IF3170 - Inteligensi Buatan

MiniMax Algorithm and Alpha Beta Pruning in Adjacency Strategy Game

Tugas Besar 1

Tanggal Pengumpulan: Rabu, 18 September 2023

Dibuat oleh

Febryan Arota Hia	13521120
William Nixon	13521123
Nicholas Liem	13521135
I Putu Bakta Hari Sudewa	13521150

Institut Teknologi Bandung
Sekolah Teknik Elektro dan Informatika
Tahun Ajaran 2023/2024

Daftar Isi

1 Pendahuluan

1.1 Penjelasan Singkat Tentang Adjacency Strategy Game

Pada tugas kecil kali ini, permainan yang akan digunakan adalah Adjacency Strategy Game. Secara singkat, Adjacency Strategy Game adalah suatu permainan dimana pemain perlu menempatkan marka (O atau X) pada papan permainan dengan tujuan memperoleh marka sebanyak mungkin pada akhir permainan (dengan jumlah ronde yang telah ditetapkan).
Aturan permainan Adjacency Strategy Game yang perlu diikuti adalah:

- Permainan dimainkan pada papan 8 x 8, dengan dua jenis pemain O dan X.
- Pada awal permainan, terdapat 4 X di pojok kiri bawah, dan 4 O di pojok kanan atas.
- Secara bergantian pemain X dan pemain O akan menaruh markanya di kotak kosong. Ketika sebuah kotak kosong diisi, seluruh kotak di sekitar yang sudah terisi marka musuh akan berubah menjadi marka pemain. Misal:

	O	
X		
X	X	O
X	X	X

  Lalu O mengisi board[1][1]

	O	
O	O	
X	O	O
X	X	X

  Perhatikan bahwa kotak pada arah diagonal tidak berubah ketika kotak kosong diisi O.
- Permainan selesai ketika papan penuh atau mencapai batas ronde yang telah ditetapkan.
- Pemenang adalah yang pemain yang memiliki marka terbanyak pada papan.

1.2 Board Evaluation Function

Fungsi evaluasi yang diberikan bertujuan untuk menilai kondisi saat ini dari papan permainan, khususnya dalam konteks permainan papan "Adjacency". Fungsi ini dimulai dengan menginisialisasi jumlah ke 0, yang akan digunakan untuk mengakumulasi skor evaluasi. Nilai X direpresentasikan dengan 1, dan O dengan -1. Kemudian, fungsi ini mengiterasi seluruh papan permainan, menjumlahkan nilai setiap sel.

Nilai-nilai tersebut dikalikan dengan 100 untuk memperbesar dampaknya pada evaluasi akhir. (Karena nilai setiap sel lah yang mempengaruhi kemenangan / kekalahan permainan, sehingga tidak boleh terpengaruhi nilai tambahan lain di bawah)

Selanjutnya, fungsi tersebut mempertimbangkan status penguncian setiap sel. Fungsi ini memeriksa sekitar setiap sel untuk menentukan apakah sel tersebut dikelilingi oleh potongan lain, membuatnya "terkunci". Evaluasi ini dilakukan untuk kedua pemain yang direpresentasikan oleh nilai -1 dan nilai 1.

Selanjutnya, fungsi mengevaluasi nilai setiap sel yang memiliki nilai 0 ("Kosong") dengan menjumlahkan nilai sel-sel tetangga yang bukan 0 (bidak yang bisa dimakan). Jika nilai posisi tersebut ¿= 2 atau ¡= -2, artinya terdapat 2 piece pemain yang berkemungkinan adalah posisi yang merugikan pemain X atau O. Oleh karena itu, nilai akan diadjust dengan banyaknya piece X atau O ini.

```
1   public static double evaluateBoard(int[][] board) {
2       int sum = 0;
3
4       for (int[] ints : board) {
5           for (int anInt : ints) {
6               sum += anInt;
7           }
8       }
9       sum *= 100;
10
11      int[] dx = {-1, 1, 0, 0};
12      int[] dy = {0, 0, -1, 1};
13
14      for (int i = 0; i < board.length; i++) {
15          for (int j = 0; j < board[i].length; j++) {
16              if (board[i][j] == 0) {
17                  int sums = 0;
18                  for (int k = 0; k < dx.length; k++) {
19                      int newRow = i + dx[k];
20                      int newCol = j + dy[k];
21
22                      if (newRow >= 0 && newRow < board.length &&
    newCol >= 0 && newCol < board[i].length) {
23                          if (board[newRow][newCol] != 0) {
24                              sums += board[newRow][newCol];
25                          }
26                      }
27                  }
28                  if (Math.abs(sums) >= 2) {
29                      sum += sums * -1;
30                  }
31              }
32          }
33      }
34
35
36      return sum;
37  }
```

1.3 Fungsi Heuristik

Pada fungsi heuristik ini terdapat beberapa fungsi yang digunakan untuk perhitungan *value* dari setiap kemungkinan langkah yang dapat diambil. Hal ini digunakan untuk memodifikasi algoritma Mini-max untuk mengekspan N simpul terbaik saja. Fungsi-fungsi tersebut adalah *countStateValue*, *countOpponentChance*, dan *countPriority*.

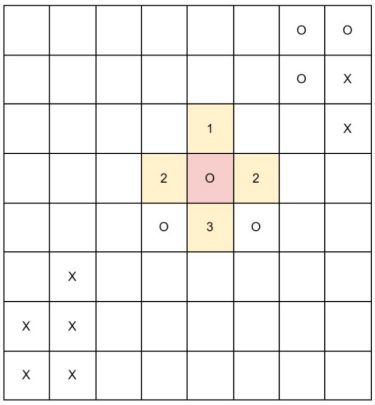
```
1
2       public static int countStateValue(int [][] board, int row, int
        col, boolean isOpponent) {
3           int result = 0;
4           int mark = isOpponent ? -1 : 1;
5
6           int[] dx = {-1, 1, 0, 0};
7           int[] dy = {0, 0, -1, 1};
8
9           for (int k = 0; k < 4; k++) {
10              int newRow = row + dx[k];
11              int newCol = col + dy[k];
12
13              if (newRow >= 0 && newRow < board.length && newCol >= 0
        && newCol < board[0].length) {
14                  if (board[newRow][newCol] == mark) {
15                      result++;
16                  }
17              }
18          }
19          return result;
20      }
21
22      public static int countOpponentChance(int[][] board, int row,
        int col, boolean isOpponent) {
23          int chance = 0;
24
25          int[] dx = {-1, 1, 0, 0};
26          int[] dy = {0, 0, -1, 1};
27
28          for (int k = 0; k < 4; k++) {
29              int newRow = row + dx[k];
30              int newCol = col + dy[k];
31
32              if (newRow >= 0 && newRow < board.length && newCol >= 0
        && newCol < board[0].length) {
33                  if (board[newRow][newCol] == 0) {
34                      int temp = countStateValue(board, newRow,
        newCol, !isOpponent);
35                      chance = Math.max(temp, chance);
36                  }
```

```
37              }
38          }
39          return chance;
40      }
41
42      public static int countPriority(int[][] board, int row, int col
    ) {
43          int priority = 0;
44
45          int[] dx = {-1, 1, 0, 0};
46          int[] dy = {0, 0, -1, 1};
47
48          for (int k = 0; k < 4; k++) {
49              int newRow = row + dx[k];
50              int newCol = col + dy[k];
51
52              if (newRow >= 0 && newRow < board.length && newCol >= 0
     && newCol < board[0].length) {
53                  if (isPriority(board, newRow, newCol)) {
54                      priority++;
55                  }
56              }
57          }
58          return priority;
59      }
60
61      public static boolean isPriority(int[][] board, int row, int
    col) {
62          int[] dx = {-1, 1, 0, 0};
63          int[] dy = {0, 0, -1, 1};
64
65          for (int k = 0; k < 4; k++) {
66              int newRow = row + dx[k];
67              int newCol = col + dy[k];
68
69              if (newRow >= 0 && newRow < board.length && newCol >= 0
     && newCol < board[0].length) {
70                  if (board[newRow][newCol] == 0) {
71                      return false;
72                  }
73              }
74          }
75          return true;
76      }
77
78      public static int calculateHeuristic(int[][] board, int row,
    int col, boolean isOpponent) {
79          int stateValue = countStateValue(board, row, col,
    isOpponent);
80          int opponentChance = countOpponentChance(board, row, col,
    isOpponent);
81          int priority = countPriority(board, row, col);
82          return stateValue - opponentChance + priority;
83      }
```

