Penerapan Algoritma Greedy dan Brute Force dalam Permainan Checkers

Muhammad Dzaki Arta - 13522149
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522149@std.stei.itb.ac.id

Abstract— **Makalah ini mengeksplorasi penerapan algoritma Brute Force dan Greedy dalam permainan Checkers. Checkers, sebuah permainan papan klasik dengan sejarah yang kaya, melibatkan pergerakan strategis dan penangkapan bidak lawan. Studi ini mengimplementasikan kedua algoritma untuk menentukan langkah optimal selama permainan. Algoritma Brute Force secara sistematis mengevaluasi semua kemungkinan langkah untuk memastikan solusi terbaik ditemukan, sementara algoritma Greedy secara heuristik memilih langkah yang tampak optimal secara lokal, seperti mempromosikan bidak menjadi "King". Melalui implementasi dalam bahasa Java, makalah ini menunjukkan kekuatan dan keterbatasan masing-masing pendekatan, menyimpulkan bahwa Brute Force menjamin solusi optimal tetapi membutuhkan waktu komputasi yang besar, sedangkan Greedy lebih efisien tetapi tidak selalu menemukan solusi optimal. Contoh praktis disediakan untuk menggambarkan kinerja dan efektivitas kedua algoritma dalam berbagai skenario permainan.**

Keywords— Checkers; Algoritma Brute Force; Algoritma Greedy; Strategi Permainan; Langkah Optimal; Implementasi Java.

I. PENDAHULUAN

Checkers, juga dikenal sebagai draughts, adalah permainan papan klasik yang melibatkan dua pemain dan papan 8x8 dengan 32 buah bidak. Permainan ini memiliki sejarah panjang dan kaya, dengan bukti arkeologis yang menunjukkan bahwa permainan serupa dimainkan sejak zaman kuno. Checkers populer di seluruh dunia dan dinikmati oleh orang-orang dari segala usia dan tingkat keahlian. Setiap pemain dalam permainan checkers memiliki masing masing 12 bidah. Pemain pertama biasanya memiliki bidak putih atau merah dan pemain kedua memiliki bidak hitam.

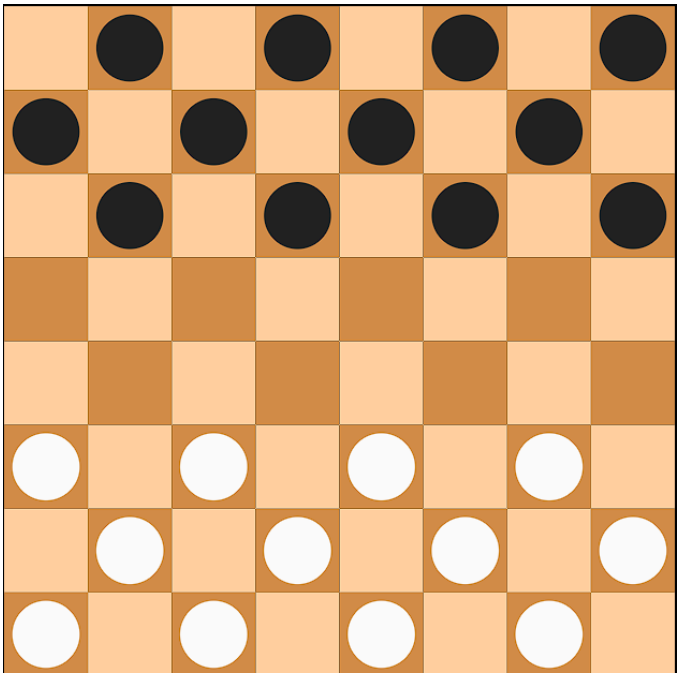
Pemain dengan bidak putih akan jalan terlebih dahulu. Disusul dengan pemain dengan bidak hitam. Setiap bidak berada pada kotak berwarna hitam pada papan permainan dengan ukuran 8x8. Bidak pada permainan *Checkers* dibedakan menjadi 2 yaitu ”Man” dan ”King”.

