Penerapan Algoritma Branch And Bound Pada Bot Catur

Jason Stanley Yoman - 13519019
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13519019@std.stei.itb.ac.id

***Abstract*—Catur merupakan permainan yang sudah ditemukan jauh di masa lalu. Catur melibatkan dua orang pemain yang menggerakkan bidak secara bergantian. Pada masa sekarang, sudah ditemukannya bot catur yang memungkinkan kita bermain tidak melawan orang lainnya namun dengan sebuah komputer. Makalah ini akan membahas bagaimana bot catur diimplementasikan secara sederhana dan bagaimana penerapan algoritma *branch and bound* dapat mempercepat proses pencarian.**

Keywords—catur; branch and bound; minimax; alpha beta pruning;

I. PENDAHULUAN

Catur adalah salah satu *strategy turn-based-game* yang dimainkan oleh dua orang pemain. Sejak kemunculan pertamanya catur telah mengalami perubahan aturan yang berakhir pada peraturan catur yang kita kenal saat ini. Cara memainkannya pun berubah. Dahulu, catur hanya dapat dimainkan oleh dua orang secara langsung di waktu dan tempat yang sama. Seiring dengan ditemukan dan berkembangnya teknologi komputer, manusia menemukan cara baru untuk memainkannya. Kini catur tidak hanya dapat dimainkan dengan orang lainnya, namun dapat juga dimainkan dengan sebuah *bot*. *Bot* catur adalah sebuah perangkat lunak yang dapat bermain bersama manusia. Dengan memberikan sebuah posisi catur, *bot* akan menentukan langkah selanjutnya.

Elemen permainan catur sendiri terdiri dari sebuah papan catur dan bidak-bidak. Papan catur berukuran 8x8. Bidak-bidak terdiri dari Raja, Ratu, Gajah, Kuda, Benteng, dan Pion. Pemain memainkan bidaknya secara bergantian.

Algoritma yang akan digunakan adalah algoritma *minimax* dengan *alpha beta pruning*. Algoritma minimax saja akan menghitung seluruh kemungkinan kombinasi gerakan. Namun, dengan optimasi *alpha beta pruning*, algoritma akan memangkas sebagian besar dari kemungkinan kombinasi gerakan.

II. LANDASAN TEORI

A. Catur

Catur adalah permainan yang dimainkan oleh dua orang. Antara satu pemain dengan yang lainnya dibedakan dengan dua buah warna yaitu putih dan hitam. Pemain yang memegang warna putih menggerakkan bidaknya terlebih dahulu di awal permainan. Papan catur terdiri dari 64 buah kotak dan memiliki dimensi 8x8. Warna dari kotak tersebut tersusun secara bergantian yang terdiri dari warna hitam dan putih. Setiap kotak memiliki tetangga (atas, bawah, kiri, dan kanan) yang berwarna warna yang berlawanan. Sedangkan tetangga diagonalnya memiliki warna yang sama.

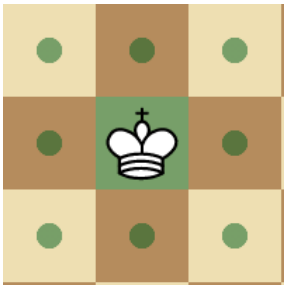
Permainan dimulai dengan pemain mendapatkan 16 buah bidak yang disusun berseberangan. Di setiap giliran, pemain menggerakkan bidaknya sesuai dengan aturan yang ada. Tujuan akhir dari permainan catur adalah melakukan skakmat ke lawan. Skakmat adalah keadaan dimana raja lawan diserang dan tidak ada cara lagi bagi lawan untuk menghentikan serangan tersebut.

Setiap kotak di papan catur memiliki nama untuk pemberian identitas pada setiap kotak. Sebuah nama tersusun atas sebuah huruf (A-G) diikuti dengan sebuah angka (1-8). Huruf menandakan urutan kolom dan angka menandakan urutan baris. Kolom paling kiri adalah Kolom A dan kolom paling kanan adalah kolom G. Baris teratas adalah baris 8 dan baris paling bawah adalah baris 1. Sistem penamaan ini diukur dari perspektif sisi putih.

