IF3170 Intelegensi Buatan

Tugas Besar 1

Minimax Algorithm and Alpha Beta Pruning in Adjacency Strategy Game

Dibuat Oleh:

13521077 Husnia Munzayana

13521132 Dhanika Novlisariyanti

13521155 Kandida Edgina Gunawan

13521156 Brigita Tri Carolina

Program Studi Teknik Informatika

Sekolah Teknik Elekto dan Informatika

Institut Teknologi Bandung

Jalan Ganesha 10, Bandung, Jawa Barat 40132

DAFTAR ISI

1. Penjelasan Objective Function

Objective function yang kami gunakan pada permainan Adjacency Strategy Game dengan pencarian Minimax Alpha Beta Pruning dan Local Search terbagi menjadi dua kondisi, yaitu kondisi ketika langkah yang dipertimbangkan merupakan langkah terakhir permainan serta kondisi ketika langkah yang dipertimbangkan bukan langkah terakhir pemain, sehingga perlu mempertimbangkan kemungkinan langkah-langkah berikutnya.

Pada kondisi pertama, yaitu ketika *objective function* digunakan untuk menentukan langkah paling optimal pada langkah terakhir pada permainan tersebut, maka objective function ditentukan berdasarkan kondisi akhir papan yang dimungkinkan. Perhitungan nilai utilitas pada kondisi ini, dilakukan dengan rumus sebagai berikut :

$$h = skor\ akhir\ Bot - skor\ akhir\ lawan$$

Perumusan *objective function* ini didasarkan pada *objective* atau tujuan utama dalam permainan. *Objective* utama yang ingin dicapai pemain dalam permainan *Adjacency Strategy Game* adalah untuk mendapatkan skor semaksimal mungkin dan melebihi skor lawan. Jika dianalisis lebih lanjut, *objective* yang perlu dicapai adalah mendapatkan skor yang lebih baik dari pemain lama dan dapat dioptimalkan dengan menentukan langkah sehingga memaksimalkan selisih atau jarak skor dengan skor lawan. Pada langkah terakhir, pemain cukup mempertimbangkan kondisi papan setelah meletakkan marka. Pemain tidak perlu mempertimbangkan kemungkinan peletakan marka selanjutnya karena langkah yang ia lakukan merupakan langkah terakhir dalam permainan.

```
else { // `roundLeft` == 0, langkah terakhir
            String oppChar;
            if(botChar.equals("O")) oppChar = "X";
            else oppChar = "O";
                        return countPlayerScore(button, botChar) -
countPlayerScore(button, oppChar);
        }
```

Pada kondisi kedua, objective function diperhitungkan dengan mempertimbangkan kondisi papan saat ini dan kemungkinan kondisi permainan pada langkah-langkah berikutnya. yang adalah pertama-tama diberikan nilai utilitas 4 untuk setiap sel yang berisi tanda, kemudian ketika terdapat sel kosong di sisi atas, kanan, kiri, dan bawah sel tersebut maka nilai utilitas dikurangi sebanyak sel kosong.

$$h = 4 - x$$

Catatan:

h : nilai utilitas untuk setiap sel O

x : banyak sel yang tidak terisi tanda di sisi atas, kiri, kanan, dan bawah dari sel O

Nilai utilitas total untuk suatu kondisi:

$$y = \sum_{i=1}^{n} (4 - x_i)$$

Dengan:

y : nilai utilitas total suatu kondisi

xi : banyak sel kosong di sekitar sel O ke-i

n: banyak sel yang berisi tanda O

Berikut ini merupakan beberapa contoh penghitungan nilai utilitas

					X	O	O
					X	O	O
						X	O
							X

X	X						
X	X						

Banyak sel dengan tanda O: 5

Pada ilustrasi di atas, tidak terdapat sel kosong di sekitar (atas, bawah, kanan, kiri) sel yang bertanda O.

Dengan demikian, nilai utilitas kondisi ini : 5 x 4 = 20

						O	O
						O	O
						X	X
						X	
X	X						
X	X						

Banyak sel dengan tanda O : 4

Banyaknya sel kosong di sekitar:

1. Sel O merah : 0
2. Sel O kuning : 1
3. Sel O biru: 1
4. Sel O hijau : 0

Dengan demikian, nilai utilitas kondisi ini: 4 x 4 - 2 = 14

Alasan digunakannya *objective function* dengan rumus seperti di atas adalah kami ingin memberikan nilai utilitas yang maksimal (4) hanya pada sel tanda O yang tidak mungkin berubah tanda lagi. Syarat suatu sel tidak mungkin mengalami perubahan tanda lagi adalah ia harus diapit oleh sel-sel lain yang berisi tanda. Sel-sel yang sudah bertanda O tapi masih memiliki peluang untuk direbut oleh lawan akan diberikan nilai utilitas yang lebih rendah . Nilai utilitas untuk setiap sel bertanda O bergantung oleh banyaknya sel kosong di sekitar sel tersebut. Untuk menghitung nilai utilitas dari suatu langkah, akan dihitung jumlah total nilai utilitas tiap selnya.

```
      public int objectiveFunction(int[] currentPost, Button[][] button)
{
         int nilaiAwal = 4;
         int nilaiMin = 0;
         int count = 1;

         for (int i = 0; i <= 7; i++) {
             for (int j = 0; j <= 7; j++) {
                  int[][] positions = {{i, j + 1}, {i - 1, j}, {i + 1, j},
{i, j - 1}};
                 if (button[i][j].getText().equals("O")) {
                     count++;
                     for (int[] post : positions) {
                         int x = post[0];
                         int y = post[1];
                         if (x >= 0 && x <= 7 && y >= 0 && y <= 7) {
                                          if (x != currentPost[0] && y !=
currentPost[1] && button[x][y].getText().equals("")) {
                                 nilaiMin--;
                             }
                         }
                     }
                 }
             }
         }
```

```
            int[][] positions = {{currentPost[0], currentPost[1] + 1},
{currentPost[0]-    1,    currentPost[1]},    {currentPost[0]    +    1,
currentPost[1]}, {currentPost[0], currentPost[1] - 1}};
        for (int[] post : positions) {
            int i = post[0];
            int j = post[1];
            if (i >= 0 && i <= 7 && j >= 0 && j <= 7) {
                if (button[i][j].getText().equals("X")) {
                    count++;
                      int[][] positions2 = {{i, j + 1}, {i - 1, j}, {i +
1, j}, {i, j - 1}};
                    for (int[] post2 : positions2) {
                        int x = post2[0];
                        int y = post2[1];
                        if (x >= 0 && x <= 7 && y >= 0 && y <= 7) {
                                        if (x != currentPost[0] && y !=
currentPost[1] && button[x][y].getText().equals("")) {
                                nilaiMin--;
                            }
                        }
                    }
                }
                else if (button[i][j].getText().equals("")) {
                    nilaiMin--;
                }
            }
        }

        int result = nilaiAwal * count + nilaiMin;

        return result;
    }
```

Gambar 1.1 Objective Function

2. Implementasi MinMax Alpha Beta Pruning

Pencarian menggunakan *Minimax Alpha-Beta Pruning* pada permainan *Adjacency Strategy Game* ini digunakan untuk menentukan aksi terbaik yang dapat dilakukan *agent* sehingga dapat menempatkan marka pada posisi yang tepat sehingga memiliki marka yang lebih banyak dari marka lawan di akhir permainan. Dalam proses pencarian menggunakan *Minimax Alpha-Beta Pruning*, terdapat beberapa hal terkait *Game Search Problem* dan *Minimax Alpha-Beta Pruning* yang perlu diperhatikan, baik dalam proses penentuan solusi maupun dalam pembentukan *game tree.* Aspek-aspek terkait pembentukan *game tree* antara lain :

- Pada pohon pencarian atau *game tree*, setiap *node* melambangkan *game state* dan *edge* atau garis melambangkan aksi yang mungkin dilakukan pemain.
- Cabang (*branch/adjacent node)* merepresentasikan setiap kemungkinan peletakan marka O maupun X, sesuai dengan giliran pemain, pada papan permainan.
- Simpul daun merepresentasikan kondisi akhir papan permainan secara lengkap ketika permainan telah berakhir (*complete configuration*).
- Kedalaman *game tree* paling banyak sebesar 2 kali batas ronde yang ditentukan, atau dapat lebih sedikit jika kondisi papan sudah penuh.

Dalam proses pencarian solusi paling optimum dengan algoritma *Minimax Alpha-Beta Pruning,* perlu diperhatikan aspek-aspek berikut:

- ***Initial state*****:** Kondisi awal permainan adalah ketika pada papan 8x8 sudah terdapat 4 marka X di pojok kiri bawah dan 4 marka O di pojok kiri atas.
- ***Players***: Pada permainan ini, terdapat 2 *players.* Kedua *players* dapat berupa Bot atau salah satu menjadi Bot dan *player* lain dimainkan secara manual oleh *user.* Dalam penerapan algoritma, *agent* atau Bot dapat bertindak untuk meletakkan marka O maupun X.
- ***Action***: Dalam setiap gilirannya, pemain dapat meletakkan sebuah marka secara optimal dan setiap marka lawan yang berada di sekitar marka pemain (di atas. bawah, kanan, dan/atau kiri) akan menjadi milik pemain.
- ***Result*****(s,a)**: Fungsi transisi yang menggambarkan terjadinya perubahan state ketika pemain melakukan sebuah aksi (meletakkan marka).

