TUGAS BESAR 1 IF3170 INTELEGENSI BUATAN
SIMPLEXITY MINIMAX ALGORITHM AND ALPHA BETA PRUNING AND LOCAL SEARCH IN SIMPLEXITY BOARD GAME

Kelompok Jeffrey Preston Bezos:

Hizkia Raditya Pratama Roosadi	13519087
Nathaniel Jason	13519108
Rizky Anggita Syarbaini Siregar	13519132
Richard Rivaldo	13519185

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2021

1. Algoritma *Minimax* dengan *Alpha Beta Pruning* pada *Simplexity*

Algoritma *Minimax* yang diimplementasikan pada permainan *Simplexity* ini terdiri dilengkapi dengan *Alpha Beta Pruning* atau penghentian pencarian untuk cabang-cabang yang sudah pasti tidak lebih baik. Algoritma *Minimax* yang digunakan dalam program dimodifikasi sedemikian rupa agar memiliki kecepatan yang efisien untuk kasus-kasus ukuran pohon yang besar dimana pencarian melebihi 3 detik (waktu berpikir maksimal bot yang tertera pada spesifikasi). Modifikasi yang dilakukan akan dijelaskan lebih lanjut pada bagian-bagian selanjutnya.

a. Fungsi Utilitas dan Heuristik yang Digunakan

Fungsi utilitas atau Objective Function yang digunakan pada algoritma *Minimax* memiliki tujuan untuk mengevaluasi kondisi papan permainan ketika sudah mencapai *final/goal state*. Kondisi yang mungkin dihasilkan pada *final state* adalah menang, kalah dan seri. Secara umum,

UtilityFunction(state) = { n * (42-jumlah kotak terisi + 1) }

Dengan n bernilai 1, -1, dan 0 untuk kondisi dimana state tersebut menghasilkan kemenangan, kekalahan, atau seri bagi pemain/bot secara berturut-turut.

Utility ini dipilih karena untuk meminimumkan jumlah round atau lamanya permainan, maka dibutuhkan sebuah fungsi yang dinamis mengikuti jalannya permainan. Kondisi n = {1, -1, 0} adalah bentuk standar kondisi akhir sebuah game.

Yang membedakan adalah nilai pengali atau faktor pengali yang digunakan pada utility function ini. Faktor pengali (42 - jumlah kotak terisi + 1) akan berdampak terhadap performansi bot yang kita miliki, dimana bot yang memiliki nilai pengali atau faktor pengali yang lebih besar akan lebih diprioritaskan untuk dipilih. Maksudnya adalah, jika ada dua state yang sama-sama menghasilkan kemenangan/kekalahan, maka state yang memiliki langkah terkecil (jumlah kotak terisi/number of round lebih sedikit), maka state inilah yang akan dipilih.

Hal lain yang juga mempengaruhi performansi bot adalah karena menggunakan algoritma Minimax dengan Alpha-Beta Pruning, maka cabang/branch dari sebuah tree yang nilai beta-nya lebih kecil atau sama dengan alpha, maka branch tersebut tidak akan dievaluasi sehingga kinerja bot semakin optimal.

b. Algoritma Minimax dengan *Alpha Beta Pruning*

Pada permainan Simplexity Board, terdapat dua orang pemain, misal Putih dan Hitam. Keduanya di-assign dengan sebuah shape, dalam kasus ini misal Kotak-Putih dan Lingkaran-Hitam. Lebih lengkap terkait aturan main dan kondisi kemenangan (game over) dapat dilihat di spesifikasi.

Misal pemain giliran pertama adalah pemain putih, seperti pada gambar di bawah ini. Terdapat 14 buah kemungkinan untuk menaruh bidak, dalam algoritma dinyatakan sebagai state, yang mungkin dijalankan oleh pemain putih. Dia dapat menaruh 2 kemungkinan shape (kotak atau lingkaran) dan menaruhnya pada 7 buah kolom. Sehingga terdapat 14 buah kemungkinan (state) yang mungkin untuk setiap turn.

Ketika pemain putih sudah menentukan pilihannya, pemain hitam juga memiliki 14 kemungkinan untuk merespons dan menaruh bidaknya ke papan. Demikian seterusnya hingga mencapai daun, dimana kondisi tersebut merupakan kondisi di antara 3 state, yaitu menang, kalah, atau seri.

Pada algoritma minimax, terdapat dua buah pihak yaitu pihak yang memaksimalkan (maximizing) dan yang meminimumkan (minimizing). Maksud meminimumkan atau memaksimumkan adalah terkait dengan value atau nilai dari langkah pihak tersebut. Misal pada kasus ini, pihak putih adalah pihak maximizing dan pihak putih adalah pihak minimizing.

Jika dilakukan evaluasi value sebuah state pada daun, dengan menggunakan utility function seperti pada nomor 3, dengan utility function,

UtilityFunction(state) = { n * (42-jumlah kotak terisi + 1) }
dengan n ={ -1, 1, 0}

Maka akan didapatkan value yang bervariasi antara setiap state. Jika menggunakan algoritma minimax, maka dapat dipastikan bahwa akan ditemukan best move atau gerakan terbaik yang menghasilkan kemenangan bagi bot yang dibuat.

Namun terdapat sebuah permasalahan, yaitu banyaknya jumlah state yang harus dievaluasi oleh bot tersebut. Seperti sudah disebutkan sebelumnya, setiap turn terdapat 14 buah state, dan setiap state tersebut memiliki 14 child state lagi, dan begitu seterusnya. Terdapat 14^{42} state pada daun, yaitu sebesar 1.3 * 1048, yang akhirnya harus dievaluasi untuk

menentukan value pada state tersebut. Jika hanya menggunakan minimax, solusi hampir tidak mungkin diraih, dikarenakan besarnya kompleksitas ruang dan waktu yang dibutuhkan untuk mengevaluasi 14^{42} state.

Solusinya adalah menggunakan Minimax Algorithm with Alpha-Beta Pruning. Konsepnya adalah menghindari untuk mengevaluasi state dimana state tersebut sudah pasti tidak diambil, yaitu dengan membandingkan value sebuah state dengan alpha atau beta. Sehingga, sub-pohon state yang tidak mungkin diambil akan dibuang atau dilakukan pruning yang mengakibatkan pencarian solusi menjadi jauh lebih cepat.

