Algoritma Branch and Bound pada Permasalahan 0-1 Knapsack

Sandy Socrates (13508044)

Program Studi Teknik Informatika 2008, Institut Teknologi Bandung
Jl. Ganesha 10, 40116 Bandung
e-mail: if18044@students.if.itb.ac.id

ABSTRAK

Knapsack, sebuah permasalahan kombinasi di mana dibutuhkan sebuah optimasi dalam pengambilan barang dengan nilai dan berat, dan dibutuhkan kombinasi pengambilan barang dengan nilai terbesar tanpa melebihi sebuah batas berat tertentu. Permasalahan ini sering mucul dalam alokasi sumber daya dengan keuangan yang terbatas. Permasalahan 0-1 Knapsack merupakan variasi masalah dari Knapsack dimana jumlah barang yang diperbolehkan hanya 0 atau 1. Salah satu cara yang dapat digunakan untuk mendapatkan hasil yang optimal dari Permasalahan 0-1 Knapsack adalah algoritma Branch and Bound yang melibatkan pohon biner untuk memodelkan permasalahan 0-1 Knapsack.

Kata kunci: Knapsack, kombinasi, Branch and Bound, pohon

1. PENDAHULUAN

Dari berbagai masalah yang sering muncul dalam kehidupan kita, optimasi selalu menarik untuk diperbincangkan. Dalam pemrograman, optimasi pun memiliki tempatnya tersendiri. Baik itu menjadi masalah yang harus dipecahkan, maupun keharusan dalam membuat program yang optimal.

Knapsack adalah salah satu masalah yang sering muncul dalam kasus optimasi dan kombinatorial. Knapsack sendiri terdiri dari beberapa varian, yaitu 0-1Knapsack, Knapsack Terbatas, dan Knapsack Tak Terbatas. Ada berbagai macam algoritma yang dapat digunakan untuk memecahkan masing- masing varian, dan yang akan kita bahas pada tulisan ini adalah Algoritma Bound and Branch yang digunakan untuk menjawab permasalahan 0-1 Knapsack. Algoritma Bound and Branch sendiri adalah algoritma yang digunakan untuk menjawab berbagai masalah optimasi, terutama pada optimasi diskrit dan kombinasional.

2. Dasar Teori

Teori dasar yang digunakan untuk membahas masalah:

2.1 Permasalahan Knapsack

Permasalahan knapsack atau rucksack adalah permasalahan optimasi kombinatorial, ilustrasinya adalah : “Diberikan sekumpulan barang, masing-masing dengan berat dan nilai, kita harus dapat menentukan jumlah dari masing-masing barang untuk dimasukan dalam sebuah wadah sehingga total beratnya kurang dari sama dengan berat yang telah ditentukan, dan total nilainya harus sebesar mungkin.” [1].

Permasalahan ini sering muncul dalam alokasi sumber daya dengan dana yang terbatas. Permasalahan yang sama juga muncul dalam kombinatorik, teori kompleksitas, kriptografi, dan matematika terapan.

Pertanyaan yang dapat mencerminkan permasalahan knapsack adalah “dapatkah nilai paling sedikit V dapat dilewati dengan berat W?”

Jika kita memiliki n buah barang, dari 1 sampai n. Masing-masing i barang memiliki nilai p_i dan berat w_i. Dengan asumsi berat tidak ada yang negatif. Dan berat maksimum yang bisa ditahan oleh tas adalah W. Maka kita bisa menggambarkan 3 variasi masalah knapsack sebagai berikut.

2.1.1 Permasalahan Knapsack 0-1

Merupakan permsalahan knapsack yang paling umum. Jumlah tiap barang i maksimum adalah 1. Jadi pada jawaban dari optimasi jumlah barang i tiap nomornya hanya bisa 0 atau 1.

Modelnya :

Maksimum dari $\sum_{i=1}^{n} x_i p_i$ (1)

Dengan $\sum_{i=1}^{n} w_i x_i \leq W , x_i \in \{0,1\}$ (2)

2.1.2 Permasalahan Knapsack Terbatas

Jumlah tiap barang i maksimum adalah b_i.

