LAPORAN TUGAS BESAR

IF3170 Inteligensi Buatan

Minimax Algorithm and Alpha Beta Pruning in Adjacency Strategy Game

Disusun Oleh:

Kelompok 24

Henry Anand Septian Radityo / 13521004 / K-03

Christophorus Dharma Winata / 13521009 / K-03

Haikal Ardzi Shofiyyurrohman / 13521012 / K-03

Maggie Zeta Rosida Simangunsong / 13521117 / K-01

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2023/2024

DAFTAR ISI

BAB I

1.1 Deskripsi

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).

BAB II

2.1 Algoritma Hill Climb

Algoritma Hill Climb adalah algoritma lokal yang bekerja terus menerus ke atas hingga mencapai solusi terbaik. Algoritma hill-climbing memiliki node yang terdiri dari *state* dan *value* untuk setiap *state*. Algoritma ini akan mencari *neighbor state* yang memiliki *value* lebih tinggi hingga mencapai puncak dari *value*. Tujuan dari algoritma ini adalah untuk mencari keadaan paling optimal yang dapat dijangkau. Proses pencarian oleh algoritma hill-climb akan terus dilakukan hingga mencapai solusi puncak tercapai. Proses ini dilakukan dengan pendekatan berbasis algoritma *greedy* dan tidak memiliki mekanisme *backtracking*. Setiap perubahan yang dilakukan oleh algoritma ini bersifat inkremental.

2.2 Algoritma Minimax

Algoritma Minimax digunakan untuk mengambil keputusan dengan tujuan mengurangi risiko kehilangan nilai maksimal, yang akan mengurangi peluang kekalahan dan meningkatkan peluang kemenangan. Algoritma ini digunakan dalam permainan yang melibatkan dua pemain dan memiliki sifat zero-sum, seperti permainan *adjacency strategy game* di mana keuntungan salah satu pemain sebanding dengan kerugian pemain lainnya. Algoritma Minimax melakukan analisis terhadap semua kemungkinan langkah yang ada, menghasilkan suatu pohon permainan yang mencakup semua skenario permainan tersebut. Algoritma Minimax dapat memilih langkah yang optimal dengan asumsi bahwa lawan akan selalu memilih langkah terbaik untuk dirinya sendiri dan terburuk untuk komputer. Prinsip inti dari algoritma Minimax ini adalah bahwa komputer akan mencari jalur maksimum (node maksimum) yang akan menghasilkan nilai maksimal dalam jalur tersebut, sementara lawan akan mencari cara untuk meminimalkan nilai komputer (node minimum). Oleh karena itu, komputer berusaha memaksimalkan kemungkinan mendapatkan nilai terendah.

2.3 Algoritma Genetic

Algoritma genetik merupakan algoritma yang memanfaatkan proses alamiah yang terinspirasi dari proses evolusi. Dalam penerapannya, algoritma ini akan memilih parent yang dianggap unggul untuk dilakukan penyilangan. Proses akan terus berlanjut dengan

mempertahankan individu yang mampu bertahan hingga mencapai keturunan yang paling baik. Di akhir proses, terdapat langkah mutasi dengan tujuan mengubah keturunan menjadi lebih baik.

Pada dasarnya, algoritma tidak lazimnya digunakan untuk permainan 2 pemain dalam mencari langkah terbaik saat ini. Namun, pada implementasi tugas ini, algoritma *Genetic* dapat diimplementasikan dengan modifikasi untuk menyesuaikan konteks *Adversial* Search.

Pada konteks *Genetic Algorithm* (GA) *adversial* search, kita meng-*encode* suatu spesimen dengan suatu urutan langkah yang dapat diambil seperti suatu kumpulan langkah yang diambil pada *game tree* di dalam algoritma *minimax* dari node akar hingga suatu node daun. Panjang dari kromosom ini secara sendirinya mewakilkan kedalaman dari *game tree* di mana kasus yang terjadi adalah langkah yang disimulasikan dari kromosom. Dengan begitu, kita bisa menentukan ke dalam dari *game tree* dari konteks GA *adversial search* hanya dengan menentukan panjang *sequence of chromosomes*. Pada akhirnya, operasi genetis seperti seleksi, *crossover*, dan mutasi dapat dilakukan pada *specimens* atau *sequence of chromosomes* dengan menyesuaikan dasar GA.

BAB III

3.1 Fungsi Objektif

Fungsi objektif yang digunakan adalah selisih antara nilai pemain satu dengan pemain lainnya. Contoh, apabila pemain satu adalah X dan pemain 2 adalah O, maka akan dilakukan perulangan untuk tiap *block* untuk mencari jumlah X dan O lalu mencari selisihnya. Dalam kode program java, fungsi ini dituliskan sebagai berikut.

```
public int evaluateBoard(int pNumber) {
        int sum = 0;
        for (char[] row :
                Positions) {
            for (char mark :
                    row) {
                if (pNumber==1){
                    if (mark == 'O') {
                        sum--;
                    } else if (mark == 'X') {
                        sum++;
                    }
                }
                else{
                    if (mark == 'O') {
                        sum++;
                    } else if (mark == 'X') {
                        sum--;
                    }
                }
            }
        }
        return sum;
    }
```

3.2 Proses Pencarian dengan *Minimax*

3.2.1 Proses Pencarian

Proses pencarian dilakukan dengan beberapa tahapan, dapat dijelaskan sebagai berikut :

1. Instansiasi bot dengan algoritma *Minimax*
2. Untuk tiap *state*, dilakukan proses `move()` yang merupakan *inner function* dari kelas bot
3. *Inner function* `move()` akan mencari dengan algoritma heuristik untuk mendapatkan koordinat yang mungkin

4. Didefinisikan fungsi heuristik untuk *minimax* dengan mencari kemungkinan koordinat yang menguntungkan atau adjacent dengan *block* lain yang terisi
5. Tiap koordinat yang diambil dari fungsi heuristik tersebut akan dicari dengan menggunakan algoritma *minimax* untuk dilakukan pencarian nilai yang terbesar.
6. Terdapat metode *pruning* yang membatasi pencarian *tree* dengan memotong pencarian untuk node yang tidak diperlukan
7. *Depth* yang diambil dari algoritma *minimax* adalah 3 dan terdapat *fallback plan* ketika terdapat kasus yang memerlukan waktu pencarian lebih dari 5 detik.
8. Nilai dari *minimax* yang terbesar akan dikembalikan dan ditandai sebagai jawaban dari `move()`

