

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

Disusun untuk memenuhi laporan tugas kecil mata kuliah Kecerdasan  
Buatan Institut Teknologi Bandung



Disusun oleh:

Nigel Sahl	13521043
Hosea Nathanael Abetnego	13521057
Rava Maulana	13521149
Hanif Muhammad Zhafran	13521157

**TEKNIK INFORMATIKA**

**INSTITUT TEKNOLOGI BANDUNG**

Jl. Ganesa No. 10, Lb. Siliwangi, Kecamatan Coblong,  
Kota Bandung, Jawa Barat, 40132

2023

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>1</b>
<b>I. DESKRIPSI PERSOALAN.....</b>	<b>1</b>
<b>II. JAWABAN PERSOALAN.....</b>	<b>1</b>
A. Fungsi yang Diimplementasikan.....	2
B. Objective Function.....	7
C. Proses Pencarian dengan Minimax Alpha Beta Pruning Pada Permainan Adjacency Strategy Game.....	7
D. Proses Pencarian dengan Algoritma Local Search pada Permainan Adjacency Strategy Game	11
E. Strategi Pencarian Langkah Optimum dengan Menggunakan Genetic Algorithm.....	13
F. Hasil Pertandingan.....	16
<b>III. REFERENSI.....</b>	<b>22</b>
<b>IV. LAMPIRAN.....</b>	<b>22</b>

# I. DESKRIPSI PERSOALAN

Tugas Kecil I pada kuliah IF3170 Inteligensi Buatan bertujuan agar peserta kuliah mendapatkan wawasan tentang implementasi algoritma MiniMax pada suatu bentuk permainan yang memanfaatkan adversarial search. Pada tugas kecil kali ini, permainan yang akan digunakan adalah Adjacency Strategy Game. Secara singkat, Adjacency Strategy Game adalah suatu permainan dimana pemain perlu menempatkan *mark* (O atau X) pada papan permainan dengan tujuan memperoleh marka sebanyak mungkin pada akhir permainan (dengan jumlah ronde yang telah ditetapkan).

## 1. Aturan Main

Aturan permainan Adjacency Strategy Game yang perlu diikuti adalah:

- Permainan dimainkan pada papan 8 x 8, dengan dua jenis pemain O dan X.
- Pada awal permainan, terdapat 4 X di pojok kiri bawah, dan 4 O di pojok kanan atas.
- Secara bergantian pemain X dan pemain O akan menaruh markanya di kotak kosong. Ketika sebuah kotak kosong diisi, seluruh kotak di sekitar yang sudah terisi marka musuh akan berubah menjadi marka pemain. Misal:

	O	
X		
X	X	O
X	X	X

Lalu O mengisi board[1][1]

	O	
O	O	
X	O	O
X	X	X

Perhatikan bahwa kotak pada arah diagonal tidak berubah ketika kotak kosong diisi O.

- Permainan selesai ketika papan penuh **atau** mencapai batas ronde yang telah ditetapkan.
- Pemenang adalah yang pemain yang memiliki marka terbanyak pada papan.

## 2. Source Code

<https://github.com/GAIB20/adversarial-adjacency-strategy-game>

## II. JAWABAN PERSOALAN

### A. Fungsi yang Diimplementasikan

Fungsi-fungsi yang akan kita buat antara lain:

#### 1. getEmptySpaces

Fungsi ini akan mengembalikan list berupa tuple (x,y) sebagai koordinat papan permainan yang dapat diambil oleh bot. Berikut adalah pseudocodenya:

```
Function getEmptySpaces(board: 2D array of strings) -> List of
integer arrays
    Initialize emptySpaces as an empty list of integer arrays
    For each row i from 0 to the number of rows in board
        For each column j from 0 to the number of columns in board
            If board[i][j] is an empty string
                Append [i, j] to emptySpaces
    Return emptySpaces
End Function
```

#### 2. getEmptySpacesHeuristic

Fungsi ini merupakan **fungsi heuristik** yang kelompok kami gunakan dalam mengoptimasi fungsi pada poin 1 getEmptySpaces. Heuristik yang digunakan dilakukan dengan melihat semua kemungkinan emptySpaces seperti poin 1 tapi dilakukan dengan pengecekan blok di kanan, kiri, atas, dan bawah dari blok yang dicek setidaknya terdapat satu yang terisi oleh “X” atau “O”. Seluruh kotak yang tidak bersebelahan dengan “X” maupun “O” pasti memiliki nilai objective function yang sama dan tidak akan menguntungkan jika diisi karena tidak ada blok musuh yang di-convert menjadi blok milik pemain. Dengan demikian, ruang pencarian menjadi jauh lebih sedikit. Berikut adalah pseudocodenya:

```

Function getEmptySpacesHeuristic(board):
    emptySpaces = empty list
    for i from 0 to ROW - 1:
        for j from 0 to COL - 1:
            if board[i][j] is empty:
                // check left
                if i - 1 >= 0 and isBotOrPlayer(board[i - 1][j]):
                    emptySpaces.add([i, j])
                    continue
                // check right
                if i + 1 < ROW and isBotOrPlayer(board[i + 1][j]):
                    emptySpaces.add([i, j])
                    continue
                // check up
                if j - 1 >= 0 and isBotOrPlayer(board[i][j - 1]):
                    emptySpaces.add([i, j])
                    continue
                // check down
                if j + 1 < COL and isBotOrPlayer(board[i][j + 1]):
                    emptySpaces.add([i, j])
    return emptySpaces