Aturan pergerakan bidak catur adalah sebagai berikut :

1. Raja

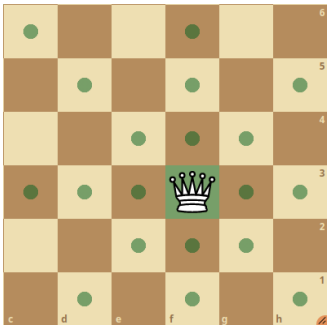
Raja dapat berpindah ke semua kotak yang bertetangga langsung dengan raja baik veritkal, horizontal maupun diagonal.



Gambar 1. Gerakan Raja

2. Ratu

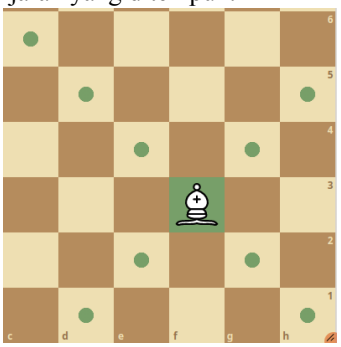
Ratu dapat bergerak ke segala arah dengan tidak ada batasan berapa jarak yang dapat ditempuh.



Gambar 2. Gerakan Ratu

3. Gajah

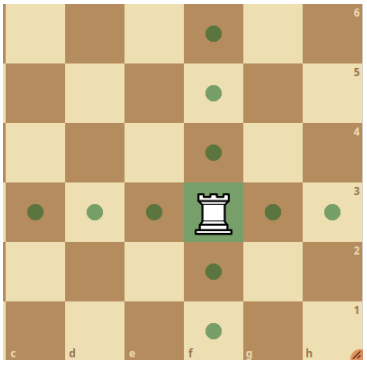
Gajah hanya dapat bergerak secara diagonal dengan tidak ada batasan jarak yang ditempuh.



Gambar 3. Gerakan Gajah

4. Benteng

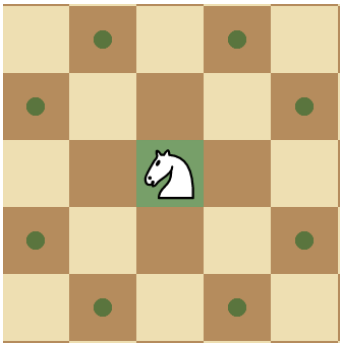
Gajah hanya dapat bergerak secara vertikal atau horizontal dengan tidak ada batasan jarak yang ditempuh.



Gambar 4. Gerakan Benteng

5. Kuda

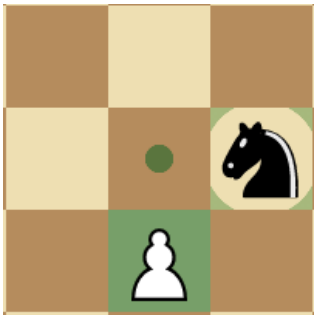
Kuda bergerak seperti huruf L serta kuda juga dapat melompati bidak lainnya.



Gambar 5. Gerakan Kuda

6. Pion

Pion hanya dapat bergerak maju. Pertama kali pion bergerak, pion dapat bergerak maju dua langkah atau 1 langkah. Namun, setelah langkah pertama dijalankan, pion hanya dapat bergerak maju satu langkah. Pion hanya dapat memakan bidak lain yang menempati diagonal depan dari pion tersebut.



Gambar 6. Gerakan Pion

B. Algoritma Branch and Bound

Algoritma *Branch and Bound* biasanya digunakan untuk persoalan optimasi dengan meminimalkan atau memaksimalkan suatu fungsi objektif. Algoritma ini juga menggunakan *Breadth First Search* untuk membangkitkan ruang solusi. Biasanya pada DFS, urutan pembangkitan simpul berdasarkan urutan pada sebuah antrian. Namun, pada *branch and bound*, setiap simpul diberikan sebuah *cost* dan urutan pembangkitan dimulai dari simpul yang memiliki cost paling sedikit.

Optimasi terjadi pada saat algoritma memproses simpul yang costnya sudah tidak dapat melebihi sebuah nilai yang sudah terbaik. Artinya, jika algoritma sudah menemukan sebuah gerakan bagus, maka tidak perlu lagi membangkitkan simpul yang nilainya akan lebih rendah dari gerakan itu. Pemangkasan simpul tersebut akan mengurangi jumlah pencarian yang akan dilakukan.

III. PEMBAHASAN

Implementasi akan dimulai dengan menentukan sebuah fungsi evaluasi untuk mengevaluasi posisi dari papan catur. Kemudian akan dijelaskan pencarian yang tidak efektif dan bagaimana penerapan *Branch and Bound* dapat membuat pencarian menjadi efektif.

