Pemrograman Dinamis untuk Optimisasi Algoritma Minimax pada AI Catur

Devin Alvaro
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13515062@std.stei.itb.ac.id

***Abstract* — A*rtificial intelligence* untuk permainan catur pada umumnya menggunakan algoritma minimax sebagai algoritma pengambilan keputusan. Hasil keputusan algoritma minimax bergantung pada kedalaman pencariannya, di mana semakin dalam pencariannya maka hasilnya semakin baik. Namun, pada permainan catur, setiap bertambah kedalaman pencarian, pada umumnya ruang pencarian bertambah secara eksponensial, sehingga waktu yang dibutuhkan untuk menambah kedalaman pencarian pun bertambah secara eksponensial. Maka, agar dapat melakukan pencarian minimax yang cukup dalam, dibutuhkan optimisasi untuk mengurangi waktu pencarian. Makalah ini membahas optimisasi minimax pada permainan catur dengan pemrograman dinamis, di mana hasil pencarian minimax disimpan dengan *state*-nya adalah kondisi permainan ditambah *depth* pencarian. Dengan pemrograman dinamis, pencarian yang sudah pernah dilakukan tidak perlu diulang lagi, sehingga dapat mengurangi jumlah pencarian dan waktu pencarian yang dibutuhkan.**

***Keywords* — minimax, *artificial intelligence*, catur, pemrograman dinamis, optimisasi**

I. PENDAHULUAN

Algoritma minimax adalah algoritma yang banyak digunakan untuk *artificial intelligence* (AI) dari permainan dengan informasi sempurna (*perfect information)*, seperti *tic-tac-toe*, *othello*, *checkers*, dan catur. Meskipun AI dari permainan-permainan tersebut sudah banyak berkembang dari algoritma minimax dasar, namun ide dasar yang digunakan tetaplah minimax.

Pada dasarnya, algoritma minimax melakukan pencarian pada suatu pohon skor secara DFS. Setiap simpul dari pohon tersebut merepresentasikan skor suatu kondisi permainan. Kondisi permainan didapatkan dari rangkaian langkah-langkah pada permainan, sehingga *path* dari simpul akar ke suatu simpul adalah rangkaian langkah-langkah permainan dari awal ke kondisi pada simpul tersebut. Karena biasanya permainan terdiri dari 2 pemain atau lebih, *parent* dari suatu simpul adalah kondisi permainan yang dihasilkan dari langkah pemain sebelumnya.

Ide dasar dari algoritma minimax adalah mencari langkah yang langkah balasan terburuknya adalah yang terbaik. Perlu dicatat bahwa “balasan terburuk” adalah balasan yang menghasilkan kondisi terburuk dari sudut pandang AI, bukan lawannya. Lebih jelasnya, untuk setiap kemungkinan langkah AI, cari langkah balasan lawan dengan skor minimal (terburuk bagi AI). Kemudian, pilih langkah AI yang skor-skor langkah balasan minimalnya adalah yang maksimal (terbaik bagi AI).

Hasil pencarian algoritma minimax sangat bergantung pada kedalaman pencariannya. Semakin dalam pencariannya, semakin jauh algoritma dapat memprediksi langkah terbaik untuk diambil. Namun, hal ini dapat menjadi masalah untuk permainan seperti catur, karena seiring bertambahnya kedalaman pencarian, ruang pencariannya pun bertambah secara eksponensial. Maka dari itu, untuk mempercepat waktu pencarian, dibutuhkan optimisasi. Salah satunya adalah optimisasi dengan pemrograman dinamis yang menjadi fokus makalah ini.

Pemrograman dinamis merupakan metode pemecahan masalah dengan memecah persoalan utama menjadi sekumpulan sub-persoalan. Komputasi solusi dilakukan dengan menyimpan hasil komputasi seluruh sub-persoalan terlebih dahulu yang kemudian dibangun untuk mendapatkan solusi persoalan utama. Dengan pemrograman dinamis, program tidak perlu melakukan komputasi yang sudah pernah dilakukan, karena hasil setiap komputasi disimpan.

