

KOMPARASI STRATEGI DYNAMIC PROGRAMMING DAN BRANCH AND BOUND UNTUK PERMASALAHAN 0/1 KNAPSACK

Dindin Dhino Alamsyah¹⁾, Anggie Nastiti²⁾, Faza Ghassani³⁾, Riki Nur Afifuddin⁴⁾

Program Studi S1 Teknik Informatika
Telkom University

Jalan Telekomunikasi Nomor 1, Bandung

e-mail: dindhinoo@gmail.com¹⁾, anggienstt21@gmail.com²⁾, oranyeu.fg@gmail.com³⁾, rikinur34@gmail.com⁴⁾

ABSTRAK

Knapsack adalah permasalahan dalam menentukan pilihan objek dari sekumpulan objek yang masing-masing mempunyai bobot/berat (weight) dan nilai/profit (value) untuk dimuat dalam sebuah media penyimpanan tanpa melebihi kapasitas media penyimpanan tersebut sehingga diperoleh hasil yang optimal. Pemilihan objek didasarkan pada kombinasi objek yang akan menghasilkan nilai tertinggi dan masih memenuhi kapasitas penyimpanan. Permasalahan knapsack terbagi menjadi dua, yaitu 0/1 knapsack dan fractional knapsack. Persoalan 0/1 knapsack adalah menentukan objek mana saja yang harus dimuat dan yang tidak dimuat, sedangkan persoalan fractional knapsack adalah menentukan berapa bagian dari masing-masing objek yang akan dimuat dalam media penyimpanan. Ada beberapa cara untuk memecahkan masalah knapsack, diantaranya menggunakan strategi algoritma dynamic programming dan branch and bound. Cara yang berbeda akan menyebabkan perbedaan proses pencapaian optimasi. Selain harus memperhatikan kemungkinan kebocoran kasus menggunakan salah satu strategi algoritma, kompleksitas algoritma yang digunakan juga menjadi pertimbangan lain bagi pemilihan algoritma untuk penyelesaian persoalan knapsack. Strategi algoritma dynamic programming dan branch and bound memiliki kelebihan dan kekurangan dalam memecahkan persoalan yang berhubungan dengan optimalisasi. Dalam makalah ini, dibahas perbedaan strategi algoritma dynamic programming dan branch and bound dalam permasalahan 0/1 knapsack.

Kata kunci: Knapsack, Dynamic programming, Branch and Bound, Kompleksitas.

1. PENDAHULUAN

Optimalisasi merupakan salah satu masalah klasik yang dihadapi dalam berbagai bidang ilmu. Banyak kasus yang

membutuhkan optimalisasi yang baik dari berbagai kombinasi *input* yang mungkin diberikan untuk mendapatkan efisiensi dalam prosesnya serta menghasilkan keluaran yang benar-benar optimal. Diantara permasalahan yang ada, beberapa contoh kasus yang termasuk dalam kasus optimasi diantaranya adalah masalah TSP (*Travelling Salesman Problem*), MST (*Minimum Spanning Tree*), dan *Knapsack Problem*. Beberapa metode algoritma yang dipakai dalam menyelesaikan beberapa kasus optimasi diantaranya metode *Dynamic Programming* dan *Branch and Bound*. Setiap metode memiliki sifat dan ciri yang berbeda-beda.

Knapsack merupakan salah satu permasalahan optimasi yang dapat diselesaikan oleh beberapa cara yang berbeda. Logika *knapsack* sering digunakan, tetapi masih banyak orang yang belum menyadarinya. Masalah *knapsack* merupakan suatu persoalan yang menarik. Dalam dunia nyata, permasalahan *knapsack* ini sering sekali digunakan terutama pada bidang (jasa) pengangkutan barang (seperti pengangkutan peti kemas dalam sebuah kapal). Dalam usaha tersebut, diinginkan suatu keuntungan yang maksimal untuk mengangkut barang yang ada dengan tidak melebihi batas kapasitas yang ada. Berdasarkan persoalan tersebut, diharapkan ada suatu solusi yang secara otomatis dalam mengatasi persoalan itu. Contoh kasus lain yang menggunakan algoritma ini antara lain pengisian barang di bagasi, pengisian barang di suatu perusahaan, pengoptimalisasi karyawan dalam suatu badan usaha, dan lain sebagainya.

Pemilihan algoritma dan penggunaan algoritma yang tepat akan membantu untuk menyelesaikan kasus *knapsack* dengan baik. Sebaliknya, ketidaktepatan memilih salah satu algoritma optimalisasi akan menyebabkan terhambatnya proses pengambilan keputusan, yang dalam makalah ini dikhususkan pada permasalahan 0/1 *knapsack*.

2. METODE

Metode yang digunakan untuk memecahkan permasalahan 0/1 *knapsack* ini adalah metode *Dynamic Programming* dan metode *Branch and Bound*.

