Kecerdasan Buatan Permainan Catur dengan *Minimax* dan *Alpha-Beta Pruning*

Sebuah Pendekatan Menggunakan Paradigma *Depth-First Search* dan *Backtracking*

Asif Hummam Rais—13517099
Teknik Informatika, Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Bandung, Jawa Barat, Indonesia
hashshura@gmail.com

***Abstrak*—"Jika saya berhasil membuat kecerdasan buatan yang mengalahkan saya sendiri, apakah artinya saya cerdas atau saya tidak cerdas?" Catur adalah permainan strategi untuk dua orang di mana setiap pemain bergantian memindahkan buah catur milik masing-masing kubu dengan tujuan "memakan" buah catur raja kubu lawan. Makalah ini membahas tentang penggunaan algoritme *minimax* dan *alpha-beta pruning* untuk menemukan gerakan terbaik untuk posisi tertentu: menjadi kecerdasan buatan untuk memenangkan permainan catur yang diperkuat oleh bantuan fungsi evaluasi heuristik. Dengan representasi catur berbentuk matriks petak catur, dapat dibuat seluruh kemungkinan keadaan permainan yang ada yang terhubung oleh langkah pemain dalam bentuk pohon.**

***Kata kunci*—catur; minimax; alpha-beta pruning; kecerdasan buatan; heuristik**

I. PENDAHULUAN

Pada Perang Dunia Kedua, seorang ilmuwan berkebangsaan Inggris Alan Turing memecahkan kode 'Enigma' yang digunakan oleh pasukan Jerman untuk mengirimkan pesan rahasia dengan aman. Alan Turing dan timnya menciptakan mesin Bombe yang digunakan untuk menguraikan pesan-pesan Enigma, yang akhirnya menjadi fondasi utama mengenai Pembelajaran Mesin. Menurut Turing, sebuah mesin yang dapat berbicara dengan manusia tanpa dia menyadari bahwa itu adalah mesin akan memenangkan "permainan imitasi" dan dapat dikatakan mesin yang "cerdas". Tahun 1956, ilmuwan komputer berkebangsaan Amerika John McCarthy mengadakan Konferensi Dartmouth, di mana konsep 'Kecerdasan Buatan' (*artificial intelligence* atau AI) pertama kali diadopsi. Pusat-pusat penelitian mulai bermunculan di seluruh Amerika Serikat untuk menjelajahi potensi dari kecerdasan buatan. Peneliti Allen Newell dan Herbert Simon berperan penting dalam mempromosikan kecerdasan buatan sebagai bidang ilmu komputer yang dapat mengubah dunia.

Pada tahun 1985, pengembangan komputer yang dapat memainkan catur oleh IBM bernama Deep Blue dimulai sebagai proyek ChipTest di Universitas Carnegie Mellon. Kepopuleran Deep Blue melonjak ketika Garry Kasparov, seorang *grandmaster* catur dan juara catur dunia asal Russia, melakukan pertandingan catur dengannya. Deep Blue dan Kasparov mengalahkan satu-sama-lain pada dua kesempatan yang berbeda. Pertandingan pertama berjalan pada 10 Februari 1996, di mana Deep Blue menjadi mesin pertama yang memenangkan permainan catur melawan juara dunia catur dalam *time control* yang reguler. Namun, Kasparov memenangkan tiga permainan selanjutnya dan seri pada dua permainan sisanya, memenangkan permainan keseluruhan dengan skor 4–2. Deep Blue kemudian dilakukan peningkatan yang masif (dengan *nickname* "Deeper Blue") untuk melawan Kasparov kembali pada Mei 1997, yang pada akhirnya dimenangkan oleh Deep Blue dengan skor 3½–2½.

Gambar 1. Kekalahan Garry Kasparov pada permainan Mei 1997

Desain Deep Blue pada dasarnya adalah penggunaan *chip* VLSI khusus untuk menjalankan pencarian *alpha-beta* (*alpha-beta pruning*) secara paralel, sebuah contoh dari GOFAI (*Good Old-Fashioned Artificial Intelligence*), bukan *deep learning* yang baru muncul pada dekade selanjutnya. Pendekatan yang dilakukan adalah *greedy* walaupun pencarian yang digunakan menggunakan paradigma *brute force* (dengan *depth-first search*). Ilmuwan komputer meyakini bahwa permainan catur adalah metriks yang cocok untuk menilai seberapa baik sebuah kecerdasan buatan itu, dan penggunaan pendekatan *greedy* yang mampu mengalahkan juara dunia catur oleh IBM membuktikan bahwa bahkan mesin yang mengimplementasi *brute-force* sekalipun sudah merupakan mesin yang "cerdas".

