Algoritma Exhaustive Search sebagai Pencari Solusi Terbaik

Marianti Putri Wulandari[1], Hudzaifah Lutfi[2], Dwi Rahardjo[3]

Laboratorium Ilmu dan Rekayasa Komputasi
Departemen Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung

E-mail : if13093@students.if.itb.ac.id[1], if13111@students.if.itb.ac.id[2],
if13113@students.if.itb.ac.id[3]

Abstrak

Selama ini metode dasar yang digunakan dalam menyelesaikan semua masalah dengan cara yang sangat sederhana adalah algoritma Brute Force. Metode Brute force yang khusus mencari solusi dari objek-objek dengan kriteria tertentu adalah Algoritma Exhaustive Search. Langkah-langkah algoritma Exhaustive Search yaitu mencoba semua kemungkinan solusi yang ada, sehingga solusi terbaik secara pasti akan ditemukan. Namun kelemahan dari Exhaustive Search adalah kompleksitas waktu yang besar sehingga tidak dapat digunakan untuk menyelesaikan masalah dalam jumlah yang besar. Karena itu, perlu adanya teknik heuristik untuk mengatasi hal tersebut.

***Kata kunci** : brute force, exhaustive search, teknik heuristik*

1. Pendahuluan

Pencarian solusi terbaik dari objek-objek dengan kriteria tertentu dengan menggunakan algoritma Exhaustive Search adalah dengan mencari semua kombinasi dan permutasi dari objek-objek yang ada. Semakin banyak objek, semakin banyak juga kemungkinan solusinya. Biasanya kompleksitas waktu dari algoritma Exhaustive Search masih eksponensial, sehingga algoritma ini cenderung untuk dihindari walaupun solusi yang ditemukan adalah solusi yang memang benar-benar terbaik.

Untuk mempercepat pencarian solusi dengan Exhaustive Search dapat digunakan suatu teknik yang disebut teknik heuristik (heuristic). Teknik ini meliputi salah satunya adalah mengeliminasi kemungkinan solusi yang tidak mungkin menjadi solusi terbaik, ataupun mengadopsi metode lain.

Contoh yang akan dijelaskan di sini adalah pengecekan bilangan prima dan Knapsack Problem.

2. Contoh Analisa Kasus

2.1 Pengecekan Bilangan Prima

Algoritma sederhana yang digunakan untuk mengecek apakah suatu bilangan adalah bilangan prima atau bukan adalah sebagai berikut.

```
function Prima (input x : integer) -> Boolean
{Menguji apakah x bilangan prima atau buka}
{Masukan : x bilangan integer positif}
{Keluaran : true jika x bilangan prima, false jika x bukan bilangan prima}

Deklarasi
      y : integer
      test : Boolean

Algoritma
      if (x < 2) then
        test <- false
      else
            if (x = 2) then
              test <- true
        else
              y <- 3
              test <- true
              while (test) and (y <= akar(x)) do
                if (x mod y = 0) then
                      test <- false
                else
                      y <- y + 1
                endif
              endwhile
              {not test or y < 2}
       endif
      endif
return test
endfunction
```

Algoritma tersebut masih sangat sederhana dan sebenarnya masih bisa dibuat lebih efisien. Alasan yang penting yaitu :

1. Bilangan genap selain dua tidak mungkin menjadi bilangan prima. Oleh karena itu,

akan lebih efisien jika sejak awal bilangan masukan dicek terlebih dahulu apakah ganjil atau genap. Jika genap (selain dua), maka fungsi harus mengembalikan nilai false. Jika ganjil, maka fungsi harus mengembalikan nilai true.

2. Bilangan masukan akan lebih efektif jika diuji pembagian hanya dengan bilangan-bilangan prima yang lebih kecil dari akar kuadratnya. Dengan demikian, *pengujian pembagian dengan bilangan-bilangan genap sebenarnya tidak perlu dilakukan*. Oleh karena itu, angka penguji harus mulai dari tiga, dan bilangan ganjil selanjutnya dengan penambahan dua.

