Penerapan dan Implementasi Algoritma *Backtracking*

Dhamma Nibbana Putra[1], Ozzi Oriza Sardjito[2], Christopher Lawrence[3]

Departemen Teknik Informatika, Fakultas Teknologi Industri, Institut Teknologi Bandung (ITB)
Kampus ITB Jl. Ganesha No. 10 Bandung
if13040@students.if.itb.ac.id[1], if13050@students.if.itb.ac.id[2], if13067@students.if.itb.ac.id[3]

Abstrak

Algoritma *backtracking* (runut balik) pada dasarnya mencari segala kemungkinan solusi seperti halnya *brute-force* dan *exhaustive search*. Yang membedakannya adalah pada *backtracking* semua kemungkinan solusi dibuat dalam bentuk pohon terlebih dahulu baru kemudian pohon tersebut dijelajahi (*explore*) secara *DFS* (*Depth Field Search)*. Secara umum algoritma ini berfungsi dengan baik untuk memecahkan masalah-masalah yang berkembang secara dinamik (*dynamic problem solving)* sehingga menjadi dasar algoritma untuk *Artificial Intelligence* (intelejensia buatan). Makalah ini akan membahas kegunaan algoritma *backtracking* dan bagaimana mengimplementasikannya dalam bahasa pemograman secara umum.

Kata kunci: *backtracking, algoritma, kegunaan backtracking, implementasi backtracking, artificial intelligence, dynamic problem solving*

1. Pendahuluan

Artificial intelligence (*AI)* diperlukan untuk membuat sebuah sistim yang dapat bekerja secara otomatis dengan campur tangan dari operator seminim mungkin. Maka dari itu riset untuk membuat suatu AI yang semakin mendekati "sempurna" gencar dilakukan. Salah satu terobosan dalam perkembangan AI pada akhir '90-an adalah dikembangkannya *supercomputer* Deep Blue dan Deep Blue II oleh IBM yang dikembangkan untuk bermain catur. Pada program-program untuk pemecahan masalah yang berkembang secara dinamik (meskipun juga dapat digunakan untuk masalah statik) seperti itu, algoritma yang paling umum digunakan adalah algoritma *backtracking*.

2. Konsep Dasar

2.1 *Dynamic problem solving*

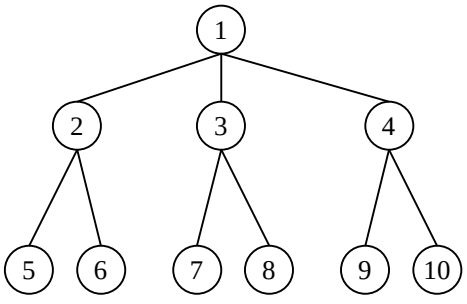
Dynamic problem solving pada dasarnya adalah memecahkan masalah yang keadaannya (*state*) selalu berubah dari-waktu ke waktu. Perubahan ini bisa mengikuti suatu pola tertentu yang dapat didefinisikan sebagai suatu fungsi terhadap waktu, atau juga tidak berpola seperti pada *board games* yang langkah selanjutnya sangat tergantung pada langkah yang lawan jalankan.

Prinsip dasarnya adalah menganalisa segala jalan menuju suatu solusi yang ada kemudian memilih salah satu jalan yang akan mengarah ke solusi terbaik. Pada kebanyakan kasus, program tidak hanya akan "melihat" beberapa langkah ke depan sebagai pertimbangan dalam menentukan pilihan solusi seperti pada permainan catur (dan sebagian besar *board games* lainnya) karena jumlah seluruh solusi sangat banyak sehingga program tidak akan berjalan dengan efisien lagi jika seluruh kemungkinan ditelusuri.

2.2 Algoritma *Backtracking*

Algoritma *backtracking* mempunyai prinsip dasar yang sama seperti *brute-force* yaitu mencoba segala kemungkinan solusi. Perbedaan utamanya adalah pada ide dasarnya, semua solusi dibuat dalam bentuk pohon solusi (pohon ini tentunya berbentuk abstrak) dan algoritma akan menelusuri pohon tersebut secara *DFS* (*depth field search)* sampai ditemukan solusi yang layak.