Berikut merupakan kelas bot dengan algoritma *Minimax* yang merupakan turunan dari *abstract class* bot.

```
public class BotMinimax extends Bot{
    private int playerNumber;
                  private    final    ExecutorService    executorService    =
Executors.newSingleThreadExecutor();
    private final long TIMEOUT = 5000;
    private Pair<Integer, Integer>[] pairs;
    /***
     * Move with minimax alpha beta pruning algorithm
     * @param buttons matrix of buttons representing the board
     * @return best coordinate to move to
     */
    @Override
    public int[] move(Button[][] buttons, int pNumber) {
        this.playerNumber = pNumber;
        Callable<int[]> minimaxTask = () -> {
            int c = 0;
            int max = Integer.MIN_VALUE;
            int mi = 0, mj = 0;
            HashSet<Pair<Integer, Integer>> uniquePairs = new HashSet<>();
            for (int i = 0; i < 8; i++) {
                for (int j = 0; j < 8; j++) {
                                    if  (buttons[i][j].getText().equals("X")  ||
buttons[i][j].getText().equals("O")) {
                        int[][] directions = {
                                {0, 1},
                                {0, -1},
                                {1, 0},
                                {-1, 0},
                        };
                        for (int[] dir : directions) {
                            int newX = i + dir[0];
                            int newY = j + dir[1];
                            if (newX >= 0 && newX < 8 && newY >= 0 && newY < 8
&& buttons[newX][newY].getText().equals("")) {
```

```
                                uniquePairs.add(new Pair<>(newX, newY));
                            }
                        }
                    }
                }
            }
            for (Pair<Integer, Integer> pair : uniquePairs) {
                int i = pair.getKey();
                int j = pair.getValue();
                if (buttons[i][j].getText().equals("")) {
                    c++;
                    BoardState position = new BoardState(buttons);
                    int minimaxValue;
                    if (this.playerNumber == 1){
                         minimaxValue = minimax(new BoardState(position, i, j,
'X'), 3, Integer.MIN_VALUE, Integer.MAX_VALUE, false, 0);
                    } else {
                         minimaxValue = minimax(new BoardState(position, i, j,
'O'), 3, Integer.MIN_VALUE, Integer.MAX_VALUE, false, 0);
                    }
                    if (max < minimaxValue) {
                        max = minimaxValue;
                        mi = i;
                        mj = j;
                    }
                }
            }
            int[] ret = {mi, mj};
            return ret;
        };
        Future<int[]> future = executorService.submit(minimaxTask);
        try {
            return future.get(TIMEOUT, TimeUnit.MILLISECONDS);
         } catch (InterruptedException | ExecutionException | TimeoutException
e) {
            return getRandomMove(buttons);
        } finally {
            future.cancel(true);
        }
    }

     public int minimax(BoardState position, int depth, int alpha, int beta,
boolean maximizingPlayer, int count){
        if (depth == 0 || position.isFull()){
            return position.evaluateBoard(playerNumber);
        }
        if (maximizingPlayer){
            Integer maxEval = Integer.MIN_VALUE;
            for (int i = 0; i < 8; i++) {
                for (int j = 0; j < 8; j++) {
                    if (position.Positions[i][j] == '_') {
                        int eval;
                        if (playerNumber == 1) {
                                eval = minimax(new BoardState(position, i, j,
'X'), depth-1, alpha, beta, false, count+1);
                        } else {
```

```
                                eval = minimax(new BoardState(position, i, j,
'O'), depth-1, alpha, beta, false, count+1);
                        }
                        maxEval = Math.max(eval, alpha);
                        alpha = Math.max(maxEval,alpha);
                        if (beta <= alpha) {
                            break;
                        }
                    }
                }
            }
            return maxEval;
        } else{
            Integer minEval = Integer.MAX_VALUE;
            for (int i = 0; i < 8; i++) {
                for (int j = 0; j < 8; j++) {
                    if (position.Positions[i][j] == '_') {
                        int eval;
                        if (playerNumber == 1) {
                                eval = minimax(new BoardState(position, i, j,
'O'), depth-1, alpha, beta, true, count+1);
                        } else {
                                eval = minimax(new BoardState(position, i, j,
'X'), depth-1, alpha, beta, true, count+1);
                        }
                        minEval = Math.min(eval, beta);
                        beta = Math.max(minEval,beta);
                        if (beta <= alpha) {
                            break;
                        }
                    }
                }
            }
            return minEval;
        }
    }
    private int[] getRandomMove(Button[][] buttons) {
        List<int[]> eligibleMoves = new ArrayList<>();
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                if (buttons[i][j].getText().equals("")) {
                    eligibleMoves.add(new int[]{i, j});
                }
            }
        }
        Random random = new Random();
        int randomIndex = random.nextInt(eligibleMoves.size());
        return eligibleMoves.get(randomIndex);
    }
}
```

3.3 Proses Pencarian dengan *Hill Climbing*

3.3.1 Proses Pencarian

Proses pencarian metode *steepest ascent hill climbing* dapat dijelaskan sebagai berikut :

1. Instansiasi bot dengan algoritma *Hill Climbing* yang merupakan *inheritance* dari abstrak kelas bot
2. Untuk tiap *state*, dilakukan pencarian dengan menggunakan fungsi `move()` yang merupakan *inner function* dari bot
3. Fungsi `move()` akan memanggil fungsi *hill climbing* yang mencari nilai terbesar untuk tiap kemungkinan dari *neighbor state*.
4. Bot kemudian akan mengembalikan koordinat dari *neighbor state* untuk melakukan pergerakan.

Berikut merupakan kelas bot dengan algoritma *Hill Climbing* yang merupakan turunan dari *abstract class* bot.

```
public class BotHillClimb extends Bot{
    private int playerNumber;
    /***
     * Move with local search hill climbing algorithm
     * @param buttons matrix of buttons representing the board
     * @return best coordinate to move to
     */
    @Override
    public int[] move(Button[][] buttons, int pNumber) {
        this.playerNumber = pNumber;
        BoardState bs = new BoardState(buttons);
        return hillclimb(bs);
    }
    /***
     * Function to find the best valued step for O in a turn
     * @param input BoardState Object, the state of the game
     * @return BoardState local maximum for O
     */
    public int[] hillclimb(BoardState input) {
        int[] Pos = {-1, -1};
        int Val = Integer.MIN_VALUE;
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                if (input.Positions[i][j] == '_'){
                    BoardState itr;
                    if (this.playerNumber == 1) {
                        itr = new BoardState(input, i, j, 'X');
                    } else {
                        itr = new BoardState(input, i, j, 'O');
                    }
```

```
                    if (itr.evaluateBoard(playerNumber) > Val) {
                        Pos[0] = i;
                        Pos[1] = j;
                        Val = itr.evaluateBoard(playerNumber);
                    }
                }
            }
        }
        return Pos;
    }
}
```

3.3.2 Justifikasi Pemilihan

Pemilihan algoritma jenis ini didasari oleh banyaknya kemungkinan yang harus dicek untuk dilakukan penilaian nilai *state* dari *neighbor*-nya. Selain itu, terdapat kriteria di mana bot dibatasi waktu untuk melakukan perhitungan sehingga langkah paling aman untuk menjamin agar bot dapat memproses dengan waktu dengan stabil adalah algoritma *steepest ascent hill-climbing.* Dengan demikian, bot akan terhindar dari eksplorasi yang terlalu dalam sehingga menyebabkan waktu lebih.

3.4 Proses Pencarian dengan Genetic Algorithm

3.4.1 Proses Pencarian

Cara kerja algoritma *Genetic* adalah sebagai berikut:

1. Membangkitkan sejumlah n spesimen sebagai populasi awal

   Spesimen adalah kelas yang memiliki atribut *chromosome* yang merupakan suatu urutan gerakan yang dibangkitkan secara random. Gerakan ini dibangkitkan dengan tetap memperhatikan validitas gerakan seperti menghindari untuk mengisi petak yang sudah berisi.
2. Menentukan *depth* dari setiap spesimen

   Spesimen dalam melakukan generasi *chromosome* hingga sebanyak nilai tertentu, nilai ini dapat ditentukan dengan menentukan *depth* secara manual, ataupun dengan *contextual* seperti banyak *round* dalam permainan dan lainnya.
3. Melakukan seleksi spesimen dengan *fitness function*, dengan *return value* dari *fitness function* merupakan perhitungan evaluasi keseluruhan *board.*

4. Menghitung probabilitas dari nilai *fitness function*

   Dengan perhitungan setiap nilai *fitness function* spesimen dibagi jumlah nilai *fitness function* keseluruhan spesimen yang ada.

5. Melakukan *random pick* spesimen.

   Random pick beberapa spesimen untuk dilakukan *crossover* dilakukan dengan berdasar pada hasil perhitungan probabilitas pada poin 4.

