Dokumentasi
Algoritma MiniMax Alpha Beta Pruning pada Knight Through

Deddy Chandra - 6181801007
Michael Sotaronggal Manurung - 6181801027
Bryan Heryanto - 6181801031

Jurusan Teknik Informatika
Universitas Katolik Parahyangan

1. Pendahuluan

1.1. Sejarah

Kuda dalam permainan catur pada sejarah seringkali dikenal dengan istilah '*Knight*'. Kuda sendiri dalam permainan catur ini disebut unik karena gerakannya yang lebih kompleks dibanding posisi lainnya seperti pion, ratu, menteri, dll. Gerakan tersebut disebut kompleks karena berbentuk L (dua kotak dalam satu arah dan berbelok pada kotak ketiga).

Keunikan lainnya yaitu kuda merupakan satu-satunya bidak dalam catur yang bisa melompati bidak lainnya. *Through* dalam bahasa Indonesia berarti melalui sedangkan *knight* adalah ksatria.

1.2. Tujuan

Membuat sebuah *AI* dengan menggunakan algoritma *Minimax* dan heuristik dengan *value* tertentu yang dapat menghitung dan megambil keputusan untuk melakukan langkah yang terbaik untuk melawan player musuh (UCT, *Alpha-Beta*, *Random*, *Human*, dan lain-lain).

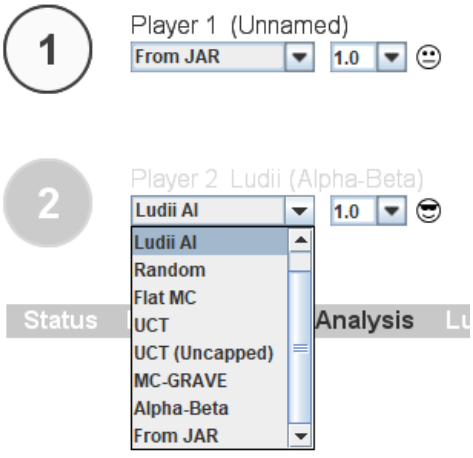
AI (*Artificial Intelligence)* atau sering dikenal dengan agen cerdas ini adalah suatu kecerdasan buatan yang dibuat supaya dapat menafsirkan data eksternal yang benar, belajar data dari data tersebut dan menggunakan pembelajaran tersebut guna mencapai tujuan dan tugas tertentu melalui adaptasi yang fleksibel.

2. Dasar Teori

2.1. Peraturan *Game Knight Through*

Berikut adalah peraturan-peraturan yang digunakan dalam bermain *Knight Through* di Ludii:

- Permainan ini membutuhkan 2 *player*, dapat berupa orang atau komputer.



Gambar 2.1.a

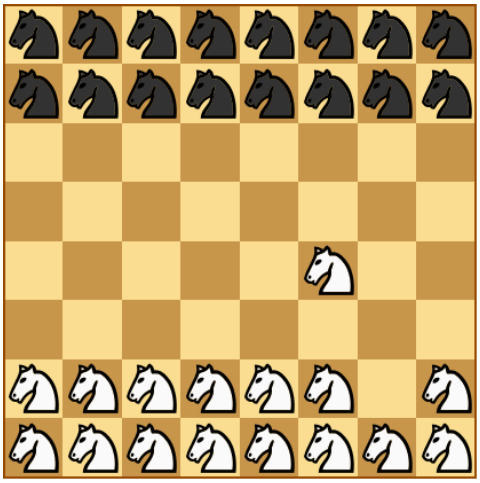
Pada gambar 2.1.a, ditunjukan terdapat 2 player, dimana di sana terdapat beberapa pilihan, dapat berupa Ludii AI, Random, dan lain-lain yang memilih AI yang dijalankan oleh player tersebut.

- *Player* memilih salah satu dari kedua warna, yaitu hitam atau putih yang nantinya akan menentukan siapa yang akan melangkah terlebih dahulu.

Gambar 2.1.b

Pada gambar 2.1.b, terdapat dua buah kuda dengan warna hitam dan putih.

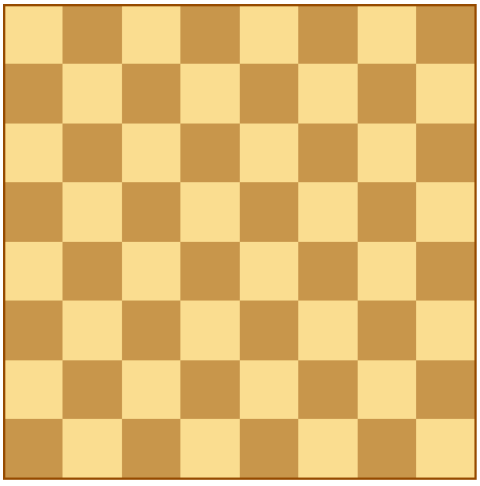
- *Player* dengan kuda berwarna putih dapat melakukan langkah pertama terlebih dahulu.



Gambar 2.1.c

Pada gambar 2.1.c, kuda berwarna putih akan melakukan langkah pertama ketika permainan dimulai.

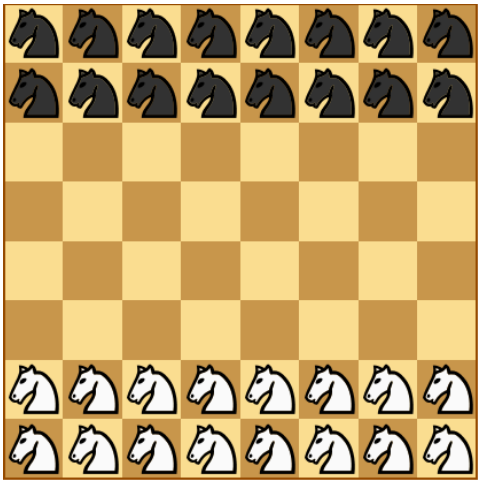
- Semua *moves* dilakukan pada *board* berukuran 8 x 8 *cell*. *Player* tidak bisa melakukan *move* yang membuat kuda keluar dari *board*.



Gambar 2.1.d

Pada gambar 2.1.d, ditampilkan bentuk dari papan yang akan digunakan untuk bermain game *Knight Through.*

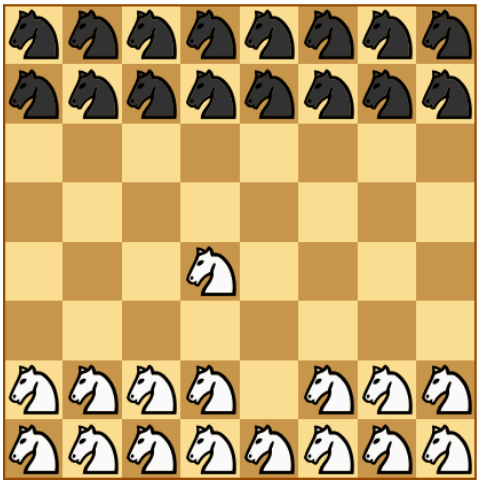
- Masing-masing *player* memiliki 16 kuda di posisi baris pertama dan baris kedua di dua buah ujung *board* player ketika permainan dimulai.



Gambar 2.1.e

Pada gambar 2.1.e, menampilkan posisi awal ketika *game Knight Through* dimulai.

- *Player* menggerakan kuda (melakukan *move*) secara bergiliran .



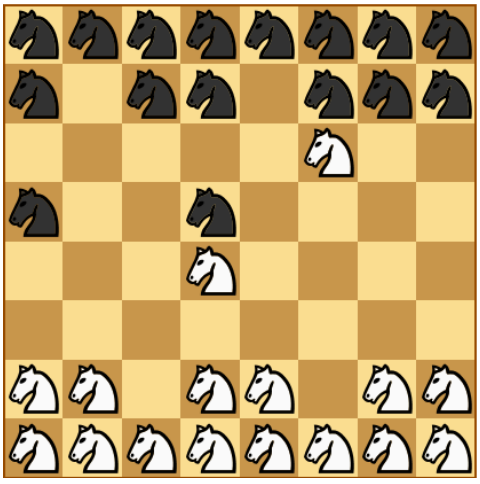
Gambar 2.1.f

Pada gambar 2.1.f, kuda berwarna putih melakukan *move* pertama.

Gambar 2.1.g

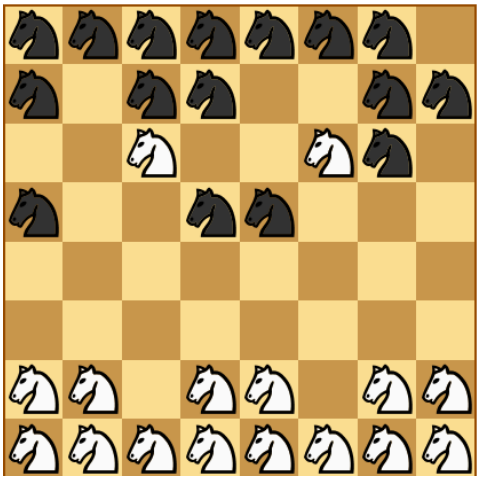
Pada gambar 2.1.g, merupakan giliran selanjutnya diganti dengan kuda berwarna hitam melakukan *move*.

- *Player* harus melakukan gerakan dalam setiap giliran.
- Kuda dapat melangkahi kuda lainnya.



Gambar 2.1.h

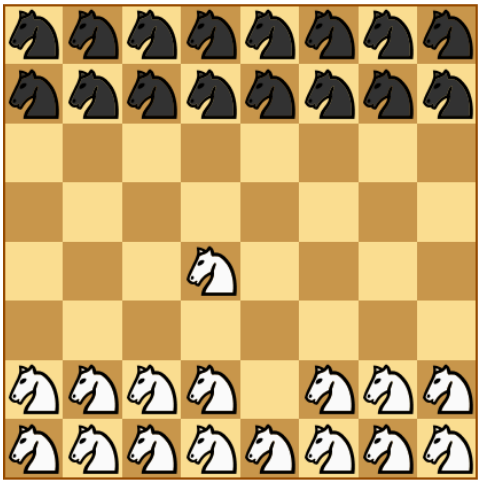
Pada gambar 2.1.h, diperlihatkan suatu *state* pada permainan game *Knight Through.*



Gambar 2.1.i

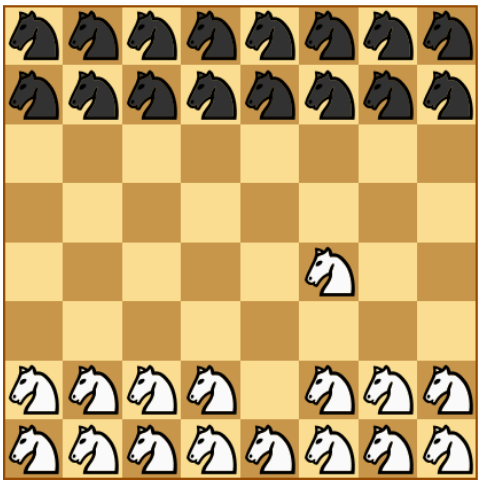
Pada gambar 2.1.i, diperlihatkan *state* setelah melakukan beberapa gerakan dari state gambar 2.1.h, dapat dilihat terdapat beberapa kuda yang melangkahi kuda di depannya.

- *Move* pada kuda berbentuk L (2 kotak maju dan 1 kotak ke kiri/kanan atau 2 kotak ke kiri/kanan dan 1 kotak maju).



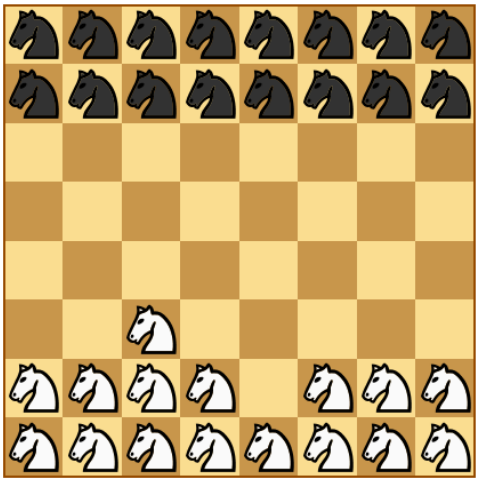
Gambar 2.1.j

Pada gambar 2.1.j, kuda melakukan 2 langkah maju dan 1 langkah ke kiri.



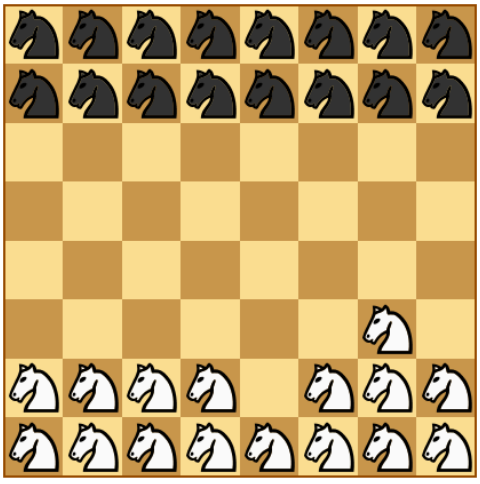
Gambar 2.1.k

Pada gambar 2.1.k, Kuda melakukan 2 langkah maju dan 1 langkah ke kanan.



Gambar 2.1.l

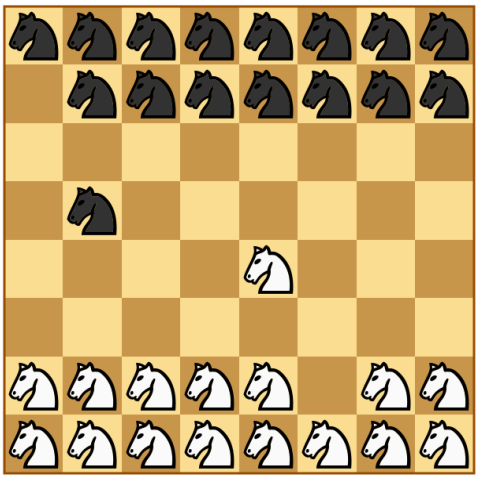
Pada gambar 2.1.l, kuda melakukan 2 langkah ke kiri dan 1 langkah maju.



Gambar 2.1.m

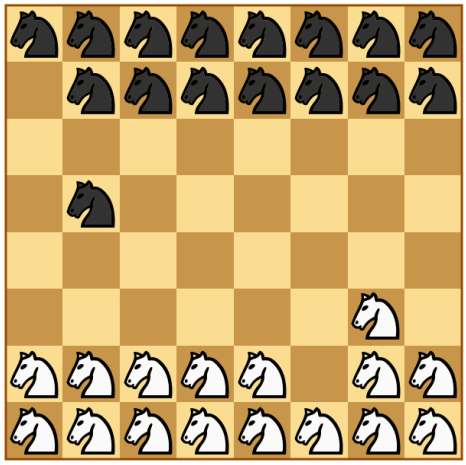
Pada gambar 2.1.m, kuda melakukan 2 langkah ke kanan dan 1 langkah maju.

- Kuda tidak dapat melakukan langkah mundur.



Gambar 2.1.n

Pada gambar 2.1.n, posisi kuda sudah bergerak melangkah kedepan.