```

### 3. depthGame

Fungsi ini mengembalikan berapa banyak jumlah sisa langkah yang ada dalam papan permainan tersebut di round tertentu. Berikut adalah pseudocodenya:

```

Function depthGame(r1: integer, isBotFirst: boolean,
sizeEmptySpaces: integer) -> integer
    Initialize depth_game as 0
    If isBotFirst is true
        Set depth_game to 2 * r1
    Else
        Set depth_game to 2 * r1 - 1
    If sizeEmptySpaces < depth_game
        Set depth_game to sizeEmptySpaces
    Return depth_game

```

```
End Function
```

#### 4. boardValue

Fungsi ini merupakan **fungsi objektif** yang menghitung nilai papan permainan berdasarkan jumlah "O" dan "X" pada papan. Setiap "O" akan menambah 1 nilai, sedangkan setiap "X" akan mengurangi 1 nilai. Nilai ini digunakan untuk mengevaluasi keuntungan bot dalam permainan. Berikut adalah pseudocodenya:

```
Function depthGame(r1: integer, isBotFirst: boolean,
sizeEmptySpaces: integer) -> integer
    Initialize depth_game as 0
    If isBotFirst is true
        Set depth_game to 2 * r1
    Else
        Set depth_game to 2 * r1 - 1
    If sizeEmptySpaces < depth_game
        Set depth_game to sizeEmptySpaces
    Return depth_game
End Function
```

#### 5. updateGameBoard

Fungsi ini digunakan untuk mengupdate papan permainan dengan langkah yang diambil oleh pemain (baik pemain manusia atau bot) dengan mengembalikan tuple state papan dan besarnya nilai objektif setelah langkah diberikan. Fungsi ini mencari petak-petak yang harus diubah berdasarkan langkah pemain dan mengganti nilai-nilai di papan sesuai dengan aturan permainan. Delta merepresentasikan perubahan nilai pada papan permainan. Berikut adalah pseudocodenya:

```

Function  updateGameBoard(board: 2D array of strings, player:
string, boardValue: integer, i: integer, j: integer) -> Pair of 2D
array of strings and integer
    Initialize startRow, endRow, startColumn, endColumn as integers
    If i - 1 < 0 Then
        startRow = i
    Else
        startRow = i - 1
    If i + 1 >= ROW Then
        endRow = i
    Else
        endRow = i + 1
    If j - 1 < 0 Then
        startColumn = j
    Else
        startColumn = j - 1
    If j + 1 >= COL Then
        endColumn = j
    Else
        endColumn = j + 1

    Initialize newBoard as a copy of the board
    For x from startRow to endRow
        newBoard = setBoard(newBoard, player, x, j)
    For y from startColumn to endColumn
        newBoard = setBoard(newBoard, player, i, y)

    Set newBoard[i][j] = player
    If player is bot
        Increment delta by 1
    Else
        Decrement delta by 1

    Return a Pair of newBoard and delta
End Function

```

## 6. setBoard

Fungsi ini digunakan untuk mengubah nilai pada papan permainan sesuai dengan langkah pemain dan mengembalikan tuple state pada papan permainan dan nilai objektifnya. Jika pemain adalah "X" dan ada "O" di petak yang sama, maka "O" akan diubah menjadi "X" dan delta dikurangi 2. Sebaliknya, jika pemain adalah "O" dan ada "X" di petak yang sama, maka "X" akan diubah menjadi "O" dan delta ditambah 2. Berikut adalah pseudocodenya:

```
Function setBoard(p: Pair of 2D array of strings and integer,
player: string, i: integer, j: integer) -> Pair of 2D array of
strings and integer
    Initialize delta as p.getValue()
    Initialize board as p.getKey()

    If player is "X" Then
        If board[i][j] is "O" Then
            Set board[i][j] = "X"
            Decrement delta by 2
        End If
    Else
        If board[i][j] is "X" Then
            Set board[i][j] = "O"
            Increment delta by 2
        End If

    Return a Pair of board and delta
End Function
```

#### 7. makeMove

makeMove adalah suatu interface yang merupakan titik masuk awal untuk membuat langkah dalam permainan. Fungsi ini diterapkan di kelas-kelas algoritma yang kami implementasikan dan memanggil fungsi lanjutan di kelas algoritma masing-masing untuk menemukan langkah terbaik.

#### 8. minimaxDecision

Fungsi ini akan dijelaskan lebih detail pada bagian D.

#### 9. steepestHillClimbing



Fungsi ini akan dijelaskan lebih detail pada bagian E.

10. Fungsi-fungsi untuk *genetic algorithm* dijelaskan pada bagian F.

## B. Objective Function

Fungsi boardValue adalah fungsi objektif yang menghitung selisih poin bot dengan poin lawan. Selisih di sini berarti menghitung jumlah simbol Bot yaitu simbol 'O' dikurangi jumlah simbol lawan yaitu simbol 'X' dalam papan.

$f(\text{state}) = \text{SumOf}(\{\text{my symbol}\}) - \text{SumOf}(\{\text{enemy's symbol}\})$   
(ilustrasi fungsi objektif)