Nama *backtrack* didapatkan dari sifat algoritma ini yang memanfaat karakteristik himpunan solusinya yang sudah disusun menjadi suatu pohon solusi. Agar lebih jelas bisa dilihat pada pohon solusi berikut:



Misalkan pohon diatas menggambarkan solusi dari suatu permasalahan. Untuk mecapai solusi (5), maka jalan yang ditempuh adalah (1,2,5), demikian juga dengan solusi-solusi yang lain. Algoritma *backtrack* akan memeriksa mulai dari solusi yang pertama yaitu solusi (5). Jika ternyata solusi (5) bukan solusi yang layak maka algoritma akan melanjutkan ke

solusi (6). Jalan yang ditempuh ke solusi (5) adalah (1,2,5) dan jalan untuk ke solusi (6) adalah (1,2,6). Kedua solusi ini memiliki jalan awal yang sama yaitu (1,2). Jadi daripada memeriksa ulang dari (1) kemudian (2) maka hasil (1,2) disimpan dan langsung memeriksa solusi (6). Pada pohon yang lebih rumit, cara ini akan jauh lebih efisien daripada *brute-force*.

Pada beberapa kasus, hasil perhitungan sebelumnya harus disimpan, sedangkan pada kasus yang lainnya tidak perlu.

3. Kegunaan *Backtrack*

Penggunaan terbesar *backtrack* adalah untuk membuat AI pada *board games*. Dengan algoritma ini program dapat menghasilkan pohon sampai dengan kedalaman tertentu dari *current status* dan memilih solusi yang akan membuat langkah-langkah yang dapat dilakukan oleh *user* akan menghasilkan pohon solusi baru dengan jumlah pilihan langkah terbanyak. Cara ini dipakai sebagai AI yang digunakan untuk *dynamic problem solving*.

Beberapa kegunaan yang cukup terkenal dari algoritma *backtrack* dari suatu masalah "statik" adalah untuk memecahkan masalah *N-Queen problem* dan *maze solver*.

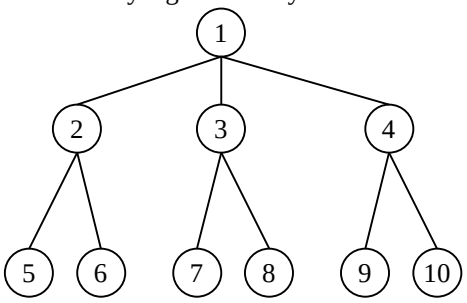
N-Queen problem adalah bagaimana cara meletakkan bidak *Queen* catur sebanyak N buah pada papan catur atau pada papan ukuran NxN sedemikian rupa sehingga tidak ada satu bidakpun yang dapat memangsa bidak lainnya dengan 1 gerakan[1]. Meskipun mungkin terdapat lebih dari satu solusi untuk masalah ini, tetapi pencarian semua solusi biasanya tidak terlalu diperlukan, tetapi untuk beberapa kasus tertentu diperlukan pencarian semua solusi sehingga didapatkan solusi yang optimal.

Maze solver adalah bagaimana cara mencari jalan keluar dari suatu *maze* (labirin) yang diberikan. Pada *maze* yang sederhana dimana *field* yang dibentuk dapat direpresentasikan dalam bentuk biner dan pada setiap petak maksimal terdapat 4 kemungkinan: atas, kanan, bawah, dan kiri. Untuk masalah ini biasanya solusi pertama yang ditemukan bukanlah solusi yang paling optimal sehingga untuk mendapatkan hasil yang optimal dibutuhkan pencarian terhadap seluruh kemungkinan solusi. Hal ini disebabkan oleh urutan pencarian yang telah ditetapkan dalam program (apakah menyelidiki kemungkinan ke arah atas dahulu atau ke arah lainnya dahulu).

4. Implementasi Algoritma *Backtracking*

Karena algoritma *backtrack* akan mencoba menelusuri semua kemungkinan solusi yang mungkin, maka hal pertama yang harus dilakukan adalah membuat algoritma dasar yang akan menjelajahi semua kemungkinan solusi. Kemudian algoritma ini diperbaiki sehingga cara pencarian solusinya dibuat lebih sistematis. Lebih tepatnya jika algoritma itu dibuat sehingga akan menelusuri kemungkinan solusi pada suatu pohon solusi abstrak.

Algoritma ini memperbaiki teknik *brute-force* dengan cara menghentikan penelusuran cabang jika jika pada suatu saat sudah dipastikan tidak akan mengarah ke solusi. Dengan demikian jalan-jalan yang harus ditempuh yajg ada dibawah *node* pda pohon tersebut tidak akan ditelusuri lagi sehingga kompleksitas program akan berkurang. Kembali pada pohon solusi yang sebelumnya:



Jika pada pengecekan status di *node* (2) sudah dapat dipastikan bahwa jalan ini tidak akan menghasilkan solusi yang layak maka penelusuran jalan langsung dibatalkan dan dalam kasus ini langsung menelusuri *node* (3). Pembatalan penelusuran pda *node* (2) secara otomatis akan menghilangkan pengecekan jalan (1,2,5) dan (1,2,6) yang merupakan faktor untuk mengurangi kompleksitas waktu yang diperlukan. Semakin cepat terdeteksi bahwa jalan yang ditempuh tidak akan mengarah ke suatu solusi yang layak maka program akan bekerja dengan lebih efisien.

