LAPORAN TUGAS BESAR 1

Minimax Algorithm and Alpha Beta Pruning in Adjacency Strategy Game

Disusun Oleh:

Fahrian Afdholi	13521031
Fakih AP	13521091
Haidar Hamda	13521105
Ammar Rasyad C	13521136

1. Penjelasan *objective function*

Objective function adalah sebuah ukuran untuk mengukur seberapa baik kinerja yang dihasilkan oleh sebuah model, diukur dari seberapa tepat hasil yang dikeluarkan dengan target capaian. Menggunakan *objective function*, kita bisa menentukan apakah model yang sedang diuji sudah sesuai dengan target kinerja yang kita harapkan.

Dalam mengukur kesesuaian nilai objektif *problem solving agent*, umumnya kita dapat menggunakan dua cara, yaitu :

a. Cost function

   Nilai fungsi akan ditentukan dari seberapa besar ketidaksesuaian nilai dari state yang sekarang dengan state yang diharapkan. Nilai yang dihasilkan biasanya berada dalam *range* bilangan negatif dengan nilai 0 sebagai *value state* optimalnya.

   Contoh: n-queen problems.

b. Reward function

   Nilai akan ditentukan dari seberapa baik kinerja yang dihasilkan oleh model dengan melihat nilai state yang saat ini dicapai. Hasil nilai yang dikeluarkan biasanya dalam bilangan positif dengan nilai yang lebih tinggi menunjukkan kinerja yang lebih baik. Namun, tidak menutup kemungkinan fungsi bisa menghasilkan nilai negatif.

   Contoh: adjacency strategy game.

2. Implementasi *Minimax Alpha Beta Pruning*

2.1. Definisi

Minimax Alpha Beta Pruning adalah sebuah algoritma yang melakukan pengecekan secara *backtracking* dengan membandingkan nilai dari anak-anaknya. Akan tetapi, untuk perbandingannya algoritma ini akan membandingkan nilai maksimum dan nilai minimum dari anak-anaknya tergantung pada kondisinya. Pada kondisi di saat pengecekan anak pertama akan dilakukan perbandingan nilai maksimum dari seluruh nilai yang ada setelahnya akan membandingkan nilai minimum dari seluruh nilai dari anak-anaknya tahapan maksimum dan minimum sampai seterusnya

pada anak paling akhir. Pada algoritma ini memiliki kompleksitas yang sama dengan algoritma *Minimax* biasa karena jika seluruh nilai terbaik terdapat pada paling akhir dia akan memiliki kompleksitas yang sama yaitu O(n!).

2.2. Implementasi

Implementasi dari kode *Minimax Alpha Beta Pruning* pada Adjacency Strategy Game adalah sebagai berikut:

Method move()

```
   @Override
    public int[] move() {
        Board board = new Board(controller, player);
        ExecutorService executorService =
Executors.newSingleThreadExecutor();
        Future<Object[]> temp = executorService.submit(() ->
minimaxValue(new Object[]{board, new int[]{0, 0}}, true,
MIN_VAL, MAX_VAL, controller.roundsLeft));
        int[] val = new int[2];
        try {
            Object[] minimax = temp.get();
            Object[] value = (Object[]) minimax[0];
            return (int[]) value[1];
        } catch (InterruptedException | ExecutionException e)
{
            return val;
        }
    }
```

Method akan memanggil method yang menghitung langkah yang tepat menggunakan minimax algorithm

Method minimaxValue()