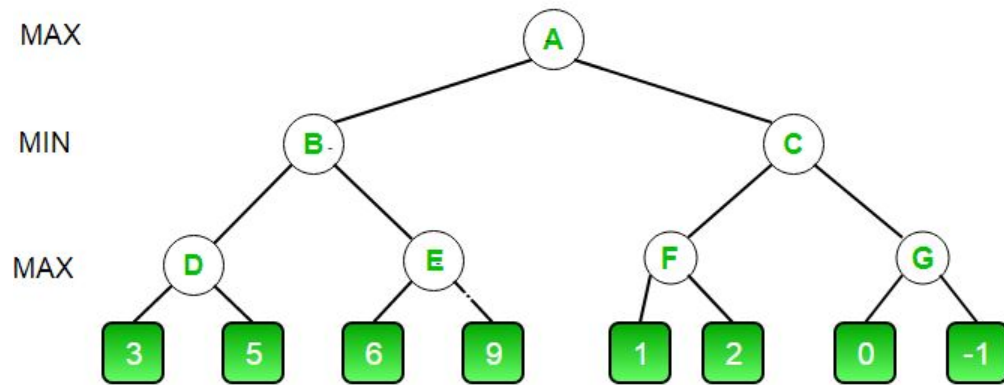
Alasan membuat fungsi objektif ini karena lebih menjangkau ke berbagai kondisi atau state dalam permainan. Pertama, jika fungsi objektif hanya didasari dengan seberapa banyak poin bot atau simbol 'O' dalam papan maka fungsi ini tidak optimal dalam memilih di antara state yang memiliki poin bot 5 dan lawan 8 dengan state yang memiliki poin bot 5 dan lawan 2, karena hanya akan memilih mungkin random atau sesuai urutan tetangga. Kedua, begitu juga halnya jika fungsi didasari dengan banyaknya poin maksimal 64 dikurangi poin lawan tidak bisa menjangkau pemilihan state yang optimal. Sehingga, kami memilih menggunakan selisih antara poin bot dengan poin lawan yang lebih bisa memilih dengan lebih baik. Contohnya dengan kondisi yang sama maka fungsi max akan memilih state dengan poin bot 5 dan lawan 2 karena memiliki nilai objektif 3 sedangkan state dengan poin bot 5 dan lawan 8 memiliki nilai objektif -3.

## C. Proses Pencarian dengan Minimax Alpha Beta Pruning Pada Permainan Adjacency Strategy Game

Fungsi minimaxDecision mengimplementasikan algoritma minimax dengan pemangkasan alpha-beta untuk membuat keputusan dalam permainan. Fungsi ini

secara rekursif menjelajahi pohon permainan untuk menemukan langkah terbaik dengan mempertimbangkan posisi yang maksimum (bot) dan minimum (pemain). Berikut adalah langkah-langkah prosesnya:

1. Cek Kondisi Terminal: Periksa apakah permainan telah mencapai kondisi terminal. Jika iya, kembalikan nilai papan saat ini dan  $(-1, -1)$  untuk koordinat.
2. Inisialisasi Variabel: Setel bestValue menjadi nilai awal yang sesuai dengan peran saat ini (maksimalkan atau minimalkan). Inisialisasi bestX dan bestY dengan -1.
3. Inisiasi bestValue dengan  $-\infty$  untuk peran memaksimalkan dan  $+\infty$  untuk meminimalkan
4. Iterasi Semua Kemungkinan Langkah yang Bisa Diambil: Loop melalui semua kotak kosong yang tersedia pada papan permainan.
5. Terapkan langkah ke papan permainan.
6. Panggil fungsi ini lagi dengan papan yang sudah disalin dan sudah diterapkan langkah dari kemungkinan langkah dengan peran kebalikan dari yang sekarang.
7. Evaluasi Nilai: Nilai yang dikembalikan oleh rekursi adalah skor dari langkah yang dianalisis. Jika skor ini lebih baik dari yang terbaik sejauh ini, perbarui bestValue, bestX, dan bestY. (Untuk peran memaksimalkan perbarui ketika nilai lebih besar dari bestValue dan sebaliknya)
8. Untuk peran memaksimalkan perbarui alpha dengan nilai maksimal antara alpha dan bestValue sedangkan untuk peran meminimalkan perbarui beta dengan nilai minimal antara beta dengan bestValue.
9. Alpha-Beta Pruning: Terapkan pemangkasan alpha-beta dengan memutuskan cabang-cabang yang tidak perlu dievaluasi. Ini dilakukan dengan memperbarui alpha atau beta dan memeriksa apakah pemangkasan diperlukan. (Jika beta lebih besar atau sama dengan dari alpha maka terapkan pruning atau pemangkasan).
10. Pengembalian Hasil: Kembalikan bestValue, bestX, dan bestY sebagai hasil dari fungsi.



Berikut adalah pseudocodenya:

```

Function minimaxDecision(board, emptySpaces, bvalue, depth, alpha, beta,
isMaximizing):
    if depth == 0:
        return [bvalue, -1, -1]

    { Maximizing Function }
    if isMaximizing:
        bestValue = -infinity
        bestX = -1
        bestY = -1
        for space in emptySpaces:
            i, j = space
            newBoard, newValue = updateGameBoard(board, bot, boardValue(board),
            i, j)
            newEmptySpaces = copy(emptySpaces)
            newEmptySpaces.remove(space)
            res = minimaxDecision(newBoard, newEmptySpaces, newValue, depth - 1,
            alpha, beta, false)
            v = res[0]
            if v > bestValue:
                bestValue = v
                bestX = i
                bestY = j
            alpha = max(alpha, bestValue)
            if beta <= alpha:
                break

```

```

{ Minimizing Function }
    else:
        bestValue = +infinity
        for space in emptySpaces:
            i, j = space
            newBoard, newValue = updateGameBoard(board, player,
            boardValue(board), i, j)
            newEmptySpaces = copy(emptySpaces)
            newEmptySpaces.remove(space)
            res = minimaxDecision(newBoard, newEmptySpaces, newValue, depth - 1,
            alpha, beta, true)
            v = res[0]
            if v < bestValue:
                bestValue = v
            beta = min(beta, bestValue)
            if beta <= alpha:
                break

        return [bestValue, bestX, bestY]