6. Melakukan *crossover* dari spesimen-spesimen

   Crossover dilakukan pada 2 spesimen dan menerapkan suatu titik *crossing point*. *Crossing point* dibangkitkan secara random dengan aturan rentang *point* terletak pada 0 sampai dengan *depth chromosome*. *Crossover* yang dilakukan adalah *one-point crossover* yaitu *crossover* pada satu titik lalu membagi dua *chromosome* dan melakukan tukar menukar sub-*chromosome* dengan spesimen lain. *Crossover One-point* dipilih dengan alasan *crossover* ini tetap menjaga fitur superior dari spesimen *parent* dan dalam beberapa bahasa pemrograman seperti bahasa C, operasi ini lebih hemat *cost* karena dapat dilaksanakan dengan *pointer*. Hasil *crossover* ini selanjutnya dimasukkan kembali ke populasi spesimen yang awal.

7. Melakukan mutasi kromosom spesimen

   Untuk menghindari langkah ilegal, mutasi pertama diterapkan untuk memeriksa dan menghapus nilai duplikat hasil dari kromosom hasil *crossover*. Nilai yang duplikat diambil yang terletak di *index* lebih akhir, lalu dilakukan random yang memperhatikan validitas gerakan. Dengan begitu, susunan kromosom pada spesimen menjadi valid. Selain itu, jika pada spesimen tidak terdapat langkah ilegal, tetap dilakukan mutasi dari elemen kromosom random menjadi langkah random lain yang memperhatikan validitas.

8. Mengambil spesimen terbaik sebagai keputusan langkah selanjutnya pada permainan

   Dari hasil spesimen yang sudah dilakukan operasi-operasi genetik, kita mengambil hasil dari semua spesimen dengan nilai tertinggi. Spesimen ini lalu, diambil elemennya yang paling pertama yang ditetapkan sebagai *next move* bot.

Berikut merupakan kelas bot dengan algoritma *Genetic* yang merupakan turunan dari *abstract class* bot.

```
public class BotGenetic extends Bot{
    protected ArrayList<Specimen> specimens;
```

```
      public void generateSpecimens(BoardState boardState, int amount, int
depth, int pNumber) {
        specimens = new ArrayList<Specimen>();
        for (int i = 0; i < amount; i++) {
            specimens.add(new Specimen(boardState, depth, pNumber));
        }
    }

    public class Specimen{
                private ArrayList<Pair<Integer, Integer>> chromosome = new
ArrayList<>();
        private int depth;
        private BoardState boardState;
        private int pNumber;
        private int value;
        private int prob;


        public int getProb() {
            return prob;
        }

        public void setProb(int prob) {
            this.prob = prob;
        }

        private Pair<Integer,Integer> generateMovement() {
                        return new Pair<>((int)Math.floor(Math.random()*8),
(int)Math.floor(Math.random()*8));
        }

        public Specimen(BoardState bs, int depth, int pNumber){
            this.boardState = new BoardState(bs);
            this.depth = depth;
            this.pNumber = pNumber;
            this.generateChromosome();
        }

        public Specimen(Specimen copy){
            this.boardState = new BoardState(copy.boardState);
            this.depth = copy.depth;
            this.pNumber = copy.pNumber;
            this.generateChromosome();
        }

        public Pair<Integer, Integer> generateValidMove() {
            Pair<Integer, Integer> candidate;
            do {
                candidate = generateMovement();
            } while (!boardState.isValidMove(candidate));
            return candidate;
        }

        public void generateChromosome() {
            for (int i = 0; i < depth; i++) {
                chromosome.add(generateValidMove());
```

```
            }
        }

        /**
         * One-point crossover genetic operation
         * @param other Specimen to cross over with
         * @return Offspring specimens
         */
        public ArrayList<Specimen> crossoverWith(Specimen other) {
            ArrayList<Specimen> offsprings = new ArrayList<>();
            int crossPoint = (int) Math.floor(Math.random()*(this.depth-1));
            Specimen offspring1 = new Specimen(this);
            Specimen offspring2 = new Specimen(other);
            for (int i = 0; i < chromosome.size(); i++) {
                if (i > crossPoint) {
                    offspring1.chromosome.set(i, other.chromosome.get(i));
                    offspring2.chromosome.set(i, this.chromosome.get(i));
                }
            }
            offsprings.add(offspring1);
            offsprings.add(offspring2);
            offsprings.get(0).duplicateMutation();
            offsprings.get(1).duplicateMutation();
            return offsprings;
        }

        public void mutate(int mutatePoint) {
            this.chromosome.set(mutatePoint, generateValidMove());
        }

        public void duplicateMutation() {
                     Map<Pair<Integer,Integer>,  Integer>  frequencyMap  =  new
HashMap<>();
            for (Pair<Integer, Integer> c :
                    this.chromosome) {
                frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);
            }
            // if there are duplicates, change last indexes of duplicates into
something valid
            for (Pair<Integer, Integer> c : frequencyMap.keySet()) {
                while (frequencyMap.get(c) != 1) {
                    this.mutate(this.chromosome.lastIndexOf(c));
                    // assuming the set operation will take care of duplicate,
frequency would be subtracted by 1
                    frequencyMap.replace(c, frequencyMap.get(c)-1);
                }
            }
        }
        @Override
        public int hashCode() {
            return Objects.hash(this.chromosome, this.boardState, this.depth);
        }
        /**
         * @return total value
         */
        public int getLeafValue(){
```

```
            BoardState bs = new BoardState(this.boardState);
            for (int i = 0; i < this.chromosome.size(); i++) {
                if (this.pNumber == 1){
                      bs = new BoardState(bs, this.chromosome.get(i).getKey(),
this.chromosome.get(i).getValue(), 'X');
                } else {
                      bs = new BoardState(bs, this.chromosome.get(i).getKey(),
this.chromosome.get(i).getValue(), 'O');
                }
            }
            return bs.evaluateBoard(this.pNumber);
        }

        public int getValue(){
            return this.value;
        }

        public void setValue(int value){
            this.value = value;
        }

    }

    public void fitnessFunction(Button[][] buttons, int pNumber){
        BoardState bs = new BoardState(buttons);
        int p = pNumber;
        for (int i = 0; i < 4; i++) {
            if (p == 1){
                specimens.add(new Specimen(bs, 8, p));
                specimens.get(i).setValue(specimens.get(i).getLeafValue());
                p = 2;
            } else {
                specimens.add(new Specimen(bs, 8, p));
                specimens.get(i).setValue(specimens.get(i).getLeafValue());
                p = 1;
            }
        }
    }

    /**
     * calculate probability of the specimen's fitness
     */
    public void calculateProbs(){
        int total = 0;
        for (Specimen specimen : specimens) {
            total += specimen.getValue();
            if (specimen.getValue() == 0) {
                specimen.setProb(25);
                total += 25;
            }
            specimen.setProb((int) Math.ceil(specimen.getValue() / total));
        }
    }


    /**
```

```
     *
     */

    @Override
    public int[] move(Button[][] buttons, int pNumber) {
        ArrayList<Specimen> parent = new ArrayList<>();
        ArrayList<Specimen> crossOverSpecimen = new ArrayList<>();
        generateSpecimens(new BoardState(buttons), 4, 8, pNumber);
        int val = 0;
        fitnessFunction(buttons, pNumber);
        calculateProbs();
        for (int i = 0; i < specimens.size(); i++) {
            val = (int) Math.ceil(Math.random()*100);
            if (val <= specimens.get(0).getProb()){
                parent.add(specimens.get(0));
                        } else if (val <= specimens.get(0).getProb() +
specimens.get(1).getProb() &&
                        val > specimens.get(0).getProb()) {
                parent.add(specimens.get(1));
                        } else if (val <= specimens.get(1).getProb() +
specimens.get(2).getProb() &&
                                        val > specimens.get(0).getProb() +
specimens.get(1).getProb()) {
                parent.add(specimens.get(2));
            } else {
                parent.add(specimens.get(3));
            }
        }

        crossOverSpecimen.addAll(parent.get(0).crossoverWith(parent.get(1)));
        crossOverSpecimen.addAll(parent.get(2).crossoverWith(parent.get(3)));

        int maxVal = Integer.MIN_VALUE;
        int index = -1;

        for (int i = 0; i < crossOverSpecimen.size(); i++) {

crossOverSpecimen.get(i).mutate((int)Math.floor(Math.random()*8.0));
            crossOverSpecimen.get(i).duplicateMutation();
            if (maxVal < crossOverSpecimen.get(i).getLeafValue()){
                maxVal = crossOverSpecimen.get(i).getLeafValue();
                index = i;
            }
        }


                                                    return new int[]
{crossOverSpecimen.get(index).chromosome.get(0).getKey(),
crossOverSpecimen.get(index).chromosome.get(0).getValue()};
    }

    public ArrayList<Specimen> getSpecimens() {
        return specimens;
    }

    public void setSpecimens(ArrayList<Specimen> specimens) {
```

```
        this.specimens = specimens;
    }
}
```