Penjelasan mengenai ketiga fungsi di atas adalah sebagai berikut:

1. *countstateValue* digunakan untuk menghitung berapa banyak marka baru yang dapat diperoleh pada suatu state (tidak termasuk marka saat ini). Tujuan function ini adalah untuk mengetahui berapa banyak marka yang dapat diambil alih. Semakin banyak marka yang dapat diambil alih, maka agen akan condong untuk mengambil langkah tersebut. Range yang dihasilkan oleh fungsi ini berkisar 1-4.

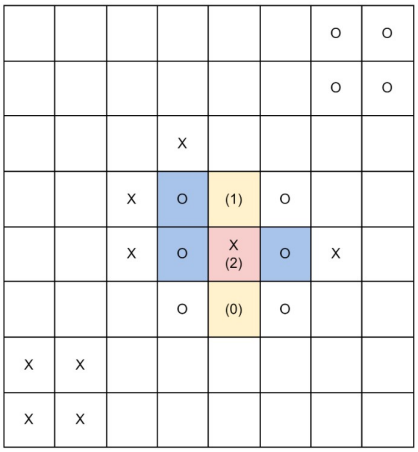
2. *countOpponentChance* digunakan untuk menghitung nilai terbesar yang mungkin didapat oleh lawan jika ia mengambil posisi di sekitar posisi yang akan agen ambil saat ini. Lebih spesifik akan dijelaskan dengan menggunakan berikut



   Dari gambar di atas, kotak merah merupakan marka yang baru ditambahkan dan kotak oranye merupakan nilai yang mungkin diperoleh oleh lawan. Nilai pada kotak oranye menunjukkan berapa banyak marka yang dapat diperoleh lawan jika meletakkan markanya pada posisi yang bersesuaian. Untuk state tersebut, maka nilai yang diperoleh adalah MAX(1, 2, 2, 3) = 3.Range nilai yang dihasilkan oleh fungsi ini berkisar 1-4.

3. *countPriority* digunakan untuk menghitung berapa banyak prioritas marka yang bisa didapat pada suatu posisi. Nilai prioritas dihitung dengan menggunakan rumus sebagai berikut,

   prioritas(s) = banyaknya marka yang dapat diambil alih, dimana marka tersebut ketiga sisinya tidak kosong



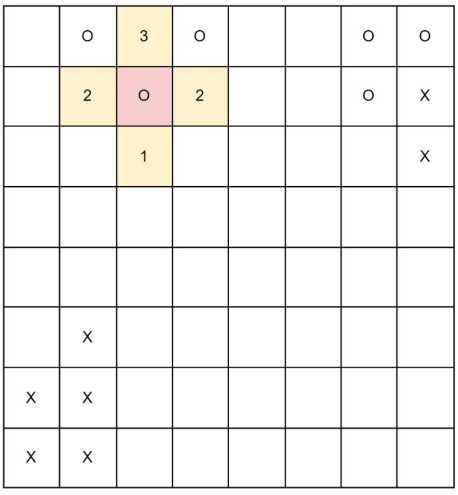
Dari gambar di atas, kotak yang berwarna biru menandakan marka yang dapat diambil alih. Terlihat ada 3 marka yang dapat diambil alih dengan ketiga sisi dari marka tersebut tidak kosong. Nilai prioritas dari state tersebut adalah 2. Range nilai yang dihasilkan oleh fungsi ini berkisar 1-4.

1.4 Penentuan Nilai Heuristik Move

Mekanisme yang digunakan adalah memilih hasil perhitungan heuristik tertinggi untuk mengambil gerakan yang paling mungkin menguntungkan current player di dalam ekspansi Mini-Max.

Heuristik Total = 1.countStateValue(s) - 1.countOponentChance(s) + 1.countPriority(s)

Dengan s adalah state saat ini. Gambar di bawah mengilustrasikan salah satu state yang akan dihitung dengan menggunakan heuristik function ini.



Kotak yang berwarna merah berisi marka yang baru saja ditambahkan. Kotak yang berwarna oranye merupakan kemungkinan nilai dari fungsi *countOponentChance*. Untuk state tersebut, akan menghasilkan nilai sebagai berikut,

Heuristik =0 - MAX(1, 2, 2, 3) + 0 = 0 - 3 = -3

Terlihat nilai *objective* adalah -3 yang menunjukkan state ini merupakan state yang menguntungkan marka lawan (marka X) – diasumsikan marka O merupakan marka yang diinginkan untuk menjadi pemenang permainan, sehingga mendapatkan bobot yang rendah dalam pemilihan gerakan.

2 Implementasi Minimax Alpha Beta Pruning

2.1 Penjelasan Implementasi Algoritma Minimax

Algoritma Minimax adalah algoritma *adversarial search* yang digunakan untuk menemukan solusi optimal dari suatu permasalahan. Algoritma ini pada dasarnya adalah algoritma yang memanfaatkan *backtracking* dan digunakan untuk permainan yang bersifat *turn-based* misalnya seperti Tic-Tac-Toe, Backgammon, Mancala, Catur, dan lain sebagainya.

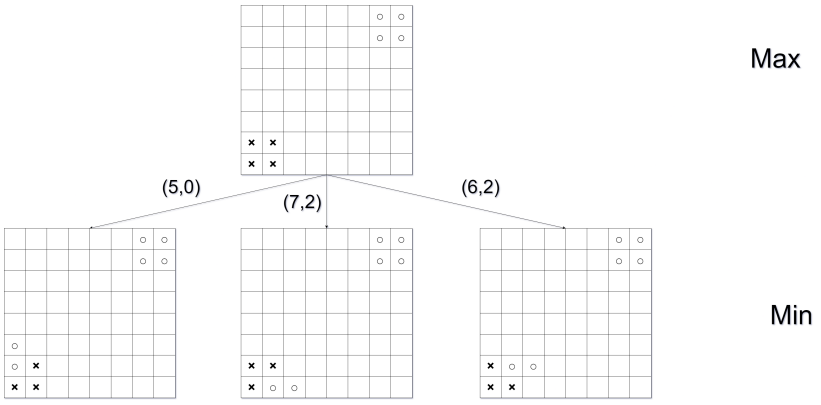
Dalam Algoritma ini, ada dua tokoh utama yang akan menjadi pembeda yakni *maximizer* dan *minimizer*, setiap tokoh ini memiliki perannya masing-masing. Untuk *maximizer* tentunya mencari nilai maksimal pada kondisi atau *state* pada saat ini. Sedangkan dengan *minimizer*, tokoh ini perannya adalah untuk mencari nilai minimum pada kondisi atau *state* saat ini. Penggunaan tokoh ini dibuat selang-seling dalam struktur data pohon.

