!=0 && gameState[x][y-1] == enemy){
            tempState[x][y-1] = player;
        }
    }
}
```

```

        if (y!=7 && gameState[x][y+1] == enemy){
            tempState[x][y+1] = player;
        }
        n++;
    }
    float val = 32;
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            val += (tempState[i][j] == "0" ? 1 : tempState[i][j] ==
"X" ? -1 : 0);
        }
    }
    return val;
}

```

f. Iterasi('geneticAlgorithm')

Terdapat fungsi iterasi '**geneticAlgorithm**' yang merupakan inti dari keseluruhan *genetic algorithm*. Pada bagian ini, iterasi dilakukan berulang-ulang dengan melakukan seleksi, reproduksi, dan mutasi hingga kriteria berhenti terpenuhi. Kriteria berhenti di sini ditetapkan sebagai mencapai nilai fitness function tertentu atau melewati batas waktu yang telah ditentukan. Fungsi ini mengembalikan nilai `int[]` `current` yang berisi langkah atau gerakan terbaik yang telah ditemukan berdasarkan nilai *fitness* terbaik.

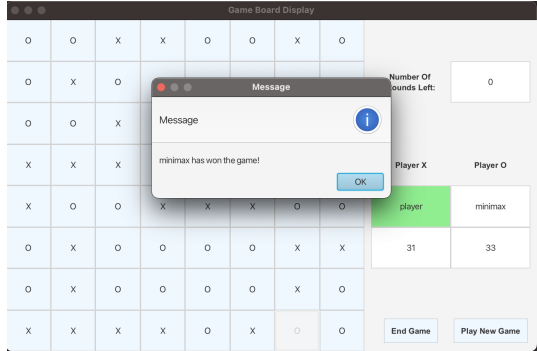
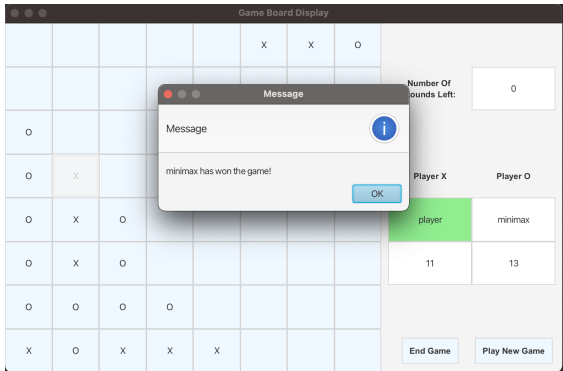
```

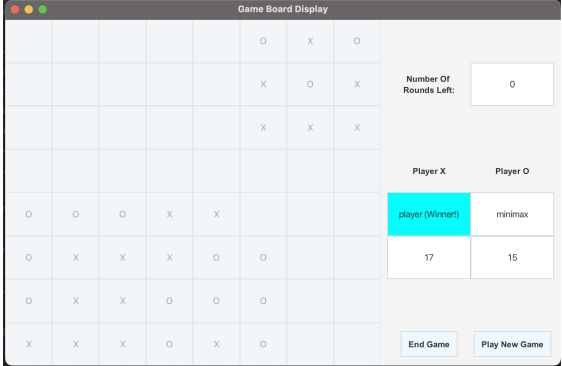
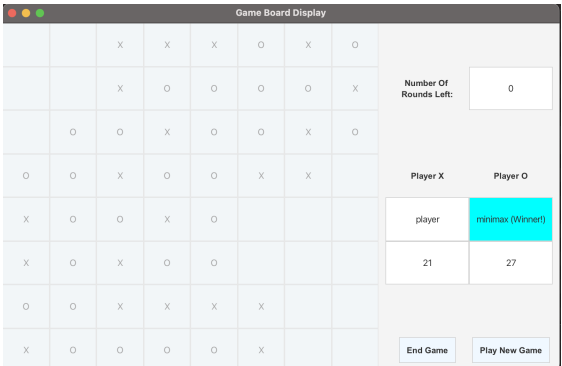
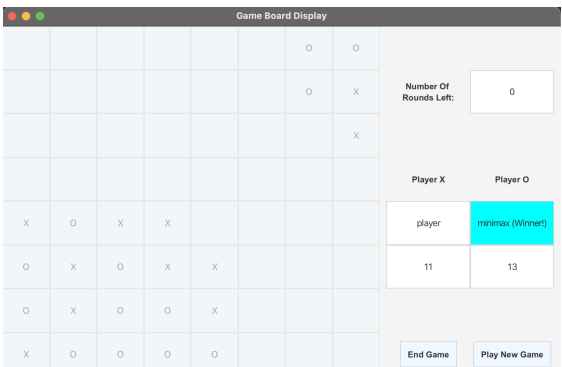
private int[] geneticAlgorithm(){
    int[][][] population = initializePopulation(gameState,
roundsLeft);
    long startTime = System.nanoTime();
    int[] current = new int[2];
    current = population[0][0];
    float bestVal = fitnessFunction(population[0]);
    for (int[][] actions : population) {
