

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



Disusun oleh:

M. Farrel Danendra Rachim	13521048
Naufal Syifa Firdaus	13521050
Louis Caesa Kesuma	13521069
Moh. Aghna Maysan Abyan	13521076

Sekolah Teknik Elektro dan Informatika
Program Studi Teknik Informatika
2023/2024

DAFTAR ISI

DAFTAR ISI.....	2
BAB I DESKRIPSI TUGAS.....	3
BAB II OBJECTIVE FUNCTION.....	5
BAB III ALGORITMA MINIMAX DAN ALPHA BETA PRUNING.....	6
BAB IV ALGORITMA LOCAL SEARCH.....	9
BAB V ALGORITMA GENETIK (GENETIC ALGORITHM).....	11
BAB VI PENJELASAN HASIL ALGORITMA.....	14
6.1. Algoritma Minimax Alpha Beta Pruning.....	14
6.2. Algoritma Local Search.....	17
6.3. Genetic Algorithm.....	19
BAB VII PENJELASAN HASIL PERTANDINGAN.....	26
7.1. Bot minimax vs manusia.....	26
7.2. Bot local search vs manusia.....	27
7.3. Bot minimax vs bot local search.....	28
7.4. Bot minimax vs bot genetic algorithm.....	29
7.5. Bot local search vs bot genetic algorithm.....	30
BAB VIII KONTRIBUSI ANGGOTA.....	32
LAMPIRAN.....	33

BAB I

DESKRIPSI TUGAS

Tugas Besar 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 marka (O atau X) pada papan permainan dengan tujuan memperoleh marka sebanyak mungkin pada akhir permainan (dengan jumlah ronde yang telah ditetapkan).

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

Catatan yang perlu diperhatikan adalah:

- a. Bot yang dibuat diharapkan akan memberikan kinerja terbaik. Untuk algoritma *local search*, berikanlah justifikasi mengapa algoritma yang dipilih merupakan algoritma yang terbaik untuk kasus ini.
- b. Mahasiswa bebas mendesain ulang file Bot, tetapi tidak boleh membuat 2 file Bot yang berbeda.

Keterangan tambahan: File berbeda maksudnya di sini adalah, misal membuat BotMiniMax, BotLocalSearch, atau sejenisnya **tanpa menerapkan prinsip OOP yang benar**. Dengan kata lain, jika BotMiniMax atau BotLocalSearch merupakan turunan / implementasi *interface* dari Bot, desain seperti ini diperbolehkan. Namun, jika keduanya **hanya copy-paste**, yang berarti menyalahi prinsip OOP yang benar, desain seperti ini dilarang.

- c. Penilaian (demo) akan menggunakan 8 ronde dengan waktu bot berpikir maksimal 5 detik. Implementasikan semacam *break* di saat bot berpikir karena program yang

diberikan tidak melakukan *break* secara otomatis saat lewat 5 detik (hal ini juga untuk mengimplementasikan algoritma kalian saat waktu berpikir lebih dari 5 detik, Bot kalian bisa melakukan *fallback plan*, misal random).

- d. Untuk pergerakan Bot yang salah / Error, akan diberikan penalti.
- e. **DILARANG KERAS** mengubah mekanisme game dalam bentuk apa pun, termasuk mekanisme throw Error dari Player/Bot yang salah mengambil pergerakan.
- f. Mengubah alur program yang mengakibatkan kesalahan penilaian dan asisten **tidak bertanggung jawab** atas kerusakan *file* kalian.

BAB II

OBJECTIVE FUNCTION

Objective function adalah fungsi yang dapat menyimpan nilai dalam semua *state* untuk dimaksimalkan atau diminimalkan. *Objective function* berguna untuk melakukan optimalisasi terhadap penentuan gerakan yang akan dilakukan selanjutnya dengan membandingkan satu *state* dengan *state* yang lain.

Objective function yang digunakan dalam membangun bot permainan ini dapat menghitung banyaknya simbol “X” dikurangi banyaknya simbol “O” dalam papan 8 x 8 pada *state* tertentu. Kedua pemain memiliki tujuan permainan yang sama dalam permainan ini, yaitu menaruh marka masing-masing pemain di papan sebanyak mungkin. Oleh karena itu, fungsi objektif tersebut memperoleh heuristik awal berupa banyaknya simbol yang dibuat, namun juga mempertimbangkan ada dua buah pemain dalam permainan ini, sehingga yang dikembalikan fungsi ini adalah selisih antara dua simbol “X” dan “O”.

Berikut rumus fungsi objektif, dengan s sebagai *state* pada saat itu:

$$\text{OBJECTIVE}(s) = X_s - O_s$$

Hasil berupa selisih inilah yang menyebabkan pemain “O” menyelidiki aksi yang dapat meminimumkan nilai $\text{OBJECTIVE}(s)$, dan pemain “X” menyelidiki aksi yang dapat memaksimalkan nilai $\text{OBJECTIVE}(s)$.

Misalnya, dalam *state* permainan di bawah, nilai *objective function* adalah $8-6 = 2$.

							o	o
							x	x
							o	x
						x	x	
x	o	o					o	
x	x							

Gambar 1. *State* dari sebuah *adjacency strategy game*.

BAB III

ALGORITMA MINIMAX DAN ALPHA BETA PRUNING

Permainan *adversarial adjacency strategy* ini memiliki dua buah pemain, yakni pemain manusia dan pemain berupa sebuah bot. Kedua pemain dapat menaruh marka berupa simbol “X” atau “O” ke dalam kotak papan yang kosong. Posisi-posisi marka setelah giliran seorang pemain yang bersifat unik dipanggil sebuah “state”. Setiap *state* memiliki skor yang berfungsi mengukur apakah langkah yang diambil seorang pemain akan membuat pemain tersebut lebih dekat ke tujuannya. Skor ini ditentukan oleh *objective function*.

Untuk mengoptimalkan kerja bot dalam permainan ini, yang masih menggunakan pergerakan secara *random*, akan diimplementasikan algoritma minimax. Algoritma minimax menghitung objektif yang memaksimalkan dan objektif yang meminimalkan. Masing-masing objektif ini diberikan kepada pemain yang berbeda - misal pemain A akan berusaha membuat nilai objektif sebesar mungkin (misal menaruh marka “X” sebanyak mungkin), sedangkan pemain B akan berusaha membuat nilai objektif sekecil mungkin (misal menaruh marka “O” sebanyak mungkin). Objektif masing-masing pemain tersebut penting bagi tiap pemain untuk menentukan di posisi mana bidak akan disimpan.

Algoritma minimax akan memakai pencarian berupa *depth-first search*, yakni melakukan pencarian dari *root node* sampai node paling ujung, yakni *leaf node*. Dalam konteks permainan ini, *root node* adalah kondisi awal permainan, yaitu 4 “X” di pojok kiri bawah, dan 4 “O” di pojok kanan atas. Sedangkan *leaf node* terjadi jika permainan mencapai *terminal state*, yaitu saat papan penuh atau permainan telah mencapai batas ronde yang ditentukan. Pencarian ini dilakukan secara rekursif, dan sepanjang jalan, algoritma akan mengembalikan semua *state* yang bertetangga dengan *state* pada saat itu juga (*state* saat marka “X” atau “O” baru ditaruh). *State* tetangga inilah yang membentuk node di kedalaman selanjutnya dalam pohon permainan (kedalaman dalam konteks permainan adalah *round* permainan). Selain itu, algoritma juga menghitung skor objektif yang dimiliki setiap *state* tersebut, yaitu banyak bidak “X” dikurang banyak bidak “O”. Setelah algoritma mencapai *leaf node*, algoritma akan membandingkan skor *state* tersebut dengan *state* lainnya dengan menggunakan *backtracking* menuju node yang belum dicari. Akhirnya, algoritma akan menggambarkan langkah yang dapat menebak gerakan dan mengalahkan pemain lawan akibat objektif kedua pemain yang berbeda: satu memaksimalkan dan satu meminimalkan.