Pemetaan sederhana permainan *adjacency strategy game* pada algoritma Minimax yang dikembangkan adalah sebagai berikut.

1. Untuk simpul akar, akan di set nilainya sebagai *maximizer* dan simpul ini menggambarkan *state* awal papan di mana hanya ada 4 buah simbol bot dan 4 buah simbol user.

2. Setiap aksi atau percabangan dari setiap simpul adalah bentuk titik-titik yang dibangkitkan melalui fungsi *getPossibleMoves* dan dibatasi secukupnya. Gunakan juga teknik *alpha-beta pruning* untuk mengurangi jumlah komputasi algoritma.

3. Simpul setelah diberikan sebuah aksi adalah simpul dengan *state* baru setelah titik tersebut dipilih pada papan.



2.2 Struktur Kode Implementasi Algoritma Minimax

Berikut adalah bentuk kode implementasi algoritma Minimax yang digunakan pada tugas ini.

```
1  public class MinimaxAgent {
2
3      public int[] move(int[][] board, boolean maximizingPlayer, int
       roundsLeft) {
4          Utils.printBoard(board);
5          TreeNode<int[]> root = new TreeNode<>(null, false);
6          TreeNode<int[]> bestMove = calculate(root, board, Double.
       NEGATIVE_INFINITY, Double.POSITIVE_INFINITY, 0, 8,
       maximizingPlayer, roundsLeft);
7          return bestMove.getData();
8      }
9
10     public TreeNode<int[]> calculate(TreeNode<int[]> node, int[][]
       board, double alpha, double beta, int depth, int maxDepth,
       boolean isMaximizingPlayer, int roundsLeft) {
11         // Return if terminal node
12         if (roundsLeft == 0 || depth == maxDepth || Utils.
       isTerminal(board)) {
13             double score = Utils.evaluateBoard(board);
14             node.setBoard(board);
15             node.setScore(score);
16             return node;
17         }
18
19         List<int[]> possibleMoves = Utils.getPossibleMoves(board,
       isMaximizingPlayer, 10);
20         // Init best move and best score
21
22         double bestScore;
23         TreeNode<int[]> bestMove = null;
24         if (isMaximizingPlayer) {
25             bestScore = Double.NEGATIVE_INFINITY;
26
27             for (int[] move : possibleMoves) {
28
29                 int[][] newState = Utils.transition(Utils.copyBoard
       (board), move, true);
30                 TreeNode<int[]> childNode = new TreeNode<>(move,
       false);
31                 node.addChild(childNode);
32
33                 TreeNode<int[]> result = calculate(childNode,
       newState, alpha, beta, depth + 1, maxDepth, false, roundsLeft -
       1);
34                 double score = result.getScore();
35
36                 if (score > bestScore) {
37                     bestScore = score;
38                     bestMove = childNode;
39                 }
40
```

```
41                  alpha = Math.max(alpha, bestScore);
42                  if (beta <= alpha) {
43                      break;
44                  }
45              }
46
47          } else {
48              bestScore = Double.POSITIVE_INFINITY;
49
50              for (int[] move : possibleMoves) {
51
52                  int[][] newState = Utils.transition(Utils.copyBoard
    (board), move, false);
53                  TreeNode<int[]> childNode = new TreeNode<>(move,
    true);
54                  node.addChild(childNode);
55
56                  TreeNode<int[]> result = calculate(childNode,
    newState, alpha, beta, depth + 1, maxDepth, true, roundsLeft -1)
    ;
57                  double score = result.getScore();
58
59                  if (score < bestScore) {
60                      bestScore = score;
61                      bestMove = childNode;
62                  }
63
64                  beta = Math.min(beta, bestScore);
65                  if (beta <= alpha) {
66                      break;
67                  }
68              }
69
70          }
71          node.setBoard(board);
72          node.setScore(bestScore);
73          return bestMove;
74      }
75
76  }
```

1. Fungsi *move* adalah fungsi yang digunakan untuk mengembalikan langkah terbaik yang dipilih oleh algoritma.

2. Fungsi *calculate* adalah fungsi rekursif yang digunakan untuk perh/itungan dalam algoritma Minimax, dalam hal ini memiliki beberapa parameter, yakni TreeNode node, int[][] board, double alpha, double beta, int depth, int maxDepth, boolean isMaximizingPlayer, int roundsLeft.

3 Implementasi Local Search

3.1 Penjelasan Singkat Local Search Algorithm

Algoritma *Local Search* yang digunakan pada penyelesaian permainan adalah algoritma *hill climbing with sideways move*. Berikut merupakan beberapa aspek dari *local search* yang digunakan:

- Initial State: State awal dari pencarian adalah kondisi board pada awal permainan.
- Neighbor: Semua kemungkinan move yang dapat diambil dari state permainan sekarang.
- Action: Memilih move ke neighbor dengan nilai objective tertinggi
- Solution: Move yang dilakukan berikutnya.

Berikut merupakan langkah-langkah dalam proses pencarian *local search*.

1. State awal berupa keadaan *board* sekarang.
2. Agen akan membuat semua *neghbor* berupa semua langkah yang dapat diambil pada saat itu.
3. Setiap neighbor akan dievaluasi nilainya berdasarkan fungsi objective
4. Berdasarkan hasil evaluasi akan dipilih *neighbor* dengan nilai tertinggi

Pada penyelesaian permainan, algoritma *hill climbing with sideways move* dipilih berdasarkan pemilihan *successor*-nya. Pada algoritma ini, *successor* yang dipilih adalah neighbor dengan objective tertinggi di antara seluruh neighbor. Sedangkan pada algoritma seperti *simulated annealing* atau *stochastic hill-climbing*, *successor* dibangkitkan dengan cara random. Hal tersebut cukup berdampak pada kemenangan karena permainan memiliki ronde yang terbatas, sebab dari itu sebisa mungkin setiap pengambilan langkah harus mengambil yang paling menguntungkan. Lalu jika dibandingkan dengan *hill climbing steepest ascent*, algoritma *sideways move* lebih menguntungkan karena *steepest ascent* memungkinkan pencarian *stuck* pada *shoulder*.