2.1. DYNAMIC PROGRAMMING

Program dinamis (*dynamic programming*) merupakan metode pemecahan masalah dengan cara menguraikan solusi menjadi sekumpulan langkah (*step*) atau tahapan (*stage*) sedemikian sehingga solusi dari persoalan dapat dipandang dari serangkaian keputusan yang saling berkaitan.^[1]

Pada penyelesaian persoalan dengan metode ini :

1. Terdapat sejumlah berhingga pilihan yang mungkin
2. Solusi pada setiap tahap dibangun dari hasil solusi tahap sebelumnya
3. Menggunakan persyaratan optimasi dan kendala untuk membatasi sejumlah pilihan yang harus dipertimbangkan pada suatu tahap

[1]

Pada program dinamis, rangkaian keputusan yang optimal dibuat dengan menggunakan prinsip optimalitas dimana jika solusi total optimal, maka bagian solusi sampai tahap terakhir juga optimal.

2.1.1. TEKNIK DYNAMIC PROGRAMMING

Teknik yang digunakan untuk metode *dynamic programming* adalah dengan algoritma sebagai berikut.

```

Input : n, w[1]....w[n], p[1]....p[n], k
for i = 0 to k
  m[0, i] = 0
for i = 1 to n
  for j = 0 to k
    if w[i] > j then
      m[i, j] = m[i-1, j]
    else
      m[i, j] = max(m[i-1, j], m[i-1, j-w[i]]+ p[i])
return m[n, k]
  
```

Perhitungan kompleksitas secara matematis untuk *Dynamic Programming* adalah sebagai berikut.

$$T(n) = \sum_{i=1}^n \sum_{j=0}^k 1 \quad (1)$$

$$T(n) = \sum_{i=1}^n k - 0 + 1 = \sum_{i=1}^n k + 1 \quad (2)$$

$$T(n) = (n - 1 + 1)(k + 1) = n(k + 1) \quad (3)$$

$$T(n) \in O(n(k + 1)) \quad (4)$$

Kompleksitas untuk algoritma tersebut adalah $O(n(k+1))$.

2.1.2. CONTOH KASUS

Diberikan sebuah kasus untuk menentukan profit maksimal jika diketahui kapasitas bobot sebesar 11 dengan object, bobot, dan profit sebagai berikut.

Tabel 1 Objek, bobot, dan profit untuk contoh kasus *dynamic programming*

Objek	Bobot	Profit
1	2	6
2	1	1
3	5	18
4	6	22
5	7	28

Penyelesaian untuk kasus tersebut dengan metode *bottom-up dynamic programming* adalah sebagai berikut.

Tabel 2 Solusi menggunakan *dynamic programming*

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	40

Karena menggunakan strategi algoritma *Dynamic Programming* dengan metode *Bottom-up*, proses ke j bergantung pada proses j-1 yang berarti bahwa tiap proses dalam metode *Dynamic Programming* bergantung pada proses sebelumnya yang mana solusi dari metode *Dynamic Programming* adalah proses terakhir. Solusi optimal dari kasus tersebut adalah profit maksimal sebesar 40.

Penyelesaian contoh kasus tersebut menggunakan strategi algoritma *Brute Force* adalah sebagai berikut.

Tabel 3 Penyelesaian dengan strategi algoritma *Brute Force*

Object	Bobot	Profit
{0}	0	0
{1}	2	6
{2}	1	1
{3}	5	18
{4}	6	22
{5}	7	28
{1,2}	3	7
{1,3}	7	24
{1,4}	8	28
{1,5}	9	34
{2,3}	6	19
{2,4}	7	23
{2,5}	8	29
{3,4}	11	40

{3,5}	12	46
{4,5}	13	50
{1,2,3}	8	25
{1,2,4}	9	29
{1,2,5}	10	35
{1,3,4}	13	46
{1,3,5}	14	52
{1,4,5}	15	56
{2,3,4}	12	41
{2,3,5}	13	47
{2,4,5}	14	51
{3,4,5}	18	68
{1,2,3,4}	14	47
{1,2,3,5}	15	53
{1,2,4,5}	16	57
{1,3,4,5}	20	74
{2,3,4,5}	19	69
{1,2,3,4,5}	21	75

Dengan membandingkan antara hasil dari strategi algoritma *Brute Force* dan hasil dari strategi algoritma *Dynamic Programming*, dapat disimpulkan bahwa strategi algoritma *Dynamic Programming* optimal.