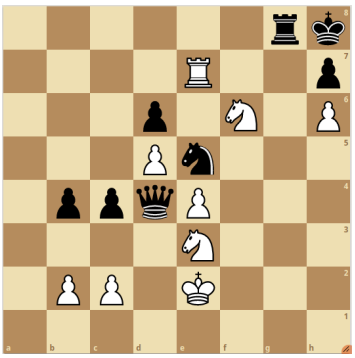
A. Evaluation Function

Evaluation Function adalah sebuah fungsi yang menerima posisi dari papan catur dan mengembalikan sebuah angka yang merepresentasikan nilai dari posisi tersebut. *Evaluation Function* juga dapat digunakan sebagai nilai heuristik pada algoritma *branch and bound*. Terdapat berbagai macam *Evaluation Function* yang sudah dikembangkan saat ini. Salah satunya yang sederhana adalah dengan menjumlahkan seluruh bidak yang masih berada di papan. Masing-masing bidak memiliki nilai tergantung dari seberapa berharganya bidak tersebut. Misalnya sebuah ratu tentunya memiliki nilai yang lebih tinggi dari pion. Nilai-nilai tersebut adalah :

Bidak	**Nilai**
Pion	100
Kuda	350
Gajah	350
Benteng	525
Ratu	1000

Tabel 1. Tabel nilai bidak

Raja bernilai tak terhingga dimana jika raja dimakan pada saat evaluasi posisi, maka posisi tersebut akan dianggap kalah karena pemain tersebut kehilangan poin yang sangat besar.



Gambar 7. Contoh Posisi

Sebagai contoh, untuk mengevaluasi posisi putih, jumlahkan seluruh poin dari bidak yang dimiliki putih. Dalam posisi tersebut, putih memiliki nilai evaluasi sebesar :

$$f(white) = 5 * v(pion) + 2 * v(kuda) + v(benteng)$$

$$f(white) = 1725$$

Dengan cara yang sama, nilai evaluasi hitam adalah :

$$f(black) = 2275$$

Fungsi evaluasi yang dijelaskan pada bagian ini merupakan fungsi evaluasi yang paling sederhana dan tentunya tidak memberikan *insight* yang akurat mengenai posisi. Seperti contoh evaluasi diatas, hitam memiliki keunggulan menurut nilai yang dihasilkan, namun apakah posisi hitam lebih baik daripada putih?. Jawabannya adalah tidak karena putih berpotensi untuk menskakmat hitam dalam satu langkah. Terdapat berbagai hal yang perlu di konsiderasi lagi ketika merancang fungsi evaluasi seperti struktur pion, keamanan raja, kotak yang dikontrol, fase game (*opening, middle game,* dan *endgame*), dll.

Untuk kasus pencarian menggunakan algoritma *minimax,* hanya diperlukan sebuah nilai untuk fungsi heuristiknya. Oleh karena itu, nilai heuristik merupakan selisih antara evaluasi putih dengan hitam.

$$h(n) = f(white) - f(black)$$

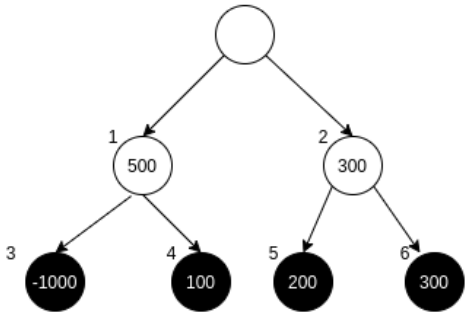
Nilai h(n) > 0 menandakan putih memiliki posisi yang lebih unggul daripada hitam dan sebaliknya jika h(n) < 0, maka hitam memiliki posisi yang lebih unggul.

B. Minimax

Misalkan akan dicari langkah terbaik bagi putih dengan kedalaman 1 buah langkah. Penyelesaian untuk masalah ini adalah dengan mengiterasi seluruh gerakan yang mungkin dan mengambil langkah yang memberikan nilai fungsi evaluasi yang paling tinggi untuk posisi itu. Sedangkan jika sekarang adalah giliran hitam, akan dicari langkah yang memiliki nilai yang paling kecil karena sesuai dengan fungsi evaluasi yang telah dijelaskan sebelumnya, nilai yang paling dekat dengan negatif *infinity* merupakan posisi yang paling bagus untuk hitam.