Dalam makalah ini, penulis akan membahas kembali mengenai pengembangan kecerdasan buatan menggunakan paradigma yang sama—*depth-first search*, *greedy*, dan *backtracking*—namun pendekatan yang sedikit berbeda: target dari bahasan kali ini adalah sistem kecerdasan buatan yang sederhana dalam artian kurang dari 200 kode baris beserta analisis kompleksitasnya.

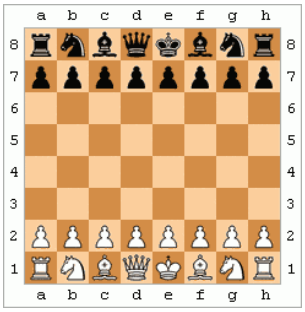
Konsep dasar yang ada pada makalah ini berpusat tentang fungsi evaluasi keadaan papan permainan, algoritme *minimax*, dan pencarian *alpha-beta*. Pada setiap langkah pengembangan kecerdasan buatan, akan dibandingkan bagaimana setiap dari konsep yang digunakan akan berpengaruh pada gaya permainan kecerdasan buatan itu sendiri.

II. Dasar Teori

A. *Permainan Catur*

Catur adalah permainan strategi dimainkan oleh dua pemain menggunakan papan berukuran 8x8 sebagai posisi tempat meletakkan buah-buah catur. Setiap pemain diberikan pasukan dalam bentuk 16 buah-buah catur dengan warna berbeda untuk masing-masing pemain—hitam dan putih—yang terdiri atas satu raja, satu menteri (biasa disebut ratu), dua benteng, dua kuda, dua gajah, dan delapan bidak.

Setiap gerakan pada pemain akan melakukan pergerakan terhadap satu buah catur yang memindahkan buah caturnya ke posisi yang mungkin didatangi oleh buah catur tanpa melompati bidak catur lain. Jika terdapat buah catur milik lawan pada posisi tujuan tersebut, akan terjadi "memakan"—yaitu menghilangkan atau mematikan buah catur milik lawan dari permainan. Masing-masing buah catur memiliki pergerakan yang berbeda. Pemain dengan warna putih akan melakukan gerakan pertama, dilanjutkan oleh pemain dengan warna hitam, kembali ke pemain warna putih, dan seterusnya sampai buah catur raja dari salah satu pemain ada yang tidak dapat mengelak dari keadaan akan termakan yang biasa disebut *skakmat*, ada salah satu pemain yang menyerah, atau kedua pemain menyatakan permainan berhenti dengan keadaan seri.



Gambar 2. Posisi awal permainan catur (sebelum ada pergerakan pemain).

Aturan pergerakan setiap buah catur didefinisikan sebagai

1. Raja (♔ ♚) hanya dapat bergerak sejauh satu petak ke 8 petak terdekat dari tempat dia singgah (lurus dan diagonal). Raja juga dapat melakukan *rokade* atau *castling* dengan benteng, yaitu memindah raja 2-3 petak ke arah benteng lalu benteng berpindah melompati raja ke belakangnya, dengan syarat raja dan benteng belum pernah bergerak, tidak ada buah catur apapun di tengahnya, dan raja tidak sedang diskak.
2. Menteri (♕ ♛) dapat bergerak ke delapan arah (lurus dan diagonal) tanpa batasan jauh petak selama tidak melompati buah catur lain.
3. Benteng (♖ ♜) dapat bergerak ke empat arah (lurus) tanpa batasan jauh petak selama tidak melompati buah catur lain.
4. Kuda (♘ ♞) dapat bergerak seperti huruf L, yaitu memanjang dua petak dan melebar satu petak, serta dapat melompati buah catur lain.
5. Gajah (♗ ♝) dapat bergerak ke empat arah (diagonal) tanpa batasan jauh petak selama tidak melompati buah catur lain.
6. Bidak (♙ ♟) dapat bergerak ke satu arah (maju ke arah lawan) satu atau dua petak pada gerakan pertama dan hanya satu petak untuk gerakan selanjutnya. Bidak dapat memakan sambil bergerak ke arah diagonal di depannya jika ada buah catur lawan. Bidak juga dapat melakukan *en passant*, yaitu memakan bidak lawan di sebelahnya yang baru saja bergerak dua petak lalu berpindah ke depan bidak tersebut. Bidak juga dapat melakukan *promote*, yaitu berubah menjadi buah catur lain jika berada pada baris terujung.

