Analisis Perbandingan Algoritma Sekuensial dan Paralel Dengan Mengimplementasikan Pohon n-Ary Pada Sebuah Game Tree

Ivan Fadillah 13516128
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13516128@std.stei.itb.ac.id

Abstract*—Game industry adalah salah satu entertainment industry yang banyak sekali melibatkan peran scientist di bidang matematika dan informatika. Salah satu penerapannya adalah dalam pembuatan game komputer berbasis single player. Dalam sejarahnya, dahulu komputer tidak bisa memenangkan manusia dalam memainkan sebuah game karena kelemahan agoritma permainan untuk menemukan langkah terbaik agar dapat memenangkan sebuah game atau penyebab lainnya adalah keterbatasan komputasi komputer dan ruang memori. Namun di era sekarang ini kemajuan di bidang teknologi dan bidang arsitektur komputer akhirnya memungkinkan komputer untuk memenangkan game melawan manusia dalam sebuah permainan yang paling kompleks. Salah satunya adalah catur, tic tac toe dan lain sebagainya dalam makalah ini akan di sajikan perbandingan algoritma dan koompleksitasnya dalam membangun sebuah game tree.

Tujuan makalah ini adalah untuk membahas, membandingkan dan menganalisis berbagai algoritma sekuensial dan paralel dari game tree, termasuk beberapa pengembangan yang dapat dilakukan untuk meningkatkan kualitas program dalam membangun sebuah game dengan konsep pohon informatika.

***Keywords*— tree, game, minimax, tic tac toe, alghorithm, game tree, game industry, catur,kompleksitas, single player**

I. PENDAHULUAN

Game single player (manusia versus komputer) memiliki sejarah yang cukup panjang. Dalam program bermain game abad sebelumnya, komputer tidak bisa memenangkan manusia karena kelemahan agoritma permainan untuk menemukan langkah terbaik agar dapat memenangkan sebuah game atau penyebab lainnya adalah keterbatasan komputasi komputer dan ruang memori. Namun di era sekarang ini kemajuan di bidang teknologi dan bidang arsitektur komputer akhirnya memungkinkan komputer untuk memenangkan game melawan manusia dalam sebuah permainan yang paling kompleks, salah satunya adalah catur. Banyak algoritma yang telah ditemukan untuk menemukan langkah terbaik untuk komputer, termasuk algoritma sekuensial seperti MiniMax [1], NegaMax [2], Negascout [3], SSS * [4] dan B * [5] . sebagai algoritma paralel seperti halnya algoritma Parallel Alpha-Beta [6]. Algoritma Paralel ini sekarang dimodifikasi untuk dijalankan tidak hanya pada CPU, tapi juga pada GPU [7] untuk memberikan solusi yang lebih cepat.

Hampir semua program *game* menggunakan metode n-ary tree untuk menemukan langkah terbaik berikutnya. Contohnya pohon untuk game Tic-TacToe [8] ditunjukkan pada Gambar 1 Angka tersebut menunjukkan hal berikut:

- Setiap node mewakili keadaan permainan.
- Akar mewakili keadaan permainan saat ini.
- Semua cabang untuk node tertentu mewakili semua jalur hukum untuk node tersebut.
- Simpul yang tidak memiliki penggantinya disebut daun.

Fungsi evaluation digunakan untuk menentukan berapa pun daun yang mewakili menang, kalah, imbang atau hanya sebuah skor. Algoritma dihentikan sebelum pemain manapun menang, kalah atau permainan berakhir dengan hasil imbang. Para *developers* biasanya melakukannya karena dalam permainan yang lebih kompleks, tidak ada algoritma praktis yang bisa mencari di seluruh pohon dalam jumlah waktu efisien meski menggunakan *power* pemrosesan paralel. Contohnya adalah permainan catur, dimana *game* ini program perlu mengevaluasi sekitar 10^{20} dan 10^{40} node masing-masing. W^D digunakan sebagai estimasi jumlah node yang perlu dikunjungi, dimana W mewakili rata-rata jumlah pergerakan legal untuk setiap node, dan D mewakili panjang permainan. Dua solusi untuk masalah ini adalah menggunakan kedalaman tetap "D" atau menggunakan waktu tertentu untuk berhenti membangun pohon.

Ada banyak metode kategorisasi untuk pohon permainan sekuensial. Namun, kategorisasi yang paling umum didasarkan pada depth-first dan breadth-first, yang digunakan dalam makalah ini. Depth-First Search "DFS" [9] yaitu algoritma akan dimulai dari akar dan dieksplorasi asalkan batasan kedalaman tidak memenuhi setiap cabang sebelum melakukan backtracking. Breadth-first Search "BFS" [10] yaitu berawal dari akar dan memeriksa semua anak dari simpul akar sebelum memeriksa semua anak dari setiap simpul anak-anak akar. Algoritma paralel sulit untuk dikategorikan sebagai kedalaman atau keluasan pertama karena beberapa algoritma paralel bekerja sebagai berikut: setiap inti atau setiap prosesor memeriksa anak dari akar menggunakan DFS atau BFS. Pada

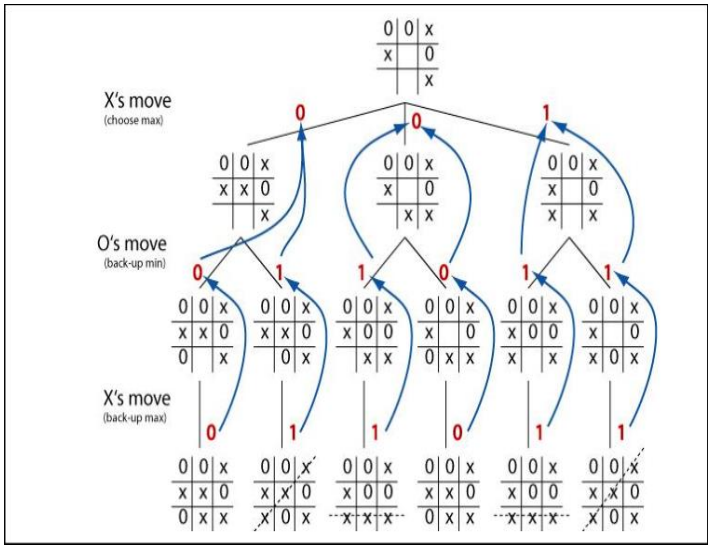
kasus pertama, distribusi anak-anak akar menggunakan algoritma BFS, namun masing-masing inti atau prosesor menggunakan DFS. Dalam hal ini, disebut sistem hybrid.