Akan tetapi, bagaimana jika ingin mencari dengan kedalaman 2. Pencarian tentu tidak bisa menggunakan cara yang diatas karena untuk langkah setelah putih memilih langkah, hitam juga dapat memilih langkah yang terbaik untuk dia. Terdapat kasus dimana pilihan terbaik bagi putih dapat berujung pada kondisi yang lebih baik bagi hitam di langkah selanjutnya.

Gambar 9. Pencarian Kedalaman 2 “Pintar”

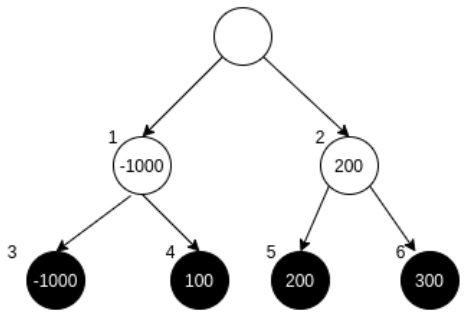


Gambar 8. Pencarian Kedalaman 2 “Tidak Pintar”

Misalnya pada gambar diatas, dengan algoritma sebelumnya, putih pasti langsung memilih langkah pada simpul 1 karena 500 > 300. Namun, setelah itu, hitam akan memilih langkah pada simpul 3 dan keadaan pun berbalik menjadi hitam yang lebih unggul.

Oleh karena itu, solusi dari masalah diatas adalah dengan algoritma *minimax*. Tidak seperti contoh di atas, *minimax* hanya akan menggunakan fungsi evaluasi pada saat simpul merupakan simpul terdalam atau posisi sudah skakmat. Nilai lainnya akan diisi tergantung dengan giliran pemain. Jika giliran pada simpul tersebut adalah hitam, maka nilai dari simpul itu adalah nilai yang terkecil dari anak-anaknya. Sedangkan jika giliran simpul tersebut adalah putih, maka nilainya adalah nilai terbesar dari anak-anaknya. Dengan begitu, *minimax* akan mencari dengan tambahan konsiderasi langkah berikutnya.

Berikut adalah pohon evaluasi dengan menggunakan *minimax*:



Pada visualisasi diatas, putih akan mengambil langkah pada simpul ke-2. Oleh karena itu, walaupun hitam memilih langkah terbaik baginya, posisi akan tetap unggul bagi putih.

```
int minimax (Position position, int depth, bool isWhite) {
  if (depth == 0 || Game.isOver()) {
    return EvaluationFunction(position);
  }
  bool nextPlayer = !isWhite;
  Position[] positions = position.generateNextPosition();
  if (isWhite) {
    int maxEvaluation = int.MinValue;
    foreach(Position pos in positions) {
      int eval = minimax(pos, depth - 1, nextPlayer);
      if (eval > maxEvaluation) {
        maxEvaluation = eval;
      }
    }
    return maxEvaluation;
  } else {
    int minEvaluation = int.MaxValue;
    foreach(Position pos in positions) {
      int eval = minimax(pos, depth - 1, nextPlayer);
      minEvaluation = Math.Min(minEvaluation, eval);
      if (eval < minEvaluation) {
        minEvaluation = eval;
        minMove = move;
      }
    }
    return minEvaluation;
  }
}
```

Gambar 10. Source Code Minimax

```
string fenNotation =
"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1";
Position startPosition = new Position(fenNotation);
int bestMove = minimax(startPosition, 3, true);
```

Gambar 11. Pemanggilan Minimax

Contoh diatas adalah contoh pemanggilan *minimax* dengan posisi awal catur yang direpresentasikan dengan notasi FEN.

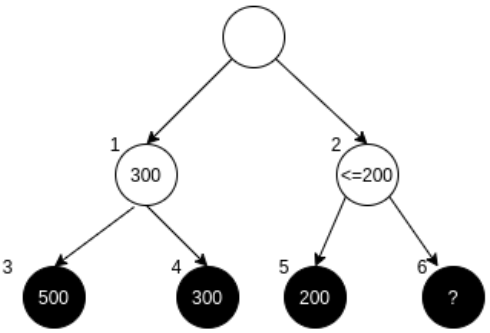
Pencarian seperti itu merupakan tipe pencarian *brute force*. Algoritma akan mencari seluruh kemungkinan yang ada tanpa ada optimasi apapun dan algoritma mencari dengan “tidak pintar”. Hal ini tentunya menjadi sebuah masalah seiring bertambahnya kedalaman pencarian. Di dalam pembukaan catur sendiri, total kombinasi gerakan pada 10 langkah pertama bisa mencapai 69,352,859,712,417 kemungkinan. Sedangkan *chess engine* yang paling populer sekarang bisa mencari sampai kedalaman 20 keatas tanpa memakan waktu yang sangat lama. Oleh karena itu, diperlukan suatu cara untuk mengoptimalkan kerja algoritma tersebut