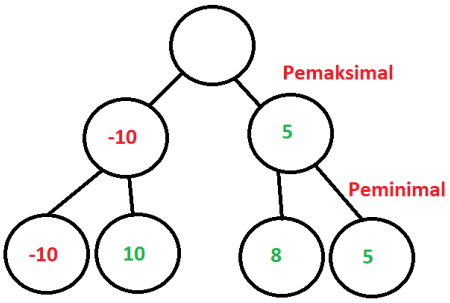
B. *Struktur Data Pohon*

Dalam ilmu komputer, pohon adalah struktur data abstrak yang mensimulasikan struktur hierarkikal dari pohon yang mengandung sebuah akar (*root*) dan subpohon berupa anak-anak (*children*) dari sebuah simpul orang tua (*parent*) yang direpresentasikan oleh kumpulan dari simpul terhubung. Struktur data pohon dapat didefinisikan secara rekursif sebagai sekumpulan dari simpul (yang dimuali dari akar), di mana setiap simpul adalah sebuah struktur data yang mengandung sebuah nilai, bersamaan dengan daftar referensi menuju simpul-simpul "anak", dengan konstrain yang tidak duplikat dan tidak menuju akar.

C. *Algoritme Minimax pada Teori Permainan*

Minimax adalah algoritme sejenis *backtracking* yang digunakan dalam pemilihan keputusan dan teori permainan untuk mencari pergerakan optimal dari seorang pemain, dengan membawa asumsi bahwa pemain lawan juga bermain dengan optimal.

Dalam algoritme *minimax*, kedua pemain akan disebut pemaksimal dan peminimal. Pemaksimal akan mencari skor tertinggi yang mungkin didapat, sedangkan peminimal akan mencari skor minimum yang mungkin didapat. Setiap keadaan pada papan permainan (*board state*) memiliki nilai yang diasosiasikan kepadanya. Pada *state* tertentu jika pemaksimal sedang berada dalam status "menang" maka nilai tersebut biasanya akan diisi oleh nilai positif, sedangkan jika peminimal yang berada dalam status "menang" maka nilai tersebut diisi oleh nilai negatif. Cara secara spesifik untuk menghitung nilainya adalah suatu fungsi heuristik yang unik untuk tiap-tiap permainan tertentu.



Gambar 3. Contoh pohon dalam algoritme *minimax*.

Sebagai contoh, jika terdapat sebuah permainan dengan empat *state* akhir seperti gambar di atas, maka jika pemaksimal memilih kiri pada gerakan pertama, tentu peminimal akan memilih simpul dengan nilai -10. Jika pemaksimal memilih kanan, peminimal akan memilih simpul dengan nilai 5. Maka, nilai dari simpul yang ada pada pemaksimal adalah -10 dan 5 untuk simpul kiri dan kanan. Karena pemaksimal mencari nilai simpul yang maksimum, maka gerakan terbaik yang harus dia pilih adalah **ke kanan**.

D. *Pencarian Alpha-Beta*

Pencarian *alpha-beta* adalah algoritme pencarian yang bertujuan untuk mengurangi jumlah simpul yang akan dievaluasi oleh algoritme *minimax* dalam pohon pencariannya. Pencarian ini akan memberhentikan proses evaluasi terhadap suatu gerakan jika setidaknya satu kemungkian sudah ditemukan yang dibuktikan lebih buruk kinerjanya dibandingkan dengan gerakan yang sudah dicek sebelumnya. Karena gerakan itu sudah pasti bukan merupakan maksimum, maka tidak perlu dievaluasi lagi. Jika pencarian *alpha-beta* dipasangkan kepada pohon *minimax* yang standar, akan mengembalikan gerakan yang sama seperti *minimax* biasa, namun mengabaikan cabang-cabang yang tidak berpengaruh terhadap keputusan akhir sehingga waktu berjalan akan lebih cepat.