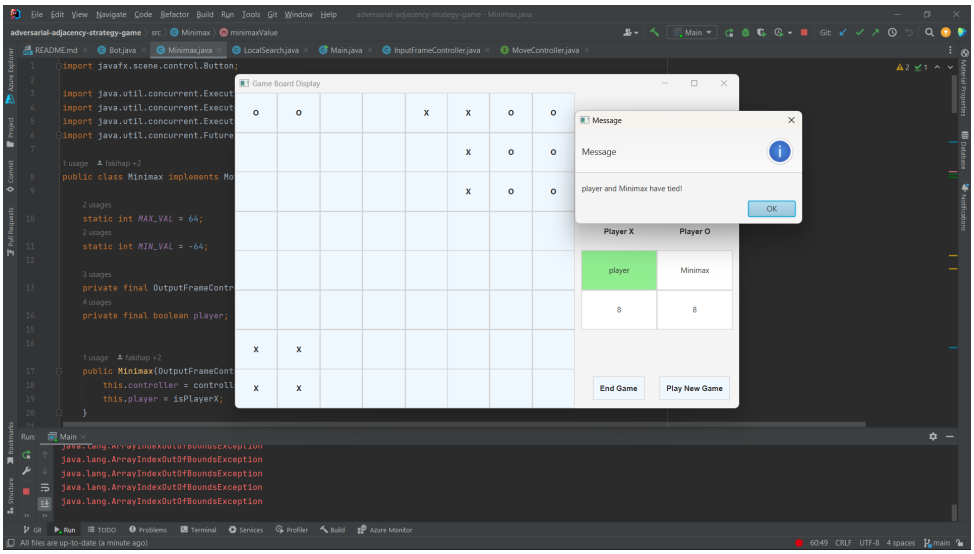
```
// find optimal value of all boards using minimax algorithm
```

```
and a-b pruning
    public Object[] minimaxValue(Object[] nodeValues, boolean
isMaximizing, int alpha, int beta, int round) {
        Board board = (Board) nodeValues[0];
        Object[] bestBranch = new Object[]{nodeValues, 0};

        if (board.isBoardFull()) {
            return new Object[]{nodeValues,
board.getFunctionValue()};
        }
        if (round == 0) {
            return new Object[]{nodeValues,
board.getFunctionValue()};
        }

        int bestVal;
        if (isMaximizing) {
            Object[][] branch =
board.getPossibleChilds(this.player);
            bestVal = MIN_VAL;

            for (Object[] objects : branch) {
                Object[] currentVal = minimaxValue(objects,
false, alpha, beta, round - 1);
                int temp = (int) currentVal[1];

                bestVal = Math.max(bestVal, temp);
                alpha = Math.max(alpha, bestVal);
                if (alpha == temp) bestBranch = currentVal;

                if (beta <= alpha) {
                    break;
                }
            }

        } else {
            Object[][] branch =
board.getPossibleChilds(!this.player);
            bestVal = MAX_VAL;
```

```
            for (Object[] objects : branch) {
                Object[] currentVal = minimaxValue(objects,
true, alpha, beta, round - 1);
                int temp = (int) currentVal[1];

                bestVal = Math.min(bestVal, temp);
                beta = Math.min(beta, bestVal);
                if (beta == temp) bestBranch = currentVal;

                if (beta <= alpha) {
                    break;
                }
            }

        }
        return bestBranch;
    }
```

2.3. Hasil Implementasi

Dengan mencobanya pada *player vs bot* didapatkan hasil sebagai berikut:



Pada bot *Minimax Alpha Beta Pruning* selalu mencari nilai terbaik

yang didapat dari analisis algoritmanya sehingga *player* akan sulit untuk menang karena algoritma ini mencari nilai kemungkinan terbaik pula si player bergerak dan menganalisis bagaimana untuk mendapatkan poin semaksimal mungkin dan poin player seminimum mungkin.

3. Implementasi *Local Search*

3.1. Definisi

Hill Climbing adalah algoritma yang awalnya memilih solusi secara random, lalu secara iteratif mencari solusi yang memaksimalkan value, atau meminimalisir cost, untuk mendapatkan solusi dengan value yang paling tinggi ("peak").

Metode *Hill climbing* yang dipilih dalam algoritma yang dibuat adalah *sideways move hill climbing*. Algoritma ini dipilih karena deterministik dan lebih efisien daripada algoritma Local Search lainnya saat digunakan dalam kasus ini. Namun dalam kasus ini terdapat *fallback* dimana bot akan memilih petak secara *random* jika algoritma memilih tempat yang sudah terpilih

3.2. Implementasi

Method move()

```
// Move using Local Search Algorithm
    @Override
    public int[] move() {
        // set initial state
        Button[][] board = controller.getButtons();

        if (pair == null) {
            pair = new MovePair((int) (Math.random() * 8), (int) (Math.random() *
8));
        }
```

```
        final MovePair[] neighbors = new MovePair[4]; // 0 = left, 1 = right, 2 =
up, 3 = down

        // check for surrounding filled squares
        int surrounding = getSurrounding(pair, neighbors, board);

        MovePair temp = switch (surrounding) {
            case 1, 2, 3 -> new MovePair(heuristicCheck(neighbors, surrounding,
board));
            default -> new MovePair(pair.a, pair.b);
        };

        // prevent object reference from changing
        pair.a = temp.a;
        pair.b = temp.b;

        // check if the square is filled
        while (board[pair.a][pair.b].getText().equals(player) ||
board[pair.a][pair.b].getText().equals(opponent)) {
            // if the square is filled, generate a new random square regardless of
surrounding filled squares
            pair.a = (int) (Math.random() * 8);
            pair.b = (int) (Math.random() * 8);
        }

        return new int[]{ pair.a, pair.b };
    }
```

Bot pertama-tama akan menentukan titik target awal secara random. Setelah itu, bot akan menentukan kesesuaian titik yang dipilih menggunakan sebuah fungsi heuristik dan menentukan titik yang akan dipilih. Untuk menghindari kesalahan, jika titik yang dipilih ternyata telah diisi, bot akan mengacak kembali titik yang akan dipilih.