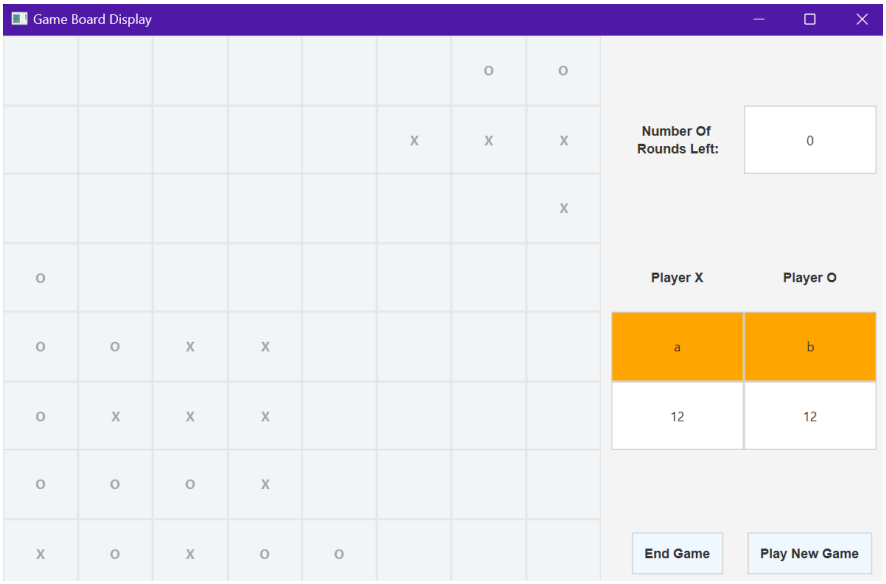
3.4.2 Justifikasi Pemilihan

Algoritma *Genetic* dapat digunakan untuk mencari solusi berdasarkan kombinasi gerakan yang lebih baik secara genetik. Algoritma Genetik memungkinkan eksplorasi beragam solusi menggunakan proses seleksi, *crossover*, dan mutasi. Selain itu, algoritma genetik memungkinkan perpaduan variasi genetik melalui *crossover* yang dapat membantu mencari solusi yang lebih baik. Meskipun algoritma genetik tidak secara lazim digunakan untuk konteks *adversial search*, algoritma ini unggul ketika tingkat kedalaman dan banyak spesimen ditingkatkan dengan skala besar. Besar *overhead* dari algoritma genetik menjadi lebih hemat dibandingkan algoritma lain seperti algoritma *minimax* dengan hasil yang lebih akurat dibandingkan algoritma *local search*.