Untuk menentukan state mana yang akan dipilih, hal ini berdasarkan pada Utility Function/Objective Function yang sudah dijelaskan sebelumnya. Intinya adalah, ketika state tersebut merupakan daun, yaitu kondisi game berakhir, maka akan dilakukan evaluasi menggunakan objective function tersebut. Untuk meningkatkan performansi, untuk dua atau lebih state yang memiliki kondisi sama, akan dibedakan value state tersebut tergantung jumlah kotak yang terisi atau banyaknya round yang sudah dilewati. State dengan kotak terisi lebih sedikit, yang menandakan bahwa state tersebut lebih efisien dibanding state lainnya, maka akan memiliki value yang lebih besar (bisa positif atau negatif, tergantung siapa yang menang). Jika state tersebut menghasilkan seri, maka value bernilai 0.

Kemudian bot, dengan algoritma minimax with alpha-beta pruning, maka akan mengambil nilai minimum atau maksimum, tergantung giliran siapa yang sedang dievaluasi, untuk menemukan langkah terbaik dalam menaruh bidak di papan permainan. Misal putih adalah pihak maximizing, maka jika terdapat dua pilihan child dengan masing-masing value bernilai 5 dan 10, maka putih akan mengambil nilai 10 karena menandakan bahwa state tersebut memiliki kondisi kemenangan bagi putih dan memiliki langkah yang lebih sedikit dibanding state dengan value 5, walau sama-sama menghasilkan kemenangan. Begitu juga dengan pihak hitam yang minimizing, maka jika terdapat dua buah state dengan masing-masing value adalah -5 dan 1, maka hitam akan memilih -5 karena lebih minimum dibanding state yang lain. Begitu seterusnya sehingga bot, dengan kondisi state tertentu, dapat menghasilkan kemenangan dengan langkah paling sedikit sehingga efisien.

Namun terdapat sebuah permasalahan, yaitu waktu pengambilan keputusan yang diharapkan yaitu di bawah 3 detik (waktu berpikir maksimal bot yang tertera pada spesifikasi). Ada beberapa modifikasi yang kami tambahkan agar dapat memastikan bahwa hasil yang diberikan

oleh *bot* mendekati optimal untuk kasus dimana *searching* tidak menyeluruh karena batasan waktu berpikir. Modifikasi utama yang kami lakukan untuk menangani kasus waktu berpikir yang lama adalah dengan mengembalikan nilai. Hal ini ditujukan agar *bot* tidak memberikan hasil yang kosong setelah berpikir. Terdapat dua pendekatan yang kami ambil untuk modifikasi yang dilakukan. Secara umum, pendekatan pertama diterapkan dengan melakukan modifikasi dengan melakukan ekspansi cabang secara acak. Untuk pendekatan kedua, dilakukan modifikasi dengan mengembalikan nilai yang terbaik serta cabang diekspansi dengan urutan nilai heuristik dari yang tertinggi.

i. Pendekatan Pertama

Pada pendekatan pertama, modifikasi yang dilakukan adalah dengan melakukan ekspansi cabang dengan acak. Karena waktu berpikir terbatas, maka tidak semua cabang yang dapat diekspansi. Jika cabang yang akan diekspansi tidak diacak, maka dapat dipastikan hampir dalam semua kasus permainan, untuk *turn* awal dimana ukuran pohon sangat besar, langkah yang diambil oleh *bot* selalu sama. Pada implementasi program, fungsi untuk menghasilkan cabang yang dapat diekspansi, atau *move* yang valid, mengembalikan suatu larik. Urutan *move* yang valid pada larik tersebut dimulai dari kolom paling kiri sampai kolom paling kanan, sehingga jika tidak diacak, maka untuk permainan pada fase awal, *bot* akan meletakkan bidak pada kolom paling kiri.

ii. Pendekatan Kedua

Modifikasi kedua yang kami lakukan adalah dengan melakukan searching pada cabang yang memiliki kecenderungan nilai yang lebih baik. Nilai ini kami dapatkan dengan mengevaluasi state dengan fungsi utilitas yang berbeda dengan yang digunakan untuk evaluasi node daun atau saat final state. Fungsi utilitas yang digunakan sama seperti fungsi utilitas pada algoritma Local Search yang kami terapkan. Pembahasan lengkap dari fungsi utilitas yang kami gunakan untuk evaluasi node bukan daun ada pada bagian selanjutnya dari laporan ini. Modifikasi ini dilakukan agar proses searching yang dilakukan oleh bot setidaknya mengarah ke cabang-cabang yang optimal. Untuk kasus white player atau saat turn pemain, state tetangga dievaluasi terhadap sudut pandang white player sehingga neighbor state yang akan diekspansi diurutkan dari

nilai yang paling tinggi. Untuk kasus black player atau saat turn musuh, state tetangga dievaluasi terhadap sudut pandang black player dan diurutkan dari nilai yang paling tinggi. Untuk kasus black player, state tetangga juga dapat dievaluasi terhadap sudut pandang white player, tetapi bedanya neighbor state akan diurutkan dari nilai yang paling kecil. Hal ini dilakukan agar memastikan bahwa dalam proses pencarian musuh adalah musuh yang optimal.

Modifikasi kedua yang kami lakukan adalah dengan mengembalikan nilai dan move terbaik sementara jika waktu berpikir habis. Ketika waktu sudah habis, maka proses pencarian diberhentikan dan dikembalikan move dengan nilai tertinggi untuk kasus maximizing atau move dengan nilai terendah untuk kasus minimizing. Move yang dikembalikan bisa saja bukanlah move dengan nilai yang paling tinggi atau rendah sebenarnya. Hal ini dilakukan agar setidaknya nilai atau move yang dikembalikan oleh bot adalah yang terbaik sejauh ini.

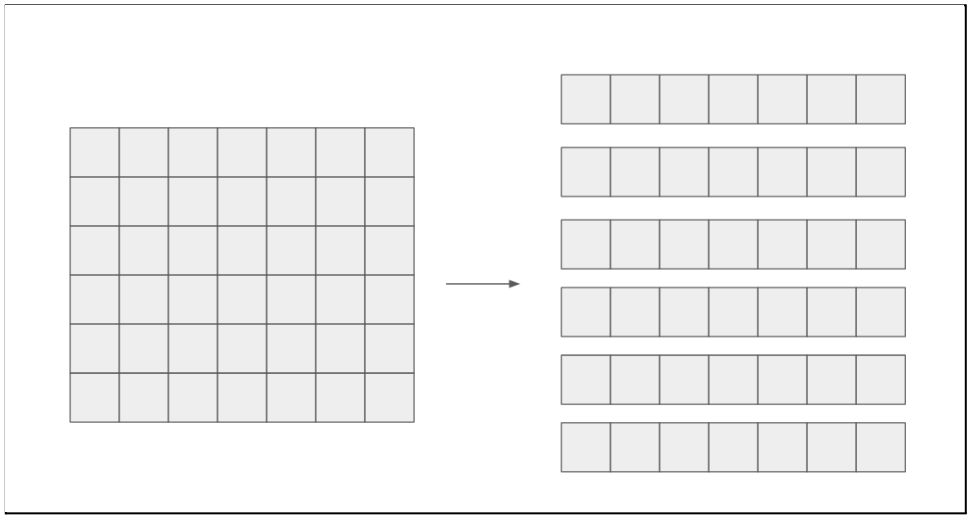
2. Algoritma *Local Search* pada *Simplexity*