```

Fungsi minimax di atas memberikan kompleksitas ruang dan waktu yang cukup tinggi karena iterasi yang sangat banyak. Untuk minimax tanpa pruning sendiri dengan jumlah kemungkinan langkah sebagai  $n$  (jumlah emptySpaces) dan depth sebagai  $d$  (jumlah *turn* bot/pemain yang tersisa), maka akan ada sebanyak  $n * (n-1) * (n-2)$  sampai  $d$  kali iterasi atau bisa ditulis seperti di bawah ini (dengan jumlah kemungkinan langkah  $n$  dan depth  $n$ ).

$${}_nP_r = \frac{n!}{(n-r)!}$$

Hal itu juga sudah dioptimasi dari fungsi terminasi yang hanya memanfaatkan perbandingan dengan depth satu kali setiap pemanggilan fungsi minimax yang secara umum fungsi biasanya harus melakukan iterasi seluruh papan untuk mengetahui apakah papan tersebut telah terisi penuh atau tidak. Optimasi kedua pada perhitungan fungsi objektif yang dilakukan seiring langkah dilakukan yang membuat lebih cepat

sehingga tidak harus mengecek keseluruhan papan untuk menentukan nilai objektifnya.

Fungsi minimax dengan alpha-beta pruning memiliki kompleksitas yang jauh lebih baik jika dalam *base case scenario* yaitu sering dalam kondisi yang dapat di-pruning. Namun, sama saja jika dalam *worst case scenario* yaitu semua state dalam kondisi yang tidak bisa di-pruning. Jika kami perkirakan untuk *average case*, kompleksitas minimax dengan pruning ini yaitu  $nPr/2$ .

Optimasi yang dapat dilakukan adalah pada emptySpaces atau kemungkinan langkah yang dapat diambil yang diubah dengan hanya mempertimbangkan langkah dengan koordinat yang bersebelahan (atas, bawah, kiri, dan kanan) dengan simbol lawan yaitu 'X' saja kecuali memang sudah tidak ada lagi koordinat yang bersebelahan dengan simbol lawan baru memperhatikan yang lain. Hal tersebut dapat diterapkan untuk kondisi memaksimalkan dan sebaliknya untuk meminimalkan dapat dilakukan dengan memperhatikan koordinat yang bersebelahan dengan bot saja. Optimasi ini dilakukan karena koordinat lain yang tidak bersebelahan memiliki nilai objektif yang sama dan lebih rendah dibandingkan koordinat bersebelahan. Dengan begitu, iterasi dapat dilakukan dengan lebih sedikit kemungkinan langkah.

#### D. Proses Pencarian dengan Algoritma Local Search pada Permainan Adjacency Strategy Game

Algoritma yang akan digunakan untuk persoalan ini adalah *Steepest Ascent Hill Climbing*. *Local Search* hanya bisa digunakan ketika kita bisa memiliki seluruh konfigurasi akhir dari suatu permainan (*complete configuration*). *Local Search* hanya memperhatikan *current state* saja dalam pengambilan keputusan. Pada permainan ini, *bot* hanya dapat bermain langkah per langkah, sehingga algoritma *Local Search* tidak bisa digunakan untuk mencari gerakan menuju *global optima* karena *bot* tidak bisa memperkirakan gerakan dari lawannya. Beberapa algoritma *local search* seperti *Hill Climbing* dengan pendekatan stokastik, *sideways move*, algoritma *simulated annealing* kurang sesuai jika diterapkan untuk persoalan ini. *Simulated annealing*

sendiri kurang sesuai karena pada pencarian yang hanya berdasarkan lokal dan tidak *complete configuration* tidak perlu adanya penurunan dalam nilai objektif karena fokus pencarian lokal setidaknya dapat mencapai *local optima*. Sama halnya dengan hill climbing dengan stokastik, tidak perlu dicari dengan *random*. Maka dari itu, pada kasus ini digunakan *Steepest Ascent Hill Climbing* untuk mencari gerakan yang terbaik hanya untuk *current state* saja, yaitu gerakan yang memiliki nilai *objective function* paling besar. Berikut langkah-langkahnya:

1. Cari seluruh kotak yang masih kosong, kemudian simpan di dalam sebuah list
2. Iterasi seluruh kotak kosong. Setiap iterasi, hitung nilai *objective function* jika *bot* menaruh markanya di kotak kosong tersebut.
3. Ambil kotak kosong yang memiliki nilai *objective function* terbesar
4. Simpan marka pada kotak kosong yang didapat pada langkah ke-3

Berikut pseudocodenya:

```
Function steepestHillClimbing(board):
    emptySpaces = getEmptySpaces(board)
    currentCoord = emptySpaces[0]
    currentState, currentValue = updateGameBoard(board, bot, boardValue,
currentCoord[0], currentCoord[1])
    boardValue = boardValue(board)

    For each nextCoord in emptySpaces do
        nextState, nextValue = updateGameBoard(board, bot, boardValue,
nextCoord[0], nextCoord[1])
        If currentValue < nextValue Then
            currentState = nextState
            currentValue = nextValue

    x = currentValue[0]
    y = currentValue[1]

    return [x, y]
```

## E. Strategi Pencarian Langkah Optimum dengan Menggunakan Genetic Algorithm

*Genetic Algorithm* secara umum memiliki 5 tahap utama yang dilakukan secara berulang sebanyak M iterasi (M merupakan parameter yang bisa diubah-ubah), yaitu sebagai berikut:

### 1. *Initial Population*

Tahap ini merupakan tahap penentuan kromosom apa saja yang akan masuk ke tahap-tahap berikutnya. Pada iterasi pertama, biasanya dilakukan *random sampling* sebanyak N kromosom (N merupakan parameter yang bisa diubah-ubah). Kemudian untuk iterasi-iterasi berikutnya, kromosom yang dipilih merupakan kromosom yang dihasilkan pada tahap terakhir, yaitu tahap *mutation*.