Untuk memperjelas pembaca, makalah ini disusun sebagai berikut:

Bagian [II], menyajikan pembahasan untuk algoritma pohon permainan sekuensial yang dikategorikan ke dalam algoritma depth-first & breadth-first algorithms. Bagian [III], menyajikan sebuah diskusi untuk algoritma tree game paralel dari sudut pandang pemrograman dan dari sudut pandang perangkat keras. Bagian, [IV], memberikan analisis algoritma depth dan algoritma breadth berdasarkan kompleksitas algoritma untuk waktu dan memori. Bagian [V], berisi kesimpulan dan makalah ini dan juga saran yang dapat meningkatkan atau mengembangkan ide dari algoritma pada makalah ini.

II. ALGORITMA SEQUENTIAL GAME TREE

Seperti yang disebutkan di bagian sebelumnya, algoritma sekuensial dikategorikan ke pencarian depth-first search [9] dan breadth-first search [10]. Selanjutnya, algoritma depth-first search dikategorikan kembali menjadi *searching* secara brute force dan *searching* secara selektif [11]. *searching* secara bruteforce mencari setiap variasi pada kedalaman tertentu, sementara *searching* secara selektif hanya melihat cabang-cabang penting. Bagian [A], menyajikan pembahasan untuk *searching* secara bruteforce pada depth-first search. Bagian [B], menyajikan pembahasan untuk *searching* secara selektif pada depth-first search. Bagian [C], menyajikan pembahasan untuk breadth-first search.



Gambar 1. Pohon permainan untuk game Tic-Tac-Toe menggunakan algoritma MiniMax.

A. Algoritma Brute-Force pada Depth-First Search

Algoritma yang paling terkenal dalam pencarian brute force adalah MiniMax [1], NegaMax [2], Alpha-Beta [12], NegaScout [3], dan Principle-Variation [13] . Berikut adalah deskripsi masing-masing algoritma ini.

Algoritma MiniMax [1] adalah algoritma tree tree yang terbagi menjadi dua tahap logis, yang pertama untuk pemain pertama yaitu komputer dan yang kedua untuk pemain kedua yaitu manusia. Algoritma mencoba untuk menemukan langkah hukum terbaik untuk komputer bahkan jika manusia memainkan langkah terbaik untuknya. Artinya, ia memaksimalkan nilai komputer saat memilih komputer bergerak, sambil meminimalkan skor itu dengan memilih langkah hukum terbaik bagi manusia saat ia memilih langkah manusia.

Pada Gambar 1 ada simulasi pencarian MiniMax untuk game TicTac-Toe [8].Setiap node memiliki nilai yang dihitung dengan fungsi evaluasi. Untuk jalur tertentu, nilai simpul daun kembali ke induknya. Artinya nilai dari setiap langkah O akan selalu memilih nilai minimum untuk komputer, sedangkan nilai untuk setiap langkah X akan selalu memilih nilai maksimum komputer.

Perhatikan Gambar 2 kode pseudo untuk algoritma MiniMax [1]. Program ini pertama kali memanggil fungsi MiniMax yang melakukan pemanggilan untuk MaxMove dan MinMove. Setiap kali fungsi MaxMove atau fungsi MinMove dipanggil secara otomatis memanggil fungsi lainnya sampai permainan berakhir, atau mencapai kedalaman yang diinginkan.

Algoritma NegaMax [2] adalah algoritma yang hampir sama dengan MiniMax. Yang membedakannya hanya menggunakan fungsi maksimisasi daripada menggunakan fungsi maksimalisasi dan minimisasi. Hal ini bisa dilakukan dengan meniadakan nilai yang dikembalikan dari anak-anak dari sudut pandang lawan daripada mencari nilai minimum. Hal ini dimungkinkan karena hubungan matematis berikut:

Max (a, b) == -Min (-a, -b) (1)

```
MinMax (GamePosition game) {
        return MaxMove (game);
}

MaxMove (GamePosition game) {
        if (GameEnded(game)) {
                return EvalGameState(game);
        }  else {
                best_move < - {};
                moves <- GenerateMoves(game);
                ForEach moves {
                   move <- MinMove(ApplyMove(game));
                   if (Value(move) > Value(best_move)) {
                         best_move < - move;
                   }
                }
               return best_move;
               }
}

MinMove (GamePosition game) {
      best_move <- {};
       moves <- GenerateMoves(game);
      ForEach moves {
                move <- MaxMove(ApplyMove(game));
                if (Value(move) > Value(best_move)) {
                      best_move < - move;
                      }
        }
return best_move;
```

Gambar 2. Algoritma MiniMax Kode Pseduo

Pada Gambar 3 adalah kode pseudo untuk algoritma NegaMax. Jelas, persamaan (1) digunakan untuk menyederhanakan algoritma MiniMax.

Algoritma Alpha-Beta [12] adalah modifikasi yang bisa diterapkan pada algoritma MiniMax atau NegaMax. Kunth dan Moore membuktikan bahwa banyak cabang dapat dihilangkan dari pohon permainan yang dapat mengurangi waktu untuk membangun sebuah pohon, dan akan memberikan hasil yang sama seperti MiniMax atau NegaMax. Gagasan utama dari algoritma ini adalah memotong cabang yang tidak berpengaruh di pohon permainan. Misalnya: Max (8, Min (5, X)) dan Min (3, Max (7, Y)). Hasilnya selalu 8 pada contoh pertama dan 3 pada contoh kedua, tidak peduli nilai dari X atau Y. Ini berarti algoritma dapat memotong node X atau Y dengan cabang-cabangnya. Algoritma Alpha-Beta menggunakan dua variabel (alpha & beta) untuk mendeteksi kasus-kasus ini, jadi nilai apapun kurang dari alpha atau lebih besar dari beta akan otomatis terpotong tanpa mempengaruhi hasil pencarian pohon.