Penulisan makalah ini terinspirasi dari *project* pribadi penulis. Dalam project tersebut, penulis mencoba mengimplementasikan AI catur dengan algoritma minimax dan alpha-beta pruning. Karena rendahnya performa AI program tersebut untuk depth tertentu, muncul ide untuk mengoptimisasinya dengan pemrograman dinamis yang terinspirasi dari implementasi pemrograman dinamis pada soal-soal *competitive programming*. Source code dan deskripsi selengkapnya mengenai project tersebut dapat dilihat pada https://github.com/devinalvaro/yachess[2].

II. Dasar Teori

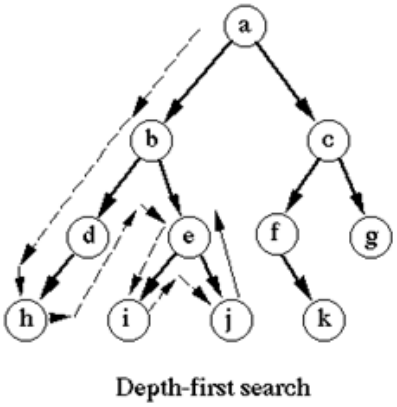
Dasar teori makalah ini mencakup algoritma depth-first search yang merupakan dasar algoritma minimax, serta pemrograman dinamis sebagai optimisasi algoritma minimax.

Selain itu akan dijelaskan juga Forsyth-Edwards Notation (FEN), yaitu notasi catur yang menggambarkan kondisi papan ke dalam string. Notasi ini akan digunakan sebagai state dari pemrograman dinamis bersama dengan *depth* pencarian.

A. Depth-first search (DFS)

Depth-first search (DFS) adalah algoritma traversal pada graf atau pohon. Prinsip utama dari algoritma DFS adalah melakukan traversal secara mendalam, yakni dari simpul sekarang (misal *u)* menelusuri terlebih dahulu simpul tetangga pertama (misal *v)* dari simpul sekarang, kemudian secara rekursif menelusuri simpul tetangga pertama (misal *w)* dari simpul *v* tersebut. Proses ini dilakukan secara terus menerus hingga dicapai simpul yang tidak memiliki tetangga yang belum dikunjungi (misal *x)*, barulah DFS menelusuri simpul tetangga berikutnya yang belum dikunjungi dari *parent* simpul *x* tersebut.

Berikut adalah gambaran algoritma DFS. Lintasan traversal graf pada contoh tersebut adalah *a-b-d-h-e-i-j-c-f-k-g*.



Gambar 1: Contoh pencarian graf dengan DFS
Sumber: https://codinglifestyle.files.wordpress.com/2011/10/102411_1147_findcontrol1.png

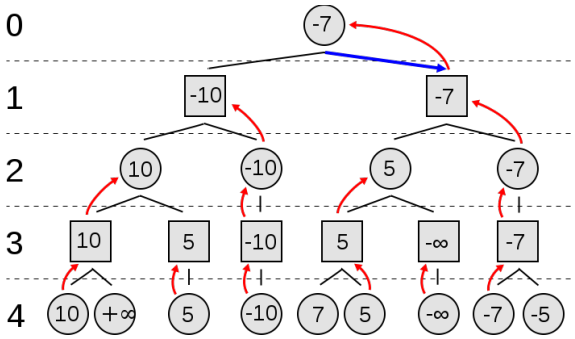
Berikut adalah contoh *pseudo-code* algoritma DFS:

```
function dfs(current_node):
  visited[current_node] = True

  for each node in current_node.neighbors:
    if not visited[node]:
      dfs(node)
```