Ide dasar dari pencarian *alpha-beta* adalah bahwa algoritme ini menyimpan dua nilai: *alpha* dan *beta*, yang merepresentasikan skor minimum yang mungkin didapatkan oleh pemain pemaksimal dan skor maksimum yang mungkin didapatkan oleh pemain peminimal, secara terurut. Awalnya, *alpha* diatur sebagai negatif-tak-hingga dan *beta* diatur sebagai positif-tak-hingga, dengan artian kedua pemain memulai permainan dengan skor terburuk masing-masing. Ketika skor maksimum pemain peminimal (pemain *beta*) dijamin akan menjadi kurang dari skor minimum pemain pemaksimal (pemain *alpha*) yang terjamin (dengan kata lain, jika *beta* ≤ *alpha*), maka pemain pemaksimal tidak perlu mempertimbangkan langkah-langkah simpul keturunan (*children node*) dari simpul itu karena mereka tidak akan pernah dicapai pada permainan sesungguhnya (jika kedua pemain bermain dengan asumsi optimal).

III. Pengembangan Kecerdasan Buatan Catur

Mengulang yang disebutkan pada bab pendahuluan, bab ini akan terbagi menjadi penentuan fungsi evaluasi keadaan papan permainan, aplikasi algoritme *minimax*, dan perbaikan lebih lanjut menggunakan pencarian *alpha-beta*.

Implementasi sistem permainan dari catur akan diabaikan pada makalah ini karena hanya akan membahas metode pengembangan kecerdasan buatannya. Diasumsikan sudah tersedia sebuah kelas **Board** yang berisi keadaan papan, dengan memiliki fungsi **getPossibleMoves** yang mengembalikan langkah-langkah yang terhubung (berupa **Board**) dan **getBoardChessPieces** yang mengembalikan matriks kotak papan catur yang berisi informasi bidak catur (**Raja**, **Menteri**, dll) dan warna dari bidak catur tersebut.

Sebagai referensi, penulis menggunakan bahasa pemrogaman JavaScript 1.8.5 untuk melakukan pengembangan kecerdasan buatan ini dan menggunakan *library* chess.js untuk membuat obyek permainan dan mengatur sistem permainan catur dan chessboard.js untuk memvisualisasikan keadaan papan permainan catur.

A. *Fungsi Evaluasi Heuristik Keadaan Papan Permainan*

Karena algoritme *minimax* yang akan dirancang terdiri atas banyak simpul berupa keadaan papan permainan catur, nilai dari simpul yang akan menjadi pembanding untuk pemain pemaksimal dan pemain peminimal harus didefinisikan oleh sebuah fungsi heuristik. Nilai ini juga yang menentukan pemain mana yang saat ini (pada keadaan tersebut) lebih "menang" (*upper-hand*) dari pemain lawan.

Penulis menyediakan dua ide fungsi heuristik. Ide pertama adalah ide yang kurang "mangkus" namun cukup masuk akal sebagai fungsi heuristik yang akan digunakan untuk menentukan nilai simpul dari pohon *minimax*: memberikan nilai untuk setiap buah catur dengan nilai tertentu, lalu menjumlahkan seluruh nilai yang ada pada papan catur.

Buah Catur Pemain Putih	Nilai untuk Fungsi Eval	Buah Catur Pemain Hitam	Nilai untuk Fungsi Eval
♔ Raja	+900	♚ Raja	-900
♕ Menteri	+90	♛ Menteri	-90
♖ Benteng	+50	♜ Benteng	-50
♘ Kuda	+30	♞ Kuda	-30
♗ Gajah	+30	♝ Gajah	-30
♙ Bidak	+10	♟ Bidak	-10

Tabel 1. Nilai setiap buah catur untuk fungsi heuristik keadaan papan.

Tabel di atas diperoleh dengan anggapan heuristik bahwa satu raja sama harga dengan sepuluh menteri, satu menteri berharga dua benteng, dua benteng berharga lima kuda, satu kuda berharga satu gajah, dan satu gajah berharga dua pion.