Versi yang disempurnakan dari algoritma NegaMax dari Gambar 3 dengan properti Alpha-Beta ditunjukkan pada gambar 4.

```
// Search game tree to given depth, and return evaluation
of
// root node.
int NegaMax(gamePosition, depth) {
    if (depth=0 || game is over)
     // evaluate leaf gamePositionition from
    // current player's standpoint
        return Eval (gamePosition);
     // present return value
    score = - INFINITY;
     // generate successor moves
    moves = Generate(gamePosition);
    // look over all moves
    for i =1 to sizeof(moves) do
    {
          // execute current move
          Make(moves[i]);
          // call other player, and switch sign of
          // returned value
          cur = -NegaMax(gamePosition, depth-1);
          // compare returned value and score
          // value, update it if necessary
          if (cur > score)
                score = cur;
                // retract current move   Undo(moves[i]);
    }
    return score;
}
```

Gambar 3. Algoritma NegaMax Kode Pseudo

Beberapa perangkat tambahan untuk algoritma Alpha-Beta, di antaranya sebagai berikut:

- Move Ordering: Kecepatan dan jumlah cutoff dari algoritma Alpha-Beta dapat berubah secara dramatis tergantung pada urutan proses pencariannya. Langkah terbaik harus diperiksa dahulu, lalu langkah terbaik kedua dan seterusnya. Ini akan memaksimalkan keefektifan algoritma. Banyak teknik yang dikembangkan untuk mengatasi masalah ini, diantaranya:
    - Iterative deepening.
    - Tabel transposisi.
    - Killer Move Heuristic.
    - Sejarah Heuristik.
- Pencarian Jendela Minimal: Algoritma Alpha-Beta bergantung pada nilai alpha dan beta untuk memotong cabang, jadi dengan mempersempit jendela pencarian dengan mengubah nilai alpha dan beta, itu akan meningkatkan kemungkinan cut offs. Banyak algoritma seperti NegaScout [3] dan MTD (f) [14] menggunakan properti ini untuk mengembangkan algoritma Alpha-Beta yang akan dibahas di bawah ini.

```
// Search game tree to given depth, and return
evaluation of
 // root node.
int AlphaBeta(gamePosition, depth, alpha,
beta)
{
    if (depth=0 || game is over)
    // evaluate leaf gamePositionition from
    // current player's standpoint
          return Eval (gamePosition);
          // present return value
    score = - INFINITY;
    // generate successor moves
    moves = Generate(gamePosition);
    // look over all moves
    for i =1 to sizeof(moves) do
    {
          // execute current move
          Make(moves[i]);
          // call other player, and switch sign of
          // returned value
          cur = -AlphaBeta(gamePosition,
          depth-1,                    -beta, -
          alpha);
           // compare returned value and score
          // value, update it if necessary
          if (cur > score) score = cur;
          // adjust the search window
           if (score > alpha) alpha = score;
          // retract current move
          Undo(moves[i]);
          // cut off
          if (alpha >= beta) return alpha;
    }
    return score;
}
```

Gambar 4. NegaMax yang Disempurnakan dengan Kode Pseudo Beta-Beta Property

Algoritma NegScout [3] dan Principal Variation Search [13] didasarkan pada algoritma *scout* yang merupakan versi algoritma Alfa-Beta yang lebih baik yang dapat membuat lebih banyak cutoff di pohon game. Ini berisi kondisi tes baru yang memeriksa apa pun simpul pertama di saudara kandungnya kurang dari atau sama dengan nilai beta atau lebih besar dari atau sama dengan nilai alfa. Jika hasil dari kondisi tersebut benar, maka algoritma tersebut memotong akar cabang untuk saudara-saudara ini, dan jika salah, maka pencarian sisanya dari saudara kandung mendapatkan nilai alpha dan beta yang baru.

Pada Gambar 5 ada kode pseudo untuk NegaScout [3]. Algoritma yang sama pada Gambar 4 dengan modifikasi pencarian jendela minimal.

B. Selektivitas algoritma di Depth-First Search

Perbedaan utama antara algoritma brute force dan algoritma selektivitas yaitu tidak tergantung pada kedalaman Gambar 5.

```
// Search game tree to given depth, and return evaluation of
// root node.
int NegaScout(gamePosition, depth, alpha, beta)
{
    if (depth=0 || game is over)
    // evaluate leaf gamePositionition from
    // current player's standpoint
          return Eval (gamePosition);
    // present return value
    score = - INFINITY;
     n = beta;
     // generate successor moves
    moves = Generate(gamePosition);
    // look over all moves
    for i =1 to sizeof(moves) do
    {
          // execute current move
          Make(moves[i]);
          // call other player, and switch sign of
          // returned value
          cur = -NegaScout(gamePosition, depth-1,
          -n, -alpha);
          if (cur > score)
          {
            if (n = beta ) OR (d <= 2)
           // compare returned value and
           // score value, update it if
           // necessary
                score = cur;
           else score = -NegaScout (gamePosition,depth-1,
          -beta, -cur);
          }
          // adjust the search window
          if (score > alpha) alpha = score;
          // retract current move   Undo(moves[i]);
          // cut off
          if (alpha >= beta) return alpha;
          n = alpha+1;
    }
     return score;
}
```

Gambar 5. Algoritma NegaScout Kode Pseudo Menggunakan Prinsip Pencarian Jendela Minimal

tetap untuk berhenti mencari di setiap cabang. Teknik yang paling umum dalam kategori ini adalah Quiescence Search and Forward Pruning.

Berikut ini adalah pembahasan tentang penerapan algoritma Quiescence Search [15] dan ProbCut [16] yang didasarkan pada teknik Forward Pruning.

Quiescence Search [15] berdasarkan gagasan depth searching variabel. Algoritma mengikuti kedalaman normal di kebanyakan cabang. Namun, di beberapa cabang algoritma mengambil tampilan yang lebih dalam dan meningkatkan kedalaman pencarian. Contohnya adalah permainan catur dimana dalam pergerakan kritis seperti cek atau promosi,

algoritma memperluas kedalaman untuk memastikan tidak ada ancaman.

Pada Gambar 6 terdapat kode pseudo abstrak untuk Pencarian Kedirgantaraan yang memperluas kedalaman dan memeriksa apakah ada tangkapan untuk potongan setelah pergerakan tertentu atau tidak.

```
int Quiesce( int alpha, int beta )
{
    int stand_pat = Evaluate();
    if( stand_pat >= beta )
        return beta;
    if( alpha < stand_pat )
        alpha = stand_pat;
    until( every_capture_has_been_examined )  {
        MakeCapture();
        score = -Quiesce( -beta, -alpha );
        TakeBackMove();
        if( score >= beta )
            return beta;
        if( score > alpha )
            alpha = score;
      }
     return alpha;
}
```

Gambar 6. Abstrak Versi Pseudo Code for Quiescence Search

Teknik pemutusan melengkapi gagasan tentang kedalaman variabel, di mana ia memotong cabang yang tidak menjanjikan. Namun, hal ini dapat menyebabkan kesalahan pada hasilnya. Banyak algoritma menerapkan gagasan teknik ini, termasuk N-Best Selective Search, ProbCut dan Multi-ProCut [16].