### 2. *Fitness Function*

*Fitness Function* merupakan suatu fungsi yang dapat mengukur seberapa bagus suatu kromosom tertentu. Semakin besar nilai *Fitness Function* dari suatu kromosom, semakin bagus kromosom tersebut. Pada tahap ini, seluruh nilai *Fitness Function* dari tiap-tiap kromosom dari tahap sebelumnya dihitung.

### 3. *Selection*

Proses *selection* merupakan proses untuk memilih kromosom yang akan digunakan pada tahap-tahap selanjutnya dengan probabilitas tertentu. Probabilitas suatu kromosom terpilih pada proses *selection* merupakan nilai *Fitness Function* kromosom tersebut dibagi dengan jumlah seluruh nilai *Fitness Function* dari tiap-tiap kromosom. Pada tahap ini, suatu kromosom dapat terpilih lebih dari 1 kali atau bahkan tidak terpilih sama sekali, tergantung dari probabilitas terpilih masing-masing kromosom.

### 4. *Cross-Over*

*Cross-over* merupakan tahap persilangan antar dua kromosom yang terpilih pada proses *selection*. Persilangan yang dimaksud adalah menukar sebagian kromosom dengan sebagian kromosom yang lain, sehingga membentuk dua kromosom baru yang saling bersilangan. Frekuensi terjadinya *cross-over* pada suatu populasi setelah proses *selection* dapat diatur dengan probabilitas tertentu yang dijadikan sebagai parameter.

## 5. *Mutation*

*Mutation* merupakan tahap penukaran salah satu posisi pada kromosom dengan nilai *random* yang lain. Sama seperti dengan *cross-over*, frekuensi terjadinya *mutation* juga dapat diatur dengan probabilitas tertentu. Jika banyak iterasi belum mencapai batas yang ditentukan, kromosom yang dihasilkan pada proses *mutation* menjadi *initial population* pada iterasi berikutnya.

Pada permainan *Adjacency Strategy Game* ini, perlu didefinisikan kromosom yang sesuai dengan permasalahan yang dihadapi. Kromosom pada permainan ini merupakan suatu sekuens dari kiri ke kanan yang melambangkan koordinat kotak kosong yang akan diambil pada langkah tersebut (Contoh:  $\{(0, 0), (2, 1), (3, 4)\}$ ). Terdapat *constraint* yang harus dipenuhi pada definisi kromosom ini, yaitu koordinat kotak kosong yang sudah terpilih di node sebelumnya tidak boleh terpilih lagi di node setelahnya. Dengan kata lain, seluruh node pada kromosom harus unik. Panjang kromosom merupakan banyak langkah dari *current state* hingga *terminal state*.

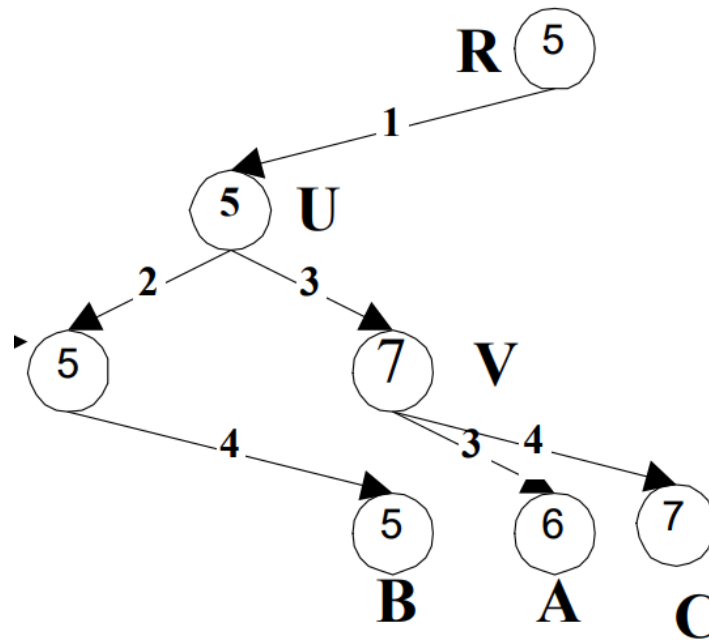
*Constraint* yang telah didefinisikan membuat proses *crossover* dan *mutation* harus dimodifikasi untuk memastikan bahwa *constraint* tersebut terpenuhi. Pada konteks permainan ini, proses tersebut akan diubah menjadi proses *swap* biasa, karena *constraint* yang sudah ditetapkan sulit untuk dipertahankan pada proses *crossover* dan *mutation* biasa. Berikut proses *swap* yang diajukan sebagai pengganti *crossover* dan *mutation*:

- Misalkan ada 2 kromosom berbeda, yaitu 12345 dan 23456.
- Pilih satu *node* secara random di setiap kromosom, kemudian tukar *node* tersebut. Misalkan proses ini menghasilkan 14345 dan 23256.
- Proses penukaran pada tahap sebelumnya menghasilkan kromosom yang melanggar *constraint*, karena terdapat *node* yang duplikat.
- Untuk mengatasi hal tersebut, ganti *node* duplikat yang lain dengan *node* yang sebelumnya ditukar. Maka hasilnya adalah 14325 dan 43256.
- Contoh lain: 12345 dan 56789  $\rightarrow$  12545 dan 36789  $\rightarrow$  12543 dan 36789

Untuk mendapatkan nilai *Fitness Function* dari setiap kromosom, gabungkan terlebih dahulu kromosom-kromosom yang ada menjadi sebuah pohon. Kemudian hitung nilai dari setiap *node* pada pohon tersebut dengan menggunakan algoritma *Minimax*. Nilai



dari *Fitness Function* adalah panjang prefix nilai node kromosom dimana nilai *node* masih tetap sama dimulai dari *leaf node*.