- **Nilai utilitas(s)** yang digunakan berdasarkan objective function yang telah dijelaskan pada poin 1.
- ***Terminal Test***: Permainan akan berhenti ketika papan penuh atau mencapai batas ronde permainan.
- **Alpha (α)** adalah nilai utilitas terbaik hingga saat ini yang didapatkan dari proses maksimasi (nilai tertinggi, *at least*)
- **Beta (β)** adalah nilai utilitas terbaik hingga saat ini yang didapatkan dari proses minimasi (nilai terendah, *at most*)
- Dalam setiap langkahnya, *agent* atau bot akan meletakkan markanya pada kotak yang dapat menghasilkan nilai utilitas paling besar (maksimal). Disisi lain, secara bergantian pemain lawan meletakkan markanya pada kotak yang dapat menghasilkan nilai utilitas paling kecil (minimum).

Proses pencarian menggunakan minimax alpha-beta pruning dapat dilakukan dengan langkah berikut:

- Dimulai dari simpul akar yang merupakan initial state, dilakukan pembangkitan kemungkinan solusi secara Depth First Search.

```
        public int[] alphaBetaSearch(Button[][] button, int
roundLeft, boolean isBotFirst, String botChar) {
        Button[][] dumbButton = copyButton(button);
            resultMinMax res = maxValue(dumbButton, roundLeft,
Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY, isBotFirst,
botChar);
        return res.getNextPosition();
    }
```

- Penelusuran dimulai dari simpul anak paling kiri, secara rekursif akan dilakukan penelusuran ke dalam hingga menemukan simpul daun (*leaf node*). Pada permainan ini, simpul yang dibangkitkan adalah semua kemungkinan posisi yang dapat di-*assign* marka yaitu semua posisi kotak kosong pada papan. Selama proses pencarian simpul daun, alpha pada node yang di *expand* di-*assign* nilai sementara $-\infty$ dan beta di-*assign* nilai sementara ∞.

```
public resultMinMax maxValue(Button[][] button, int roundLeft,
double alpha, double beta, boolean isBotFirst, String botChar) {
        List<int[]> emptyPos = generate_empty_cell(button);
        if (roundLeft == 0 || emptyPos.size() == 0) {
                                    return new resultMinMax(null,
objectiveFunction(null, button, roundLeft, botChar));
        }

        int newRoundLeft = roundLeft;
        if (!isBotFirst) newRoundLeft--;

        double currentValue = Double.NEGATIVE_INFINITY;
        int[] currentMove = null;
        String oppChar;
        if (botChar.equals("O")) oppChar = "X";
        else oppChar = "O";

        long startTime = System.currentTimeMillis();
        for (int[] possiblePos : emptyPos) {
                                        List<int[]> changeLabel =
changeAdjacent(possiblePos, button, botChar);

            // Change label button

button[possiblePos[0]][possiblePos[1]].setText(botChar);
            if (changeLabel.size() > 0){
                for(int[] adjPos:changeLabel){

button[adjPos[0]][adjPos[1]].setText(botChar);
                }
            }
```

```
public resultMinMax minValue(Button[][] button, int roundLeft,
double alpha, double beta, boolean isBotFirst, String oppChar) {
        List<int[]> emptyPos = generate_empty_cell(button);
```

```
        String botChar;
        if (oppChar.equals("O")) botChar = "X";
        else botChar = "O";

        if (roundLeft == 0 || emptyPos.size() == 0) {
                                    return   new   resultMinMax(null,
objectiveFunction(null, button, roundLeft, botChar));
        }

        int newRoundLeft = roundLeft;
        if (isBotFirst) newRoundLeft--;

        double currentValue = Double.POSITIVE_INFINITY;
        int[] currentMove = null;

        long startTime = System.currentTimeMillis();
        for (int[] possiblePos : emptyPos) {
                                        List<int[]>   changeLabel   =
changeAdjacent(possiblePos, button, oppChar);

            // Change label button

button[possiblePos[0]][possiblePos[1]].setText(oppChar);
            if (changeLabel.size() > 0){
                for(int[] adjPos:changeLabel){

button[adjPos[0]][adjPos[1]].setText(oppChar);
                }
            }
```

- Ketika menemukan simpul daun, dilakukan perhitungan untuk menentukan nilai utilitas pada state tersebut. Nilai utilitas tersebut kemudian menjadi nilai utilitas sementara pada setiap node parent.

```
        if (roundLeft == 0 || emptyPos.size() == 0) {
                                return new resultMinMax(null,
objectiveFunction(null, button, roundLeft, botChar));
        }
```

```
        if (roundLeft == 0 || emptyPos.size() == 0) {
                                return new resultMinMax(null,
objectiveFunction(null, button, roundLeft, botChar));
        }
```

- Pencarian dilanjutkan hingga dapat menentukan nilai simpul daun lain.

```
            resultMinMax result = minValue(button, newRoundLeft,
alpha, beta, isBotFirst, oppChar);
```

```
            resultMinMax result = maxValue(button, newRoundLeft,
alpha, beta, isBotFirst, botChar);
```

- Selama proses rekursif, dalam setiap kedalaman pohon pencarian, nilai utilitas dari node parent akan selalu dievaluasi.
  - Jika node tersebut memiliki peran sebagai agent atau MAX dan terdapat simpul anak yang memiliki nilai lebih besar dari nilai utilitas node sementara, maka nilai utilitas node tersebut akan diganti dengan nilai maksimal yang baru (Update nilai α pada node tersebut).

```
            double nextValue = result.getValue();
            if (nextValue > currentValue) {
                currentValue = nextValue;
                currentMove = possiblePos;
                alpha = Math.max(alpha, currentValue);
            }
```

- Jika node tersebut memiliki peran sebagai lawan atau MIN dan terdapat simpul anak yang memiliki nilai lebih kecil dari nilai utilitas node sementara, maka nilai utilitas node tersebut akan diganti dengan nilai minimal yang baru. (Update nilai β pada node tersebut).

```
            double nextValue = result.getValue();
            if (nextValue < currentValue) {
                currentValue = nextValue;
                currentMove = possiblePos;
                beta = Math.min(beta, currentValue);
            }
```

- Pemotongan atau penghentian pencarian pada node atau branch tertentu dapat dilakukan ketika diketahui bahwa nilai dari node tersebut tidak akan lebih baik daripada nilai alpha (untuk nilai MAX) dan beta (untuk nilai MIN) yang telah didapatkan sekarang, sehingga agent tidak akan memilih nilai dari branch tersebut.

```
if (currentValue <= alpha) return new resultMinMax(currentMove,
currentValue);

if (currentValue >= beta) return new resultMinMax(currentMove,
currentValue);
```

- Pencarian dikatakan mencapai terminal state atau selesai ketika setiap node yang tidak di pruning telah di expand.

```
        if (roundLeft == 0 || emptyPos.size() == 0) {
                                return new resultMinMax(null,
objectiveFunction(null, button, roundLeft, botChar));
        }
```

```
        if (roundLeft == 0 || emptyPos.size() == 0) {
                                return new resultMinMax(null,
objectiveFunction(null, button, roundLeft, botChar));
        }
```

Dari proses tersebut, didapatkan bahwa pada proses pembangunan *game tree*, terdapat $(8\times8) - (4\times2) = 56$ kemungkinan posisi penempatan marka oleh pemain yang mendapat giliran pertama.. Kemudian, untuk setiap kemungkinan tersebut terdapat 55 kemungkinan posisi yang dapat dipilih oleh pemain lain. Penelusuran *game tree* secara lengkap diperlukan kompleksitas sebesar $O(b^m)$ dengan b adalah banyaknya kemungkinan aksi pada setiap node dan m adalah maksimum kedalaman game tree. Hal ini tentunya memerlukan waktu komputasi yang cukup lama dan cenderung tidak *feasible* untuk diimplementasikan. Namun, dengan optimasi melalui algoritma *Minimax Alpha-Beta Pruning* memungkinkan dilakukannya pemotongan/*cut* sub-pohon pencarian yang tidak akan memberikan nilai yang lebih baik dari nilai sekarang. Hal ini dapat mengurangi kompleksitas sehingga algoritma yang dihasilkan dapat lebih praktis dan memungkinkan untuk diimplementasikan/dilakukan (*feasible*). Adanya batasan ronde yang dapat diaplikasikan menjadi kedalaman maksimal (*maximum depth*) dari game tree juga dapat diterapkan sehingga algoritma bisa lebih *feasible*, lebih praktis karena penelusuran tidak perlu dilakukan hingga kedalaman game tree maksimum.