B. Minimax

Minimax adalah algoritma pengambilan keputusan yang banyak digunakan untuk *artificial intelligence* pada *perfect-information game* seperti catur. Seperti telah dijelaskan pada pendahuluan, ide dasar dari algoritma minimax adalah mencari langkah sendiri yang langkah balasan terburuknya (bagi AI) adalah yang terbaik (bagi AI). Berikut adalah contoh pohon pencarian minimax beserta penjelasannya:



Gambar 2: Contoh pohon pencarian pada algoritma minmax
Sumber: https://codinglifestyle.files.wordpress.com/2011/10/102411_1147_findcontrol1.png

Pada pohon minimax tersebut, simpul dengan depth dari akar genap (0, 2, dan 4) adalah simpul yang merepresentasikan langkah sendiri sehingga skornya dimaksimalkan (algoritma mencari langkah yang balasan terburuknya adalah yang terbaik). Sedangkan simpul dengan depth dari akar ganjil (1 dan 3) adalah simpul yang merepresentasikan langkah balasan sehingga diminimalkan (algoritma mencari langkah balasan terburuk).

Pada gambar, simpul akar memiliki 2 simpul anak dengan skor -10 dan -7. Karena simpul akar adalah simpul langkah AI, maka diambil skor maksimal yaitu -7. Lebih lanjut, simpul anaknya memiliki 2 anak dengan skor 5 dan -7. Karena simpul tersebut adalah simpul langkah lawan, maka diambil skor minimal yaitu -5. Begitu seterusnya proses rekursif tersebut berlanjut hingga seluruh pohon telah ditelusuri dengan algoritma minimax.

Berikut adalah contoh *pseudo-code* algoritma minimax:

```
function minimax(depth, is_maximising):
  if depth == 0:
    return board.score()
  else:
    best_score = INFINITY if is_maximising
                          else -INFINITY

    for each move in board.possible_moves:
       board.do(move)

       score = minimax(depth - 1, not
                                  is_maximising)
       if is_maximising:
         best_score = max(best_score, score
       else:
         best_score = min(best_score, score)

              board.undo()


    return best_score
```

C. Pemrograman Dinamis

Pemrograman dinamis adalah metode pemecahan masalah dengan mengurai persoalan menjadi sekumpulan sub-persoalan identik. Karena sub-persoalan lebih kecil, maka lebih mudah dikomputasi dibandingkan solusi utama. Hasil komputasi kumpulan sub-persoalan ini kemudian dibangun untuk mengkomputasi solusi persoalan utama.

Pada pemrograman dinamis, hasil komputasi setiap sub-persoalan disimpan, sehingga bila suatu saat dibutuhkan, tidak perlu dikomputasi ulang. Karena pemrograman dinamis menghilangkan komputasi yang sia-sia, pemrograman dinamis sering digunakan untuk mengoptimisasi program. Yang paling menarik adalah persoalan-persoalan *non-polynomial* seperti TSP dan Knapsack dapat direduksi menjadi *pseudo-polynomial* dengan pemrograman dinamis.

Salah satu contoh pemrograman dinamis yang umum adalah bilangan Fibonacci:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Dari observasi terhadap barisan bilangan Fibonacci tersebut, didapatkan bahwa fungsi yang menghasilkan bilangan Fibonacci ke-n memiliki properti rekursif, yaitu:

$$f(n) = f(n - 1) + f(n - 2)$$

$$f(n - 1) = f(n - 2) + f(n - 3)$$

$$f(n - 2) = f(n - 3) + f(n - 4)$$

...

Karena f(n) memiliki sub-persoalan yang identik yaitu f(n - 1) dan f(n - 2), maka fungsi mencari bilangan Fibonacci ke-n dapat dioptimisasi dengan pemrograman dinamis, di mana untuk mengkomputasi f(n), dikomputasi terlebih dahulu f(n - 1) dan f(n - 2) dan disimpan hasilnya, begitu seterusnya secara rekursif.

Pada pemrograman dinamis dikenal istilah *state*, yaitu kondisi atau properti dari sub-persoalan yang dikomputasi. Pada contoh persoalan Fibonacci di atas, *state* dari persoalan adalah n.