Namun, algoritma minimax sendiri tidak cukup untuk menyelesaikan permainan ini dengan efisien. Beban algoritma akan menjadi berat akibat ukuran papan 8 x 8 yang besar, sehingga seiring berkembangnya permainan, pohon permainan juga semakin berkembang secara eksponensial. Banyak cabang setiap ronde akan mencapai ratusan ribuan, bahkan jutaan. Hal ini

dapat menyebabkan keputusan menaruh bidak “X” atau “O” yang tidak optimal atau hasil yang tidak diinginkan. Selain itu, algoritma minimax juga menganggap pergerakan kedua pemain bersifat optimal. Kesalahpahaman ini dapat menyebabkan prediksi algoritma menjadi tidak akurat. Oleh karena itu, diperlukan teknik tambahan untuk menyelesaikan permasalahan ini.

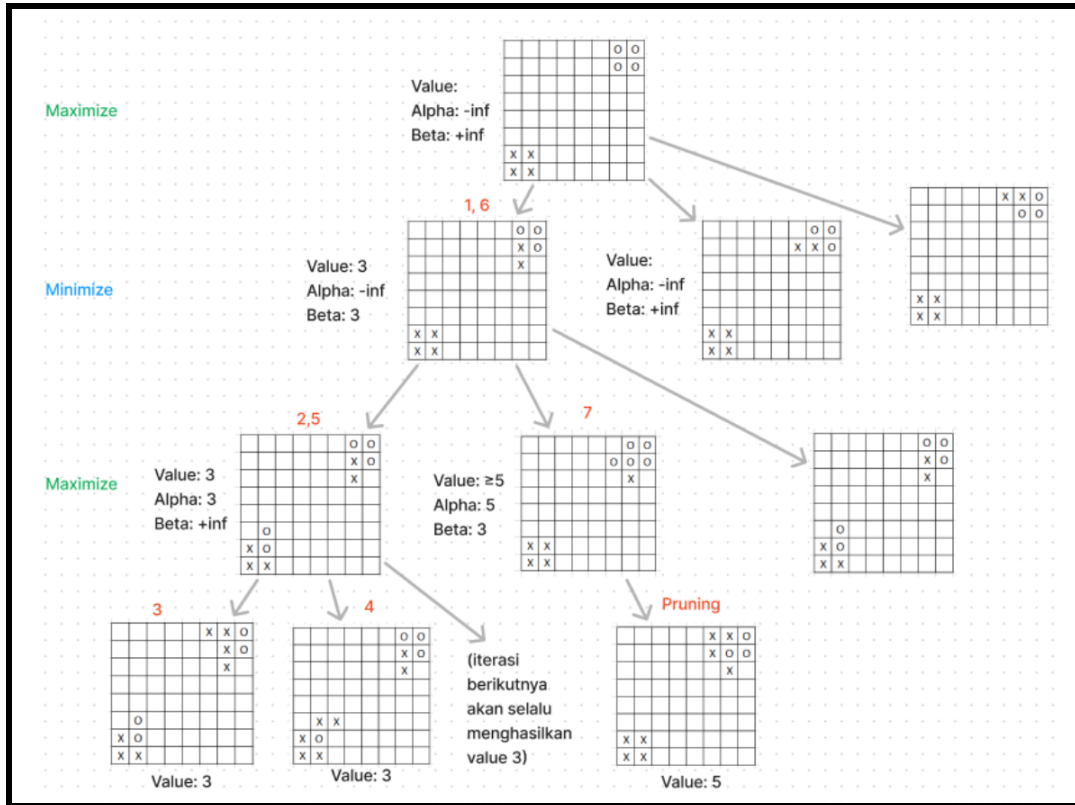
Salah satu solusi yang paling efektif adalah alpha beta pruning. Alpha beta pruning adalah teknik yang mengurangi banyak node yang dijelajahi algoritma minimax, lebih tepatnya cabang yang sudah pasti menyebabkan hasil yang lebih buruk dari gerakan sebelumnya. Dengan proses eliminasi ini, efisiensi algoritma minimax akan meningkat secara drastis tanpa mempengaruhi pembuatan keputusan selanjutnya.

Sesuai namanya, terdapat dua buah parameter dalam alpha beta pruning: alpha, yaitu pilihan nilai terbesar di jalur yang memaksimalkan (skor maksimal), serta beta, yaitu pilihan nilai terkecil di jalur yang meminimalkan (skor minimal). Nilai alpha dan beta akan sering berubah seiring pergerakan algoritma minimax ke dalam pohon permainan. Adapun pruning dilakukan dengan syarat $\alpha \geq \beta$. Jika kondisi ini terjadi pada giliran pemain yang memaksimalkan, cabang yang menyebabkan kondisi ini akan diabaikan dan dieliminasi, karena cabang ini akan menguntungkan pemain lawan yang tujuannya meminimalkan. Begitu juga keadaannya jika kondisi terjadi pada giliran pemain yang meminimalkan, cabang akan menguntungkan pemain lawan yang tujuannya memaksimalkan.

Namun, walaupun dengan metode *pruning*, waktu eksekusi yang diperlukan masih agak lama. Untuk itu, dalam membuat program ini, kami menggunakan kedalaman maksimal 2 untuk menyesuaikan waktu komputasi tiap gerakan yang terbatas sebanyak lima detik. Jika kedalaman ditingkatkan, eksekusi algoritma tidak optimal karena melebihi batas waktu lima detik, sehingga memiliki peluang yang lebih besar untuk mengambil keputusan yang lebih buruk dengan mengambil *fallback plan* gerakan *random*.

Perlu ditekankan juga bahwa gerakan pemain lawan selanjutnya tidak selalu bergerak secara optimal (karena berupa manusia atau bot dengan algoritma yang berbeda). Hal ini menyebabkan komputasi tambahan bagi algoritma minimax karena langkah yang telah didefinisikan menjadi tidak optimal.

Berikut adalah salah satu ilustrasi pohon permainan *adjacency strategy* dengan tiga ronde dengan *bot* di giliran pertama, sebanyak tujuh buah iterasi:



Gambar 2. Ilustrasi pohon dari sebuah *adjacency strategy game* dengan tiga ronde, bot giliran pertama, dan iterasi sebanyak tujuh kali.

BAB IV

ALGORITMA LOCAL SEARCH

Algoritma *minimax* bukanlah satu-satunya teknik yang dapat digunakan untuk menentukan langkah optimal dalam meletakkan marka. Salah satu algoritma lain yang dapat diterapkan adalah algoritma *local search*. Algoritma *local search* yang akan digunakan juga disertakan dengan teknik *hill-climbing*. Algoritma dimulai dari state awal permainan, lalu algoritma akan menerapkan loop yang terus berjalan ke arah nilai objektif yang makin besar (bisa juga makin kecil).

Algoritma *local search* hanya memperdulikan apakah langkah yang diambilnya adalah langkah paling optimal pada saat itu tanpa memperhatikan konsekuensinya. Sehingga sangat umum bagi algoritma *local search* untuk terjebak pada *local optimum*. Selagi algoritmanya tidak terjebak di *local optimum*, algoritmanya masih mungkin untuk mencapai *global optimum*. Namun karena sangat mungkin algoritmanya terjebak di *local optimum*, itu berarti bahwa algoritma *local search* kurang cocok untuk menghasilkan solusi paling optimal.