Man adalah bidak biasa dengan kemampuan hanya dapat berjalan maju kekanan dan kekiri. Man punya kemampuan untuk lompat dengan syarat terdapat bidak musuh di petak diagonal Man, dan petak yang dilompati haruslah kosong. Man dapat melompati petak berkali-kali dan setiap bidak musuh yang dilompat akan ”termakan” atau bidak musuh tersebut akan . Pada saat Man telah sampai di petak akhir Man akan berubah menjadi ”King”. King biasanya di tandai dengan 2 buah bidak checkers yang ditumpuk. Ketika Man bisa melakukan jump maka dia haruslah melakukannya.

King bisa bergerak diagonal maju dan diagonal mundur. King juga bisa melompat diagonal maju dan melompat diagonal mundur. Oleh karena itu, king menjadi piece terkuat dalam permainan Checkers.

Pada awal permainan 2 buah pemain akan diberi masing-masing 12 men dan akan di letak di 3 baris petak hitam pertama dari masing-masing pemain. Sehingga akan terdapat 12 men putih di bagian bawah papan, dan 12 men hitam di bagian atas papan.

Gambar 1. Tampilan awal permainan Checkers



Untuk memenangkan permainan Chekers bidak lawan harus tidak dapat bergerak atau seluruh bidak lawan sudah termakan alias tidak ada lagi bidak lawan di papan.

II. DASAR TEORI

A. Algoritma Brute Force

Algoritma Brute Force (BFA) merupakan algoritma pencarian solusi secara menyeluruh Algoritma Brute Force (Brute Force Algorithm) hadir sebagai pendekatan yang lugas namun kuat, secara sistematis mengevaluasi semua solusi yang mungkin untuk mencapai solusi yang optimal. Kesederhanaan dan jaminan optimalitasnya menjadikannya alat yang berharga untuk menangani jenis masalah tertentu. Namun, tuntutan komputasinya dapat menjadi batasan signifikan untuk masalah besar atau kompleks.

Pada intinya, BFA terletak pada evaluasi menyeluruhnya terhadap setiap solusi yang dapat dibayangkan dalam ruang masalah. Pendekatan cermat ini memastikan bahwa tidak ada solusi potensial yang terlewatkan, yang mengarah pada identifikasi solusi terbaik secara absolut.

Algoritma ini biasanya memecahkan pesoalan dengan sangat sederhana, langsung, jelas caranya. Terdapat beberapa karakteristik dari BFA ini yaitu

- **Jaminan Optimalitas:** BFA memberikan jaminan teguh untuk menemukan solusi optimal. Dengan tidak meninggalkan kemungkinan yang belum teruji, ia memastikan bahwa solusi yang diidentifikasi benar-benar yang terbaik.
- **Kesederhanaan dan Kemudahan Implementasi**: Kesederhanaan konseptual BFA diterjemahkan menjadi implementasi yang mudah. Pendekatan algoritma yang jelas dan terstruktur membuatnya relatif mudah untuk diprogram dan dipahami.
- **Keserbagunaan:** BFA dapat diterapkan pada berbagai jenis masalah, terutama yang memiliki ruang solusi yang terbatas dan terdefinisi dengan baik.

Algoritma ini juga memiliki beberapa keterbatasan.

- **Kompleksitas Eksponensial:** Kelemahan utama BFA terletak pada kompleksitas eksponensialnya. Saat ruang masalah berkembang, jumlah solusi potensial tumbuh secara eksponensial, yang menyebabkan peningkatan drastis dalam waktu komputasi yang diperlukan. Hal ini membuat BFA tidak praktis untuk masalah skala besar atau sensitif waktu.
- **Inefisien untuk Ruang Solusi Besar:** Saat berhadapan dengan masalah dengan sejumlah besar solusi potensial, evaluasi menyeluruh BFA menjadi tidak efisien. Algoritma alternatif yang berfokus pada pemangkasan ruang pencarian atau menggunakan heuristik mungkin lebih cocok.

Inti dari Algoritma Brute Force terletak pada evaluasi menyeluruhnya terhadap setiap solusi yang dapat dibayangkan dalam ruang masalah. Pendekatan cermat ini memastikan bahwa tidak ada solusi potensial yang terlewatkan, yang mengarah pada identifikasi solusi terbaik secara absolut.