D. Forsyth-Edwards Notation (FEN)

Forsyth Edwards Notation (FEN) adalah notasi yang mendeskripsikan posisi bidak-bidak pada suatu permainan catur dalam satu baris teks. Tujuan diperkenalkannya FEN pada makalah ini karena akan digunakan sebagai *state* pemrograman dinamis yang mengoptimisasi algoritma minimax. Oleh karenanya, FEN yang akan digunakan telah disederhanakan agar cukup hanya memenuhi tujuan tersebut.

FEN yang akan digunakan dibagi menjadi 2 bagian. Bagian pertama adalah penanda posisi bidak-bidak catur dan bagian kedua adalah penanda giliran pemain, di mana ‘w’ berarti giliran pemain putih sedangkan ‘b’ berarti giliran pemain hitam.

Bagian pertama dibagi lagi menjadi 8 *section* yang dipisah oleh karakter ‘/’. Setiap *section* merepresentasikan *row* papan dari atas ke bawah. Dalam setiap *section*, terdapat huruf yang merepresentasikan bidak-bidak catur dari kiri ke kanan sesuai SAN*. Selain karakter, *section* juga dapat diisi angka yang merepresentasikan banyak kolom yang diantarai angka tersebut.

Berikut adalah contoh FEN posisi awal catur:

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w
```

Gambar 3: Kondisi permainan awal catur
Sumber: http://www.dhammawiki.com/images/e/e1/Chess_diagram.JPG

Dengan notasi FEN, posisi seperti apapun dalam permainan catur dapat dikonversi menjadi string saja. Hal ini memungkinkan penggunaan FEN sebagai *key* dari struktur data *map* dengan *value* skor dari posisi permainan catur tersebut.

*SAN (Standard Algebraic Notation) adalah notasi untuk merepresentasikan bidak catur ke dalam huruf. Huruf kapital berarti berarti bidak putih sedangkan huruf kecil berarti bidak hitam. Berikut adalah daftar huruf SAN dan bidak yang direpresentasikannya:

p/P: pion (pawn)	n/N: kuda (knight)
b/B: gajah (bishop)	r/R: benteng (rook)
q/Q: ratu (queen)	k/K: raja (king)

III. Implementasi dan Eksperimen

A. Fungsi Evaluasi

Karena algoritma minimax memproses skor dari keadaan papan catur pada tiap simpul di pohon, diperlukan fungsi evaluasi yang memberi skor dari keadaan papan.

Pada intinya, skor didapatkan dari nilai material dan posisi dari setiap bidak yang tersisa di papan. Pada teori catur, nilai material adalah kekuatan setiap bidak relatif terhadap pion. Pion bernilai 1, kuda dan gajah bernilai 3, benteng bernilai 5, ratu bernilai 9, dan raja bernilai tak hingga. Sedangkan nilai posisi adalah nilai plus/minus suatu bidak di posisi tertentu. Misalnya, bidak kuda mendapat nilai plus bila ada di tengah-tengah papan,

namun mendapat nilai minus bila berada di pinggiran. Karena nilai posisi bervariasi untuk tiap bidak dan posisinya pada papan, maka tidak akan dibahas secara lebih detil pada makalah ini. Selain itu, nilai bidak hitam dikali -1 agar nilainya berlawanan dengan putih.

Berikut *pseudo-code* fungsi evaluasi yang digunakan:

```
function score():
  sum = 0

  for each piece in board.pieces:
    sum += piece.material_value +
               piece.position_value

  if piece.color == 'black':
    sum *= -1

  return sum
```

Untuk selanjutnya, fungsi evaluasi permainan diabstraksi menjadi *method* `board.score()`.

B. Implementasi

Perlu diperhatikan bahwa karena fungsi yang meng-*convert* keadaan board menjadi FEN cukup rumit, maka implementasinya tidak dibahas secara detail pada makalah ini. Untuk selanjutnya, fungsi merubah kondisi permainan menjadi FEN diabstraksi menjadi *method* `board.FEN()`.

Implementasi pemrograman dinamis pada algoritma minimax ini adalah dengan menyimpan `board.score()` sebagai *value* pada struktur data map `board_memo` dengan *key*-nya `board.state()` = `board.FEN() + depth` yang merupakan kedua state yang digunakan, yaitu kondisi permainan dan kedalaman pencarian.

Berikut ini adalah *pseudo-code* dari algoritma minimax yang dioptimisasi dengan pemrograman dinamis:

```
function minimax(depth, is_maximising):
  if board.state() in board_memo:
    return board_memo[board.state]
  else if depth == 0:
    board_memo[board.state()] = board.score()
    return board_memo[board.state()]
  else:
    best_score = INFINITY if is_maximising
                          else -INFINITY

    for each move in board.possible_moves:
       board.do(move)

       score = minimax(depth - 1, not
                          is_maximising)
       board_memo[board.state()] = score

       if is_maximising:
         best_score = max(best_score, score)
       else:
         best_score = min(best_score, score)

       board.undo()
    board_memo[board.state()] = best_score
    return board_memo[board.state()]
```

Pada fungsi minimax tersebut, `if board.FEN() in board_memo` memeriksa apabila keadaan permainan sekarang sudah pernah dikomputasi dan disimpan sebelumnya, kemudian `return board_memo[board.FEN()]` mengembalikan hasil komputasi tersebut.

Selanjutnya fungsi memeriksa base case dengan `else if depth == 0` kemudian menyimpan hasil evaluasi langsung skor permainan ke `board_memo` sebelum mengembalikannya.

Apabila kedua kondisi tersebut tidak terpenuhi, maka program harus menelusuri pohon pencarian ke *depth* selanjutnya. `for each move in board.possible_moves` mengeksplorasi seluruh kemungkinan langkah selanjutnya, kemudian secara rekursif menjalankan fungsi minimax untuk *depth* selanjutnya. Setelahnya, dicari `best_score` dengan memaksimalkan atau meminimalkan skor yang ada sesuai konsep minimax yang telah dijelaskan pada Dasar Teori. Sebelum mengembalikan `best_score`, fungsi menyimpannya terlebih dahulu ke `board_memo` sebagai bagian dari optimisasi pemrograman dinamis pada minimax tersebut.

C. Eksperimen

Karena program diimplementasi dengan bahasa Python, mulai dari *depth* tertentu *runtime* program ini menjadi cukup lambat. Oleh karena itu, eksperimen pada program ini terdiri dari 4 *testcase* saja. Masing-masing *testcase* adalah *game* catur yang terdiri dari 5 giliran, di mana pada *testcase* pertama dan ketiga AI memegang bidak hitam dan pada *testcase* kedua dan keempat AI memegang bidak putih. *Depth* pencarian pada *testcase 1* dan *2* adalah 3, sedangkan *depth* pencarian pada *testcase 3* dan *4* adalah 4.

Pada eksperimen ini, pada setiap *testcase* dibandingkan jumlah pemanggilan fungsi *minimax* antara dua program yang sama, perbedaannya yang satu menerapkan pemrograman dinamis dan yang satunya lagi tidak. Hipotesis eksperimen ini adalah pada program yang menerapkan pemrograman dinamis, jumlah pemanggilan fungsi *minimax* berkurang karena pemrograman dinamis mengeliminasi pemanggilan fungsi (komputasi) rekursif yang tidak perlu.

Berikut adalah langkah-langkah catur dari setiap *testcase* dalam notasi standar catur. Arti notasi tersebut tidak perlu dipikirkan, yang penting menggambarkan bahwa keempat *testcase* adalah 4 *game* catur dengan langkah-langkah yang berbeda. Selain itu, 'W' berarti langkah putih dan 'B' berarti langkah hitam. Tanda '+' berarti adalah warna yang dimainkan oleh AI.