Algoritma *Local Search* yang diimplementasikan pada permainan *Simplexity* ini terdiri dari dua jenis, yaitu *Hill Climbing* dan *Simulated Annealing*. Masing-masing algoritma akan dimodifikasi untuk memenuhi kebutuhan dari permainan *Simplexity*. Adapun pembangunan dua algoritma dimaksudkan untuk memberikan kemungkinan alternatif pilihan pencarian meskipun heuristik yang digunakan pada keduanya adalah sama.

a. Fungsi Utilitas dan Heuristik yang Digunakan

Heuristic Function akan digunakan untuk mengevaluasi ‘nilai’ dari suatu *state*. Semakin besar nilai dari Heuristic Function maka kita dapat mengatakan bahwa *state* tersebut semakin baik. Maksud dari baik disini adalah menuju kemenangan pemain. Secara garis besar Heuristic Function akan menghitung *state value* berdasarkan seberapa dekat dengan kemenangan dan seberapa dekat dengan kekalahan. Heuristic Function akan memberi nilai positif jika dekat dengan kemenangan dan negatif jika dekat dengan kekalahan. Heuristic Function akan mengevaluasi semua macam kemungkinan kemenangan yang ada, mulai dari arahnya (vertikal, horizontal, diagonal dengan kemiringan positif, dan diagonal dengan kemiringan negatif).

Contoh perhitungan nilai pada Heuristic Function adalah sebagai berikut. Misal kita akan hitung *state value* pada arah horizontal. Kita pertama - tama akan membagi papan sesuai dengan arah yang kita pilih, yaitu horizontal seperti pada gambar berikut.



Setiap potongan horizontal akan dievaluasi secara terpisah. Selanjutnya kita akan mengevaluasi untuk setiap 4 kotak berdampingan atau grup yang mungkin pada potongan tersebut. Angka 4 untuk jumlah kotak berdampingan dipilih karena dibutuhkan 4 kotak untuk meraih kemenangan pada permainan. Pembagiannya dapat dilihat pada gambar berikut.

Pada setiap grup yang akan dievaluasi, kita akan menghitung nilainya berdasarkan bidak - bidak yang ada didalamnya. Perhitungan nilai akan mengikuti kondisional sebagai berikut.

```
if (grup memiliki 4 bidak) then
        if (grup memiliki 4 bidak dengan shape lawan) then
              value = negatif infinity
        else if (grup memiliki 4 bidak dengan shape pemain) then
              value = 100.000 (nilai positif yang sangat besar
        tapi kurang dari infinity)
```

else if (grup memiliki 3 *piece* dengan *shape* atau *color* lawan) **then**

value = 100.000 (nilai positif yang sangat besar tapi kurang dari infinity) **#ini berarti musuh di block oleh move kita**

else if (grup memiliki 2 *piece* dengan *shape* atau *color* lawan) **then**

value = 0

else:

value = 4

else if (grup memiliki 1 kotak kosong dan 3 bidak) **then**

if (grup memiliki 3 bidak dengan *shape* atau *color* lawan) **then**

value = negatif infinity

else if (grup memiliki 3 bidak dengan *shape* atau *color* pemain) **then**

value = 10

else if (grup memiliki 2 bidak dengan *shape* atau *color* pemain) **then**

value = 4

else: # 2 bidak lawan

value = -4

else if (grup memiliki 2 kotak kosong dan 2 bidak) **then**

if (grup memiliki 2 bidak dengan *shape* atau *color* lawan) **then**

value = -4

else

value = 4

else if (grup memiliki 3 kotak kosong dan 1 bidak) **then**

if (grup memiliki 1 bidak dengan *shape* atau *color* lawan) **then**

value = -1

else

```
                value = 1
#if there are no pieces
value = 0
```

Dapat dilihat bahwa untuk kasus dimana terdapat 4 bidak dengan *shape* lawan, maka sudah dapat dipastikan kita kalah, sehingga kita menambahkan *value* dengan nilai negatif *infinity*, dengan tujuan agar *state* yang sedang dihitung ini tidak dipilih. Hal yang sama terjadi ketika terdapat 3 bidak dengan *shape* atau *color* lawan, karena dengan begitu pada ronde selanjutnya yang dimana lawan akan jalan, maka lawan sudah pasti menang karena tinggal meletakan satu bidak yang sesuai.
Proses perhitungan seperti ini akan dilakukan untuk perhitungan *value* untuk arah horizontal, vertikal, diagonal dengan kemiringan positif, dan diagonal dengan kemiringan negatif.

b. Algoritma *Hill Climbing*

Algoritma *Hill Climbing* merupakan salah satu algoritma dasar yang ada pada kategori algoritma *Local Search*. Secara *default,* algoritma *Hill Climbing* dilakukan dengan menggunakan varian *Steepest Ascent*, yaitu me-*generate* semua kemungkinan suksesor yang ada, lalu selalu memilih suksesor dengan nilai utilitas tertinggi yang lebih baik dari keadaan sekarang dan kemudian berhenti saat tidak ada lagi suksesor yang lebih baik dari *state* saat ini.
Algoritma ini dapat dikatakan sebagai algoritma yang terlalu sederhana karena kemungkinan untuk terjebak pada *local maximum* seperti *shoulder* dan *flat* sangat tinggi. Oleh karena itu, dari algoritma ini kemudian muncul banyak variasi algoritma *Hill Climbing* lainnya. Salah satu algoritma yang dimaksud adalah *Hill Climbing with Sideways Move*.
Meskipun membutuhkan waktu yang cenderung lebih lama dibandingkan varian *Steepest Ascent*, varian ini dapat dikatakan sebagai varian yang lebih baik. Pada varian ini suksesor mungkin dipilih dari *state* lain yang memiliki nilai utilitas sama dengan keadaan sekarang. Hal ini meningkatkan kemungkinan agen tidak terjebak pada *shoulder* dan mampu mencapai *global maximum*. Algoritma jenis ini yang dipilih untuk diimplementasikan (dengan sedikit modifikasi) untuk permainan *Simplexity*.
Pertama, akan dicari terlebih dahulu semua kemungkinan gerak yang bisa dilakukan agen sesuai kondisi papan pada saat tertentu. Hal ini kemudian

menghasilkan daftar suksesor dari status tersebut. Kemudian, untuk setiap gerakan yang mungkin, dilakukan *copy* kondisi papan pada saat ini untuk kemudian mengaplikasikan gerakan tersebut pada papan yang telah disalin. Hal ini ditujukan agar papan yang ada pada saat ini tidak rusak atau berubah.