Walaupun memiliki kekurangan tersebut, algoritma *local search* masih bisa dibilang layak untuk digunakan. Waktu yang digunakan untuk komputasi hasil solusi algoritma *local search* bisa dibilang jauh lebih singkat dibandingkan dengan algoritma lain seperti *minimax*. Jadi, algoritma *local search* sangat cocok untuk digunakan jika ingin menghasilkan solusi yang setidaknya layak dalam waktu yang singkat.

Algoritma *local search* yang akan digunakan pada program ini adalah yang menggunakan *hill-climbing algorithm*. Algoritma *hill-climbing* identik dengan yang namanya *neighbor state*. *Neighbor state* bisa dibilang adalah *state* yang mungkin dicapai dari *current state* dengan melakukan satu langkah. *Hill-climbing algorithm* sendiri memiliki beberapa variasi, antara lain:

1. *Steepest Ascent Hill-Climbing*

Steepest ascent hill-climbing adalah variasi dari algoritma *hill-climbing* yang hanya akan pindah ke *neighbor state* yang nilai objektifnya lebih besar darinya.

2. *Sideways Move Hill-Climbing*

Sideways move hill-climbing adalah variasi dari algoritma *hill-climbing* yang hanya akan pindah ke *neighbor state* yang nilai objektifnya sama atau lebih besar darinya.

3. *Random Restart Hill-Climbing*

Random restart hill-climbing adalah variasi dari algoritma *hill-climbing* yang mencari solusinya mirip seperti variasi *steepest ascent*, namun akan melakukan beberapa kali iterasi ulang sehingga dapat menghasilkan beberapa *neighbor state* yang kemungkinan besar nilainya berbeda-beda. *Neighbor states* tersebut kemudian dipilih yang mana yang paling optimal.

4. *Stochastic Hill-Climbing*

Stochastic hill-climbing adalah variasi dari algoritma *hill-climbing* yang akan mencari solusinya dengan cara men-generate setiap *state*-nya dan kemudian mencari *state* mana yang paling optimal. Proses *generate state* tersebut dapat diulang berkali-kali sesuai dengan keputusan pembuat algoritmanya.

Variasi dari algoritma *hill-climbing* yang akan digunakan untuk permainan ini adalah *steepest-ascent hill-climbing*. *Stochastic hill-climbing* dan *random restart hill-climbing* kurang cocok karena pengulangan yang dilakukan oleh kedua algoritma tersebut dilakukan untuk mencari konfigurasi *final state* dengan nilai *objective* terbesar dan bukan *neighbor state*. Untuk *steepest-ascent hill-climbing* dan *sideways move hill-climbing* bisa dibilang sama kelayakannya untuk permainan ini, dan kedua algoritma tersebut memiliki kekurangannya masing-masing dalam menentukan *neighbor state* mana yang mau dituju. *Sideways move hill-climbing* umumnya digunakan untuk menghindari *local optimum*, namun pada kasus ini hal tersebut kurang dapat dipastikan. Jadi, untuk permainan ini kami memutuskan untuk menggunakan variasi *steepest-ascent* karena dengan *steepest-ascent* sudah pasti *neighbor state* yang dipilih akan lebih baik dari *current state*. Pada pengimplementasiannya sendiri, perbedaan dari *steepest ascent* dan *sideways move* hanyalah *tile* mana yang dipilih. *Steepest ascent* akan memilih *tile* pertama dengan nilai *objective* terbesar, dan *sideways move* akan memilih *tile* terakhir dengan nilai *objective* terbesar.

Berikut adalah langkah-langkah yang dilakukan untuk mencari langkah selanjutnya dengan metode *steepest ascent hill-climbing*.

1. Dilakukan proses *looping* pada papan untuk mencari nilai *objective* dari setiap *tile* pada papan.
2. Kemudian dipilih papan pertama yang memiliki nilai *objective* terbesar.
3. Kemudian algoritmanya akan mengembalikan koordinat papan yang akan dijadikan langkah selanjutnya *bot*.

BAB V

ALGORITMA GENETIK (*GENETIC ALGORITHM*)

Genetic algorithm adalah pendekatan yang meniru proses biologis untuk menghasilkan solusi optimal pada masalah tertentu. Genetic algorithm bekerja dengan cara membangkitkan “successor state” dengan menggabungkan dua “parent state” melalui seleksi, persilangan, dan mutasi. Proses-proses tersebut menghasilkan state yang tidak bertetangga dengan state pada saat itu juga. *Genetic Algorithm* mempunyai beberapa komponen khusus merepresentasikan *problem* yang diberikan seperti berikut.

- State

State adalah sebuah keadaan dalam permasalahan. Secara umum state digambarkan dengan sebuah kode atau kromosom yang dapat merepresentasikan keadaan masalah dalam bentuk yang bisa diolah oleh algoritma.

Dalam implementasi, state adalah sebuah tanda X ataupun O yang ada dalam permainan dengan dilambangkan oleh bilangan koordinat yang dijadikan satu seperti tabel pada gambar dibawah ini.

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	10	11	12	13	14	15	16	17
2	20	21	22	23	24	25	26	27
3	30	31	32	33	34	35	36	37
4	40	41	42	43	44	45	46	47
5	50	51	52	53	54	55	56	57
6	60	61	62	63	64	65	66	67
7	70	71	72	73	74	75	76	77

Gambar 3. *State* dari sebuah *adjacency strategy game* yang digambarkan dalam bentuk tabel dengan isinya merupakan bilangan koordinat yang tersedia.

- Individu

Individu adalah sebuah kumpulan dari state yang dilambangkan dengan barisan state/kromosom yang sebaris dan membentuk sebuah string. Individu ini mewakili sebuah solusi dari permasalahan yang disajikan.

Implementasi individu adalah sebagai urutan aksi yang dilakukan kedua pemain dengan menggunakan koordinat state sebagai tempat diletakkannya tanda pemain. Sebagai contoh jika ada individu berikut (12,1,56,23) maka berarti pemain pertama meletakan tandanya (misal X) di kotak 1,2 pemain kedua meletakan tandanya di kotak 0,1 dan seterusnya.

- **Fitness Function**

Fungsi tujuan digunakan untuk menilai sebuah Individu dari kriteria dan aspek yang ditentukan sebelumnya untuk menentukan apakah Individu sudah memenuhi target state yang diinginkan atau tidak.

Pada bot, *fitness function* menggunakan rumus seperti berikut

$$f(x, y) = \text{jumlah kotak } x - \text{jumlah kotak } x'$$

Dengan kata lain, di state yang sekarang, jumlah kotak pemain dikurangi dengan jumlah kotak lawannya.

- **Population**

Populasi adalah sekumpulan individu yang dikumpulkan dalam satu representasi data untuk kemudian diproses oleh algoritma.

Populasi pada bot dihasilkan secara acak dengan jumlah yang dapat diatur melalui variabel di dalam kelas bot itu sendiri.

Berikut adalah penjelasan untuk setiap tahapan dalam algoritma genetik:

1. Pertama, harus **dibuat satu populasi** yang memiliki individu dengan jumlah yang ditentukan. State/kromosom yang terdapat pada individu di-*generate* secara random.

Pada implementasi, semua populasi di-*generate* secara random dengan banyaknya state/kromosom dalam individu adalah sama dengan jumlah round yang tersisa dan banyaknya individu dalam populasi ditentukan oleh pengguna.