Pada Algoritma Brute Force yang dipakai untuk checkers. Program akan mencari apakah ada bidak yang dapat memakan, dan kemudian program akan me iterasi satu per satu bidak tersebut dan kemudian program akan mencari jalan terbaik di setiap bidak sehingga akan mendapatkan 1 solusi terbaik.

B. Algoritma Greedy

Algoritma Greedy (GA) hadir sebagai pendekatan pragmatis dan intuitif untuk pemecahan masalah, menawarkan solusi heuristik untuk berbagai masalah optimasi. Berbeda dengan evaluasi menyeluruh dari semua kemungkinan oleh Algoritma Brute Force, GA membuat pilihan terbaik secara lokal pada setiap langkah, membangun solusi secara bertahap. Efisiensi ini menjadikannya pilihan menarik untuk berbagai aplikasi.

Pada intinya, GA beroperasi dengan prinsip sederhana: buat pilihan terbaik secara lokal pada setiap langkah, dengan harapan bahwa efek kumulatif dari pilihan lokal ini akan mengarah pada solusi optimal secara global. Pendekatan heuristik ini mengorbankan jaminan optimalitas demi efisiensi, membuatnya cocok untuk masalah di mana menemukan solusi optimal mungkin rumit atau memakan waktu.

Algoritma greedy dapat memecahkan masalah dengan lebih cepat. Terdapat beberapa karakteristik dari GA ini yaitu

- **Efisiensi:** Pendekatan bertahap GA membuatnya efisien secara komputasi, terutama untuk masalah dengan ruang solusi yang besar.
- **Kesederhanaan:** Konsep GA mudah dipahami dan mudah diakses oleh berbagai pengguna.
- **Kemampuan Beradaptasi:** GA dapat diadaptasi ke berbagai domain masalah dengan mendefinisikan fungsi heuristik yang sesuai untuk memandu proses pengambilan keputusan.

Algoritma ini juga memiliki beberapa keterbatasan.

- **Tidak Ada Jaminan Optimalitas:** Sifat heuristik GA tidak menjamin bahwa ia akan menemukan solusi optimal. GA dapat terjebak dalam optima lokal, mengabaikan solusi yang lebih baik di tempat lain dalam ruang pencarian.
- **Sensitivitas terhadap Desain Heuristik:** Kinerja GA sangat bergantung pada kualitas fungsi heuristik. Fungsi heuristik yang dirancang dengan buruk dapat menghasilkan solusi yang suboptimal atau bahkan salah.
- **Penerapan Spesifik Masalah:** Efektivitas GA sangat bergantung pada masalah. GA bekerja dengan baik untuk masalah dengan struktur yang jelas dan fungsi tujuan yang dapat didekomposisi.

Inti dari Algoritma Greedy terletak pada evaluasi yang memilih secara heuristik. Pendekatan ini cenderung lebuh efisien untuk persoalan yang besar tetapi akan sensitif terhadap desain heuristik dan biasanya akan terdapat persoalan yang tidak memiliki solusi dikarenakan pendekatan heuristiknya.

Algoritma Greedy akan di pakai dalam permainan checkers dengan fungsi heuristik yaitu “Man” yang dapat menjadi raja. Pertama program akan melakukan iterasi untuk mencari piece mana saja yang dapat digerakan dan kemudian program akan

mencari piece yang dapat menjadi King dan mencari solusi terbaik dari piece yang dapat menjadi King tersebut.

III. IMPLEMENTASI

Program yang dibuat akan di implementasikan dalam bahasa java. Progam mempunyai 3 kelas utama yaitu kelas checkers yang berisi main program. Kelas bruteForceSolution yang berisi implementasi solusi brute force. Dan kelas greedySolution yang berisi implementasi kelas greedy. Terdapat juga kelas load yang berisi implementasi muat untuk file yang akan di gunakan sebagai masukan. Brute Force solution akan mempunyai fungsi fungsi yang dapat membantu dalam penyelesaian solusi tersebut dan fungsi utama yaitu solution me-aplikasikan algoritma brute force. Greedy Solution juga akan memiliki fungsi fungsi yang dapat membantu dalam penyelesaian pesoalan menggunakan algoritma greedy