Setelah papan disalin dan diaplikasikan gerakan yang sedang dicek, maka akan dilakukan perhitungan nilai utilitas status baru dengan heuristik yang telah dijelaskan pada bagian sebelumnya. Nilai utilitas ini kemudian dicatat dan disimpan untuk setiap gerakan. Dari daftar gerakan dan nilai evaluasi yang ada, maka akan selalu diambil gerakan dengan nilai evaluasi terbesar.

Secara tidak langsung, hal tersebut akan membuat agen mampu melakukan *sideways move* apabila nilai utilitas maksimal dari daftar suksesor yang ada adalah sama dengan kondisi saat ini. Hal ini kemudian menyebabkan tidak diperlukannya evaluasi heuristik terhadap kondisi papan saat ini.

Hal tersebut juga mengeliminasi adanya kemungkinan terminasi saat tidak dihasilkannya gerakan yang lebih baik saat ini, yang tentunya tidak mungkin diaplikasikan untuk permainan *Simplexity* ini. Selain itu, juga disediakan *sentinel* pada implementasi program yang memilih gerakan secara acak jika tidak ada evaluasi yang dilakukan oleh agen.

c. Algoritma *Simulated Annealing*

Meskipun algoritma *Hill Climbing with Sideways Move* telah memberikan pencarian yang lebih luas, tetapi pada dasarnya algoritma *Hill Climbing* bukanlah algoritma yang *complete*. Dengan mengkombinasikan algoritma *Hill Climbing* yang efisien dengan algoritma *Random Walk* yang *complete*, maka kemudian terbentuklah algoritma yang dikenal dengan *Simulated Annealing*.

Algoritma *Simulated Annealing* ini terinspirasi dari proses *annealing* pada industri, yaitu pemanasan besi yang kemudian dibiarkan mendingin secara perlahan. Algoritma ini menggunakan sebuah *temperature* yang dibuat dengan menggunakan fungsi waktu. Nilai dari *temperature* ini akan terus menurun sampai mencapai 0 dan memberhentikan pencarian.

Selama pencarian berlangsung, dipilih satu suksesor secara acak untuk dievaluasi apabila menghasilkan status yang lebih baik. Evaluasi dilakukan dengan menghitung *delta E*, yaitu selisih nilai heuristik status

suksesor dengan status saat ini. Jika nilai *delta E* positif, maka suksesor langsung dipilih sebagai *next state*.
Jika sebaliknya, maka kemudian dihitung nilai eksponen dari perbandingan *delta E* dengan nilai temperatur pada saat itu. Jika nilai ini melebihi *threshold* tertentu, maka agen bisa melakukan gerakan ke suksesor tersebut. Hal ini memungkinkan adanya gerakan *downhill* yang memungkinkan lebih banyak pencarian untuk status permainan.
Pada permainan *Simplexity*, fungsi waktu yang digunakan merupakan selisih dari waktu yang telah dialokasikan dengan waktu program saat ini. Adapun alokasi waktu yang dimaksud adalah total waktu saat agen dipanggil ditambahkan dengan waktu berpikir tiga detik. Selama waktu berpikir belum habis, maka agen akan memilih suksesor secara acak dari status saat itu.
Kemudian, akan dilakukan perhitungan nilai *delta E* untuk kedua status. Nilai ini dihitung dengan mekanisme yang mirip seperti pada *Hill Climbing*, yaitu dengan menyalin kondisi papan saat ini dan mengaplikasikan gerakan tersebut. Bedanya, pada teknik ini diperlukan evaluasi status saat ini untuk dicari selisihnya terhadap suksesor yang sedang diperiksa.
Jika *delta E* yang dihasilkan bernilai positif, maka kemudian dicek lagi apabila *delta E* saat ini lebih besar dari nilai gerakan maksimum pada saat itu, yang diinisialisasi dengan nilai yang sangat kecil pada awal algoritma. Jika lebih besar, maka gerakan dan nilai evaluasi maksimum pada saat itu digantikan dengan nilai yang didapat untuk gerakan yang sedang diperiksa. Hal serupa juga terjadi saat nilai *delta E* bernilai negatif, yaitu dihitung eksponen yang telah disebutkan sebelumnya lebih besar dari probabilitas yang telah ditetapkan.
Jika temperatur sudah bernilai kurang dari atau sama dengan batas waktu yang ditentukan, dalam hal ini 10^{-4}, maka algoritma akan mengembalikan gerakan yang dicatat pada saat itu. Jika tidak ada gerakan yang dicatat, maka kemudian dipilih gerakan secara acak dari status papan saat itu.

Dari kedua penjelasan di atas, dapat dilihat bahwa terdapat cukup banyak modifikasi yang dilakukan baik pada implementasi algoritma *Hill Climbing* maupun *Simulated Annealing* untuk menyesuaikan algoritma *Local Search* dengan permainan *Simplexity* ini. Umumnya, pada kedua

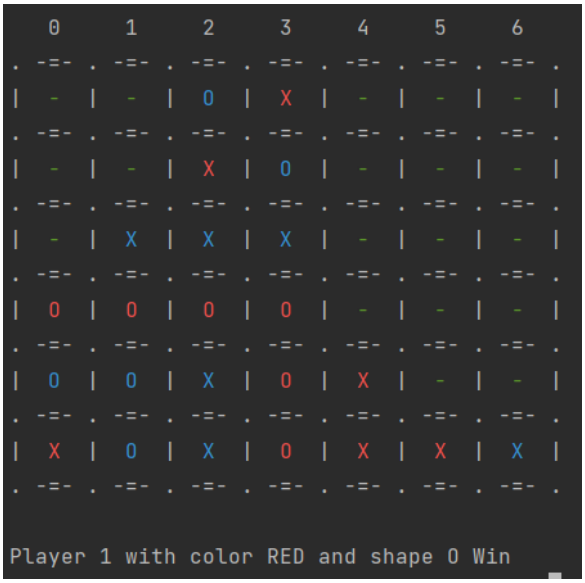
gerakan ini disisipkan sifat *greedy* agar agen bisa memilih aksi terbaik yang memberikan *neighbor* dengan nilai evaluasi yang paling baik.