        if (fitnessFunction(actions) > bestVal) {
            bestVal = fitnessFunction(actions);
            current = actions[0];
        }
    }
}

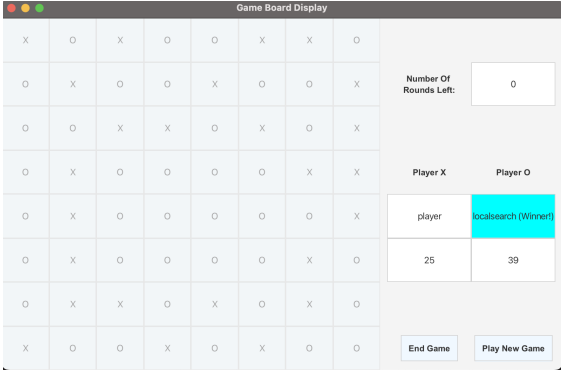
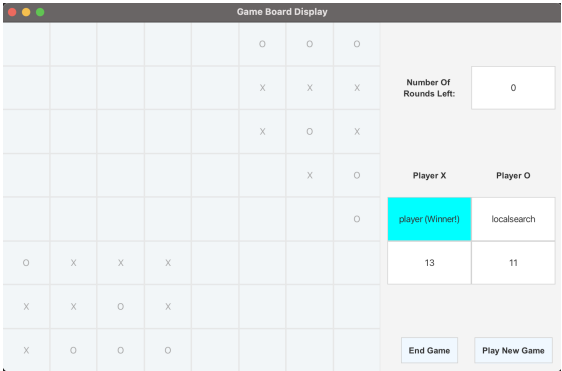
```

```
        while (bestVal < fitVal && System.nanoTime() - startTime <
TIMEOUT){
            population =
Arrays.copyOf(mutatePopulation(reproducePopulation(selection(populati
on))),population.length);
            for (int[][] actions : population) {
                if (fitnessFunction(actions) > bestVal) {
                    bestVal = fitnessFunction(actions);
                    current = actions[0];
                }
            }
            System.out.println(current[0] + " " + current[1]);
            return current;
        }
    }
```

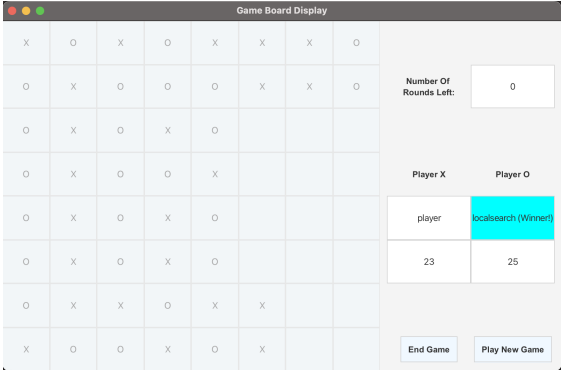
## V. Hasil Pengujian

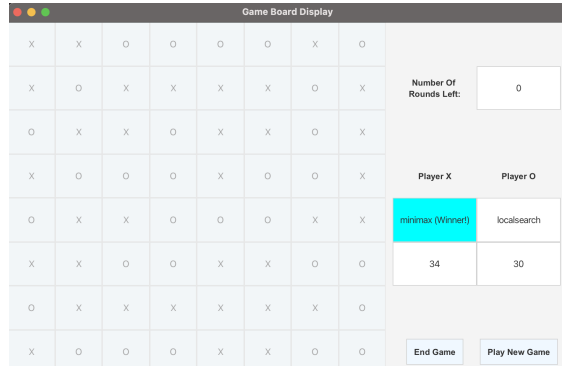
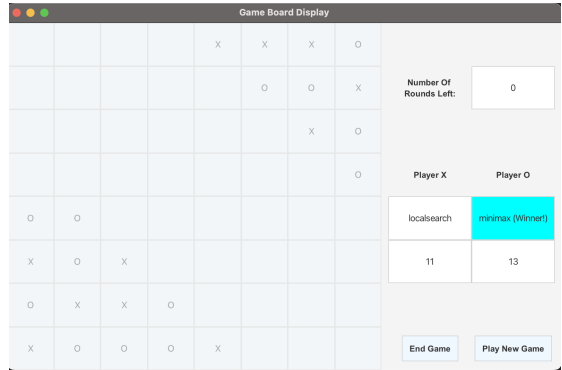
A. Minimax vs Manusia				
Match 1	Jumlah Ronde	Pemain Pertama	Bot Menang	Screenshot
	28	Manusia	1	
Match 2	Jumlah Ronde	Pemain Pertama	Bot Menang	Screenshot
	8	Manusia	1	
Match 3	Jumlah Ronde	Pemain Pertama	Bot Menang	Screenshot

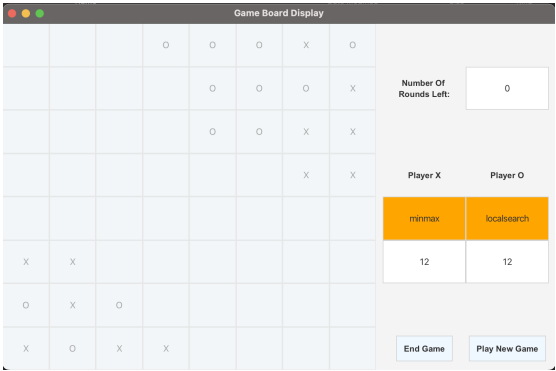
	12	Bot	0	
Match 4	Jumlah Ronde	Pemain pertama	Bot Menang	Screenshot
	20	Manusia	1	
Match 5	Jumlah Ronde	Pemain pertama	Bot Menang	Screenshot
	8	Bot	1	
Total Bot Menang: 4			Persentase Kemenangan: $\frac{4}{5} * 100\% = 80\%$	

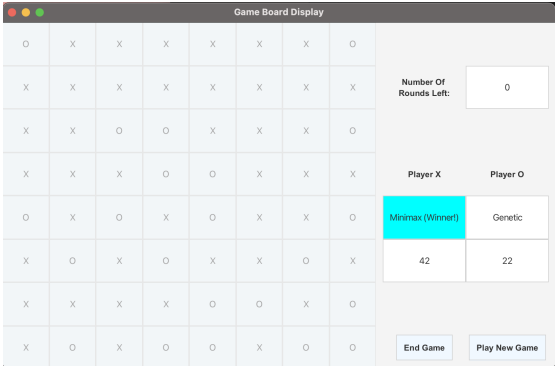
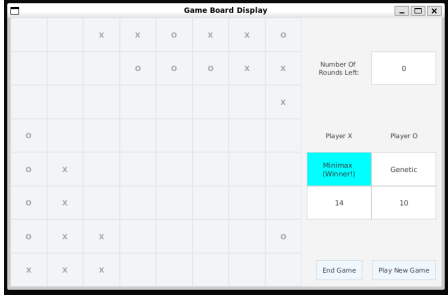