Gambar 2.1.o

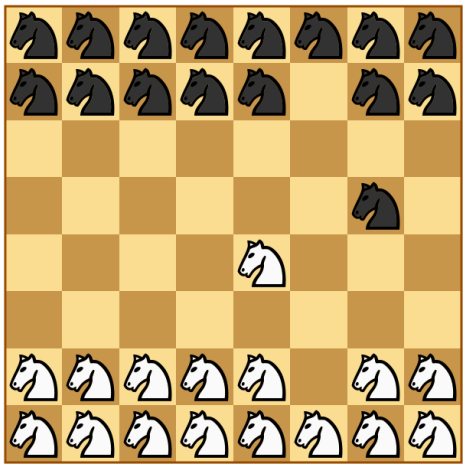
Pada gambar 2.1.o, posisi kuda setelah mencoba melakukan langkah yang salah karena mundur 1 baris.

That is not a valid move.

Gambar 2.1.p

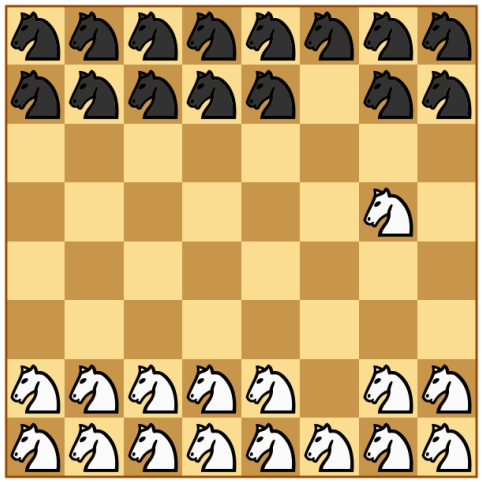
Pada Gambar 2.1.p, jika *user* mencoba melakukan gerakan mundur seperti dari 2.1.n menjadi 2.1.o maka akan muncul status di Ludii bahwa *move* tidak dapat dilakukan.

- *Player* hanya bisa memakan kuda lawan dan tidak bisa bergerak menduduki atau memakan kuda sendiri.



Gambar 2.1.q

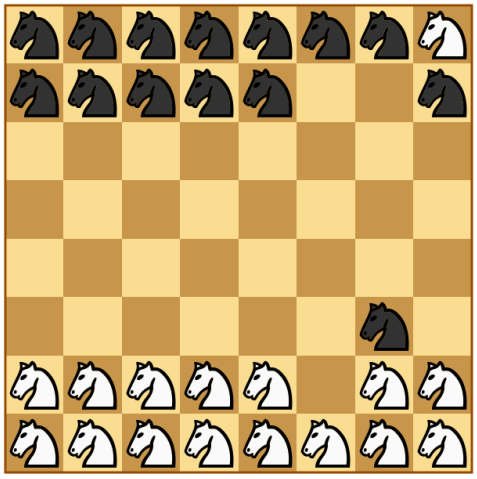
Pada gambar 2.1.q adalah posisi awal kuda putih sebelum memakan kuda hitam.



Gambar 2.1.r

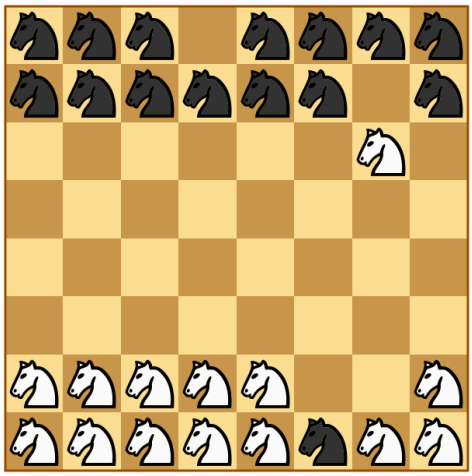
Pada gambar 2.1.r, posisi akhir kuda putih dari 2.1.q saat memakan kuda hitam, posisinya menduduki kuda hitam sebelumnya seperti di gambar 2.1.r.

- *Player* yang pertama kali berhasil menduduki posisi baris pertama lawan adalah pemenangnya.



Gambar 2.1.s

Pada gambar 2.1.s, posisi terminal saat kuda putih menang, jika salah satu kuda putih sudah mencapai baris pertama kuda hitam.



Gambar 2.1.t

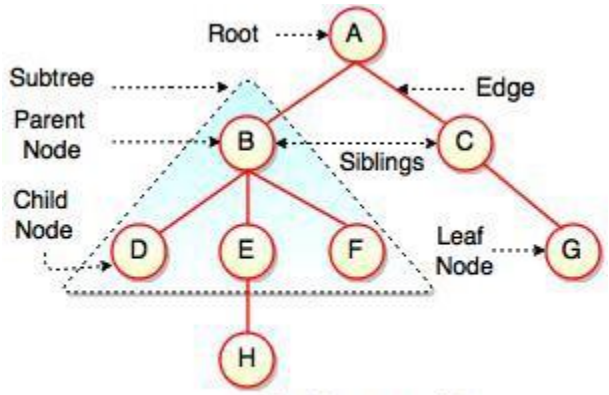
Pada gambar 2.1.t, posisi *terminal* saat kuda hitam menang, jika salah satu kuda hitam sudah mencapai baris pertama kuda putih.

2.2. Penjelasan Terminologi atau Istilah

Sebelum melanjutkan berikut adalah beberapa daftar serta penjelasan singkat istilah yang akan sering digunakan di dalam dokumentasi ini:

- *node*(simpul) : Sebuah unit yang biasanya digunakan dalam sebuah *graph* atau *tree(graph* terhubung yang tidak memiliki *circuit*). Biasanya unit tersebut direpresentasikan dengan sebuah titik. *Node* bisa dihubungkan dengan satu atau beberapa buah *edge*(sisi). Dalam Knight Through node ini dapat digunakan untuk menyimpan suatu *state* atau value.
- *state* : kondisi dari sebuah sistem ketika dilakukannya interaksi terhadapnya. Dalam *game* Knight Through salah satu *state* yang bisa didapat adalah cell dimana letak kuda dengan warna tertentu berada dan cell yang tidak ditempati kuda.
- agen : menurut https://www.britannica.com/technology/agent agen adalah sebuah program knighkomputer yang melakukan aksi secara kontinu dan otomatis yang bekerja atas nama atau merepresentasikan suatu individu atau organisasi. Di Knight Through sendiri bisa dibilang bahwa agen adalah *player* yang memainkan sebuah *game*.
- *move* : Suatu aksi yang dilakukan oleh agen untuk melakukan perubahan terhadap sebuah *state*.
- *search tree* : Suatu *tree* yang berisi *node-node* yang merepresentasikan suatu state dan *value* tertentu. Digunakan untuk melakukan sebuah pencarian.
- *root* : *node* yang berada pertama ketika sebuah *tree* dibentuk, *node* ini tidak memiliki *parent node*.
- *depth* : Kedalaman dari suatu *tree* diukur dari *root*-nya.
- *parent node* : Merupakan *node* yang dihubungkan dengan sebuah sisi dengan *node* yang menjadi patokannya. *Depth* dari *node* ini berada di *depth*-1 dari *node* yang menjadi patokannya.
- *child node* : Merupakan *node* yang dihubungkan dengan sebuah sisi dengan *node* yang menjadi patokannya. *Depth* dari *node* ini berada di *depth*+1 dari *node* yang menjadi patokannya.

- *leaf node* : Merupakan sebuah *node* yang sudah tidak memiliki *child node*.



Gambar 2.2.a Berikut adalah gambar relasi dari *node* dalam sebuah *tree* yang kami ambil dari https://medium.com/@kamalovotash/nodes-in-trees-data-structure-c354e98b42d5.

Sekian, beberapa penjelasan terminologi yang mungkin membantu pembaca untuk mengerti dokumentasi ini lebih dalam. Untuk menghindari ambiguitas, diharapkan terlebih dahulu pembaca mengerti istilah - istilah yang umum digunakan dalam *graph* dan sistem cerdas.

2.3. Algoritma Minimax Alpha Beta Pruning

Minimax Alpha-Beta Pruning adalah sejenis algoritma *backtracking* yang digunakan dalam *decision making* dan *game theory.* Algoritma ini bertujuan untuk mencari *move* optimal bagi player dengan asumsi player lawan bermain secara optimal juga. Dalam *Minimax* ada 2 *player* yang disebut *maximizer* dan *minimizer*. *Maximizer* adalah *player* yang mencoba mendapat nilai setinggi mungkin dan *minimizer* adalah *player* yang mencoba mendapat nilai seminimal mungkin.

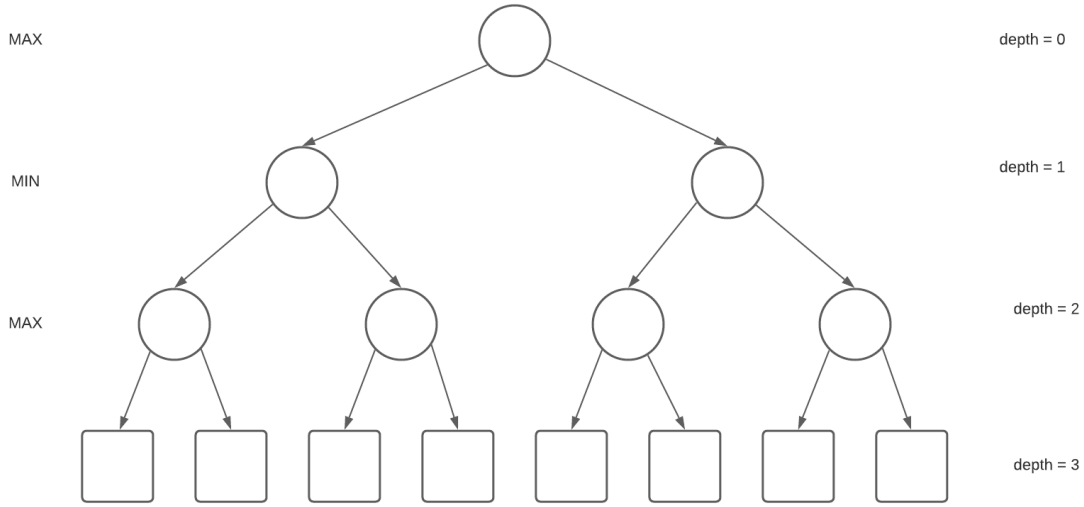
Referensi: Geeks for Geeks

Minimax adalah algoritma yang dikembangkan dari *Minimax* dimana pada algoritma ini terdapat *pruning* dari *value* yang didapat, apabila *value* tersebut sudah dianggap tidak bisa berkembang menjadi lebih baik lagi . Di dalam *Alpha-Beta Pruning* setiap *node* akan memiliki 2 buah nilai yaitu nilai *Alpha* dan nilai *Beta*. *Alpha* akan menyimpan nilai maksimum pada *node* dan nilai *Beta* akan menyimpan nilai minimum pada sebuah *node*. Pertama, semua nilai *alpha* pada *node* akan bernilai minus *infinity* dan untuk nilai *Beta* akan bernilai maksimum *infinity*. Setelah itu, algoritma ini akan menelusuri anak kiri hingga mencapai suatu *leaf* terlebih dahulu. Selanjutnya algoritma akan masuk ke tahap *backtracking* dimana dalam tahap ini pencarian akan mundur secara rekursif sampai kembali ke *root*. Perlu diperhatikan sebelum mundur dari suatu *node*, *node* tersebut harus menyelesaikan terlebih dahulu iterasi untuk mendapatkan *value* dari *child node* tersebut.

Di tahap *backtracking*, *node* dengan *depth* genap(*node maximizer*/*alpha*) akan membandingkan nilai *alpha* nilai dengan setiap *child node*, jika nilai hasil kembalian *child node* lebih besar dari *alpha*, maka nilai pada *alpha* akan diubah menjadi nilai dari *child node* tersebut. Sebaliknya, jika pohon berada pada *depth* ganjil(*node minimizer*/*beta*), maka nilai *beta* akan dibandingkan dengan setiap *child node* dari *node*

tersebut. Jika nilai dari *child node* lebih kecil dari nilai *beta node* dari node tersebut, maka nilai *beta* akan digantikan dengan nilai dari *child node* nya.

Jika nilai *alpha* lebih besar atau sama dengan *beta* maka algoritma dapat melakukan *pruning* (pemotongan pada *edge*(sisi) dalam sebuah *node*) pada *siblings* lainnya sehingga dapat menghemat kompleksitas waktu pada saat menjalankan algoritma *Minimax*.



Gambar 2.3.a *Minimax*

Pada gambar 2.3.a *minimax* terdapat sebuah *root* dan *node-node* lainnya. *Node* paling bawah atau *leaf* berisi nilai sedangkan *node* yang bukan *leaf*, akan mengambil pilihan nilai tergantung pada ketinggian pohon, jika ketinggian pohon ganjil maka *node* tersebut akan memilih nilai maksimal dari anaknya, dan sebaliknya jika *node* genap maka *node* tersebut akan memilih nilai terkecil.

***Pseudo Code* Dari *Minimax Alpha-Beta Pruning*:**

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Kompleksitas waktu untuk algoritma ini adalah O(b^m) dan kompleksitas ruangnya adalah O(bm).
Referensi: PPT kelas PSC Adversarial Search Universitas Katolik Parahyangan oleh Lionov,PH.D.

2.4. Algoritma Cutting Off

Cutting off adalah sebuah pendekatan algoritma dalam hal mengontrol besarnya atau luasnya *search* agar besarnya berukuran *fixed* bukan dinamis. Cara ini bertujuan agar *depth*(kedalaman) dari *node* tidak terlalu dalam hingga melewati batas waktu yang diharapkan. Dalam implementasi kami pada saat melakukan *searching* ketika kedalaman sudah mencapai 5 yang ditentukan, maka algoritma akan mengembalikan nilai terbaik dari *searching* yang sudah dilakukan. Pendekatan ini bisa saja menghasilkan nilai yang buruk, karena bergantung pada hasil dari *evaluation function*.

Referensi: Halaman 129 Artificial Intelligence A Modern Approach oleh Stuart J. Russell and Peter Norvig.

2.5. Algoritma Move Ordering

Algoritma *Move Ordering* adalah algoritma melakukan seleksi terhadap setiap *move* yang didapatkan. Pada awalnya algoritma ini akan mensorting semua *move*-nya terlebih dahulu dengan metode dan berdasarkan heuristik tertentu. Setelah itu akan dipilih *move* dengan syarat pengambilan tertentu. Pada algoritma ini, Move Ordering diimplementasikan dengan cara mengambil 5 *move* terbaik, 14 *move random*, dan 1 *move* terburuk. Tujuan dari pengambilan *move-move* tersebut adalah agar algoritma ini tidak hanya berfokus pada *move-move* yang baik saja, dikarenakan *move* yang buruk atau *move* yang *random* juga berkemungkinan akan mendapatkan hasil yang lebih bagus ketika di jalankan.

