Pemodelan AI dalam Permainan *Snake* dengan Algoritma *Branch and Bound*

Indra Soaloon Situmorang
Nim : 13505085

Sekolah Teknik Elektro dan Informatika
Jalan Ganesha 10 Bandung
e-mail : if15085@students.if.itb.ac.id

ABSTRAK

Makalah ini akan membahas implementasi algoritma *branch and bound* dalam pemodelan AI dalam permainan *snake*. Model AI yang dimaksudkan adalah model AI agar si ular dalam permainan *snake* tersebut tidak menabrak dinding atau dirinya sendiri saat mengambil apel.

Kata kunci : *snake*, *artificial intelligence*, algoritma *branch and bound.*

1. PENDAHULUAN

Permainan *snake* merupakan permainan populer dalam telepon selular beberapa tahun yang lalu. Inti dari permainan ini adalah agar *snake* yang kita kontrol mendapatkan sebanyak-banyaknya apel tanpa membentur dinding atau bagian tubuhnya sendiri. Semakin banyak apel yang *snake* dapatkan, tubuhnya akan tumbuh sehingga semakin panjang.

Sejatinya permainan ini merupakan *single player game* atau permainan yang dimainkan sendiri. Dalam permainan ini komputer hanya memunculkan apel secara random di layar untuk dimakan. Kesuksesan permainan ini bergantung kepada kecepatan dan perhitungan sang pemain agar ular yang dikontrolnya tidak terjebak dinding atau bagian tubuhnya sendiri.

Kelakuan seperti inilah yang ingin dicoba diwujudkan oleh penulis. Untuk memodelkan kelakuan seperti ini terdapat beberapa algoritma yang dapat dipilih seperti algoritma *brute force*, algoritma *greedy*, program dinamis, ataupun algoritma DFS dan BFS. Contohnya dengan algoritma *greedy* si ular akan bergerak dengan mencari jarak terpendek dirinya dengan apel.

Dari beberapa algoritma di atas algoritma yang dipilih untuk memodelkan kelakuan ular ini adalah algoritma *branch and bound* yang merupakan modifikasi / optimasi dari algoritma BFS. Algoritma ini dirasa cocok karena algoritma ini mendukung perhitungan untuk mencari jalan terpendek menuju apel sementara menghindari bertubrukan dengan dinding atau bagian tubuhnya atau membuat dirinya sendiri terkurung. Hal ini tidak didapat dari algoritma lain seperti DFS dan *brute force.*

2. METODE

Algoritma yang dipakai dalam pemodelan ini adalah algoritma *branch and bound*. Algoritma ini sejatinya adalah algoritma pencarian solusi dengan pencarian melebar atau *breadth first search* (BFS). Dalam algoritma BFS solusi dicari dengan membentuk pohon ruang status yang merupakan pohon dinamis.

Simpul-simpul di dalam pohon diinamis yang memenuhi kendala menyatakan status persoalan. Suatu operator mentransformasikan persoalan dari sebuah status ke status yang lani. Solusi persoalan dinyatakan dengan satu atau lebih status yang disebut status solusi. Status solusi yang merupakan simpul daun disebut status tujuan. Himpunan semua status solusi disebut ruang solusi. Seluruh simpul di dalam pohon dinamis disebut ruang status. Dan pohonnya dinamakan juga pohon ruang status. Akar pada pohon ruang status menyatakan status awal sedangkan daun menyatakan status solusi.

BFS mencari solusi persoalan pada pohon ruang status yang dibentuk secara dinamis dengan cara semua simpul pada aras *d* dibangkitkan terlebih dahulu sebelum simpul-simpul pada aras *d*+1. Simpul BFS memerlukan sebuah antrian untuk menyimpan simpul-simpul yang akan dibangkitkan. Simpul-simpul yang dibangkitkan disimpan di belakang antrian.

2.1 Algoritma *Branch and Bound*

Pada algoritma Branch and Bound, pencarian ke simpul solusi dapat dipercepat dengan memilih simpul hidup berdasarkan nilai ongkos (*cost*). Setiap simpul hidup diasosiasikan dengan sebuah ongkos yang menyatakan nilai batas (*bound*). Batas ini dapat berupa batas bawah (*lower bound*) ataupun batas atas (*upper bound*) masing-masing menyatakan batas maksimum atau batas minimum yang diperlukan untuk membangkitkan sebuah simpul.

Pemberian nilai batas akan ideal jika kita mengetahui letak simpul solusi, namun untuk kebanyakan persoalan letak simpul solusi tidak diketahui. Sehingga nilai batas untuk setiap simpulnya umumnya hanya berupa taksiran ataupun perkiraan.