Method heuristicCheck()

```
    // meta-heuristic algorithm
    private int[] heuristicCheck(MovePair[] neighbors, int surrounding, Button[][]
```

```
board) {
        MovePair point = new MovePair(pair.a, pair.b);

        for (MovePair neighbor : neighbors) {
            if (neighbor != null) {
                point = getEmptySquare(neighbor, surrounding, board);
            }
        }

        return new int[]{ point.a, point.b };
    }
```

Menggunakan informasi yang disediakan method ini akan menentukan petak kosong yang paling sesuai dengan menggunakan perhitungan tertentu.

Method getEmptySquare()

```
 private MovePair getEmptySquare(MovePair neighbor, int surrounding, Button[][]
board) {
        int highestSurrounding = -1;
        MovePair point = new MovePair(pair.a, pair.b);

        for (int i = neighbor.a - 1; i <= neighbor.b; i += 2) {
            for (int j = neighbor.b - 1; j <= neighbor.b; j += 2) {
                if (i >= 0 && i < 8 && j >= 0 && j < 8) {
                    // get the highest surrounding heuristic
                    MovePair localPoint = new MovePair(i, j);
                    int localSurrounding = getSurrounding(localPoint, new
MovePair[4] /* ignore */, board);
                    if (highestSurrounding < localSurrounding) {
                        highestSurrounding = localSurrounding;
                        point = localPoint;
                    }
                }
            }
        }
```

```
        if (highestSurrounding > surrounding) {
            return point;
        } else {
            return new MovePair(pair.a, pair.b);
        }
    }
```

Menggunakan informasi yang disediakan, method ini akan menentukan petak kosong yang paling sesuai dengan melakukan penghitungan jumlah petak terisi yang menjadi tetangga petak yang akan dipilih.

Method getSurrounding()

```
private int getSurrounding(MovePair point, MovePair[] neighbors, Button[][] board)
{
        // check if the surrounding squares are filled by other players
        int surrounding = 0;

        if (point.a > 0 && board[point.a - 1][point.b].getText().equals(opponent))
{
            neighbors[2] = new MovePair(point.a - 1, point.b);
            surrounding++;
        }

        if (point.a < 7 && board[point.a + 1][point.b].getText().equals(opponent))
{
            neighbors[3] = new MovePair(point.a + 1, point.b);
            surrounding++;
        }

        if (point.b > 0 && board[point.a][point.b - 1].getText().equals(opponent))
{
            neighbors[0] = new MovePair(point.a, point.b - 1);
            surrounding++;
        }

        if (point.b < 7 && board[point.a][point.b + 1].getText().equals(opponent))
```

```
{
            neighbors[1] = new MovePair(point.a, point.b + 1);
            surrounding++;
        }

        return surrounding;
    }
```

Digunakan untuk menghitung banyak tetangga yang terisi

Class MovePair

```
    private static class MovePair {
        int a;
        int b;

        public MovePair(int[] array) {
            this(array[0], array[1]);
        }

        public MovePair(int a, int b) {
            this.a = a;
            this.b = b;
        }
    }
```

Kelas yang merepresentasikan petak yang akan dipilih

3.3. Hasil Implementasi

4. Implementasi *Genetic Algorithm*

Pada pengimplementasian genetic algorithm dalam penyelesaian permasalahan adversarial search bisa memanfaatkan koordinat dalam pembentukan kromosom nya. Beberapa penjelasan terkait peng-implementasian genetic algorithm sebagai berikut:

1. Kromosom
   Sebuah kromosom didefinisikan sebagai urutan koordinat langkah yang dipilih
2. Nilai fitness
   Nilai fitness didefinisikan sebagai seberapa seberapa tinggi rasio perbandungan jumlah simbol bot dengan lawan pada suatu keadaan
3. Crossover
   Crossover adalah proses penggabungan 2 kromosom untuk membentuk generasi baru. Sebagai contoh 2 kromosom 123456 dan 987654 dilakukan crossover dengan crossover point 4, maka akan terbentuk 123654 dan 987456
4. Mutasi
   Mutasi adalah proses untuk menukar 1 bagian kromosom dengan yang lain secara acak dengan harapan mendapatkan kromosom dengan nilai fitness lebih baik