Meskipun dalam praktiknya, proses pencarian solusi optimal dengan *Minimax Alpha-Beta Pruning* ini masih memerlukan waktu yang cukup lama ketika ada banyak ronde dalam permainan. Hal ini disebabkan oleh kedalaman (*depth*) pohon permainan adalah sejumlah dua kali lipat dari jumlah ronde permainan. Dengan tingkat kedalaman yang tinggi ini, pencarian solusi menjadi sangat lambat. Untuk mengatasi masalah ini, kami telah mengimplementasikan sebuah *fallback function* yang akan digunakan jika pencarian solusi *Minimax Alpha-Beta Pruning* memakan waktu lebih dari 5 detik. Penggunaan *fallback function* ini tentu saja dapat mengurangi efektivitas dari penggunaan algoritma Minimax Alpha-Beta Pruning.

```
if (elapsedTime > 5000){
    System.out.println("System time out");
    return getFallbackMove(button, botChar, botChar);
}
```

```
      public resultMinMax getFallbackMove(Button[][] button, String
playerTurn, String botChar){

              List<int[]> emptyPos = generate_empty_cell(button);

              double maxVal = Double.NEGATIVE_INFINITY;

              int[] maxPosition = {-1, -1};


              for (int[] possiblePos : emptyPos) {

                                          List<int[]>   changeLabel   =
changeAdjacent(possiblePos, button, playerTurn);


                  // Change label button


button[possiblePos[0]][possiblePos[1]].setText(playerTurn);

                  if (changeLabel.size() > 0){

                      for(int[] adjPos:changeLabel){


button[adjPos[0]][adjPos[1]].setText(playerTurn);

                      }

                  }


                   int value = objectiveFunction(possiblePos, button,
0, botChar);

                  if (value > maxVal) {

                      maxVal = value;

                      maxPosition = possiblePos;
```

```
                }


                // Undo label changed

button[possiblePos[0]][possiblePos[1]].setText("");
                if (changeLabel.size() > 0) {
                    for (int[] adjPos: changeLabel) {
                        if (playerTurn.equals("O")){

button[adjPos[0]][adjPos[1]].setText("X");
                        } else {

button[adjPos[0]][adjPos[1]].setText("O");
                        }
                    }
                }
            }
            return new resultMinMax(maxPosition, maxVal);
        }
```

3. Implementasi Local Search: Stochastic Hill Climbing

Algoritma *Local Search* yang kami gunakan dalam permainan *Adversarial Adjacency Strategy Game* adalah algoritma *Stochastic Hill Climbing*. Berikut ini merupakan aspek-aspek penting yang digunakan pada algoritma *local search Stochastic Hill Climbing*.

a. *Initial State* → *State* awal dari pencarian adalah solusi posisi awal yang dipilih secara acak dari *array* posisi sel yang kosong pada papan yang telah dibangkitkan sebelumnya.
b. Max Iterasi → Pada algoritma *Stochastic Hill Climbing* dibutuhkan untuk menentukan max iterasi yang akan menghentikan *loop*. Max iterasi yang digunakan adalah sebanyak posisi sel yang kosong saat ini. Maka max iterasi yang digunakan bersifat dinamis dan akan berubah pada setiap *turn* membuat kemungkinan solusi optimum ditemukan lebih cepat.
c. *Neighbor* → *Neighbor* yang dibangkitkan adalah semua kemungkinan *state* yang mungkin dijalankan dari posisi sekarang.
d. *Action* → Berpindah ke *neighbor* tertinggi yang dihitung dari *objective function*.
e. Solusi → Solusi berada dalam bentuk *array of int* (misal; x,y) yang menyatakan posisi di mana tanda harus diletakkan selanjutnya setelah *initial state*.

Pada pengimplementasiannya dalam game, algoritma ini akan dipanggil setiap bot melakukan *turn*. Oleh karena itu, *initial state* akan berganti sesuai dengan *state game turn* saat ini setelah lawan melakukan gerakannya. Berikut adalah langkah-langkah pencarian solusi menggunakan algoritma *Local Search Stochastic Hill Climbing*.

a. State awal merupakan keadaan awal sesaat setelah lawan melakukan gerakannya. Hal ini digambarkan dengan men-*generate* posisi sel yang kosong pada papan kemudian solusi awal dipilih secara acak dari posisi-posisi sel yang kosong tersebut.

```
List<int[]> positions = generate_empty_cell(button);
Random random = new Random();
int randomIdx = random.nextInt(0, positions.size());
current[0] = positions.get(randomIdx)[0];
current[1] = positions.get(randomIdx)[1];
```

b. Setelah itu, ditentukan max iterasi yang merupakan banyaknya posisi sel yang kosong pada papan. Tentunya iterasi dimulai dari 0.

```
int maxIteration = positions.size();
int iteration = 0;
```

c. Dilakukan *while loop* sebanyak kemungkinan gerakan yaitu sebanyak max iterasi, pada tiap iterasi posisi *current* awal tadi akan diubah dengan *neighbor* ketika nilai *objective function neighbor* lebih tinggi dari nilai *objective function current*. *Neighbor* dipilih secara acak dari posisi-posisi sel yang kosong tadi.

```
while (iteration < maxIteration) {
   int randomIdx2 = random.nextInt(0, positions.size());
   next[0] = positions.get(randomIdx2)[0];
   next[1] = positions.get(randomIdx2)[1];

   if (objectiveFunction(next, button) > objectiveFunction(current,
button)) {
       current[0] = next[0];
       current[1] = next[1];
   }
   long elapsedTime = System.currentTimeMillis() - startTime;
   if (elapsedTime > 5000) {
       System.out.println("System time out");
       return new int[]{current[0], current[1]};
   }
   iteration++;
}

return new int[]{current[0], current[1]};
```

d. Iterasi yang dilakukan akan menghasilkan posisi dari sel kosong dengan nilai *objective function* tertinggi, posisi tersebut kemudian akan menjadi solusi dari *turn bot* saat ini dan bot akan menandai posisi tersebut di papan.

e. Setelah pencarian selesai, maka *turn* akan berpindah ke lawan, setelah lawan selesai melakukan gilirannya, *turn* akan berganti lagi dan algoritma *local search* akan dipanggil kembali.

Pada permainan ini digunakan algoritma *Local Search Stochastic Hill Climbing* karena terdapat beberapa keuntungan dibandingkan dengan algoritma *Local Search* lainnya, di antaranya:

a. *Speed and efficiency,* pada *Adjacency Strategy Game* kecepatan pemilihan keputusan merupakan salah satu hal yang krusial. Algoritma ini cenderung memiliki lebih cepat dalam hal komputasi dibandingkan dengan algoritma *Simulated Annealing*.
b. *Simplicity*, algoritma *Stochastic* cenderung lebih *simple* untuk diimplementasikan dibandingkan dengan algoritma *Local Search* lainnya. Hal ini karena pembangkitan neighbor dilakukan dengan hanya memilih *successor* secara acak sehingga algoritma pemilihan *neighbor* cenderung lebih *simple*. *Simplicity* ini tentunya tidak mengurangi kinerja dari algoritma ini.
c. Dengan memilih *neighbor* secara acak, maka algoritma ini dapat dimaksimalkan untuk menghindari suatu *local optima* (situasi di mana didapatkan solusi yang baik namun bukan solusi terbaik).
d. Mengurangi *overhead* dalam komputasi, *Stochastic Hill Climbing* cenderung memiliki evaluasi yang sedikit sehingga membuatnya lebih efisien dalam hal komputasi.
e. Pada algoritma ini juga terdapat max iterasi yang dapat diatur sesuai dengan *state* atau kebutuhan dalam permainan.

4. Implementasi Genetic Algorithm

Algoritma *Genetic* juga coba kami terapkan untuk menentukan langkah terbaik yang dapat diambil *bot* dalam permainan *Adversarial Adjacency Strategy Game*. Berikut ini adalah beberapa aspek penting yang perlu diperhatikan dalam implementasi algoritma *Genetic*:

a. *Move* : *Move* merepresentasikan koordinat papan (x, y) yang dapat diakses oleh pemain. Pada kode program, *move* memiliki tipe data *array of integer* yang berisi 2 elemen, yaitu x dan y.
b. Representasi gen/kromosom : Kromosom merepresentasikan rangkaian *move* yang dapat dilakukan pemain. Pada kode program, kromosom direpresentasikan

oleh suatu variabel, *chromosome*, yang memiliki tipe data *2-dimensional array of integer*. Jumlah *move* yang dapat ditampung oleh 1 kromosom tergantung pada jumlah ronde tersisa yang dapat dimainkan oleh *bot*.

c. Populasi : Populasi merepresentasikan seluruh kemungkinan *chromosome* yang dapat dipilih sebagai *best chromosome* oleh *bot* yang kemudian *moves* di dalam *chromosome* tersebut akan digunakan pada giliran permainan *bot*. Pada kode program, populasi memiliki tipe data 3-*dimensional array of integer*.

d. Fungsi *fitness*: Fungsi *fitness* merupakan suatu fungsi yang digunakan untuk mengukur dan mengevaluasi sejauh mana sebuah *chromosome* dapat menjadi solusi permainan bagi *bot*.

e. Operator genetika:

- *Selection*

  Proses *selection* merupakan proses pemilihan *chromosome-chromosome parents* yang akan digunakan untuk reproduksi generasi berikutnya.