3. Analisis masalah

3.1. Identifikasi Masalah

Dalam permainan *Knight Through* di Ludii, kami berniat untuk membuat agen cerdas yang dapat memilih *move* secara rasional agar *move* tersebut bisa membawa agen cerdas yang dibuat oleh kami tersebut ke dalam sebuah *state* kemenangan. Untuk itu diperlukan sebuah solusi untuk menyelesaikan masalah tersebut. Di awal analisis mungkin cara yang paling mudah dipikirkan adalah mencari menggunakan *random.* Sayangnya *random* bukanlah solusi yang baik dikarenakan nantinya tidak dapat diketahui baik atau buruknya suatu gerakan yang dilakukan agen berdasar hasil fungsi *random-nya*.

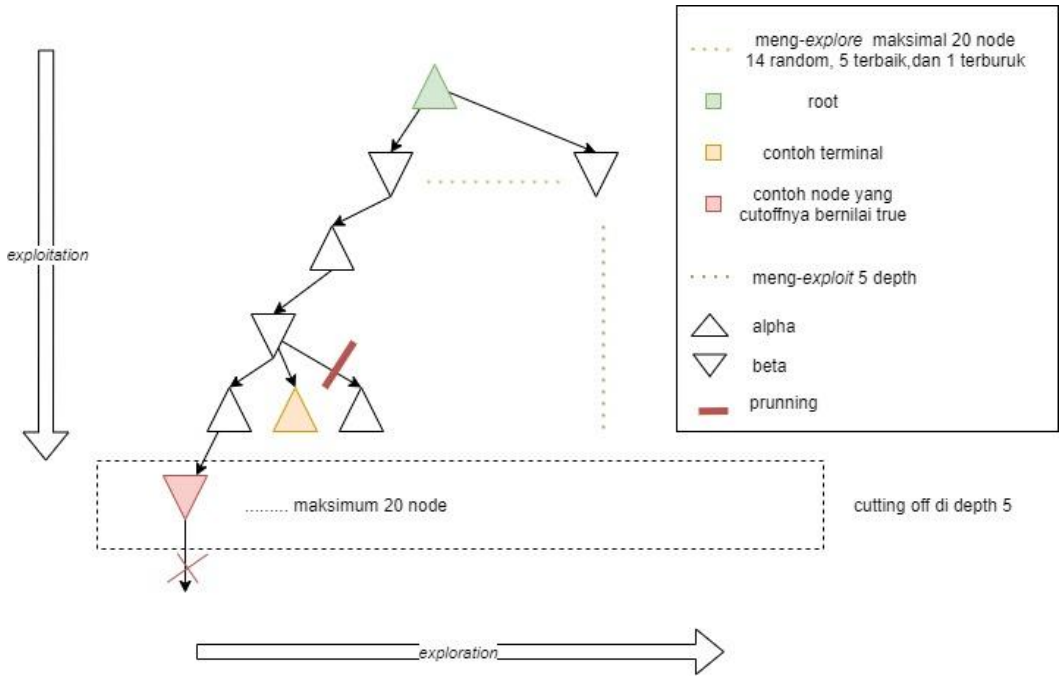
Telah diketahui *random* bukanlah cara yang tepat untuk menyelesaikan masalah kali ini. Berangkat dari situ, diperlukan suatu metode dimana agen dapat memilih *move*(gerakan) secara lebih optimal dan tepat. Solusi yang lebih baik pun diusulkan, dimana semua *state* bisa disimpan saat melakukan iterasi. Solusi ini adalah solusi yang tepat, namun komputer saat ini masih memiliki limitasi dari segi memori dan kecepatan komputasi untuk melakukan pencarian terhadap semua *state* yang mungkin.

Dari masalah pada paragraf sebelumnya, munculah ide yang lebih dikembangkan lagi yaitu *state* dari pencarian *move* yang dilakukan bisa dipotong. Namun permasalahan berikutnya muncul, yaitu memotong *state* yang sedang dicari tanpa petunjuk atau aturan menyebabkan pada akhirnya solusi yang didapat adalah hasil yang buruk juga. Sehingga diperlukan sebuah cara yang dapat memungkinkan mencari *move* seoptimal mungkin tanpa harus mencari dari semua *state* pada permainan.

Masalah berikutnya ialah bahwa lawan dari agen yang kami buat juga memiliki kecerdasan untuk memilih langkah-langkah secara rasional juga. Sehingga untuk mengalahkan lawan, kami harus membuat agen yang rasional dan lebih cerdas dibanding lawan kami.

3.2. Analisis Penyelesaian Masalah

Analisis Algoritma



Gambar 3.2.a Berikut adalah ilustrasi algoritma yang digunakan dalam analisis algoritma dibawah. Di dalamnya sudah terdapat *tag-tag* definisi yang dapat membantu penjelasan algoritma dibawah.

Di awal penelitian, untuk membantu mencari solusi optimum kami menggunakan algoritma standard *Minimax*. Namun, tentunya seperti telah dibahas diatas, algoritma *Minimax* ini bisa dipotong dengan menggunakan *Alpha-Beta Pruning*. Hal tersebut tentunya sangat standar dilakukan dalam pembuatan agen cerdas menggunakan *Minimax* untuk menangani kasus dimana jika *node* men-*exploit* suatu *child node* dimana nilai minimum dan maksimum sudah tidak dapat berkurang atau bertambah, maka *Minimax* yang menggunakan *Alpha-Beta Pruning* ini bisa langsung mem-*prune node* tersebut.

Namun ternyata pada implementasi yang kami lakukan, method *Minimax Alpha-Beta pruning* ini sendiri tidak bisa berjalan karena *search tree* yang dibuat oleh kami terlalu besar. Dikarenakan adanya masalah tersebut kami pada akhirnya memutuskan untuk mengambil inspirasi dari teknik *Cutting off Search*. Dengan adanya *Cutting off Search* pada algoritma kami, diputuskan suatu *node* bisa dinyatakan merupakan *Cutting off* apabila *node* tersebut sudah mencapai *depth* 5. Dari situ dapat disimpulkan bahwa dalam pencarian di algoritma ini suatu *node* tidak akan di-*exploit* lagi apabila *depth*-nya sudah mencapai 5 atau sudah mencapai terminal. Kami menentukan *depth* 5 didasarkan beberapa percobaan sebelumnya menggunakan 6, 7, dan 8. Namun dengan menggunakan *depth* 5 *exploration* jadi lebih seimbang ditandakan dengan tingkat kemenangan yang meningkat.

Minimax Alpha-Beta Pruning dengan *Cutting off* ini sudah cukup memuaskan. Namun pada *state* tertentu di *depth* 5, program dengan algoritma tersebut tidak dapat berjalan lagi pada komputer yang digunakan oleh kami dikarenakan adanya *memory error*. Hal tersebut disebabkan oleh ketidakmampuan komputer dalam menampung *search tree* yang masih besar. Tentunya, hal tersebut menjadi masalah karena disini kami berharap agar algoritma kami bisa menghandle sampai *depth* 5. Alasannya, agar harapannya bisa mengalahkan pencarian *Alpha-beta pruning* 1 s Ludii yang rata-rata *ber-depth* 4. Supaya

dapat menghandle hal ini, maka diperlukan suatu metode untuk dapat memotong pencarian ketika melakukan *exploration*.

Memotong pencarian pada *exploration* bisa saja mengambil ide lagi dari *Cutting off Search* yang dibuat sebelumnya yaitu dengan men-*set* suatu value yang *fix* untuk menjadi maksimal banyak-nya *exploration* yang dapat dilakukan dalam suatu *node*. Namun, apabila kami menggunakan teknik itu bisa jadi *node* yang sebenarnya sudah memiliki *value* yang baik tidak ikut terhitung ketika melakukan sebuah *search* tersebut. Untuk itu, diperlukan sebuah metode agar dapat menyimpan *node* dengan *value* yang sementara dianggap baik. Setelah dianalisis, ternyata menyimpan *node* yang dianggap baik tidak cukup. *Node* dengan *value* yang terbaik belum tentu merupakan pilihan terbaik untuk dilakukan *exploitation* terhadapnya maka kami juga mengambil *node* secara *random* dan *node* terburuk untuk menambah *variance* dari hasil algoritma tersebut. Ketika mengimplementasi kami menggunakan *sorting* untuk mengetahui mana *node* dengan *value* yang terbaik sehingga tahap ini disebut sebagai *Move Ordering*.

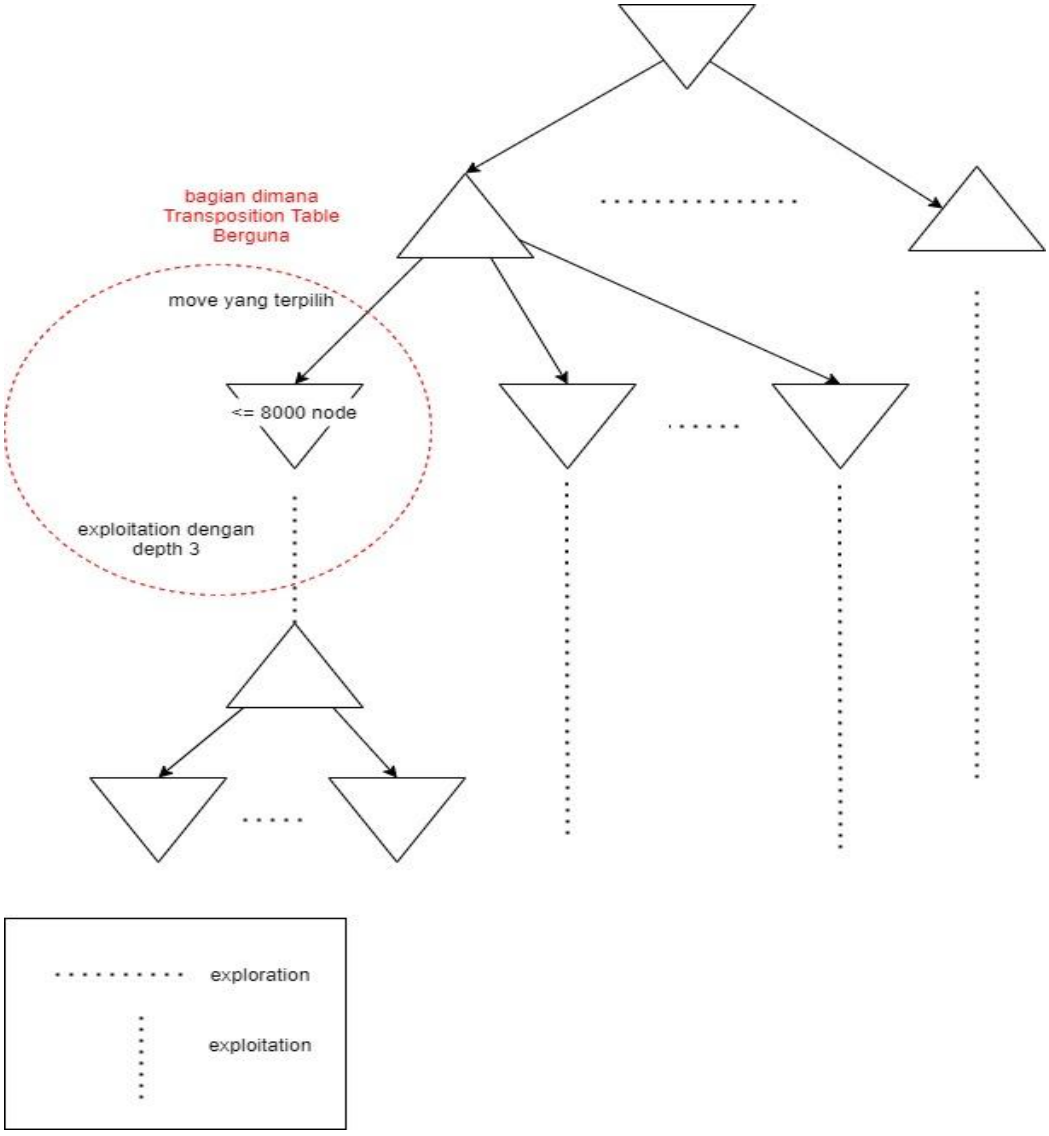
Teknik *Move Ordering* yang algoritma kami implementasikan memerlukan suatu heuristik untuk menghitung mana *node* dengan nilai atau bisa disebut *evaluation value* terbaiknya. Untuk itu, disini kami menyediakan sebuah mekanisme untuk menghitung heuristik tersebut di setiap *node* yang di *explore* oleh kami dimana nantinya di detailkan dalam bagian implementasi.

Transposition Table

Apabila Mencari dalam banyak *textbook* dan *paper*, banyak algoritma *Minimax* dengan *Alpha-Beta Pruning* yang diimplementasikan menggunakan *Transposition Table* untuk menangani kasus yang berulang. Namun, pada saat ini kami memutuskan untuk tidak memakai *Transposition Table* tersebut. Alasan utama kami tidak memakai *Transposition Table* adalah sedikitnya *state* berulang yang dimiliki oleh *game Knight Through* ini. Hal tersebut diakibatkan oleh tidak adanya *move* mundur dalam *game Knight Through*. Dari situ bisa disimpulkan apabila kuda sudah melakukan suatu *move*, *state* saat ini tidak akan pernah sampai lagi ke *state* sebelum melakukan *move* tersebut.

Kami menyadari bahwa dengan menggunakan *Transposition Table* akan tetap ada *move* berulang yang dapat dipercepat karena telah disimpan di dalamnya. Namun, dalam *game* ini dari suatu *state* agen tidak bisa membuat *state* kembali ke *state* yang sama setelah melakukan sebuah *move*. Sehingga dari situ dapat dilihat bahwa, *value node* tersimpan yang dapat kembali digunakan hanyalah *grand-child node*(*child node* dari *child node*) dengan *action move* yang terpilih oleh lawan dari agen tersebut. Sehingga, apabila dilihat lebih *detail*, jumlah *state* berulang pada *game* dimana algoritma diimplementasikan tidak terlalu banyak secara signifikan, sampai harus menyimpan *value*nya dalam *Transposition Table*.

Depth yang diambil oleh algoritma ini pun tidak terlalu dalam yaitu 5. Setelah dipilih menggunakan *Move Ordering* dalam sebuah *exploration*, sebuah *node* dapat mengeksplor dan hanya dapat mendapat value dari maksimal 20 *node*. Apabila menggunakan *Transposition Table* berdasarkan analisis tadi, maka perhitungan yang dapat dipercepat pada *node* berikutnya hanyalah perhitungan dari *root* sampai depth 3. Secara *worst case* kami dapat menghitung maksimal jumlah *node* yang dipercepat oleh *Transposition Table* yaitu 20^3 *nodes* atau 8000 *node* saja.



Gambar 3.2.b

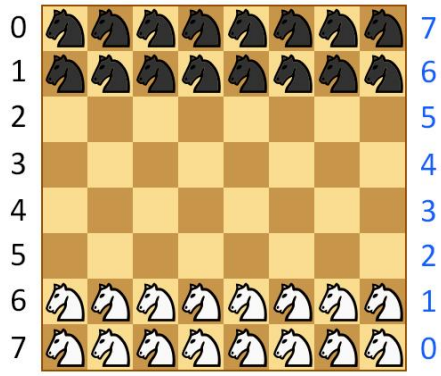
Pada Gambar 3.2.b terdapat garis putus-putus merah yang merupakan gambaran dari *node* tersimpan dalam *Transposition Table* yang terpakai dalam penentuan keputusan move berikutnya. Sedangkan garis putus-putus hitam merupakan penggambaran singkat node yang didapat ketika di-*explore* atau di-*exploit*.