Untuk membantu pemberian taksiran nilai umumnya digunakan fungsi heuristik yang dapat berupa fungsi apapun asalkan fungsi tersebut mampu

memodelkan ongkos (*cost*) untuk membantu mencapai simpul solusi. Fungsi heuristik untuk menghitung taksiran nilai tersebut dinyatakan secara umum sebagai:

$$c(i) = f(i) + g(i)$$

dengan
c(*i*) = ongkos untuk simpul *i*
f(*i*) = ongkos mencapai simpul *i* dari akar
g(*i*) = ongkos mencapai simpul tujuan dari simpul (*i*)

Nilai c digunakan untuk mengurutkan pencarian. Jika pencarian adalah pencarian dengan biaya terkecil maka simpul berikutnya yang dipilih untuk diekspansi adalah simpul yang memiliki c minimum. Bila sebaliknya, simpul yang diekspansi adalah simpul yang memiliki c maksimum.

Skema umum algoritma *branch and bound* adalah sebagai berikut :

1. Masukkan simpul akar ke dalam antrian Q. Jika simpul akar adalah simpul solusi maka solusi telah ditemukan. Stop.
2. Jika Q kosong, tidak ada solusi. Stop.
3. Jika Q tidak kosong, pilih dari antrian Q simpul *i* yang mempunyai c(*i*) paling kecil (jika *least cost search*) atau c(*i*) paling besar (jika *most cost search*)
4. Jika simpul *i* adalah simpul solusi, berarti solusi sudah ditemukan, stop. Jika simpul *i* bukan simpul solusi, maka bangkitkan semua anak-anaknya. Jika *i* tidak mempunyai anak, kembali ke langkah 2.
5. Untuk setiap anak *j* dari simpul *i*, hitung c(*j*) dan masukkan semua anak-anak tersebut ke dalam antrian Q.
6. Kembali ke langkah 2.

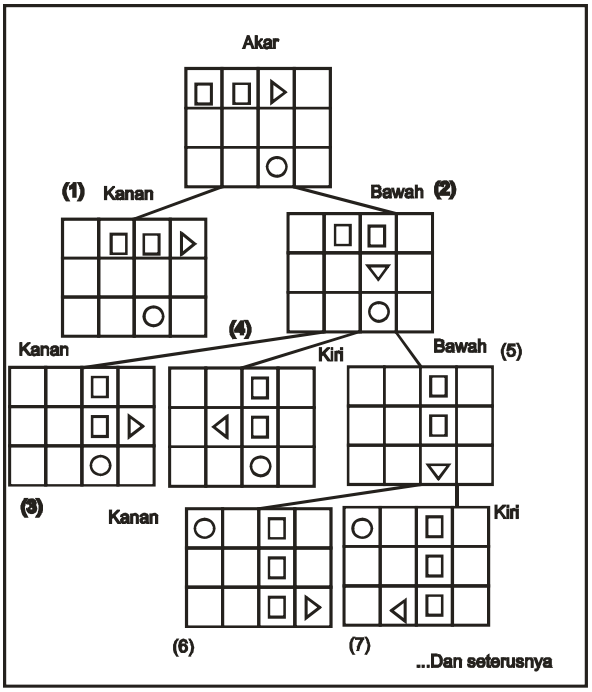
2.1 Modifikasi *Branch and Bound*

Untuk memodelkan perilaku *snake* ini diperlukan beberapa modifikasi karena algoritma*branch and bound* murni belum memadai untuk memodelkan perilaku *snake*.

Modifikasi pertama adalah tujuan (*goal*) dari algoritma tersebut. Pada *branch and bound* murni, algoritma mencari simpul tujuan dengan sesedikit mungkin membangkitkan simpul anak. Sedangkan tujuan dari permainan ini adalah kita bertahan selama mungkin dalam permainan *snake* ini sehingga tujuan utama dalam algoritma ini adalah membuat simpul sedalam mungkin.

Namun simpul yang dibangkitkan harus menjamin bahwa *snake* tidak mati. Simpul yang dibangkitkan pula harus menjamin bahwa *snake* berhasil mendapatkan apel dengan jumlah langkah yang sesedikit mungkin.

Pembangkitan simpul ini dapat lebih jelas dilihat pada gambar berikut :



Gambar 1. Pembangkitan simpul yang diinginkan dalam permainan *snake*

Dalam gambar ini diperlihatkan pembangkitan simpul yang diinginkan dalam permainan *snake*. Simbol kotak melambangkan bagian tubuh dari *snake*. Simbol segitiga melambangkan kepala dan arah gerak *snake*. Simbol lingkaran melambangkan apel yang harus diambil *snake*.