- *Crossover*

  Proses *crossover* merupakan proses pertukaran informasi genetik antara 2 *chromosome* untuk menghasilkan *chromosome* anak yang baru

- *Mutation*

  Proses *mutation* merupakan proses penggantian beberapa gen dari *chromosome* secara acak untuk menciptakan variasi pada keturunan.

f. *Mutation rate*: Probabilitas dilakukannya sebuah mutasi. Mutasi akan dilakukan jika angka *random* yang di-*generate* berada di bawah nilai *mutation rate*

g. *Population size*: Ukuran populasi yang akan di-*generate* secara *random* di awal permainan

h. *Max generations*: Batas atas banyak reproduksi generasi baru yang akan dilakukan

Berikut ini beberapa langkah dari *genetic algorithm* yang telah kami implementasikan:

1. Menginisiasi populasi awal dengan ukuran maksimal populasi adalah 100. Setiap individu dalam populasi adalah kromosom yang merepresentasikan serangkaian langkah yang mungkin dalam permainan. Proses pembentukan populasi dilakukan

secara *random* dan *moves* pada kromosom diambil secara acak dari list koordinat papan yang masih kosong. Proses generasi titik secara *random* untuk suatu kromosom akan dilakukan sampai tidak ada titik yang sama dalam 1 kromosom dan jumlah titik dalam kromosom sama dengan ukuran dari kromosom.

```
public int[][][]initializePopulation(List<int[]> emptyCells, int roundLeft){
    int[][][] population = new int[POPULATION_SIZE][][];
    if(roundLeft == 0){
        return new int[][][]{};
    }
    for(int i = 0; i < POPULATION_SIZE; i++){
        int[][] chromosome = new int[roundLeft][2];
        for(int j = 0; j < roundLeft; j++){
            int index_list;
            int x, y;
            boolean hasDuplicates;
            do {
                Random random = new Random();
                index_list = random.nextInt(emptyCells.size());
                x = emptyCells.get(index_list)[0];
                y = emptyCells.get(index_list)[1];
                hasDuplicates = false;
                for (int k = 0; k < j; k++) {
                    if (chromosome[k][0] == x && chromosome[k][1] == y) {
                        hasDuplicates = true;
                        break;
                    }
                }
            } while (hasDuplicates);
            chromosome[j][0] = x;
            chromosome[j][1] = y;
        }
        population[i] = chromosome;
    }
    return population;
```

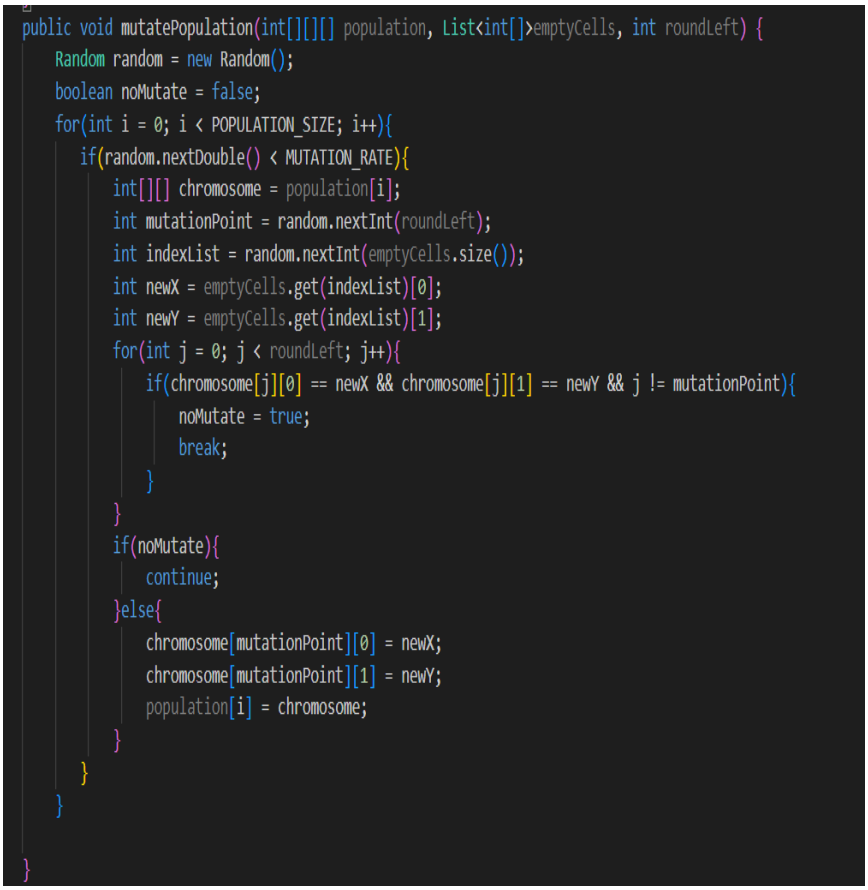
2. Setiap kromosom dalam populasi kemudian dievaluasi dengan menggunakan fungsi *fitness* untuk menghitung nilai utilitas tiap kromosom, kemudian dicari kromosom terbaik pada generasi tersebut. Fungsi *fitness* yang digunakan sama dengan fungsi objektif yang digunakan pada *local search*.

```java
public int evaluateChromosome(int[][] chromosome, Button[][] buttons, int roundLeft, String player){
    int chromosomeValue = 0;
    for(int[]c : chromosome){
        chromosomeValue += (objectiveFunction(c, buttons, roundLeft, player));
    }
    return chromosomeValue;
}
```

3. Untuk pembentukan generasi baru, dilakukan proses *crossover* pada populasi yang ada saat ini. Proses *crossover* dilakukan secara *random* baik dalam pemilihan 2 *parents*-nya serta dalam penentuan *crossover point*. Apabila *chromosome* anak yang dihasilkan ternyata menghasilkan rangkaian langkah yang tidak *valid* (terdapat titik koordinat yang sama dalam satu kromosom), *crossover* tersebut gagal dilakukan.

```java
public int[][][] crossoverPopulation(int[][][] population, int roundLeft){
    Random random = new Random();
    int[][][] newPopulation = new int[POPULATION_SIZE][][];
    newPopulation = population;
    boolean skipCrossOver = false;
    for(int i = 0; i < POPULATION_SIZE; i++){
        int parent1Index = random.nextInt(POPULATION_SIZE);
        int parent2Index = random.nextInt(POPULATION_SIZE);
        int[][] parent1 = population[parent1Index];
        int[][] parent2 = population[parent2Index];
        int[][] childChromosome1 = new int[roundLeft][2];
        int[][] childChromosome2 = new int[roundLeft][2];
        childChromosome1 = parent1;
        childChromosome2 = parent2;
        int crossoverPoint = random.nextInt(roundLeft);
        for(int j = crossoverPoint; j < roundLeft; j++){
            childChromosome1[j][0] = parent2[j][0];
            childChromosome1[j][1] = parent2[j][1];
            for(int k = 0; k < roundLeft; k++){
                if(Arrays.equals(childChromosome1[k],childChromosome1[j])&& k != j){
                    skipCrossOver = true;
                    break;
                }
            }
            if(skipCrossOver){
                break;
            }
            childChromosome2[j][0] = parent1[j][0];
            childChromosome2[j][1] = parent1[j][1];
        }
        newPopulation[parent1Index] = childChromosome1;
        newPopulation[parent2Index] = childChromosome2;
    }
    population = newPopulation;
    return population;
}
```

4. Selain proses *crossover*, akan dilakukan pula proses *mutation* pada *rate* tertentu untuk menciptakan variasi pada generasi baru. Apabila angka random yang di-*generate* berada di bawah *mutation rate* (0.2), proses *mutation* akan dilakukan. Kromosom yang akan mengalami *mutation* akan di-*generate* secara acak dan apabila dihasilkan anak yang tidak *valid*, *mutation* akan gagal dilakukan.



```
public void mutatePopulation(int[][][] population, List<int[]>emptyCells, int roundLeft) {
    Random random = new Random();
    boolean noMutate = false;
    for(int i = 0; i < POPULATION_SIZE; i++){
        if(random.nextDouble() < MUTATION_RATE){
            int[][] chromosome = population[i];
            int mutationPoint = random.nextInt(roundLeft);
            int indexList = random.nextInt(emptyCells.size());
            int newX = emptyCells.get(indexList)[0];
            int newY = emptyCells.get(indexList)[1];
            for(int j = 0; j < roundLeft; j++){
                if(chromosome[j][0] == newX && chromosome[j][1] == newY && j != mutationPoint){
                    noMutate = true;
                    break;
                }
            }
            if(noMutate){
                continue;
            }else{
                chromosome[mutationPoint][0] = newX;
                chromosome[mutationPoint][1] = newY;
                population[i] = chromosome;
            }
        }
    }

}
```

5. Proses inisialisasi populasi, *crossover*, *mutation,* serta penentuan kromosom terbaik akan dilakukan terus-menerus secara iteratif sampai batas iterasi MAX_GENERATIONS (banyak generasi maksimum yang dihasilkan) atau sampai waktu berpikir bot mencapai 5 detik.

```java
public int[][] runGeneticAlgorithm(Button[][] buttons, int roundLeft, boolean isBotFirst, String botChar){
    long startTime = System.currentTimeMillis();
    Random random = new Random();
    int[][] bestChromosome = null;
    int bestFitness = 0;

    if(roundLeft == 0){
        return new int[][]{};
    }
    for(int gen = 0; gen < MAX_GENERATIONS; gen++){
        List<int[]>emptyCells = generate_empty_cell(buttons);
        int[][][] population = initializePopulation(emptyCells, roundLeft);
        for(int[][] chromosome : population){
            int fitness = evaluateChromosome(chromosome, buttons, roundLeft, botChar);
            if(fitness > bestFitness){
                bestFitness = fitness;
                bestChromosome = chromosome;
            }
        }
        long elapsedTime = System.currentTimeMillis() - startTime;
        if(elapsedTime > 5000){
            break;
        }
        int[][][] newPopulation = crossoverPopulation(population, roundLeft);
        mutatePopulation(newPopulation, emptyCells, roundLeft);
        population = newPopulation;
    }
    return bestChromosome;
}
```

6. Kromosom terbaik terakhir yang terbentuk akan diambil *move* pertamanya dan dijadikan langkah oleh *bot* pada giliran tersebut.

