Laporan Tugas Kecil 1 IF3170 Inteligensi Buatan
Semester I Tahun 2023/2024

Minimax Algorithm and Alpha Beta Pruning in Adjacency Strategy Game

Disusun oleh:

Azmi Hasna Zahrani	13521006
Ahmad Nadil	13521024

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2023

Daftar Isi

I. Struktur Program

Pada program Adversarial Adjacency-Strategy-Game terdapat source program yang berisi file Bot.java, InputFrameController.java, OutputFrameController.java, Main.java, InputFrame.fxml, OutputFrame.fxml, dan Style.css.

1.1 Main.java

Merupakan file main yang digunakan untuk melakukan inisialisasi objek-objek yang ada di dalam permainan Adversarial Adjacency-Strategy-Game.

1.2 Bot.java

Merupakan file source code program bot yang akan ditulis. File ini akan berisi semua fungsi yang dibutuhkan untuk membuat perhitungan baik menggunakan algoritma local search ataupun minimax alpha beta pruning.

1.3 InputFrameController.java

Merupakan file yang berisi semua fungsi yang mendefinisikan tampilan input permainan sebelum permainan dimulai. Terdapat beberapa input field yang digunakan seperti checkbox untuk memilih pemain yang pertama melangkah, textfield untuk input nama pemain, serta combobox yang digunakan untuk memilih banyaknya ronde permainan.

1.4 OutputFrameController.java

Merupakan file yang berisi semua fungsi yang mendefinisikan tampilan output permainan berupa game board berbentuk kotak-kotak dengan ukuran 8x8, serta atribut lain seperti scoreboard yang di dalamnya terdapat nama pemain serta jumlah score masing-masing pemain, penghitung ronde permainan, button untuk menghentikan permainan, serta button untuk memulai permainan baru.

1.5 Style.css

Merupakan file yang digunakan untuk styling tampilan pada frame input maupun output.

II. Rencana Kelas/Fungsi yang Digunakan

Kode Lengkap Implementasi Program: https://github.com/IceTeaXXD/Tucil1_13521006_13521024

2.1 Class

Akan dibuat sebuah kelas abstrak yang akan diturunkan menjadi bentuk-bentuk bot lainnya. Dengan menerapkan prinsip OOP ini, akan memudahkan pembuatan program khususnya saat penggantian algoritma yang digunakan.

a. Abstract Class Bot: Merupakan kelas abstrak yang akan diturunkan ke bentuk-bentuk bot lainnya yang menggunakan algoritma berbeda.
b. Class MinimaxBot: Merupakan kelas bot yang mengimplementasikan algoritma minimax.
c. Class HillClimbingBot: Merupakan kelas bot yang mengimplementasikan algoritma Hill-Climbing Search

2.2 Fungsi

2.2.1 Abstract Class Bot

public abstract int[] move(Button[][] board, int roundsLeft)

2.2.2 Class HillClimbingBot

public int[] move(Button[][] board, int roundsLeft)
/*
Parameter:
- Board: State board saat ini
- roundsLeft: Sisa round pada game

Return: Seubah tuple yang merupakan koordinat pergerakkan bot
*/

```
    int[] move = new int[2];
    int bestScore = Integer.MIN_VALUE;
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            // Check if the board is available
            if (board[i][j].getText().equals("")) {
                // Calculate the score of the current move
                int score = evaluate(board, i, j);
                if (score > bestScore) {
                    bestScore = score;
                    move = new int[] { i, j };
                }
            }
        }
    }
```

```
    return move;
```

private int evaluate(Button[][] board, int row, int col)
/*
Parameter:
- Board: State board saat ini
- row: Indeks baris pada board yang akan diletakkan
- col: Indeks kolom pada board yang akan diletakkan

Return : jumlah kotak yang berubah akibat penempatan pada indeks tersebut
*/

```
    int score = 0;
    if (row - 1 >= 0 && board[row - 1][col].getText().equals("X")) {
        score++;
    }
    if (row + 1 < 8 && board[row + 1][col].getText().equals("X")) {
        score++;
    }
    if (col - 1 >= 0 && board[row][col - 1].getText().equals("X")) {
        score++;
    }
    if (col + 1 < 8 && board[row][col + 1].getText().equals("X")) {
        score++;
    }
    return score;
```