Implementasi kode sebagai berikut

Method move()

```
    @Override
    public int[] move(){
        Button[][] board= controller.getButtons();
        int n=board.length;
        int m=board[0].length;
//        List<int[]> emptySquares=getEmptySquares(board);
        String[][] boardString=new String[n][m];
        for (int i=0;i<n;i++){
            for (int j=0;j<m;j++){
                boardString[i][j]=board[i][j].getText();
            }
        }
        String botSymbol="O";
        String opponentSymbol="X";
        if (this.isPlayerX){
            botSymbol="X";
            opponentSymbol="O";
        }
        List<Chromosome> chromosomes=new ArrayList<Chromosome>();
        List<Chromosome> ret=generate(boardString,chromosomes,false,botSymbol, opponentSymbol,2);
        List<Chromosome> rett=fitnessFunction(ret);
//        Chromosome chromosome1 = select(rett);
//        Chromosome chromosome2 = select(rett);
//        List<Chromosome> crossOvers=crossover(chromosome1,chromosome2,0);

//        for (Chromosome r :
//                rett) {
//            System.out.println(r.toString());
//        }
        List<Chromosome> newChromosome=newCrossover(boardString,rett,1,botSymbol, opponentSymbol);
        for (int i=0;i<10;i++){
            newChromosome=newCrossover(boardString,newChromosome,1,botSymbol,opponentSymbol);
        }
        for (Chromosome c :
                newChromosome) {
            System.out.println(c.toString());
        }

        return newChromosome.stream().max(Comparator.comparing(v -> v.getFitness())).get().getGen().get(0);
//        return rett.get(0).getGen().get(0);
    }
```

Method generate()
Parameter count menjadi penentu sedalam apa kromosom dibangkitkan

```
    public List<Chromosome> generate(String[][] board, List<Chromosome> chromosomes,boolean isOpponentTurn,String botSymbol, String opponentSymbol, int count){
        List<int[]> emptySquares=getEmptySquares(board);
//        System.out.println(count);
        if (count<1){
            return chromosomes;
//            return null;
        }
        if (emptySquares.isEmpty()){
            return chromosomes;
//            return null;
        } else {
            String[][] newBoard=board;
            System.out.println(count);
//            for (int i=0;i<newBoard.length;i++){
//                for (int j=0;j<newBoard[0].length;j++){
//                    System.out.print(newBoard[i][j]+" ");
//                }
//                System.out.print("\n");
//            }
            if (isOpponentTurn){
                int[] bestMove=getBestOpponentMove(newBoard,botSymbol);
                newBoard[bestMove[0]][bestMove[1]]=opponentSymbol;
                return generate(newBoard,chromosomes,!isOpponentTurn,botSymbol,opponentSymbol,count-1);
            } else {
                List<Chromosome> newChromosomes=new ArrayList<Chromosome>();

                if (chromosomes.isEmpty()){
                    newChromosomes=generateChromosome(emptySquares);
                } else {
                    for (Chromosome chromosome :
                            chromosomes) {
                        for (int i=0;i<board.length;i++){
                            for (int j=0;j<board[i].length;j++){
//                                System.out.println(i+" "+j);
                                if (board[i][j].equals("")){
                                    Chromosome ne=new Chromosome();
                                    ne.copy(chromosome);
//                                    System.out.println(ne.toString());
                                    ne.addGen(new int[]{i,j});
//                                    System.out.println(ne.toString());
                                    if (!isInside(newChromosomes,ne)){
                                        newChromosomes.add(ne);
                                    }
                                }
                            }
                        }
                    }
                }
```

```
//                for (Chromosome c :
//                        newChromosomes) {
//                    System.out.println(c);
//                }

                List<List<Chromosome>> ret=new ArrayList<List<Chromosome>>();
                List<Chromosome> rett=new ArrayList<Chromosome>();
                newBoard=copyBoard(board);
//                for (Chromosome chromosome:
//                     newChromosomes) {
                    newBoard=copyBoard(board);
//
////                    System.out.println("aaaaa");
//                    int[] botMove=chromosome.getGen().get(chromosome.getGen().size()-1);
//                    newBoard=updateBoard(newBoard,botMove,botSymbol,opponentSymbol);
//
//                    chromosome.setFitness(fitness(newBoard,botSymbol,opponentSymbol));
//                    List<Chromosome> temp = generate(newBoard,newChromosomes,isOpponentTurn,botSymbol,opponentSymbol,count-1);
//                    for (Chromosome c :
//                            temp) {
//                        if (!isInside(rett, c)) {
//                            rett.add(c);
//                        }
//                    }
//                }
                List<Chromosome> temp = generate(newBoard,newChromosomes,isOpponentTurn,botSymbol,opponentSymbol,count-1);

//                rett.addAll(generate(newBoard,newChromosomes,isOpponentTurn,botSymbol,opponentSymbol,count-1));
                for (Chromosome c:
                     temp) {
                    newBoard=copyBoard(board);
                    int[] botMove=c.getGen().get(c.getGen().size()-1);
                    newBoard=updateBoard(newBoard,botMove,botSymbol,opponentSymbol);
                    c.setFitness(fitness(newBoard,botSymbol,opponentSymbol));
//                    System.out.println(c.toString());
                    if (!isInside(rett,c)){
                        rett.add(c);
                    }
                }
                System.out.println(Integer.toString(rett.size()));
                return rett;
            }
        }
    }
```

```
public List<Chromosome> generateChromosome(List<int[]> emptyBoard){
    List<Chromosome> ret=new ArrayList<Chromosome>();
    for (int[] coordinate :
            emptyBoard) {
        ret.add(new Chromosome().addGen(coordinate));
    }
    return ret;
}
```