A. Main program

Main program akan meminta masukan dari user untuk tata letak piece yang akan digunakan nantinya. User dapat memasukan secara manual atau masukan nama file yang merepresentasikan turn dan susunan board checkers. Turn akan hanya berisi nomor 1 atau 2 yang menandakan turn player 1 atau player 2. Matrix 8x8 susunan papan yang merepresentasikan papan permainan checkers, angka 1 pada matrix papan menandakan bahwa bidak tersebut punya player 1 , angka 2 pada papan menandakan bidak tersebut dimiliki oleh player 2, dan angka 0 pada papan menandakan petak kosong.

```
<Turn Player>
<Matrix 8x8 susunan board>
```

Contoh file masukan user dengan turn 1

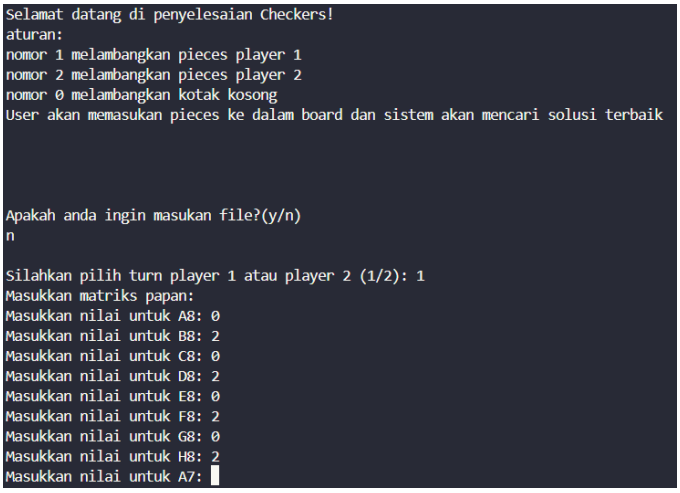
```
1
0 0 0 0 0 0 0 0
2 0 2 0 0 0 2 0
0 0 0 1 0 0 0 0
2 0 2 0 0 0 2 0
0 0 0 0 0 2 0 1
1 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1
0 0 1 0 0 0 0 0
```

Contoh file masukan user dengan turn 2

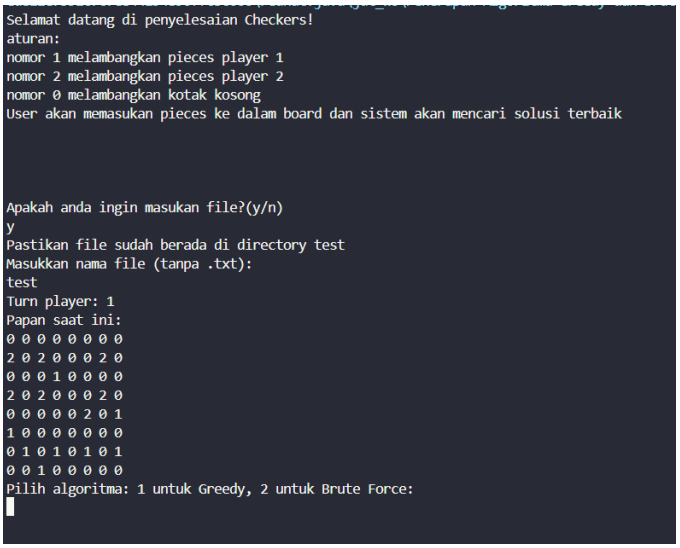
```
2
0 0 0 2 0 2 0 0
0 0 0 0 0 0 0 0
0 2 0 2 0 0 0 2
2 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0
0 0 1 0 0 0 2 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 1 0
```

Pada file tersebut turn player merupakan player 1 dan matrix 8x8 merepresentasikan kondisi papan saat ini. Oleh karena itu program nantinya akan mencari solusi terbaik untuk player 1 menggunakan algoritma Brute Force ataupun Greedy.