2.2.3 Class MinimaxBot

public int[] move(Button[][] board, int roundsLeft)
/*
Parameter:
- Board: State board saat ini
- roundsLeft: Sisa round pada game

Return: Seubah tuple yang merupakan koordinat pergerakkan bot
*/

public int minimax(Button[][] board, int depth, int alpha, int beta, boolean isMaximizing, int roundsLeft, int i, int j)
/*
Parameter:
- Board: State board saat ini
- Depth: Kedalaman tree pada pencarian
- Alpha: Nilai alpha untuk alpha-beta pruning
- Beta: Nilai beta untuk alpha-beta pruning

- isMaximizing: Boolean untuk menandakan apakah sedang memaksimasi atau meminimasi scoring
- roundsLeft: Menandakan sisa round

*/

```
    if roundsLeft is 0 or depth is X:
      // X dapat diubah sesuai depth yang diinginkan, jika ingin
      mengiterasi seluruh tree, hapus kondisi depth
        return evaluate(board)
    prune = false
    if isMaximizing:
        bestScore = -infinity
        for i in range(8):
            if prune:
                break
            for j in range(8):
                if board[i][j] is empty:
                    copyBoard = copy(board)
                    copyBoard[i][j] = "O"
                    copyBoard = updateGameBoard(i, j, copyBoard, "O")
                    score = minimax(copyBoard, depth + 1, alpha, beta,
false, roundsLeft - 1)
                    bestScore = max(score, bestScore)
                    alpha = max(alpha, score)
                    if beta <= alpha:
                        prune = true
                        break
        if bestScore == -infinity:
            print("Maximizing No move available")
        return bestScore
    else:
        bestScore = infinity
        for i in range(8):
            if prune:
                break
            for j in range(8):
                if board[i][j] is empty:
                    copyBoard = copy(board)
                    copyBoard[i][j] = "X"
                    copyBoard = updateGameBoard(i, j, copyBoard, "X")
                    score = minimax(copyBoard, depth + 1, alpha, beta,
true, roundsLeft - 1)
                    bestScore = min(score, bestScore)
                    beta = min(beta, score)
                    if beta <= alpha:
                        prune = true
                        break
```

```
        if bestScore == infinity:
            print("Minimizing No move available")
        return bestScore
```

private Button[][] updateGameBoard(int i, int j, Button[][] board, String player)

/*

Parameter:

- i, j : Indeks baris dan kolom yang akan ditempatkan
- board: State board saat ini
- player: simbol player yang akan ditempatkan

*/

```
    if (i - 1 >= 0 && !board[i - 1][j].getText().equals(player)) {
        board[i - 1][j].setText(player);
    }
    if (i + 1 < 8 && !board[i + 1][j].getText().equals(player)) {
        board[i + 1][j].setText(player);
    }
    if (j - 1 >= 0 && !board[i][j - 1].getText().equals(player)) {
        board[i][j - 1].setText(player);
    }
    if (j + 1 < 8 && !board[i][j + 1].getText().equals(player)) {
        board[i][j + 1].setText(player);
    }

    return board;
```

public int evaluate(Button[][] board)

/*

Parameter:

- Board: State board yang akan dihitung

Return: Hasil evaluasi skoring pada board, merupakan selisih jumlah simbol Bot (O) dan simbol Player (X)

*/

```
    int playerXScore = 0;
    int playerOScore = 0;
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (board[i][j].getText().equals("X")) {
                playerXScore++;
            } else if (board[i][j].getText().equals("O")) {
                playerOScore++;
            }
        }
    }
    return playerOScore - playerXScore;
```

III. Objective Function yang Digunakan

3.1 Class HillClimbingBot

3.1.1 *public int[] move(Button[][] board, int roundsLeft)*

Fungsi ini bertujuan untuk melakukan iterasi sepanjang board dan melakukan simulasi penempatan / gerakan pada sebuah kotak yang kosong. Setelah mensimulasi pergerakan tersebut, nantinya fungsi ini akan memanggil fungsi *evaluate* untuk melakukan kalkulasi gerakan yang mengembalikan nilai dari gerakan tersebut (akan dijelaskan lebih lanjut). Pada fungsi move ini akan juga ditentukan gerakan manakah yang memiliki nilai terbaik. Gerakan yang memiliki nilai terbaik akan dikembalikan dalam bentuk tuple yang berisi indeks baris dan kolom gerakan terbaik.

