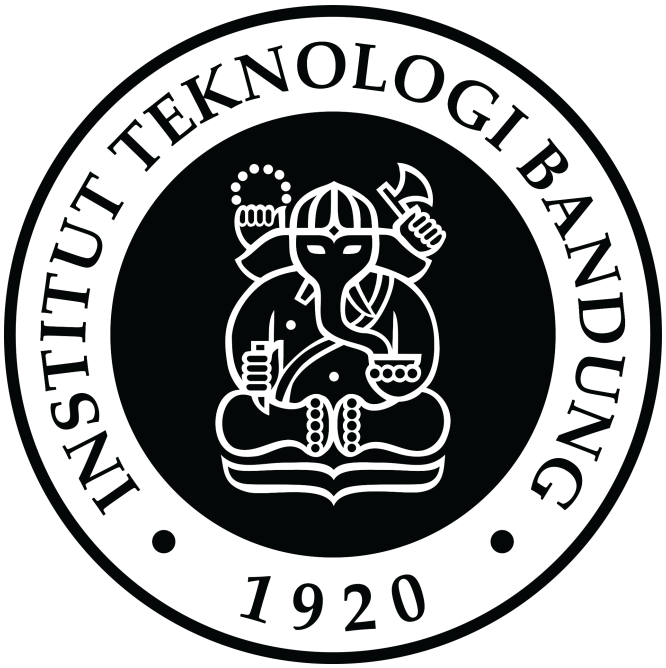
# Tugas Besar 1

# Minimax Algorithm and Alpha Beta Pruning in Adjacency Strategy Game

IF3170 Intelegensi Buatan




Penulis:

| | |
|---|---|
| Melvin Kent Jonathan | 13521052 |
| Daniel Egiant Sitanggang | 13521056 |
| Yobel Dean Christopher | 13521067 |
| Hobert Anthony Jonatan | 13521079 |

**18 Oktober 2023**

# 1. Pendahuluan

Adjacency Strategy Game adalah suatu permainan di mana pemain perlu menempatkan marka (O atau X) pada papan permainan dengan tujuan memperoleh marka sebanyak mungkin pada akhir permainan. Jenis permainan ini memerlukan pemain untuk memahami dengan cerman hubungan antara unit atau elemen di dalam permainan. Dengan fokus pada pengaturan dan pengelolaan posisi tertentu, permainan menawarkan tantangan yang menarik sehingga menuntut pemikiran strategis dan taktis yang cerdas.

Pada Tugas Besar kali ini, pendekatan Intelegensi Buatan digunakan untuk mengembangkan strategi atau taktik permainan yang akan mengoptimalkan peluang kemenangan. Dalam hal ini, Minimax Algorithm dan Alpha Beta Pruning adalah dua konsep kunci yang memainkan peran integral dalam pengembangan strategi permainan dan penyelesaian masalah terkait. Mengidentifikasi langkah terbaik sambil mempertimbangkan respons lawan merupakan tantangan yang kompleks. Minimax Algorithm adalah teknik yang digunakan untuk pengambilan keputusan, sementara Alpha Beta Pruning adalah metode yang dirancang untuk mengurangi jumlah simpul yang dieksplorasi dalam pohon permainan, menghasilkan waktu eksekusi yang lebih efisien. Masalah kinerja dan kompleksitas pohon permainan sering kali menjadi tantangan dalam pengembangan permainan berbasis Intelegensi Buatan, dan penerapan yang efisien dari algoritma yang digunakan dapat membantu mengatasi masalah tersebut. Oleh karena itu, diperlukan eksplorasi penggunaan efektif dari algoritma tersebut dalam Adjacency Strategy Game.

Selain itu, pendekatan lain seperti Local Search dan Genetic Algorithm juga akan dipertimbangkan dan diimplementasikan dalam mengembangkan strategi permainan. Local Search adalah metode heuristik yang digunakan untuk menemukan solusi yang memadai dalam ruang pencarian yang besar, sementara Genetic Algorithm adalah teknik terinspirasi dari teori evolusi yang menggabungkan prinsip seleksi alami dan rekombinasi genetik untuk menemukan solusi yang optimal dalam populasi solusi yang diperkirakan. Dengan mempertimbangkan dan menerapkan kedua pendekatan ini bersama dengan Minimax Algorithm dan Alpha-Beta Pruning, diharapkan Tugas Kecil ini dapat memberikan wawasan yang komprehensif tentang berbagai teknik yang dapat digunakan dalam pengembangan strategi permainan yang adaptif dan efisien.

# 2. Isi

## 2.1. *Objective Function*

### 2.1.1 Volatile vs Non Volatile Point

Nilai dari *objective function* terdiri dari dua komponen yang diberikan bobot berbeda. Komponen pertama adalah *non-volatile point* yang didapat dengan menghitung **selisih** antara [jumlah bidak dari bot yang tidak dapat diakuisisi lagi oleh lawan (tidak terdapat kotak kosong pada *adjacent* bidak)] dengan [jumlah bidak dari lawan yang tidak dapat diakuisisi oleh bot]. Komponen kedua adalah *volatile point* yang didapat dengan menghitung **selisih** antara [jumlah bidak dari bot yang masih dapat diakuisisi oleh lawan (dengan cara menaruh bidak lawan pada *adjacent* dari bidak bot)] dengan [jumlah bidak dari lawan yang masih dapat diakuisisi bot] . Komponen pertama diberi bobot 2 dan komponen kedua diberi bobot 1. Maka diperoleh perumusan berikut:

$$non - volatile\ point = sum\ of\ immutable\ bot\ pawn - sum\ of\ immutable\ opponent\ pawn$$
$$volatile\ point = sum\ of\ mutable\ bot\ pawn - sum\ of\ mutable\ opponent\ pawn$$
$$objective\ function = non - volatile\ point * 4 + volatile\ point * 1$$

*Objective function* diperoleh dengan menghitung selisih *pawn* yang dimiliki oleh bot dan lawan karena tujuan akhir dari permainan adalah untuk memiliki jumlah *pawn* bot yang lebih banyak dari lawan. Lalu, terdapat pembagian *immutable* dan *mutable pawn* karena *pawn* yang menempati sebuah kotak dapat berubah-ubah hingga tidak ada lagi kotak kosong pada *adjacent*. Jumlah *immutable pawn* lah yang sebenarnya menentukan kemenangan pemain. Namun, *mutable pawn* juga perlu dimasukkan ke dalam perhitungan nilai *objective function* karena *mutable pawn* adalah cikal bakal terbentuknya *mutable pawn.* Bobot *immutable pawn* 4 kali lebih besar dari bobot *mutable pawn* karena untuk memperoleh sebuah *immutable pawn* diperlukan sebanyak 4 *adjacent pawn* yang mengelilinginya, sehingga dianggap 4 kali lebih sulit diperoleh (banyak langkah yang harus ditempuh).

*Objective function* tersebut memberikan nilai dengan variasi yang paling signifikan dibandingkan dengan *naive objective function* dan *linear volatility objective function.* Karena itu, *objective function* ini dapat digunakan dalam menghitung dan membandingkan *successor-successor* dalam jumlah kecil. Oleh sebab itu, *objective function* ini digunakan sebagai

*evaluation function* untuk menghitung nilai papan permainan secara statik dalam algoritma Local Search.

### 2.1.2 Linear Volatility

Berbeda dengan *objective function* "Volatile vs Non Volatile Point" yang mengevaluasi setiap *pawn* secara biner, Linear Volatility diperoleh menghitung selisih total skor volatilitas dari setiap *pawn* dari masing-masing player. Skor volatilitas dari suatu *pawn* yang diperoleh dengan menghitung banyaknya *pawn* sejenis yang berada di kiri, kanan, atas, bawah. Dengan begitu, variasi dari skor *objective function* Linear Volatility akan jauh lebih linear daripada *objective function* Volatile vs Non-Volatile. Berikut adalah perumusan daripada Linear Volatility *objective function.*

$$objective\ function\ =\ sum\ of\ volatility\ score\ bot\ pawn\ -\ sum\ of\ volatility\ score\ opponent\ pawn$$

*Objective function* tersebut memberikan nilai yang paling komprehensif dibandingkan dengan *naive objective function* dan *volatile vs non-volatile objective function.* Karena itu, *objective function* ini dapat digunakan dalam menghitung dan membandingkan *successor-successor* dalam jumlah besar. Oleh sebab itu, *objective function* ini digunakan sebagai *evaluation function* untuk menghitung nilai papan permainan secara statik dalam algoritma Minimax yang dilakukan *cut-off* (tidak menyentuh *termination state*).