Modelnya :

Maksimum dari $\sum_{i=1}^{n} x_i p_i$ (3)

Dengan $\sum_{i=1}^{n} w_i x_i \leq W , x_i \in \{0,1, \ldots, b_i\}$ (4)

2.1.3 Permasalahan Knapsack Tak Terbatas

Jumlah tiap barang i tidak dibatasi.

2.2 Algoritma Branch and Bound

Branch and Bround sangat mirip dengan *backtracking* pada pohon untuk memecahkan permasalahan. Perbedaannya adalah metoda branch and bound tidak membatasi cara kita dalam melintasi pohon dan hanya digunakan dalam permasalahan optimasi.

Metode branch and bound menghitung sebuah angka pada sebuah *node* untuk menentukan apakah *node* itu menjanjikan atau tidak. Angka yang didapat merupakan batas harga pada sebuah solusi yang bisa didapatkan dengan memperluas daerah luar *node*. Jika batas yang didapatkan tidak lebih baik dari harga dari solusi terbaik yang ditemukan sebelumnya, maka *node* itu tidak menjanjikan.[2]

Diluar itu, maka *node* menjanjikan. Karena harga optimal adalah minimum pada beberapa masalah dan maksimum pada masalah lainnya. Dengan kata "lebih baik" kita maksudkan lebih besar atau lebih kecil tergantung masalahnya. Seperti pada kasus algoritma *backtracking*, branch and bound pada umumnya bersifat eksponensial, bagaimanapun untuk hal-hal yang besar algoritma ini sangat efisien.

Algoritma *backtracking* untuk memecahkan 0-1 Knapsack sesungguhnya adalah algoritma branch and bound juga. Karena pada algoritma itu, fungsi yang menjanjikan menghasilkan hasil yang salah jika harga dari batas kurang dari nilai maksimum sementara. Bagaimanapun juga sebuah algoritma *backtracking* tidak menggunakan keuntungan sebenarnya dari branch and bound. Selain menggunakan batas untuk menentukan apakah sebuah *node* menjanjikan atau tidak, kita bisa membandingkan batas dari *node* yang menjanjikan dan mengunjungi anak dari *node* dengan batas yang paling baik. Dengan cara ini kita bisa mendapatkan hasil optimal dengan lebih cepat dari pada mengunjungi seluruh *node*. Pendekatan ini disebut *best-first search with branch-and-bound pruning.* Pendekatan ini merupakan modifikasi dari *breadth-first search with branch and bound pruning.*

Breadth-first search pada pohon adalah cara pencarian pada pohon dengan mengunjungi akar, kemudian seluruh anak di tingkat 1, kemudian seluruh anak di tingkat 2, sampai anak di tingkat n. Anak dikunjungi dari kiri ke kanan.

3. Pembahasan

Untuk melihat implementasi algoritma branch and bound pada 0-1 knapsack kita bisa lihat pada *breadth-first search with branch and bound pruning* dan *best-first search with branch and bound pruning*.

3.1 Breadth-First Search with Branch and Bound Pruning

Misalkan diberi persoalan 0-1 Knapsack dengan n=4 dan W==16, serta

Tabel 1 Contoh Permasalahan 0-1 Knapsack

Barang- i	p_i	w_i	p_i / w_i
1	40$	2	20$
2	30$	5	6$
3	50$	10	5$
4	10$	5	2$

Pada contoh di atas, barang telah diurut mengecil sesuai dengan p_i / w_i dengan menggunakan *breadth-first search with branch and bound pruning* cara kita mengerjakan adalah :

-membuat pohon biner yang berisi kemungkinan kombinasi barang pada *node*.

-Menjadikan berat dan keuntungan sebagai berat total dan keuntungan total dari barang yang telah dimasukan dalam *node*.

-Mengecek apakah sebuah node menjanjikan atau tidak dengan melihat bound dan berat total.