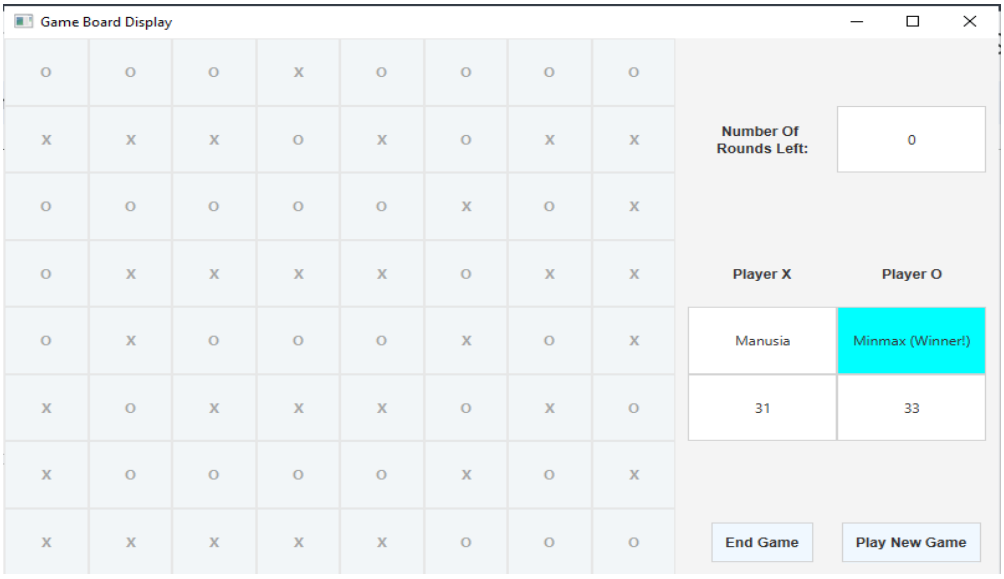
```java
public int[] move(Button[][] button, int roundLeft, boolean isBotFirst, String botChar) {

    int[][] bestChromosome = runGeneticAlgorithm(button, roundLeft, isBotFirst, botChar);
    if(bestChromosome.length == 0){
        return new int[]{};
    } else {
        return bestChromosome[0];
    }
}
```

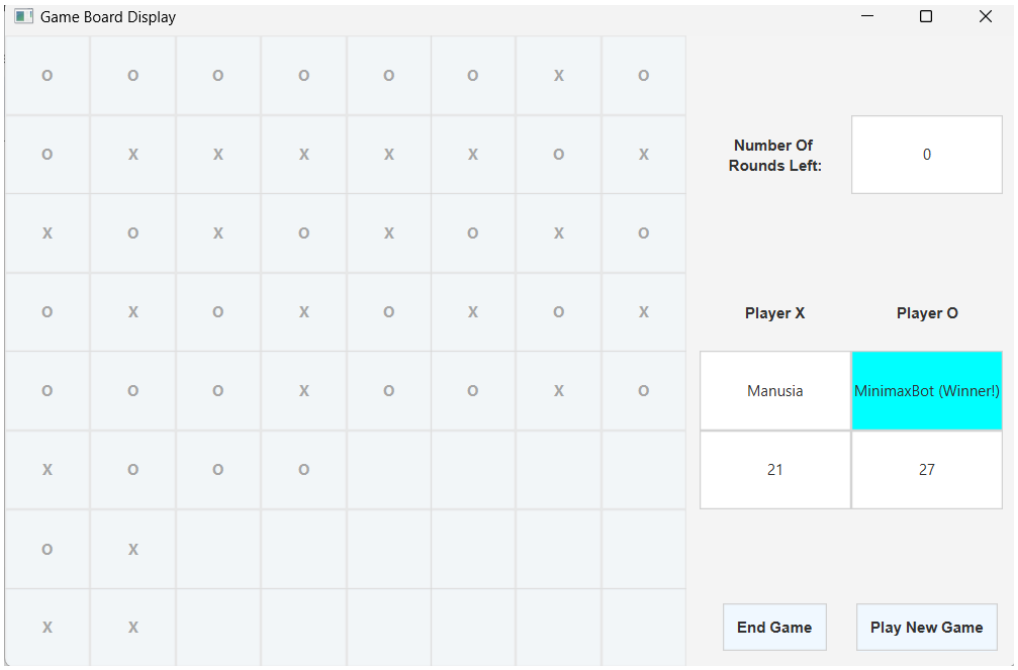
5. Hasil Pertandingan

a. Bot *minmax* vs manusia

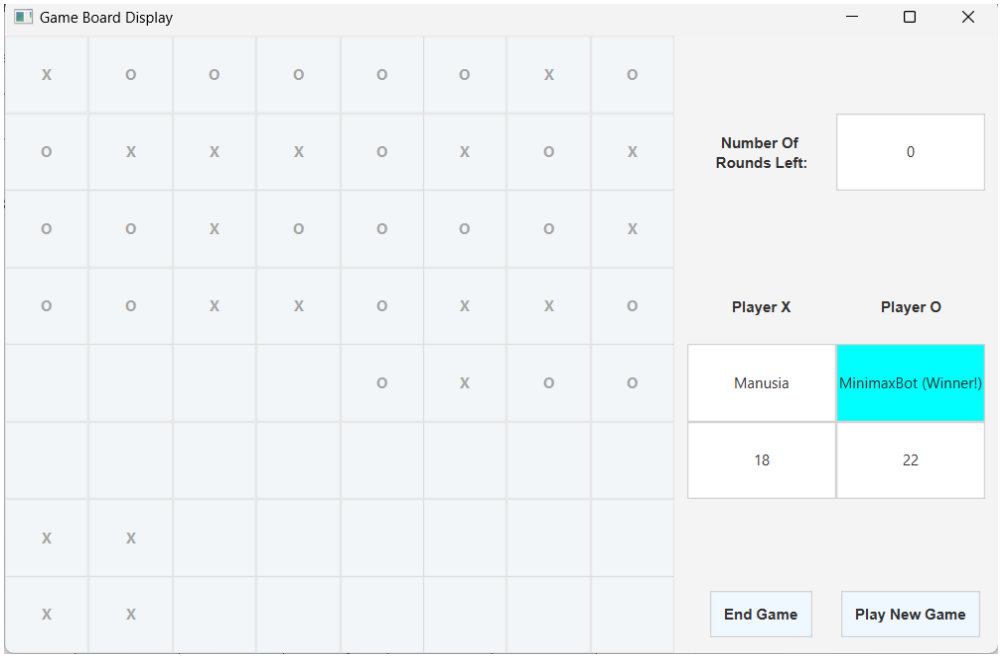
Pertandingan			Player	Score
Match 1	28 Ronde	1st Player	Manusia	31
		2nd Player	Bot	33
Match 2	20 Ronde	1st Player	Manusia	21
		2nd Player	Bot	27
Match 3	16 Ronde	1st Player	Manusia	18
		2nd Player	Bot	22
Match 4	12 Ronde	1st Player	Manusia	14
		2nd Player	Bot	18
Match 5	8 Ronde	1st Player	Manusia	12
		2nd Player	Bot	12
Hasil Pertandingan (Minmax vs Manusia) = 4 : 0				



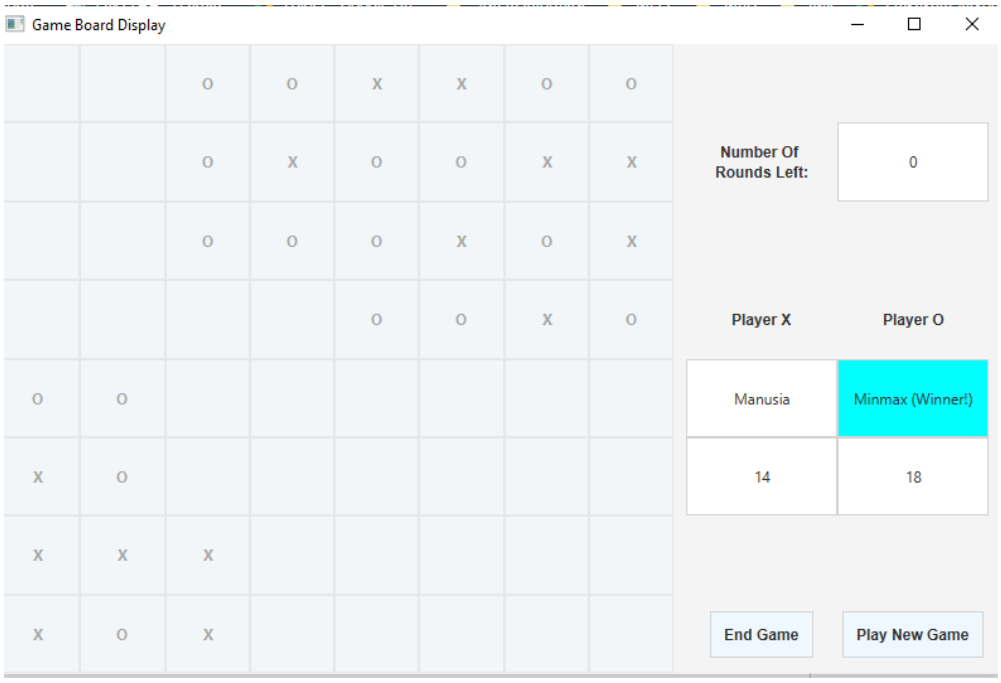
Gambar 5.a.1 Pertandingan Manusia vs Minmax Bot