Gambar 3.1 Tampilan program saat masukan manual dari user



Gambar 3.2 Tampilan program saat masukan file dari user



B. bruteForceSolution

Kelas bruteForceSolution merupakan kelas yang akan meimplementasikan algoritma brute force dan akan melakukan iterasi terhadap seluruh piece yang bisa digerakan dan akan mencari poin tertinggi untuk piece tersebut. Terdapat beberapa fungsi pendukung untuk kelas ini yaitu

- findAllPieceHasMove

Fungsi ini akan mengembalikan setiap piece yang bisa digerakan

```
public List<String> findAllPieceHasMove() {
    List<String> list = new ArrayList<>();
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if ((turn == 1 && board[i][j] == 1) || (turn == 2 && board[i][j] == 2)) {
                if (isPieceHasMove(j, i)) {
                    list.add(j + " " + i);
                }
            }
        }
    }
    return list;
}
```

- findBestMoveBruteForce

Fungsi ini akan melakukan iterasi kepada setiap piece dan akan mengkalkulasi piece yang bisa melompati paling banyak

```
public List<String> findBestMoveBruteForce(List<String> listPiece) {
    List<String> bestMoves = new ArrayList<>();
    String bestPiece = listPiece.get(0);
    int maxJumps = 0;

    for (String piece : listPiece) {
        int x = Integer.parseInt(piece.split(" ")[0]);
        int y = Integer.parseInt(piece.split(" ")[1]);
        List<String> jumps = jumpPieceList(x, y, new ArrayList<>());
        int jumpCount = jumps.size();
        if (jumpCount > maxJumps) {
            maxJumps = jumpCount;
            bestPiece = piece;
            bestMoves = jumps;
        }
    }

    bestMoves.add(0, bestPiece); // Add the initial position to the bestMoves list
    return bestMoves;
}
```

- jumpPieceList

Fungsi ini akan melakukan pencarian rekursif untuk menentukan banyak jump terbanyak yang bisa dilakukan dari 1 piece

```
public List<String> jumpPieceList(int x, int y, List<String> result) {
    if (x < 0 || x >= size || y < 0 || y >= size || !isPieceHasMove(x, y)) {
        return result;
    } else {
        if (isPieceCanJumpLeft(x, y)) {
            result.add((x - 2) + " " + (turn == 2 ? y + 2 : y - 2));
            jumpPieceList(x - 2, (turn == 2 ? y + 2 : y - 2), result);
        }
        if (isPieceCanJumpRight(x, y)) {
            result.add((x + 2) + " " + (turn == 2 ? y + 2 : y - 2));
            jumpPieceList(x + 2, (turn == 2 ? y + 2 : y - 2), result);
        }
        return result;
    }
}
```

- executeBestMove

Setelah mendapatkan gerakan terbaik fungsi ini akan melakukan eksekusi dan menampilkan ke layar solusi gerakan yang diberikan

```
private void executeBestMove(List<String> bestMove) {
    char x = (char) ('A' + Integer.parseInt(bestMove.get(0).split(" ")[0]));
    int y = Integer.parseInt(bestMove.get(0).split(" ")[1]);
    String index = x + "" + (8 - y);
    System.out.println("Player " + turn + " bergerak dari " + index);
    for (int i = 1; i < bestMove.size(); i++) {
        x = (char) ('A' + Integer.parseInt(bestMove.get(i).split(" ")[0]));
        y = Integer.parseInt(bestMove.get(i).split(" ")[1]);
        index = x + "" + (8 - y);
        System.out.println("Player " + turn + " melompati " + index);
```

```
        }
    }
```

- solution

Fungsi solution merupakan fungsi yang menggabungkan fungsi-fungsi pendukung dan akan melakukan pencarian solusi bruteforce untuk masalah yang diberikan.

```
    public void solution() {
        System.out.println("Solusi menggunakan algoritma Brute Force:");
        printBoard();

        List<String> listPiece = findAllPieceHasMove();
        if (listPiece.isEmpty()) {
            System.out.println("Tidak ada bidak yang bisa bergerak player " + turn + " kalah.");
            return;
        }

        List<String> bestMove = findBestMoveBruteForce(listPiece);
        if (bestMove.isEmpty()) {
            System.out.println("Tidak ada bidak yang bisa makan");
            String piece = listPiece.get(0);
            int x = Integer.parseInt(piece.split(" ")[0]);
            int y = Integer.parseInt(piece.split(" ")[1]);
            String move = findMove(piece);
            if (move != null) {
                System.out.println("Player " + turn + " bergerak dari " + x + " " + y + " ke " + move);
            } else {
                System.out.println("Tidak ada langkah yang bisa dilakukan player " + turn + " kalah.");
            }
        } else {
            executeBestMove(bestMove);
        }
    }
```