Contohnya pada gambar diatas, terdapat 3 kromosom, yaitu kromosom A, B, dan C. Untuk kromosom A, urutan nilai tiap *node* dari *leaf node* menuju *root node* adalah 6755. Prefix terpanjang yang memiliki nilai yang sama dari *leaf node* adalah 6, sehingga nilai *Fitness Function*-nya adalah 1. Untuk kromosom B, urutan nilai tiap *node* dari *leaf node* menuju *root node* adalah 5555. Prefix terpanjang yang memiliki nilai yang sama dari *leaf node* adalah 5555, sehingga nilai *Fitness Function*-nya adalah 4. Untuk kromosom C, urutan nilai tiap *node* dari *leaf node* menuju *root node* adalah 7755. Prefix terpanjang yang memiliki nilai yang sama dari *leaf node* adalah 77, sehingga nilai *Fitness Function*-nya adalah 2. Dengan contoh diatas, dapat disimpulkan bahwa nilai minimum dari *Fitness Function* pada permainan ini adalah 1 dan nilai maksimumnya adalah tinggi dari pohon yang dibentuk.

Setelah pendefinisian ulang proses-proses pada *genetic algorithm* yang telah disebutkan di atas, maka dapat disusun sebuah *genetic algorithm* yang memanfaatkan algoritma *Minimax* dengan langkah-langkah sebagai berikut:

1. Tentukan batas iterasi atau waktu maksimum sebesar M.
2. Tentukan probabilitas dari proses *swap* sebesar P.
3. Tentukan jumlah kromosom sebanyak N yang akan di-*sampling* ke dalam *initial population*.

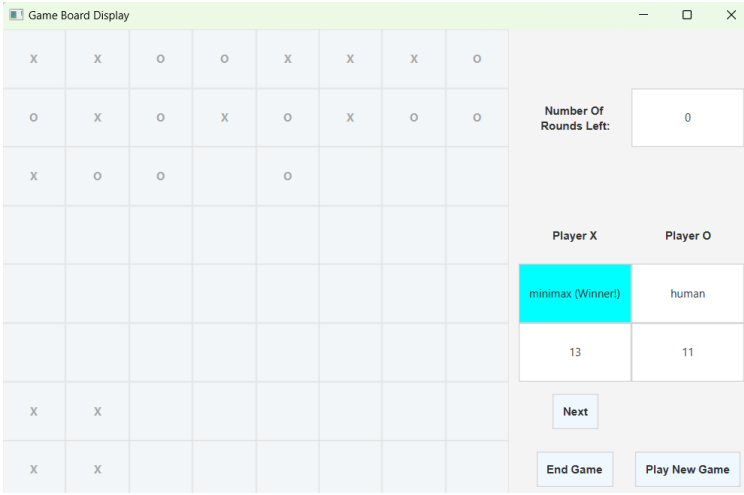
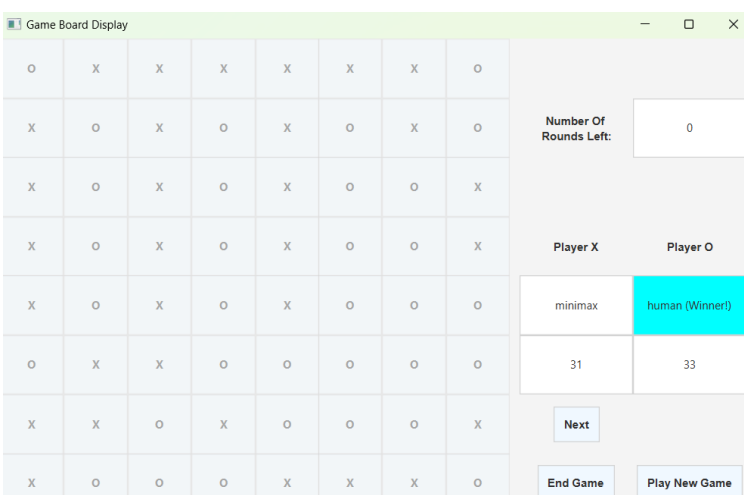
4. Lakukan proses *random sampling* untuk mengambil N kromosom.
5. Gabungkan kromosom-kromosom tersebut menjadi suatu pohon.
6. Hitung setiap *node* dari pohon yang dibentuk menggunakan algoritma *Minimax*.
7. Hitung nilai *Fitness Function* dari setiap kromosom.
8. Hitung probabilitas *selection* setiap kromosom menggunakan nilai *Fitness Function* pada tahap 7.
9. Lakukan proses sampling dengan probabilitas tiap kromosom merupakan probabilitas yang telah dihitung pada tahap 8 untuk menghasilkan sebanyak N kromosom dengan himpunan kromosom pada tahap 4-8 sebagai ruang sampel.
10. Lakukan proses *swap* dengan probabilitas dilakukannya proses *swap* adalah P.
11. Ganti *initial population* pada tahap 4 dengan kromosom hasil *swap*.
12. Ulangi tahap 5-11 hingga mencapai iterasi ke-M.
13. Lakukan tahap 5-7 untuk himpunan kromosom yang dihasilkan setelah tahap 12.
14. Pilih kromosom dengan nilai *Fitness Function* paling tinggi.
15. Isi kotak kosong pada papan permainan dengan koordinat yang pertama pada kromosom terpilih.