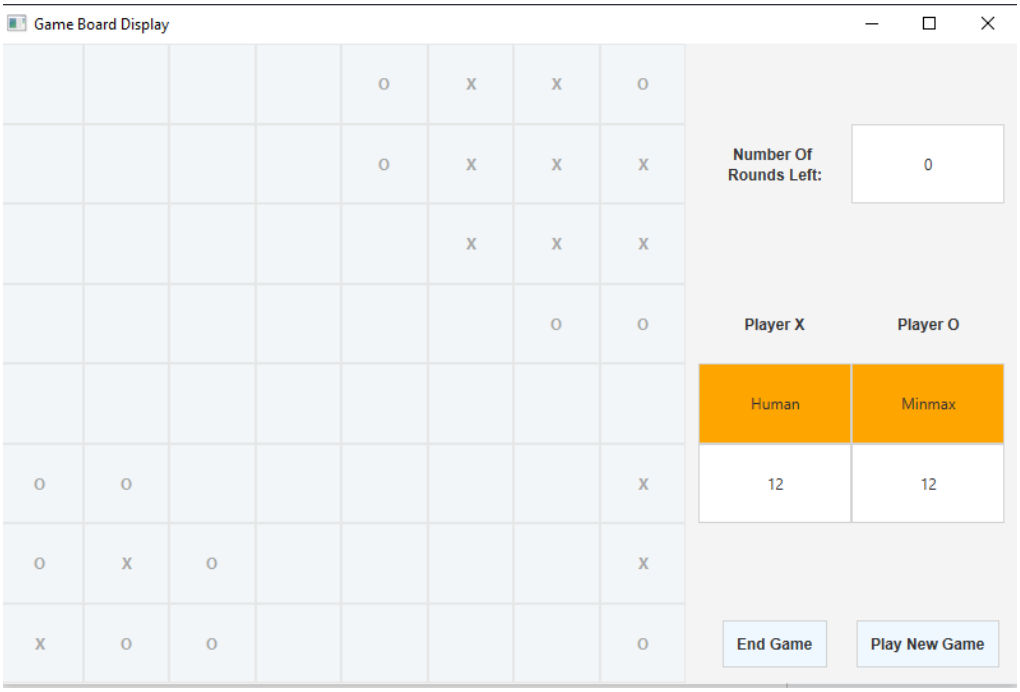
Gambar 5.a.3 Pertandingan Manusia vs Minmax Bot



Gambar 5.a.3 Pertandingan Manusia vs Minmax Bot



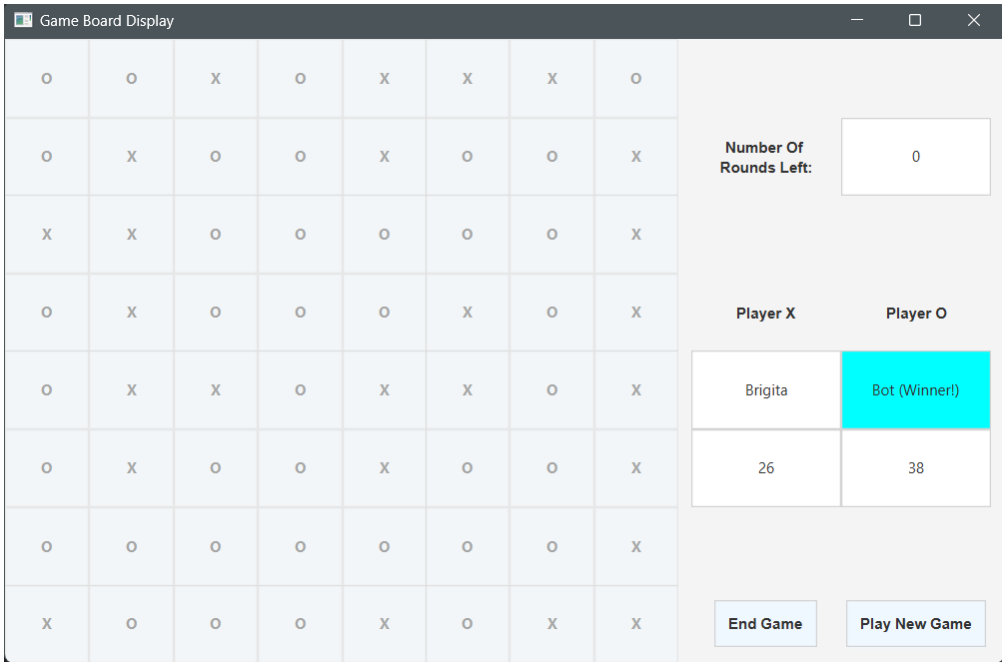
Gambar 5.a.4 Pertandingan Manusia vs Minmax Bot



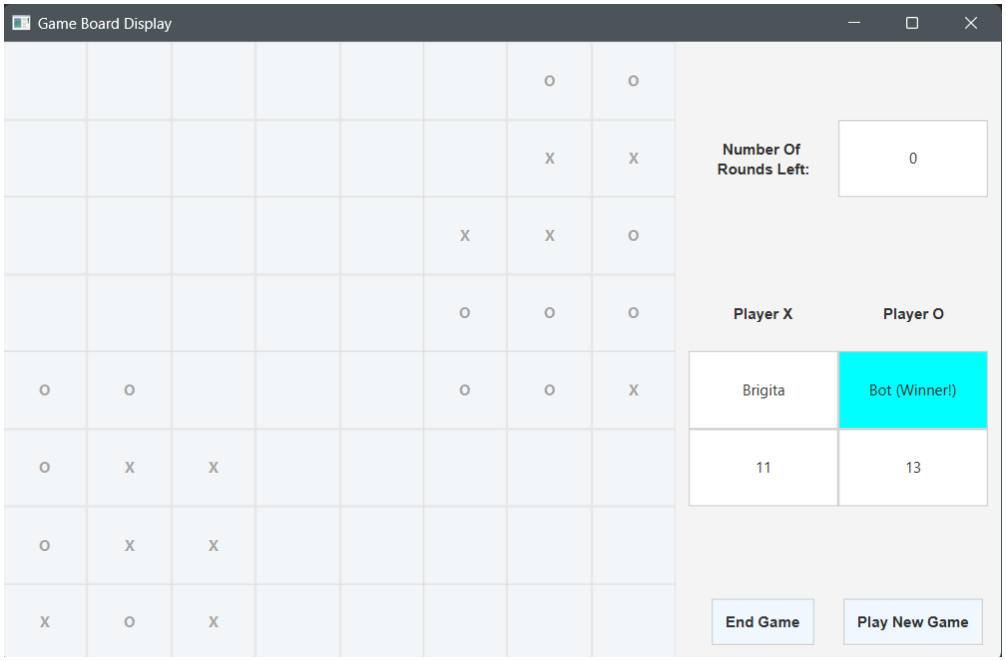
Gambar 5.a.5 Pertandingan Manusia vs Minmax Bot

b. Bot *local search* vs. manusia

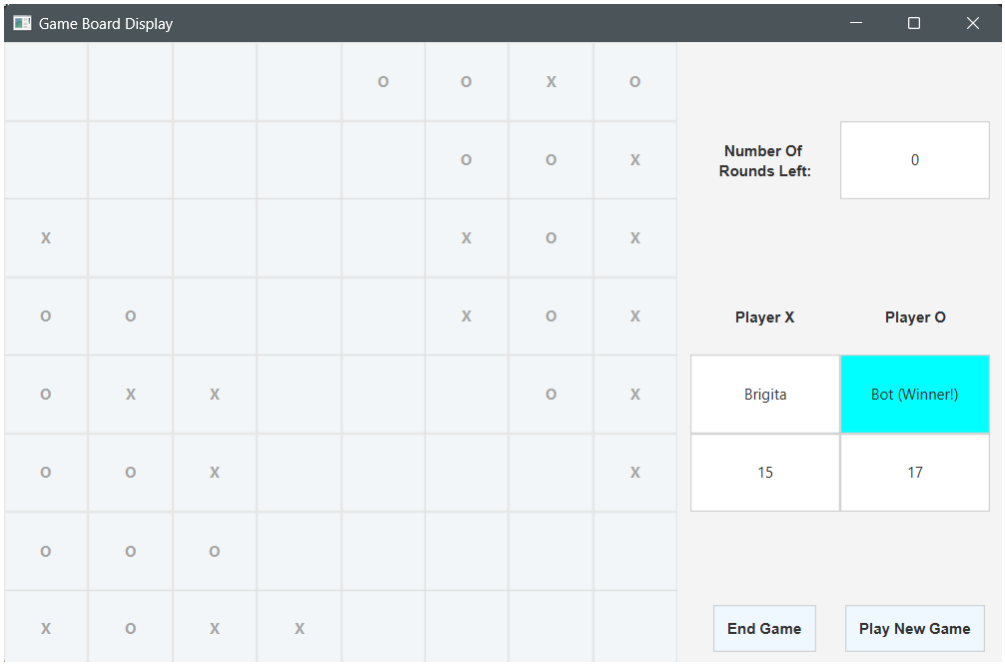
Pertandingan			Player	Score
Match 1	28 Ronde	1st Player	Manusia	26
		2nd Player	Bot	38
Match 2	12 Ronde	1st Player	Manusia	15
		2nd Player	Bot	17
Match 3	8 Ronde	1st Player	Manusia	11
		2nd Player	Bot	13
Hasil Pertandingan (Local search vs. manusia) = 3: 0				