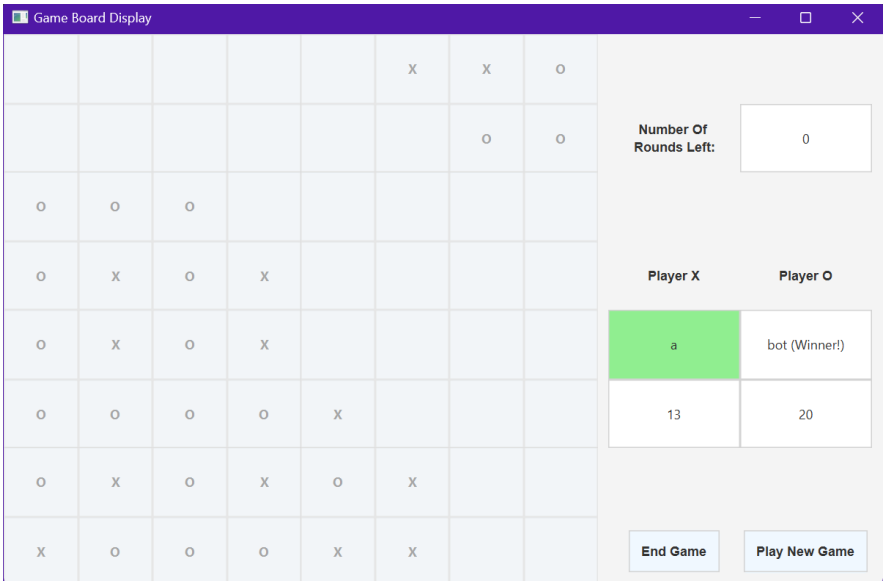
BAB IV

4.1 Minimax vs Manusia

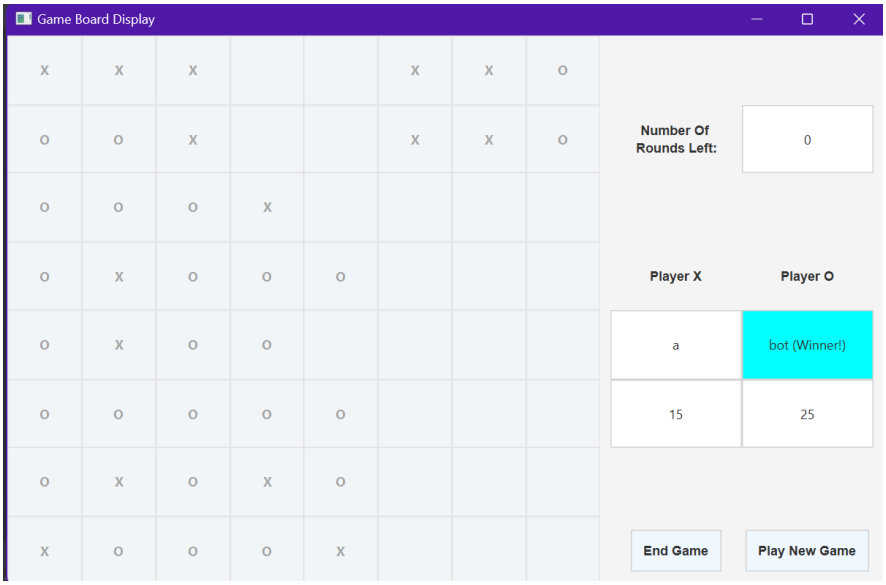
4.1.1 8 Ronde



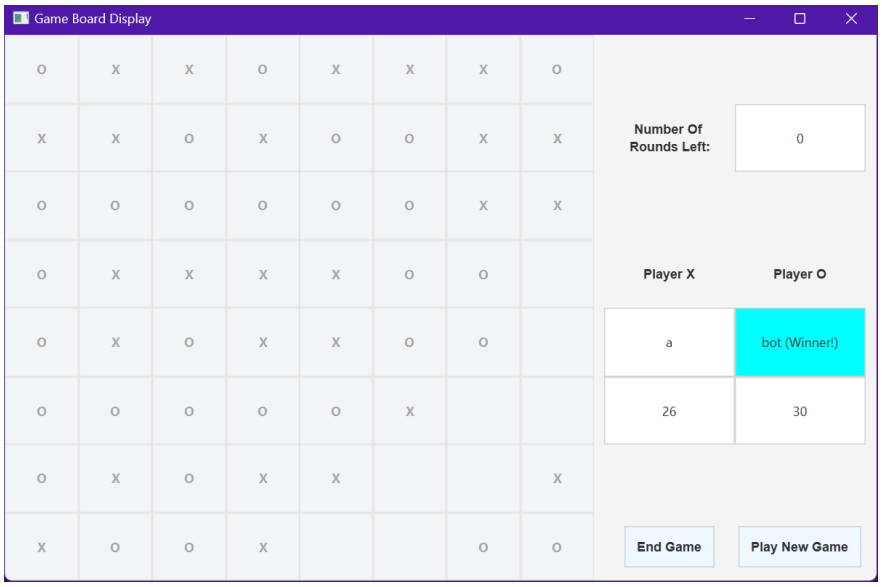
Pada gambar diatas yang merupakan pertandingan pertama antara Manusia dengan Bot Minimax dengan jumlah ronde 8, didapatkan hasil seri antara Bot Minimax dengan Manusia dengan hasil akhir 12-12.



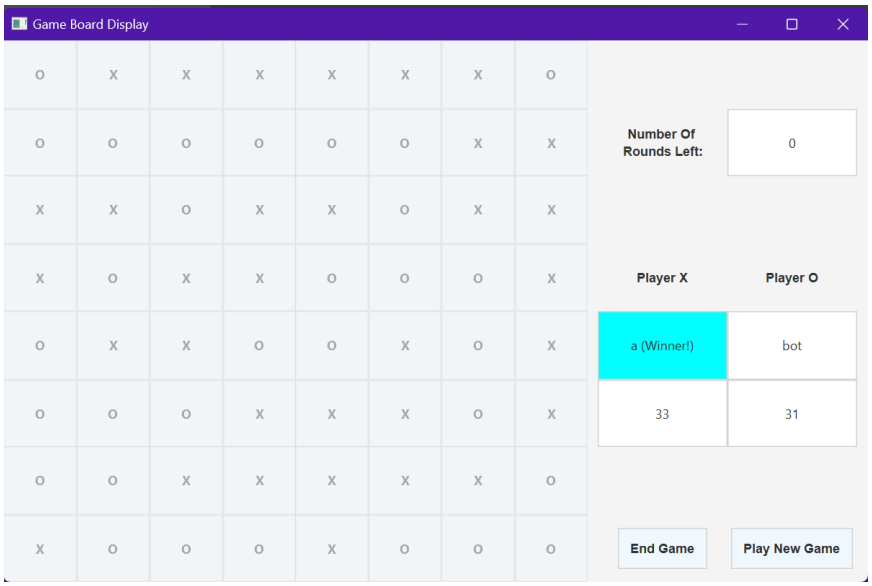
Pada gambar diatas yang merupakan pertandingan kedua antara Manusia dengan Bot Minimax dengan jumlah ronde 8, didapatkan Bot Minimax meraih kemenangan, dengan hasil akhir 13-20.



Pada gambar diatas yang merupakan pertandingan ketiga antara Manusia dengan Bot Minimax dengan jumlah ronde 16, didapatkan Bot Minimax meraih kemenangan dengan hasil akhir 15-25.



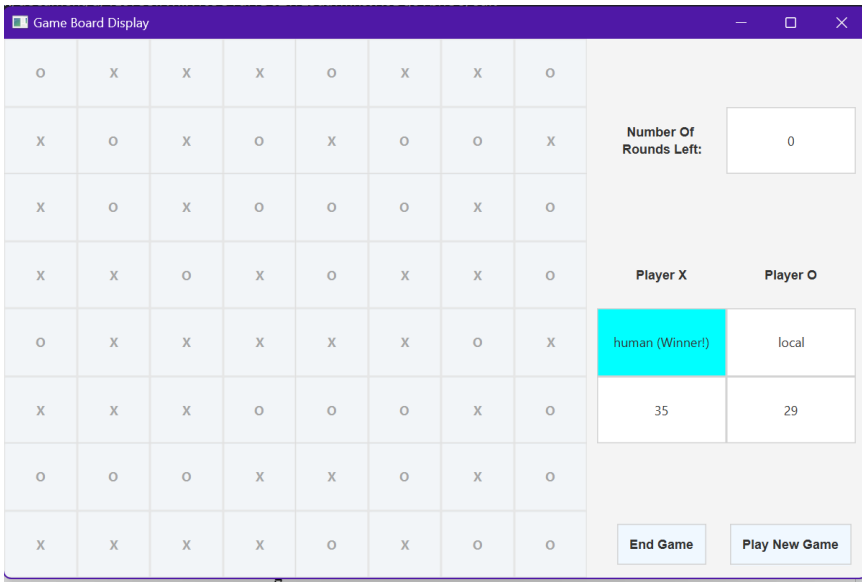
Pada gambar diatas yang merupakan pertandingan keempat antara Manusia dengan Bot Minimax dengan jumlah ronde 24, didapatkan Bot Minimax meraih kemenangan dengan hasil akhir 26-30.



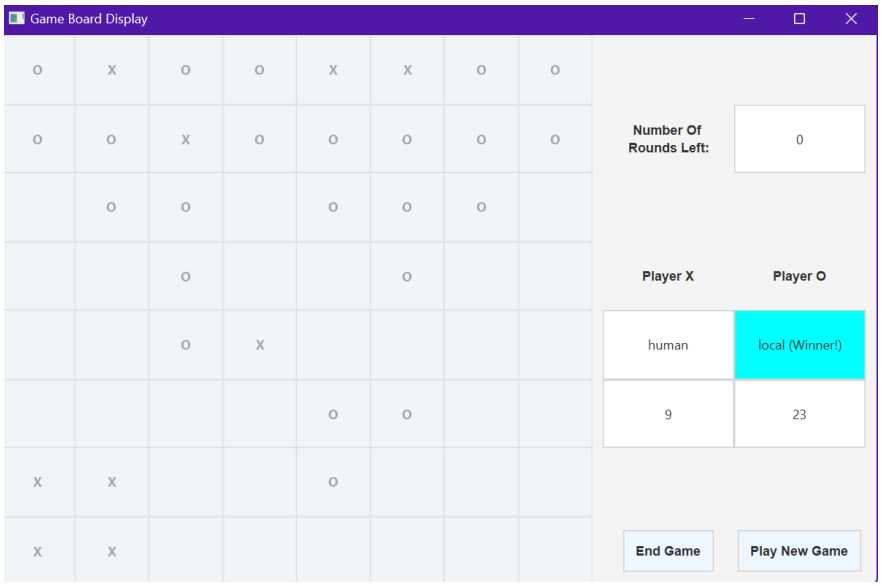
Pada gambar diatas yang merupakan pertandingan terakhir antara Manusia dengan Bot Minimax dengan jumlah ronde 24, didapatkan Manusia meraih kemenangan dengan hasil akhir 33-31.

Dapat disimpulkan pada 5 pertandingan yang telah dilakukan antara Manusia dengan Bot Minimax didapatkan hasil bahwa Bot Minimax memenangkan pertandingan sebanyak 3 kali, Manusia 1 kali dan seri 1 kali.