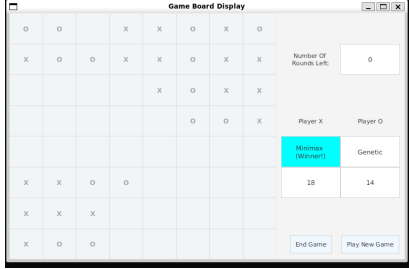
B. Local Search vs Manusia				
Match 1	Jumlah Ronde	Pemain Pertama	Bot Menang	Screenshot
	28	Manusia	1	
Match 2	Jumlah Ronde	Pemain Pertama	Bot Menang	Screenshot
	8	Manusia	0	
Match 3	Jumlah Ronde	Pemain Pertama	Bot Menang	Screenshot



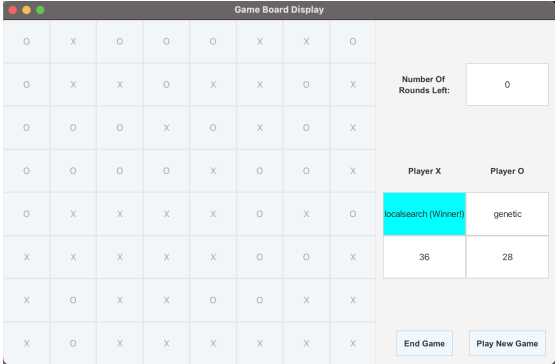
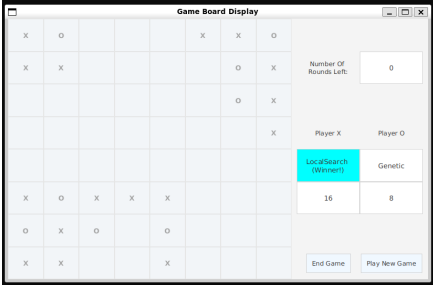
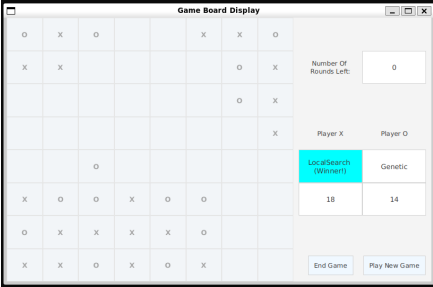
	20	Bot	1	
Total Bot Menang: 2			Persentase Kemenangan: $\frac{2}{3} * 100\% = 66.67\%$	

C. Minimax vs Local Search				
Match 1	Jumlah Ronde	Pemain Pertama	Bot Pemenang	Screenshot
	28	Minimax	Minimax	
Match 2	Jumlah Ronde	Pemain Pertama	Bot Pemenang	Screenshot
	8	Local Search	Minimax	

Match 3	Jumlah Ronde	Pemain Pertama	Bot Pemenang	Screenshot
	8	Minimax	Seri	
<b>Total Bot Minimax Menang: 3</b>		<b>Persentase Kemenangan Bot Minimax:</b> $3/3 * 100\% = 100\%$		
<b>Total Bot Local Search Menang: 1</b>		<b>Persentase Kemenangan Bot Local Search:</b> $1/3 * 100\% = 33.34\%$		

D. Minimax vs Genetic				
Match 1	Jumlah Ronde	Pemain Pertama	Bot Pemenang	Screenshot
	28	Minimax	Minimax	
Match 2	Jumlah Ronde	Pemain Pertama	Bot Pemenang	Screenshot
	8	Genetic	Minimax	
Match 3	Jumlah Ronde	Pemain Pertama	Bot Pemenang	Screenshot
	12	Minimax	Minimax	
Total Bot Minimax Menang:			<b>Persentase Kemenangan Bot Minimax:</b> $3/3 * 100\% = 100\%$	

<b>Total Bot Genetic Menang:</b>	<b>Persentase Kemenangan Bot Genetic:</b> $0/3 * 100\% = 0\%$
----------------------------------	--