Pada blok:

```
newBoard=copyBoard(board);
  for (Chromosome chromosome:
       newChromosomes) {
      newBoard=copyBoard(board);

        System.out.println("aaaaa");
      int[] botMove=chromosome.getGen().get(chromosome.getGen().size()-1);
      newBoard=updateBoard(newBoard,botMove,botSymbol,opponentSymbol);

      chromosome.setFitness(fitness(newBoard,botSymbol,opponentSymbol));
      List<Chromosome> temp = generate(newBoard,newChromosomes,isOpponentTurn,botSymbol,opponentSymbol,count-1);
      for (Chromosome c :
              temp) {
          if (!isInside(rett, c)) {
              rett.add(c);
          }
      }
  }
```

Akan dibangkitkan kromosom baru berdasarkan kromosom terbaru dan best action lawan

Method fitness()

```
    public float fitness(String[][] board,String botSymbol, String opponentSymbol){
        int bot=boardCount(board,botSymbol);
        int opponent=boardCount(board,opponentSymbol);
        return (float)bot/(float) opponent;
    }

    public List<Chromosome> fitnessFunction(List<Chromosome> chromosomes)
    {
        float sumVal= (float) chromosomes.stream().mapToDouble(o->o.getFitness()).sum();
        List<Chromosome> ret = new ArrayList<Chromosome>();
        ret.addAll(chromosomes);
        for (Chromosome c :
                ret) {
//            System.out.println("before");
//            System.out.println(c.toString());
            c.setFitness(c.getFitness()/sumVal);
//            System.out.println("after");
//            System.out.println(c.toString());
        }
        Collections.sort(ret, Comparator.comparing(Chromosome::getFitness));
        return chromosomes;
    }
```

```
    public List<Chromosome> newFitness(String[][] board, List<Chromosome> chromosomes,String botSymbol, String opponentSymbol){
        String[][] newBoard= copyBoard(board);
        List<Chromosome> newChromosomes = new ArrayList<Chromosome>();
        newChromosomes.addAll(chromosomes);
//        float sum=(float) chromosomes.stream().mapToDouble(o->o.getFitness()).sum();
//        System.out.println(sum);
        for (Chromosome c :
                newChromosomes) {
            newBoard=copyBoard(board);
            for (int[] coordinate:
            c.getGen()){
                newBoard=updateBoard(newBoard,coordinate,botSymbol,opponentSymbol);
                newBoard=updateBoard(newBoard,getBestOpponentMove(newBoard,botSymbol),opponentSymbol,botSymbol);
            }
            c.setFitness(fitness(board,botSymbol,opponentSymbol));
//            System.out.println("newf "+c.getFitness());
        }

        return newChromosomes;
    }
```

Method crossover()

```
    public List<Chromosome> crossover(Chromosome chromosome1, Chromosome chromosome2, int crossoverPoint){
        Chromosome newChromosome1 = new Chromosome();
        Chromosome newChromosome2 = new Chromosome();
        for (int i=0; i<crossoverPoint;i++){
            newChromosome1.addGen(chromosome1.getGen().get(i));
            newChromosome2.addGen(chromosome2.getGen().get(i));
        }
        for (int j=crossoverPoint; j<chromosome1.getGen().size();j++){
            newChromosome1.addGen(chromosome2.getGen().get(j));
            newChromosome2.addGen(chromosome1.getGen().get(j));
        }
        List<Chromosome> ret=new ArrayList<Chromosome>();
        ret.add(newChromosome1);
        ret.add(newChromosome2);
        return ret;
    }

    public List<Chromosome> newCrossover(String[][] board,List<Chromosome> chromosomes,int crossoverPoint,String botSymbol, String opponentSymbol){
        List<Chromosome> newChromosomes=new ArrayList<Chromosome>();
//        System.out.println("newc"+chromosomes.size());
//        for (Chromosome c :
//                chromosomes) {
//            System.out.println("aa "+ c.toString());
//        }
        for (int i=0;i<20;i++){
            Chromosome chromosome1 = select(chromosomes);
            Chromosome chromosome2 = select(chromosomes);
            newChromosomes.addAll(crossover(chromosome1,chromosome2,crossoverPoint));
        }
        newChromosomes=newFitness(board,newChromosomes,botSymbol,opponentSymbol);
        return fitnessFunction(newChromosomes);
    }

    public Chromosome select(List<Chromosome> chromosomes){
        Random random=new Random();
        float rfloat=random.nextFloat();
        float sum=0;
        for (Chromosome c :
                chromosomes) {
            sum += c.getFitness();
            if (sum>rfloat){
                return c;
            }
        }
        return chromosomes.get(chromosomes.size()-1);
    }
```