## 2.2. *Minimax* dan *Alpha Beta Pruning*

Salah satu algoritma yang dapat diterapkan pada Adversarial Game serta menjanjikan hasil yang pasti optimal adalah algoritma Minimax. Optimasi algoritma minimax dengan melakukan *alpha-beta pruning* menurunkan rata-rata kompleksitas waktu algoritma dari O(b^m) menjadi O(b^m/2).

Dalam mengimplementasikan algoritma minimax dengan alpha-beta pruning, maka perlu ditentukan beberapa hal sebagai berikut.

1. Terminal state.

   Terminal state adalah kasus dimana suatu *adversarial game* dianggap selesai dan dapat ditentukan pemenangnya. Dalam kasus *Adjacency Strategy Game*, terminal state adalah

kondisi round = 0 atau tidak ada lagi langkah yang dapat dilakukan oleh masing-masing pemain.

2. Utility function.

   Utility function adalah fungsi yang memberikan *utility value* yaitu nilai yang menentukan pemenang dari permainan tersebut. Pada kasus *Adjacency Strategy Game, utility value* bernilai 1 jika pemain maksimasi (bot) menang, 0 jika kedua pemain seri, dan -1 jika pemain minimasi menang.

Berikut algoritma Minimax dan Alpha Beta Pruning yang digunakan dalam bentuk *Pseudocode*.

```
Procedure MinimaxBot():
    Initialize constants (MAX_PLAYER, MIN_PLAYER, INF, DEF_LEAF_GENERATED,
EXECUTOR_NUM)
    Initialize evaluator, generator, executorService, and idsExecutor

    Initialize method to get the best move
        Print current state board and additional information
        Run IDSRunner to find the best move
        Return the result

    Initialize method to find the best move using Minimax algorithm
        Calculate maximum depth
        Return the best move without using threading

    Initialize method to find the best move using Minimax algorithm with a
specified maximum depth
        Get the current state and possible moves
        Initialize bestMove and bestMoveValue
        Initialize alpha and beta values
        Initialize futures list for multi-threading
        For each possible move, submit a task to the executorService
        Iterate through the possible moves and update the best move and best move
value
        Handle any exceptions and reset the executorService if needed
        Return the best move

    Initialize method for the Minimax algorithm
        Generate successors for the current game state
        Check for termination or depth limit
        If the current player is MAX_PLAYER
            Iterate through the successors and find the maximum value
            Update alpha value
        Else
            Iterate through the successors and find the minimum value
            Update beta value
        Return the best successor value

    Initialize method to get the generator
```

```
        Return the generator

    Initialize method to get the evaluator
        Return the evaluator

    Define IDSRunner class to find the best move
        Define volatile variables for the result and depth
        Initialize bot for the IDSRunner
        Define methods to get the result and depth
        Run the algorithm to find the best move
            Find the final depth based on the minimum of possible successors and
rounds left
            Iterate through the depths and find the best move
```

Dalam mengimplementasikan algoritma minimax ini, terdapat beberapa *tweak* agar algoritma tersebut dapat menjadi *feasible*. **Pertama**, dilakukan *cut-off* untuk membatasi depth dari pohon pencarian. **Kedua**, dilakukan pendekatan *iterative deepening search* agar algoritma minimax dapat terus menambah kedalamannya hingga terjadi *timeout.* **Ketiga**, dilakukan evaluasi papan permainan secara statik jika hingga *timeout*, algoritma minimax masih belum mencapai *terminate state.*

Algoritma dimulai dengan membangkitkan *successor* dari *state* bot saat algoritma mulai dijalankan, yaitu seluruh petak kosong pada papan permainan. Kemudian, dibentuk pohon minimax dengan maksimum kedalaman secara bertahap (*iterative deepening search).* Jika hingga waktu yang ditentukan (5 detik) algoritma tidak dapat mencapai *termination state,* nilai dari papan akan dihitung secara statik dengan menggunakan *evaluation function* berupa *linear volatility* sebagaimana yang telah disebutkan sebelumnya. Jika dalam waktu yang ditentukan algoritma minimax telah mencapai *termination state*, maka *evaluation function* bukan lagi menjadi acuan, melainkan akan digunakan *utility function* untuk menghitung nilai dari suatu papan.

Untuk mempercepat *run-time* dari algoritma minimax dalam bahasa pemrograman java, implementasi minimax dilakukan dengan melakukan *multi-threading.*

## 2.3. *Local Search*

Berikut algoritma Local Search yang digunakan dalam bentuk *Pseudocode*.

```
procedure LocalBot():
    initialize temperature
    initialize threshold
    repeat for sejumlah iterasi tertentu:
        evaluate current state
        select langkah random yang valid untuk bot
        predict langkah lawan menggunakan pendekatan greedy
        evaluate neighbor state berdasarkan langkah lawan
        calculate perbedaan antara neighbor value and current value
        if neighbor value > current value:
            accept neighbor solution
        else:
            calculate probabilitas penerimaan berdasarkan temperature and
threshold
            if acceptance probability > threshold:
                accept neighbor solution
            decrease temperature menggunakan cooling rate
    end repeat
end procedure
```

Algoritma Local Search yang diterapkan dalam kode tersebut merupakan sebuah pendekatan heuristik yang digunakan untuk menemukan solusi optimal dalam ruang pencarian yang besar. Langkah pertama dari algoritma ini adalah inisialisasi `temperature` dan `threshold` yang akan mempengaruhi probabilitas penerimaan solusi yang lebih buruk. Kemudian, algoritma melakukan iterasi hingga mencapai batas iterasi maksimum atau hingga mendapatkan solusi yang diterima. Selama iterasi, algoritma akan mencoba langkah-langkah acak yang valid untuk memasukkan tanda 'O' pada simpul yang kosong dalam keadaan sementara. Selanjutnya, algoritma akan menentukan langkah lawan yang paling optimal berdasarkan evaluasi keadaan yang diprediksi, di mana lawan diasumsikan melakukan langkah greedy. Untuk setiap sel yang kosong pada papan permainan, langkah tersebut dicoba dan dievaluasi untuk mengetahui apakah langkah tersebut memberikan keuntungan yang lebih baik. Jika ya, langkah tersebut dianggap sebagai langkah terbaik. Setelah langkah-lawan selesai, evaluasi keadaan saat ini dan keadaan setelah langkah-lawan dibandingkan. Algoritma menghitung nilai perbedaan antara keadaan tetangga dan keadaan saat ini, lalu menggunakan *temperature* dan *threshold* untuk menentukan penerimaan solusi berdasarkan probabilitas. Jika keadaan setelah langkah-lawan memberikan

keuntungan yang lebih besar, langkah tersebut diterima. Jika tidak, probabilitas penerimaan dihitung berdasarkan perbedaan nilai dan *temperature*, dan *temperature* kemudian dikurangi sesuai dengan *cooling rate*. Jika penerimaan terjadi, atau probabilitas penerimaan melebihi ambang batas, proses pencarian dihentikan. Setelah iterasi selesai, algoritma akan mengembalikan langkah yang telah dipilih sebagai solusi terbaik yang diterima.

Algoritma ini berusaha untuk mencari langkah terbaik untuk Bot dengan mempertimbangkan langkah-lawan. Jika langkah-lawan memberikan keuntungan yang lebih baik, langkah tersebut diterima berdasarkan perhitungan probabilitas. Jika tidak, maka algoritma akan mencoba langkah baru hingga suatu kondisi penerimaan tercapai.

Algoritma ini dipilih karena kesederhanaannya dan kemampuannya untuk bekerja secara efisien dalam pencarian ruang solusi yang berdekatan. Terutama, Local Search dapat mengatasi masalah kompleksitas yang tinggi dengan mengeksplorasi solusi yang berada dalam jarak terbatas dari solusi saat ini, sehingga lebih cepat dalam menemukan solusi yang memadai dalam waktu yang lebih singkat.