Turn	*Testcase 1*		*Testcase 2*		*Testcase 3*		*Testcase 4*	
	W	B+	W+	B	W	B+	W+	B
1	e4	Nf6	Nf3	d5	e4	Nc6	Nf3	d5
2	e5	Ne4	Nc3	Nf6	d4	e6	e3	Nc6
3	d3	Nc5	Ne5	Bf5	Nf3	Nf6	Bb5	Bd7
4	d4	Ne4	Nb5	c6	e5	Nd5	Nc3	e6
5	Nf3	Nc6	Nd4	c5	c4	Bb4	Bxc6	Bxc6

Eksperimen menghasilkan data berikut.

Turn	Jumlah pemanggilan fungsi *minimax* (n)			
	Testcase 1 (depth = 3)		*Testcase 2* (depth = 3)	
	Tanpa DP	Dengan DP	Tanpa DP	Dengan DP
1	13,780	8,650	9,322	5,782
2	15,253	9,344	15,313	9,332
3	22,953	13,669	19,184	11,550
4	21,151	13,026	29,239	17,430
5	23,918	13,948	33,297	20,622
Total	97,055	58,636	106,355	64,716

Turn	Jumlah pemanggilan fungsi *minimax* (n)			
	Testcase 3 (depth = 4)		*Testcase 4* (depth = 4)	
	Tanpa DP	Dengan DP	Tanpa DP	Dengan DP
1	419,165	110,419	206,603	77,634
2	765,642	259,162	438,401	154,468
3	1,378,393	453,569	736,141	252,591
4	1,110,015	373,645	874,339	271,489
5	1,382,968	452,983	1,516,225	489,498
Total	5,056,183	1,649,778	3,771,709	1,245,680

Dari data-data tersebut, akan dihitung efisiensi dari implementasi pemrograman dinamis dengan rumus berikut:

Efisiensi = ((n – m) / n) x 100%

n = jumlah pemanggilan fungsi *minimax* pada program tanpa implementasi pemrograman dinamis

m = jumlah pemanggilan fungsi *minimax* pada program dengan implementasi pemrograman dinamis

Turn	*Efisiensi*			
	Testcase 1	*Testcase 2*	*Testcase 3*	*Testcase 4*
1	37.22%	37.97%	73.65%	62.42%
2	38,73%	39.12%	66.15%	64.76%
3	40.44%	39.79%	67.09%	65.68%
4	38.41%	40.38%	66.33%	68.94%
5	41.6%	38.06%	67.24%	67.71%
Total	39.58%	39.15%	67.37%	66.97%

Berdasarkan data-data eksperimen tersebut, didapatkan bahwa mengimplementasikan pemrograman dinamis pada algoritma *minimax* dapat mengurangi jumlah komputasi hingga kira-kira 39.36% pada *depth* pencarian 3 dan 67.17% pada *depth* pencarian 4.

IV. Analisis

A. Analisis Data

Berdasarkan data eksperimen, jelas bahwa pada program minimax yang mengimplementasikan pemrograman dinamis, jumlah pemanggilan fungsi *minimax* berkurang secara cukup signifikan. Hal ini dikarenakan dengan pemrograman dinamis, komputasi yang sudah pernah dilakukan tidak perlu diulang lagi. Dengan demikian, hipotesis pada bab sebelumnya terbukti benar.

Selain itu, terdapat tren pada data, di mana seiring meningkatnya *depth* pencarian, efisiensi implementasi pemrograman dinamis meningkat secara signifikan. Hal ini ditunjukkan dari rata-rata efisiensi pada *testcase 1* dan *2* (dengan *depth* pencarian 3) yaitu 39.36%, sedangkan rata-rata efisiensi pada *testcase 3* dan *4* (dengan *depth* pencarian 4) adalah 67.17%. Peningkatan efisiensi ini disebabkan karena seiring meningkatnya *depth*, semakin banyak sub-persoalan yang berulang, sehingga pengambilan hasil komputasi secara pemrograman dinamis menjadi semakin sering.

B. Analisis Kompleksitas

Algoritma minimax memiliki kompleksitas $O(b^d)$ di mana d adalah *depth* dan b adalah branching factor, yakni rata-rata banyaknya anak dari tiap simpul. Hal ini dikarenakan pada *depth* pertama, banyak simpul yang dicari sekitar b. Selanjutnya ketika menambah *depth* baru, setiap simpul pada *depth* saat ini akan memiliki kira-kira b simpul anak baru.