Selain dari segi jumlah, algoritma kami juga sudah meng-implement sebuah method dinamakan kami `changeHeuristic()`. Dimana method tersebut nantinya bisa menghitung value dalam waktu konstan (kecuali di *node root*). Sehingga ketimbang harus mengecek di semua *node* apakah *value* tersebut sudah ada di *Transposition Table* atau belum yang pastinya juga memakan kompleksitas waktu untuk melakukan hashing, kami memilih menghitung saja secara langsung value di *node* tersebut dengan waktu yang sama-sama konstan.

Analisis heuristik

Dalam algoritma *Minimax Alpha-beta pruning heuristic evaluation function* bisa digunakan untuk menentukan kondisi *state* yang direpresentasikan di dalam *node* pada sebuah *search tree*. Heuristik

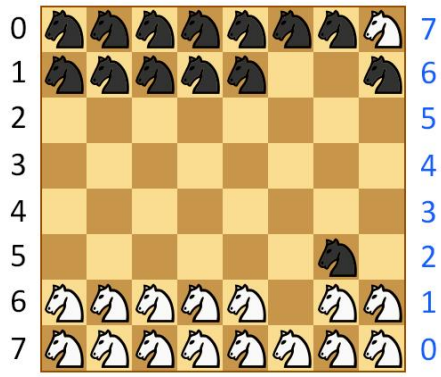
tersebut kami buat sendiri dengan ide yang didapat dari baris dimana kuda tersebut berada. Dimisalkan kami memberi angka 0-7 relative berdasarkan posisi awal masing-masing kuda sebagai indeks dari baris dimana sebuah kuda berada seperti pada gambar 3.2.c



Gambar 3.2.c

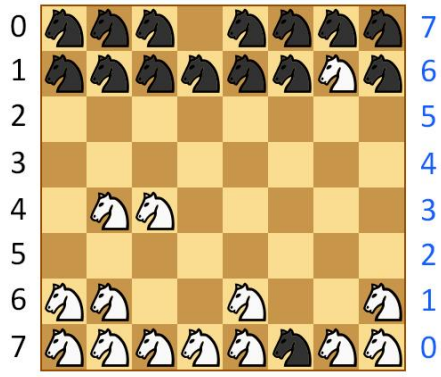
Pada gambar 3.2.c, menunjukan indeks baris yang disebutkan relatif terhadap warna. Tulisan berwarna hitam menjelaskan indeks baris untuk kuda yang hitam. Sedangkan tulisan berwarna biru menandakan indeks baris untuk kuda yang putih. Semakin dekat posisi kuda player pada wilayah musuh (semakin dekat dengan 7), maka kuda tersebut akan semakin dekat ke state kemenangan. Sedangkan, apabila player dengan warna berbeda mendekati wilayah player yang diharapkan menang(semakin dekat dengan 0) berarti player yang diharapkan menang tersebut berarti dekat dengan state kekalahan.

Dari situ, secara kasat mata kami dapat melihat bahwa semakin besar indeks dimana sebuah kuda berada, maka semakin dekat pula kuda tersebut ke *state* kemenangan. Bisa dilihat juga dari gambar 3.2.c, apabila sudah ada kuda yang mencapai indeks 7 berarti *player* dengan warna kuda yang mencapai 7 tersebut bisa dinyatakan sebagai pemenang. Sebaliknya apabila sudah ada kuda dengan warna berlawanan yang mencapai indeks 0, maka *player* dengan warna kuda berlawanan yang mencapai 0 tersebut dinyatakan sebagai pemenang dari *game* ini. Semua fakta-fakta tersebut membuat kami dapat menyimpulkan bahwa indeks baris dapat menjadi ide dasar dari *heuristic evaluation function* yang dimiliki oleh kami.



Gambar 3.2.d

Pada gambar 3.2.d, merepresentasikan posisi terminal saat kuda putih menang, jika salah satu kuda putih sudah mencapai baris pertama kuda hitam.



Gambar 3.2.g

Pada gambar 3.2.g, merepresentasikan posisi terminal saat kuda hitam menang, jika salah satu kuda hitam sudah mencapai baris pertama kuda putih.

Pada ide dasar awal, kami hanya melihat baris kuda dari sisi satu buah warna saja. Tentunya dengan cara tersebut pun sudah dapat diketahui apakah langkah yang dilakukan oleh kuda ini baik atau tidak. Namun, setelah diteliti lebih lanjut, ditemukan bahwa letak baris musuh juga berpengaruh terhadap terhadap baik atau buruknya suatu *state* permainan sekarang. Dari situ kami mengetahui bahwa kami harus memasukan faktor posisi baris musuh juga kedalam perhitungan heuristik algoritma tersebut.

Dari paragraf sebelumnya, Diketahui bahwa parameter indeks barisan merupakan heuristik yang baik digunakan dalam menghitung *evaluation function* yang diimplementasikan oleh algoritma kami. Namun, setelah ditelaah lebih dalam, kami menemukan satu faktor lagi yaitu jumlah dari kuda yang dimiliki oleh masing-masing warna. Apabila bermain *Knight Through*, semakin sedikit jumlah kuda pada sebuah warna, semakin kecil pula kemungkinan kuda dengan jumlah sedikit itu untuk menang. Hal tersebut disebabkan oleh kemungkinan move kuda ke *state* berikutnya yang lebih sedikit. Berdasar pada fakta tersebut kami berpikir untuk membuat perhitungan selisih antara jumlah kedua warna kuda. Semakin besar selisih kuda antara kedua warna kuda terbentuk, semakin besar pula perbedaan antara state kekalahan dan kemenangan yang terbentuk. Untuk itu, agar dapat menang kami harus mendapat nilai pengurangan antara jumlah kuda dengan warna yang diharapkan menang dan jumlah kuda dengan warna yang diharapkan kalah dengan jarak antar nilai yang sebesar-besarnya.

Setelah melakukan analisis yang dijelaskan dalam paragraf diatas, disimpulkan bahwa ada dua parameter yang mempengaruhi perhitungan heuristik kami yaitu posisi baris pada sebuah kuda dan jumlah kuda. Dari situ, kami membuat sebuah *evaluation function* dari hasil penggabungan dua parameter tersebut kedalam satu heuristik evaluation function. Perhitungannya akan kami detailkan di dengan rumus yang berada di bawah:

Untuk *root*:

$$i = indeks\ baris\ relatif\ ke\ warna\ tertentu\ (untuk\ lebih\ lengkapnya\ lihat\ gambar)$$

$$x_i = jumlah\ kuda\ dengan\ warna\ yang\ diharapkan\ menang(alpha)\ dengan\ indeks\ baris\ i$$

$$y_i = jumlah\ kuda\ dengan\ warna\ yang\ diharapkan\ kalah(beta)\ dengan\ indeks\ baris\ i$$

$$w^x = bobot\ evaluation\ untuk\ warna\ kuda\ yang\ diharapkan\ menang$$

$$w^y = bobot\ evaluation\ untuk\ warna\ kuda\ yang\ diharapkan\ kalah$$

$$f1\left(x_i, w_i^x\right) = \sum_{i=0}^{6} x_i * w_i^x$$

$$f2\left(y_i, w_i^y\right) = \sum_{i=0}^{6} y_i * w_i^y$$

$$eval\left(x_i, w_i^x, y_i, w_i^y\right) = f1\left(x_i, w_i^x\right) + f2\left(y_i, w_i^y\right)$$

Perhitungan diatas diimplementasikan dengan melakukan iterasi *cell* 8x8 sambil mengecek isi dari *cell*. Nantinya dengan menggunakan `whatCell()` apabila terdapat kuda yang mengisi suatu *cell* maka indeks *cell* tersebut akan diambil. Indeks *cell* yang diambil bisa diubah menjadi indeks baris dengan bagi dengan 8 maka didapat posisi kuda tersebut diberikan indeks antara 0 - 7. Indeks baris tersebut dapat dikalikan dengan bobot atau dalam perhitungan diatas direpresentasikan dengan *w*. Pada algoritma yang kami buat semua perhitungan tersebut kami implementasikan dalam method `initHeuristic()`.

Untuk yang bukan root:

$$eval_{parent}\left(x_i, w_i^x, y_i, w_i^y\right) = evaluation\ function\ dari\ parent\ disimpan\ pada\ node\ parent$$

$$\Delta eval = perubahan\ eval\ setelah\ dilakukannya\ move\ pada\ parent$$

$$to = indeks\ baris\ parent\ ketika\ melakukan\ move\ dari\ node\ parent\ ke\ node\ ini$$

$$from = indeks\ baris\ parent\ ketika\ melakukan\ move\ dari\ node\ parent\ ke\ node\ ini$$

$$imusuh = indeks\ baris\ relatif\ pada\ warna\ musuh$$

Apabila *parent* adalah *alpha*:

$$\Delta eval = w_{from}^x - w_{to}^x$$

Apabila *parent* adalah *beta*:

$$\Delta eval = w_{from}^y - w_{to}^y$$

Apabila kuda yang diharapkan menang memakan kuda lawan:

$$\triangle eval = \triangle eval + w^{y}_{imusuh}$$

Apabila kuda yang diharapkan menang dimakan kuda lawan:

$$\triangle eval = \triangle eval - w^{x}_{imusuh}$$

Evaluation function baru dapat *node* yang bukan merupakan root bisa dihitung dengan:

$$eval(from, to) = eval_{parent} + \triangle eval$$

Berbeda dengan cara perhitungan di root pada perhitungan kali ini kami membutuhkan sebuah *node* yang mana *node* tersebut sudah memiliki *parent*. Dari *parent node* bisa didapatkan *move* yang membuat *parent* bisa berubah ke *move* ini dimana dalam implementasi kami hal tersebut kami simpan dalam atribut action. Untuk itu hasil dari evaluation function pada *parent* harus bisa didapatkan. Dimana, hal tersebut bisa didapatkan dengan kompleksitas waktu konstan menggunakan `getValue()`.

apabila mencapai terminal state:

$$eval(kalah) = \quad -\infty$$

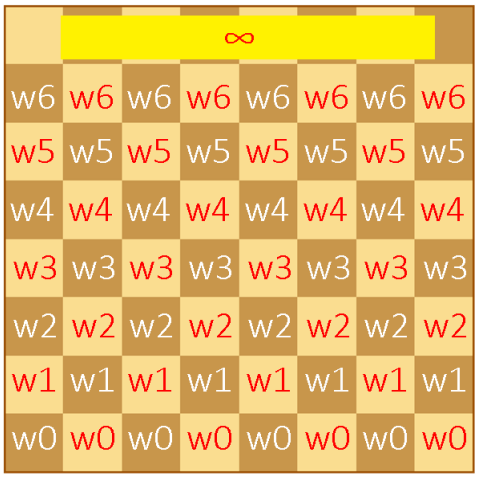
$$eval(menang) = \infty$$

$$eval(kalah) < eval\left(x_{i'}, wx_{i'}, y_{i'}, wy_{i}\right) < eval(menang)$$

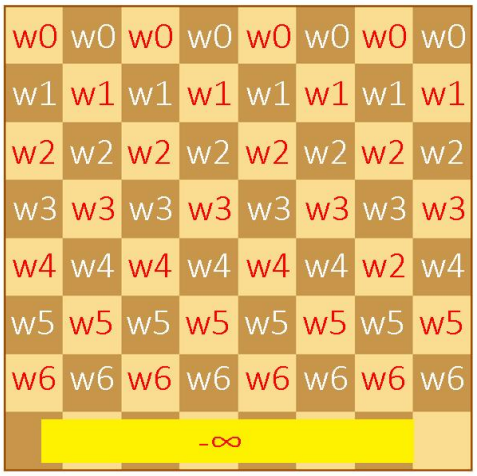
Apabila *node* sudah berada di *terminal state* maka apabila *node* tersebut berada di state kalah maka hasil dari *evaluation function* di *node* tersebut adalah $-\infty$atau value minimal ketika meng-*assign* nilai *beta* di awal. Sedangkan apabila *node* tersebut berada di state menang maka hasil dari *evaluation function* di *node* tersebut adalah ∞atau value maksimal ketika meng-*assign* nilai *alpha* di awal.

Intinya, dari cara-cara di atas perhitungan *evaluation function* yang kami lakukan adalah menjumlahkan jumlah dari bobot yang diberikan pada suatu *cell* yang ditempati oleh kuda dengan bobot heuristik yang diberikan. Setelah itu dicarikan selisih dari perhitungan *evaluation function* antara player yang diharapkan menjadi pemenang dan player yang diharapkan kalah dan dijadikan sebagai *evaluation function*. Untuk lebih detailnya kami menyediakan gambar sebagai ilustrasi yang bisa dijadikan acuan.

Gambar Ilustrasi dari Perhitungan diatas:



Gambar 3.2.h

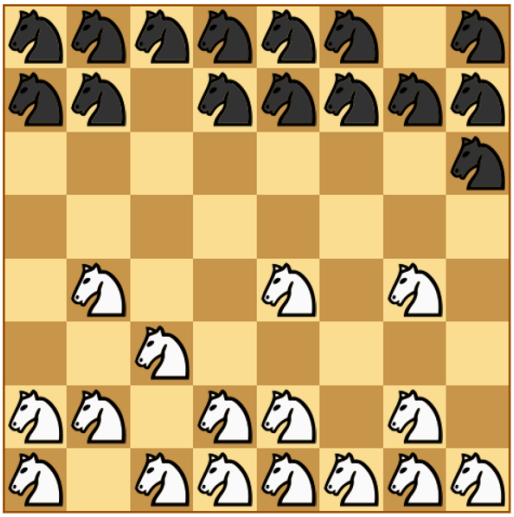


Gambar 3.2.i

Gambar 3.2.h adalah ilustrasi gambar perhitungan $f1\left(x_i, w_i^x\right)$ dan gambar 3.2.i adalah gambar ilustrasi perhitungan $f2\left(y_i, w_i^y\right)$. Maka *evaluation value*-nya bisa didapat dengan mengurangkan *value* pada gambar 3.2.i dan *value* pada gambar 3.2.h. Selain itu diperhatikan apabila sudah ada kuda yang mencapai state kemenangan maka akan langsung di *return* $\infty / -\infty$.