2. Semua Individu dalam populasi tersebut **dinilai dengan *fitness function*** dan diseleksi berdasarkan hasilnya. Semakin besar skor hasil, semakin besar kemungkinan sebuah Individu dipilih.

Proses seleksi, *crossover*, dan mutasi dilakukan per dua pasangan individu pada bot dan diiterasi hingga populasi awal habis dan digantikan dengan yang baru. Bot akan langsung memilih dua individu dengan skor fitness function yang terbesar diantara populasi.

3. Dari hasil seleksi, individu akan saling **di-*crossover* satu sama lain** sehingga state dalam individu-individu tersebut saling ditukar dengan *crossover point* yang telah ditentukan.

Crossover point pada bot adalah setengah dari panjang sebuah individu. Dan dua individu tersebut ditukar state sesuai dengan index yang ditunjukkan oleh *crossover point*.

4. Lalu, selayaknya sebuah reproduksi DNA, **state dalam Individu bisa bermutasi**, berganti menjadi state lain, dengan kemungkinan mutasi dan indeks mutasi yang dapat ditentukan.

Khusus untuk mutasi, pada bot setelah *crossover* mungkin saja ada satu atau lebih state yang berada dalam satu individu. Karena hal tersebut tidak diperbolehkan, maka salah satu dari state tersebut akan dimutasikan menjadi state acak yang belum diisi tanda sebelumnya.

5. Individu-Individu yang sudah melewati tahap ini menjadi populasi yang baru dan proses diulang kembali.

Proses seleksi, *crossover*, dan mutasi akan berlangsung terus hingga jumlah populasi awal habis dan tergantikan oleh populasi hasil pemrosesan yang baru.

6. Algoritma melakukan repetisi terus menerus hingga didapatkan satu Individu yang memenuhi goal state atau solusi dari permasalahan yang diberikan. Jika tidak ditemukan, maka algoritma dapat diatur untuk berhenti setelah iterasi tertentu.

Pada akhirnya keseluruhan proses diulang kembali dengan populasi baru yang dihasilkan hingga mendapat individu dengan fitness skor tertinggi atau jumlah iterasi maksimal tercapai yang juga dapat dikonfigurasi dalam variabel bot.

BAB VI

PENJELASAN HASIL ALGORITMA

6.1. Algoritma *Minimax Alpha Beta Pruning*

Berikut adalah contoh kode implementasi yang mengikuti penjelasan implementasi pada bab sebelumnya.



```
import javafx.scene.control.Button;
import java.util.Objects;

public class Minimax extends Bot{

    @Override
    public int[] move(Button[][] tiles, int roundLeft, String player) {
        int[] move = new int[2];
        int maxScore = -9999;
        int alpha = -9999;
        int beta = 9999;
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {

                if (tiles[i][j].getText().isEmpty()) {
                    Button[][] tiles2 = copyBoardGame(tiles);

                    tiles2 = updatetiles(i, j, tiles2, player);
                    int score = minimax(tiles2, alpha, beta, false, roundLeft,
0, player);

                    if (score >= maxScore) {
                        move[0] = i;
                        move[1] = j;
                        maxScore = score;
                    }
                }
            }
        }
        return move;
    }

    public int minimax(Button[][] tiles, int alpha, int beta, boolean
isMaximizing, int roundLeft, int depth, String player) {
        // max depth: 2
        if (roundLeft == 0 || depth == 2) {
            // There are no rounds left
            int XScore = 0;
            int OScore = 0;
            // Count the total score
            for (int i = 0; i < 8; i++) {
                for (int j = 0; j < 8; j++) {
                    if (tiles[i][j].getText().equals("O")) {
                        OScore++;
                    } else if (tiles[i][j].getText().equals("X")) {
                        XScore++;
                    }
                }
            }
            if (Objects.equals(player, "X")) {
                return XScore - OScore;
            } else {
                return OScore - XScore;
            }
        }
        if (isMaximizing) {
            for (int i = 0; i < 8; i++) {
                for (int j = 0; j < 8; j++) {
                    if (tiles[i][j].getText().isEmpty()) {
                        Button[][] tiles2 = copyBoardGame(tiles);
                        tiles2 = updatetiles(i, j, tiles2, player);
                        int score = minimax(tiles2, alpha, beta, false, roundLeft,
depth + 1, player);
                        alpha = Math.max(alpha, score);
                        if (beta - alpha < 1) break;
                    }
                }
            }
        } else {
            for (int i = 0; i < 8; i++) {
                for (int j = 0; j < 8; j++) {
                    if (tiles[i][j].getText().isEmpty()) {
                        Button[][] tiles2 = copyBoardGame(tiles);
                        tiles2 = updatetiles(i, j, tiles2, player);
                        int score = minimax(tiles2, alpha, beta, true, roundLeft,
depth + 1, player);
                        beta = Math.min(beta, score);
                        if (beta - alpha < 1) break;
                    }
                }
            }
        }
    }
}
```

Gambar 4. Kode implementasi algoritma *minimax alpha beta pruning* (bagian 1)

```

    } else {
        return OScore - XScore;
    }
}

boolean isPruning = false;
// Game is not over yet
if (isMaximizing) {
    // Maximizing
    String pickedPlayer = "";
    if (Objects.equals(player, "X")) {
        pickedPlayer = "X";
    } else {
        pickedPlayer = "O";
    }
    int maxScore = -9999;
    for (int i=0; i < 8; i++) {
        if (!isPruning) {
            for (int j=0; j < 8; j++) {
                if (tiles[i][j].getText().isEmpty()) {
                    Button[][] tiles2 = copyBoardGame(tiles);
                    tiles2 = updateTiles(i, j, tiles2, pickedPlayer);
                    int score = minimax(tiles2, alpha, beta, false,
roundLeft - 1, depth + 1, player);
                    if (score > maxScore) {
                        maxScore = score;
                    }
                    if (score > alpha) {
                        alpha = score;
                    }
                    if (alpha >= beta) {
                        isPruning = true;
                        break;
                    }
                }
            }
        }
        else {
            break;
        }
    }
    return maxScore;
} else {
    // Minimizing
    String pickedPlayer = "";
    if (Objects.equals(player, "X")) {
        pickedPlayer = "O";
    } else {
        pickedPlayer = "X";
    }
    int minScore = 9999;
    for (int i=0; i < 8; i++) {
        if (!isPruning) {
            for (int j=0; j < 8; j++) {
                if (tiles[i][j].getText().isEmpty()) {
                    Button[][] tiles2 = copyBoardGame(tiles);
                    tiles2 = updateTiles(i, j, tiles2, pickedPlayer);
                    int score = minimax(tiles2, alpha, beta, true,
roundLeft - 1, depth + 1, player);
                    if (score < minScore) {
                        minScore = score;
                    }
                    if (score < beta) {
                        beta = score;
                    }
                }
            }
        }
    }
    return minScore;
}

```

Gambar 5. Kode implementasi algoritma *minimax alpha beta pruning* (bagian 2)

```

        if (alpha >= beta) {
            isPruning = true;
            break;
        }
    }
} else {
    break;
}
}
return minScore;
}
}

public Button[][] updateTiles(int i, int j, Button[][] tiles, String player)
{
    tiles[i][j].setText(player);
    if (i + 1 < 8 && !tiles[i + 1][j].getText().equals(player)) {
        tiles[i + 1][j].setText(player);
    }
    if (i - 1 >= 0 && !tiles[i - 1][j].getText().equals(player)) {
        tiles[i - 1][j].setText(player);
    }
    if (j + 1 < 8 && !tiles[i][j + 1].getText().equals(player)) {
        tiles[i][j + 1].setText(player);
    }
    if (j - 1 >= 0 && !tiles[i][j - 1].getText().equals(player)) {
        tiles[i][j - 1].setText(player);
    }
    return tiles;
}
}

```

Gambar 6. Kode implementasi algoritma *minimax alpha beta pruning* (bagian 3)

6.2. Algoritma *Local Search*

Berikut adalah contoh kode implementasi yang mengikuti penjelasan implementasi pada bab sebelumnya.