3.1.2 *private int evaluate(Button[][] board, int row, int col)*

Fungsi evaluate akan melakukan simulasi penempatan pada kotak yang kosong. Fungsi ini akan melakukan penilaian terhadap gerakan tersebut dengan menghitung berapa banyak kotak *adjacent* yang berubah simbolnya menjadi simbol O (simbol yang digunakan bot). Hal ini akan menentukan seberapa baik gerakan tersebut, semakin bagus gerakan tersebut, semakin banyak kotak yang berubah simbolnya.

Rumus Perhitungan:

$$score = PlayerOCount(boardState) - PlayerXCount(boardState)$$

3.2 Class MinimaxBot

3.2.1 *public int[] move(Button[][] board, int roundsLeft)*

Sama dengan fungsi pada Class HillClimbingBot, fungsi ini bertujuan untuk melakukan iterasi sepanjang board dan melakukan simulasi penempatan / gerakan pada sebuah kotak yang kosong. Setelah mensimulasi pergerakan tersebut, nantinya fungsi ini akan memanggil fungsi *minimax* untuk melakukan kalkulasi gerakan yang mengembalikan nilai dari gerakan tersebut (akan dijelaskan lebih lanjut). Pada fungsi move ini akan juga ditentukan gerakan manakah yang memiliki nilai terbaik. Gerakan yang memiliki nilai terbaik akan dikembalikan dalam bentuk tuple yang berisi indeks baris dan kolom gerakan terbaik.

3.2.2 *public int minimax(Button[][] board, int depth, int alpha, int beta, boolean isMaximizing, int roundsLeft)*

Fungsi ini merupakan fungsi utama untuk melakukan kalkulasi dan penilaian terhadap sebuah gerakan. Fungsi ini merupakan fungsi rekursif yang akan terus

berjalan sampai ronde permainan habis atau sampai tingkat kedalaman tertentu. Fungsi minimax akan jauh lebih optimal hasilnya dibanding algoritma *Local Search*, karena akan mendapatkan *Global Optimum*. Hal ini dikarenakan ada fungsi ini akan terus mencari sampai *terminal state*, yaitu sampai ronde habis atau kedalaman tertentu.