Pada akar terlihat kondisi awal saat permainan pertama dimulai. Simpul yang bisa dibangkitkan adalah simpul dengan gerak ular selanjutnya adalah ke kanan atau ke bawah karena simpul dengan gerak ular ke atas atau ke kiri dapat mengakibatkan ular mati.

Pada aras kedua simpul yang ingin diteruskan adalah simpul dengan gerak ular ke bawah karena apel berada di bawah sehingga simpul tersebut merupakan awal dari simpul dengan ongkos termurah untuk mencapai apel. Kemudian pada aras terakhir dengan model AI ini ular akan bergerak ke kanan dan seterusnya sehingga ular tidak bisa bergerak lagi.

2.2 Fungsi Heuristik yang Digunakan

Untuk memodelkan perilaku yang diinginkan seperti di atas diperlukan fungsi pembangkitan ongkos yang sesuai. Fungsi pembangkitan ongkos ini merupakan kunci dari algoritma *branch and bound* karena algoritma

ini akan membangkitkan simpul sesuai dengan ongkos simpul tersebut.

Dalam hal ini fungsi heuristik yang digunakan adalah :

$$\mathbf{c}(i) = \mathbf{f}(i) + \mathbf{g}(i) + \mathbf{h}(i)$$

dengan

c(*i*) = ongkos untuk simpul ***i***

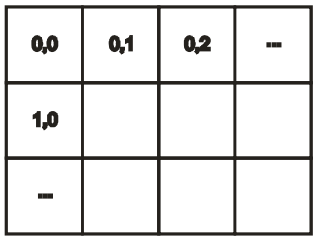
f(*i*) = jarak dari kotak yang dituju dengan simpul *i* ke apel (jarak dihitung dengan rumus jarak dua titik dalam bidang koordinat kartesius)

g(*i*) = ongkos jika simpul ***i*** mengarah ke dinding atau ke bagian tubuh dari ular, bernilai sangat besar karena kita ingin menghindari tabrakan

h(*i*) = adalah ongkos yang dibangkitkan jika ular masuk ke wilayah yang dibuatnya.

Simpul yang dibangkitkan adalah simpul dengan **c(*i*)** terkecil.

Dalam permainan ular ini setiap kotak mempunyai koordinat tertentu dengan ketentuan sebagai berikut :



Gambar 2. Aturan koordinat bidang permainan *snake*

Sehingga pada aras kedua di gambar 1 nilai f(*i*) untuk bergerak ke kanan adalah

$$\mathbf{f(1)} = \sqrt{(2-0)^2 + (2-3)^2} = \sqrt{5}$$

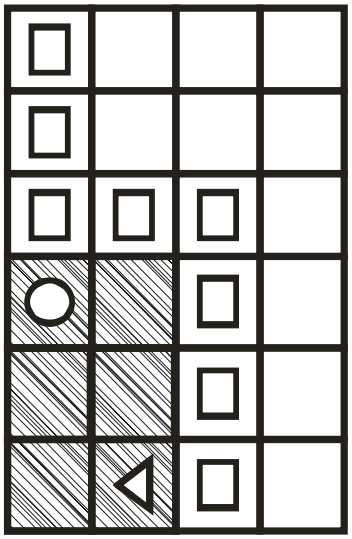
namun karena nilai akar cenderung mempersulit perhitungan maka ongkos untuk **f(2)** dapat dikuadratkan sehingga menjadi

f(1) = 5

Untuk nilai **g(1)** adalah 0 karena simpul **g(1)** tidak menabrak dinding. Namun, tinjau apabila simpul yang bergerak ke atas dicoba dibuat dari simpul akar maka nilai **g** untuk simpul tersebut adalah 99999 nilai ini sangat besar karena kita tidak ingin ular bergerak menabrak dinding. Nilai **g** untuk simpul yang akan menabrak tubuh ular juga akan bernilai 99999.

Untuk nilai **h** tinjau simpul **7** pada simpul tersebut ular sudah membuat wilayah yaitu segi empat dari koordinat (0,0) sampai koordinat (2,1). Nilai **h** akan dibangkitkan berdasarkan wilayah yang sudah dibuat tersebut. Pembangkitan nilai **h** ini harus dilakukan dengan hati-hati karena kita tidak ingin ular terkurung dalam wilayah yang dibuatnya sendiri. Jika ular sudah terkurung maka ular akan mati dan hal ini kita hindari.