Contoh:

Diberikan *state* pada *node root* suatu *search tree* seperti gambar berikut dengan asumsi *alpha* adalah kuda dengan warna hitam:



Gambar 3.2.j

Pada gambar 3.2.j, *state* pada *node root* suatu *search tree*

Lalu bobot yang dipakai algoritma ini adalah sebagai berikut:

$$w^x = \{1, 1, 4, 4, 8, 640, 640\}$$

$$w^y = \{1, 1, 4, 4, 8, 1280, 1280\}$$

Maka didapatkan hasil perhitungan *evaluation function* seperti berikut:

$$f1\left(x_i, w_i^x\right) = 7 * 1 + 7 * 1 + 1 * 4 + 0 * 4 + 0 * 8 + 0 * 640 + 0 * 640$$

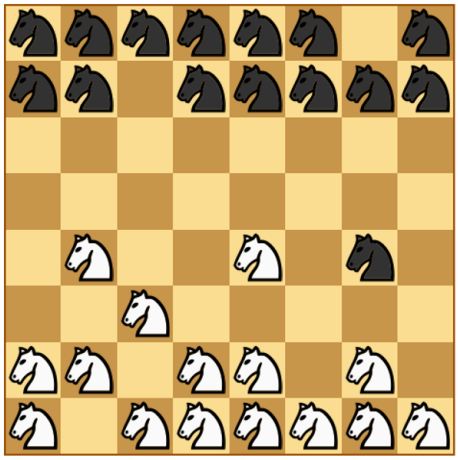
$$f1\left(y_i, w_i^y\right) = 18$$

$$f2\left(y, w_i^y\right) = 7 * 1 + 5 * 1 + 1 * 4 + 3 * 4 + 0 * 8 + 0 * 1280 + 0 * 1280$$

$$f2\left(x_i, w_i^x\right) = -28$$

$$eval\left(x_i, w_i^x, y_i, w_i^y\right) = 18 - 28 = -10$$

Selanjutnya ketika melakukan *exploration* kami mendapat *state* baru dengan kuda hitam paling atas memakan kuda putih paling kanan atas.



Gambar 3.2.k

Pada gambar 3.2.k, state baru yang merupakan *child* dari *root*, dengan kuda hitam paling atas memakan kuda putih paling kanan atas

Maka, karena *node* ini bukan merupakan *root*, *node* ini bisa dihitungnya dengan:

$\Delta eval = 8 - 4 = 4$

$\Delta eval = 4 + 4 = 8$

$eval(3, 5) = -10 + 8 = -2$

Perhitungan *heuristic evaluation function* dengan cara diatas akan dilakukan dalam algoritma secara berulang dalam setiap *node* dalam *search treenya*. Perhitungannya sendiri sudah dilakukan ketika *node* dibuat sebelum melakukan *exploration* sehingga *evaluation function* ini juga dapat digunakan ketika kami melakukan *Move Ordering*. Dari situ kami bisa memberi syarat *Move Ordering* yang kami buat yaitu dengan memilih 5 dengan *evaluation value* terbaik, memilih 1 dengan *evaluation value* terburuk, dan 14 memilih secara *random*.

Pembobotan:

$$w_0^x = w_1^x < w_2^x \leq w_3^x < w_4^x < w_5^x \leq w_6^x$$

$$w_0^y = w_1^y < w_2^y \leq w_3^y < w_4^y < w_5^y \leq w_6^y$$

$$2 * w_6^y, 2 * w_5^y \geq w_6^x, w_5^x$$

Selanjutnya pada implementasi, nantinya diperlukan perhatian lebih detail terhadap bagian pembobotan atau*w*di *evaluation function* heuristik-nya. Pembobotan dapat sangat berpengaruh terhadap bagus atau tidaknya suatu heuristik ketika ketika dijalankan. Pertama-tama, perhatikan perhitungan $2 * w_6^y, 2 * w_5^y \geq w_6^x, w_5^x$, perhitungan tersebut menjelaskan bahwa jumlah kuda berwarna yang diharapkan kalah yang terdapat di baris 5 dan 6 harus berjumlah 2 kali lipat jumlah kuda berwarna yang diharapkan menang yang terdapat di baris 5 dan 6 (posisi relatif ke masing-masing warna kuda dijelaskan pada gambar 3.2.c) . Penyebabnya adalah apabila ada kuda-kuda yang mengancam *player* yang diharapkan menang, kuda tersebut harus dipilih dan dimakan terlebih dahulu ketimbang memilih menyerang player lainnya seperti kasus-nya digambarkan pada ilustrasi di bawah.

Gambar 3.2.I

Contoh posisi kuda dimana disini sebenarnya kuda hitam apabila maju 1 *move* bisa mengancam kuda putih. Namun disini terdapat kuda putih yang sudah mengancamnya oleh karena itu kuda putih tersebut harus dimakan terlebih dahulu untuk menyelamatkan diri.

Selain detail pembobotan di paragraf sebelumnya, masih terdapat pula detail lain yang harus diperhatikan. Perhatikan bahwa dalam perhitungan di atas untuk memberi bobot kami telah memberi syarat lebih kecil, sama dengan, atau lebih kecil sama dengan sebagai syaratnya. Besar dari bobot-bobot tersebut tidak boleh dicari secara sembarang, melainkan dicari dengan memperkirakan kondisi *state* tertentu apabila ada kuda pada beberapa indeks di baris tertentu. Apabila kuda *player* berada di indeks baris 0 dan 1 pasti *value* dari kuda tersebut berbobot sama karena berarti kuda tersebut belum bergerak sama sekali. Sedangkan, apabila sudah masuk indeks baris 2 dan 3 kuda tersebut sudahlah maju satu

langkah. Sehingga, bobot kuda tersebut pastilah lebih besar dari indeks baris 0 dan 1 karena dianggap lebih dekat lagi ke *state* kemenangan. Selanjutnya di baris 4 bobot juga lebih besar karena sudah melewati setengah *board*. Terakhir ada indeks baris 5 - 6 kuda yang mencapai baris sini pasti sudah memberikan ancaman kekalahan kepada lawan. sehingga harus memberikan value yang lebih besar dari total value apabila semua kuda bisa berada di baris 2 dan 3. Sekali lagi, ditegaskan bahwa pada bagian ini kami memakai asumsi indeks baris relatif berdasarkan posisi awal dari masing-masing warna kuda seperti yang ditegaskan oleh ilustrasi gambar 3.2.c.

Dari Seluruh hasil Analisis di atas kami akan membuat eksperimen dengan melawankan agen yang dimiliki oleh kami dengan agen Ludii. Dari sana kami akan mencatat jumlah rata-rata jumlah move yang diperlukan untuk menang. Selanjutnya akan dihitung rata-rata *time* untuk melakukan sebuah *move* dengan sampling dari suatu *trial*. Selain itu kami akan mencoba mengubah bobot heuristik dari kami dan melawankan dengan bobot heuristik sebelumnya.

4. Implementasi

4.1. Terminologi Ludii

- `Game`, objek yang memuat semua peraturan, *equipment*, fungsi, dan sebagainya. Diperlukan untuk memulai setiap permainan di Ludii *Player*.
- `Trial`, kumpulan semua *history* atau *record* dari gerakan yang telah dipakai selama permainan berlangsung. Trial dalam Ludii ini bisa disimpan sebagai hasil evaluasi dari AI yang telah dibuat.
- `State`, tempat menyimpan semua *properties* dari *game state* (tidak termasuk *move* yang ada di *history*, karena itu terdapat pada *trial*).
- `Context`, berisi objek `Context` dari `Trial` yang sedang berjalan dan biasanya digunakan sebagai objek untuk *mem-passing method-method.* Objek ini juga berfungsi sebagai *pointer* bagi "*higher level object*" seperti *Game* dan "*lower level object*" seperti `State` dan `Trial`.
- `Action`, adalah *atomic object* yang pada saat diaplikasikan pada sebuah *game state* akan memanipulasi 1 *property* dari objek *game state*. Action ini bisa digunakan untuk mensimulasikan Move ke sebuah game state.
- `Move`, objek yang berguna untuk merepresentasikan satu buah gerakan yang dilakukan oleh agent yang berada di Ludii.

Referensi: https://ludiitutorials.readthedocs.io/en/latest/

4.2. State representation

Pada *library* yang disediakan oleh Ludii sudah terdapat sebuah objek yang dapat digunakan untuk menyimpan seluruh objek(*higher level object*/*lower level object*). Dimana hal tersebut telah dijelaskan pada bagian terminologi Ludii sebelumnya. Objek tersebut adalah `Context` maka dalam algoritma kami, setiap *node* kami menyimpan objek ini.

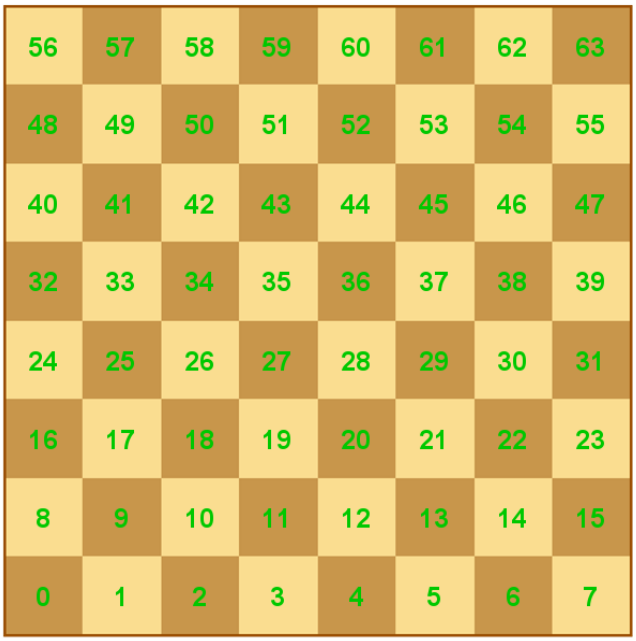
Karena dokumentasi Ludii tidak terlalu rinci maka diperlukan decompiler untuk melihat isi dari librarynya untuk decompiler sendiri bisa didownload dari link http://java-decompiler.github.io/. Setelah membuka jar Ludii menggunakan decompiler tersebut maka didapatkan pada suatu class bernama `ContainerState` dimana `ContainerState` ini meng-*extend* class `State`. Fungsi dari `ContainerState` ini adalah untuk mengetahui *state* dari *board* yang dimiliki sekarang dan dapat kami access apabila dibungkus dengan sebuah *iterator*.

Apabila dilihat dalam kode beginilah cara mengakses `ContainerState`:

```
for ( ContainerState containerState : context.state().containerStates())
```

Referensi: Java Decompiler dan
https://github.com/Ludeme/LudiiExampleAI/blob/master/src/experiments/Tutorial.java

Selanjutnya untuk mendapatkan isi dari `ContainerState` diatas bisa digunakan *method* `whatCell(int index)` dimana `index` adalah indeks *cell* (kotak pada catur) posisinya yang ingin diketahui isinya di papan catur tersebut sesuai gambar berikut:



Gambar 4.2.a

Method `whatCell(int index)` ini sendiri dalam *Knight Through* memiliki kemungkinan *return* 3 buah *integer* yaitu 0, 1,dan 2. Arti dari masing masing angka akan dijelaskan sebagai berikut:

- 0: *Cell* tidak ditempati oleh kuda.
- 1: *Cell* ditempati oleh kuda putih.
- 2: *Cell* ditempati oleh kuda hitam.

Sehingga Apabila diilustrasikan bisa dibayangkan dalam *class Container State* dia menyimpan *state* dari *board* seperti ilustrasi berikut:

Gambar 4.2.b

i=0-7	1	1	1	1	1	1	1	1
i=8-15	1	1	1	1	1	1	1	1
i=16-23	0	0	0	0	0	0	0	0
i=24-31	0	0	0	0	0	0	0	0
i=32-39	0	0	0	0	0	0	0	0
i=40-47	0	0	0	0	0	0	0	0
i=48-55	2	2	2	2	2	2	2	2
i=56-63	2	2	2	2	2	2	2	2

Pada gambar 4.2.b ini merupakan contoh salah satu *state* secara grafik dan representasinya di `ContainerState`(sebenarnya *array* 1D).

4.3. Struktur dan Definisi *Class*

Dalam implementasinya kami menggunakan beberapa algoritma sebagai sumber referensi dan kami juga melakukan penyesuaian pada algoritma agar cocok untuk digunakan pada *game Knight Through*.

Setelah melakukan berbagai perbaikan dan penambahan, pada akhirnya, kami mendapatkan bahwa dalam implementasi yang kami lakukan dibuatkan 4 *class* yaitu: `AlphaBetaAI`, `Node`, `Heuristics`, dan `BantuanSorting`.

`AlphaBetaAI.class`

`AlphaBetaAI` merupakan *class* yang menjalankan seluruh algoritma yang kami buat. *Class* ini sendiri meng-*extend class* AI dari *library* Ludii dalam `package util.AI`. Pada algoritma ini kami mengimplementasikan 2 *method* dari *class* AI tersebut yaitu `selectAction(Game game, Context context, double maxSeconds, int maxIterations, int maxDepth)` dan `InitAI(final Game game, final int playerID),`.

Selain method yang kami implementasi dari *class* `AI`, pada class `AlphaBetaAI` kami juga membuat *method* implementasi sendiri lainnya yaitu `alphaBeta()`, `isCuttingOff()` , dan `SelectMove()`.

Atribut:

1. **`bestMove : Move`**
   Fungsi dari atribut ini adalah menyimpan gerakan terbaik yang dapat dilakukan suatu *move* ketika dijalankannya fungsi .
2. **`playerID : int`**
   Fungsi dari atribut ini adalah menyimpan player yang sekarang menjadi *alpha*.
3. **`analysisReport : String`**
   Menampung seluruh *string* yang ingin dikeluarkan dalam panel di *tab analysis*.

`Method:`

1. **`initAI()`**
   Method yang dijalankan oleh *program* Ludii sebelum `selectAction()`. Dalam *method* ini, pada implementasinya kami hanya memakai *parameter* `playerID`. `PlayerID` ini berfungsi untuk menentukan *player* mana yang memakai *AI* yang kami buat atau dalam kasus algoritma ini yang ditentukan untuk menjadi *alpha*. Nantinya supaya dapat digunakan oleh *class parameter* `playerID` diisikan ke atribut `playerID`.
2. **`selectAction(Game game, Context context, double maxSeconds, int maxIterations, int maxDepth) : Move`**
   Method yang berguna untuk menentukan *move* mana yang akan kami lakukan selanjutnya. Dalam method ini, untuk mempermudah, kami mengabaikan *parameter* `maxSeconds`, `maxIteration`, serta `maxDepth` dan hanya memakai *parameter* `context` dan *game*. Awalnya method ini akan membuat sebuah *object Node root* yang bisa nantinya di-*expand* dengan menggunakan *pointer* ke *object-object Node* yang akan terbentuk pada saat algoritma dijalankan.

   Setelah itu, nantinya di dalam *method* ini kami akan memanggil *method* untuk melakukan Algoritma *MiniMax AlphaBeta Pruning*. Dari hasil algoritma tersebut akan didapatkan *Move* terbaik dan disimpan dalam atribut *`bestMove`*. Setelah *`bestMove`* ditetapkan pada akhir method atribut *`bestMove`* tersebut di *return* untuk dibaca dalam Ludii ketika agen memilih suatu *move*.

3. **`generateAnalysisReport():String`**
   Method yang dapat di implementasi dari `AI` fungsinya menampilkan *string* pada *tab analysis* di Ludii setelah melakukan *move*. Disini kami mengimplementasikan sehingga mengeluarkan *string* dari atribut `analysisReport` terinspirasi dari Github Ludii.