3. Analisis Hasil Pertandingan Bot yang Dibangun

a. Bot *Minimax* vs Bot *Local Search*

i. Bot *Minimax First Move*

Pada bagian ini, bot *Minimax* dijadikan sebagai pemain yang pertama kali melakukan gerakan. Ketika dihadapkan dengan bot *Local Search* dengan algoritma *Hill Climbing*, maka didapatkan hasil bahwa agen *Minimax* berhasil menang telak terhadap bot *Hill Climbing*. Berikut merupakan tangkap layar untuk salah satu hasil status akhir dari percobaan ini.



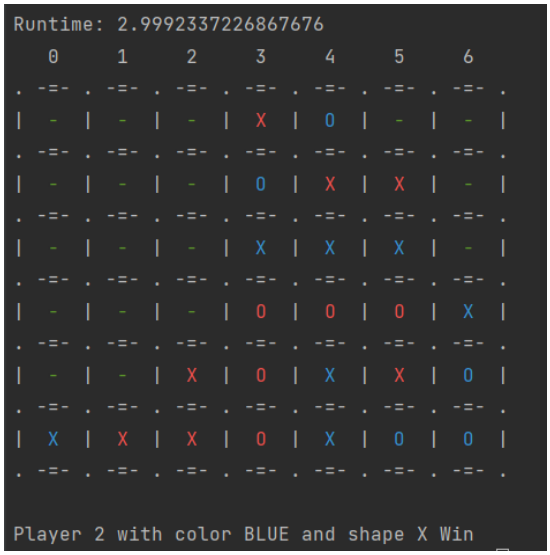
Minimax First Move vs Local Search Hill Climbing

Hal tersebut sangat mungkin terjadi karena algoritma *Hill Climbing* merupakan algoritma yang cukup dangkal dalam implementasinya. Algoritma yang dapat terbilang cukup sederhana karena hanya mengimplementasikan prinsip *greedy* tanpa gerakan *downhill* sama sekali juga merupakan salah satu kemungkinan penyebab hal ini dapat terjadi.

Namun, hal sebaliknya terjadi ketika agen *Minimax* dihadapkan pada agen *Local Search* dengan algoritma *Simulated Annealing*. Pada pertandingan ini, hasil yang didapat adalah 60% *win rate* untuk agen *Simulated Annealing*.

Hal tersebut menandakan bahwa algoritma *Simulated Annealing* lebih baik dibandingkan algoritma *Hill Climbing*. Kemungkinan penyebab dari hal ini adalah karena algoritma *Simulated Annealing* memberikan kemungkinan gerakan *downhill*, sehingga menyebabkan pencarian yang lebih *complete* dibandingkan dengan *Hill Climbing*. Berikut merupakan beberapa hasil tangkap layar dari pertandingan yang dilakukan.

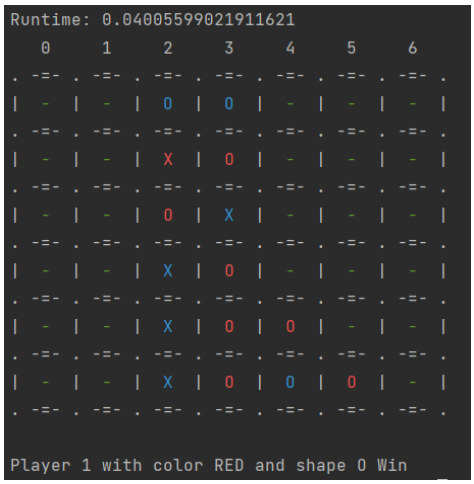
Minimax First Move vs Simulated Annealing: Win



Minimax First Move vs Simulated Annealing: Lose

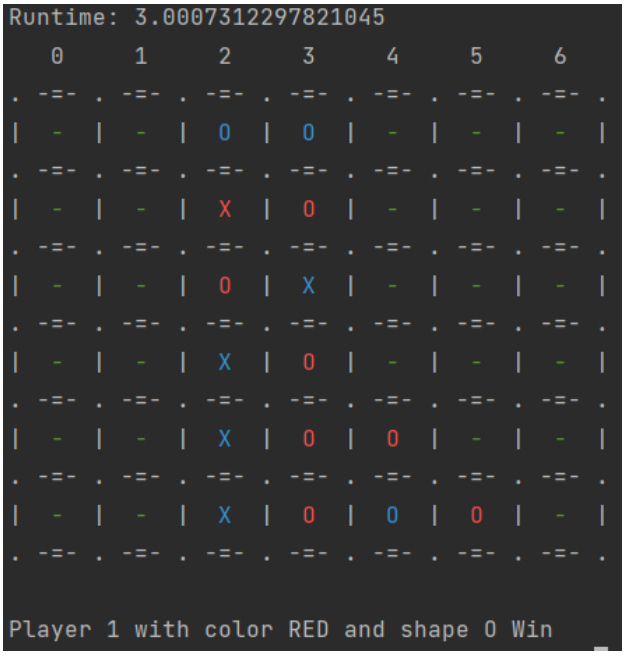
ii. Bot *Local Search First Move*

Pada saat agen *Local Search* ditentukan sebagai agen yang melakukan gerakan pertama, hasil yang didapat pada pertandingan yang diujikan cukup mengejutkan. Pertama, untuk agen *Local Search* yang dibuat dengan menggunakan algoritma *Hill Climbing*, agen *Hill Climbing* berhasil menang telak terhadap agen *Minimax*. Hal tersebut mungkin terjadi karena agen *Minimax* mungkin belum bisa menentukan pilihan gerakan yang paling optimal untuk waktu berpikir selama 3 detik. Di sisi lain, algoritma *Hill Climbing* memberikan keputusan gerakan dalam waktu yang cukup singkat, seperti yang dapat dilihat pada tangkap layar berikut.



Local Search First Move: Hill Climbing

Hasil yang serupa juga didapatkan pada algoritma *Local Search* dengan *Simulated Annealing*. Agen *Minimax* kalah telak dengan agen *Local Search* versi ini. Namun, agen *Simulated Annealing* tentunya memberikan waktu eksekusi yang lebih lama karena waktu berpikir tiga detik digunakan sepenuhnya. Berikut merupakan tangkap layar dari salah satu *state* akhir pertandingan kedua agen.