Untuk definisi lengkap pembangkitan nilai **h** tinjau kasus berikut :



Gambar 3. Contoh kasus pembangkitan nilai h

Pada kasus ini ular sudah membuat wilayah yaitu kotak-kotak yang diarsir. Jumlah wilayah yang dimiliki adalah **9** karena perhitungan wilayah berdasarkan kuadrat panjang atau lebar tertinggi. Hal ini agar gerakan ular selalu mengeliminasi wilayah dengan gerakan spiral. Untuk gerakan selanjutnya, nilai **h** harus dihitung. Nilai **h** dihitung berdasarkan perhitungan berikut:

$$\mathbf{h}(i) = \mathbf{l} - \mathbf{j}(i) + \mathbf{k}(i)$$

dengan

j(*i*) = jarak dari kotak yang dicapai oleh simpul ***i*** dengan ekor ular

k(*i*) = wilayah yang dibuat jika ular bergerak ke simpul ***i***

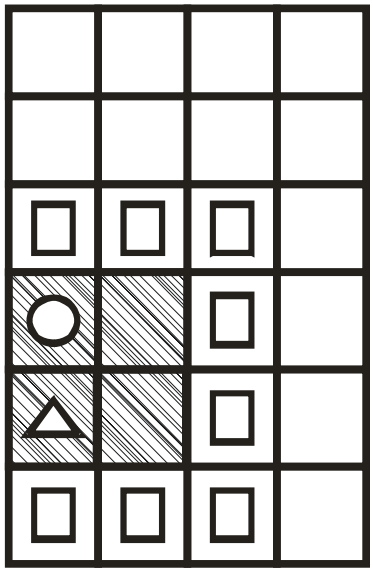
l = panjang badan ular

Pada kasus seperti gambar tiga simpul yang bisa dibangkitkan adalah simpul dengan arah gerak ke kiri atau ke atas mari kita tinjau *cost* untuk kedua simpul tersebut.

f(atas) = 2
g(atas) = 0
h(atas) = 9-16 + 9
c(atas) b f(atas) + **g(atas)** + **h(atas)** = 4

f(kiri) = 2
g(kiri) = 0
h(kiri) = 9-16 + 4
c(kiri) = **f(kiri)** + **g(kiri)**+**h(kiri)** = 1

Dari perhitungan tersebut ular akan bergerak ke kiri. Dari sana simpul yang bisa dibangkitkan hanyalah simpul dengan gerakan ular ke atas, maka ular akan ke atas. Sehingga posisi ular akan seperti :



Gambar 4. Contoh kasus pembangkitan nilai h

Dari posisi ini ular bisa bergerak ke atas atau ke kanan. Namun bila ular bergerak ke atas maka ular akan mati karena ular akan bertambah panjang bila memakan apel dan akan terbentur oleh tubuhnya sendiri. Dalam dua kasus ini tinjau perhitungan berikut:

f(atas) = 0
g(atas) = 0
h(atas) = 9-1 + 4
c(atas) = f(atas) + g(atas) + h(atas) = 12

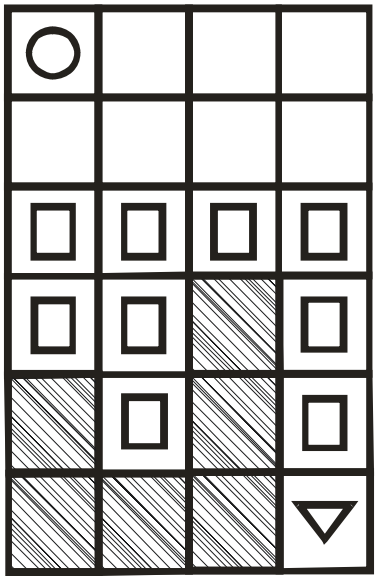
f(kanan) = 2
g(kanan) = 0
h(kanan) = 9 – 5 + 4
c(kanan) = f(kanan) + g(kanan) + h(kanan) = 10

Dengan perhitungan seperti ini, ular akan bergerak ke arah kanan dan ular akan selamat dari jebakan. Selanjutnya ular akan bergerak ke atas kemudian ke kanan.

Fungsi heuristik ini cukup efektif untuk menghindari kejadian seperti ini. Namun tidak selamanya fungsi **h** dihitung karena fungsi **h** akan dihitung jika dan hanya jika ular sudah mempunyai wilayah. Ular memiliki wilayah jika dan hanya jika dia sudah "mengurung" sejumlah kotak atau dengan kata lain bagian tubuhnya sudah menempel dengan dua dinding yang tidak berseberangan. Untuk kasus belum memiliki wilayah fungsi **f** dan **g** sudah cukup mewakili.