Pencarian Selektif N hanya mencari gerakan N-terbaik terbaik di setiap simpul. Semua saudara lainnya untuk gerakan N-terbaik akan otomatis terputus.

Kedua algoritma ProbCut Multi-ProCut menggunakan hasil *searching* untuk menentukan kemungkinan pencarian yang lebih dalam akan mengubah nilai alpha dan beta atau tidak.

Algoritma ProbCut [16] menggunakan teknik korelasi statistik untuk memotong cabang, karena ditemukan adanya korelasi kuat antara nilai yang diperoleh dari kedalaman yang berbeda. Relasi tersebut dijelaskan oleh Micheal Buro sebagai berikut: V_D = a * V_D '+ b + e (2)

Dimana V_D berarti nilai kedalaman yang diberikan, a & b adalah bilangan real dan e adalah kesalahan terdistribusi normal dengan mean nol.

Karena nilai ≈ 1, b ≈ 0 dan σ2 kecil dalam fungsi evaluasi yang paling stabil, probabilitas V_D> = beta dapat diprediksi dari persamaan ekuivalen berikut:

V_D '> = ((1 / Φ (P)) * σ + beta-b) / a (3)

Selanjutnya, probabilitas V_D <= alpha dapat diprediksi dari persamaan ekuivalen berikut:

V_D '<= (- (1 / Φ (P) * σ + alpha -b) / a (4)

Pada Gambar 7 ada implementasi abstrak untuk algoritma ProbCut. Ingat terserah Anda untuk memilih nilai D, D ', batas cutoff (1 / Φ (P)), a, b dan σ. Algoritma ini dapat memberikan hasil yang lebih cepat daripada algoritma brute force. Namun, dibutuhkan banyak parameter yang akurat, yang mungkin sulit untuk dipilih dan dapat menyebabkan kesalahan pada hasilnya.

```
int alphaBetaProbCut(int α, int β, int depth)
{
    const float T(1.5);
    const int DP(4);
    const int D(8);
    if ( depth == 0 )
        return evaluate();
    if ( depth == D ) {
        int bound;
        // v >= β with prob. of at least p?
        // yes => cutoff */
        bound = round( ( T * σ + β - b) / a );
        if ( alphaBetaProbCut( bound-1, bound,
          DP) >= bound )
          return β;
          // v <= α with prob. of at least p?
          // yes => cutoff */
          bound = round( (-T * σ + α - b) / a );
          if ( alphaBetaProbCut( bound, bound+1,
        DP) <= bound )
          return α;
    }
          // the remainder of alpha-beta goes here     ...
}
```

Gambar 7. Abstrak Versi Pseudo Code for ProbCut Search tanpa implementasi alphabeth

Algoritma Multi-ProbCut [16] menggeneralisasi gagasan ProbCut dengan menggunakan batas dan pemeriksaan cutoff tambahan, termasuk memungkinkan lebih banyak parameter regresi dan batas cutoff, menggunakan banyak pasangan kedalaman dan menggunakan pendalaman iteratif internal untuk pencarian dangkal.

C. Breadth-First Search

Seperti yang disebutkan sebelumnya, BFS dimulai dari simpul akar kemudian mengunjungi anak pertamanya setelah itu mengunjungi semua saudaranya dari kedalaman yang sama sebelum bergerak ke kedalaman berikutnya. Salah satu masalah dengan teknik ini; Ini membutuhkan memori yang besar untuk menyimpan data simpul. Banyak algoritma menggunakan teknik ini seperti algoritme algoritma NegaC * [17], MTD (f) [14], SSS * [4], B * [5] dan Monte-Carlo [18] yang dibahas di

bawah ini. Algoritma NegaC * [17] menggunakan jendela minimal dengan algoritma Alpha-Beta failsoft seperti algoritma NegaScout [3], namun mengurai pohon dengan cara Breadth-First. Teknik gagal-lembut menggunakan dua variabel lebih banyak daripada algoritma Alpha-Beta untuk memotong lebih banyak cabang.

Pada Gambar 8 ada implementasi pseudo code abstrak dari algoritma NegaC *.

```
int negaCStar (int min, int max, int depth)
{
        int score = min;
        while (min != max)
       {
              alpha = (min + max) / 2;
              score = failSoftAlphaBeta (alpha,
alpha + 1, depth);
             if ( score > alpha)
                 min = score;
             else
                 max = score;
         }
         return score;
}
```

Gambar 8. Implementasi Pseudo Code yang Abstrak dari Algoritma NegaC *

Algoritma MTD (f) [14], yang merupakan kependekan dari "Memory-enhanced Test Driver" juga menggunakan teknik leastwindow seperti algoritma NegaScout [3], namun melakukannya dengan efisien. Ini diperkenalkan sebagai perangkat tambahan pada Algoritma AlphaBeta seperti yang disebutkan sebelumnya. Ini juga menggunakan dua variabel lagi untuk menentukan upper-bound dan lower-bound. Algoritma Alpha-Beta normal hanya menggunakan variabel alfa dan beta dengan -∞ & ∞ sebagai permulaan dan nilainya diperbaharui satu kali pada setiap panggilan untuk membuat satu-satunya nilai pengembalian terletak di antara nilai alfa dan beta. Namun, MTD (f) dapat mencari lebih dari satu kali di setiap panggilan Alpha Beta [12] dan menggunakan batas yang dikembalikan untuk bertemu dengannya menggunakan lowerbound dan upper-bound untuk membuat cutoffs pohon yang lebih cepat. Selanjutnya, algoritma menggunakan tabel transposisi untuk menyimpan dan mengambil data tentang bagian pohon pencarian untuk menggunakannya kemudian untuk mengurangi over-head memeriksa ulang status permainan yang sama. Namun, ia menggunakan ruang memori untuk menyimpan data ini, yang membutuhkan lebih banyak ruang memori. Gambar 9 menunjukkan kode pseudo untuk algoritma MTD (f) tanpa penerapan algoritma Alpha-Beta yang dijelaskan pada Gambar 4. SSS * [4] adalah algoritma pencarian pertama yang terkenal, yaitu non-directorial algoritma pencarian Algoritma ini berkembang menjadi beberapa jalur sekaligus untuk mendapatkan informasi global dari pohon pencarian. Namun, ia mencari lebih sedikit simpul daripada algoritma kedalaman-pertama yang pasti seperti algoritma Alfa-Beta. Algoritma menyimpan informasi untuk semua simpul aktif yang belum dipecahkan dalam daftar dalam urutan menurun tergantung pada kepentingannya. Informasi terdiri dari tiga bagian:

- N: pengenal unik untuk setiap simpul.
- S: status dari masing-masing node apa pun itu tinggal atau telah dipecahkan.
- H: nilai penting untuk node.

```
function MTDF(root, f, d)
{
    g := f
    upperBound := +∞
    lowerBound := -∞
    while lowerBound < upperBound
    {
          if g = lowerBound then
                β := g+1
          else
                β := g
          g := AlphaBetaWithMemory (root, β -1,
β , d)
          if g < β then
                upperBound := g
          else
                lowerBound := g
    }
    return g
}
```

Gambar 9. Kode Pseudo untuk Algoritma MTD (F)

Tanpa Implementasi Algoritma Alpha-Beta Yang Telah Digambarkan Sebelumnya pada Gambar 4

Inti dari algoritma SSS * [4] bergantung pada dua fase:

- Ekspansi Node: Ekspansi top-down dari strategi Min.
- Solusi: Pencarian bottom-up untuk strategi Max terbaik.

Gambar 10 menunjukkan kode pseudo untuk algoritma SSS * menggunakan tiga fungsi push, pop dan insert untuk menyimpan, menghapus dan memperbarui informasi node. Algoritma Monte-Carlo Tree "MCTS" [18] membuat terobosan dalam teori permainan dan bidang ilmu komputer. Algoritma ini didasarkan pada eksplorasi acak dari pohon game. Algoritma ini juga menggunakan hasil nilai sebelumnya yang diperiksa untuk node. Setiap kali algoritma berjalan, ia menghasilkan estimasi nilai yang lebih baik. Namun, pohon permainan berangsur-angsur tumbuh dalam memori, yang merupakan kelemahan utama dari algoritma pertama. Algoritma terdiri dari empat fase, yang diulang asalkan masih ada waktu bagi komputer untuk berpikir: Seleksi fase, dimulai dari simpul akar; Ini melintasi

pohon permainan dengan memilih gerakan yang paling menjanjikan sampai mencapai simpul daun.

- Fase perluasan, jika jumlah kunjungan mencapai ambang batas yang telah ditentukan, daun diperluas untuk membangun pohon yang lebih besar.
- Simulasi fase, menghitung nilai hasil daun dengan melakukan play-out pada saat itu.
- Tahap propagasi balik, ia menelusuri kembali jalur pohon permainan dari daun ke akar untuk memperbarui nilai-nilai yang berubah dalam fase simulasi.

```
int SSS* (node n; int bound)
{
     push (n, LIVE, bound);
     while ( true )
    {
          pop (node);
           switch ( node.status ) {
               case LIVE:
                  if (node == LEAF)
                     insert (node, SOLVED,
          min(eval(node),h));
                   if (node == MIN_NODE)
                      push (node.1, LIVE, h);
                   if (node == MAX_NODE)
                     for (j=w; j; j--)
                      push (node.j, LIVE, h);
                   break;
                case SOLVED:
                   if (node == ROOT_NODE)
                     return (h);
                    if (node == MIN_NODE) {
                     purge (parent(node));
                     push (parent(node), SOLVED, h);
                   }
                     if (node == MAX_NODE) {
                       if (node has an unexamined brother)
                         push (brother(node), LIVE, h);
                       else
                         push (parent(node), SOLVED, h);
                   }
                    break;
             }
       }
  }
```

Gambar 10. Kode Pseudo untuk Algoritma SSS *

Gambar 11 menunjukkan kode pseudo untuk algoritma MCTS menggunakan empat fase yang dijelaskan sebelumnya. B * [5] adalah algoritma akhir yang akan dijelaskan di BFS. Algoritma ini menemukan jalur paling murah dari simpul ke simpul tujuan manapun dari satu tujuan yang lebih mungkin. Ide utama algoritma ini didasarkan pada:

 Berhenti saat satu jalur lebih baik dari yang lain.

 Fokus eksplorasi pada jalur yang akan menyebabkan berhenti. Algoritma memperluas pencarian berdasarkan strategi pembuktian terbaik dan penyangkalan. Dalam strategi terbaik, algoritme memilih nodus dengan batas atas tertinggi karena memiliki probabilitas tinggi untuk menaikkan batas bawahnya lebih tinggi daripada batas atas nodus lainnya saat mengembang. Di sisi lain, strategi penyangkalan menolak memilih node terikat tertinggi berikutnya karena memiliki probabilitas yang baik untuk mengurangi batas atas menjadi kurang dari batas bawah anak terbaik saat ia berkembang.

```
Data: root node
Result: best move
while (has time) do
      current node ← root node
      /* The tree is traversed
      while (current node ϵ T ) do
          last node ← current node
          current node ← Select(current node)
      end

      /* A node is added
      last node ← Expand(last node)

       /* A simulated game is played
      R ← P lay simulated game(last node)

       /* The result is backpropagated
      current node ← last node
      while (current node ϵ T ) do
          Backpropagation(current node, R)
          current node ← current node.parent
      end
end

return best move = argmaxN ϵ Nc (root node)
```

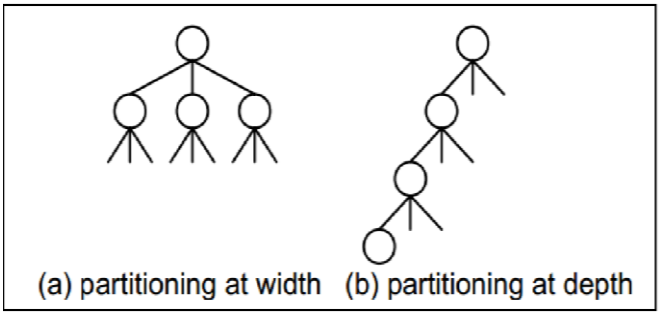
Gambar 11. Pseudo-Code untuk MCTS Algorithm

Kemajuan teknologi arsitektur komputer dan pelepasan multiprosesor dan komputer multi-core, memungkinkan algoritma yang dapat dipartisi menjadi segmen independen dapat dipecahkan dengan lebih cepat. Banyak perangkat tambahan yang dilakukan pada algoritma sekuensial untuk membuatnya berjalan secara paralel dan juga algoritma baru dirancang untuk komputasi paralel. Masalah komputasi paralel adalah trade-off antara overhead komunikasi dan sinkronisasi dan manfaat menjelajahi banyak node dalam waktu yang bersamaan secara paralel. Hal ini membuat speedup bersifat sub linier daripada linier. Bagian [A], menyajikan pembahasan tentang berbagai teknik & algoritma yang dibuat uskuntuk memecahkan masalah ini. Bagian [0], menyajikan sebuah diskusi untuk paralelisme pohon pencarian game dari sudut pandang perangkat keras.