Selain itu, algoritma Local Search memungkinkan peningkatan solusi secara inkremental, yang memungkinkan bot untuk menyesuaikan langkahnya berdasarkan perubahan kondisi permainan secara real-time. Algoritma Local Search memungkinkan pencarian solusi yang lebih fleksibel dengan mengeksplorasi berbagai kemungkinan langkah berdasarkan keadaan saat ini, tanpa memerlukan pengetahuan penuh tentang lingkungan permainan. Dengan menggunakan pendekatan ini, bot dapat secara adaptif menyesuaikan langkahnya berdasarkan evaluasi saat ini dan mempertimbangkan apakah langkah yang diusulkan dapat memberikan perbaikan yang signifikan dalam permainan. Contohnya, Algoritma Local Search mampu menangani variabel acak yang terjadi dalam permainan, seperti langkah-lawan yang tidak dapat diprediksi. Dengan memperhitungkan probabilitas penerimaan, algoritma ini dapat secara adaptif menyesuaikan strategi untuk menanggapi perubahan atau tindakan yang tidak terduga dari lawan.

## 2.4. *Genetic Algorithm*

Agar dapat mengimplementasi *genetic algorithm* pada permainan ini, diperlukan beberapa modifikasi pada algoritma, yakni dengan menetapkan batasan-batasan berikut:

- *States* dari *genetic algorithm* (berbeda dengan *state* dari permainan) didefinisikan sebagai suatu *final state* dari permainan, yakni kondisi posisi bidak-bidak pada papan yang dimungkinkan terjadi.
- *Randomly generated initial states* (dari *genetic algorithm*) akan diperoleh dengan melakukan simulasi permainan dari awal hingga akhir hingga mencapai terminasi.
- *Initial states* (dari *genetic algorithm*) yang dihasilkan dibatasi hanya berjumlah 8.
- Papan akan diberikan nomor 1 - 64 untuk mewakili koordinat setiap kotak.
- Satu *gene* dari *chromosome* didefinisikan sebagai pasangan langkah (pasangan koordinat kotak) yang diambil bot dan lawan. Maka dari itu, pada permainan 28 *round* akan dihasilkan 28 *gene* pada satu *chromosome.*
- Domain dari langkah adalah seluruh koordinat pada papan yang masih kosong pada suatu *state* permainan. Oleh karena itu, domain akan terus berkurang seiring berjalannya permainan.
- Koordinat yang diambil oleh bot akan dilakukan dengan memilih *random* dari domain langkah yang ada, sedangkan koordinat yang diambil oleh lawan akan memaksimalkan *objective function* (dapat dilihat di bagian III dokumen ini). Hal ini dilakukan untuk mensimulasikan langkah yang representatif dari lawan.
- Nilai *fitness function* adalah jumlah bidak yang dimiliki bot di akhir permainan.
- Proses pemilihan *parent chromosomes* yang akan di-*crossover* mengikuti protokol *random selection* dengan setiap parent mendapatkan probabilitas yang proporsional dengan nilai *fitness function* masing-masing.
- *Parent chromosomes* yang dipilih hanya berjumlah 4 dan akan menghasilkan kembali 8 anak *chromosome*.
- Proses *crossover* pada 2 *chromosome* hanya dapat terjadi pada bagian yang memiliki himpunan *gene* yang sama untuk memastikan bahwa tidak ada koordinat yang terisi 2 kali. Hal ini dapat dilakukan dengan membentuk suatu *selection function* maupun membuat *constraint function* yang akan mematikan *chromosome* anak yang tidak mungkin dicapai (memiliki koordinat yang lebih dari 1 kali terisi bidak).

- Setiap *chromosome* anak akan dihitung ulang nilai *fitness function*-nya dengan melakukan simulasi permainan dari awal sampai akhir menggunakan urutan langkah/*gene* dari *chromosome* anak tersebut.
- *Mutation* tidak dapat dilakukan dengan hanya mengubah nilai satu *gene,* melainkan harus dilakukan pada pasangan *gene* dengan menukar posisinya. Hal ini dilakukan demi memenuhi keteraturan dari *state* papan akhir.
- Proses *crossover* dan *mutation* akan terus dilakukan hingga generasi *G* yang memiliki sebuah *emerging winning chromosome pattern*.

Berikut algoritma Genetic yang digunakan dalam bentuk *Pseudocode*.

```
Procedure GeneticAlgorithm():
    Initialize initial population
    For each iteration sampai termination condition terpenuhi:
        Merge parent population dan offspring
        Sort population berdasarkan nilai fitness function
        Prune population untuk mempertahankan jumlah tetap
    Calculate dan return chromosome dengan nilai fitness function tertinggi

Procedure GeneratePopulation(whiteSpots):
    Initialize population sebagai empty list
    For each chromosome dalam jumlah yang ditentukan:
        Create chromosome baru dengan langkah-langkah acak dari daftar
whiteSpots
        Add chromosome baru ke population
    Return population

Procedure GenerateOffsprings(parentPopulation):
    Initialize daftar offsprings sebagai empty list
    Select parent chromosomes yang akan di-cross-over
    For each pasangan parent yang terpilih:
        Perform ordered crossover untuk menghasilkan dua anak kromosom
        Dengan probabilitas tertentu, perform mutation pada satu atau kedua
offspring chromosomes
        Add kedua offspring chromosomes ke daftar offsprings
    Return daftar offsprings

Procedure OrderedCrossover(parent1, parent2):
    Initialize dua offspring chromosomes dengan null list
    Determine rentang yang akan di-cross-over pada chromosome
    Copy gen dari parent pertama ke child pertama dan dari parent kedua ke
child kedua dalam rentang yang ditentukan
    Fill sisa gen dengan gen yang belum ada dari parent yang berlawanan
```

```
untuk mencegah duplikasi
    Return kedua offspring chromosomes

Procedure SelectParentChromosomes(population):
    Initialize daftar chromosomes terpilih sebagai empty list
    Calculate total nilai fitness function dari population
    For each chromosome dalam population:
        Calculate probabilitas relatif chromosome terpilih berdasarkan nilai
fitness function
        Select the chromosome berdasarkan nilai probabilitas yang dihasilkan
    Return daftar chromosomes terpilih
```

Prosedur GeneticAlgorithm diawali dengan menginisialisasi populasi awal. Kemudian memasuki loop yang terus berlanjut sampai kondisi terminasi terpenuhi. Dalam loop tersebut, populasi orangtua dan keturunan digabungkan, diikuti dengan pengurutan populasi berdasarkan nilai *fitness function*. Populasi dipangkas untuk mempertahankan jumlah individu yang tetap. Akhirnya, prosedur menghitung dan mengembalikan kromosom dengan nilai *fitness function* tertinggi.

Prosedur GeneratePopulation mengambil daftar whiteSpots sebagai masukan dan menginisialisasi daftar kosong yang disebut populasi. Ini membuat kromosom baru untuk jumlah iterasi yang ditentukan, di mana setiap kromosom memiliki langkah acak dari daftar whiteSpots. Setiap kromosom yang baru dibuat ditambahkan ke populasi, yang akhirnya dikembalikan.

Dalam prosedur GenerateOffsprings, sebuah daftar kosong bernama offsprings diinisialisasi. Ini memilih kromosom orangtua untuk melakukan *cross-over*. Untuk setiap pasangan orangtua yang dipilih, dilakukan percampuran terurut untuk menghasilkan dua kromosom keturunan. Dengan probabilitas tertentu, juga dilakukan mutasi pada satu atau kedua kromosom keturunan. Kedua kromosom keturunan kemudian ditambahkan ke daftar offsprings, yang akhirnya dikembalikan.