*Genetic algorithm* yang didefinisikan masih *feasible* karena batas iterasi maksimum dan jumlah kromosom pada suatu populasi dapat dibatasi pada angka tertentu. Semakin besar nilai iterasi maksimum dan jumlah kromosom dari suatu populasi, probabilitas menemukan kromosom dengan *Fitness Function* optimal akan semakin tinggi. Namun, penambahan dua nilai tersebut akan membuat proses pencarian semakin lama.

## F. Hasil Pertandingan

1. Bot *minimax* vs manusia

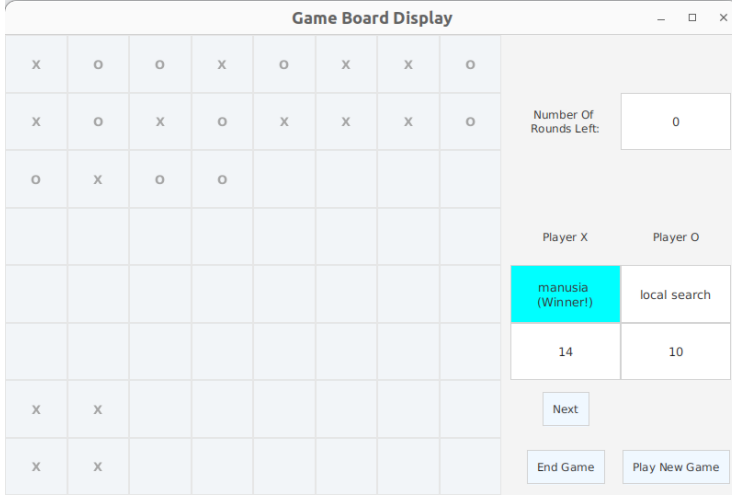
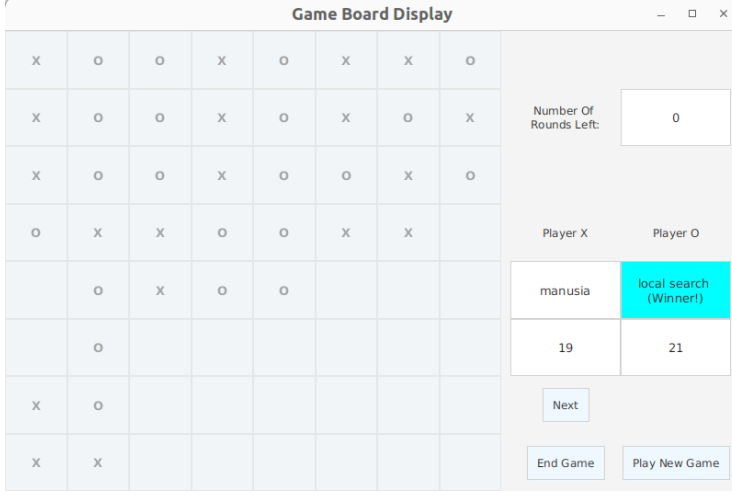
Jumlah Ronde	Pemenang	Screenshot Hasil Akhir
-----------------	----------	------------------------

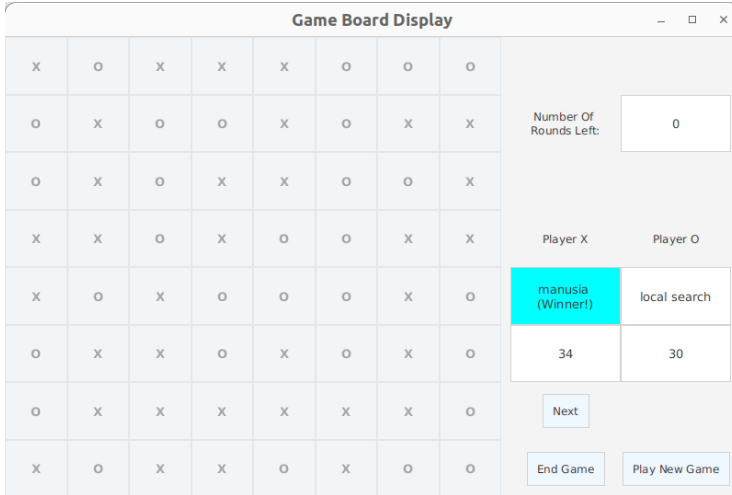
8	Minimax	
16	Minimax	
28	Manusia	

Pada pertandingan ini, dapat dilihat bahwa algoritma minimax yang kami terapkan tidak selalu optimal karena kami menerapkan batasan untuk ruang pencarian dan kedalaman permainan hingga proses terminasi untuk mengurangi waktu pencarian. Pada kenyataannya, algoritma minimax akan optimal jika fungsi heuristik yang diterapkan benar dan tidak ada batasan dalam pencarian sehingga bot akan terus melakukan

pencarian untuk semua blok kosong dan untuk kedalaman asli permainan hingga permainan berakhir.