4. **`alphaBeta()`**

   Merupakan sebuah method yang mengimplementasikan *search* menggunakan *Minimax Alpha-Beta Pruning*. Implementasi dari *method* ini kami ambil dan modifikasi dari website geeksforgeeks.

   Referensi:
   https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/.

   Method ini diimplementasikan dengan cara rekursif. *Base case* ditemukan apabila *parameter* dari *node* yang dijadikan patokan sekarang sudah mencapai *terminal state* atau *node* tersebut merupakan merupakan *cutting off*. Ketika Mencapai *base case method* ini akan mengembalikan *heuristic evaluation value* dari *node* yang sekarang sedang berada dalam *parameter* dalam sebuah iterasi rekursif.

   Berbeda halnya jika method ini sedang tidak berada di *base case* maka *method* ini akan segera mengecek apakah *node* yang sedang berada dalam iterasi rekursif ini adalah *alpha* atau *beta*. Setelah di cek maka akan dipanggil method `selectMove()` untuk mengambil mana *node* yang terpilih untuk di *exploit* lebih lanjut.

5. **`isCuttingOff(int depth)` : boolean**

   Method ini adalah *method* untuk mengecek *depth node* saat sedang melakukan *exploitation node.* Dalam method ini dicek apakah *depth* dari *parameter* masukan sudah mencapai 5. Jika sudah maka akan mengembalikan nilai *true* dan apabila tidak akan mengembalikan nilai *`false`*.

6. **selectMove(Node node,int currentId) : `FastArrayList<Node>`**

   Method ini berfungsi untuk melakukan algoritma *Move Ordering*. Didalamnya input dari parameter `node` akan di sort menggunakan class `BantuanSorting` dana hasilnya akan diletakan dalam sebuah `FastArrayList<Node>` dan dikembalikan. Dalam *method* ini terdapat dua detail yang harus diperhatikan yaitu pembuatan `Context` baru dan penggunaan method `removeSwap()`. `Context` haruslah dibuat lagi karena apabila tidak membuat `Context` baru maka *pointer* tetap akan mengacu pada `Context` sebelumnya yang berada dalam suatu node. Sedangkan disini apabila ingin mensimulasikan `Context` di node selanjutnya ketika melakukan *move* dengan method baru dengan *method* `.game().apply()`. Detail selanjutnya adalah method spesial dari `FastArrayList` yang diberikan oleh Ludii yaitu `removeSwap()`. `removeSwap()` ini digunakan untuk membuang value dari sebuah `FastArrayList` sembari me-*return* value yang dibuang tersebut. Penggunaan method ini bisa didapatkan dengan Java Decompiler atau melihatnya di Ludii Github UCT .

   Selanjutnya disini juga kami memakai cara melakukan *random* yang cukup baru dari java yaitu `ThreadLocalRandom` yang menurut berbagai sumber lebih baik digunakan ketimbang class Random biasa dan digunakan juga di Ludii Ludii Github Random AI.

Node.class

Kelas *Node* dalam algoritma merupakan kelas yang membantu memodelkan atau merepresentasikan sebuah *node*. Kelas *Node* ini sangat penting karena keputusan *move* yang akan diambil nantinya akan didasarkan oleh hasil perhitungan *evaluation value* yang dihitung di dalam kelas ini. Kelas *Node* ini juga memiliki 6 *getter*, 1 *setter*, 1 *constructor* dan 5 *method* lainnya yang digunakan sebagai perhitungan *heuristic evaluation value*. Kelas *Node* ini sendiri terinspirasi dari kelas *Node* yang berasal dari implementasi UCT yang dibuat oleh Ludii (referensi: Ludii Github UCT).

Atribut:

1. **parent : Node**
   Atribut yang merupakan *pointer* ke sebuah Node yang merupakan *parent* dari *Object* Node pemilik atribut ini.
2. **state : Context**
   Atribut penting yang berisi *state* dari *object* Node pemilik atribut ini. Atribut ini nantinya memungkinkan penggunanya untuk mendapat *state* posisi semua kuda berada dengan tanpa harus mensimulasikan statenya menggunakan *array*.
3. **action : Move**
   Atribut penting yang berisi *move* untuk berubah dari state *parent* ke *state* objek Node pemilik atribut ini.
4. **children : FastArrayList<*Node*>**
   ArrayList spesial buatan Ludii yang berisi *pointer* ke Node yang merupakan *children* dari *Object* Node pemilik atribut ini.
5. **possibleMove : FastArrayList<*Move*>**
   ArrayList spesial buatan Ludii yang diisikan *move-move valid* yang bisa dilakukan oleh Node pemilik atribut ini.
6. **value : double**
   Atribut yang menampung hasil dari perhitungan *evaluation function* yang telah dihitung berdasarkan *state* di objek Node pemilik atribut ini.
7. **isTerminal : boolean**
   Atribut yang mendeskripsikan apakah *node* ini adalah *terminal* atau sudah tidak memiliki *move* berdasarkan *state* di objek Node pemilik atribut ini.
8. **heuristic : double[]**
   Atribut yang nantinya mengacu pada bobot yang disimpan disimpan secara static dalam class Heuristic. Bobot yang diacu disini adalah bobot untuk kuda yang merupakan *alpha*.
9. **heuristicmusuh : double[]**
   Atribut yang nantinya mengacu pada bobot yang disimpan disimpan secara static dalam class Heuristic. Bobot yang diacu disini adalah bobot untuk kuda yang merupakan *beta*.
10. **currentPlayer : integer**
    Atribut yang digunakan untuk menentukan siapa yang menjadi player untuk dianalisa di *Object* Node pemilik atribut ini. Cara menganalisanya adalah membandingkannya dengan atribut maximizingPlayer.
11. **maximizingPlayer : integer**
    Atribut yang digunakan oleh *Object* Node pemilik atribut ini untuk mengidentifikasi siapa yang menjadi *alpha* dari sisi agen yang berjalan kali ini.

Method:

1. **`constructor(Node parent,Move action,Context state,int currentPlayer,int maximizingPlayer)`**
   Constructor pada object `Node` ini digunakan untuk menginisiasi atribut-atribut pada `node` tersebut. Terdapat dua kondisi, jika *pointer* `parent` pada `Node` tersebut bukan *null*, maka *parent* dari `node` ini adalah *node* itu sendiri dan akan memanggil *method* `changeHeuristic(action)`. sedangkan jika *pointer* `parent` pada *node* tersebut adalah *null* ,maka akan memanggil *method* `initHeurisctic()`. Kedua *method* ini sangatlah berbeda dalam kompleksitas dimana akan dijelaskan berikutnya bagian berikutnya.
2. **`addChildren(Node child)`**
   Method ini berfungsi sebagai *setter* untuk atribut `children` yang merupakan sebuah *pointer* ke *object* `Node` lainnya.
3. **`changeHeuristic(Move action)`**
   Method ini berfungsi untuk memilih dan memanggil *method* untuk perhitungan *evaluation function* konstan yang tepat sesuai dengan alpha sekarang. Berdasarkan atribut `maximizingPlayer` *method* ini dapat memanggil *method* `heuristicPlayerOne()` atau `heuristicPlayerTwo()`. Dapat dipastikan *method* ini hanya akan terpanggil apabila *node* yang memanggilnya bukanlah *node root*.
4. **`getAction()`** : **Move**
   Method yang berfungsi untuk mendapat *move* yang didapat dari object `Move` yang dilakukan untuk mencapai state ini. Method ini cukup penting karena membantu method `changeHeuristic()` dalam menurunkan kompleksitas waktu perhitungan *evaluation function*.
5. **`getAllPossibleMove()`**
   Method ini berfungsi untuk men-generate semua *move* yang possible dilakukan oleh *node* ini dan menginisialisasikan nilainya ke dalam atribut `possibleMove`.
6. **`getCurentNodePlayer()`**
   Method yang berfungsi sebagai getter dari atribut `currentPlayer`.
7. **`getMoves()`: `FastArrayList<Move>`**
   method tanpa *parameter* dengan *return type* yang berfungsi sebagai *getter* dari atribut `possibleMove` telah digenerate oleh `getAllPossibleMove()`.
8. **`getState() : Context`**
   Method yang berfungsi sebagai *getter* dari atribut `state`.
9. **`getValue() : double`**
   method yang berfungsi sebagai *getter* atribut `value`.
10. **`heuristicPlayerOne()`**
    Method yang berfungsi untuk menghitung nilai *evaluation function* apabila *player* dengan kuda warna putih dipilih menjadi *alpha*. *Method* ini adalah *method* yang memiliki fungsionalitas dan hasil yang sama dengan method `initHeuristic()` namun dapat menjalankan perhitungan *evaluation function* dengan kompleksitas waktu konstan.
    Method ini merupakan salah satu penyebab inti cepatnya seluruh perhitungan di algoritma AlphaBeta implementasi kami. Di awal, *method* ini akan mengambil evaluation function yang dimiliki oleh *parent node* dengan method getValue(). Lalu, bisa didapatkan *pointer* Object `Move` *parent* untuk mencapai *node* yang menjalankan *method* ini dengan menggunakan `getAction()` pada *node* ini. Lalu, dari *object* `Move` bisa didapatkan keterangan dari *move* pada *parent* sebelumnya dengan *method* `.to()` dan `.from()`. Dari sana bisa dicari berapa banyak

perubahan hasil dari evaluation yang terjadi ketika *move* tersebut dilakukan hanya dengan operator aritmatika +(plus) atau -(minus).
Apabila dilihat pada paragraf sebelumnya, operasi yang dilakukan pada *method* ini tidaklah banyak. Namun, diperlukan ketelitian dalam membuat kondisi-kondisi dan indeks dalam *method* ini. Terdapat kondisi dimana kuda memakan kuda lawan atau lawan memakan kuda yang dimiliki oleh *alpha*. Apabila kondisi tersebut terjadi besaran dari perubahan bobot harus dikurangi atau ditambah. Indeks bobot juga harus diberikan secara tepat supaya mengakses indeks *array* pada *class* `Heuristics` secara tepat dan tidak terbalik. Hal tersebut kami atasi dengan menggunakan operator aritmatika minus (-).

11. **`heursticPlayerTwo()`**
    Method yang sama dengan `heursticPlayerOne()` namun dalam *method* ini, dapat menghitung heuristik dari *player* untuk kuda dengan warna hitam yang dipilih sebagai *alpha*.
12. **`initHeuristic()`**
    Method ini memiliki beberapa fungsionalitas dan hasil perhitungan yang sama dengan **`changeHeuristic()`**. Tetapi *method* ini memiliki kompleksitas waktu yang lebih lama dibanding *method* tersebut. Oleh karena itu, *method* ini hanya digunakan untuk *root* dari sebuah *search tree*. Cara *method* ini menghitung *evaluation function* sendiri adalah dengan menggunakan `Context` untuk mendapat isi dari setiap cell seperti yang dijelaskan di *state space* representation. Setelah itu, melakukan iterasi kepada setiap *cell* yang ada di dalam *game board Knight Through*.

    Pada awalnya terdapat atribut value berisi 0 yang digunakan untuk menampung hasil dari evaluation function. Setelah itu, method ini akan melakukan iterasi *cell* 8x8, dengan cara melakukan looping 64 kali di dalam iterator dari `ContainerState` sembari memanggil method `whatCell(int index)` untuk mengetahui isi dari setiap *cell* yang berada di *board* permainan. Setelah itu akan dicek isi dari `ContainerState` tersebut, apabila *cell* tersebut mengandung kuda, maka kami akan menyimpan indeks dari *cell* tersebut.

    Selanjutnya, dalam implementasi ini, kami mengkonversi indeks dari *cell* menjadi indeks baris dengan menggunakan operator aritmatika / (bagi). Apabila sudah diketahui hasil dari pembagian tersebut *method* ini akan mengambil bobotnya indeks tersebut dari *class* `Heuristics`, lalu menambahkan nilai dari bobot tersebut kedalam sebuah variabel sementara.

    Method akan berhenti bekerja ketika iterasi selesai dan atribut `value` sudah selesai diubah. Dalam implementasinya, kondisi dari *method* harus dibuat secara hati-hati dikarenakan posisi indeks posisi kuda warna hitam terbalik dengan posisi kuda warna putih. Namun hal tersebut dapat *dihandle* secara teliti dengan membalik *value* tersebut operator aritmatika - (minus). Selain itu diperlukannya juga perhatian terhadap siapa yang menjadi *alpha* dikala method ini dijalankan karena hal tersebut akan berpengaruh terhadap positif atau negatifnya perhitungan untuk *evaluation function*.
13. **`isTerminal() : boolean`**
    Method yang berfungsi untuk mengecek apakah permainan sudah mencapai *terminal state* atau belum. *Method* dijalankan dengan menggunakan atribut *state* dan memanggil *method* untuk *Object* `Context` yaitu `trial().over` yang akan mengembalikan nilai *true* atau *false*. Apabila mengembalikan *true* maka state tersebut adalah terminal. Apabila mengembalikan *false* maka

state tersebut bukan *terminal*. Method ini sendiri digunakan berdampingan dengan `isCuttingOff()` di dalam method `AlphaBeta()`.

`BantuanSorting.class`

Class ini adalah *class* untuk membantu sorting pada saat *Move Selection* dilakukan. *Class* ini sendiri meng-*implement* `Comparable` dari Java. Dimana operasi *sorting* pada kelas ini dilakukan dengan cara membanding value dari *node* kelas dan value dari *node parameter* yang ingin di-*sort*. Nantinya Java akan menyediakan cara *sorting* yang *stable* $^{n}\log_{n}$ untuk digunakan.

Atribut:

1. **`node:Node`**
   Merupakan atribut yang digunakan untuk menyimpan *pointer* `Node` yang di *point* olehnya supaya seakan "terbungkus" dan dapat di-*sort*.

Method:

1. **`constructor()`** dengan parameter *node* yang digunakan untuk menginisialisasi atribut *node* dengan parameternya.
2. **`compareTo()`** dengan parameter o yang mempunyai return type int. Method ini adalah hasil *override* dari implement Comparable<BantuanSorting> di kelas BantuanSorting. Fungsi dari *method* ini adalah melakukan *sorting value* yang ada pada *node* dengan cara membuat *if-else statement* yang berisi kondisi perbandingan *value* di atribut *node* dengan *node* di *parameter* dan mengembalikan nilai dari -1, 0, atau 1.

`Heuristic.class`

Model class yang berisi nilai-nilai statis yang digunakan sebagai nilai-nilai bobot berdasarkan posisi kuda di papan permainan. *Heuristic* disini dibuat menjadi dua sebagai pemodelan dari w^x dan w^y. Untuk indeksnya sendiri relatif terhadap indeks baris pada warna masing-masing kuda. Untuk itu supaya dapat digunakan dengan tepat pengindeksan tersebut *handle* ketika menghitung *evaluation value* dengan operator aritmatika tertentu.

Atribut:

1. **`heuristic : static double[]`**
   Atribut yang langsung di-*assign* serta bersifat *static*. Atribut ini menyimpan bobot yang indeksnya sesuai dengan indeks relatif terhadap warna kuda yang ditentukan menjadi *alpha* atau warna kuda yang diharapkan menang karena merupakan kuda dengan yang dimiliki oleh agen pemilik atribut ini.
2. **`heuristicmusuh : static double[]`**
   Atribut yang langsung *di-assign* serta bersifat *static*. Atribut ini menyimpan bobot yang indeksnya sesuai dengan indeks relatif terhadap warna kuda yang ditentukan menjadi *beta* atau warna kuda yang diharapkan kalah karena merupakan lawan dari agen pemilik atribut ini.