Algoritmanya setelah dimodifikasi adalah sebagai berikut.

```
function Prima (input x : integer) -> Boolean
{Menguji apakah x bilangan prima atau buka}
{Masukan : x bilangan integer positif}
{Keluaran : true jika x bilangan prima, false jika x bukan bilangan prima}

Deklarasi
      y : integer
      test : Boolean

Algoritma
      if (x < 2) then
        test <- false
      else
        if (x mod 2 = 0) then
              if (x = 2) then
                test <- true
              else
                test <- false
              endif
else
      y <- 3
              test <- true
              while (test) and (y < akar (x)) do
                if (x mod y = 0) then
                      test <- false
                else
                      y <- y + 2
                endif
                endwhile
                {not test or y < 2}
endif
       endif
return test
endfunction
```

Dengan mengeliminasi kemungkinan-kemungkinan tersebut, algoritma akan lebih efisien dan solusi yang dihasilkan sudah pasti merupakan solusi terbaik. Waktu penyelesaian pada kasus terburuk yang diperlukan oleh algoritma pertama adalah T(n) = $\left|\sqrt{n}\right|$. Setelah diperbaiki, waktu penyelesaian pada kasus terburuk akan lebih cepat dua kali lipat, yaitu T(n) = $\left|\sqrt{n}\right|$ / 2. Pada kasus terbaik yaitu saat bilangan masukan adalah genap selain dua, algoritma hanya akan melakukan satu kali pengujian.

2.2 Persoalan Knapsack

Masalah dari Knapsack Problem adalah :
"Diberikan n buah objek dan sebuah knapsack dengan kapasitas tertentu (K). Setiap objek memilki property bobot w dan keuntungan p. Objektif persoalan adalah bagaimana memilih objek-objek yang dimasukkan ke dalam knapsack sehingga tidak melebihi kapasitas knapsack namun memaksimumkan total keuntungan yang diperoleh."

Masalah tersebut dapat diselesaikan dengan menggunakan algoritma Exhaustive Search. Langkah -langkahnya adalah:

1. Misalkan terdapat n objek yang memiliki profit dan berat masing-masing dan kapasitas knapsack sebanyak Maks.
2. Cari himpunan bagian dari n objek
3. Uji jumlah berat dari himpunan bagian tersebut. Jika beratnya melebihi kapasitas, kembali ke nomor 2. Jika tidak, lanjut ke nomer 3.
4. Uji jumlah profit dari himpunan bagian tersebut. Jika profitnya melebihi profit maksimal sementara, simpan profit baru sebagai profit maksimal dan himpunan bagiannya. Jika tidak, kembali ke nomor 2.
5. Jika seluruh himpunan bagian sudah diuji, himpunan bagian dengan profit maksimal terakhir adalah solusi terbaik dari knapsack

Algoritma tersebut membutuhkan waktu berorde 2^n untuk mencari himpunan bagian, dan n untuk masing-masing pengujian berat dan profit. Karena itu, algoritma ini memiliki waktu penyelesaian berorde $n2^n$. Walaupun akan menghasilkan solusi yang benar-benar terbaik, tetapi algoritma ini tidak mungkin digunakan jika objek berjumlah besar.
Sebenarnya dalam kasus ini, algoritma Exhaustive Search akan menjadi sangat efisien jika dipadu dengan algoritma runut-balik atau backtracking. Algoritma Runut-Balik (backtracking) adalah algoritma yang berbasis pada Depth First Search yang hanya melakukan pencarian ke arah solusi saja yang dipertimbangkan. Dengan metode ini, kemungkinan solusi yang tidak layak akan dieliminasi secara efektif sehingga waktu yang diperlukan untuk menyelesaikan masalah akan berkurang.

Berikut ini adalah penyelesaian masalah knapsack dengan menggunakan algoritma runut-balik.

Ketentuan :

1. Simpul melambangkan status setiap elemen. Elemen berstatus 1 jika elemen tersebut dimasukan ke dalam knapsack. Elemen berstatus 0 jika elemen tersebut tidak dimasukan. Contoh {1, 0, 1, 0, 0} artinya adalah hanya elemen ke 1 dan 3 dimasukan ke knapsack.
2. Pohon yang dibentuk adalah pohon biner. Di tiap simpul dengan kedalaman h terjadi pengujian terhadap elemen ke-h. Cabang ke kanan melambangkan simpul orang tua dimasukan ke knapsack, dan sebaliknya. Dengan demikian, status di setiap simpul selalu berubah.
3. Simpul mati artinya adalah jumlah berat sudah melebihi kapasitas.