A. Teknik Paralelisme Game Tree

Game Seperti yang telah disebutkan sebelumnya banyak teknik dirancang untuk mengatasi masalah overhead. Salah satunya adalah teknik "Shared Hash Table", yang menyimpan informasi tentang node di pohon permainan sehingga bisa digunakan oleh prosesor. Hal ini mengurangi komunikasi over-head antara prosesor atau core, terutama jika prosesor tidak berada pada komputer fisik yang sama.

Banyak algoritma menggunakan teknik sebelumnya dan juga teknik lainnya, termasuk ABDADA, Parallel Alpha-Beta, PVS Paralel, YBWC dan Jambore dan Dynamic Tree Splitting. ABDADA adalah algoritma pencarian yang disinkronisasi dan didistribusikan secara longgar yang dirancang oleh Jean-Christophe. Algoritma menggunakan teknik tabel hash serta menambahkan lebih banyak informasi untuk /setiap node seperti jumlah prosesor yang mencari node ini. Paralel Alpha-Beta [6] adalah versi paralel dari algoritma Alpha-Beta yang telah dibahas sebelumnya. Ide dasarnya adalah membelah pohon pencarian menjadi pohon sub-pencarian dan menjalankan masing-masing di inti tertentu atau lebih dalam kasus prosesor multi-inti dan satu atau lebih dalam kasus sistem multi-prosesor. Masalah algoritma ini adalah kompleksitas penerapannya. Namun, bisa memaksimalkan pemanfaatan core atau prosesor. Dua metode pemisahan pohon ditunjukkan pada Gambar 12.



Gambar 12. Partisi Pohon Game untuk Pencarian Alpha-Beta

Algoritma PVS [13] mengekspresikan setiap simpul sebagai benang yang bisa berjalan secara paralel. Namun, sebelum menjalankannya secara paralel, masalah ketergantungan data yang ada antar benang harus dipecahkan. Solusi sederhana adalah mendapatkan nilai awal yang dibutuhkan dari simpul pertama di antara setiap saudara kandung dan kemudian menjalankan saudara-saudara yang tersisa secara paralel. Tugas sekuensial dan paralel untuk algoritma PVS menggunakan dua prosesor ditunjukkan pada Gambar 13, sementara kode pseudo untuk algoritma ditunjukkan pada Gambar 14.

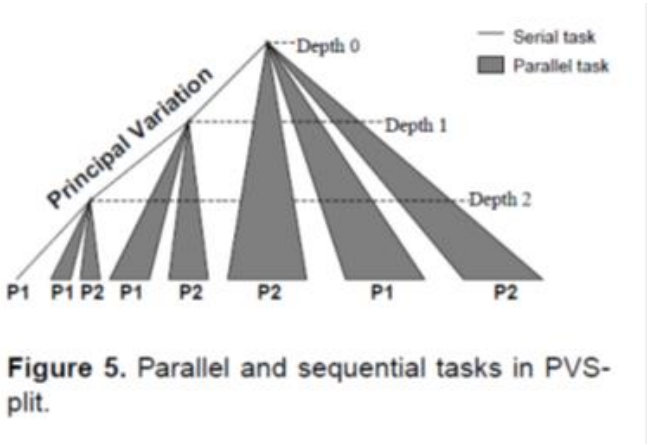


Figure 5. Parallel and sequential tasks in PVS-plit.

Gambar 13. Sequential dan Paralel oleh Dua Prosesor menggunakan PVS

Algoritma YBWC dan Jambore [19] ditunjukkan, berdasarkan algoritma rekursif yang mengunjungi simpul pertama pada saudara kandung sebelum memijah simpul sibling yang tersisa secara paralel. Menggunakan teknik ini karena node pertama bisa menghasilkan cutoff sehingga tidak menyia-nyiakan waktu prosesor lain dengan mencari di node saudara atau akan menghasilkan yang lebih baik.

Kode pseudo dari algoritma ditunjukkan pada Gambar 15.

```
PVSplit (Node curnode, int alpha, int beta, int result)
{
    if(cur_node.is_leaf)
        return Evaluate(cur_node);
    succ_node = GetFirstSucc(cur_node);
    score = PVSplit(curnode, alpha, beta);
     if(score > beta)
        return beta;
    if(score > alpha)
        alpha = score;
    //Begin parallel loop
    while(HasMoreSuccessors(curnode))
    {
        succ_node = GetNextSucc(curnode);
        score = AlphaBeta(succnode,alpha,beta);
        if(score > beta)    return beta;
        if(score > alpha)    alpha = score;
    }
    //End parallel loop
    return alpha;
}
```

Gambar 14. Kode Pseudo untuk Algoritma PVS Menggunakan Fungsi Alpha-Beta

Algoritma DTS [20] adalah algoritma pencarian pohon permainan paralel yang paling kompleks dan ada beberapa implantasi darinya. Namun, hal itu memberikan performa

terbaik dalam sistem multiprocessors simetris. Sebuah pseudo-code dari algoritma DTS yang disajikan pada Gambar 16. Tabel I menunjukkan percepatan untuk ketiga algoritma dibandingkan dengan menggunakan prosesor 1, 2, 4, 8, dan 16.

TABEL I. ALGORITMA PARADEL POPULER GAME ALGORITHMS SPEED UP

Algoritma	Nomor processor				
	1	2	4	8	16
PVS	1.0	1.8	3.0	4.1	4.6
YBWC	1.0	1.9	3.4	6.1	10.9
DTS	1.0	2.0	3.7	6.6	11.1

Semua algoritma paralel sebelumnya membutuhkan bahasa pemrograman paralel atau *library* yang bisa menangani benang dan komputasi paralel. Banyak perpustakaan dan bahasa pemrograman dilepaskan untuk mendukung paralelisme CPU secara umum. Namun, perpustakaan paling terkenal yang digunakan dalam algoritma tree game paralel adalah MPI (Message Passing Interface) [21] yang merupakan perpanjangan bahasa pemrograman C. MPI menangani beban sinkronisasi, komunikasi dan manajemen sumber daya terdistribusi. Versi terbaru dari MPI adalah versi 2 yang mendukung pemrograman C ++ dan object-oriented.