3.2.3. *private int evaluate(Button[][] board, int row, int col)*

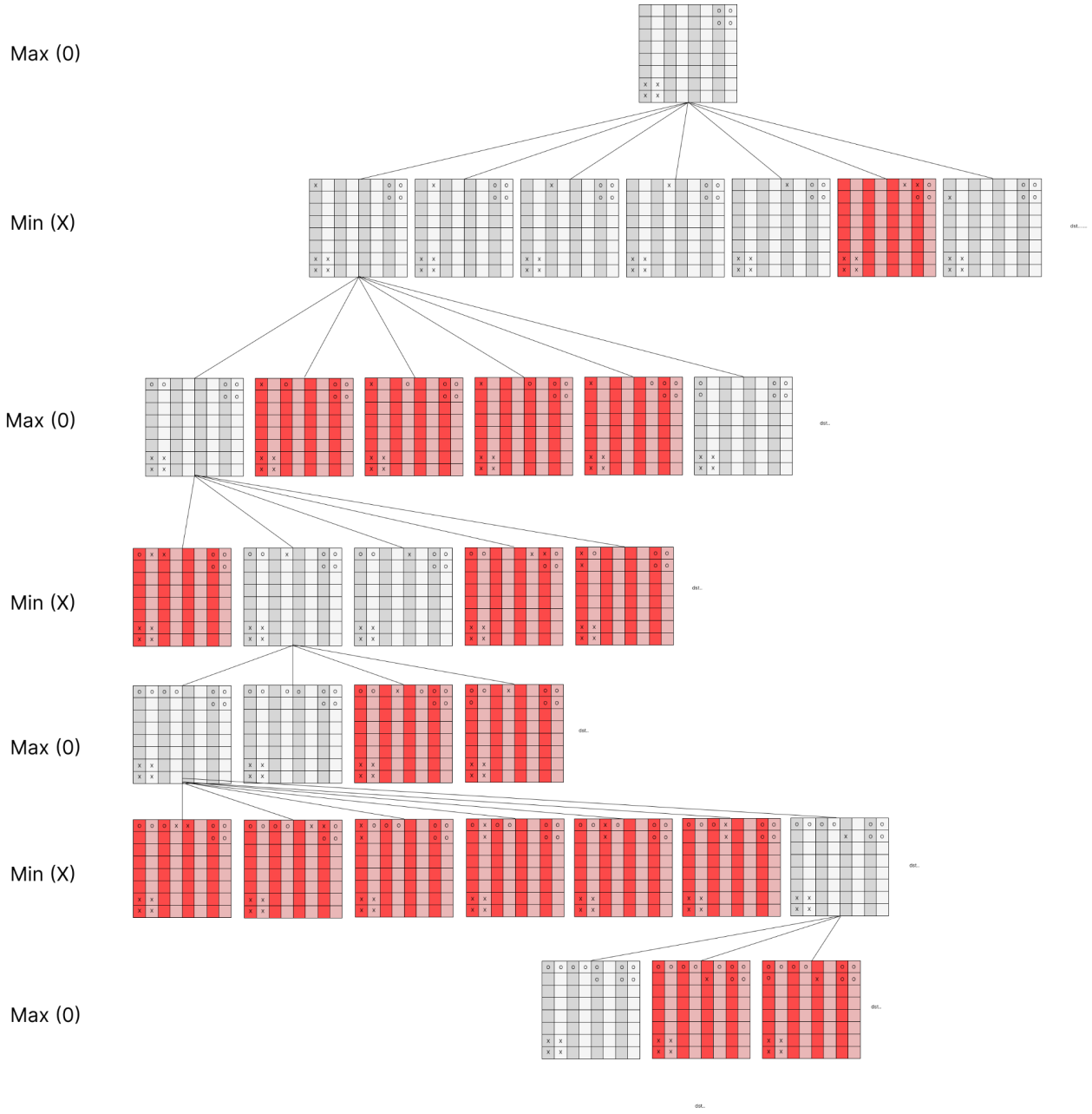
Fungsi evaluate akan melakukan simulasi penempatan pada kotak yang kosong. Fungsi ini akan melakukan penilaian terhadap gerakan tersebut dengan menghitung berapa banyak kotak *adjacent* yang berubah simbolnya menjadi simbol O (simbol yang digunakan bot). Hal ini akan menentukan seberapa baik gerakan tersebut, semakin bagus gerakan tersebut, semakin banyak kotak yang berubah simbolnya.

Rumus Perhitungan:

$$score = PlayerOCount(boardState) - PlayerXCount(boardState)$$

IV. Proses Pencarian Minimax Alpha Beta Pruning pada Permainan Adjacency Strategy Game

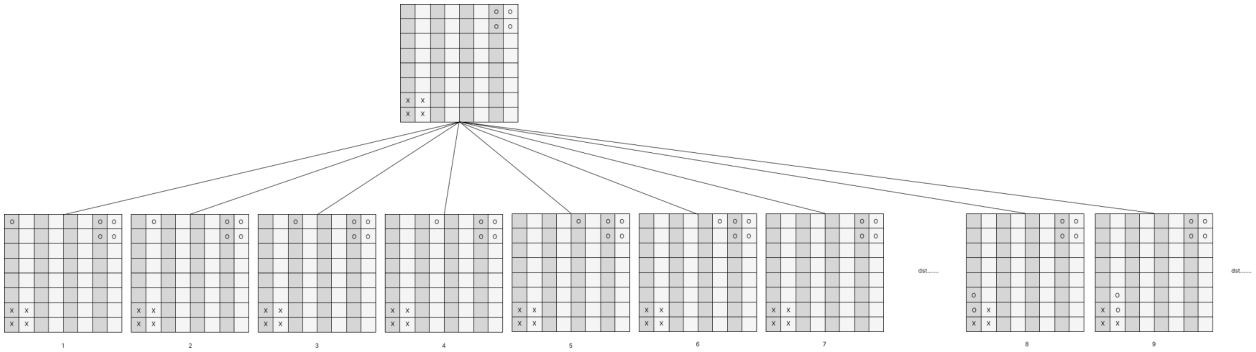
Permainan Adjacency Strategy Game dapat dimainkan dengan proses pencarian minimax dengan alpha beta pruning. Proses pencarian minimax alpha beta pruning dilakukan dengan cara memberi nilai maksimal pada langkah bot dan memberi nilai minimal pada langkah lawan. Hasil akhir dari algoritma ini akan menghasilkan *Global Optimum* dikarenakan algoritma ini mengiterasi seluruh kemungkinan yang dapat dihasilkan oleh state permainan. Algoritma Minimax dapat direpresentasikan pada tree di bawah ini.