Local Search First Move: Simulated Annealing

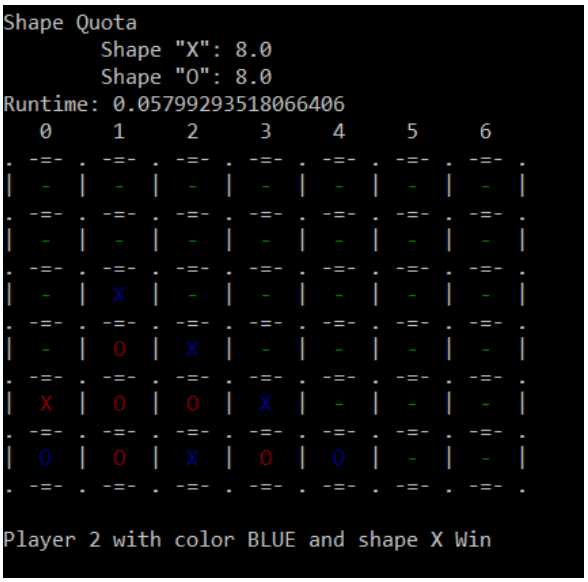
Dari beberapa uji coba yang dilakukan, dapat juga diketahui bahwa ternyata urutan agen yang memulai pertandingan sangat menentukan keberjalanan pertandingan. Agen yang memulai permainan lebih awal tentunya memiliki pilihan lebih dalam mengendalikan permainan melalui *piece* pertama yang diletakkannya.

b. Bot *Local Search* vs Manusia

i. Manusia *First Move* Bot *Hill Climbing*

Dari hasil uji coba bot local search melawan manusia, dengan kondisi manusia *move* pertama. didapatkan bahwa bot *hill climbing* memiliki persentase kemenangan sebesar 4:1 atau 80%. Hasil kemenangan ini cukup tinggi. Hasil kemenangan yang tinggi ini dapat dicapai karena bot memang berusaha mencari *move* yang selalu optimum secara lokal berdasar *state* dari board yang sedang dihadapi. Sedangkan, manusia memiliki cara berpikir yang dapat menyebabkan error / *move* yang pada akhirnya akan menghasilkan move yang tidak optimal. Dari hasil permainan, penulis juga mengobservasi bahwa pemain manusia biasanya hanya mempertimbangkan *piece* yang sesuai dengan kondisi

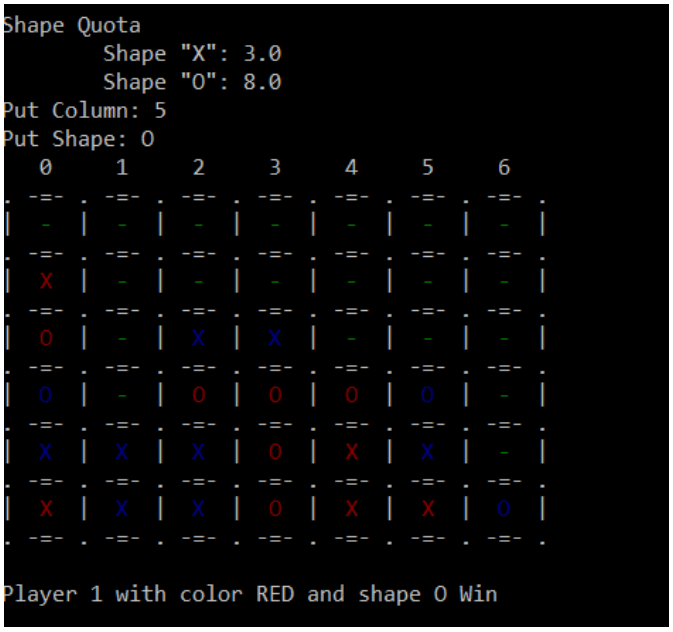
kemenangannya. Misal (O, Merah) sementara bot local search diatur sedemikian untuk mempertimbangkan juga pergerakan dari lawan (contoh: di program terdapat kondisional yang memberi nilai *state* buruk apabila lawan memiliki 3 *piece* dalam board berturut-turut). Hal ini juga berkontribusi ke error yang mungkin dilakukan manusia.



Player First Move Hill Climbing

ii. Manusia *Second Move* Bot *Hill Climbing*

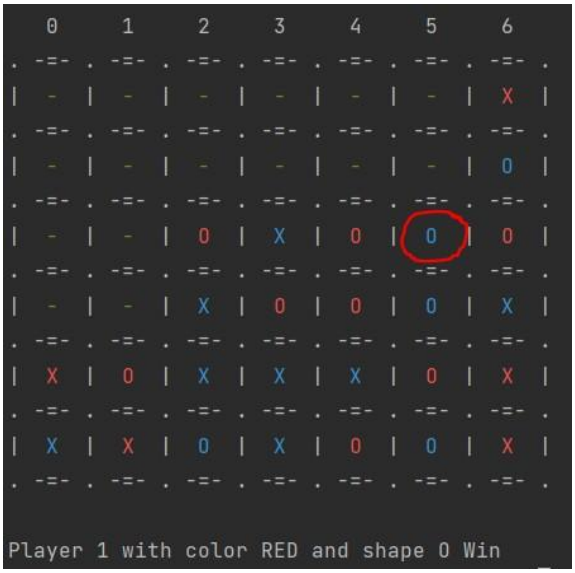
Hasil yang serupa dapat diobservasi bahkan ketika bot *hill climbing* bergerak pertama. Bot *hill climbing* memiliki persentase kemenangan yang cukup tinggi yaitu 80%. Hal ini juga dapat dicapai karena bot selalu memilih gerakan yang akan menghasilkan *state value* paling tinggi. Algoritma dari bot *hill climbing* dan heuristik yang digunakan diatur sedemikian sehingga bot akan selalu bergerak ke titik optimum lokal. Terjadinya *human error* juga bisa berkontribusi ke tingginya persentase kemenangan dari bot ini. Walaupun begitu, bot masih bisa kalah karena algoritma *hill climbing* selalu berusaha mencari *state value* paling tinggi dari momen sekarang sehingga kemungkinan terjadi *stuck* ketika mencapai optimum lokal. Dibandingkan titik optimum global dimana bot akan selalu menang, di titik optimum lokal, masih ada *move* dari manusia yang memungkinkan membuat bot kalah.



Player Second Move Hill Climbing

iii. Manusia *First Move* Bot *Simulated Annealing*

Ketika menggunakan algoritma *simulated annealing*, bot juga memiliki persentase kemenangan yang serupa yaitu 80%. Hal ini juga dapat terjadi karena beberapa faktor seperti bot selalu memilih gerakan yang akan menghasilkan *state value* paling tinggi. Ketika *neighbor state value* > *current state value* maka bot otomatis akan melakukan move ke *state* tersebut. Di sisi lain, bot masih bisa melakukan pergerakan yang pada akhirnya menghasilkan kekalahan karena algoritma *simulated annealing* dapat move secara random (dengan probabilitas yang kecil). Namun, pada mayoritas kasus, bot akan membuat move yang menuju titik optimum lokal.