Jika *node* ada pada level *i* dan *node* pada level *k* adalah *node* yang membuat berat>W, maka

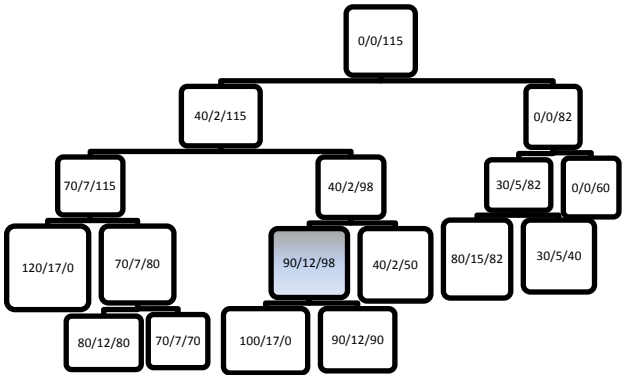
$$berattotal = berat + \sum_{j=1+1}^{k-1} w_j \quad (4)$$

$$batas = (keuntungan + \sum_{j=i+1}^{k-1} p_j) + (W - berattotal) X \frac{p_k}{w_k} \quad (5)$$

Node tidak menjanjikan jika batas kurang dari sama dengan keuntungan maksimal sementara. Dan *node* juga tidak menjanjikan jika

$$berat \geq W \quad (6)$$

Ilustrasi dari pohon yang dihasilkan dapat dilihat pada gambar 1.



Gambar 1. Pohon biner dari proses breadth-first search with branch and bound pruning

Keterangan: a/b/c,
a=keuntungan
b=berat
c=batas

Hasil optimasi diberi warna abu-abu.

Node paling atas adalah akar (0,0), node di bawahnya diberi level i (1 sampai 4). Dari kiri ke kanan diberi nomor j (1 dan seterusnya)u. Jadi node 40/2/115 disebut node (1,1) dan (0/0/82) disebut node (1,2).

Node dengan berat lebih dari W maka batasnya akan diisi 0 dan anak-anaknya tidak perlu diperiksa, perhatikan node (3,1) dan (4,3).

Algoritma breadth-first search with branch and bound pruning [2]

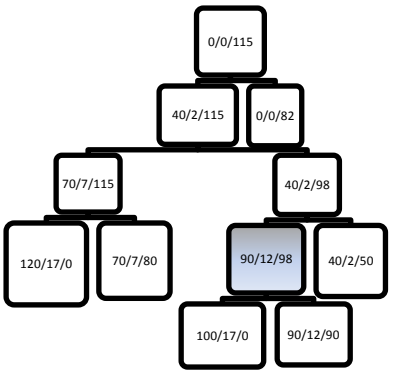
```
procedure
breadth_first_search_with_branch_and_bound
(T:state_space_tree; var best: number);

var
    Q : queue_of_node;
    u,v: node;
begin
    initialize(Q);
    v:=root of T;
    enqueue(Q,v);
    best:= value(v);
    while not empty (Q) do
        dequeue(Q,v)
        for each child u of v do
            if value(u) is better than best then
                best:=value (u)
            end;
            if  bound(u) is better than best then
                enqueue(Q,u)
            end
        end
    end
end;
```

3.2 Best-First Search with Branch and Bound Pruning

Pada umumnya, strategi *breadth-first-search* tidak memiliki keuntungan dari backtracking. Tapi kita bisa meningkatkan pencarian kita dengan menggunakan batas tidak hanya untuk menentukan apakah *node* itu menjanjikan atau tidak. Setelah mengunjungi semua anak dari sebuah *node*, kita bisa melihat *node* mana saja yang menjanjikan untuk dikunjungi. Ingat kembali bahwa *node* yang menjanjikan adalah *node* yang memiliki batas yang lebih baik dari solusi sementara. Dengan cara ini, sering kita dapat menemukan solusi optimal lebih cepat dari pada dengan mengecek satu persatu.