C. greedySolution

Kelas greedy solution merupakan kelas untuk mencari solusi dengan menggunakan algoritma greedy. Untuk greedy yang dipakai merupakan pendekatan piece yang bisa menjadi King. Ada beberapa fungsi tambahan untuk pendekatan algoritma greedy yaitu

- isPieceCanBeKing

Fungsi ini akan memeriksa apakah piece bisa menjadi King dengan melompati piece-piece musuh

```
    public boolean isPieceCanBeKing(int x, int y) {
        List<String> jumps = new ArrayList<>();
        List<String> list = jumpPieceList(x, y, jumps);
        for (String jump : list) {
            int newY = Integer.parseInt(jump.split(" ")[1]);
            if ((turn == 2 && newY == size - 1) || (turn == 1 && newY == 0)) {
                return true;
            }
        }
        return false;
    }
```

- findBestMove

Fungsi ini akan mencari pergerakan terbaik dengan memanfaatkan fungsi isPieceCanBeKing untuk algoritma greedy nantinya

```
    public List<String> findBestMove(List<String> listPiece) {
        List<String> bestMoves = new ArrayList<>();
        String bestPiece = listPiece.get(0);
        int maxJumps = 0;

        for (String piece : listPiece) {
            int x = Integer.parseInt(piece.split(" ")[0]);
            int y = Integer.parseInt(piece.split(" ")[1]);
            if(isPieceCanBeKing(x, y)) {
                bestPiece = piece;
                List<String> jumps = jumpPieceList(x, y, new ArrayList<>());
                int jumpCount = jumps.size();
                System.out.println("Bidak " + x + " " + y + " bisa melompati " + jumpCount + " bidak");
                if (jumpCount > maxJumps) {
                    maxJumps = jumpCount;
                    bestPiece = piece;
```

```
                }
            }
        }

        bestMoves.add(bestPiece);
        bestMoves.addAll(jumpPieceList(
            Integer.parseInt(bestPiece.split(" ")[0]),
            Integer.parseInt(bestPiece.split(" ")[1]),
            new ArrayList<>()
        ));

        return bestMoves;
    }
```

- solution

Fungsi ini akan mencari solusi terbaik dari persoalan yang diberikan menggunakan algoritma greedy

```
    public void solution() {
        System.out.println("Solusi menggunakan algoritma Greedy:");
        printBoard();

        List<String> listPiece = findAllPieceHasMove();
        if (listPiece.isEmpty()) {
            System.out.println("Tidak ada bidak yang bisa bergerak player " + turn + " kalah.");
            return;
        }

        List<String> bestMove = findBestMove(listPiece);
        if (bestMove.isEmpty()) {
            System.out.println("Tidak ada bidak yang bisa makan");
            String piece = listPiece.get(0);
            int x = Integer.parseInt(piece.split(" ")[0]);
            int y = Integer.parseInt(piece.split(" ")[1]);
            String move = findMove(piece);
            if (move != null) {
                System.out.println("Player " + turn + " bergerak dari " + x + " " + y + " ke " + move);
            } else {
                System.out.println("Tidak ada langkah yang bisa dilakukan player " + turn + " kalah.");
            }
        } else {
            executeBestMove(bestMove);
        }
    }
```

D. Test Case

- Test case 1

File test case yang akan digunakan adalah sebagai berikut

```
1
0 0 0 0 0 0 0 0
2 0 2 0 0 0 2 0
0 0 0 1 0 0 0 0
2 0 2 0 0 0 2 0
0 0 0 0 0 2 0 1
1 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1
0 0 1 0 0 0 0 0
```