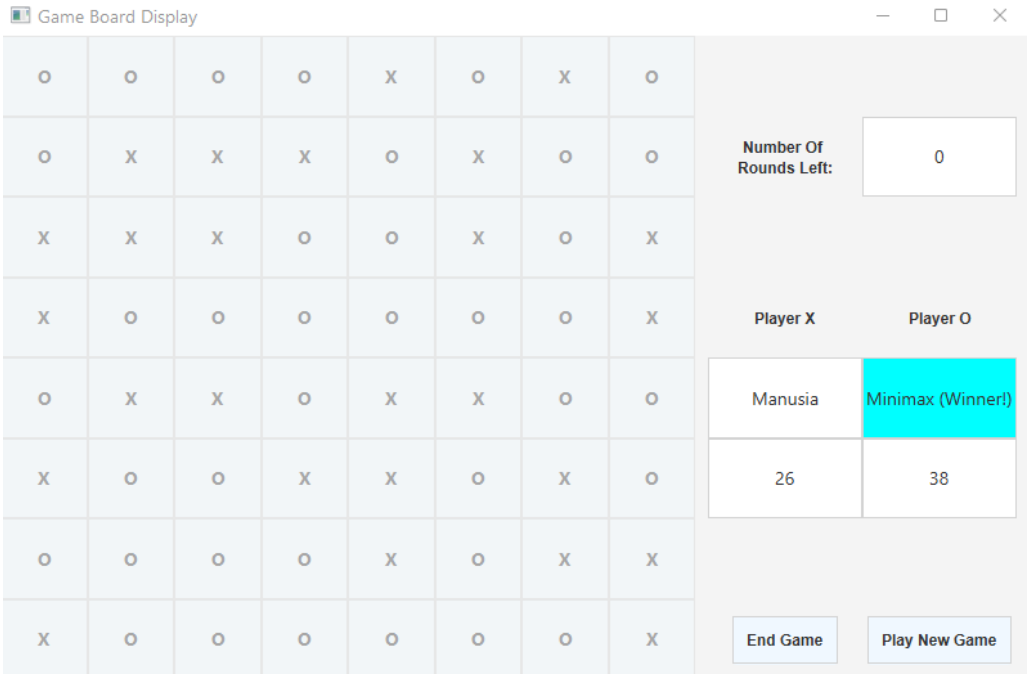
Prosedur OrderedCrossover menginisialisasi dua kromosom keturunan sebagai daftar null dan menentukan rentang yang akan dicampur pada kromosom. Ini menyalin gen dari orangtua pertama ke anak pertama dan dari orangtua kedua ke anak kedua dalam rentang yang ditentukan. Ini mengisi sisa gen dengan gen unik dari orangtua yang berlawanan untuk mencegah duplikasi. Akhirnya, prosedur mengembalikan kedua kromosom keturunan.

Prosedur SelectParentChromosomes menginisialisasi daftar yang disebut kromosom terpilih sebagai daftar kosong. Ini menghitung total nilai *fitness function* dari populasi. Untuk setiap kromosom dalam populasi, dihitung probabilitas relatif kromosom terpilih berdasarkan nilai *fitness function*. Kromosom dipilih berdasarkan nilai probabilitas yang dihasilkan dan ditambahkan ke daftar kromosom terpilih, yang akhirnya dikembalikan.

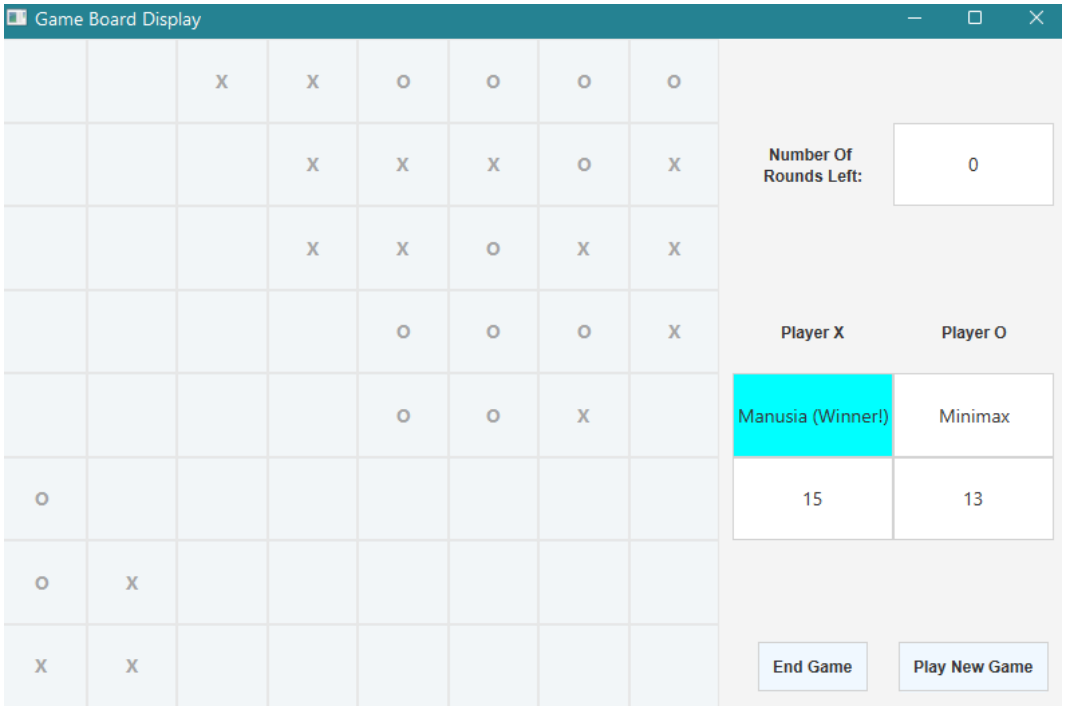
# 3. Eksperimen

Berikut adalah hasil percobaan dari bot yang telah kami rancang :