Gambar 15. Kode Pseudo untuk Algoritma Jambore

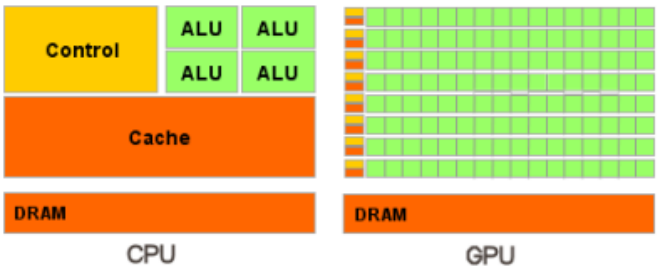
```
jamboree(CNode n, int α, int β, int b)
DTS(root) {
while (Stopping_criterion() == false)
{
    //One processor search to ply = N
    SearchRoot(root);
    //Detect free processors, and begin tree split
    Split(node v);
    //Initialize new threads.
    ThreadInit();
    //Copy a “split block” to begin a new search
    CopytoSMP(node v);
    SearchSMP(node v);
  }
    ThreadStop();
}
            if (s >= β) abort_and_return s;
            if (s >  α) α = s;
            if (s >  b) b = s;
          }
    }
    return b;
}
```

Gambar 16. Pseudo Code untuk Algoritma DTS

Library untuk algoritma kecerdasan buatan yang berjalan pada CPU adalah Microsoft Task Parallel Library (TPL) [22]. *Library* ini menggunakan konsep komputasi terikat CPU yang terbatas berdasarkan notasi dan konsep mereplikasi tugas menggunakan teknik pencurian-kerja. Ini lebih efektif untuk mengembangkan algoritma paralel seperti DTS dan YBWC. Pustaka pemrograman lainnya dirancang untuk menjalankan algoritma paralel di GPU daripada CPU, termasuk CUDA [23]. Namun, beberapa algoritma diimplementasikan dengan menggunakan pustaka ini karena kompleksitas pohon pencarian pemrograman yang menggunakan pustaka ini. Di sisi lain, algoritma yang diimplementasikan pada GPU menunjukkan kecepatan yang lebih baik daripada CPU.

B. CPU & GPU untuk Pencarian Pohon Game Paralel

Dalam lima belas tahun sebelumnya, semua penelitian berfokus pada perancangan algoritma paralel yang dapat berjalan secara paralel pada multicore atau multi-prosesor. Namun, tren baru bidang pohon pencarian adalah merancang dan menerapkan algoritma yang bisa berjalan paralel di GPU. Awal GPU dibangun hanya untuk komputasi grafis. Namun, dalam 10 tahun terakhir GPU menjadi platform komputasi paralel umum. Gagasan GPU adalah memiliki ratusan atau ribuan core sederhana yang bisa menjalankan thread secara paralel dengan GFLOPS yang lebih tinggi daripada CPU. Di sisi lain, CPU berisi beberapa core yang kuat atau beberapa prosesor multi-kuat yang bisa menjalankan lebih banyak instruksi dan memiliki kecepatan waktu lebih cepat daripada GPU. Seperti yang disebutkan sebelumnya beberapa algoritma dimodifikasi untuk mendukung GPU. Namun, dalam lima tahun ke depan banyak algoritma AI akan dirancang untuk menggunakan kekuatan GPU. Gambar 17 menunjukkan arsitektur CPU dan GPU.



Gambar 17. Perbedaan Antara Arsitektur CPU dan GPU

IV. ANALISIS ALGORITMA GAME TREE

Sebagai pohon permainan dikategorikan sesuai dengan kriteria tertentu, evaluasi algoritma sebelumnya dikategorikan sesuai dengan kriteria spesifik; yaitu:

- Kelengkapan: apapun jika algoritma menemukan solusinya jika ada.
- Time Complexity: jumlah node yang dihasilkan.

- Kompleksitas Ruang: jumlah maksimum node dalam memori selama pencarian.
- Optimalitas: apapun jika algoritma selalu menemukan solusi paling murah atau terbaik. Istilah berikut digunakan untuk mengukur kompleksitas waktu dan ruang:
- B: faktor cabang maksimum.
- D: kedalaman solusi terbaik.
- M: kedalaman maksimum ruang negara.
- L: depth cut-off

Tabel II membandingkan berbagai kategori algoritma berurutan berdasarkan kriteria sebelumnya.

TABEL II. ANALISIS ALGORITMA SEQUENTIAL

Kriteria	Breadth -First	Depth -First	Depth -Limited	Iterative Depending
Completeness	Yes	No	Yes, if l >= d	Yes
Time	b^d	b^m	b^l	b^d
Space	bd	b^m	b^l	b^d
Optimality	Maybe	No	No	Maybe

Semua algoritma yang termasuk dalam kategori apa pun harus sesuai dengan kriteria tabel yang disebutkan. Namun, ini adalah skenario terburuk. Dalam kebanyakan kasus, nilai ruang dan waktu yang lebih baik dapat ditemukan. Contohnya adalah algoritma Alpha-Beta dimana rata-rata waktunya sama dengan $b^{3m/4}$. Analisis kompleksitas algoritma pencarian pohon game paralel lebih sulit daripada algoritma sekuensial. Biasanya, algoritma pencarian pohon game paralel dapat dianalisis dalam hal:

- Kompleksitas waktu T (n): Berapa kali langkah dibutuhkan?
- Kompleksitas prosesor P (n): Berapa banyak prosesor yang digunakan?
- Kompleksitas kerja W (n): Berapa total pekerjaan yang dilakukan oleh semua prosesor?