Gambar 3.3 solusi test case menggunakan algoritma greedy

```
Solusi menggunakan algoritma Greedy:
0 0 0 0 0 0 0 0
2 0 2 0 0 0 2 0
0 0 0 1 0 0 0 0
2 0 2 0 0 0 2 0
0 0 0 0 0 2 0 1
1 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1
0 0 1 0 0 0 0 0
Bidak 3 2 bisa melompati 1 bidak
Bidak 7 4 bisa melompati 2 bidak
Player 1 bergerak dari H4
Player 1 melompati F6
Player 1 melompati H8
```

Gambar 3.4 Solusi test case 1 menggunakan algoritma Brute Force

```
Solusi menggunakan algoritma Brute Force:
0 0 0 0 0 0 0 0
2 0 2 0 0 0 2 0
0 0 0 1 0 0 0 0
2 0 2 0 0 0 2 0
0 0 0 0 0 2 0 1
1 0 0 0 0 0 0 0
0 1 0 1 0 1 0 1
0 0 1 0 0 0 0 0
Player 1 bergerak dari H4
Player 1 melompati F6
Player 1 melompati H8
```

```
Solusi menggunakan algoritma Brute Force:
0 0 0 2 0 2 0 0
0 0 0 0 0 2 0 0
0 2 0 2 0 0 1 0
2 0 0 0 0 0 2 0
0 1 0 1 0 0 0 0
0 0 1 0 2 0 2 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 1 0
Player 1 bergerak dari D2
Player 1 melompati F4
Player 1 melompati H6
```

- Test Case 2

```
1
0 0 0 2 0 2 0 0
0 0 0 0 0 2 0 0
0 2 0 2 0 0 1 0
2 0 0 0 0 0 2 0
0 1 0 1 0 0 0 0
0 0 1 0 2 0 2 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 1 0
```

Gambar 3.4 Solusi Test Case 2 menggunakan algoritma Greedy

```
Solusi menggunakan algoritma Greedy:
0 0 0 2 0 2 0 0
0 0 0 0 0 2 0 0
0 2 0 2 0 0 1 0
2 0 0 0 0 0 2 0
0 1 0 1 0 0 0 0
0 0 1 0 2 0 2 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 1 0
Bidak 6 2 bisa melompati 1 bidak
Player 1 bergerak dari G6
Player 1 melompati E8
```

Gambar 3.5 Solusi Test Case 2 menggunakan algoritma Brute Force

IV. KESIMPULAN

Pada penelitian ini, algoritma Brute Force cenderung memiliki solusi dengan memakan bidak lawan terbanyak, algoritma ini akan mencari keseluruhan bidak yang dapat di gerakan. Oleh karena itu algoritma Brute Force akan memakan waktu lebih lama dibanding algoritma greedy, akan tetapi dikarenakan ruang lingkup permainan checkers yang tergolong rendah akan sulit membedakan perbandingan waktu antara algoritma greedy dan brute force.

Algoritma greedy cenderung dapat memposisikan bidak dengan letak yang strategis yaitu memposisikan bidak menjadi "King". Akan tetapi algoritma ini tidak optimal dikarenakan tidak bisa mencari solusi jika tidak ada bidak yang dapat menjadi king, alias fungsi heuristik nya nol.

Sehingga penulis menarik kesimpulan bahwa algoritma Brute Force dalam permainan *Checkers* cocok untuk segala situasi tetapi akan sulit untuk memposisikan *Man* menjadi *King*. Sedangkan Algoritma Greedy cocok untuk memposisikan *Man* menjadi *King* yang akan memiliki letak yang strategis

REFERENCES

[1] Salem Massachussetts. Introductory Rules for CHECKERS https://www.hasbro.com/common/instruct/Checkers.PDF (Accessed: 10 juni 2024)

[2] Munir, R. (2024). Algoritma Brute Force (Bagian 1). Accessed at : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf (Accessed: 12 Juni 2024)

[3] Munir, R. (2024). Algoritma Brute Force (Bagian 2). Accessed at : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag2.pdf (Accessed: 12 Juni 2024)

[4] Munir, R. (2024). Algoritma Greedt (Bagian 1). Accessed at : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf (Accessed: 12 Juni 2024)

[5] Munir, R. (2024). Algoritma Greedt (Bagian 2). Accessed at : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag2.pdf (Accessed: 12 Juni 2024)

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Juni 2024

Muhammad Dzaki Arta 13522149