Method mutation()

```
}
public Chromosome mutation(Chromosome chromosome, int mutationIdx){
    Chromosome ret=new Chromosome();
    Random random=new Random();
    int newX=random.nextInt(8);
    int newY=random.nextInt(8);
    ret.copy(chromosome);
    ret.updateGen(mutationIdx,new int[]{newX,newY});
    return ret;
}
```

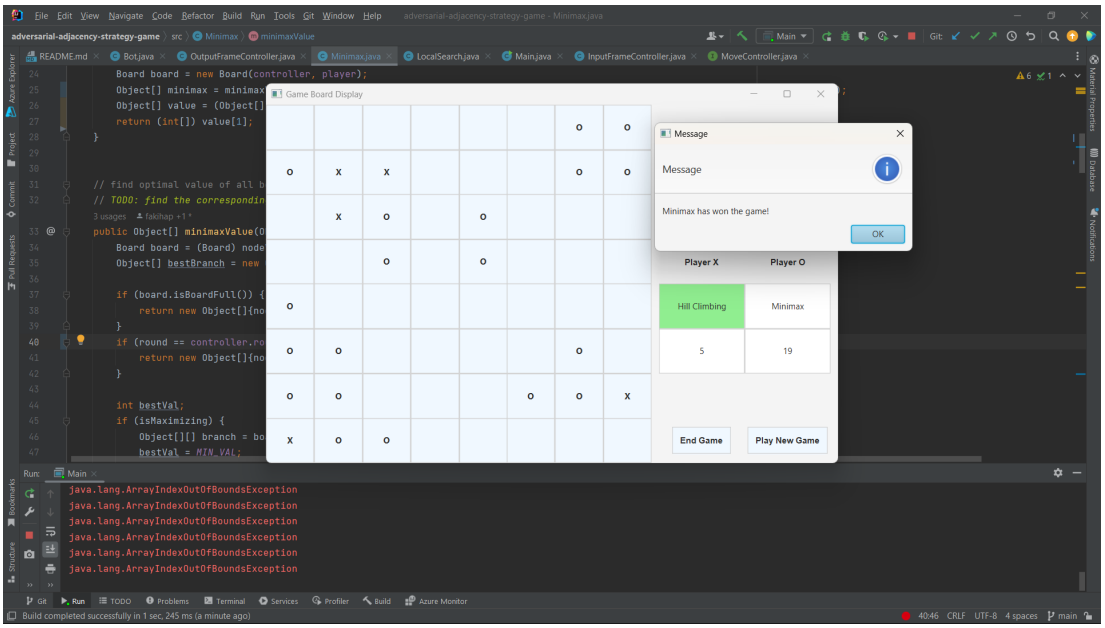
Class Chromosome

```
class Chromosome{
    private ArrayList<int[]> gen;
    private float fitness=0;

    public Chromosome(/*int[] coordinate*/){
        this.gen=new ArrayList<int[]>();
//            this.gen.add(coordinate);
    }

    public List<int[]> getGen(){
        return this.gen;
    }

    public float getFitness() {
        return fitness;
    }

    public Chromosome addGen(int[] coordinate){
        this.gen.add(coordinate);
        return this;
    }

    public void popGen(){
        this.gen.remove(this.gen.size()-1);
    }

    public void updateGen(int idx, int[] newGen){this.gen.set(idx,newGen);}

    public void setFitness(float fitness){ this.fitness=fitness;}

    @Override
    public String toString(){
        String ret= "";
        for (int[] g :
                this.gen) {
            ret=ret.concat("{").concat(Integer.toString(g[0])).concat(" ").concat(Integer.toString(g[1])).concat("}").concat(" ");
        }
        ret=ret.concat(Float.toString(this.fitness));
        return ret;
    }

    public boolean isEquals(Chromosome chromosome) {
        for (int[] g :
                this.gen) {
            for (int[] g2 :
                    chromosome.getGen()) {
                if (g[0] != g2[0] || g[1] != g2[1] || this.fitness != chromosome.fitness) {
                    return false;
                }
            }
        }
        return true;
    }

    public void copy(Chromosome chromosome){
        this.gen.addAll(chromosome.getGen());
        this.fitness=chromosome.fitness;
    }
}
```