A screenshot of a code editor with a dark background and light-colored text. The code is in Java and implements a HillClimbing algorithm. It starts with an import statement for javafx.scene.control.Button. Then, a public class HillClimbing extends Bot is defined. Inside the class, there is a method objective_function that takes a 2D array of Button objects (tiles), coordinates (x, y), and a player name as input. The method initializes playerScore to 1 and EnemyScore to 0. It then traverses all tiles. If a tile is empty, it returns -64. Otherwise, it enters a loop where it checks the current tile and its adjacent tiles (up, down, left, right). If the current tile is the player's, it increments playerScore. If it's the enemy's, it increments EnemyScore. If it's an empty tile, it increments playerScore. The method returns playerScore - EnemyScore. There is also an @Override method move that takes a 2D array of Button objects, a roundLeft value, and a player name as input. It initializes maxValue to -64 and returns it.

```
import javafx.scene.control.Button;

public class HillClimbing extends Bot {
    // function for calculating user's score if player insert value at (x, y)
    public int objective_function(Button[][] tiles, int x, int y, String player)
    {
        int playerScore = 1;
        int EnemyScore = 0;
        // traverse all tiles
        if (!tiles[x][y].getText().isEmpty()) {
            return -64;
        } else {
            for (int i = 0; i < 8; i++) {
                for (int j = 0; j < 8; j++) {
                    String tile = tiles[i][j].getText();
                    String enemy;

                    if (player.equals("X")) {
                        enemy = "0";
                    } else {
                        enemy = "X";
                    }

                    if ((i == x-1 && j == y) || (i == x+1 && j == y) || (i == x
&& j == y-1) || (i == x && j == y+1)) {
                        // check the adjacent
                        if (tile.equals(player)) {
                            playerScore++;
                        } else if (tile.equals(enemy)) {
                            // this means the adjacent is the enemy's tile, so
                            naturally that tile became player's
                            playerScore++;
                        }
                    } else {
                        if (tile.equals(player)) {
                            playerScore++;
                        } else if (tile.equals(enemy)) {
                            EnemyScore++;
                        }
                    }
                }
            }
            return playerScore - EnemyScore;
        }
    }

    @Override
    public int[] move(Button[][] tiles, int roundLeft, String player) {
        int maxValue = -64; // assume this because the value cant get any lower
        than this
    }
}
```

Gambar 7. Kode implementasi algoritma *local search* (bagian 1)

```

int[] bestPosition = {-1, -1}; // assume this because this is invalid

// this loop returns the first maximum value found
for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 8; j++) {
        int objectiveValue = objective_function(tiles, i, j, player);

        if (maxValue < objectiveValue) {
            // check first if the coordinates are valid or not
            if (tiles[i][j].getText().isEmpty()) {
                maxValue = objectiveValue;
                bestPosition[0] = i;
                bestPosition[1] = j;
            }
        }

        // System.out.print(objectiveValue);
        // System.out.print(" ");
    }
    // System.out.println("");
}
// System.out.println("-----");
// System.out.print(bestPosition[0]);
// System.out.print(", ");
// System.out.println(bestPosition[1]);

// returns the coordinates for the next action
return new int[] {bestPosition[0], bestPosition[1]};
}
}

```

Gambar 8. Kode implementasi algoritma *local search* (bagian 2)

6.3. Genetic Algorithm

Berikut adalah contoh kode implementasi yang mengikuti penjelasan implementasi pada bab sebelumnya.

```
import javafx.scene.control.Button;

import javax.swing.*;
import javax.swing.plaf.synth.SynthTextAreaUI;
import java.lang.reflect.Array;
import java.util.Objects;
import java.util.Random;
import java.util.Arrays;
import java.util.ArrayList;

public class Genetic extends Bot {
    private final int maxIteration = 100;
    private final int populationCount = 100;
    private ArrayList<Integer> pool = new ArrayList<>();

    public Genetic(){
        for (int i = 0; i < 78; i++) {
            if((i > 7 && i < 10)){
                continue;
            }
            else if ((i > 17 && i < 20)) {
                continue;
            }
            else if ((i > 27 && i < 30)) {
                continue;
            }
            else if ((i > 37 && i < 40)) {
                continue;
            }
            else if ((i > 47 && i < 50)) {
                continue;
            }
            else if ((i > 57 && i < 60)) {
                continue;
            }
            else if ((i > 67 && i < 70)) {
                continue;
            }
            pool.add(i);
        }
        // System.out.println(pool);
    }
    @Override
    public int[] move(Button[][] tiles, int roundLeft, String player) {
        ArrayList<Integer> bannedCell = banCell(tiles);
        // System.out.println("genetic start");
        ArrayList<Integer> chromosome =
```

Gambar 9. Kode implementasi *genetic algorithm* (bagian 1)

```

genetic(roundLeft,bannedCell,maxIteration,tiles);
    return generateMove(chromosome);
}

public ArrayList<Integer> copyList(ArrayList<Integer> originalList){
    ArrayList<Integer> copyList = new ArrayList<>();

    for (Integer element : originalList) {
        // Create new objects for the copy
        Integer copyElement = element.intValue();
        copyList.add(copyElement);
    }

    return copyList;
}

public ArrayList<ArrayList<Integer>>
copyMultilist(ArrayList<ArrayList<Integer>> originalList){
    ArrayList<ArrayList<Integer>> copiedList = new ArrayList<>();

    for(ArrayList<Integer> element : originalList){
        ArrayList<Integer> copyElement = copyList(element);
        copiedList.add(copyElement);
    }

    return copiedList;
}

public ArrayList<Integer> banCell(Button[][] tiles){
    ArrayList<Integer> bannedCell = new ArrayList<>();

    for (int x = 0; x < 8; x++){
        for(int y = 0; y < 8; y++){
            if(Objects.equals(tiles[x][y].getText(), "X") ||
Objects.equals(tiles[x][y].getText(), "0")){
                String cellStr = Integer.toString(x) + Integer.toString(y);
                bannedCell.add(Integer.valueOf(cellStr));
            }
        }
    }

    return copyList(bannedCell);
}

public int[] coordinateFromCell(int cell){
    int[] coordinate = new int[2];

    if (cell < 8){
        coordinate[0] = 0;
        coordinate[1] = cell;
    }
    else{
        String cellStr = Integer.toString(cell);

        coordinate[0] = Integer.valueOf(cellStr.substring(0,1));
        coordinate[1] = Integer.valueOf(cellStr.substring(1));
    }
    return coordinate;
}