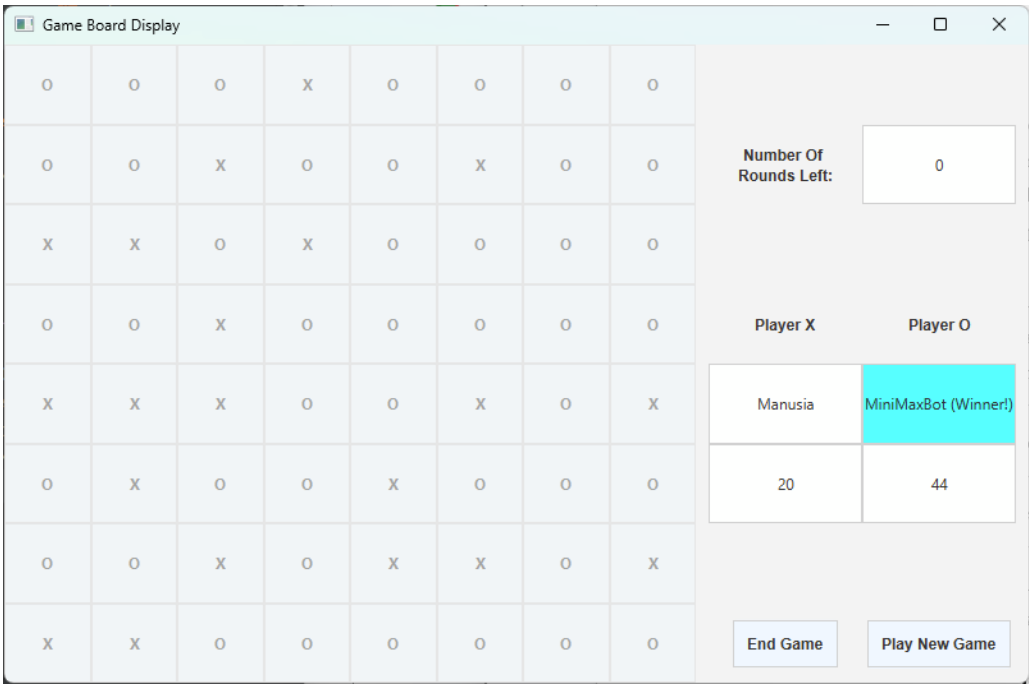
## 3.1. Manusia vs Minimax



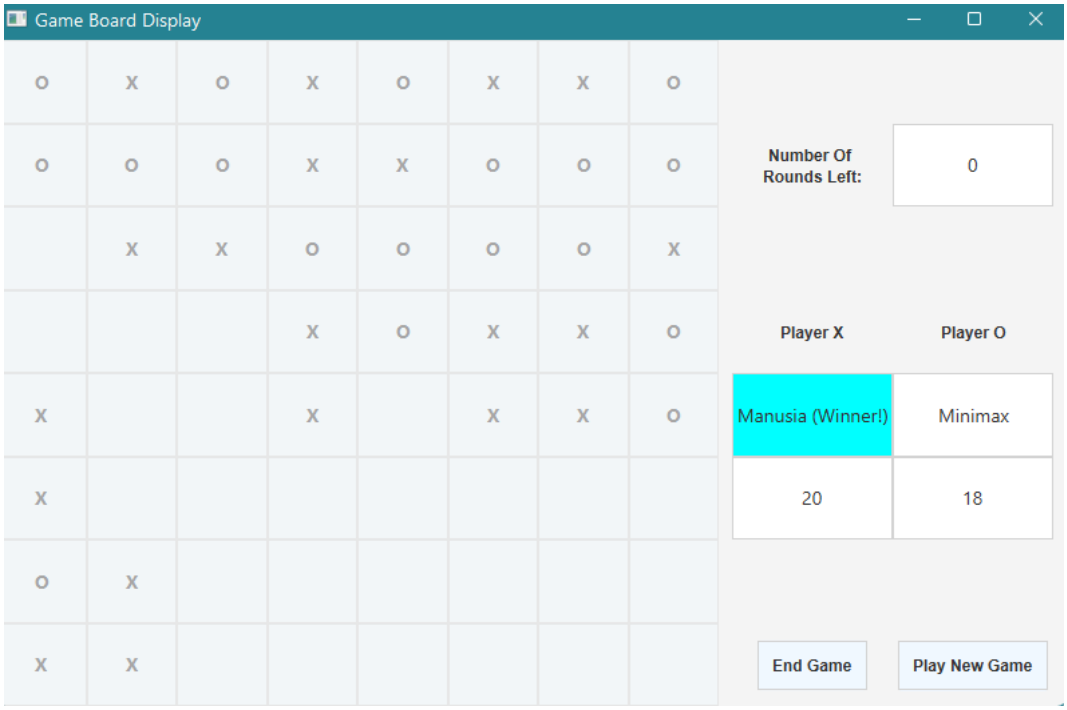

Percobaan 1 (28 ronde)



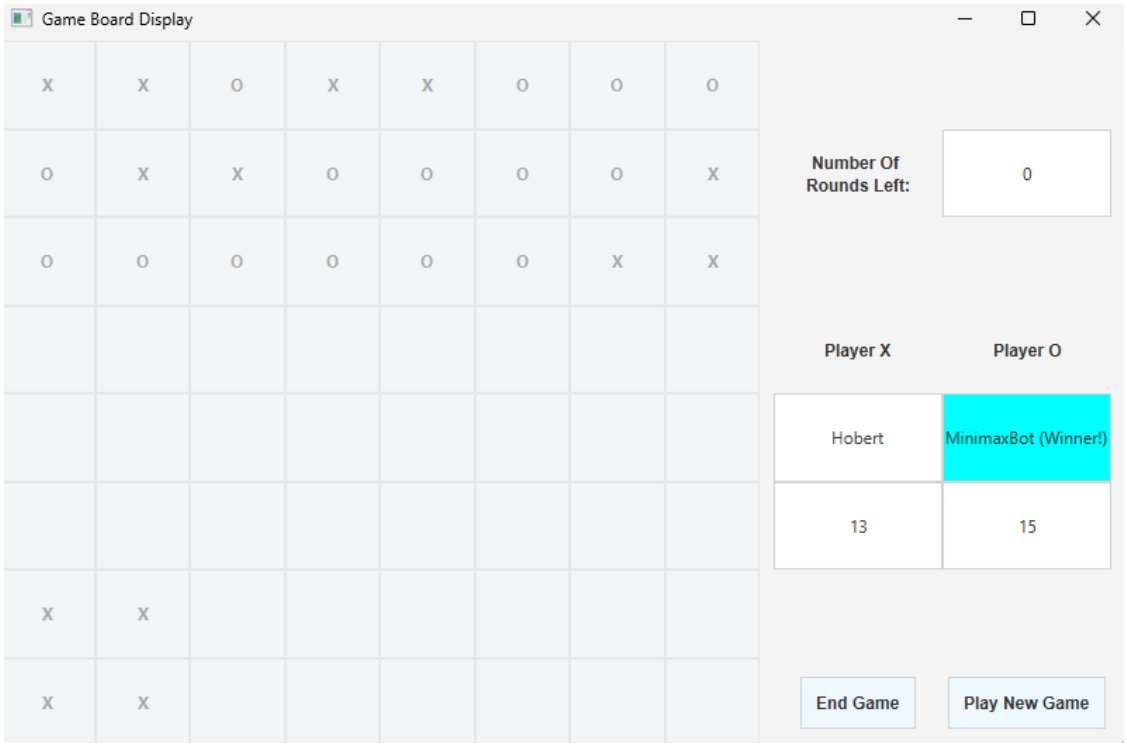

Percobaan 2 (10 ronde)




Percobaan 3 (28 ronde)



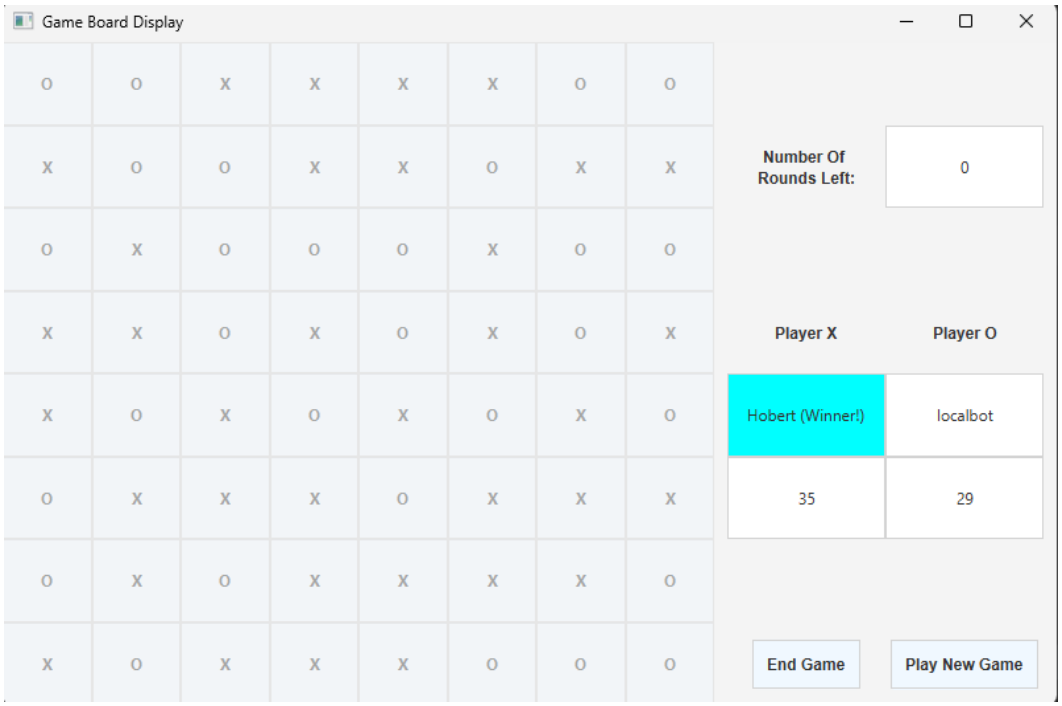

Percobaan 4 (15 ronde)