4.4. Catatan Tambahan Implementasi Custom AI di Ludii

Untuk implementasi kami seperti yang kami katakan di bagian analisis kami tidak memakai *Transposition Table.* Selanjutnya dalam Ludii Player sendiri secara *default* akan ada *checkbox* yang menunjukan waktu yang diperlukan dalam menjalankan suatu algoritma. Namun dalam implementasi kami waktu dalam Ludii player tersebut diabaikan mengingat kami ingin syarat *Cutting Off Search* yang

kami pilih adalah menggunakan *depth* bukan waktu. Bisa diargumentasikan pula bahwa waktu yang dipakai oleh kami tidak akan melebihi 3 s.

Gambar 4.4.a
Pada gambar 4.4.a, hal yang perlu diabaikan ketika menjalankan jar agen cerdas implementasi kami di ludii.

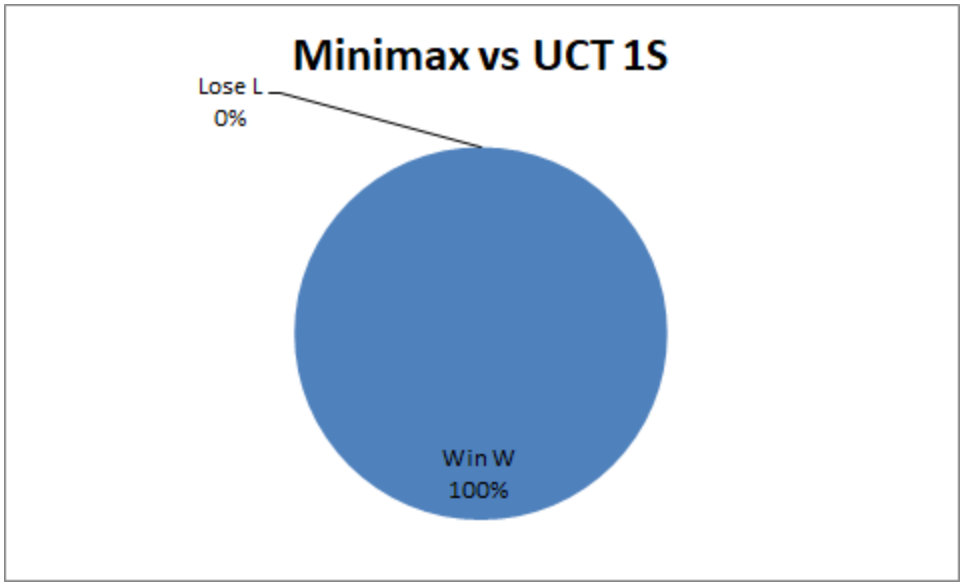
Apabila ingin lebih cepat lagi, bisa juga membuat `initHeuristic()` hanya dilakukan sekali pada state awal dengan menyimpan seluruh hasil *evaluation function dari grandchildren* (*children* dari *children*) *node* yang digunakan. Namun cara ini hanya memberikan *trade-off* memori dan waktu. Kami memilih untuk menghemat memori sehingga kami biarkan saja untuk setiap kali agen ingin memilih *move* menggunakan `selectAction()` ketika dalam *root* di *search tree* dia dibiarkan terlebih dahulu untuk mengiterasi terlebih dahulu 64 *cell* pada *board* tersebut.

5. Eksprimen

Kami melakukan iterasi pada eksperimen dengan menggunakan algoritma *Minimax* kami melawan algoritma UCT 1S hingga 10S. Kami melakukan semua eksperimen dengan komputer yang sama agar menjaga lingkungan eksperimen tetap. Berikut spesifikasi komputer yang kami gunakan,

- *Processor*: Intel i3-3220
- VGA: Nvidia GTX 650 Ti
- RAM: 16 GB DDR4
- *Memory*: SSD (*Write* 540MB/s & *Read* 430MB/s)

***Minimax Alpha-Beta* vs UCT 1 s**
Eksperimen dilakukan sebanyak 20 kali dan Algoritma *Minimax Alpha-Beta* memenangkan semua permainannya hingga mendapatkan rasio kemenangan 100%. Kemudian mendapatkan rata-rata *turn* di angka 22.3 langkah per game.



Gambar 5.a

Chart yang menggambarkan tingkat kemenangan agen melawan UCT 1s.

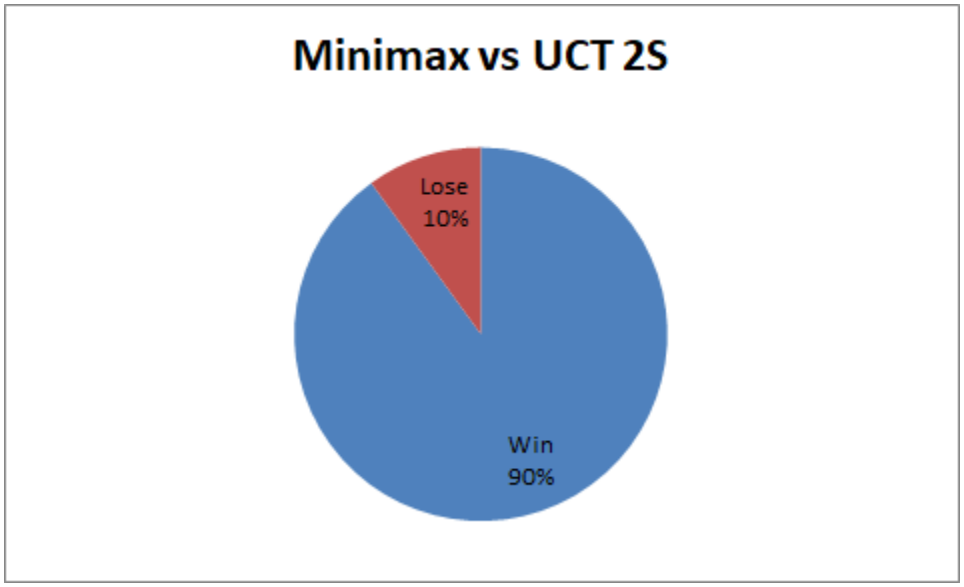
Gambar 5.b

Contoh gambar *state* terakhir dari eksperimen melawan UCT 1 s .

***Minimax Alpha-Beta* vs UCT 2 s**

Eksperimen dilakukan sebanyak 20 kali dan Algoritma *Minimax Alpha-Beta* memenangkan semua permainannya hingga mendapatkan rasio kemenangan 90% dan rata-rata *turn* 25.9 langkah per *game*. Pada percobaan kali ini terdapat kekalahan setelah kami melihat ulang langkah-langkahnya. Kami melihat bahwa kekalahan bukan dari kesalahan fatal. Contoh kesalahan fatal, dimana seharusnya kuda memakan lawan agar lawan tidak menang tetapi kuda algoritma memilih untuk menggerakan yang lain dan kalah.

Jadi memang terjadi anomali dimana posisi yang diambil oleh lawan itu juga sangat optimal hingga kuda kami tidak punya gerakan sama sekali untuk mencegah kekalahan.



Gambar 5.c

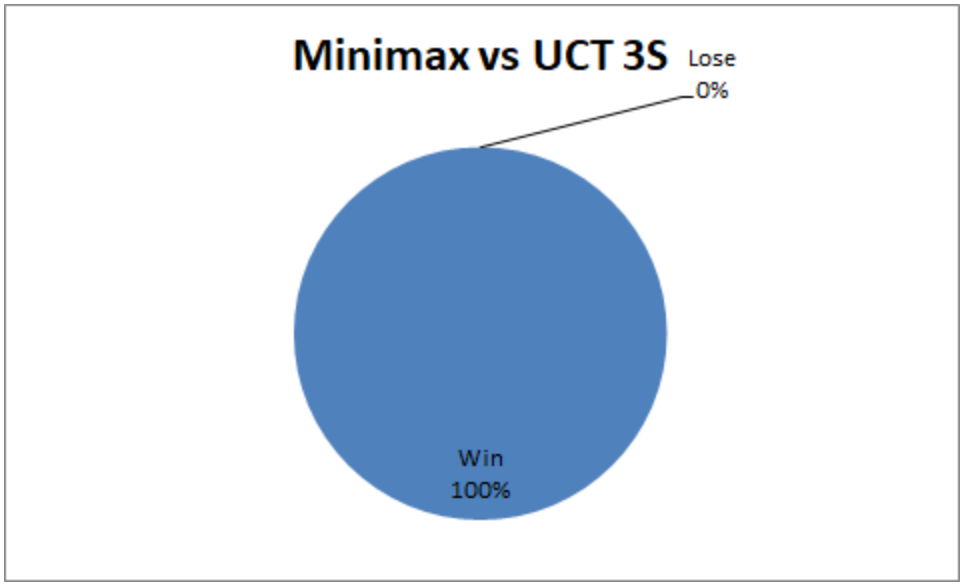
Chart yang menggambarkan tingkat kemenangan agen melawan UCT 2 s.

Gambar 5.d

Contoh gambar *state* terakhir dari eksperimen melawan UCT 2 s .

Minimax Alpha-Beta vs UCT 3 s

Eksperimen dilakukan sebanyak 10 kali dan Algoritma *Minimax Alpha-Beta* memenangkan semua permainannya hingga mendapatkan rasio kemenangan 100%. Kemudian mendapatkan rata-rata *turn* di angka 24 langkah per *game*.



Gambar 5.e

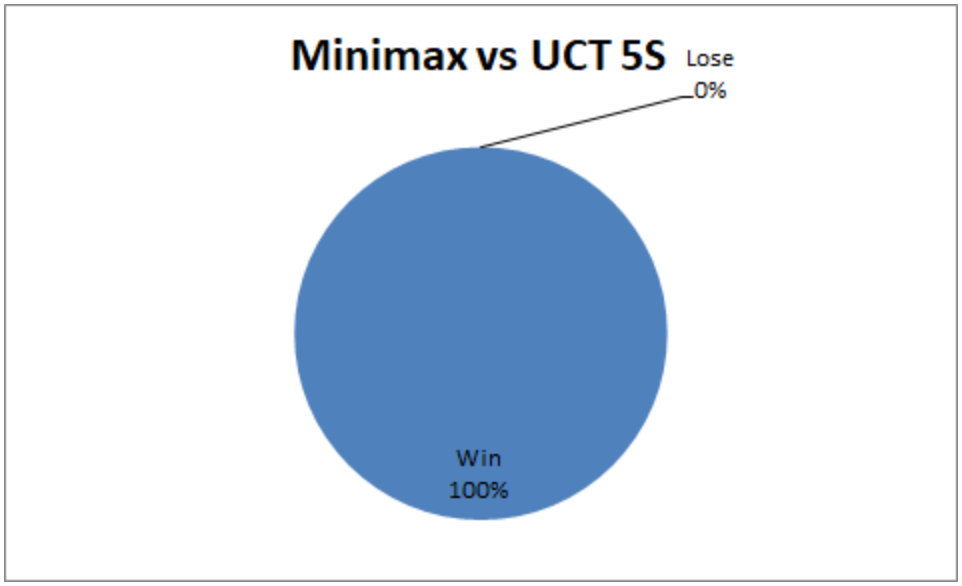
Chart yang menggambarkan tingkat kemenangan agen melawan UCT 3 s.

Gambar 5.f

Contoh gambar *state* terakhir dari eksperimen melawan UCT 3 s.

Minimax Alpha-Beta vs UCT 5S

Eksperimen dilakukan sebanyak 10 kali dan Algoritma *Minimax Alpha-Beta* memenangkan semua permainannya hingga mendapatkan rasio kemenangan 100%. Kemudian mendapatkan rata-rata *turn* di angka 26.25 langkah per *game*.



Gambar 5.g

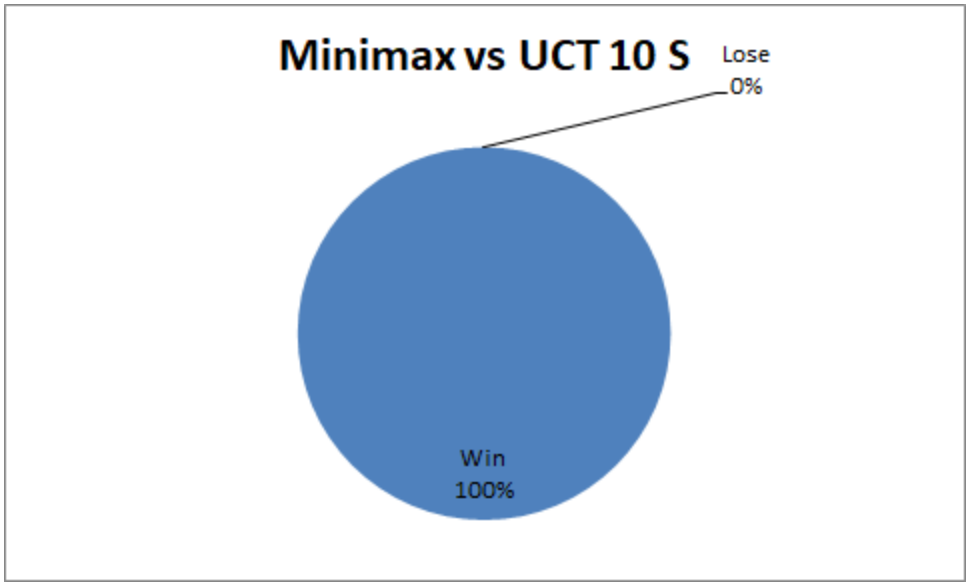
Chart yang menggambarkan tingkat kemenangan agen melawan UCT 5 s.

Gambar 5.h

Contoh gambar *state* terakhir dari eksperimen melawan UCT 5 s.

Minimax Alpha-Beta vs UCT 10S

Eksperimen dilakukan sebanyak 10 kali dan Algoritma *Minimax Alpha-Beta* memenangkan semua permainannya hingga mendapatkan rasio kemenangan 100%. Kemudian mendapatkan rata-rata *turn* di angka 28.4 langkah per *game*.