Langkah-langkah :

1. Status awal adalah {0,0,..,0}. Jumlah objek adalah n.
2. Cek elemen pertama.
3. Jika layak dimasukkan, status elemen uji berubah menjadi 1. Jika tidak, status tetap.
4. Jika simpul hidup, cek di kedalaman selanjutnya.
5. Jika simpul mati, cek di simpul sebelumnya yang masih hidup.
6. Jika sudah mencapai kedalaman n (semua objek sudah diuji), status di simpul tersebut adalah solusinya.

Algoritma tersebut sangat efisien karena simpul-simpul mati tidak akan dilanjutkan lagi, sehingga dapat mengurangi jumlah operasi. Namun kelemahan dari algoritma ini adalah bahwa solusi pertama yang dihasilkan adalah solusi yang pertama kali ditemukan sehingga belum tentu solusi terbaik.

Hal ini dapat diatasi dengan menguji seluruh kemungkinan solusi tanpa mempertimbangkan perkembangan dari simpul mati. Hal ini merupakan algoritma Exhaustive Search yang dipadu dengan algoritma runut balik. Dengan ketentuan yang sama dengan algoritma sebelumnya, langkah-langkah dari algoritma setelah dimodifikasi :

1. Status awal adalah {0,0,..,0}. Jumlah objek adalah n. Jika jumlah berat semua objek lebih kecil atau sama dengan kapasitas Knapsack, maka solusi terbaik adalah {1, 1, 1,...., 1}. Jika tidak, lanjut ke nomor 2.
2. Cek elemen pertama.
3. Jika layak dimasukkan, status elemen uji berubah menjadi 1. Jika tidak, status tetap.
4. Jika simpul hidup, cek di kedalaman selanjutnya.
5. Jika simpul mati, cek di simpul sebelumnya yang masih hidup.
6. Jika sudah mencapai kedalaman n (semua objek sudah diuji), uji profit di simpul tersebut. Jika profit di simpul tersebut lebih besar dari profit sementara, jadikan profit di simpul tersebut sebagai profit sementara. Lanjutkan dengan mengecek simpul sebelumnya yang masih hidup.
7. Jika semua simpul sudah diuji, profit sementara terakhir adalah solusi terbaik dari knapsack.

Solusi yang dihasilkan sudah pasti solusi yang terbaik karena semua kemungkinan sudah diuji. Waktu penyelesaian algorima tersebut pada kasus terburuk adalah berorde 2^n. Walaupun masih berorde eksponensial, namun pada kasus rata-rata, jumlah operasi akan berkurang karena adanya simpul mati, sehingga waktu penyelesaian akan lebih cepat.

Algoritma tersebut masih mungkin lebih diefisienkan lagi dengan trik-trik tertentu seperti pengurutan objek-objek terlebih dahulu berdasarkan berat dan lain-lain. Dengan begitu, waktu penyelesaian akan lebih cepat dan solusi terbaik akan didapat.

3. Kesimpulan

Algoritma exhaustive search adalah algoritma yang terbaik dalam hal mencari solusi terbaik dengan sifat tertentu. Waktu penyelesaiannya yang lama sebenarnya dapat dipersingkat dengan menggunakan teknik heuristik, contohnya dengan mengeliminasi kemungkinan solusi yang tidak mungkin menjadi solusi terbaik, ataupun dengan memadukan algoritma tersebut dengan algoritma lain.

Daftar Pustaka

1. Rinaldi Munir, *Diktat Kuliah Strategi Algoritmik*, Departemen Teknik Informatika Institut Teknologi Bandung, 2005.
2. A. Chu and Y. Lin, *Parallelization of Primes in PPP*, 2002.
3. M. Agrawal, N. Kayal, and N. Saxena, *PRIMES is in P*, 2002