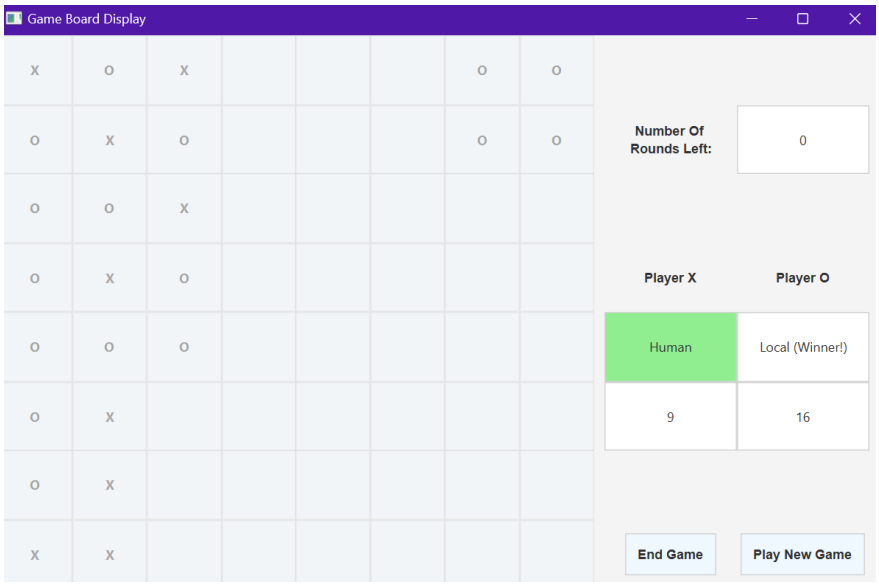
4.2 Local Search vs Manusia



Gambar di atas adalah permainan dari bot dengan algoritma local search melawan pemain manusia dengan memainkan 28 rounds, hasil akhir dari permainan tersebut adalah 35-29 kemenangan pemain manusia



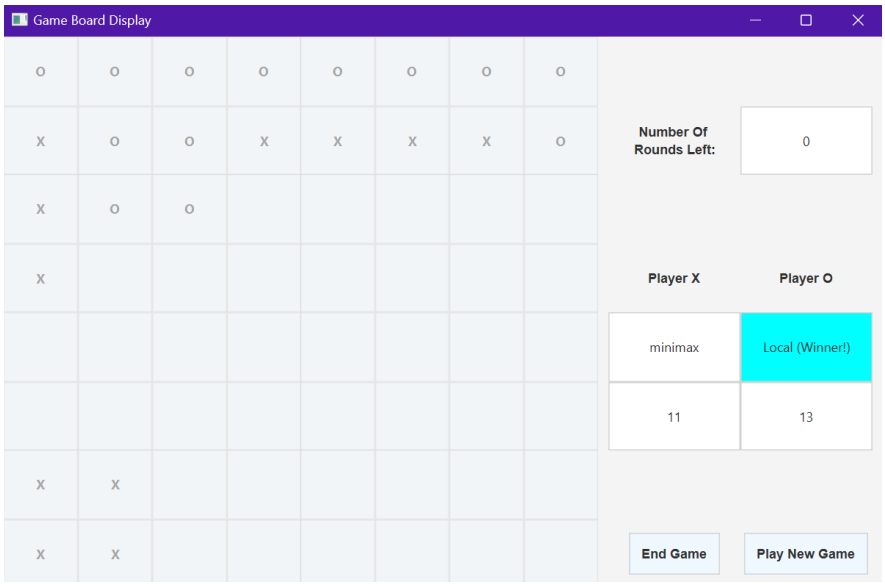
Gambar di atas adalah permainan dari bot dengan algoritma local search melawan pemain manusia dengan memainkan 12 rounds, hasil akhir dari permainan tersebut adalah 9-23 kemenangan bot local search



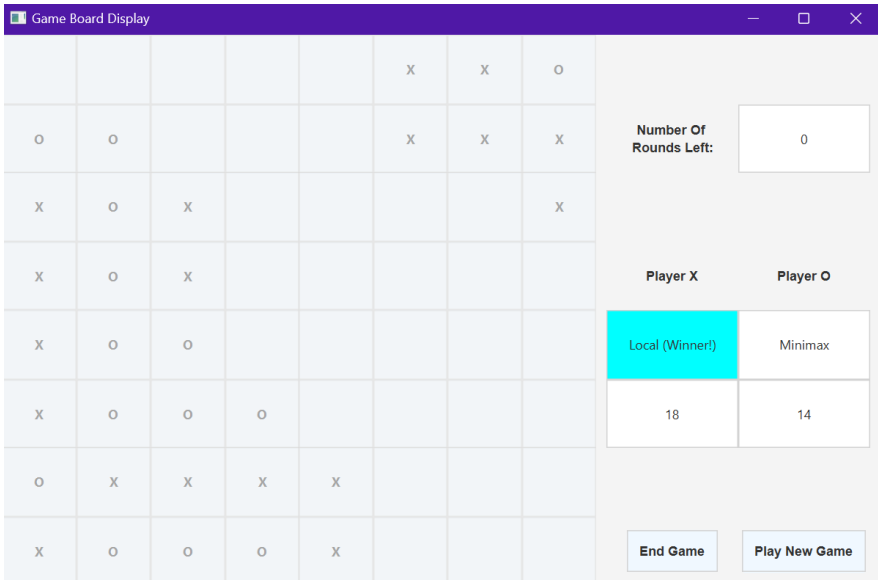
Gambar di atas adalah permainan dari bot dengan algoritma local search melawan pemain manusia dengan memainkan 8 rounds, hasil akhir dari permainan tersebut adalah 9-16 kemenangan pemain manusia.

Dapat disimpulkan pada 3 pertandingan yang telah dilakukan antara Manusia dengan Bot Local Search didapatkan hasil bahwa Bot Search memenangkan pertandingan sebanyak 1 kali, Manusia 2 kali, dan tanpa ada seri.

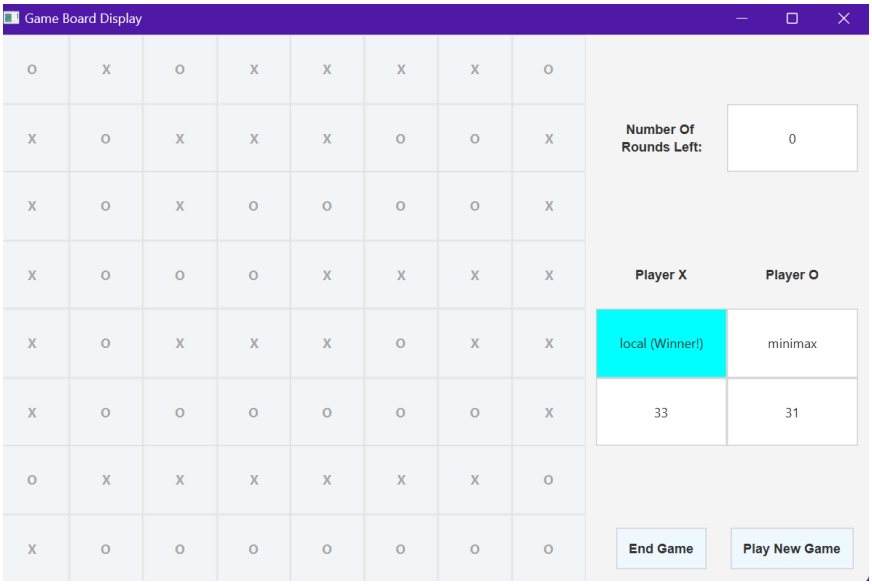
4.3 Minimax vs Local Search



Gambar di atas adalah permainan dari bot dengan algoritma local search melawan bot dengan algoritma minimax dengan memainkan 8 rounds, hasil akhir dari permainan tersebut adalah 11-13 kemenangan bot local search