Langkah	Total kombinasi
1	20
2	400
3	8,902
4	197,281
5	4,865,609
6	119,060,324
7	3,195,901,860
8	84,998,978,956
9	2,439,530,234,167
10	69,352,859,712,417

Tabel 2. Perbandingan Langkah dan Total Kombinasi

C. *Alpha Beta Pruning*

Alpha Beta Pruning adalah algoritma untuk mengoptimasi *minimax* dengan konsep *branch and bound*. Algoritma ini menghilangkan jumlah pencarian yang dilakukan. Ide dari optimasi ini adalah, ketika sudah ditemukannya gerakan yang bagus, ia akan mengeliminasi cabang lainnya jika terdapat satu saja gerakan yang tidak dapat melebihi nilai dari gerakan bagus. Algoritma menggunakan 2 nilai, yaitu *alpha* dan *beta*. *Alpha* merepresentasikan nilai minimum yang didapatkan oleh putih sedangkan *beta* merepresentasikan nilai maksimal yang didapatkan hitam.



Gambar 12. Ilustrasi *Alpha Beta Pruning*

Pada contoh diatas, untuk mengevaluasi simpul 2, maka akan dilakukan evaluasi pada simpul 5 dan 6. Ketika mengevaluasi simpul 5, didapatkan nilai 200. Pertanyaannya adalah apakah kita perlu mengevaluasi simpul 6??. Jika simpul 6 memiliki nilai dibawah 200, maka nilai di simpul 6 menjadi nilai evaluasi dari simpul 2. Dan jika simpul 6 memiliki nilai diatas 200, maka 200 merupakan nilai evaluasi dari simpul 2. Dengan analisa itu, kita dapat menyimpulkan nilai dari simpul 2 lebih kecil sama dengan 200. Dengan begitu, tentunya putih akan memilih langkah pada simpul 1 karena 300 pasti lebih besar dari nilai yang lebih kecil atau sama dengan 200. Oleh karena itu, simpul 6 tidak usah dihitung lagi karena apapun nilainya, yang dipilih adalah simpul 1.

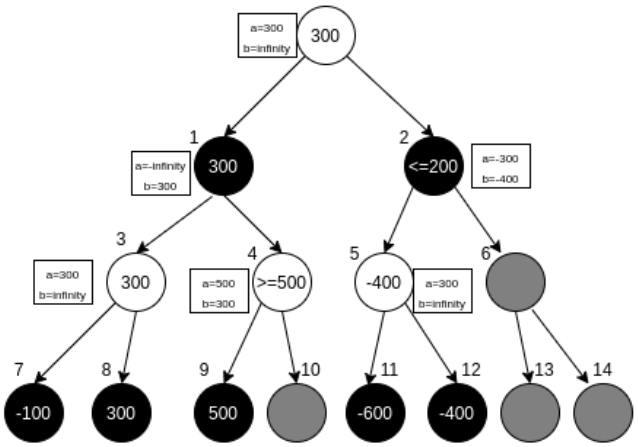
```
int minimax (Position position, int depth, int alpha, int beta, bool
 isWhite) {
      if (depth == 0 || Game.isOver()) {
        return EvaluationFunction(position);
      }
      bool nextPlayer = !isWhite;
      Position[] positions = position.generateNextPosition();
      if (isWhite) {
        int maxEvaluation = int.MinValue;
        foreach(Position pos in positions) {
          int eval = minimax(pos, depth - 1, alpha, beta, nextPlayer);
          if (eval > maxEvaluation) {
            maxEvaluation = eval;
          }
          alpha = Math.Max(alpha, eval);
          if (beta <= alpha) {
            break;
          }
        }
        return maxEvaluation;
      } else {
        int minEvaluation = int.MaxValue;
        foreach(Position pos in positions) {
          int eval = minimax(pos, depth - 1, alpha, beta, nextPlayer);
          minEvaluation = Math.Min(minEvaluation, eval);
          if (eval < minEvaluation) {
            minEvaluation = eval;
            minMove = move;
          }
          beta = Math.Min(beta, eval);
          if (beta <= alpha) {
            break;
          }
        }
        return minEvaluation;
      }
    }
```