Untuk kembali pada *state* sebelumnya (dalam kondisi *backtrack*) maka agar hasil perhitungan sampai dengan *state* tersebut harus disimpan dalam memori. Penyimpanan ini paling mudah dilakukan pada bahasa pemograman yang telah bisa menangani fungsi-fungsi atau prosedur-prosedur secara rekursif, sehingga manajemen memori akan dilakukan sepenuhnya oleh *compiler*. Pada bahasa pemograman yang lain, algoritma *backtrack* masih dapat diimplementasikan meskipun manajemen memori harus dilakukan oleh *programmer*. Cara menajemen memori yang baik adalah menggunakan *pointer* atau *dynamic array* karena kedalaman pohon solusi yang harus ditelusuri biasanya bervariasi dan tidak dapat ditentukan.

Algoritma *backtrack* dapat diimplementasikan dengan mudah pada bahasa-bahasa pemograman yang telah men-*support* pemograman rekursif. Bahasa pemograman yang nyaman digunakan adalah Pascal atau Java. Bahasa Pascal dipilih karena bisa diprogram secara rekursif dan mendukung

penggunaan *pointer*. Sedangkan bahasa Java meskipun lebih rumit tetapi dapat bekerja secara rekursif dan sangat mudah dalam membuat *dynamic array*.

Skema yang umum digunakan pada pemograman dengan fungsi rekursif adalah telusuri solusi yang ada kemudian cek *state* program apakah sedang menuju ke suatu solusi. Jika ya maka panggil kembali fungsi itu secara rekursif. Kemudian cek apakah solusi sudah ditemukan (jika hanya perlu mencari sebuah solusi) atau semua kemungkinan solusi sudah diperiksa (jika ingin mengecek semua kemungkinan solusi). Jika ya maka langsung keluar dari prosedur atau fungsi tersebut. Kemudian pada akhir fungsi kembalikan semua perubahan yang dilakukan pada awal fungsi. Pada bahasa pemograman yang telah mendukung pemanggilan secara rekursif semua *state* setiap fungsi akan diatur oleh *compiler*. Dengan skema ini maka jika program tidak memiliki solusi maka *state* akhir program akan sama dengan *state* awal program.

5. Kesimpulan

Algoritma *backtrack* sangat berguna untuk mencari solusi-solusi jumlah kombinasi jalan yang diperlukan untuk mencari solusi tersebut selalu berubah terhadap waktu ataupun respon *user* (dinamik). Dalam penerapannya algoritma *backtrack* juga mudah diimplementasikan dengan bahasa pemograman yang sudah mendukung pemanggilan fungsi atau prosedur secara rekursif. Untuk masalah-masalah yang mempunyai kemungkinan solusi yang kompleks, seperti permainan catur, sebaiknya kedalaman pohon dibatasi sehingga tidak menghabiskan waktu yang sangat lama (tentu saja batas kedalaman pohon ini relatif - bergantung pada kecepatan *cycle clock* prosesor yang tersedia). Secara umum, semakin dalam pohon yang ditelusuri, semakin akurat pula jalan menuju solusi (dalam hal *board games* maka tingkat kesulitan AI semakin bertambah). Untuk alokasi memori yang akan dipakai untuk menyimpan jalan solusi sebaiknya menggunakan *dynamic array* mengingat sebagian besar program yang menggunakan algoritma ini menghasilkan solusi yang tidak dapat diprediksi berhubung dengan sifat program yang akan sangat bergantung pada *state* program yang berubah setiap saat. Saran akhir dalam penggunaan algoritma *backtrack*, sebagaimana dengan algoritma-algoritma yang lain adalah kombinasikan dengan algoritma yang lain. Beberapa metoda dari *exhaustive search* bisa digunakan untuk menentukan apakah solusi yang sedang dijelajahi (*explore*) menuju ke solusi yang diharapkan.

Daftar Pustaka

[1]http://www.informatika.org/~rinaldi/algoritma%20runut-balik.ppt
[2]http://www.htdp.org/2001-01-18/book/node219.htm
[3]http://www.cs.ubc.ca/~silito/papers/thesis/final/node11.html