Sebagai contoh, jika kita mengerjakan soal yang sama dengan contoh pada *breadth-first search with branch and bound pruning*. Berattotal dan batas kita hitung dengan cara yang sama, dan cara penggambaran per *node* juga sama. Hasil pun sama yaitu *node* yang diberi warna. Tapi hasil yang diperoleh adalah seperti di bawah



Gambar 2. Pohon biner dari proses best-first search with branch and bound pruning

Keterangan: a/b/c,
a=keuntungan
b=berat
c=batas

Langkah-langkah yang dilakukan untuk mendapatkan pohon di atas adalah :

1. Kunjungi *node* (0,0) (akar)
   - Masukan keuntungan dan berat menjadi $0 dan 0
   - Hitung batasnya dengan rumus (5) yang telah diberikan
   - Set solusi sementara (maxprofit)=0
2. Kunjungi *node* (1,1)
   - Hitung keuntungan dan berat menjadi $40 dan 2

- Karena beratnya 2 dan kurang dari sama dengan 16, dan keuntungan lebih dari maxprofit, maka maxprofit = $40
- Hitung batasnya menjadi $115

3. Kunjungi *node* (1,2)
   - Hitung keuntungan dan berat menjadi $0 dan 0
   - Hitung batasnya menjadi $82
4. Menentukan node yang menjanjikan, node yang belum dikembangkan dengan batas terbesar
   - Karena *node* (1,1) memiliki batas $115 dan node (1,2) memiliki batas $82, *node* (1,1) adalah *node* yang belum dikembangkan dengan batas terbesar, maka (1,1) *node* yang menjanjikan. Karenanya kita akan mengunjungi anak-anaknya
5. Kunjungi *node* (2,1)
   - Hitung keuntungan dan beratnya menjadi $70 dan 7
   - Karena beratnya 7 kurang dari sama dengan 16, harga W, dan keuntungannya kebih besar dari $40, harga maxprofit, maka maxprofit = $70
   - Hitung batasnya menjadi $115
6. Kunjungi *node* (2,2)
   - Hitung keuntungan dan beratnya menjadi $40 dan 2
   - Hitung batasnya menjadi $98
7. Menentukan *node* yang belum dikembangkan yang menjanjikan dengan batas terbesar
   - *Node* yang didapat adalah (2,1). Karenanya kita akan mengunjungi anak-anaknya.
8. Kunjungi *node* (3,1)
   - Hitung keuntungan dan beratnya menjadi $120 dan 17
   - Menjadikan (3,1) sebagai *node* yang tidak menjanjikan karena beratnya lebih besar dari 16, harga W, dengan menjadikan batas= 0$
9. Kunjungi *node* (3,2)
   - Hitung keuntungan dan beratnya menjadi $70 dan 7
   - Hitung batasnya menjadi $80
10. Menentukan *node* yang belum dikembangkan yang menjanjikan dengan batas terbesar
    - *Node* yang didapat adalah (2,2). Karenanya kita akan mengunjungi anak-anaknya.
11. Kunjungi *node* (3,3)
    - Hitung keuntungan dan batasnya menjadi $90 dn 12
    - Karena beratnya 12 kurang dari sama dengan 16, harga W, dan keuntungannya kebih besar dari $70, harga maxprofit, maka maxprofit = $90
    - Pada tingkat ini, *node* (1,2) dan (3,2) menjadi tidak menjanjikan karena batasnya $82 dan $80 kurang dari sama dengan $90, harga dari maxprofit yang baru
    - Hitung batasnya menjadi $98
12. Kunjungi *node* (3,4)
    - Hitung keuntungan dan beratnya menjadi $40 dan 2
    - Hitung batasnya menjadi $50
    - Menentukan bahwa *node* ini tidak menjanjikan karena batasnya, $50, kurang dari sama dengan $90, maxprofit.
13. Menentukan *node* yang belum dikembangkan yang menjanjikan dengan batas terbesar
    - Tinggal satu *node* yang belum dikembangkan, *node* yang menjanjikan adalah *node* (3,3). Maka kita akan mengunjungi anak-anaknya.
14. Kunjungi *node* (4,1)
    - Hitung keuntungan dan beratnya menjadi $100 dan 17
    - Menentukan bahwa *node* (4,1) tidak menjanjikan karena 17 lebih besarsama dengan 16, harga W.
15. Kunjungi *node* (4,2)
    - Menghitung keuntungan dan beratnya menjadi $90 dan 12
    - Menghitung batasnya menjadi $90
    - Menentukan bahwa *node* (4,2) tidak menjanjikan karena batasnya $90 kurang dari sama dengan $90, harga maxprofit. Daun dari pohon secara otomatis menjadi tidak menjanjikan karena batas mereka tidak dapat melebihi maxprofit.