Gambar 5.b.1 Pertandingan Manusia vs Local Bot



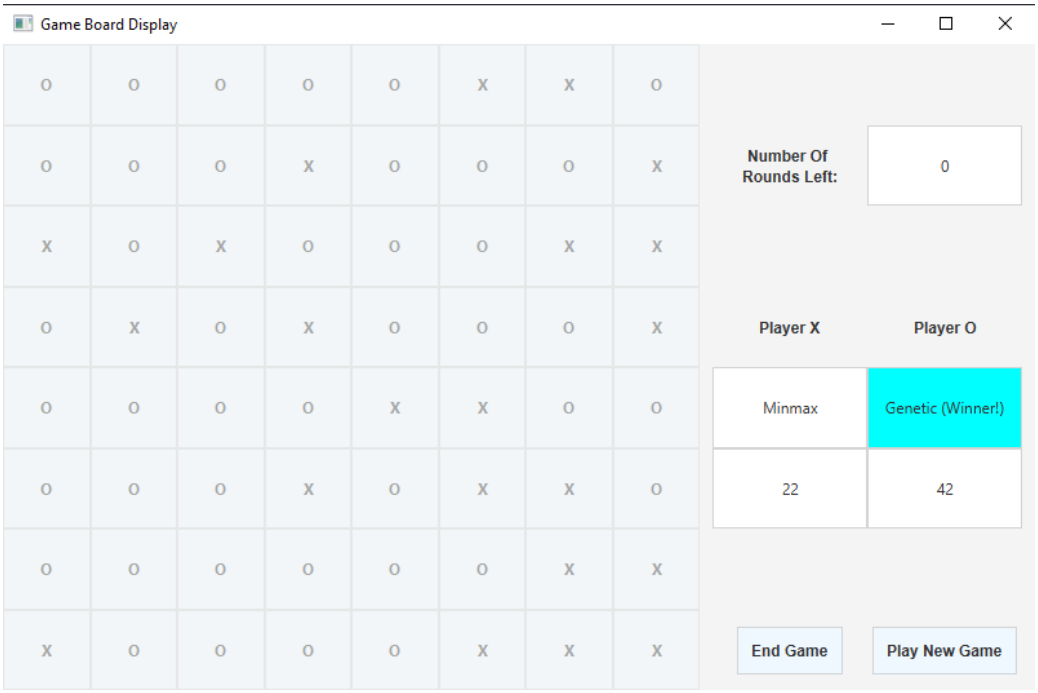
Gambar 5.b.2 Pertandingan Manusia vs Local Bot



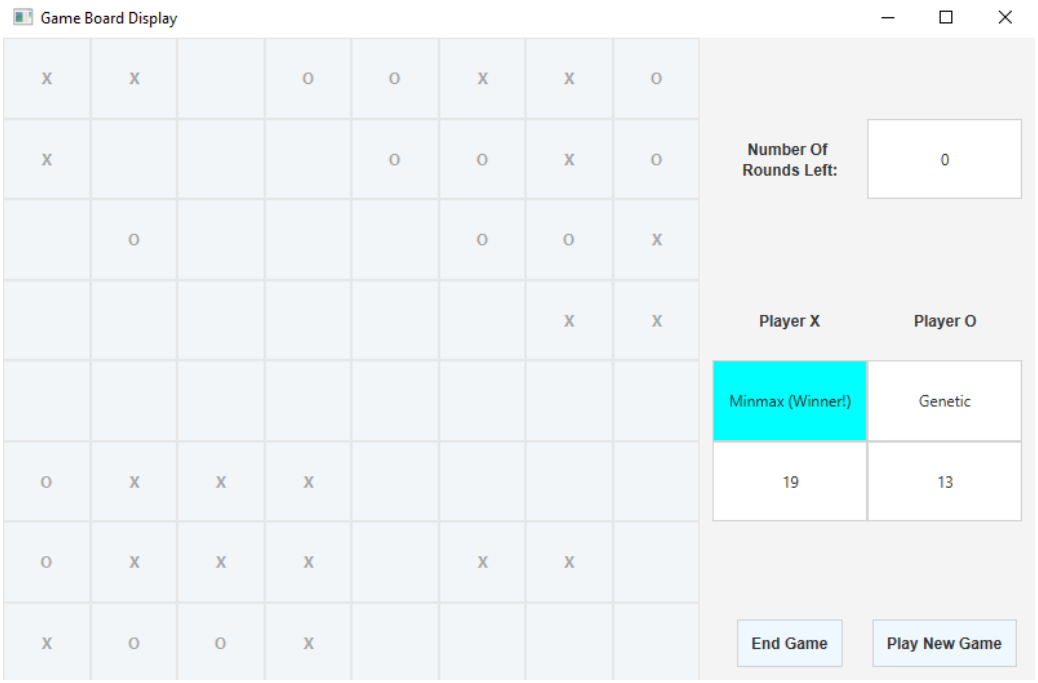
Gambar 5.b.3 Pertandingan Manusia vs Local Bot

c. Bot *minimax* vs bot *genetic algorithm*

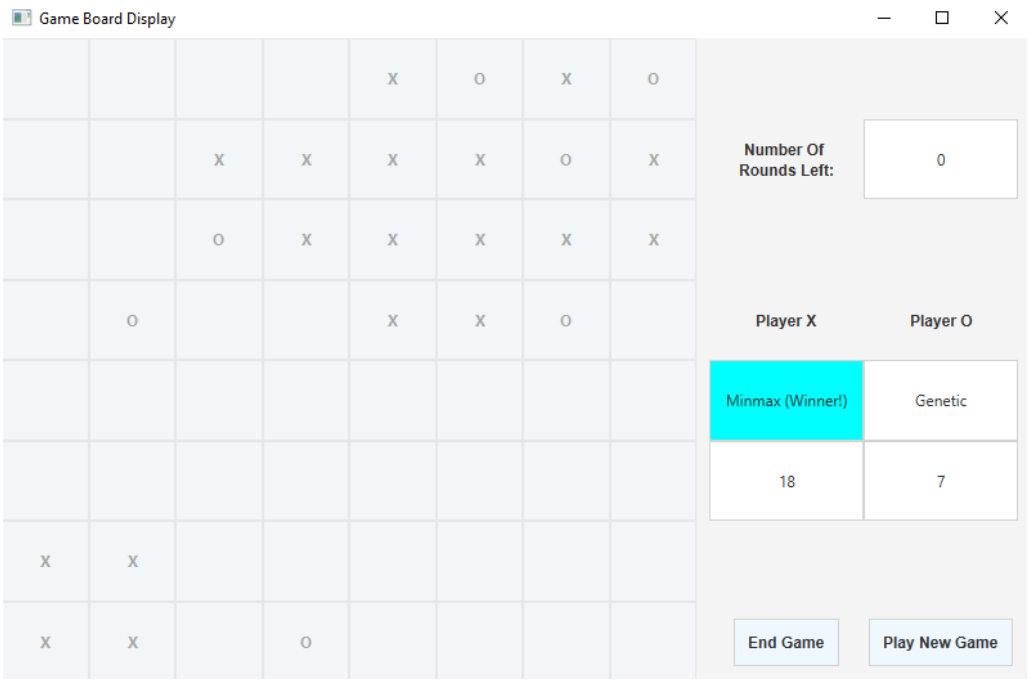
Pertandingan			Player	Score
Match 1	28 Ronde	1st Player	Minimax Bot	22
		2nd Player	Genetic Bot	42
Match 2	12 Ronde	1st Player	Minimax Bot	19
		2nd Player	Genetic Bot	13
Match 3	8 Ronde	1st Player	Minimax Bot	18
		2nd Player	Genetic Bot	7
Hasil Pertandingan (Minmax vs Genetic Algorithm) = 3 : 0				