```

Gambar 10. Kode implementasi *genetic algorithm* (bagian 2)

```

    }

    public int generateCell(ArrayList<Integer> cellBanned){
        ArrayList<Integer> numberPool = copyList(pool);
        Random random = new Random();

        for(Integer element : cellBanned){
            numberPool.remove(Integer.valueOf(element));
        }

        int cellIndex = random.nextInt(numberPool.size());
        return numberPool.get(cellIndex);
    }

    public ArrayList<ArrayList<Integer>> generatePopulation(int populationCount,
    int genesCount, ArrayList<Integer> bannedCellDef){
        ArrayList<ArrayList<Integer>> population = new ArrayList<>();
        ArrayList<Integer> bannedCell = copyList(bannedCellDef);

        for(int i = 0; i < populationCount; i++){
            ArrayList<Integer> chromosomeTemp = new ArrayList<>();
            // System.out.println("loop in");
            for(int j = 0; j < genesCount; j++){
                // System.out.println("ban " + bannedCell.size());
                int cell = generateCell(bannedCell);
                bannedCell.add(cell);
                chromosomeTemp.add(cell);
                // System.out.println("i="+ i+ "j=" + j + " " + chromosomeTemp);
            }
            // System.out.println("loop out");
            bannedCell.clear();
            bannedCell = copyList(bannedCellDef);
            population.add(copyList(chromosomeTemp));
            chromosomeTemp.clear();
        }
        // System.out.println("inside" + population);
        return population;
    }

    public char[][] updateGameBoard(Button[][] tiles){
        char[][] gameBoard = new char[8][8];

        for (int x = 0; x < 8; x++){
            for(int y = 0; y < 8; y++){
                if(tiles[x][y].getText() == "X"){
                    gameBoard[x][y] = 'X';
                }
                else if (tiles[x][y].getText() == "0") {
                    gameBoard[x][y] = '0';
                }
                else{
                    gameBoard[x][y] = ' ';
                }
            }
        }
    }
}

```

Gambar 11. Kode implementasi *genetic algorithm* (bagian 3)

```

        return gameBoard;
    }

    public int fitnessFunction(ArrayList<Integer> chromosome, char[][] gameBoard){

        //deep copy
        char[][] gameBoardTemp = new char[gameBoard.length][gameBoard[0].length];
        for (int i = 0; i < gameBoard.length; i++) {
            for (int j = 0; j < gameBoard[0].length; j++) {
                gameBoardTemp[i][j] = gameBoard[i][j];
            }
        }

        //game simulation
        for(int i = 0; i < chromosome.size(); i++) {
            int[] coordinate = coordinateFromCell(chromosome.get(i));
            int x = coordinate[0];
            int y = coordinate[1];
            char player = 'O';

            if(i % 2 == 0){
                player = 'X';
            }

            gameBoardTemp[x][y] = player;

            // convert cells
            if(y-1 >= 0 && gameBoardTemp[x][y-1] != ' '){ gameBoardTemp[x][y-1] =
player; }
            if(y+1 <= 7 && gameBoardTemp[x][y+1] != ' '){ gameBoardTemp[x][y+1] =
player; }
            if(x-1 >= 0 && gameBoardTemp[x-1][y] != ' '){ gameBoardTemp[x-1][y] =
player; }
            if(x+1 <= 7 && gameBoardTemp[x+1][y] != ' '){ gameBoardTemp[x+1][y] =
player; }
        }

        // count score
        int xCount = 0;
        int oCount = 0;

        for (int i = 0; i < 8; i++){
            for(int j = 0; j < 8; j++){
                if(gameBoardTemp[i][j] == 'X'){ xCount++; }
                else if(gameBoardTemp[i][j] == 'O'){ oCount++; }
            }
        }
        return oCount-xCount;
    }

    public ArrayList<Integer> selection(ArrayList<ArrayList<Integer>> population,
Button[][] tiles){
        ArrayList<Integer> chosen = new ArrayList<>();
        char[][] gameBoard = updateGameBoard(tiles);
        int maxScore = -999;
        int chosenIndex = 0;
    }

```

Gambar 12. Kode implementasi *genetic algorithm* (bagian 4)

```

        for(int i = 0; i < population.size();i++) {
            int score = fitnessFunction(population.get(i), gameBoard);
            if (maxScore < score) {
                chosen = population.get(i);
                maxScore = score;
                chosenIndex = i;
            }
        }
        //      System.out.println(maxScore);
        //      System.out.println(chosen);
        population.remove(chosen);
        return chosen;
    }

    public ArrayList<Integer>[] crossover(int crossPoint, ArrayList<Integer>
chosen_1, ArrayList<Integer> chosen_2){
        ArrayList<Integer>[] childs = new ArrayList[2];

        ArrayList<Integer> partition_1 = copyList(chosen_1);
        ArrayList<Integer> partition_2 = copyList(chosen_2);
        int temp1 = 0;
        int temp2 = 0;

        for(int i = crossPoint; i < chosen_2.size();i++){
            // swap
            temp1 = partition_1.get(i);
            temp2 = partition_2.get(i);

            partition_1.remove(i);
            partition_2.remove(i);

            partition_1.add(i, temp2);
            partition_2.add(i, temp1);
        }

        childs[0] = partition_1;
        childs[1] = partition_2;

        return childs;
    }

    public void mutation(ArrayList<Integer>[] childs, ArrayList<Integer>
bannedCellDef){
        int foundIndex = -1;
        boolean found = false;
        // ganti gene yang sama dengan sebelumnya

        for (int childIndex=0 ; childIndex < 2 ; childIndex++){
            ArrayList<Integer> bannedCell = copyList(bannedCellDef);
            bannedCell.addAll(childs[childIndex]);

            for (int i = 0; i < childs[childIndex].size(); i++) {
                int currentElement = childs[childIndex].get(i);

                for (int j = i + 1; j < childs[childIndex].size(); j++) {
                    if (currentElement == childs[childIndex].get(j)) {
                        found = true;
                        foundIndex = j;
                    }
                }
            }
        }
    }

```

Gambar 13. Kode implementasi *genetic algorithm* (bagian 5)

```

        break;
    }
}
if(found){
    childs[childIndex].remove(foundIndex);
    // System.out.println("in found " + bannedCell);
    childs[childIndex].add(foundIndex, generateCell(bannedCell));
    found = false;
}
}
bannedCell = bannedCellDef;
}
}

public ArrayList<Integer> genetic(int roundLeft, ArrayList<Integer>
bannedCellDef,int maxIter, Button[][] tiles){
    // System.out.println("in");
    ArrayList<Integer> bestChromosome = new ArrayList<>();
    int bestScore = -999;
    ArrayList<Integer> chosen_1 = new ArrayList<>();
    ArrayList<Integer> chosen_2 = new ArrayList<>();
    ArrayList[] childs = new ArrayList[2];
    ArrayList<ArrayList<Integer>> population =
generatePopulation(populationCount, roundLeft,bannedCellDef);

    // System.out.println("Initialize " + roundLeft + bannedCellDef);
    // System.out.println("ban count " + bannedCellDef.size());
    // System.out.println(bannedCellDef);

    for(int iteration=0; iteration < maxIter; iteration++){
        ArrayList<ArrayList<Integer>> newPopulation = new ArrayList<>();
        // System.out.println("Number Iteration: " + iteration );
        // find best individual
        for(int i = 0; i < population.size();i++){
            int score = fitnessFunction(population.get(i),
updateGameBoard(tiles));
            if(score >= bestScore){
                bestScore = score;
                bestChromosome = copyList(population.get(i));
            }
        }
        // System.out.println("best " + bestChromosome + bestScore);
        // System.out.println(population);
        if(bestScore == 63){
            break;
        }
        int a = 0;
        while(!population.isEmpty()){
            // System.out.println("Number Selection: " + a);
            // System.out.println("inside" + population);
            // System.out.println("selection in" + i);
            chosen_1 = selection(population, tiles);
            chosen_2 = selection(population, tiles);
            // System.out.println("selection out" + chosen_1 + chosen_2);

            // cross over masih belum memperhitungkan langkah yg sama
            // System.out.println("cross over in" + chosen_1 + chosen_2);