Pada tree di atas merepresentasikan bagaimana algoritma minimax dengan alpha beta pruning bekerja. Pada tree di atas, satu state digambarkan dengan satu tingkatan tree. Operasi pencarian akan tetap dilanjutkan ketika pada nilai Max (O) didapatkan nilai terbesar pada state dan akan berhenti (tidak akan dilakukan operasi pencarian lagi di state setelahnya) ketika ditemukan nilai yang lebih kecil dari nilai maksimal yang seharusnya didapat pada state tersebut. Begitu pula pada operasi Min (X) apabila ditemukan nilai yang lebih besar dari nilai state saat ini maka state tersebut akan dimatikan dan tidak akan dilanjutkan pencarian ke state berikutnya. Hal ini dapat mempersingkat operasi dalam algoritma karena tidak perlu menghitung kemungkinan keseluruhan state.

V. Proses Pencarian dengan Algoritma Local Search pada Permainan Adjacency Strategy Game

Salah satu algoritma *local search* yang dapat digunakan pada permainan *adjacency strategy game* adalah *hill-climbing search*. *Hill-climbing search* berjalan dengan mengiterasi peletakan bidak pemain dan mengambil skor tertinggi yang dapat diperoleh pada state tersebut. Algoritma ini berfokus pada skor sendiri tanpa memperdulikan skor lawan. Ilustrasi algoritma dapat dilihat pada tree di bawah ini.



Dari ilustrasi algoritma di atas, kemungkinan solusi yang akan diambil adalah state yang diawali oleh ilustrasi nomor 8 atau 9 karena kedua state tersebut akan menghasilkan skor yang paling tinggi. Hal ini karena kedua state ini memiliki potensi untuk mencapai skor maksimum global yang diinginkan dalam permainan ini.

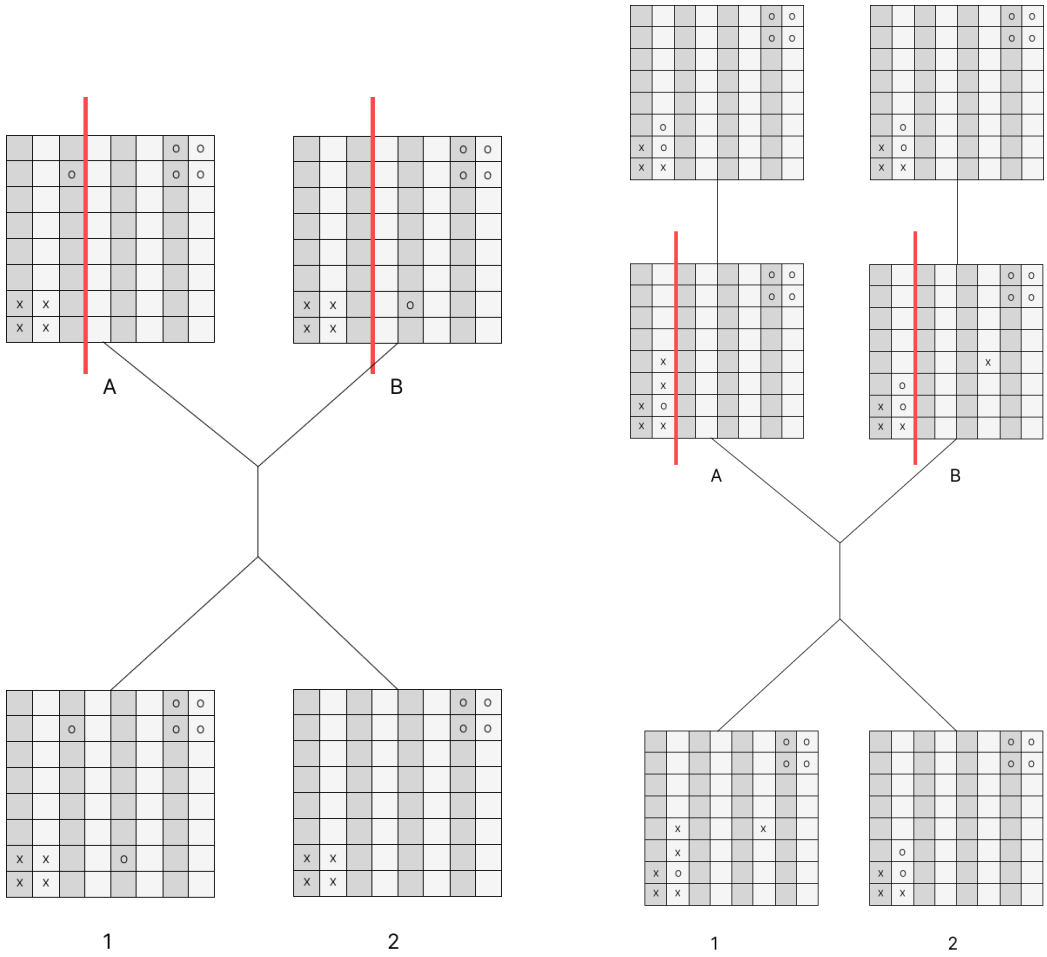
Pada permainan ini, algoritma *hill-climbing search* hanya dapat diterapkan untuk mendapatkan *Local Optimum*. Hal ini disebabkan tidak ada *depth* atau kedalaman pada tree, atau dalam kata lain hanya mencari gerakan dengan nilai terbaik pada state saat ini dan tidak melakukan kalkulasi lebih lanjut untuk state selanjutnya.