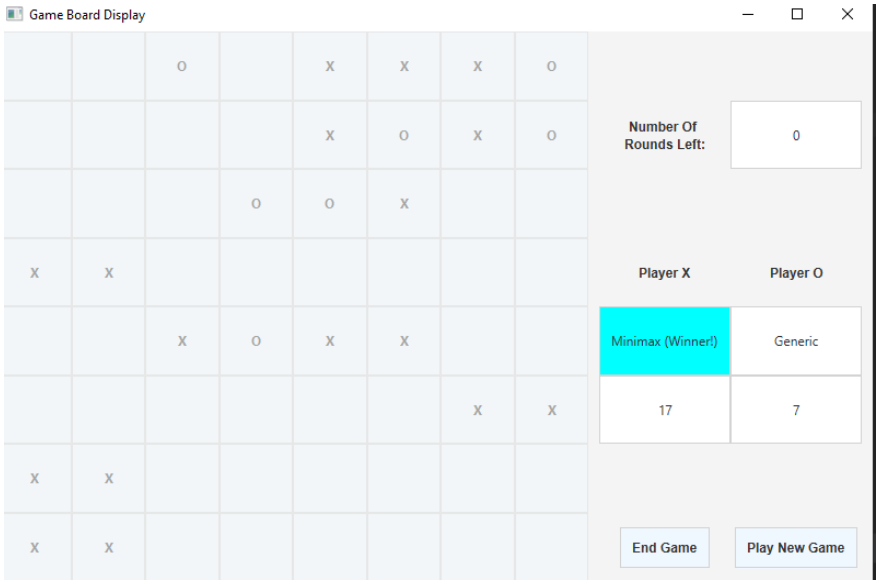
Gambar di atas adalah permainan dari bot dengan algoritma local search melawan bot dengan algoritma minimax dengan memainkan 12 rounds, hasil akhir dari permainan tersebut adalah 18-14 kemenangan bot local search



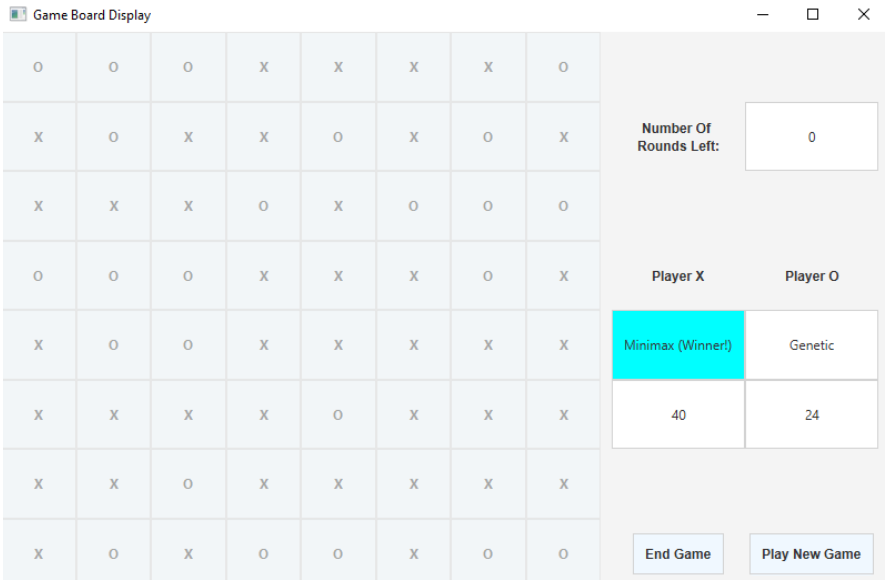
Gambar di atas adalah permainan dari bot dengan algoritma local search melawan bot dengan algoritma minimax dengan memainkan 28 rounds, hasil akhir dari permainan tersebut adalah 33-31 kemenangan bot local search.

Dapat disimpulkan pada 3 pertandingan yang telah dilakukan antara Bot Minimax dengan Bot Local Search didapatkan hasil bahwa Bot Local Search memenangkan seluruh pertandingan.

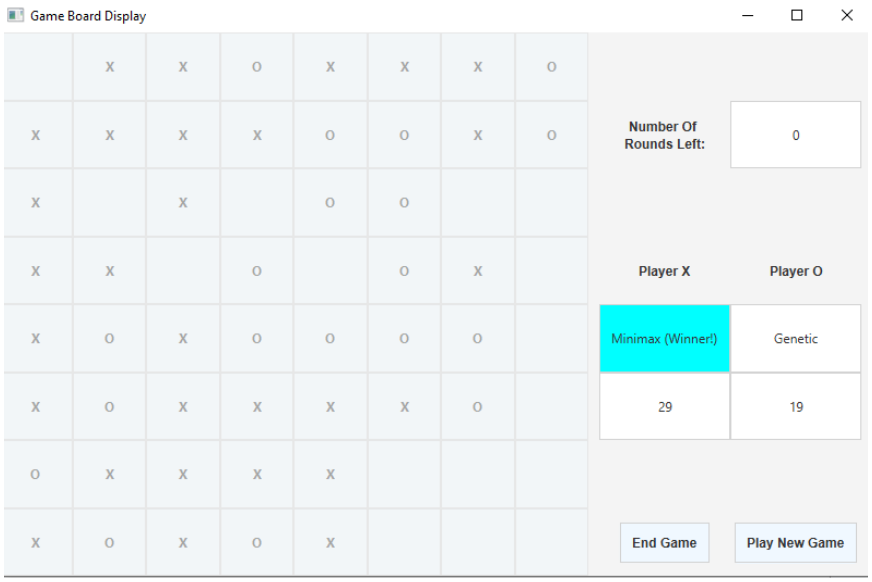
4.4 Minimax vs Genetic Algorithm



Gambar di atas adalah permainan dari bot dengan algoritma minimax melawan bot dengan algoritma genetik dengan memainkan 8 rounds, hasil akhir dari permainan tersebut adalah 17-7 kemenangan bot minimax



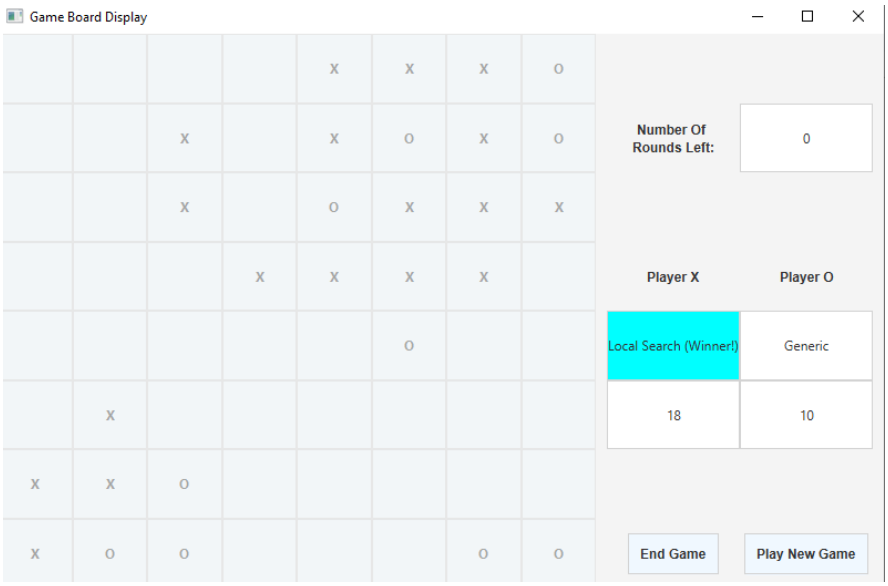
Gambar di atas adalah permainan dari bot dengan algoritma minimax melawan bot dengan algoritma genetik dengan memainkan 28 rounds, hasil akhir dari permainan tersebut adalah 40-24 kemenangan bot minimax.