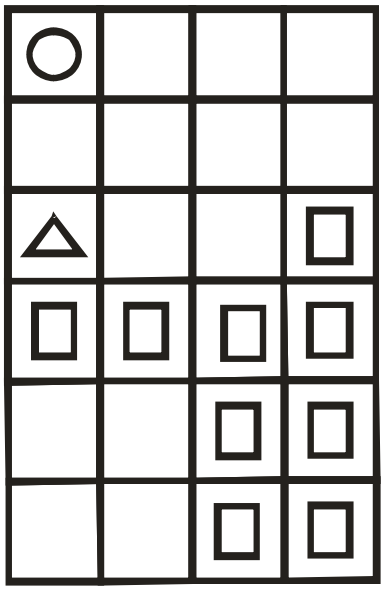
3. PENGETESAN

Pengetesan lebih lanjut dilakukan dengan kasus berikut :



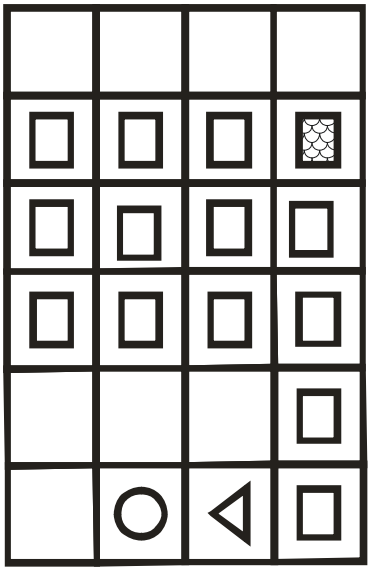
Gambar 5. Contoh kasus untuk pengetesan

Dengan algoritma ini langkah-langkah ular seperti berikut : Langkah pertama ular akan bergerak ke kiri karena itu adalah satu-satunya langkah yang mungkin. langkah berikutnya adalah ke atas, lalu ke atas lagi, kemudian ke kiri, lalu ke kiri, kemudian ke atas sehingga status ular tersebut akan seperti :



Gambar 6. Hasil pengetesan

Namun untuk panjang ular yang sangat besar, algoritma ini gagal memberikan hasil yang memuaskan karena langkah-langkah awal sebelum ular memiliki wilayah mengakibatkan ular terkurung di wilayahnya sehingga apapun kombinasi pergerakan tidak dapat membuat ular dapat keluar dari wilayah tersebut. Hal itu ditemukan dalam kasus berikut :



Gambar 7. Contoh kasus gagal

Algoritma ini tidak bisa mencegah agar ular tidak mengalami keadaan seperti ini karena panjang ular yang besar seringkali mengganggu perhitungan dan dalam kasus belum memiliki wilayah algoritma ini hanya mencari jarak terdekat ke apel sehingga gerakan pada awal-awal terkesan "ceroboh" karena tidak memikirkan akibat dari gerakan tersebut.

IV. KESIMPULAN

Algoritma *branch and bound* merupakan pengembangan dari algoritma BFS yang membangun pohon dinamis dengan memperhitungkan ongkos untuk membangkitkan setiap simpulnya.

Algoritma *branch and bound* cukup berhasil untuk memodelkan kelakuan ular dalam permainan *snake*. Hal ini disebabkan pencarian rute dengan berbagai perhitungan merupakan karakteristik dari algoritma *branch and bound*. Sehingga implementasi kelakuan ini dengan algoritma *branch and bound* lebih "natural" karena karakteristik yang sama.

Implementasi heuristik seperti yang telah dibahas cukup berhasil dalam kasus-kasus di mana panjang ular belum terlalu panjang dan dalam kasus ular sudah memiliki wilayah. Tetapi, implementasi ini belum matang dalam memperhitungkan pergerakan ular saat belum mempunyai wilayah sehingga kasus seperti pada gambar 7 dapat terjadi yaitu ular terkurung di wilayahnya dan tidak dapat keluar dengan kombinasi gerakan apapun.

Untuk memperbaiki hal itu, algoritma ini bisa dikembangkan lebih lanjut dengan memodelkan fungsi pembangkitan ongkos (*cost*) yang lebih kompleks dan lebih memperhitungkan akibat selanjutnya dari setiap langkah. Fungsi seperti itu memiliki kemungkinan yang lebih besar untuk memperbaiki algoritma ini. Mungkin algoritma dengan program dinamis bisa lebih berhasil memodelkan kelakuan AI ini.

REFERENSI

[1] Rinaldi Munir, "Strategi Algoritmik", 2005

[2] Robert Sedgewick, "Algorithm", Addison-Wesley, 1984.