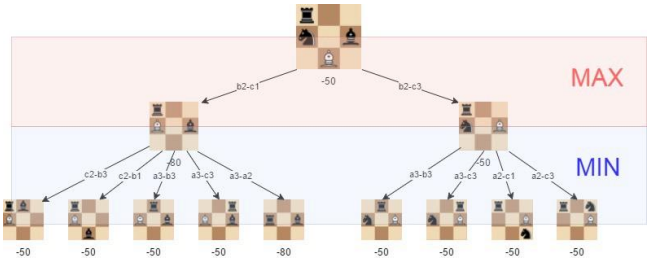
```
// PSEUDOCODE
evaluate(Board b):
 score <- 0
 for (piece in b.getBoardChessPieces):
  if (piece.type = Raja):    s <- 900
  if (piece.type = Menteri): s <- 90
  if (piece.type = Benteng): s <- 50
  if (piece.type = Kuda):    s <- 30
```

```
  if (piece.type = Gajah):    s <- 30
  if (piece.type = Bidak):  s <- 10
  if (piece.color = White): score += s
  if (piece.color = Black): score -= s
 -> score
```

B. *Aplikasi Algoritme Minimax*

Dengan keberadaan fungsi evaluasi heuristik, selanjutnya adalah merancang pohon pencarian di mana algoritme yang dibuat dapat memilih gerakan yang terbaik. Hal ini dilakukan oleh algoritme *minimax*.

Pada algoritme *minimax* yang akan dibuat, pohon rekursif dari keseluruhan gerakan akan ditelusuri sampai kedalaman tertentu, lalu posisinya akan dievaluasi pada *daun* terakhir dari pohon. Setelah itu, akan dikembalikan nilai tertinggi atau terendah dari simpul anak ke simpul orang tuanya, tergantung apakah saat itu adalah giliran pemain putih atau pemain hitam (mencoba untuk mencari nilai minimal atau maksimal dari setiap tingkat pohon).



Gambar 4. Pohon hasil bentukan algoritme *minimax*. Perhatikan bahwa gerakan paling optimal untuk pemain pemaksimal adalah b2-c3, dikarenakan apabila memilih b2-c1 makan pemain lawan akan memilih a3-a3 sehingga skor menjadi -80.

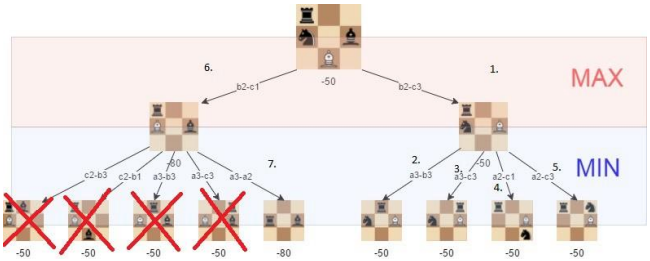
```
// PSEUDOCODE
miniMax(kedalaman, Board b, isPemaksimal):
 if (kedalaman = 0)
  -> -evaluateBoard(b)
 if (isPemaksimal):
  gerakanBaik = -9999
  for (n in b.getPossibleMoves):
   gerakanBaik = max(
    gerakanBaik,
    miniMax(kedalaman-1,n,!isPemaksimal)
   )
  -> gerakanBaik
 else:
  gerakanBaik = 9999
  for (n in b.getPossibleMoves):
   gerakanBaik = min(
    gerakanBaik,
    miniMax(kedalaman-1,n,!isPemaksimal)
   )
    -> gerakanBaik
```

Hasil dari pencarian algoritme *minimax* ini tentu sangat bergantung pada kedalaman pencarian yang akan dilakukan. Sedikit analisis menerangkan bahwa kompleksitas waktu algoritme *minimax* ini dalam notasi big-O adalah $O(n^d)$ di mana n adalah banyaknya gerakan yang mungkin dilakukan dan d adalah kedalaman dari pohon *minimax* yang dibentuk. Hal ini disebabkan karena setiap pemanggilan *getPossibleMoves* akan mengeluarkan n keadaan papan baru, yang mana keadaan seluruh papan itu akan ditelusuri lagi oleh fungsi *miniMax* dan setiap papan akan memanggil *getPossibleMoves* milik mereka masing-masing yang lain. Karena nilai tersebut dapat menjadi sangat besar, akan dilakukan optimisasi menggunakan pencarian *alpha-beta*.

C. *Pencarian Alpha-Beta*

Pencarian *alpha-beta* adalah metode optimisasi untuk algoritme *minimax* agar membuat program dapat mengabaikan beberapa cabang pada pohon pencarian tanpa mengubah arti dari *minimax* (dalam artian hasil yang dikeluarkan tidak akan berbeda). Menggunakan pencarian *alpha-beta* dapat membantu mengevaluasi pohon pencarian *minimax* menggunakan kedalaman yang lebih dalam, namun sumber yang digunakan lebih sedikit.