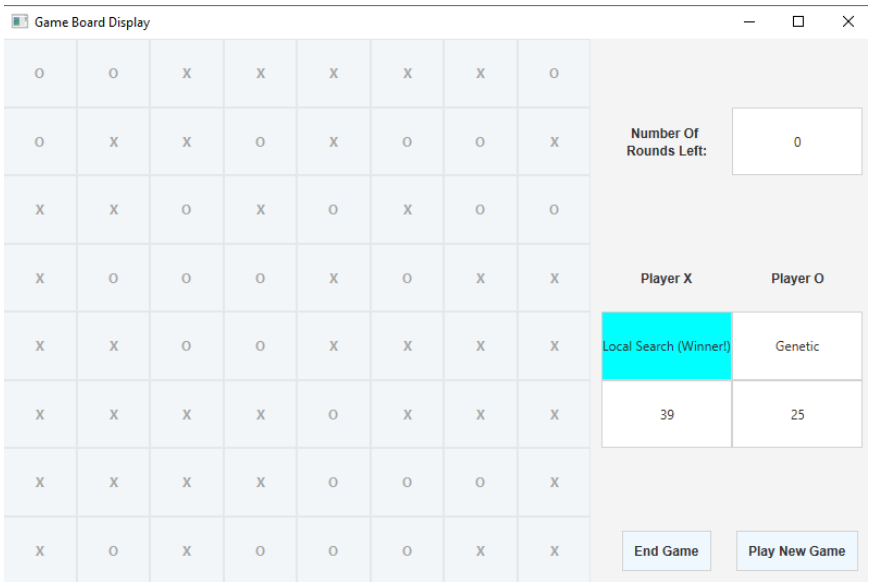
Gambar di atas adalah permainan dari bot dengan algoritma minimax melawan bot dengan algoritma genetik dengan memainkan 20 rounds, hasil akhir dari permainan tersebut adalah 29-19 kemenangan bot minimax.

Dapat disimpulkan pada 3 pertandingan yang telah dilakukan antara Bot Minimax dengan Bot Genetic didapatkan hasil bahwa Bot Minimax memenangkan seluruh pertandingan.

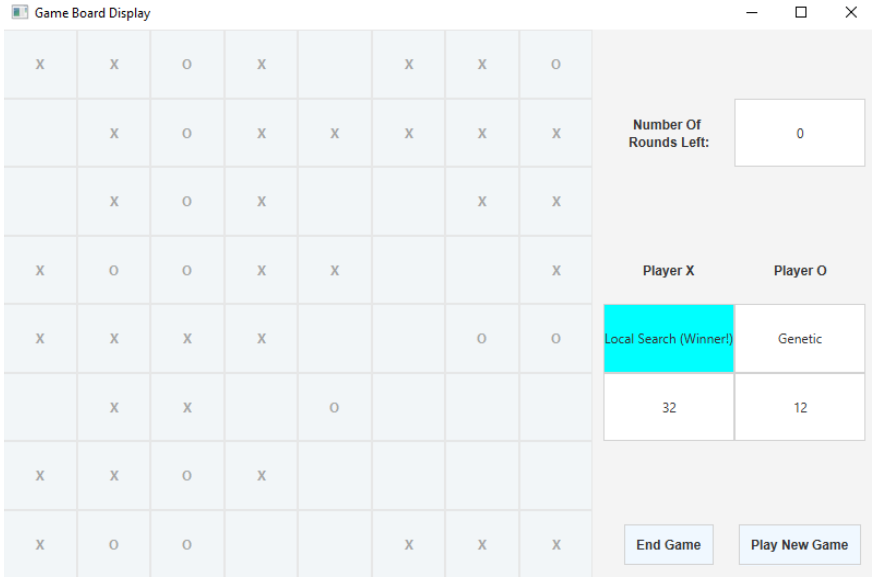
4.5 Local Search vs Genetic Algorithm



Gambar di atas adalah permainan dari bot dengan algoritma local search melawan bot dengan algoritma genetik dengan memainkan 10 rounds, hasil akhir dari permainan tersebut adalah 18-10 kemenangan bot Local Search



Gambar di atas adalah permainan dari bot dengan algoritma local search melawan bot dengan algoritma genetik memainkan 28 rounds, hasil akhir dari permainan tersebut adalah 39-25 kemenangan bot Local Search



Gambar di atas adalah permainan dari bot dengan algoritma local search melawan bot dengan algoritma genetik dengan memainkan 18 rounds, hasil akhir dari permainan tersebut adalah 32-12 kemenangan bot Local Search.

Dapat disimpulkan pada 3 pertandingan yang telah dilakukan antara Bot Genetic dengan Bot Local Search didapatkan hasil bahwa Bot Local Search memenangkan seluruh pertandingan.

4.6 Persentase Kemenangan Tiap Bot

1. Local Search Bot: 88.89% (8/9)
2. Minimax Bot: 54.54% (6/11)
3. Genetic Bot: 0% (0/6)

BAB V

5.1 Kesimpulan

Persentase kemenangan algoritma *minimax* melawan manusia dari 5 pertandingan adalah 60%. Maka dapat disimpulkan algoritma *minimax* bekerja dengan baik melawan manusia pada *Adjacency Strategy Game*.

Persentase kemenangan algoritma *local search* melawan manusia dari 3 pertandingan adalah 66.67%. Maka dapat disimpulkan algoritma *local search* bekerja dengan baik melawan manusia pada *Adjacency Strategy Game*.

Persentase kemenangan algoritma *local search* melawan algoritma *minimax* dari 3 pertandingan adalah 100%. Maka dapat disimpulkan algoritma *local search* bekerja dengan baik melawan algoritma *minimax* pada *Adjacency Strategy Game*.

Persentase kemenangan algoritma *minimax* melawan algoritma *genetic* dari 3 pertandingan adalah 0%. Maka dapat disimpulkan algoritma *minimax* bekerja dengan baik melawan algoritma *genetic* pada *Adjacency Strategy Game*.

Persentase kemenangan algoritma *local search* melawan algoritma *genetic* dari 3 pertandingan adalah 0%. Maka dapat disimpulkan algoritma *local search* bekerja dengan baik melawan algoritma *genetic* pada *Adjacency Strategy Game*.

5.2 Saran

Pada algoritma yang telah kami buat ada beberapa pilihan algoritma. Untuk algoritma hill-climbing, terdapat beberapa pilihan lain seperti penggunaan stochastic hill-climbing dan random restart hill-climbing untuk dilakukan eksplorasi lebih lanjut. Fungsi heuristik dari algoritma Minimax juga dapat dilakukan eksplorasi lebih lanjut, selanjutnya penggunaan fungsi objektif juga sangat mempengaruhi kinerja dari algoritma karena fungsi objektif yang menilai *value* dari sebuah *state*. Untuk *Genetic Algorithm*, diperlukan rekursifitas yang semakin banyak rekursifnya maka nilai *fitness function*-nya semakin tinggi sehingga nilai *fitness function*-nya mendekati nilai global optimal.

5.3 Kontribusi Kelompok

13521004: Laporan, *Minimax Algorithm, Java Frame GUI.*

13521009: Laporan, *Genetic Algorithm, Hill-Climbing.*

13521012: Laporan, *Genetic Algorithm.*

13521117: Laporan, *Hill-Climbing.*