Hill-climbing search bekerja dengan algoritma sebagai berikut:

```
current <- Make-Node(problem.Initial-state)
loop do
     neighbor <- a highest-valued successor of current
     if neighbor.value <= current.value
     current <- neighbor
```

VI. Strategi Permainan Langkah Optimum dengan Genetic Algorithm

Penerapan *genetic algorithm* pada permainan ini tidak mungkin dilakukan. Alasannya adalah karena algoritma tersebut menghasilkan state-state yang tidak bertetangga dengan state saat ini karena terjadi kawin silang dan mutasi. Maka dari itu, algoritma tersebut tidak dapat diterapkan karena dalam 1 giliran, kita hanya dapat bergerak satu kali, atau dalam kata lain kita hanya dapat sampai pada state tetangga. Hal ini disebabkan oleh tahap crossover yang akan menyebabkan bidak-bidak yang telah diletakkan pada papan permainan berubah dengan menyalahi aturan. Kejadian ini terbagi ke beberapa kasus, yaitu permainan kembali ke state sebelumnya (parent), peletakan bidak pemain yang lebih dari satu, serta permainan kembali ke state awal (kondisi 4 bidak bot dan 4 bidak pemain). Contoh kasus-kasus tersebut akan dijelaskan dengan ilustrasi gambar di bawah ini.



Pada gambar sebelah kiri, terjadi kasus crossover yang menghasilkan bot menaruh bidaknya sebanyak 2 (pada papan nomor 1) dan permainan yang kembali ke state awal (pada papan nomor 2). Sedangkan pada gambar sebelah kanan, terjadi kasus crossover yang mengakibatkan permainan kembali ke state sebelumnya dengan menghapus bidak pemain X yang telah diletakkan (pada papan nomor 2). Kedua ilustrasi tersebut menggambarkan pelanggaran-pelanggaran peraturan permainan sangat mungkin terjadi apabila adjacency strategy

game dilakukan dengan genetic algorithm. Faktor penyebabnya adalah penentuan peletakan bidak dan posisi crossover yang dilakukan secara random. Maka dari itu, disimpulkan genetic algorithm tidak bisa menyelesaikan permainan Adjacency Strategy Game.

REFERENSI

[1] GeeksforGeeks. (2018, October 5). *Minimax Algorithm in Game Theory | Set 1 (Introduction) - GeeksforGeeks*. GeeksforGeeks. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

[2] GeeksForGeeks. (2016, July 24). *Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning)*. GeeksforGeeks. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

[3] Khodra, Masayu Leylia. 2023. Adversarial Search.

[4] Khodra, Masayu Leylia. 2023. Genetic Algorithm.

[5] Khodra, Masayu Leylia. 2023. Hill-climb Algorithm.