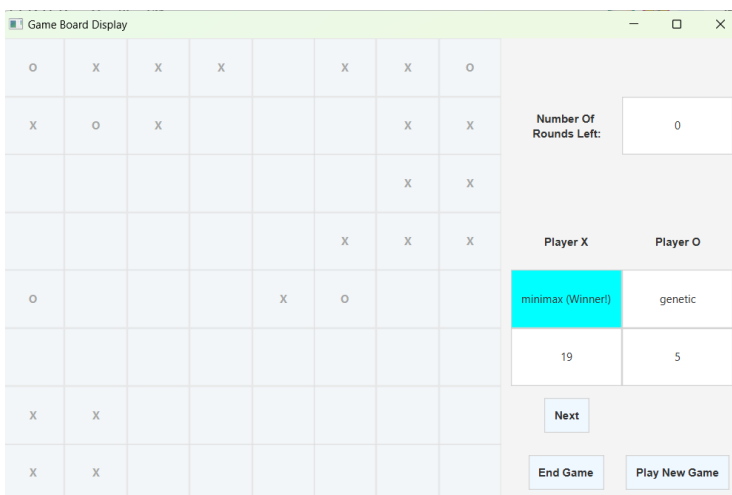
## 2. Bot *local search* vs manusia

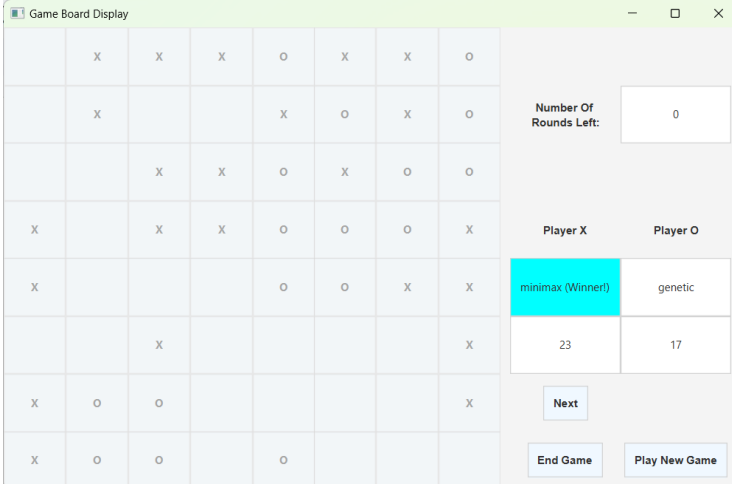
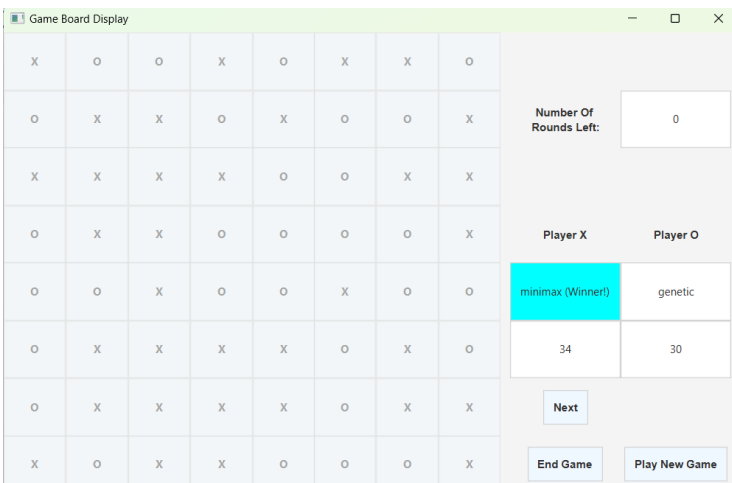
Jumlah Ronde	Pemenang	Screenshot Hasil Akhir
8	manusia	
16	<i>Local search</i>	

28	manusia	
----	---------	--

Cara bermain *Bot Local Search* akan selalu mencari *move* yang akan menghasilkan simbol miliknya yang paling banyak, sehingga *bot* akan selalu menyerang posisi simbol musuh. *Bot* akan mencari dari lokasi terdekat dari atas (pencarian dimulai dari atas) itulah mengapa penyerangan selalu dilakukan dari atas jika lawan menyerang ke posisi lawan terdekat dari atas.

### 3. Bot *minimax* vs *genetic algorithm*

Jumlah Ronde	Pemenang	Screenshot Hasil Akhir
8	Minimax	

16	Minimax	
28	Minimax	

Pada hasil pertandingan di atas, Bot Minimax berpeluang tinggi untuk menang algoritma minimax akan membuat ruang pencarian hingga proses terminasi atau sampai batas yang ditentukan dengan menerapkan fungsi objektif pada setiap giliran permainan. Namun, algoritma yang kami terapkan tidak selalu optimal karena ada batasan yang kami terapkan untuk mengurangi ruang dan waktu pencarian. *Genetic algorithm* tidak menghasilkan langkah yang terbaik, ini disebabkan oleh pemilihan langkah yang dilakukan dengan permutasi kotak-kotak kosong. Karena kesempatan untuk memilih kotak yang tidak baik lumayan besar, dengan dilakukannya pemilihan langkah *random* dan adanya mutasi dan kegiatan algoritma *genetic* lainnya, kesempatan untuk mendapatkan langkah yang kurang baik menjadi lebih besar.

#### 4. Bot *local search* vs *genetic algorithm*

Jumlah Ronde	Pemenang	Screenshot Hasil Akhir
-----------------	----------	------------------------

8	<i>Local search</i>	
16	<i>Local search</i>	
28	<i>Local search</i>	

*Bot local search* seperti pada pertandingan sebelumnya akan selalu menyerang. Akan tetapi, *bot genetic algorithm* terlihat seperti bergerak secara *stochastic*. Ini dikarenakan *randomness* dari permutasi yang dilakukan pada kotak-kotak kosong yang ditemukan. Oleh karena itu, jika semua generasi yang dihasilkan tidak memiliki langkah mengisi

kotak kosong di dekat simbol lawan, maka *next best* yang diambil adalah lokasi random lainnya.

### III. REFERENSI

- <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning>
- [https://www.researchgate.net/publication/220375025\\_Adversarial\\_Search\\_by\\_Evolutionary\\_Computation](https://www.researchgate.net/publication/220375025_Adversarial_Search_by_Evolutionary_Computation)

### IV. LAMPIRAN

- [GitHub - NerbFox/adversarial-adjacency-strategy-game](#)