Gambar 13. Source Code Minimax dengan *Alpha Beta Pruning*

```
string fenNotation =
"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1";
Position startPosition = new Position(fenNotation);
int bestMove = minimax(startPosition, 3, int.MinValue, int.MaxValue,
true);
```

Gambar 14. Pemanggilan Minimax dengan *Alpha Beta Pruning*

Berikut contoh ilustrasi dari menggunakan algoritma *Alpha Beta Pruning*



Gambar 15. Ilustrasi *Alpha Beta Pruning*

Pada saat mengevaluasi simpul 4, nilai beta sudah lebih kecil dari alpha. Ini menunjukkan ada gerakan lain yang menguntungkan bagi hitam (Gerakan tersebut adalah gerakan pada simpul 3). Sehingga, simpul tersebut tidak akan dipilih lagi karena hitam pasti akan memilih gerakan yang lainnya. Hal ini juga didukung dengan nilai alpha yang akan meningkat sehingga ekspresi beta <= alpha sudah tidak mungkin lagi bernilai *false*.

Pada saat mengevaluasi simpul 6, nilai beta sudah lebih kecil dari alpha. Ini menunjukkan ada gerakan lain yang menguntungkan bagi putih (Gerakan tersebut adalah gerakan pada simpul 1). Sehingga simpul tersebut tidak akan dipilih lagi karena putih pasti akan memilih gerakan yang lainnya. Hal ini juga didukung dengan nilai beta yang akan menurun sehingga ekspresi beta <= alpha sudah tidak mungkin lagi bernilai false.

Banyaknya simpul yang dipotong bergantung pada urutan gerakan yang dibangkitkan dan ini juga berpengaruh pada kecepatan dari algoritma. Skenario terbaik dari *alpha beta pruning* adalah gerakan terbaik didapatkan pada iterasi pertama sehingga gerakan lainnya dapat dieliminasi. Sedangkan skenario terburuk dari algoritma ini adalah ketika gerakan terbaik ditemukan pada iterasi terakhir sehingga tidak ada gerakan yang dieliminasi. Untuk menghasilkan urutan gerakan yang dapat memaksimalkan *pruning,* merupakan topik lainnya yang disebut *move ordering* dan topik tersebut tidak dibahas di makalah ini.

IV. KESIMPULAN

Prinsip *branch and bound* dapat digunakan untuk mengoptimasi pencarian gerakan terbaik dari sebuah posisi dalam catur. Cara kerjanya adalah dengan mengeliminasi simpul yang tidak mencapai sebuah fungsi kondisi layak. Algoritma tersebut dinamakan *Alpha Beta Pruning*. Masih banyak ruang pengembangan untuk bot yang sederhana ini dimulai dari fungsi evaluasi yang lebih menggambarkan kondisi catur yang sebenarnya, *pruning* yang lebih efektif, dan *move ordering*.

V. UCAPAN TERIMAKASIH

Penulis mengucapkan syukur kepada Allah yang Maha Esa karena berkat RahmatNya-lah penulis dapat menyelesaikan makalah ini dengan baik. Penulis juga ingin mengucapkan terima kasih kepada semua pihak yang telah membantu penulis dalam membuat makalah berjudul "Penerapan algoritma branch and bound pada bot catur". Penulis juga berterimakasih kepada Ir. Rila Mandala, M.Eng.,Ph.D. selaku pembimbing dan dosen mata kuliah IF2211 / Strategi Algoritma.

REFERENCES

[1] https://www.fide.com/FIDE/handbook/LawsOfChess.pdf. Diakses pada tanggal 10 Mei 2021 pukul 21.17

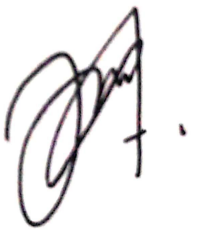
[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Branch-and-Bound-2021-Bagian1.pdf. Diakses pada tanggal 10 Mei 2021 pukul 14.00

[3] https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/. Diakses pada tanggal 10 Mei 2021 pukul 15.00

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, 10 Mei 2021



Jason Stanley Yoman
13519019