3.2 Struktur Kode Implementasi Algoritma Local Search

```
1
2 public int[] move (int[][] board, boolean isX) {
3     List<int[]> possibleMoves = Utils.getPossibleMoves(board, isX,
      64);
4     double maxScore = isX ? Double.MIN_VALUE : Double.MAX_VALUE;
5     int[] bestMove = new int[2];
6
7     long startTime = System.currentTimeMillis();
```

```
8       long endTime = startTime + 5000; //Set timeout
9
10      for (int[] move : possibleMoves) {
11          int[][] newState = Utils.transition(Utils.copyBoard(board),
        move, isX);
12
13          double score = Utils.evaluateBoard(newState);
14          if ((!isX && score < maxScore) || (isX && score > maxScore)
       ) {
15              maxScore = score;
16              bestMove[0] = move[0];
17              bestMove[1] = move[1];
18          }
19
20          long currentTime = System.currentTimeMillis();
21          if (currentTime >= endTime) {
22              break;
23          }
24      }
25      return new int[]{bestMove[0], bestMove[1]};
26  }
```

4 Implementasi Genetic Algorithm

4.1 Penjelasan Singkat Genetic Algorithm

4.1.1 Definisi umum algoritma genetik

Genetic Algorithm secara sederhana terdiri dari beberapa langkah utama:

1. Bentuk sebanyak k jumlah state random.
2. Gunakan *fitness function* untuk menentukan *state value*.
3. Lakukan proses seleksi pada *parent state* berdasarkan *fitness function*.
4. Lakukan proses *cross-over* di mana titik *cross-over*nya random.
5. Lakukan mutasi terhadap suksesor yang dibentuk.
6. Lakukan langkah 2-5 berulang kali hingga mendapatkan hasil yang sesuai dengan *termination condition*,

4.1.2 Algoritma *genetic minimax search*

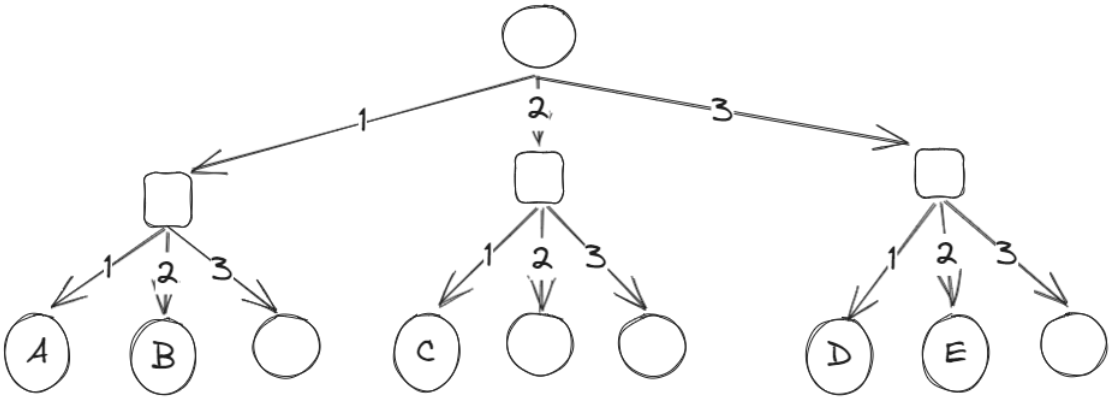
Berdasarkan usulan yang diberikan dari Tzung-Pei Hong dkk, untuk permasalahan permainan yang berbasis giliran (dua pemain) evaluasi minimax adalah suatu hal yang wajib. Untuk mengimplementasi algoritma genetik dalam permasalahan ini dapat dilakukan dengan menggabungkan teknik minimax dengan algoritma genetik sehingga menghasilkan teknik baru yakni *genetic minimax search* [HHL01].

Perbedaannya dengan pendekatan lain yang disebutkan di atas, algoritma genetik akan menggunakan satuan generasi untuk melakukan proses evolusinya. Artinya, setiap permainan yang dijalankan merupakan satu generasi. Setiap generasi yang dilakukan, akan dilakukan evaluasi yang sesuai untuk menentukan suksesor yang lebih baik.

Dalam menyusun algoritma genetik, ada beberapa hal yang penting untuk ditentukan. Beberapa di antaranya adalah bagaimana algoritma mendefinisikan sebuah kromosom, bagaimana dan apa fungsi *fitness* yang dirancang, dan bagaimana proses *cross-over* dilakukan.

4.1.3 Pengkodean kromosom

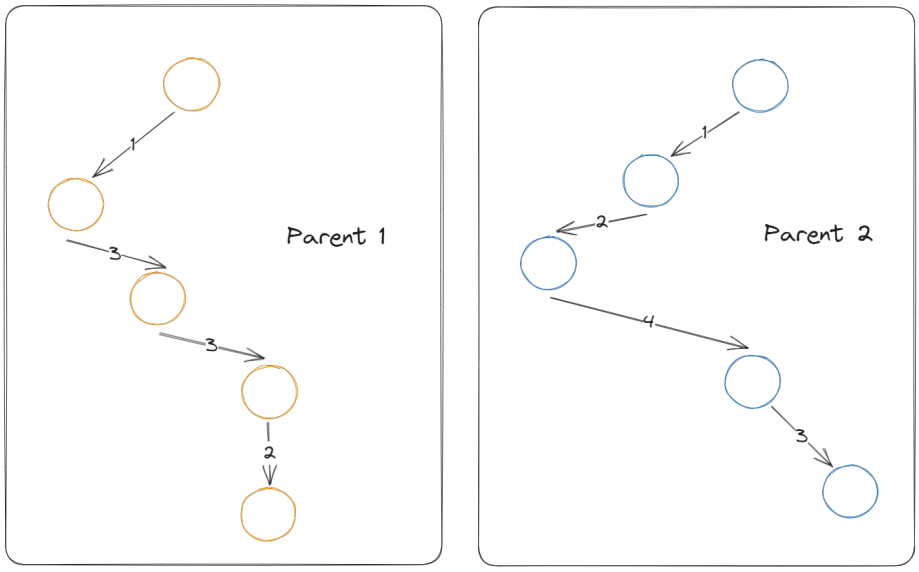
Proses pengkodean kromosom yang diproposisikan adalah sekuens dari kemungkinan langkah yang mungkin dilakukan pada permainan *adjacency*. Berikut adalah contoh representasinya dalam bentuk gambar.

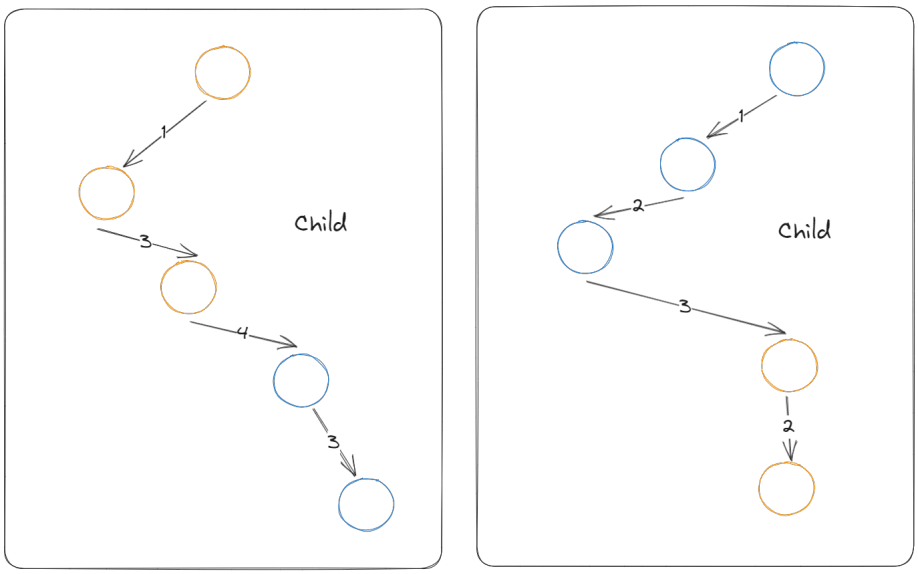


Nilai A dapat dikodekan menjadi 11, B menjadi 12, C menjadi 21, D menjadi 31, E menjadi 32. Setiap representasi huruf kapital di sini nantinya akan menjadi state pada permainan. `Sedangkan, angka pada graf pohon di atas dapat diganti menjadi koordinat di mana` ***`mark`*** `ditempatkan pada papan permainan`.