Karena sudah tidak ada *node* yang belum dikembangkan dan menjanjikan. Maka proses selesai dan maxprofit = $90.

Dengan langkah-langkah ini, maka jumlah *node* yang harus kita periksa berkurang dari17 pada *breadth-first search* menjadi 11 pada *best-first search.* Dan untuk kasus dengan skala yang lebih besar, perbedaan yang terlihat akan semakin besar. Walaupun begitu tidak ada jaminan bahwa nilai best yang keluar di tengah adalah optimasi dari permasalahan 0-1 Knapsack. Sehingga tetap saja proses ini harus ditelusuri sampai akhir.

Pada penerapannya, *best-first search* akan lebih baik menggunakan antrian prioritas (*priority queue*). Pada *priority queue*, elemen dengan prioritas tertinggi selalu dihilangkan pada proses selanjutnya. Pada *best-first search* elemen dengan prioritas tertinggi adalah *node* dengan batas terbaik.

Algoritma best-first search with branch and bound pruning [2]

procedure
best_first_search_with_branch_and_bound
(*T*:state_space_tree; var best: number);

```
var
    PQ : priority_queue_of_node;
    u,v: node;
begin
    initialize(PQ);
    v:=root of T;
    best:= value(v);
    insert(PQ,v);
    while not empty (PQ) do
        remove(PQ,v)
        if bound(v) is better than best then
              for each child u of v do
                  if value(u) is better than best then
                      best:=value (u)
                  end;
                  if  bound(u) is better than best then
                    insert(PQ,u)
                  end
              end
        end
    end
end;
```

Selain menggunakan *priority queue*, telah ditambahkan fungsi untuk menghapus node dari *priority queue*, ini akan terjadi jika sebuah *node* sudah dianggap tidak lagi menjanjikan, maka dia akan dihapus.

IV. KESIMPULAN

Dari seluruh penjelasan di atas, di ambil kesimpulan :

- Knapsack adalah permasalahan kombinasi yang melibatkan optimasi, dan terdapat 3 variasi masalah dilihat dari jumlah tiap barang *i* yang ditangani. Yaitu 0-1 Knapsack, Knapsack Terbatas, dan Knapsack Tak terbatas.
- Algoritma Branch and Bound dalam pemecahan 0-1 Knapsack mirip dengan *backtracking* dalam pencarian di pohon
- Ada dua macam Algoritma Branch and Bound dalam masalah 0-1 Knapsack, yaitu *breadth-first search with branch and bound pruning* dan *best-first search with branch and bound pruning*.
- Pemecahan masalah 0-1 Knapsack mengunakan Algoritma Branch and Bound melibatkan kombinatorial, optimasi, dan pohon.

REFERENSI

[1] ----------,"Knapsack Problem," http://en.wikipedia.org/wiki/Knapsack_problem.htm (Tanggal akses: 19 Desember 2009, pukul 17.20 WIB)

[2] Richard E. Neapolitan dkk, "Foundation of Algorithms", D.C. Heath and Company, 1996.

[3] Jean Clausen, "Branch and Bound Algorithms -Principles and Examples" Department of Computer Science, University of Copenhagen, 4-19

[4] ----------,"Branch and Bound" http://www.academic.marist.edu/~jzbv/algorithms/Branch%20and%20Bound.htm (Tanggal akses: 19 Desember 2009, pukul 18.16 WIB)

[5] ----------,"Branch and Bound" http://en.wikipedia.org/wiki/Branch_and_bound (Tanggal akses: 19 Desember 2009, pukul 18.10 WIB)