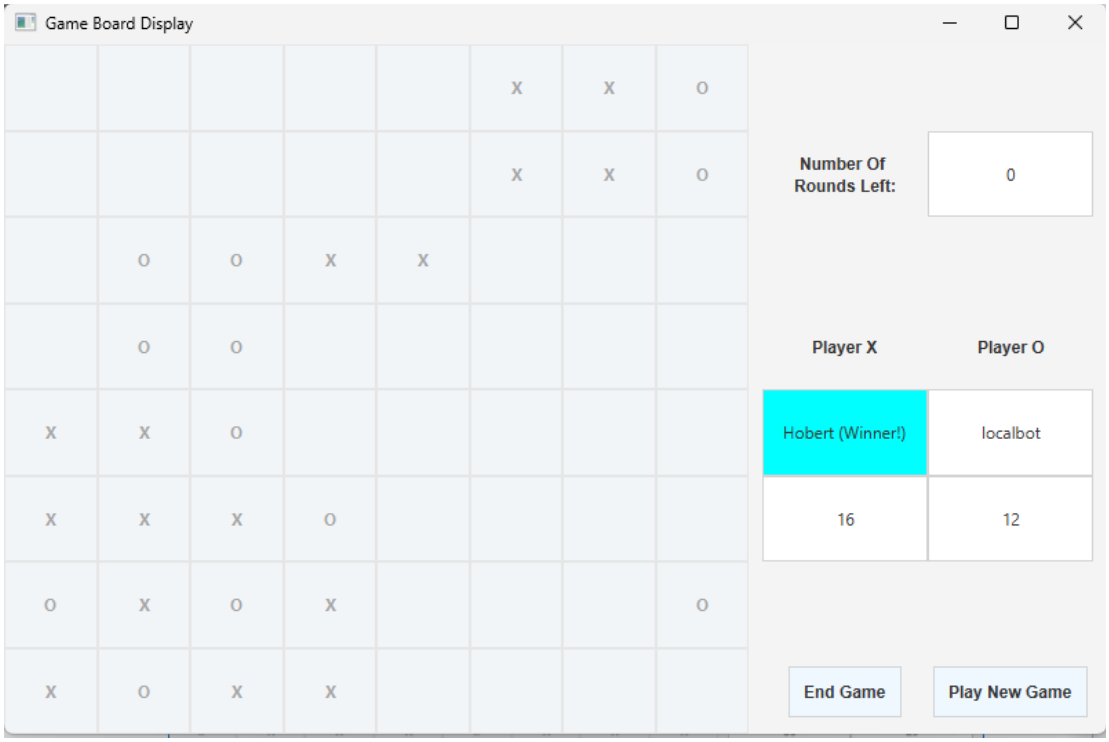

Percobaan 5 (10 ronde)

Analisis : dari 5 kali percobaan, terlihat bahwa bot minimax memenangkan sebagian besar pertandingan. Secara umum, manusia menggunakan pola greedy karena mengincar poin terbaik pada suatu state saja sedangkan bot minimax menggunakan algoritma minimax.

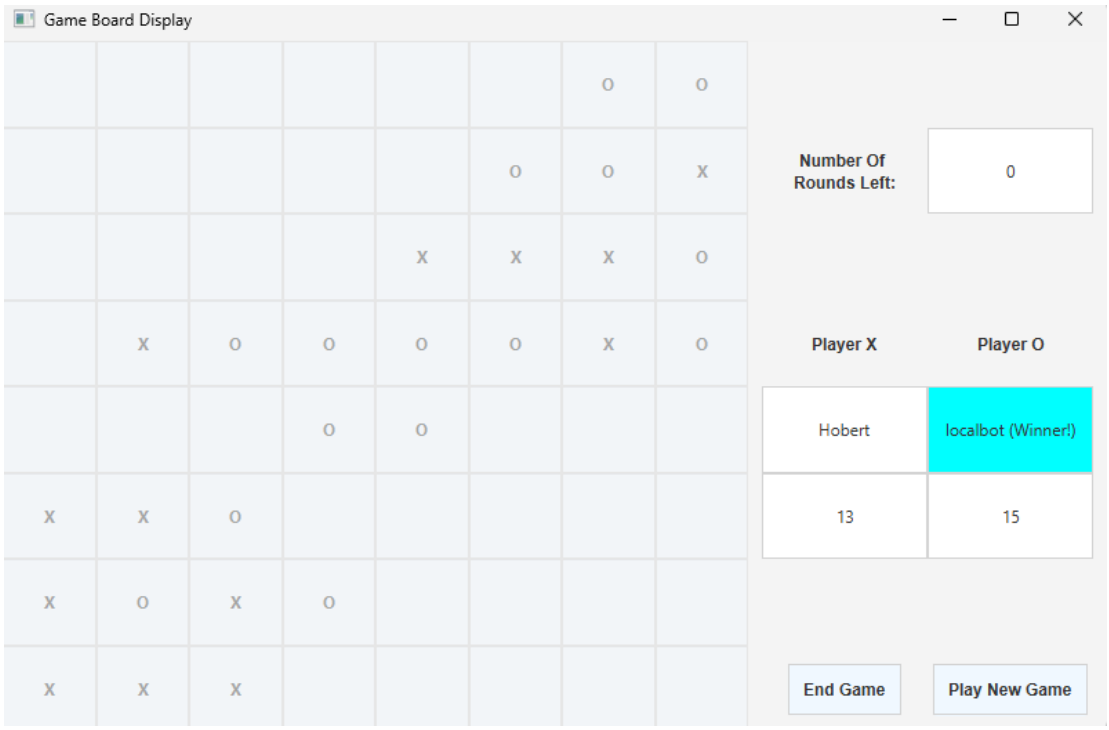
### 3.2. Manusia vs Local Search




Percobaan 1 (28 ronde)



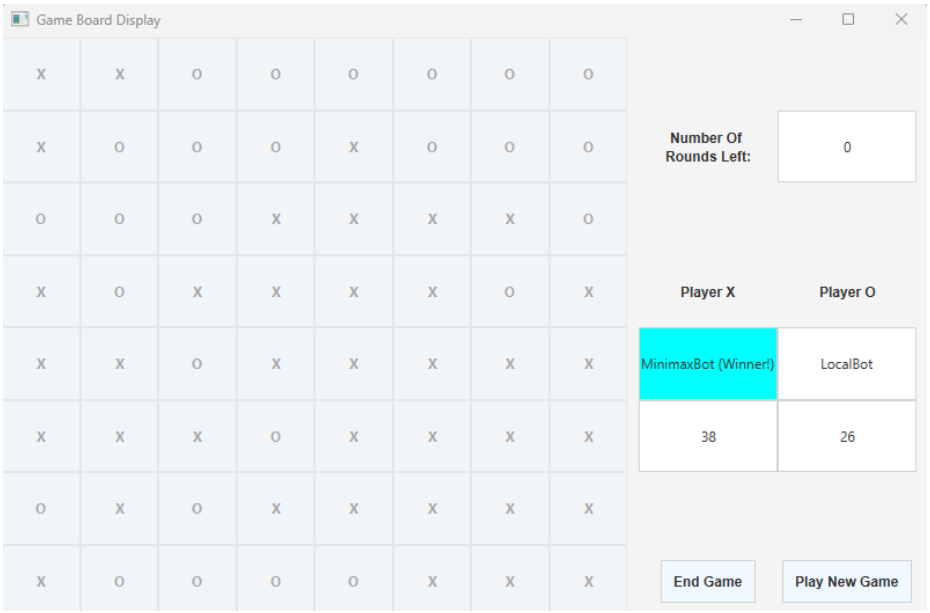

Percobaan 2 (10 ronde)



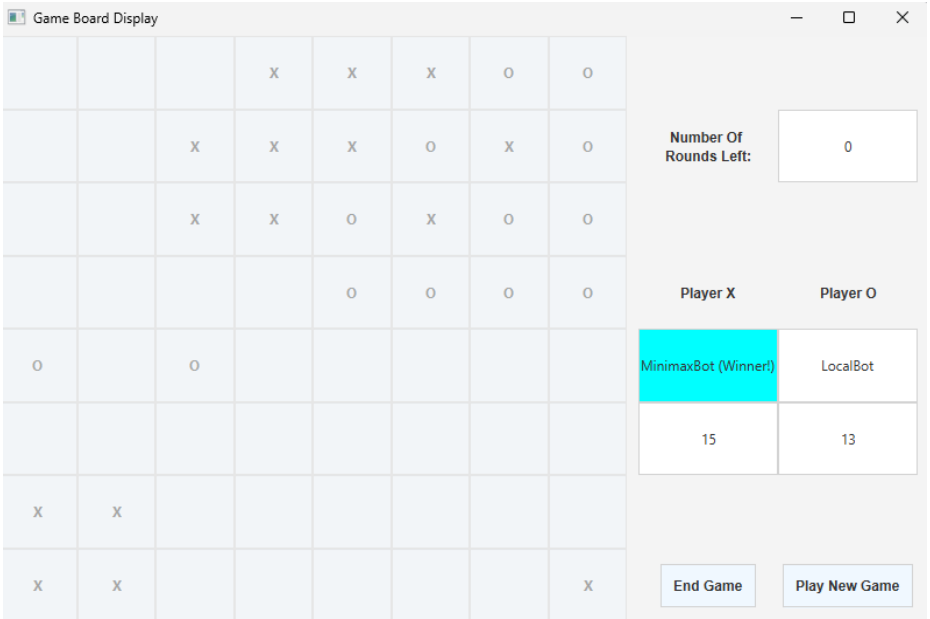

Percobaan 3 (10 ronde)

Analisis : dari 3 kali percobaan, terlihat bahwa manusia memenangkan sebagian besar pertandingan dan bot hanya memenangkan 1 pertandingan. Local search pada kasus ini masih kurang baik dalam pertandingan ini.