Pencarian *alpha-beta* didasarkan pada situasi ketika kita dapat berhenti mengevaluasi bagian dari pohon pencarian jika kita menemukan gerakan yang membawakan kepada keadaan yang lebih buruk daripada keadaan yang sudah ditemukan sebelumnya. Pencarian *alpha-beta* tidak mempengaruhi keluaran dari algoritme *minimax*, namun hanya membuatnya lebih cepat. Algoritme *alpha-beta* ini juga akan lebih efisien jika kita menemukan gerakan-gerakan yang baik di awal pencarian.



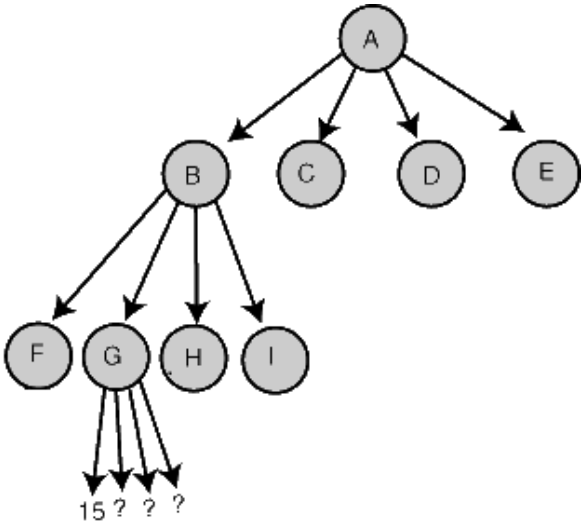
Gambar 5. Posisi yang tidak perlu dievaluasi jika menggunakan pencarian *alpha-beta* sebagai improvisasi dari *minimax*.

```
// PSEUDOCODE
miniMax(kedalaman, Board b, alpha, beta
isPemaksimal):
 if (kedalaman = 0)
  -> -evaluateBoard(b)
 if (isPemaksimal):
  gerakanBaik = -9999
  for (n in b.getPossibleMoves):
   gerakanBaik = max(
    gerakanBaik,
    miniMax(kedalaman-1,n,alpha,beta,
!isPemaksimal)
   )
   alpha = max(alpha, gerakanBaik)
   if (beta <= alpha):
    -> gerakanBaik
  -> gerakanBaik
```

```
 else:
  gerakanBaik = 9999
  for (n in b.getPossibleMoves):
   gerakanBaik = min(
    gerakanBaik,
    miniMax(kedalaman-1,n,alpha,beta,
!isPemaksimal)
   )
   beta = min(beta, gerakanBaik)
   if (beta <= alpha):
    -> gerakanBaik
  -> gerakanBaik
```

Hasil dari pencarian algoritme *minimax* dengan pendekatan improvisasi algoritme *alpha-beta* menyebabkan jumlah keadaan (posisi) yang harus ditelusuri berkurang secara signifikan. Uji coba secara lokal oleh penulis menjelaskan bahwa gerakan kecerdasan buatan yang dilakukan oleh algoritme *minimax* membutuhkan evaluasi terhadap sekitar 900,000 keadaan, sedangkan menggunakan penelusuran *alpha-beta* hanya membutuhkan evaluasi terhadap sekitar 60,000 keadaan.