Gambar 5.c.1 Pertandingan Minmax Bot vs Genetic Bot



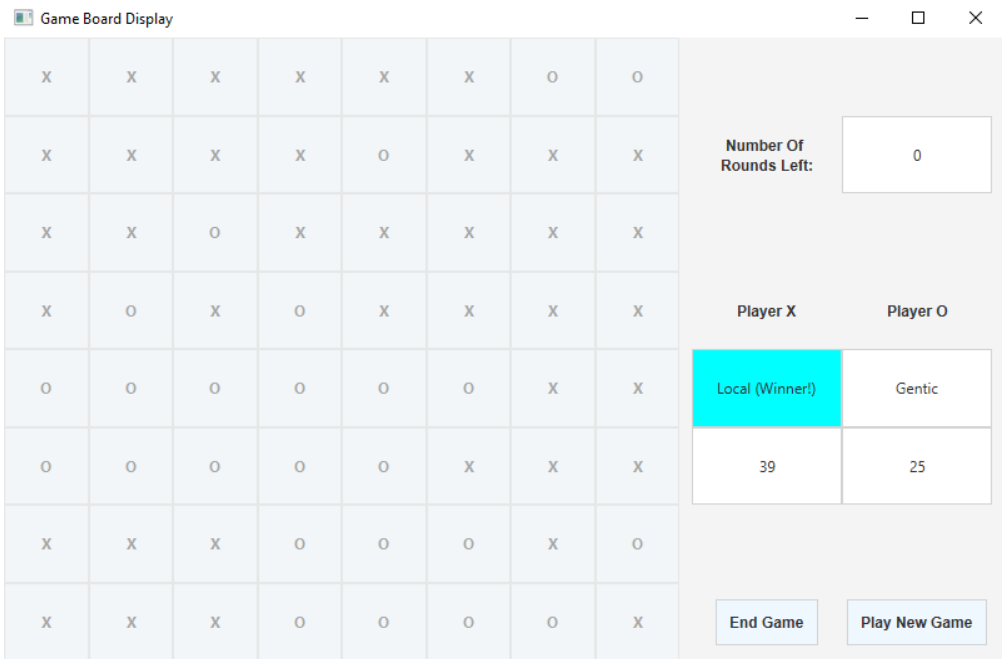
Gambar 5.c.2 Pertandingan Minmax Bot vs Genetic Bot



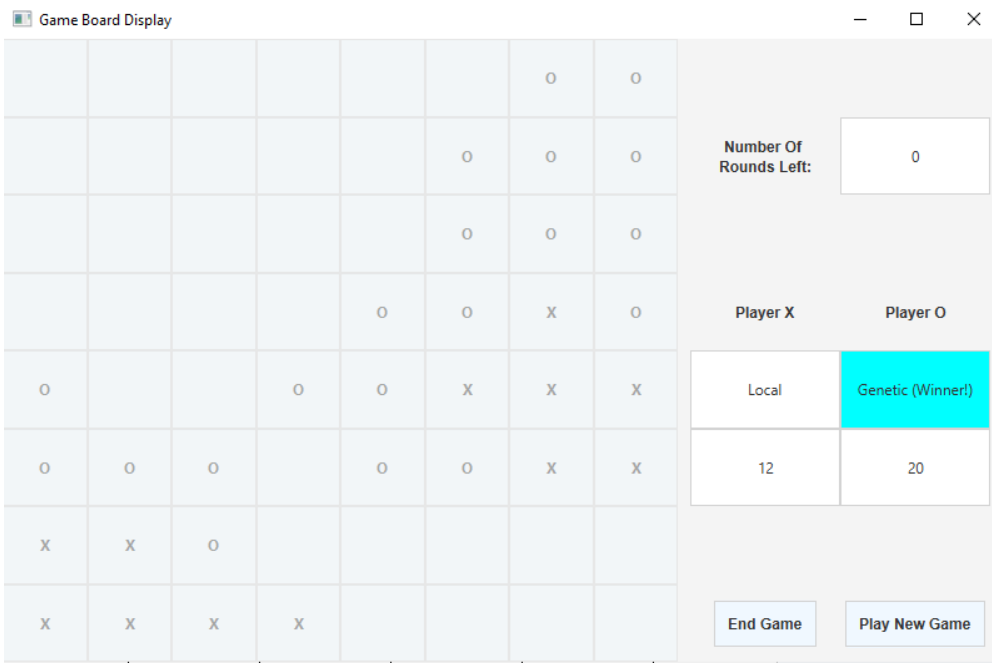
Gambar 5.c.3 Pertandingan Minmax Bot vs Genetic Bot

d. Bot *local search* vs bot *genetic*

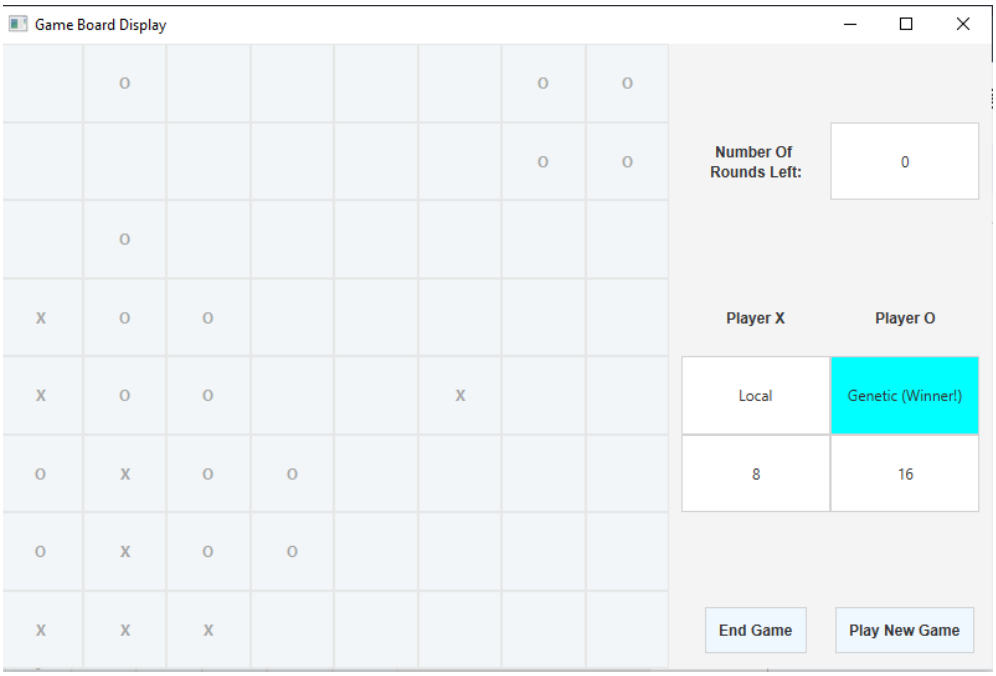
Pertandingan			Player	Score
Match 1	28 Ronde	1st Player	Local Bot	39
		2nd Player	Genetic Bot	25
Match 2	12 Ronde	1st Player	Local Bot	12
		2nd Player	Genetic Bot	20
Match 3	8 Ronde	1st Player	Local Bot	8
		2nd Player	Genetic Bot	16
Hasil Pertandingan (Local Search vs Genetic) = 1 : 2				



Gambar 5.d.1 Pertandingan Local Bot vs Genetic Bot



Gambar 5.d.2 Pertandingan Local Bot vs Genetic Bot



Gambar 5.d.3 Pertandingan Local Bot vs Genetic Bot

6. Kesimpulan dan Saran

Dari ketiga uraian algoritma diatas, yaitu algoritma *MiniMax*, *Local Search,* dan *Genetic Algorithm*, dapat disimpulkan bahwa algoritma dengan hasil terbaik merupakan algoritma *MiniMax*. Hal ini terjadi karena apa algoritma MiniMax mengeksplorasi semua state space game dan memilih aksi yang terbaik ketika semuanya sudah dieksplorasi. Hal ini menguntungkan bagi bot karena bot bisa memilih aksi yang memaksimalkan skor bot dan meminimalkan skor lawan. Berdasarkan implementasi yang telah dilakukan pada permainan, didapatkan bahwa performa dari algoritma *Local Search* bergantung pada *objective function.* Oleh karena itu, algoritma *Local Search* yang dipilih tidak terlalu mempengaruhi hasil pertandingan namun tentunya berpengaruh pada waktu komputasi. Ketika ingin melakukan perbaikan atau meningkatkan hasil pertandingan dengan algoritma *Local Search*, maka bagian yang fokus diperbaiki adalah bagian *objective function*. Di lain sisi, performa dari algoritma *Genetic Algorithm* sangat bergantung pada pembangkitan populasi dan keturunan yang dilakukan. Karena pembangkitan populasi dan keturunan dilakukan secara *random* dan jumlah iterasi yang dilakukan, baik untuk populasi maupun generasi dibatasi hanya sampai 100 (untuk alasan optimasi), langkah yang dihasilkan dari algoritma ini seringkali kurang efektif jika dibandingkan dengan algoritma *MiniMax*.

7. Link Repository

Algoritma untuk setiap Bot, baik *Minimax Alpha-Beta Pruning, Local Search,* dan *Genetic Algorithm* yang telah dijelaskan poin-poin sebelumnya, diimplementasikan pada pranala berikut:

https://github.com/kandidagunawan/Tubes1_13521077.git

8. Kontribusi

NIM	Nama	Kontribusi
13521077	Husnia Munzayana	1. Implementasi minmax bot (Bot terpilih) 2. Membuat laporan
13521132	Dhanika Novlisariyanti	1. Implementasi local bot 2. Membuat laporan
13521155	Kandida Edgina Gunawan	1. Implementasi minmax bot 2. Implementasi genetic 3. Membuat laporan
13521156	Brigita Tri Carolina	1. Implementasi local bot (Bot terpilih) 2. Membuat laporan