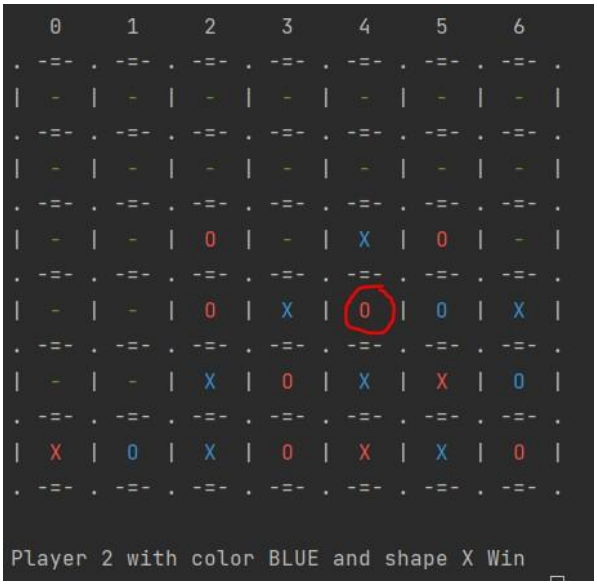


Manusia First Move Bot Simulated Annealing

Dari contoh kasus di atas, dapat dilihat bahwa bot juga dapat menilai *state value* sedemikian rupa sehingga pemain mengalami kesulitan dalam menentukan gerakan. Titik yang dilingkari adalah gerakan pemain manusia yang meletakkan bentuk O di kolom 5 yang adalah kondisi menang dari bot.

iv. Manusia *Second Move* Bot *Simulated Annealing*

Dari kasus uji coba, bot *simulated annealing* dengan *move* pertama juga menghasilkan persentase kemenangan yang serupa yaitu 80%. Penulis mengobservasi bahwa pemain manusia mengatakan bot sulit untuk dikalahkan. Terlebih lagi karena bot melakukan gerakan pertama, bot memiliki kontrol lebih dalam penentuan nilai *state* yang dapat menghasilkan titik optimum yang akan membuat bot menang. Bot masih bisa melakukan pergerakan yang pada akhirnya menghasilkan kekalahan karena algoritma *simulated annealing* dapat move secara random (dengan probabilitas yang kecil). Namun, pada mayoritas kasus, bot akan membuat move yang menuju titik optimum lokal.



Manusia Second Move Simulated Annealing

Dalam contoh kasus di atas, *piece* yang dilingkari adalah gerakan manusia untuk melakukan *block* pada bot sehingga tidak terjadi susunan 4 *piece* secara horizontal. Namun, bot dapat menggunakan penilaian state value untuk mencari bahwa move selanjutnya, jika ditambahkan X di kolom 4 akan menghasilkan 4 *piece* secara diagonal yang adalah kondisi menang.

c. Bot *Minimax* vs Manusia

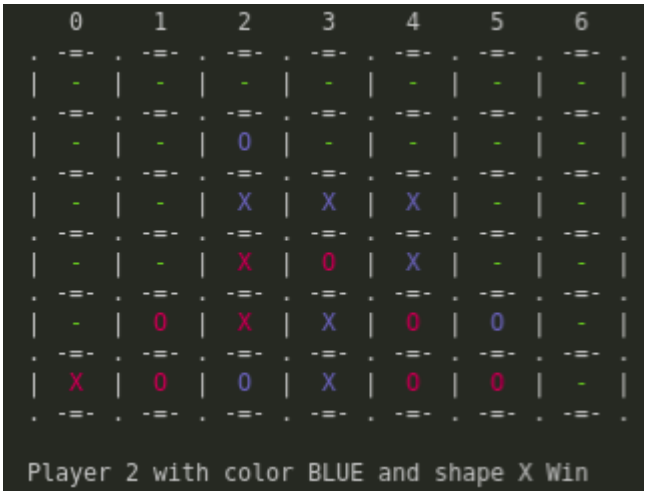
Pada bagian ini, dilakukan 5 kali percobaan masing-masing pada 4 buah jenis pertandingan, yaitu manusia *first move* atau *second move* melawan bot *minimax* pendekatan pertama atau pendekatan kedua. Sehingga pada bagian ini, terdapat 20 kali percobaan manusia vs bot minimax.

i. Manusia *First Move* Bot *Minimax* Pendekatan Pertama

Dari hasil uji coba manusia *first move* melawan bot minimax pendekatan pertama, didapatkan bahwa persentase kemenangan adalah 60:40, dimana manusia *first move* memiliki persentase kemenangan 60%.

Kondisi itu terjadi karena pada minimax dengan pendekatan pertama, urutan *state* yang akan dievaluasi pada setiap levelnya dilakukan pengacakan secara *random*, sehingga kinerja bot tidak

maksimal. Hal ini mengakibatkan ada kemungkinan bahwa state yang dievaluasi di awal-awal adalah state yang sebenarnya cukup buruk, yaitu state yang jika dievaluasi sampai ke daun memiliki nilai heuristik yang buruk. Juga dikarenakan waktu berpikir yang sangat sedikit, bot minimax belum selesai untuk mengevaluasi seluruh kemungkinan state hingga ke daun, sehingga bot akan mengembalikan *current best move* sebagai *move* yang akan dipilih. Gambar di bawah merupakan salah satu kemenangan bot minimax dari dua kemenangan yang dimilikinya.



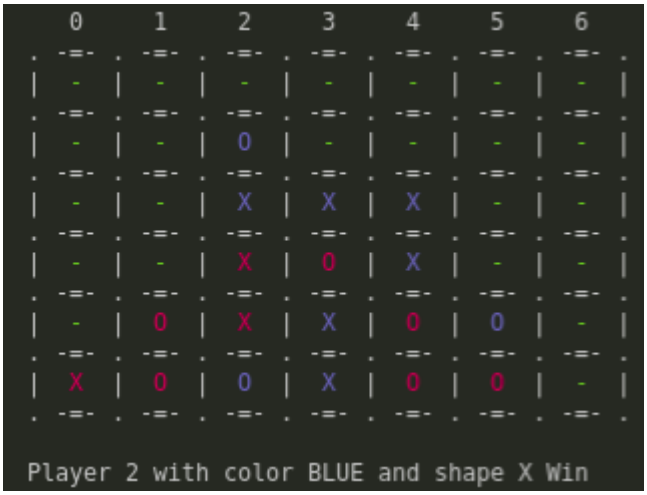
Manusia First Move* vs *Bot Minimax Pendekatan Pertama

ii. Manusia *Second Move* Bot *Minimax* Pendekatan Pertama

Dari hasil uji coba manusia *second move* melawan bot minimax pendekatan pertama, didapatkan bahwa persentase kemenangan adalah sama seperti pada bagian (i), yaitu 60:40, dimana manusia *second move* memiliki persentase kemenangan 60%.