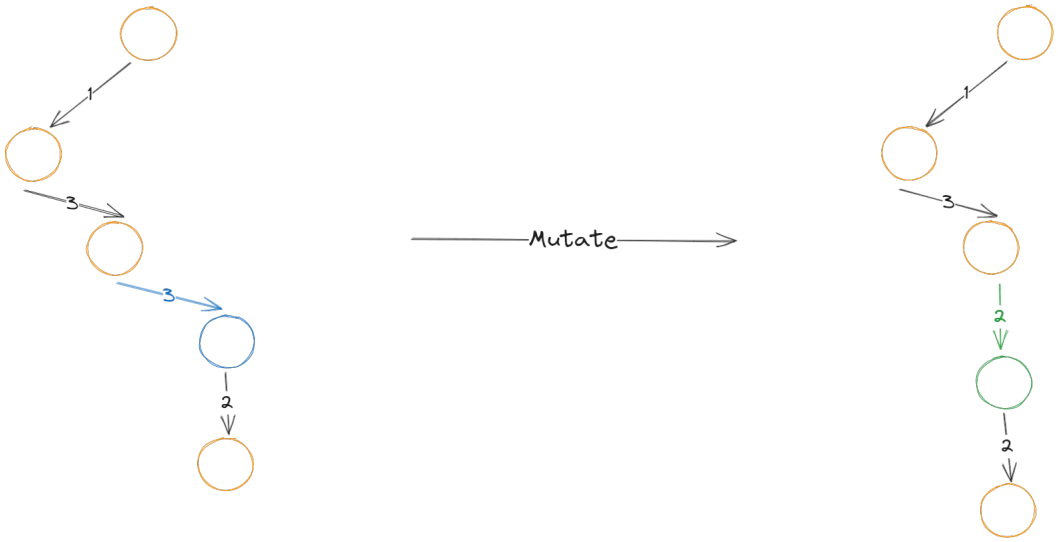
4.1.4 Metode *crossover* dan mutasi

Implementasi *crossover* pada permainan berbasis giliran untuk dua orang dapat dilakukan dengan mengganti urutan pohon. Contohnya dapat dilihat pada gambar di bawah ini.



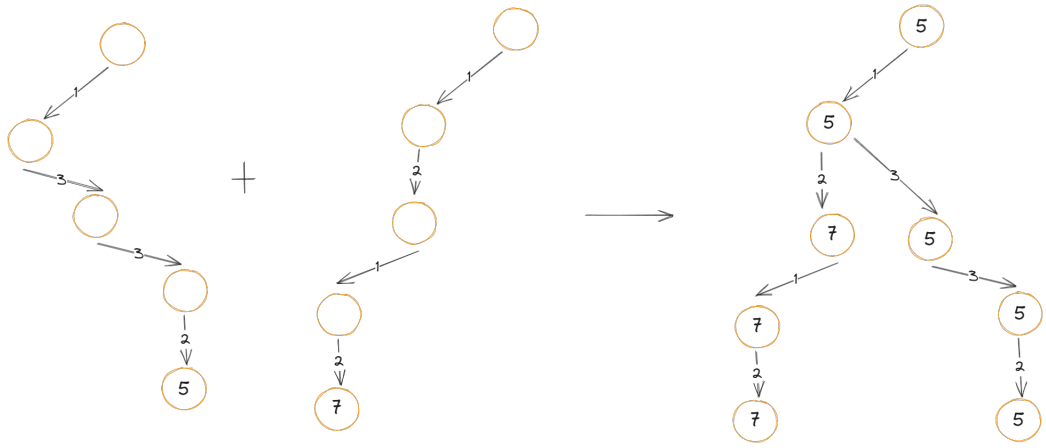


Untuk permasalahan mutasi, mutasi hanya mengubah suatu simpul yang misalnya sebelumnya kita menggunakan aksi α berubah menjadi aksi β. Ilustrasinya dapat dilihat pada gambar di bawah ini.



4.1.5 Fungsi *fitness* dengan metode pohon reservasi

Sifat permainan ini adalah berasis giliran dan dua orang, maka algoritma harus menggunakan prinsip dari minimax sebagai salah satu faktor dalam penentuan fungsi *fitness* supaya pembandingan kromosom dapat dilakukan dengan akurasi yang tinggi, artinya juga *fitness* dari kromosom ke kromosom suksesor harus tetap terjaga. Salah satu proposal Hong adalah dengan penggunaan pohon reservasi. Ide utamanya adalah untuk melihat state mana yang masih dipertahankan dari daun atau simpul terminal hingga tingkatan di atasnya, atau seberapa dalam propagasi suatu nilai simpul terminal dan semakin dalam maka nilai *fitness*nya semakin tinggi. Perhatikan contoh berikut.



Dapat dilihat bahwa ranting di sebelah kanan dengan urutan 1332 dipilih sebagai kromosom yang tetap sebab nilai evaluasi (5) memiliki jarak propagasi terjauh dibandingkan dengan nilai evaluasi (7). Kesimpulannya adalah pendefinisian fungsi *fitness* adalah tinggi level dari bawah di mana sebuah simpul daun atau simpul terminal dapat mempertahankan nilai evaluasi (dalam contoh di atas adalah angka 5 dengan ketinggian 5 juga). Hong mendefinisikan bahwa H adalah tinggi dari pohon reservasi dan level maksimal yang dapat dicapai sebuah simpul terminal adalah K, maka fungsi *fitness*nya adalah sebagai berikut:

$$Fitness(Kromosom) = H - K + 1$$

4.1.6 Struktur Kode Implementasi Algoritma Local Search

Berikut merupakan cuplikan beberapa kelas berikut fungsi-fungsi penting di dalam implementasi algoritma genetik. Untuk kode lengkapnya, silahkan mengacu pada repository github kami.

```
1
2  public class GeneticAgent {
3
4      int CHROMOSOME_SIZE = 4;
5
6      public int[] move(int[][] board, boolean maximizingPlayer, int
       roundsLeft) {
7          Population.isMaximizing = maximizingPlayer;
8          Individual.maximizingPlayer = maximizingPlayer;
9
10         Population population = new Population(500, board);
11         Population.board = board;
12         population.makeReservationTree();
13
14         int generations = 100;
15         Individual best = population.getBestIndividual();
16
17         for (int generation = 0; generation < generations;
       generation++) {
18             Population newPopulation = population.evolve();
19             population = newPopulation;
20             population.makeReservationTree();
```

```
21              Individual bestGeneration = population.
    getBestIndividual();
22              if (best.fitness < bestGeneration.fitness) {
23                  best = bestGeneration;
24              }
25          }
26
27          return best.chromosome.get(0);
28      }
29  }
30
31   class Individual {
32          static int id_count = 0;
33          public int id;
34          private List<int[]> chromosome;
35          private double fitness;
36
37          private int[][] board;
38          public static boolean maximizingPlayer;
39
40
41          public Individual(List<int[]> chromosome, int[][] board) {
42              this.chromosome = chromosome;
43              int[][] currentBoard = board;
44              boolean isMaximizingPlayer = false;
45
46              for (int i = 0; i < chromosome.size(); i++) {
47                  int[] selectedMove = chromosome.get(i);
48                  currentBoard = Utils.transition(Utils.copyBoard(
    currentBoard), selectedMove, isMaximizingPlayer);
49
50                  this.board = currentBoard;
51              }
52
53              this.fitness = Utils.evaluateBoard(this.board);
54
55              this.chromosome = chromosome;
56              this.id = id_count;
57              id_count += 1;
58          }
59
60          public Individual(int[][] initialBoard) {
61              List<int[]> chromosome = new ArrayList<>();
62              int[][] currentBoard = initialBoard;
63              boolean isMaximizingPlayer = maximizingPlayer; // Start
     with minimizing player
64
65              // Generate a sequence of moves
66              for (int moveNumber = 0; moveNumber < CHROMOSOME_SIZE;
    moveNumber++) {
67                  List<int[]> possibleMoves = Utils.getPossibleMoves(
    currentBoard, isMaximizingPlayer, 5);
68
69                  if (possibleMoves.isEmpty()) {
70                      break;
71                  }
```

```
72
73                  int randomMoveIndex = new Random().nextInt(
    possibleMoves.size());
74                  int[] selectedMove = possibleMoves.get(
    randomMoveIndex);
75
76                  chromosome.add(selectedMove);
77
78                  currentBoard = Utils.transition(Utils.copyBoard(
    currentBoard), selectedMove, isMaximizingPlayer);
79
80                  this.board = currentBoard;
81
82                  isMaximizingPlayer = !isMaximizingPlayer;
83              }
84
85              this.fitness = Utils.evaluateBoard(this.board);
86
87              this.chromosome = chromosome;
88              this.id = id_count;
89              id_count += 1;
90          }
91
92
93          public Individual crossover(Individual parent2) {
94              int crossoverPoint = new Random().nextInt(this.
    getChromosome().size());
95
96              List<int[]> childChromosome = new ArrayList<>();
97              childChromosome.addAll(this.getChromosome().subList(0,
    crossoverPoint));
98              childChromosome.addAll(parent2.getChromosome().subList(
    crossoverPoint, parent2.getChromosome().size()));
99
100             return new Individual(childChromosome, board);
101         }
102
103         // Mutation function
104         public void mutate(double mutationRate, int[][] currBoard)
    {
105             if (Math.random() < mutationRate) {
106                 int randomPoint = new Random().nextInt(this.
    getChromosome().size());
107                 List<int[]> possibleMoves = Utils.getPossibleMoves(
    currBoard, (randomPoint % 2 == 1), 1);
108                 int[] mutateMove = possibleMoves.get(0);
109                 chromosome.set(randomPoint, mutateMove);
110                 }
111             }
112         }
113     }
114 }
115
116 class Population {
117
118     private ReservationTree tree;
```

```
119     private List<Individual>individuals;
120     double mutationRate = 0.01;
121     public static boolean;
122     public static int[][] board;
123
124
125     public Population(int size, int[][] initialBoard) {
126         individuals = new   ArrayList<>();
127         this.board = initialBoard;
128         for (int i = 0; i < size;   i++) {
129             Individual individual   = new  Individual(initialBoar
     );
130             individuals.add(indivi  ual);
131         }
132     }
133
134     public void makeReservationTree() {
135         ReservationTree.isMaximizing = this.isMaximizing;
136         tree = new ReservationTree();
137         for (Individual individual : individuals) {
138         tree.attachIndividual(individual);
139             }
140             tree.propagate();
141         }
142
143     public Population evolve() {
144         Population newPopulation = new Population();
145         while (newPopulation.size() < individuals.size()) {
146                 Individual parent1 = selectParent();
147                 Individual parent2 = selectParent();
148                 Individual child = parent1.crossover(parent2);
149                 child.mutate(mutationRate, this.board);
150                 if (!child.isValid(board)) {
151                     continue;
152                 }
153                 newPopulation.addIndividual(child);
154             }
155             return newPopulation;
156         }
157
158         public Individual selectParent() {
159             Individual selectedParent = null;
160             double totalFitness = individuals.stream().mapToDouble(
    i -> tree.getFitnessValue(i)).sum();
161             double randomValue = Math.random() * totalFitness;
162
163             for (Individual individual : individuals) {
164                 randomValue -= tree.getFitnessValue(individual);
165                 if (randomValue <= 0) {
166                     selectedParent = individual;
167                     break;
168                 }
169             }
170
171             return selectedParent;
172         }
```

```
173
174         public void printFitness() {
175             tree.print(tree.root, 0);
176         }
177     }
178 }
179
180 public class ReservationTree {
181         private Node root;
182
183         public HashMap<Integer, Integer> fitness;
184
185         static public boolean isMaximizing = false;
186
187
188         public void attachIndividual(Individual individual) {
189             fitness.put(individual.id, 1);
190             Node currentNode = root;
191             int level = 0;
192             double leafValue = individual.getFitness(); // Use
    fitness as the leaf value
193
194             // Traverse the tree based on the individual's
    chromosome
195             for (int[] move : individual.getChromosome()) {
196                 Node child = currentNode.getChild(move);
197
198                 if (child == null) {
199                     child = new Node(move, level + 1, Double.
    NEGATIVE_INFINITY);
200                     currentNode.addChild(child);
201                 }
202
203                 currentNode = child;
204                 level++;
205             }
206             currentNode.setLeafValue((int) leafValue);
207             currentNode.addId(individual.id);
208         }
209
210         public void propagate() {
211             calculateFitnessValues(this.root);
212         }
213
214         private double calculateFitnessValues(Node node) {
215             if (node.isLeaf()) {
216                 return node.getLeafValue();
217             } else {
218                 double maxChildValue = Double.NEGATIVE_INFINITY;
219                 List<Integer> maxIds = new ArrayList<>();
220                 double minChildValue = Double.POSITIVE_INFINITY;
221                 List<Integer> minIds = new ArrayList<>();
222
223                 for (Node child : node.getChildren()) {
224                     double childValue = calculateFitnessValues(
    child);
```

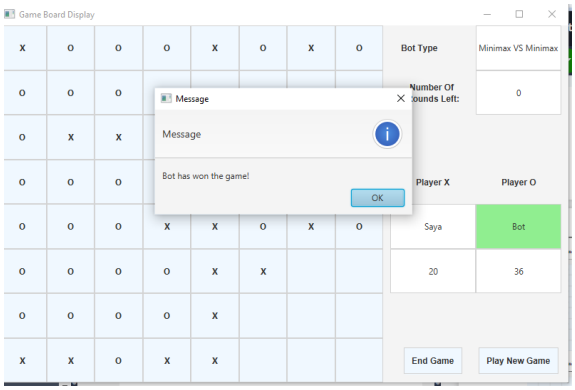
```
225                     List<Integer> childrenIds = child.getIds();
226
227                     if (childValue > maxChildValue) {
228                         maxChildValue = childValue;
229                         maxIds.clear();
230                         maxIds.addAll(childrenIds);
231                     } else if (childValue == maxChildValue) {
232                         maxIds.addAll(childrenIds);
233                     }
234
235                     if (childValue < minChildValue) {
236                         minChildValue = childValue;
237                         minIds.clear();
238                         minIds.addAll(childrenIds);
239                     } else if (childValue == minChildValue) {
240                         minIds.addAll(childrenIds);
241                     }
242                 }
243
244                 if (node.isMaxLevel()) {
245                     node.setNodeValue(maxChildValue);
246                     node.setId(maxIds);
247                     for (Integer id : maxIds) {
248                         fitness.put(id, fitness.get(id) + 1);
249                     }
250                 } else {
251                     node.setNodeValue(minChildValue);
252                     node.setId(minIds);
253                     for (Integer id : minIds) {
254                         fitness.put(id, fitness.get(id) + 1);
255                     }
256                 }
257
258                 return node.getNodeValue();
259             }
260         }
261
262         public double getFitnessValue(Individual individual) {
263             int indivId = individual.id;
264             int fitness_ = fitness.get(indivId);
265             return fitness_*fitness_;
266         }
267 3
268 }
```