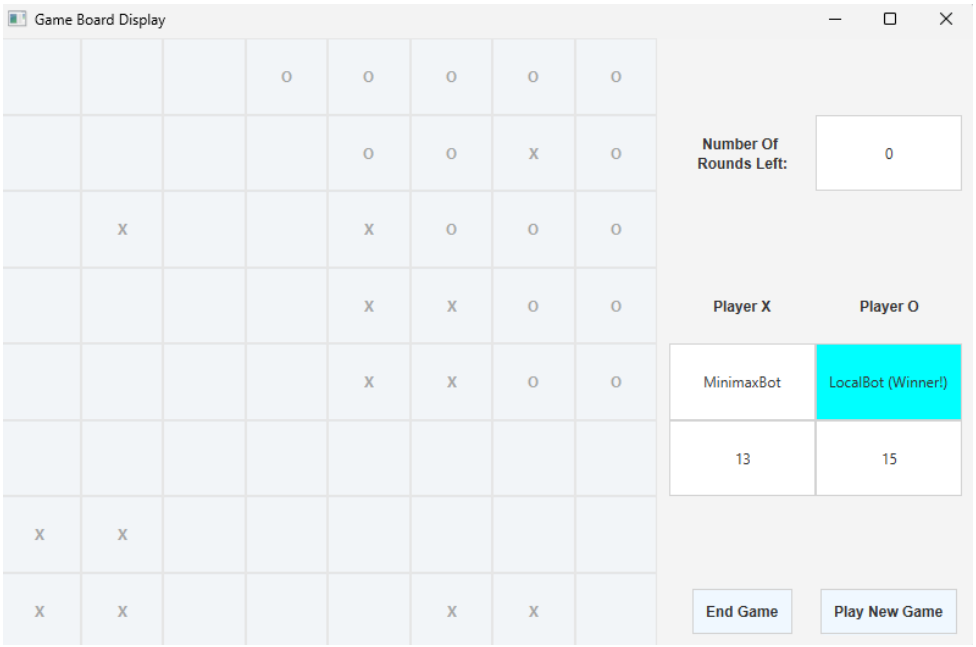
### 3.3. Minimax vs Local Search




Percobaan 1 (28 ronde)

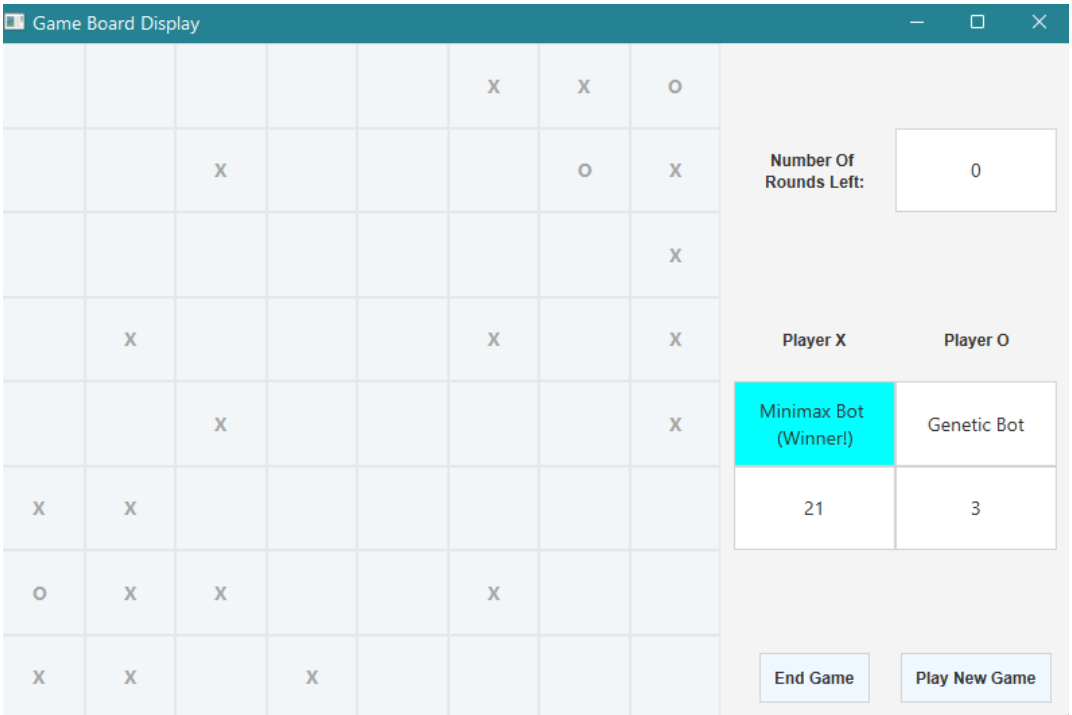



Percobaan 2 (10 ronde)



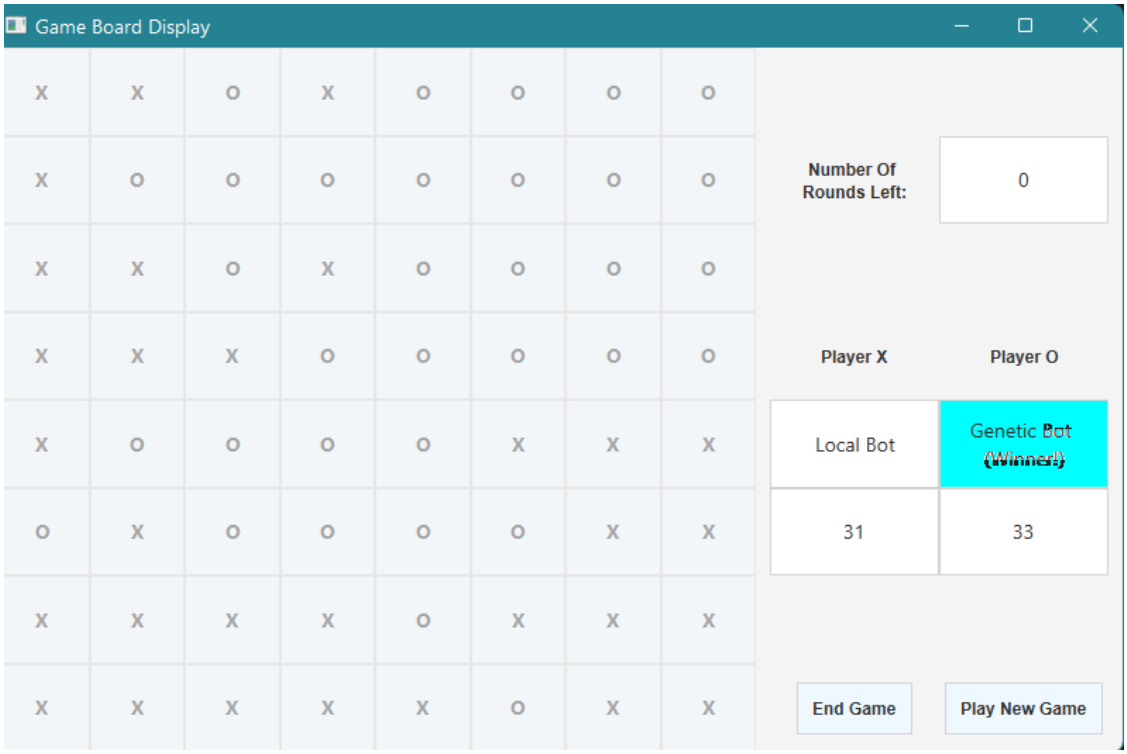

Percobaan 3 (10 ronde)