```

Gambar 14. Kode implementasi *genetic algorithm* (bagian 6)


```

//      // cross over masih belum memperhitungkan langkah yg sama
//      System.out.println("cross over in" + chosen_1 + chosen_2);
//      childs = crossover(chosen_1.size()/2, chosen_1, chosen_2);
//      System.out.println("cross over out" + childs[0] + childs[1]);
//      mutation(childs, bannedCellDef);
//      System.out.println("mutation out");

//      newPopulation.add(childs[0]);
//      newPopulation.add(childs[1]);
//      a++;
//  }
//      System.out.println("Loop 2");
//      System.out.println("before" + population);
//      population = copyMultiList(newPopulation);
//      System.out.println("after" + population);
//  }
//      System.out.println("return " + bestChromosome);
//      return bestChromosome;
//  }

//  public int[] generateMove(ArrayList<Integer> chromosome){
//      System.out.println(chromosome);
//      return coordinateFromCell(chromosome.get(0));
//  }
}

```

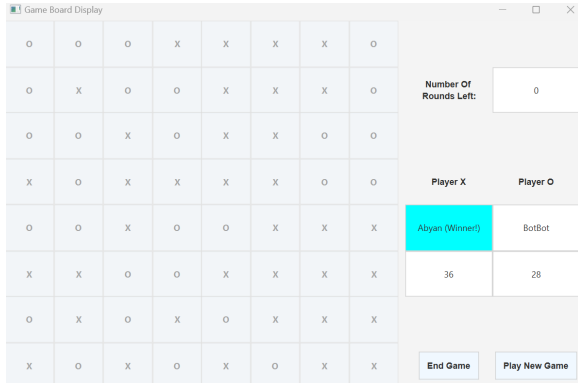
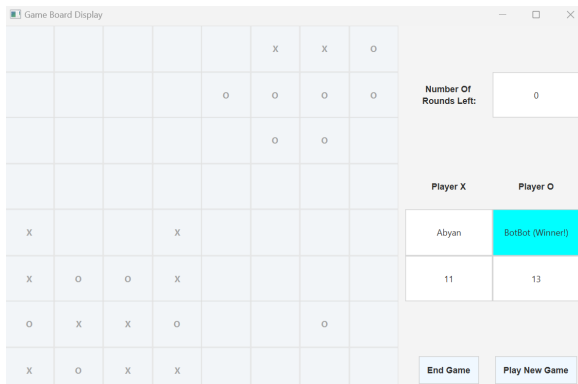
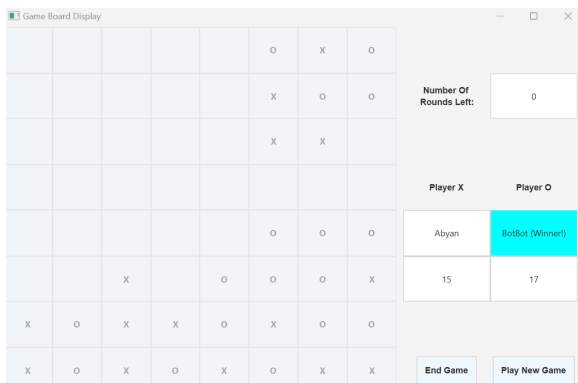
Gambar 15. Kode implementasi *genetic algorithm* (bagian 7)

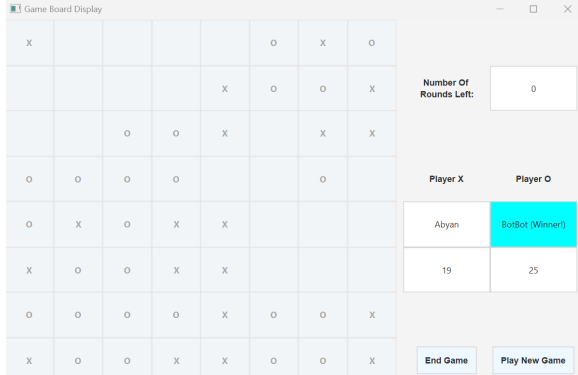
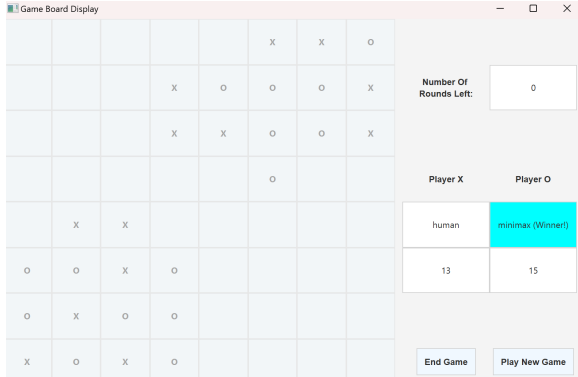
BAB VII

PENJELASAN HASIL PERTANDINGAN

7.1. Bot minimax vs manusia

Tabel 1. Hasil Pertandingan Bot Minimax vs Manusia.

Percobaan	Ronde	Screenshot	Skor akhir	Pemenang
1	28		36-28	Manusia
2	8		11-13	Bot
3	12		15-17	Bot

4	18		19-25	Bot
5	10		13-15	Bot

Jumlah kemenangan manusia: 1

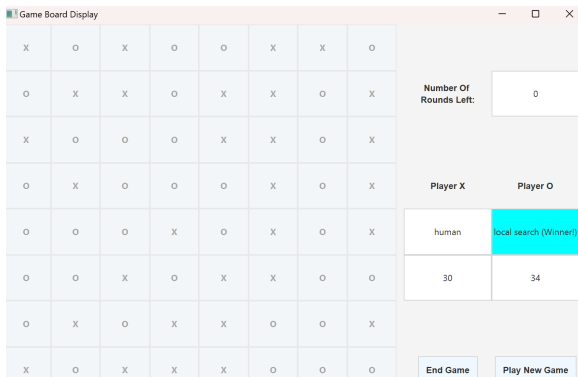
Jumlah kemenangan bot: 4

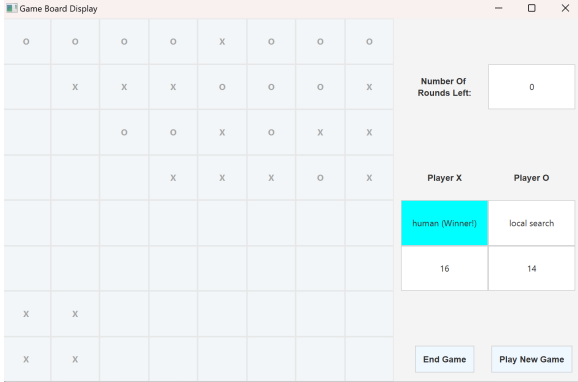
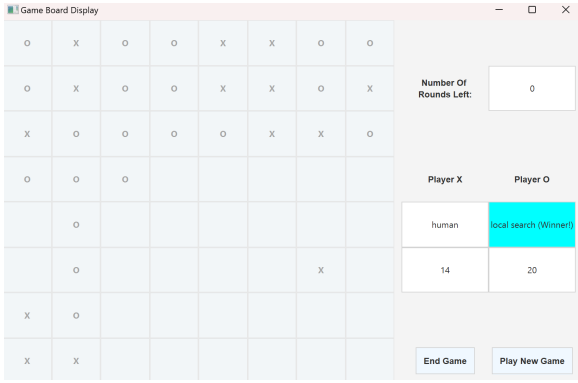
Persentase kemenangan manusia: 20%

Persentase kemenangan bot: 80%

7.2. Bot local search vs manusia

Tabel 2. Hasil Pertandingan Bot Local Search vs Manusia.