5 Perbandingan Hasil

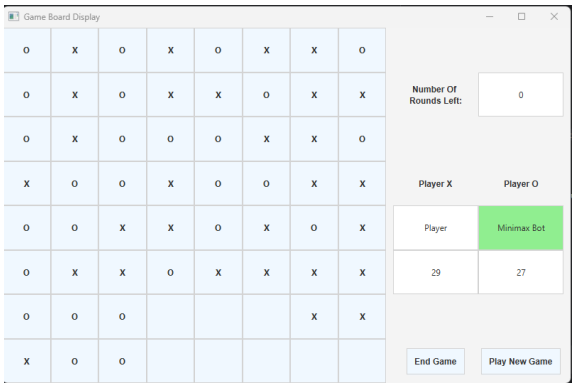
5.1 Minimax vs Human

5.1.1 Full Round (Bot Win)

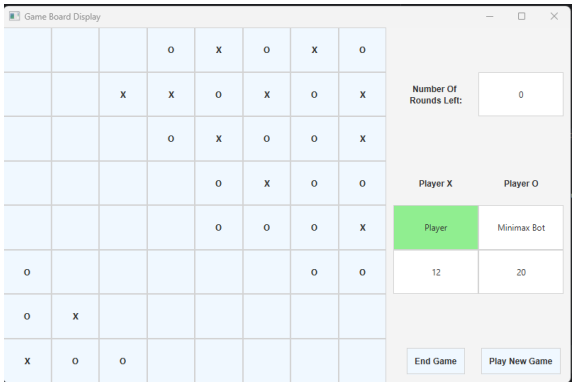
5.1.2 24 Round (Bot Win)



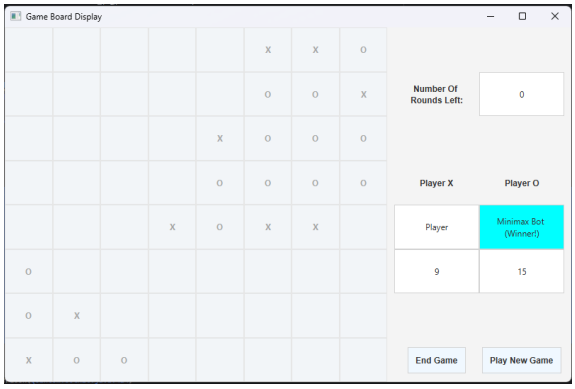
5.1.3 24 Round (Bot Win)



5.1.4 12 Round (Bot Win)



5.1.5 8 Round (Bot Win)

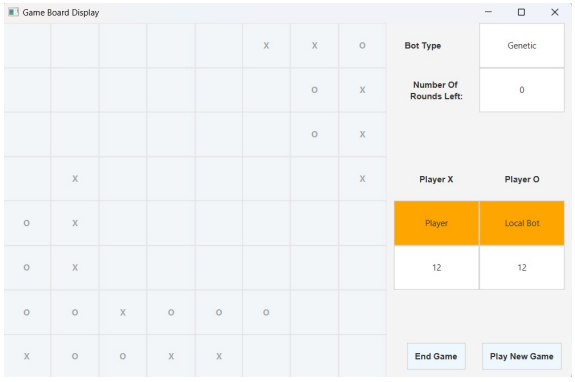


Human vs Minimax Bot

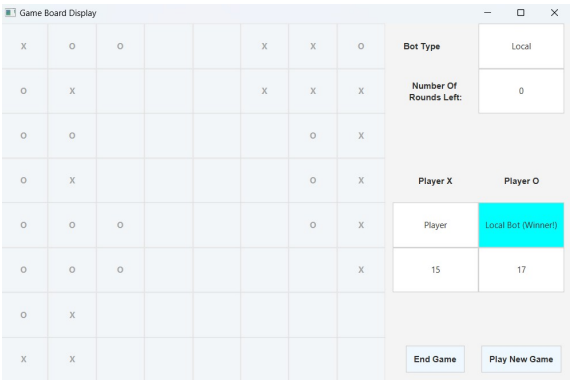
1. Total Minimax Bot win = 5
2. Total Human win = 0
3. Bot Win Percentage = 5/5 = 100.0%

5.2 Local vs Manusia

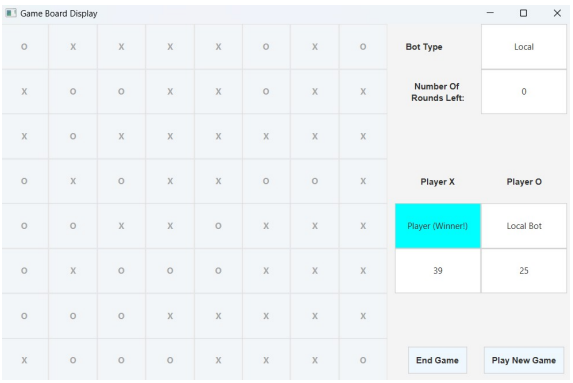
5.2.1 8 Rounds (Tie)



5.2.2 12 Rounds (Bot Win)



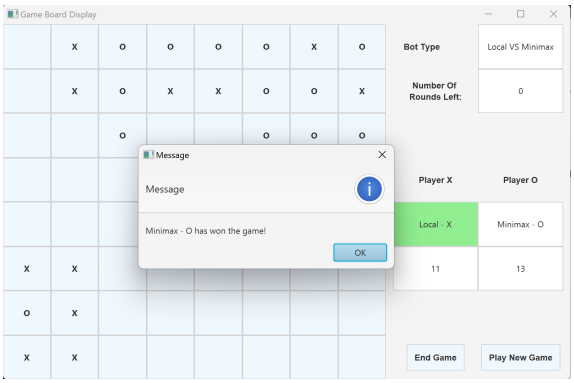
5.2.3 Full Board (Player Win)



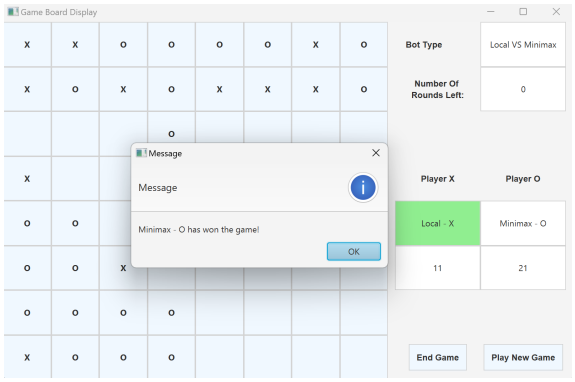
1. Win Rate = 33%
2. Tie Rate = 33%
3. Lose Rate = 33%