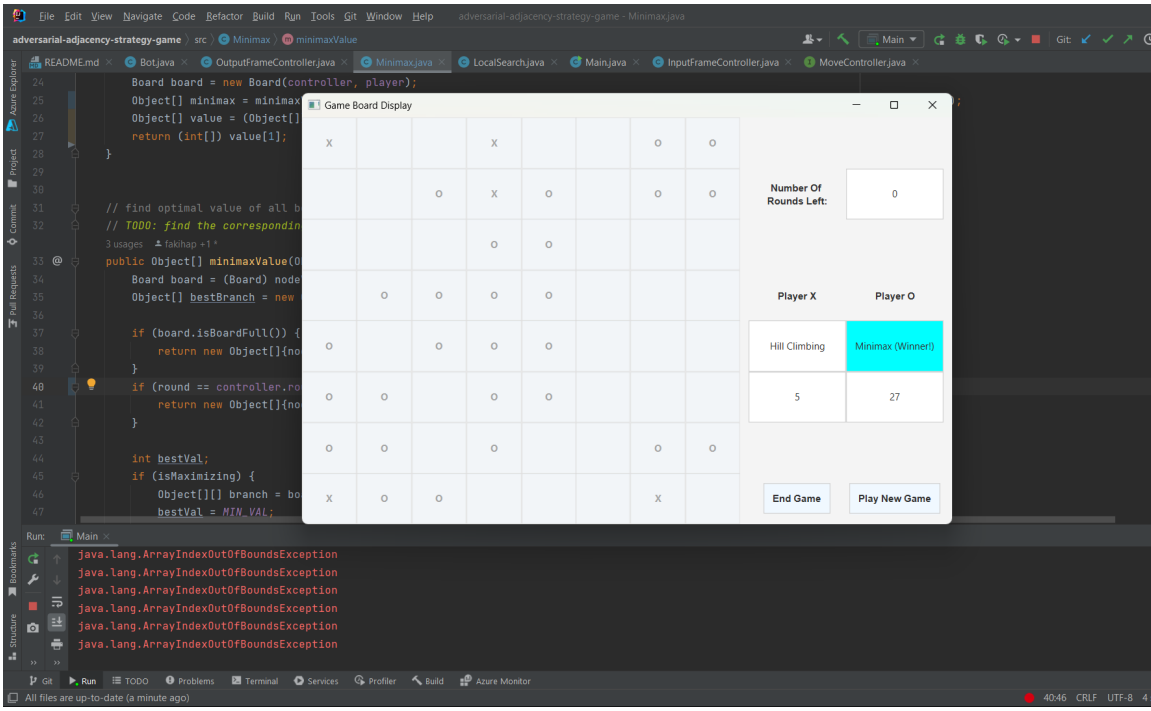
5. Hasil Pertandingan

5.1. Minimax Alpha Beta Pruning vs Local Search

5.1.1. 8 ronde

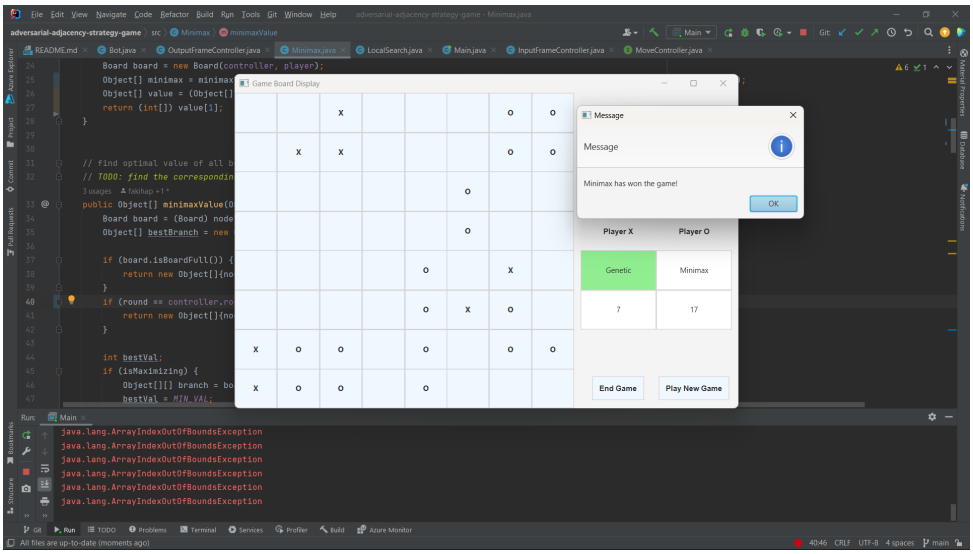


5.1.2. 12 ronde



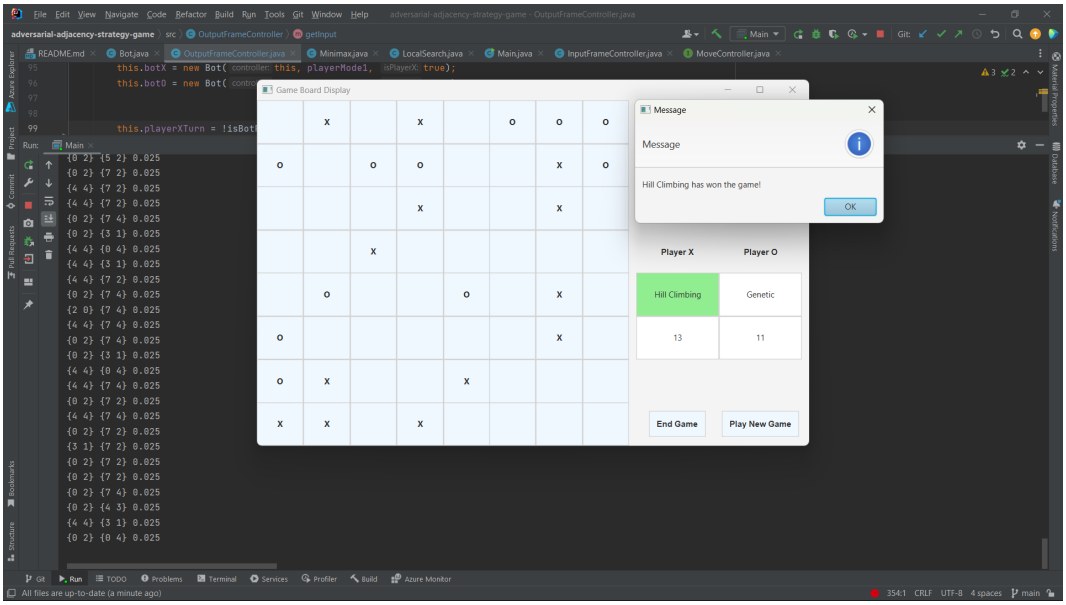
5.2. Minimax Alpha Beta Pruning vs Genetic Algorithm

5.2.1. 8 ronde

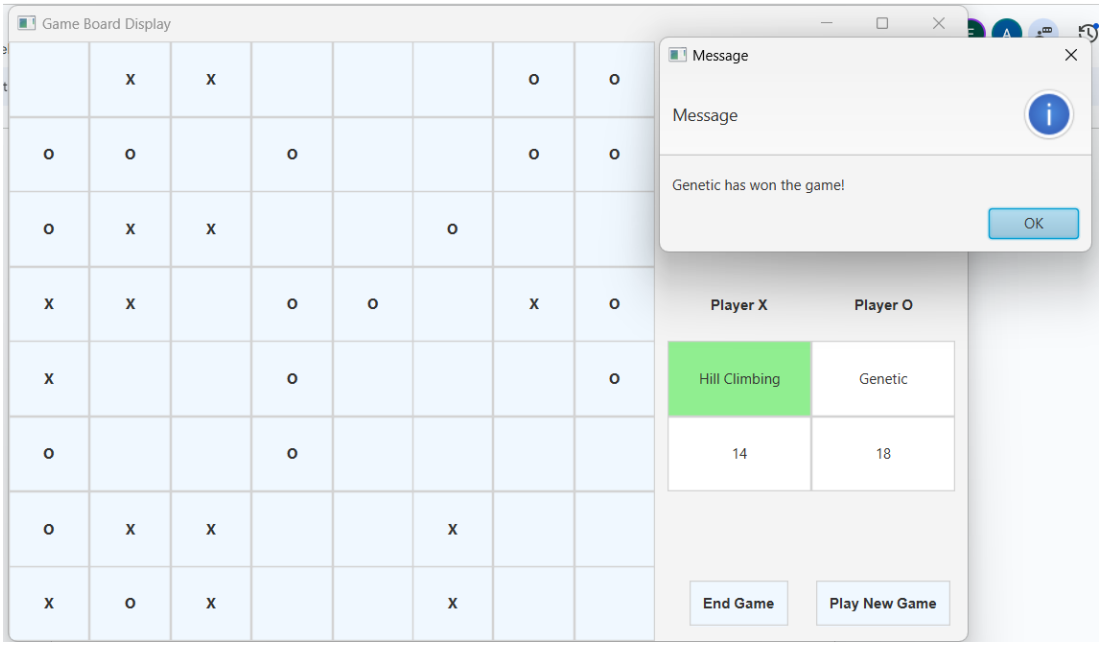


5.3. Genetic Algorithm vs Local Search

5.3.1. 8 Ronde



5.3.2. 12 Ronde



5.3.3. 26 Ronde

6. Pembagian Tugas

Nama	NIM	Pembagian Tugas
Fahrian Afdholi	13521031	*Minimax ab pruning algorithm*, Laporan
Fakih Anugerah P	13521091	*Minimax ab pruning algorithm*, Javafx: bot mode selection implementation, Laporan
Haidar Hamda	13521105	*Genetic algorithm,* Laporan
Ammar Rasyad C	13521136	*Local Search, Minimax,* JavaFX Asynchronous Bot Operation