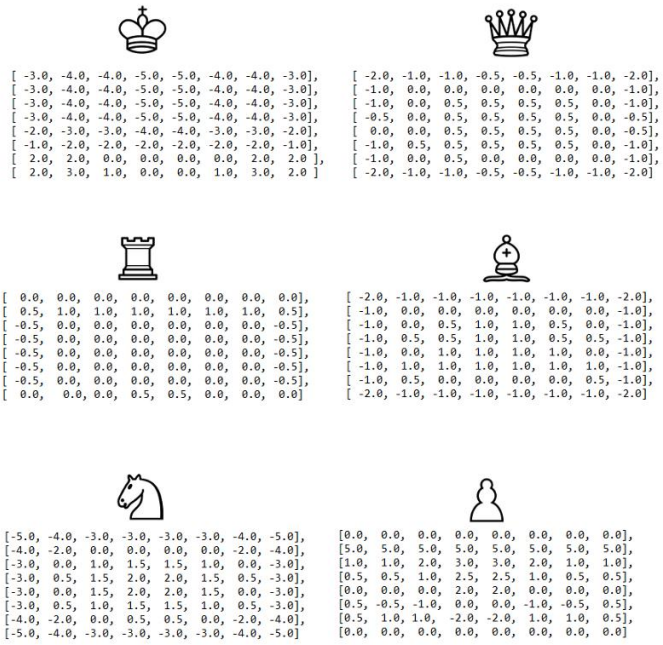
Hal ini disebabkan karena penelusuran *alpha-beta* dapat mengabaikan simpul-simpul yang sudah pasti lebih buruk. Asumsikan penelusuran memiliki konstan (atau rata-rata) percabangan *b* dengan kedalaman pencarian *d*. Jika pencarian yang dilakukan terurut secara terburuk, simpul yang dievaluasi adalah O(*b***b***b*...). Jika pencarian yang dilakukan terurut secara terbaik (langkah terbaik selalu dicari pertama), simpul yang dievaluasi adalah sekitar O(*b**1**b**1*…) atau "melewati" setiap dua level. Sehingga, *best-case* dari penelusuran *alpha-beta* adalah O($\sqrt{n^d}$), ketika setiap simpul yang ditelusuri pertama adalah simpul terbaik sehingga dapat mengabaikan seluruh simpul yang lain.



Gambar 6. Kasus *best-case* pada penelusuran *alpha-beta*. Perhatikan bahwa daripada menelusuri seluruh simpul seperti *minimax* (artinya menelusuri 4*4*4*4 simpul), *alpha-beta* dapat "mengabaikan" setiap dua level (hanya menelusuri (4*1*4*1).

D. Fungsi Evaluasi yang Lebih Baik

Fungsi evaluasi yang pertama dipakai cukup *naif* karena hanya menghitung material yang ditemukan di papan. Untuk meningkatkan ini, kita dapat menambahkan faktor evaluasi yang membedakan posisi buah catur di papan. Sebagai contoh, kuda di tengah lebih baik daripada di ujung karena memiliki lebih banyak opsi pergerakan.



Gambar 7. Skala penilaian per lokasi papan yang disarankan *chess programming wiki*.

IV. KESIMPULAN

Kekuatan dari algoritme permainan catur terletak pada bahwa ia tidak membuat kesalahan. Walaupun begitu, tetap saja pada dasarnya kecerdasan buatan itu tidak memiliki kepahaman terhadap strategi permainan catur yang lebih mendalam.

Dengan kode sumber yang baru saja dibuat, algoritme tersebut dapat memainkan catur dasar. Bagian "AI" pada catur pun dapat ditulis dengan kurang dari 200 baris kode, artinya konsep dasarnya memang sederhana untuk diimplementasi.

Beberapa peningkatan yang dapat lebih mengimprovisasi algoritme:

1. Pengurutan pergerakan agar pencarian *alpha-beta* dapat mencari gerakan terbaik pertama-tama.
2. Pembangkitan gerakan (*getPossibleMoves*) yang lebih cepat.
3. Evaluasi spesifik terhadap permainan akhir (misalnya mengevaluasi keadaan skakmat).

V. KESIMPULAN

Penulis ingin memberikan rasa terima kasih kepada Dr. Ir. Rinaldi Munir, M. T. sebagai dosen pengajar mata kuliah IF2211 Strategi Algoritma di kelas penulis. Penulis juga ingin memberi terima kasih kepada komunitas Stack Overlflow dan Free Code Camp yang sangat membantu terkait tutorial mengenai JavaScript serta *library* catur dan visualisasi catur. Penulis juga berterima kasih kepada rekan-rekan dan keluarga penulis yang selalu memberikan inspirasi dan semangat untuk tidak pernah menyerah.

VI. REFERENSI

[1] Russell, Stuart J.; Norvig, Peter (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Upper Saddle River, New Jersey: Pearson Education, Inc. p. 167. ISBN 0-13-604259-7.

[2] Knuth, D. E.; Moore, R. W. (1975). "An Analysis of Alpha–Beta Pruning" (PDF). Artificial Intelligence. 6 (4): 293–326. doi:10.1016/0004-3702(75)90019-3

[3] Chess, a subsection of chapter 25, Digital Computers Applied to Games, of Faster than Thought, ed. B. V. Bowden, Pitman, London (1953).

[4] Osborne, Martin J., and Ariel Rubinstein. A Course in Game Theory. Cambridge, MA: MIT, 1994. Print.

VII. 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, 26 April 2019

Asif Hummam Rais
13517099