5.3 Minimax vs Local

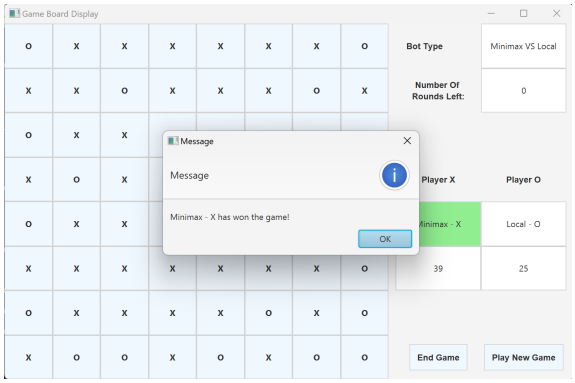
5.3.1 8 Rounds (Minimax Win)



5.3.2 12 Rounds (Minimax Win)



5.3.3 Full Board (Minimax Win)

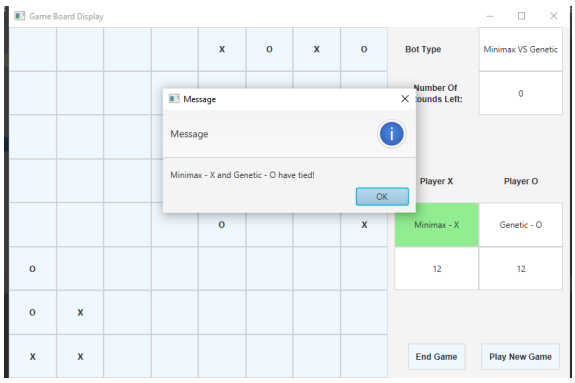


Statistik hasil,

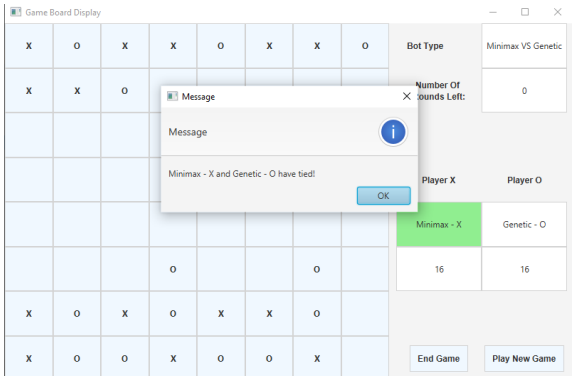
	Win	Lose	Tie
Minimax	100%	0%	0%
Local	0%	100%	0%

5.4 Minimax vs Genetic

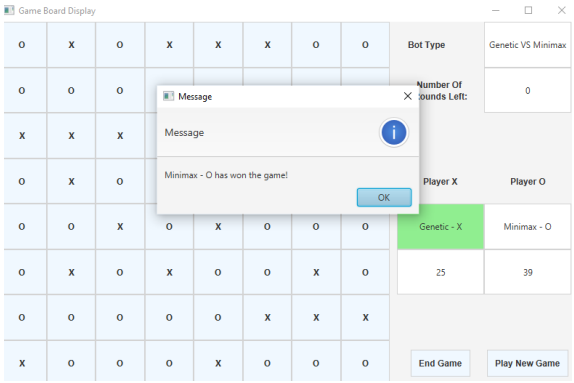
5.4.1 8 Rounds (Tie)



5.4.2 12 Rounds (Tie)



5.4.3 Full Board (Minimax Win)

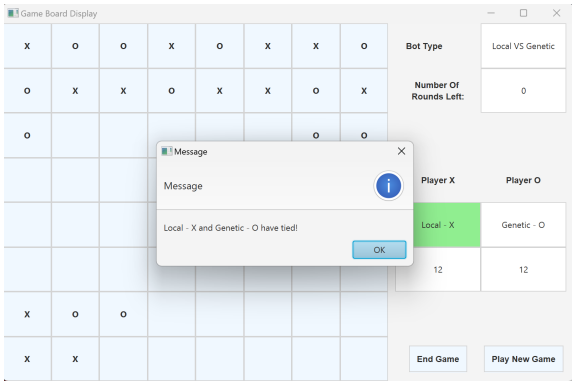


Genetic Algorithm at 12 steps/moves (Chromosomes). Minimax at 8 steps/-moves (depth)

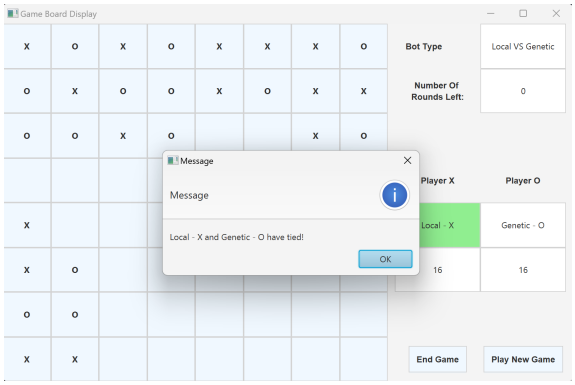
1. Win Rate = 0%
2. Tie Rate = 66%
3. Lose Rate = 33%

5.5 Local vs Genetic

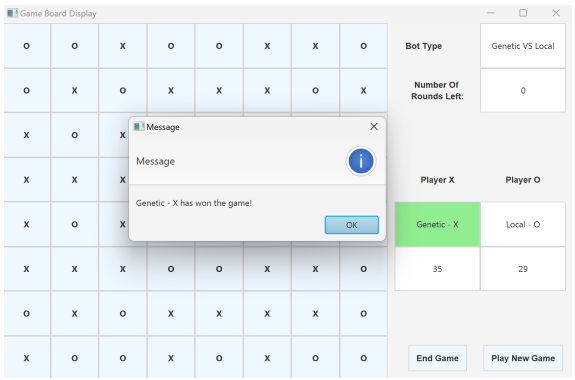
5.5.1 8 Rounds (Tie)



5.5.2 12 Rounds (Tie)



5.5.3 Full Board (Genetic Win)



Statistik hasil,

	Win	Lose	Tie
Genetic	33%	0%	66.67%
Local	0%	33%	66.67%

6 Kontribusi Anggota Kelompok

NIM	Nama	Pengerjaan
13521120	Febryan Arota Hia	Local Search Algorithm
13521123	William Nixon	Genetic Algorithm, Minimax
13521135	Nicholas Liem	MiniMax Algorithm
13521150	I Putu Bakta Hari Sudewa	Local Search Algorithm, GUI Integration

Repository: https://github.com/williamnixon20/TubesAI1

7 Daftar Pustaka

GeeksforGeeks. *Minimax Algorithm in Game Theory (Set 4: Alpha-Beta Pruning)*. 2023. URL: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/ (visited on 09/14/2023).

Hong, Tzung-Pei, Ke-Yuan Huang, and Wen-Yang Lin. "Adversarial Search by Evolutionary Computation". In: *Evolutionary Computation* 9 (Sept. 2001), pp. 371–385. DOI: 10.1162/106365601750406046.

Wikipedia. *Local Search (Optimization)*. 2023. URL: https://en.wikipedia.org/wiki/Local_search_(optimization) (visited on 09/14/2023).