Gambar 5.i

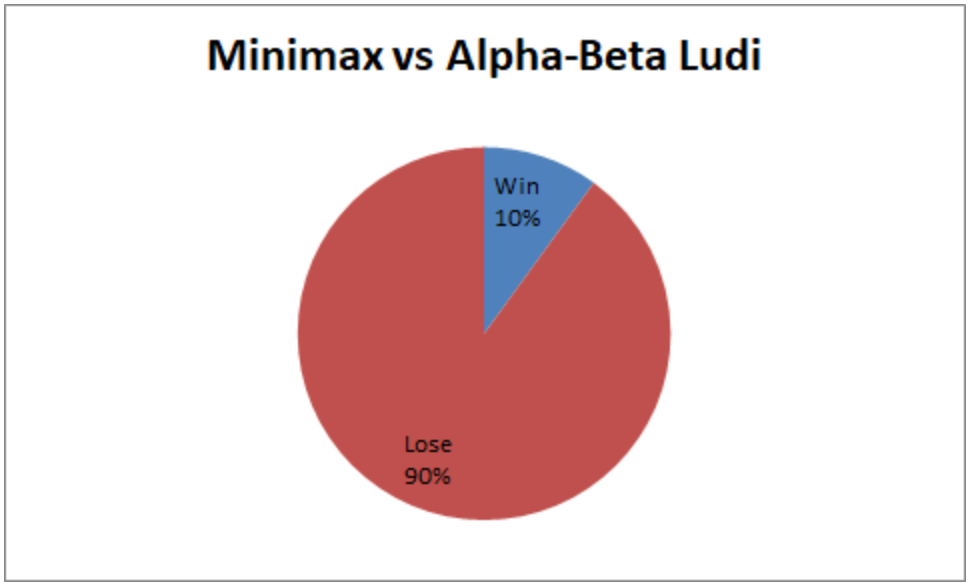
Chart yang menggambarkan tingkat kemenangan agen melawan UCT 10s.

Gambar 5.j

Contoh gambar *state* terakhir dari eksperimen melawan UCT 10 s.

***Minimax Alpha-Beta Pruning* vs *Alpha-Beta* Ludii**

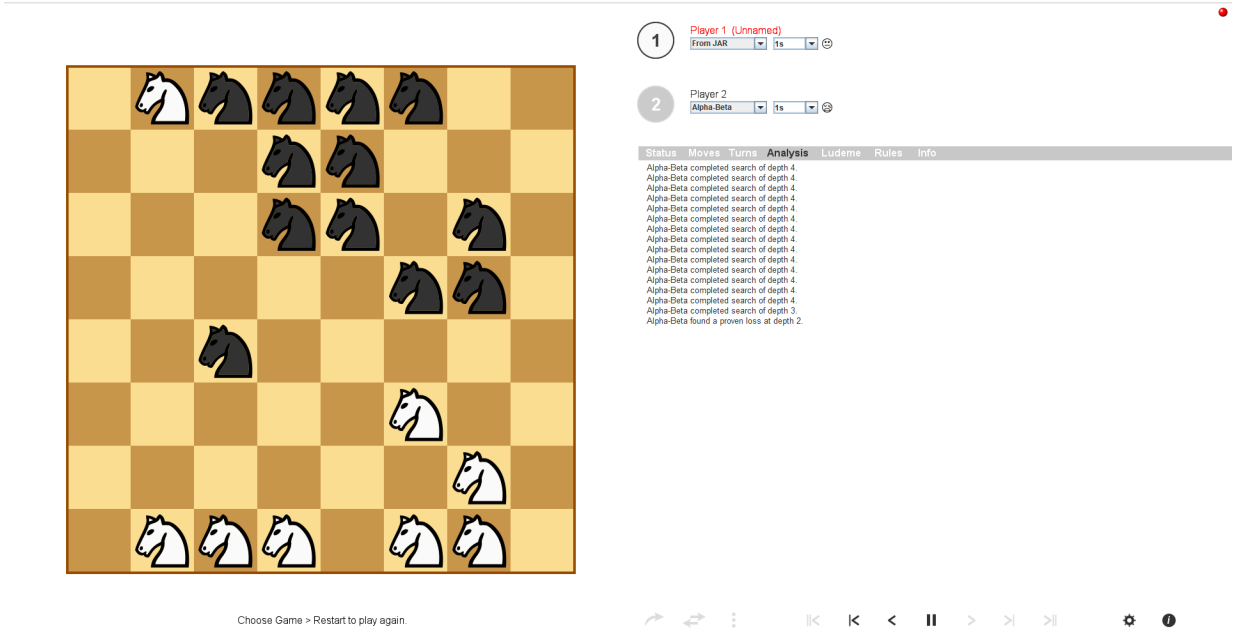
Eksperimen dilakukan sebanyak 20 kali dan Algoritma *Minimax Alpha-Beta* kalah hampir semua permainannya hingga mendapatkan rasio kemenangan 10%. *Alpha-Beta* Ludii dalam eksperimen ini menggunakan waktu 1 detik. Eksperimen lanjutan dengan menggunakan waktu yang lebih lama kami rasa sudah tidak relevan, karena pada waktu 1 detik algoritma Ludii sudah dapat melakukan *complete search* pada *node* dan menemukan proven win pada gerakan-gerakan akhir permainan.



Gambar 5.k *Chart* yang menggambarkan tingkat kemenangan agen melawan Alpha-Beta Ludii 1 s.

Gambar 5.l

Contoh gambar *state* terakhir dari eksperimen melawan Alpha-Beta Ludii 1 s.



Gambar 5.m

Chart yang menggambarkan tingkat kemenangan agen melawan Alpha-Beta Ludii 1 s.

Rata-rata lama waktu algoritma *Minimax* melakukan *move* tiap *turn*

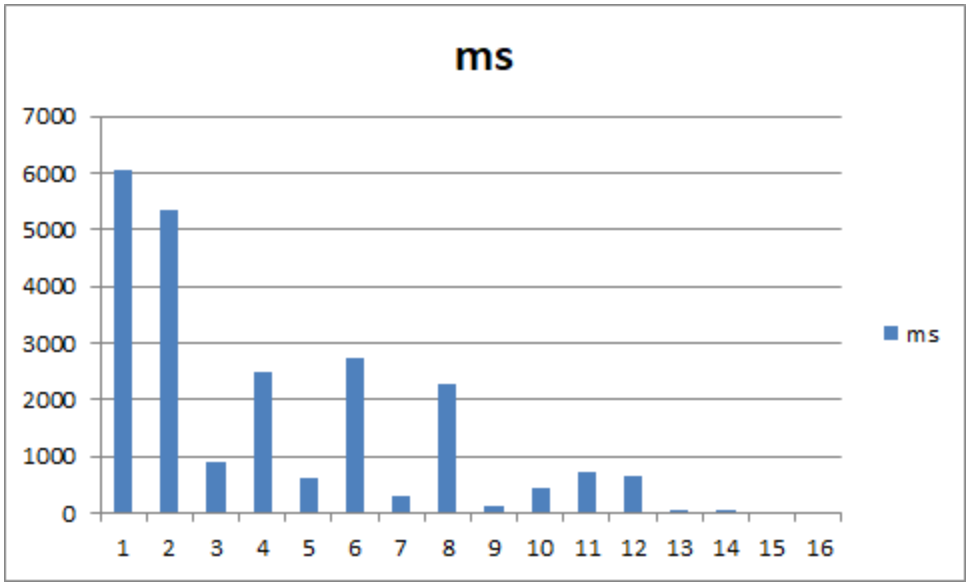
Sebelum kami melaporkan hasil eksperimen, kami akan memaparkan bobot heuristik yang kami gunakan dibawah ini,

$$w^x = \{1, 1, 4, 4, 8, 640, 640\}$$

$$w^y = \{1, 1, 4, 4, 8, 1280, 1280\}$$

Setelah itu kami menggunakan fitur *Input/Output* pada *Analysis* di Ludi *Player* untuk mendapatkan waktu algoritma *Minimax* bekerja setiap *turn*-nya. Dengan *range* waktu dari 0ms hingga 6052ms kami mendapatkan rata-rata waktu di angka 1424.625ms per *turn*.

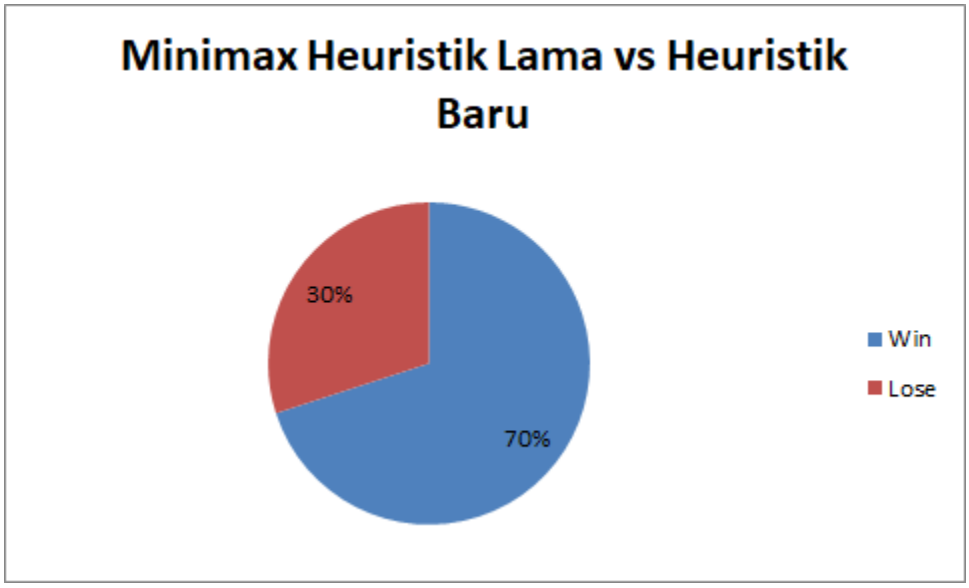
-



Gambar 5.n

***Minimax Alpha-Beta* bobot heuristik lama *vs* bobot heuristik baru**

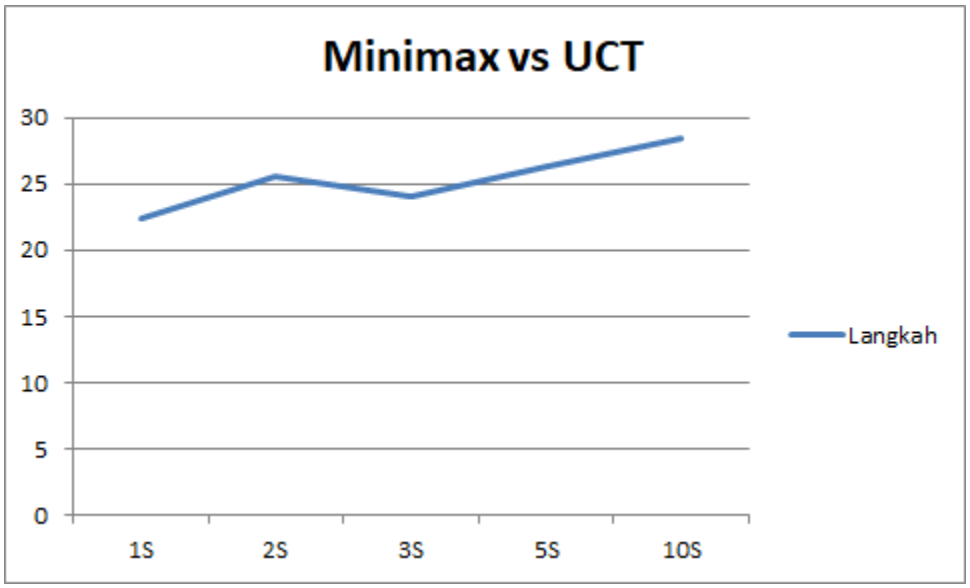
Eksperimen dilakukan sebanyak 10 kali(bertukar warna setelah 5 kali percobaan) dan Algoritma *Minimax Alpha-Beta* dengan bobot heuristik lama mendapatkan rasio kemenangan sebesar 70%. Kemudian terdapat juga rata-rata *turn* per *game* ialah 37.5 langkah per *game*.



Gambar 5.o

6. Kesimpulan

Pada percobaan yang kami lakukan terhadap beberapa algoritma yang tersedia pada Ludii AI Player. Kami dapat menyimpulkan beberapa kesimpulan yang kami dapat dari eksperimen ini. Pertama ialah menurut data rata-rata langkah tiap game Minimax melawan UCT 1 detik hingga 10 detik menunjukan tren kenaikan pada tiap waktu UCT ditingkatkan.



Gambar 6.a

Kemudian kesimpulan yang bisa kami ambil dari hasil eksperimen algoritma *Minimax Alpha-Beta Pruning* melawan UCT 1 detik hingga 10 detik, bahwa algoritma kami lebih optimal dibanding UCT 1 sampai 10 detik. Walaupun pada *Minimax* vs UCT 2 detik terdapat kekalahan tetapi angka rasio kemenangannya masih cukup tinggi yaitu di 90% dan kekalahannya terjadi bukan karena kesalahan fatal pada algoritma yang menyebabkan kekalahan cuma-cuma. Tetapi kekalahan dihasilkan dari permainan yang cukup lama dan jumlah kuda sudah berkurang banyak sehingga tidak ada gerakan yang bisa menghindari kekalahan.

Lalu kami pun mendapat kesimpulan bahwa algoritma kami masih belum cukup optimal untuk dapat menang melawan algoritma *Alpha-Beta* Ludii. Kesimpulan kami tarik dari kecilnya rasio kemenangan di 10% dan algoritma dan heuristik lawan begitu efisien hingga dalam waktu 1 detik sudah menyelesaikan *complete search* pada *node*.

Eksperimen terakhir kami ialah menguji algoritma *Minimax Alpha-Beta Pruning* kami dengan algoritma *Minimax* dengan bobot heuristik yang berbeda. Kemudian dari hasil eksperimen kami dapat menyimpulkan bahwa algoritma kami dengan bobot heuristik yang lama masih lebih optimal dibanding dengan bobot heuristik yang baru. Kami mengambil kesimpulan ini berdasarkan rasio kemenangan oleh bobot heuristik lama diatas 50% yaitu di 70%.

7. Referensi

GeeksforGeeks:
https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/
Java:
https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadLocalRandom.html
StackOverflow:
https://stackoverflow.com/questions/23396033/random-over-threadlocalrandom
Ludii:
https://ludiitutorials.readthedocs.io/en/latest/
https://github.com/Ludeme/LudiiExampleAI
https://github.com/Ludeme/LudiiAI
https://github.com/Ludeme/LudiiExampleAI/blob/master/src/experiments/Tutorial.java
https://github.com/Ludeme/LudiiExampleAI/blob/master/src/mcts/ExampleUCT.java
https://github.com/Ludeme/LudiiTutorials/blob/master/src/ludii_tutorials/RunningTrials.java
https://github.com/Ludeme/LudiiExampleAI/blob/master/src/random/RandomAI.java
https://github.com/Ludeme/LudiiAI/blob/master/AI/src/search/minimax/AlphaBetaSearch.java
Medium:
https://medium.com/@kamalovotash/nodes-in-trees-data-structure-c354e98b42d5
Wikipedia:
https://en.wikipedia.org/wiki/State_(computer_science)
https://en.wikipedia.org/wiki/Vertex_(graph_theory)
Java Decompiler:
http://java-decompiler.github.io/
Artificial Intelligence A Modern Approach oleh Stuart J. Russell and Peter Norvig
Youtube - CrackConcepts:
https://www.youtube.com/watch?v=_i-lZcbWkps&t=173s&ab_channel=CrackConcepts
Referensi Kelas:
PPT kelas PSC Adversarial Search Universitas Katolik Parahyangan oleh Lionov,PH.D.
Referensi Lainya:
https://www.wholesalechess.com/blog/history-and-tips-about-chess-knights/
https://www.britannica.com/technology/agen