### 3.4. Minimax vs Genetic Algorithm

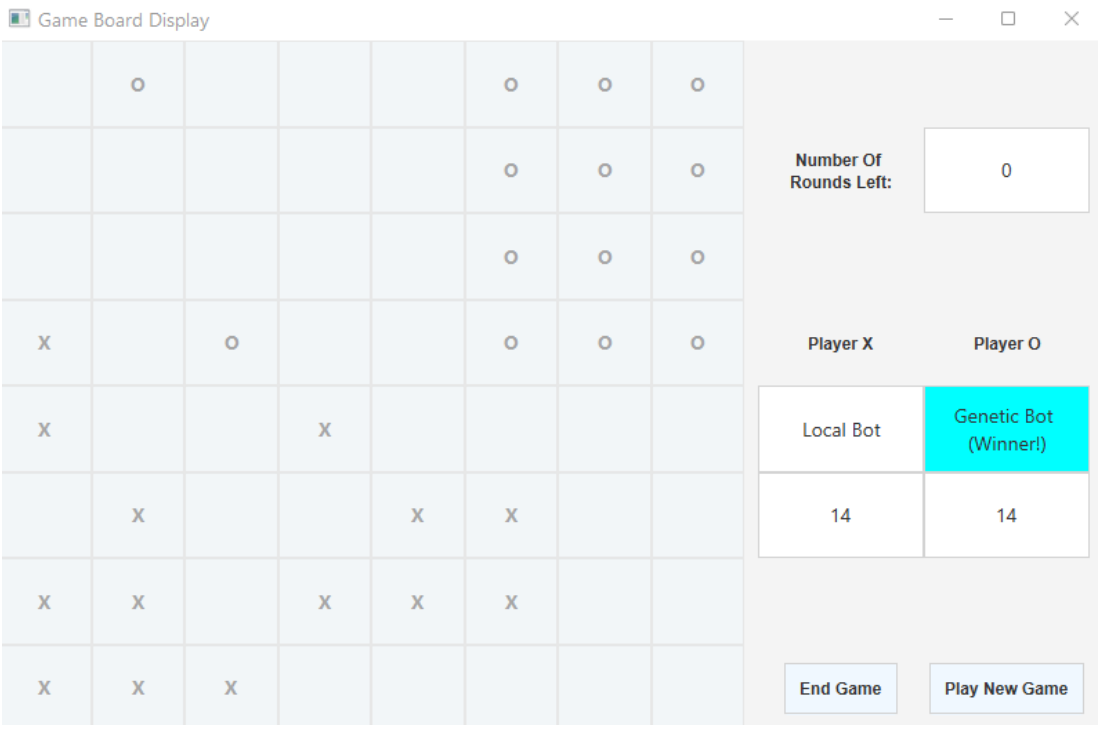



Percobaan 1 (8 ronde)

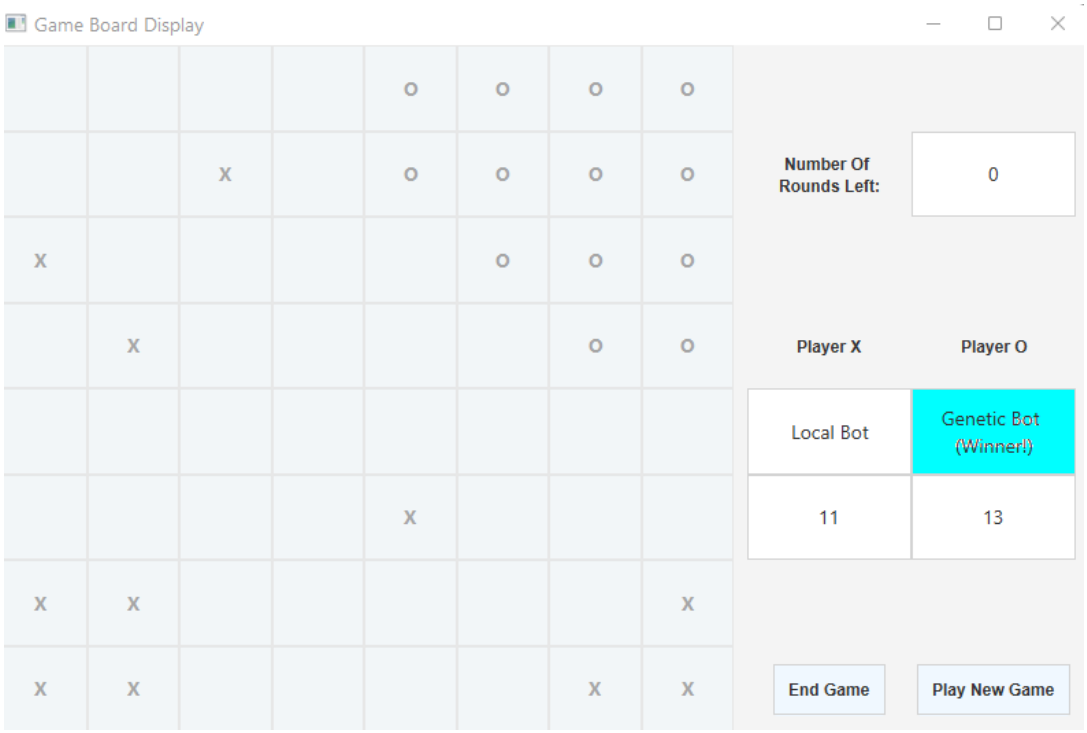
### 3.5. Local Search vs Genetic Algorithm




Percobaan 1 (28 ronde)




Percobaan 2 (10 ronde)




Percobaan 3 (8 ronde)

# 4. Kesimpulan

Algoritma Minimax, Local Search, dan Genetic Algorithm adalah tiga teknik yang berbeda yang digunakan dalam bidang kecerdasan buatan dan optimisasi. Masing-masing memiliki kegunaan dan aplikasi yang berbeda tergantung pada jenis masalah yang dihadapi.

1. Algoritma Minimax:
    - Menawarkan keuntungan dalam situasi permainan yang terstruktur dengan sempurna.
    - Mampu menghasilkan solusi optimal dalam permainan dengan ruang pencarian yang terbatas.
    - Meskipun menawarkan solusi terbaik dibandingkan dengan kedua algoritma lainnya, namun bisa membutuhkan waktu komputasi yang cukup tinggi, terutama pada permainan dengan ruang pencarian yang kompleks, sehingga perlu dilakukan *tweak* terhadap algoritma tersebut.
2. Local Search:
    - Dapat menemukan solusi yang cukup baik, terutama dalam waktu komputasi yang relatif singkat.
    - Tidak sepenuhnya dapat menjamin solusi optimal, terutama ketika ruang pencarian sangat kompleks atau tidak terstruktur dengan baik.
    - Lebih cocok untuk permainan yang memiliki ruang pencarian yang lebih terbatas dan kurang kompleks.
3. Genetic Algorithm:
    - Mampu menemukan solusi yang kompetitif, terutama dalam permainan dengan ruang pencarian yang kompleks.
    - Memerlukan waktu komputasi yang lebih lama dibandingkan dengan local search, tetapi dapat menghasilkan solusi yang lebih baik.
    - Mampu menang dalam beberapa kasus, tetapi tidak seterampil minimax dalam menemukan solusi optimal, terutama pada permainan dengan ruang pencarian yang sangat kompleks.

Dari perbandingan di atas, meskipun algoritma minimax dapat mengalahkan local search dan genetic algorithm secara konsisten, local search memiliki keunggulan dalam waktu komputasi yang lebih cepat. Sementara genetic algorithm, meskipun membutuhkan waktu komputasi yang lebih lama, mampu mengungguli local search dalam beberapa kasus tertentu. Oleh karena itu, pemilihan algoritma tergantung pada kebutuhan spesifik permainan, termasuk seberapa kompleks ruang pencarian dan seberapa pentingnya waktu komputasi.

Kontribusi setiap anggota dalam kelompok.

| | |
|---|---|
| Melvin Kent Jonathan | Bot dengan Genetic Algorithm search |
| Daniel Egiant Sitanggang | Bot dengan algoritma minimax alpha beta pruning |
| Yobel Dean Christopher | Bot dengan algoritma minimax alpha beta pruning |
| Hobert Anthony Jonatan | Bot dengan algoritma local search |