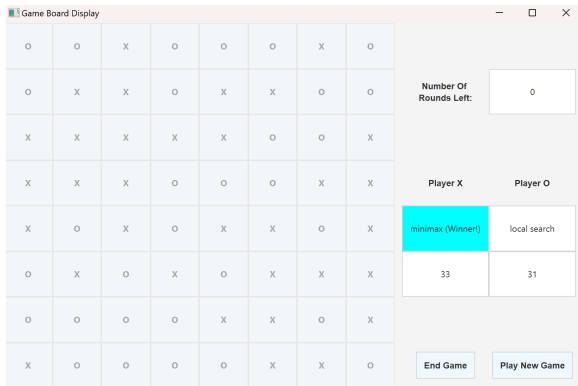
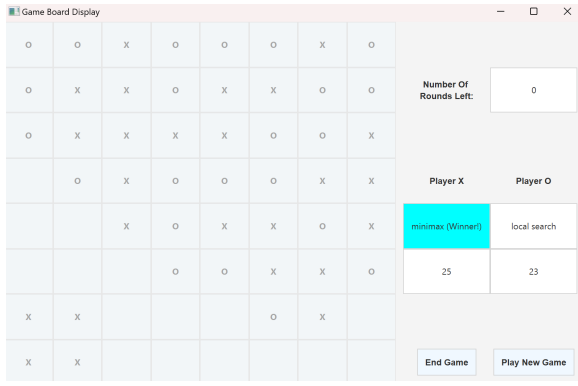
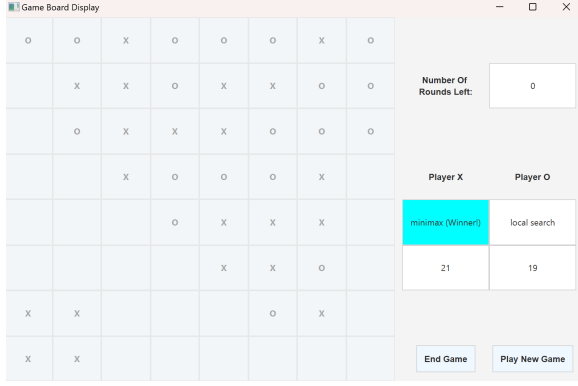
Percobaan	Ronde	Screenshot	Skor akhir	Pemenang
1	28		30-34	Bot

2	11		16-14	Human
3	13		14-20	Bot

Jumlah kemenangan manusia: 1
Jumlah kemenangan bot: 2
Persentase kemenangan manusia: 33,3%
Persentase kemenangan bot: 66,7%

7.3. Bot minimax vs bot local search

Tabel 3. Hasil Pertandingan Bot Minimax vs Bot Local Search.

Percobaan	Ronde	Screenshot	Skor akhir	Pemenang
1	28		33-31	Minimax
2	20		25-23	Minimax
3	16		21-19	Minimax

Jumlah kemenangan bot minimax: 3

Jumlah kemenangan bot *local search*: 0

Persentase kemenangan bot minimax: 100%

Persentase kemenangan bot *local search*: 0%

7.4. Bot minimax vs bot genetic algorithm

Tabel 4. Hasil Pertandingan Bot Minimax vs Bot Genetic Algorithm.

Percobaan	Ronde	Screenshot	Skor akhir	Pemenang
1	28		33-31	Bot <i>Minimax</i>
2	14		23-13	Bot <i>Minimax</i>
3	10		18-10	Bot <i>Minimax</i>

Jumlah kemenangan bot minimax: 3

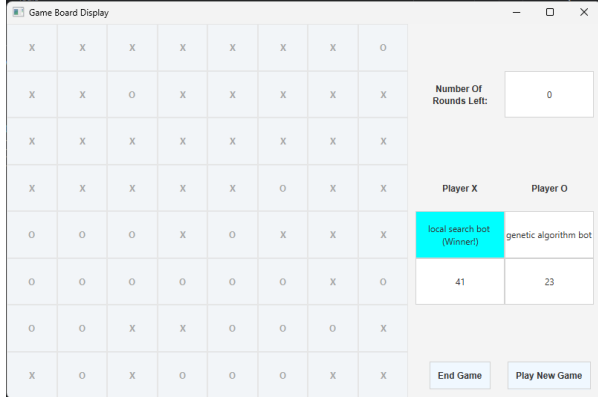
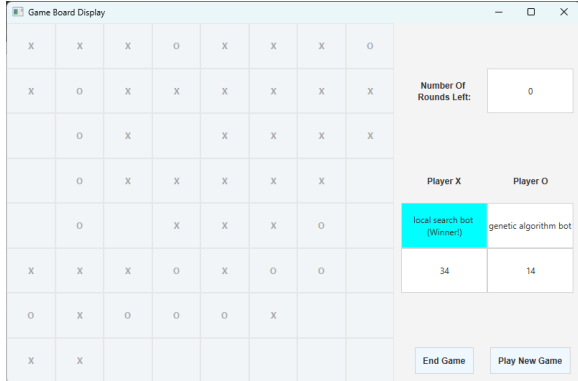
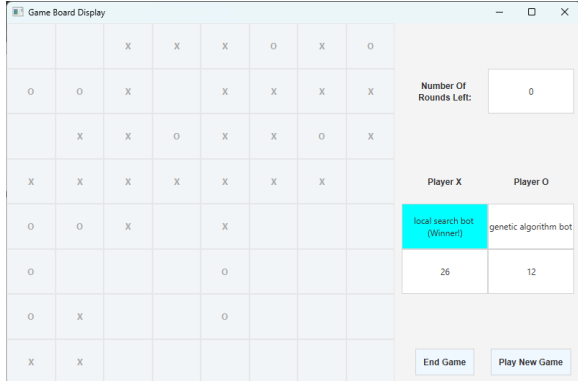
Jumlah kemenangan bot *genetic algorithm*: 0

Persentase kemenangan bot minimax: 100%

Persentase kemenangan bot *genetic algorithm*: 0%

7.5. Bot local search vs bot genetic algorithm

Tabel 5. Hasil Pertandingan *Bot Local Search* vs *Bot Genetic Algorithm*.

Percobaan	Ronde	Screenshot	Skor akhir	Pemenang
1	28		41-23	Bot <i>local search</i>
2	20		34-14	Bot <i>local search</i>
3	15		26-12	Bot <i>local search</i>

Jumlah kemenangan bot *local search*: 3

Jumlah kemenangan bot *genetic algorithm*: 0

Persentase kemenangan bot *local search*: 100%

Persentase kemenangan bot *genetic algorithm*: 0%

BAB VIII

KONTRIBUSI ANGGOTA

Tabel 6. Daftar NIM, nama, serta tugas yang dilakukan oleh tiap anggota kelompok.

NIM	NAMA	TUGAS
13521048	M. Farrel Danendra Rachim	Minimax, update UI
13521050	Naufal Syifa Firdaus	Genetic algorithm, update UI
13521069	Louis Caesa Kesuma	Local search, update UI
13521076	Moh. Aghna Maysan Abyan	Laporan, test case

LAMPIRAN

Link github: <https://github.com/Breezy-DR/adversarial-adjacency-strategy-game.git>