Penjelasan mengapa hal tersebut terjadi sama seperti penjelasan pada bagian (i), yaitu karena ada peluang bot minimax mengevaluasi state yang memiliki heuristic value yang rendah terlebih dahulu. Penjelasan lain mengapa nilai *winrate* antara bagian (i) dan bagian (ii) ini adalah karena pada kasus manusia vs bot minimax, tidak terdapat perbedaan yang signifikan siapa terlebih dahulu yang menaruh bidak di papan. Hal ini dikarenakan bot minimax selalu menentukan keputusannya berdasarkan asumsi bahwa musuh yang dilawannya selalu menghasilkan keputusan

terbaik pada setiap langkah atau round, sehingga bot minimax akan memberikan *best move* juga untuk merespon *best move* yang dipilih lawannya. Sehingga urutan siapa yang terlebih dahulu menang tidak memberikan efek atau perubahan apapun terhadap jalannya pertandingan.



Manusia *Second Move* Bot *Minimax* Pendekatan Pertama

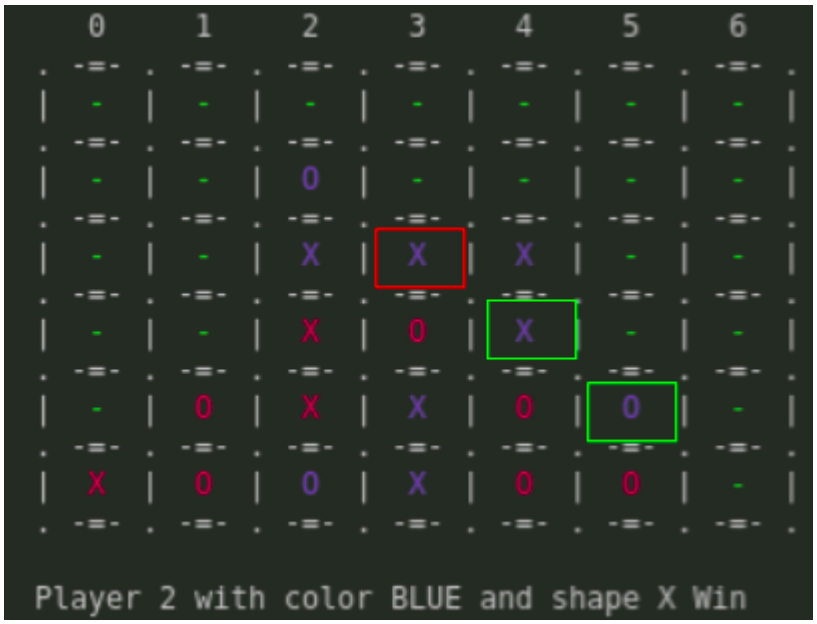
iii. Manusia *First Move* Bot *Minimax* Pendekatan Kedua

Dari hasil uji coba manusia *first move* melawan bot minimax pendekatan kedua, didapatkan bahwa persentase kemenangan adalah 20:80, dimana manusia dengan *first move* memiliki persentase kemenangan 20%.

Pada pendekatan kedua, ada modifikasi yang dilakukan, yaitu dengan menambahkan fungsi heuristic untuk mengevaluasi cabang yang akan diekspansi. Fungsi heuristic yang digunakan sama seperti yang kami implementasikan pada Local Search. Hal ini berarti kinerja dan perilaku bot pada awal-awal permainan akan mirip seperti bot Local Search. Karena pada saat awal permainan, ketinggian pohon dan jumlah node atau state yang akan diekspansi berjumlah banyak. Seiring berjalannya waktu dalam permainan, turn demi turn dilalui, jumlah node atau state yang akan diekspansi berkurang secara eksponensial. Pada permainan jumlah node daun yang ada pada pohon mengikuti 14^n dimana n adalah jumlah koordinat yang tidak memiliki bidak atau kosong pada papan permainan. Seiring berjalannya turn pada permainan, *coverage* dari

node yang diekspansi pada pohon semakin tinggi, dikarenakan ketinggian yang berkurang serta jumlah node daun yang berkurang. Hal ini mengakibatkan bot akan semakin optimal karena pencarian menjadi lebih lengkap.

Berikut adalah contoh hasil akhir permainan dimana bot berhasil memenangkan permainan.



Manusia First Move Bot Minimax Pendekatan Kedua

Dari contoh kasus di atas, dapat dilihat bahwa bot pada peletakan bidak yang ada di kotak hijau, bot sudah dapat mencari pohon pada *coverage* yang cukup tinggi sehingga langkah yang dipilih mendekati optimal. Peletakan bidak pada kedua kotak hijau dilakukan untuk mempersiapkan kemenangan dengan kondisi menang seperti itu dengan tujuan juga dapat melakukan *block* pada kemenangan musuhnya.

iv. Manusia *Second Move* Bot *Minimax* Pendekatan Kedua

Dari hasil uji coba manusia *first move* melawan bot minimax pendekatan kedua, didapatkan bahwa persentase kemenangan adalah 20:80, dimana manusia dengan *second move* memiliki persentase kemenangan 20%. Kemenangan bot tidak berbeda dengan percobaan untuk kondisi permainan dimana manusia *first*

move. Hal ini dikarenakan algoritma minimax akan selalu menganggap lawan adalah lawan yang optimal. Hal ini juga dikarenakan pada permainan *simplexity* sendiri tidak ada keunggulan atau kekurangan berdasarkan urutan jalan pemain (pemain *first move* atau *second move*).

Berikut adalah contoh hasil akhir permainan dimana bot berhasil memenangkan permainan

Manusia *Second Move* Bot *Minimax* Pendekatan Kedua

4. Pembagian Tugas

Tabel Pembagian Tugas

NIM	Tugas
13519087	Pembuatan laporan dan implementasi heuristik untuk *local search* dan *Minimax,* analisis *local search* vs manusia.
13519108	Pembuatan laporan dan implementasi algoritma *Minimax* dengan *Alpha Beta Pruning*, analisis manusia vs *bot Minimax* pendekatan kedua
13519132	Pembuatan laporan dan implementasi algoritma *Minimax* dengan *Alpha Beta Pruning*, analisis manusia vs *bot Minimax* pendekatan pertama
13519185	Pembuatan laporan dan implementasi algoritma *Local Search* (*Hill Climbing* dan *Simulated Annealing*), analisis *bot vs bot* dan *local search* vs manusia.