Algoritma Minimax sekuensial dengan modifikasi alpha-beta memiliki kompleksitas waktu O (b^m) dalam kasus terburuk, O ($b^{3m/4}$) dalam kasus rata-rata dan O ($b^{m/2}$) dalam kasus terbaik; Dimana b adalah faktor percabangan dan m adalah kedalaman maksimum pohon. Total kerja yang dilakukan oleh algoritma alpha-beta paralel sama dengan total kerja yang dilakukan oleh algoritma sekuensial. Oleh karena itu, tabel III merangkum kompleksitas waktu dari algoritma alfa-beta paralel dalam kasus terbaik, rata-rata dan terburuk. Dengan meningkatkan jumlah prosesor atau core, algoritma tersebut dapat memberikan kecepatan yang lebih baik. Namun, tidak akan pernah mencapai kecepatan yang ideal, karena overhead komunikasi antara node untuk berbagi nilai alpha dan beta serta overhead sinkronisasi.

TABEL III. ANALISIS ALGORITMA PARALEL

Criterion	Case	Complexity
Time Complexity	Worst	$T(n) = O\left(\frac{b^m}{p}\right)$
	Average	$T(n) = O\left(\frac{b^{\frac{3m}{4}}}{p}\right)$
	Best	$T(n) = O\left(\frac{b^{\frac{m}{2}}}{p}\right)$
Processor Compleixty	Worst	$P(n) = O(n)$
	Avarage	
	Best	
Work Compleixty	Worst	$W(n) = O(b^m)$
	Avarage	$W(n) = O\left(b^{\frac{3m}{4}}\right)$
	Best	$W(n) = O\left(b^{\frac{m}{2}}\right)$

V. KESIMPULAN

Dalam pembahasan game pohon n-ary yang disajikan dalam makalah ini, termasuk algoritma sekuensial dan paralel. Algoritma sekuensial dibahas secara rinci algoritma umum di depth-first dan breadth-first. Selanjutnya, ikhtisar algoritma paralel umum serta arsitektur perangkat keras untuk pencarian pohon permainan paralel dipresentasikan. Pada akhirnya, sebuah analisis algoritma berdasarkan empat kriteria telah dibahas. Penggunaan pendekatan berorientasi layanan untuk memperluas pohon pencarian ke dalam sistem terdistribusi akan memecahkan banyak masalah terdistribusi, Saran yang dapat diberikan untuk makalah ini adalah menerapkan algoritma penelusuran menggunakan pustaka OpenCL yang memungkinkan kode berjalan di GPU dan CPU, atau *library* CUDA untuk menghasilkan algoritma pencarian pohon game paralel yang lebih berkualitast. Fitur baru paralelisme dinamis di CUDA v5.5 memungkinkan algoritma berbasis rekursi berjalan lebih cepat di GPU, dengan menghilangkan waktu inisialisasi CPU dari masing-masing kernel, yang membuat pengembangan permainan kompleks menjadi lebih mudah. Kedua paralelisme dinamis dan fitur memori terpadu seingga dapat meningkatkan kecepatan algoritma pencarian pohon AI.

REFERENCES

[1] S. Russell and P. Norvig, Artificial intelligence: a modern approach, 3rd ed. Prentice Hall Press, 2009, p. 1152.

[2] G. T. Heineman, G. Pollice, and S. Selkow, "Path Finding in AI," in Algorithms in a Nutshell, 1st ed., O'Reilly Media, 2008, pp. 213–217.

[3] H.-J. Chang, M.-T. Tsai, and T. Hsu, "Game Tree Search with Adaptive Resolution," in Advances in Computer Games SE - 26, vol. 7168, H. J. Herik and A. Plaat, Eds. Springer Berlin Heidelberg, 2012, pp. 306–319. [4]

[4] U. Lorenz and T. Tscheuschner, "Player Modeling, Search Algorithms and Strategies in Multi-player Games," in Advances in Computer Games SE - 16, vol. 4250, H. J. Herik, S.-C. Hsu, T. Hsu, and H. H. L. M. (Jeroen. Donkers, Eds. Springer Berlin Heidelberg, 2006, pp. 210–224.

[5] Y. Tsuruoka, D. Yokoyama, and T. Chikayama, "Game-Tree Search Algorithm Based on Realization," ICGA J., vol. 25, no. 3, pp. 145–152, 2002.

[6] V. Manohararajah, "Parallel Alpha-Beta Search on Shared Memory Multiprocessor," Computer Engineering University of Toronto, 2001.

[7] D. Strnad and N. Guid, "Parallel alpha-beta algorithm on the GPU," CIT, vol. 19, no. 4, pp. 269–274, 2011.

[8] J. Habgood and M. Overmars, "Clever Computers: Playing Tic-TacToe," in The Game Maker's Apprentice SE - 13, Apress, 2006, pp. 245– 257.

[9] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Depth-first search," in Introduction to Algorithms, 3rd ed., MIT Press, 2009, pp. 603–612.

[10] W. Ertel, "Search, Games and Problem Solving," in Introduction to Artificial Intelligence SE - 6, Springer London, 2011, pp. 83–111

[11] M. Schadd, "Selective Search in Games of Different Complexity," Maastricht University, 2011

[12] D. Knuth, Selected Papers on Analysis of Algorithms. California: Center for the Study of Language and Information, 2000.

[13] M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and E. C. D. van der Werf, "Enhanced forward pruning," Inf. Sci. (Ny)., vol. 175, no. 4, pp. 315–329, 2005.

[14] K. Shibahara, N. Inui, and Y. Kotani, "Adaptive Strategies of MTD-f for Actual Games," in CIG, 2005.

[15] M. Schadd and M. Winands, "Quiescence Search for Stratego," in BNAIC, 2009, pp. 225–232.

[16] A. X. Jiang and M. Buro, "First Experimental Results of ProbCut Applied to Chess," Adv. Comput. Games, vol. 10, 2003.

[17] D. Rutko, "Fuzzified Tree Search in Real Domain Games," in Advances in Artificial Intelligence SE - 13, vol. 7094, I. Batyrshin and G. Sidorov, Eds. Springer Berlin Heidelberg, 2011, pp. 149–161.

[18] J. Hashimoto, A. Kishimoto, K. Yoshizoe, and K. Ikeda, "Accelerated UCT and Its Application to Two-Player Games," Adv. Comput. Games, 2011.

[19] J. Steenhuisen, "Transposition-Driven Scheduling in Parallel TwoPlayer State-Space Search," Delft University of Technology, 2005.

[20] T. A. N. Ying, L. U. O. Ke-lu, C. Yu-rong, and Z. Yi-min, "Performance Characterization of Parallel Game-tree Search Application Crafty," vol. 4, no. 2, pp. 2–7, 2006. 2012

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, 3 Desember 2017

Ivan Fadillah
13516128