<b>E. Local Search vs Genetic</b>				
<i>Match 1</i>	Jumlah Ronde	Pemain Pertama	Bot Pemenang	<i>Screenshot</i>
	28	Local Search	Local Search	
<i>Match 2</i>	Jumlah Ronde	Pemain Pertama	Bot Pemenang	<i>Screenshot</i>
	8	Genetic	Local Search	
<i>Match 3</i>	Jumlah Ronde	Pemain Pertama	Bot Pemenang	<i>Screenshot</i>
	12			

<b>Total Bot Local Search Menang:</b>	<b>Persentase Kemenangan Bot Local Search:</b> $3/3 * 100\% = 100\%$
<b>Total Bot Genetic Menang:</b>	<b>Persentase Kemenangan Bot Genetic:</b> $0/3 * 100\% = 0\%$

## VI. Kontribusi Anggota

<b>Nama</b>	<b>Nim</b>	<b>Kontribusi</b>
<b><i>Source Code</i></b>		
Puti Nabilla Aidira	13521088	MiniMaxBot
Aulia Mey Diva Annandya	13521103	GeneticAlgorithmBot
Tabitha Permalla	13521111	ObjectiveFunction, GeneticAlgorithmBot
Althaaf Khasyi Atisomya	13521130	LocalSearchBot, InputFrame
<b><i>Laporan</i></b>		
Puti Nabilla Aidira	13521088	Bab II, Bab V
Aulia Mey Diva Annandya	13521103	Bab IV
Tabitha Permalla	13521111	Bab I, Bab IV, Bab V
Althaaf Khasyi Atisomya	13521130	Bab III

## Referensi

- Khodra, M.L. (2019). Modul 3: Beyond Classical Search. KK IF – Teknik Informatika – STEI ITB, IF3170 Inteligensi Buatan.
- Khodra, M.L. (2019). Modul 4: Adversarial Search. KK IF – Teknik Informatika – STEI ITB, IF3170 Inteligensi Buatan.s
- Russell, S. J., & Norvig, P. (2018). Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall.

## *Link Source Code*

[https://github.com/Bitha17/Tubes1\\_13521088](https://github.com/Bitha17/Tubes1_13521088)