Pada program minimax yang mengimplementasi pemrograman dinamis, kompleksitas algoritma didapat dari komplesitas ruang pencarian dikali kompleksitas mendapatkan data yang sudah disimpan. Pada kasus minimax dengan dp ini, berarti kompleksitasnya adalah jumlah posisi catur yang sudah disimpan (n) dikali *depth* pencarian (*d*) dikali kompleksitas mendapatkan data dari struktur data map, yaitu log(n).

Kompleksitas minimax dengan dp:

O(n * d * log(n)),

n = jumlah kondisi permainan tersimpan
d = *depth* pencarian
logn(n) = mendapatkan data dari map

Akan tetapi, karena jumlah kondisi permainan (n) pada permainan catur sangat banyak dan sulit diprediksi, kompleksitas tersebut mungkin tidak dapat menjadi perkiraan yang pasti akan *runtime* dari program minimax yang dioptimisasi dengan pemrograman dinamis.

V. Kesimpulan

Hasil algoritma minimax bergantung pada *depth* pencarian, di mana semakin dalam *depth* pencarian, semakin baik hasilnya. Akan tetapi, pada permainan catur, seiring penambahan *depth*, ruang pencarian meningkat secara eksponensial.

Pemrograman dinamis dapat mengoptimisasi algoritma minimax karena hasil setiap komputasi disimpan sehingga komputasi yang sama tidak perlu diulang. Setelah dilakukan eksperimen terhadap implementasi pemrograman dinamis pada minimax, didapatkan hasil bahwa pemrograman dinamis dapat mengoptimisasi progrma secara cukup signifikan, yaitu mengurangi jumlah pencarian kira-kira 39.36% pada pencarian dengan *depth* 3 dan 67.17% pada pencarian dengan *depth* 4.

Terlebih, peningkatan efisiensi optimisasi dari *depth* 3 ke 4 tersebut menunjukkan bahwa seiring peningkatan *depth*, efisiensi optimisasi meningkat juga. Hal ini menunjukkan bahwa pemrograman dinamis dapat semakin efisien pada algoritma minimax dengan *depth* yang semakin dalam.

Dengan kata lain, implementasi pemrograman dinamis secara langsung mengurangi waktu pencarian dan secara tidak langsung meningkatkan kualitas hasil pencarian dengan algoritma minimax, karena memungkinkan pencarian dengan *depth* semakin dalam.

VI. Ucapan Terima Kasih

Penulis mengucapkan terima kasih kepada Bu Dr. Nur Ulfa Maulidevi, S.T., M.Sc., Bu Dr. Masayu Leylia Khodra, S.T., M.T., dan Bapak Dr. Ir. Rinaldi Munir, M.T. sebagai dosen mata kuliah Strategi Algoritma. Penulis juga mengucapkan terima kasih kepada seluruh penulis yang tulisannya dijadikan referensi untuk makalah ini. Terakhir, penulis berterima kasih kepada orang tua dan teman-teman atas dukungannya dalam menyelesaikan makalah ini.

Referensi

[1] Munir, Rinaldi. 2009. *Diktat Kuliah Strategi Algoritma*. Bandung: Program Studi Teknik Informatika Institut Teknologi Bandung.
[2] https://github.com/devinalvaro/yachess/ diakses pada 16 Mei 2017.
[3] http://www.mychess.de/ChessNotation.htm/ diakses pada 17 Mei 2017.

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, 17 Mei 2017

Devin Alvaro