2.1.3. RANCANGAN PROGRAM

Rancangan program untuk metode *dynamic programming* adalah sebagai berikut.

```

1 program knapsackDP;
2 uses crt;
3 var
4   n,k,i,j : integer;
5   w,p : array of integer;
6   m : array of array of longint;
7 begin
8   clrscr;
9   writeln('input :');
10  readln(n);
11  setlength(w, n); setlength(p, n);
12  for i:=1 to n do readln(w[i], p[i]);
13  readln(k);
14  setlength(m, n+1, k+1);
15  for i:=0 to k do m[0,i] := 0;
16  for i:=1 to n do
17    for j:=0 to k do
18      begin
19        if w[i] > j then m[i,j] := m[i-1,j]
20        else
21          if m[i-1,j] > m[i-1,j-w[i]] + p[i] then m[i,j] := m[i-1,j]
22          else m[i,j] := m[i-1,j-w[i]] + p[i];
23        end;
24      end;
25  writeln('output :');
26  writeln(m[n][k]);
27  readln;
28 end.

```

Gambar 1 Program knapsack dengan dynamic programming

Hasil program berdasarkan contoh kasus adalah sebagai berikut.

```

D:\Programming\Pascal\knapsackdp.exe
input :
5
2 6
1 1
5 18
6 22
7 28
11
output :
40

```

Gambar 2 Hasil program dengan dynamic programming

Dengan menambahkan source code :

```

1 uses crt,sysutils,dateutils;
2 var
3   start, duration : TDateTime;
4 begin
5   start := Now;
6   //fungsi
7   duration := MilliSecondsBetween(Now, start);
8   writeln(duration:0:5);
9   readln;
10 end.

```

Gambar 3 Hitung waktu eksekusi dynamic programming

Sehingga program menjadi :

```

1 program knapsackDP;
2 uses crt,sysutils,dateutils;
3 var
4   n,k,i,j : integer;
5   w,p : array of integer;
6   m : array of array of longint;
7   start, duration : TDateTime;
8 begin
9   clrscr;
10  writeln('input :');
11  readln(n);
12  setlength(w, n); setlength(p, n);
13  for i:=1 to n do readln(w[i], p[i]);
14  readln(k);
15  setlength(m, n+1, k+1);
16  start := Now;
17  for i:=0 to k do m[0,i] := 0;
18  for i:=1 to n do
19    for j:=0 to k do
20      begin
21        if w[i] > j then m[i,j] := m[i-1,j]
22        else
23          if m[i-1,j] > m[i-1,j-w[i]] + p[i] then m[i,j] := m[i-1,j]
24          else m[i,j] := m[i-1,j-w[i]] + p[i];
25        end;
26      end;
27  duration := MilliSecondsBetween(Now, start);
28  writeln('output :');
29  writeln(m[n][k]);
30  writeln('durasi : ', duration:0:5);
31  readln;
32 end.

```

Console

```

Copyright (c) 1993-2015 by Florian Klaempfl and others
Target OS: Win32 for i386
Compiling knapsackDP.pas
Linking knapsackDP.exe
30 lines compiled, 0.4 sec, 73024 bytes code, 4292 bytes data
<<< Process finished. (Exit code 0)
===== READY =====

```

Gambar 4 Dynamic programming dengan waktu eksekusi

Waktu eksekusi algoritma *dynamic programming* berdasarkan contoh kasus diatas adalah lebih kecil dari 0.000001 ms yang dapat diliat dari gambar berikut.

```

D:\Programming\Pascal\knapsackdp.exe
input :
5
2 6
1 1
5 18
6 22
7 28
11
output :
40
durasi : 0.00000

```

Gambar 5 Waktu eksekusi dynamic programming

2.2. BRANCH AND BOUND

Metode algoritma umum yang digunakan untuk mencari solusi optimal dari berbagai permasalahan optimasi, terutama optimasi diskrit maupun kombinatorial adalah *Branch and Bound*. Algoritma ini memiliki kesamaan dengan algoritma runut-balik dimana metode pencarian dalam ruang solusi dilakukan secara sistematis dimana ruang solusi tersebut diimplementasikan kedalam pohon status. [2]

Solusi yang dibangun oleh algoritma *Branch and Bound* adalah BFS (*Breadth First Search*) berbeda dengan algoritma runut balik yang menggunakan skema DFS (*Depth First Search*). [3]

Algoritma *Branch and Bound* memiliki dua langkah, yaitu:

1. *Branch*, untuk membangun sebuah cabang pohon yang mungkin menuju solusi.
2. *Bound*, untuk menghitung node mana yang menjadi *active node* (E-node) dan node mana yang menjadi *dead node* (D-node) dengan menggunakan syarat batas *constraint* (kendala).

[4]

Metode ini dapat digunakan dalam menyelesaikan berbagai masalah menggunakan *Search Tree*, seperti:

1. Traveling Salesman Problem
2. N-Queen Problem
3. 15 Puzzle Problem
4. 0/1 Knapsack Problem
5. Shortest Path

[2]

Nilai batas untuk setiap simpul umumnya berupa taksiran atau perkiraan. Fungsi heuristik untuk menghitung taksiran cost yaitu:

$$\begin{aligned}
 c(i) &= f(i) + g(i), \text{ ongkos untuk simpul } i \\
 f(i) &= \text{ongkos mencapai simpul } i \text{ dari akar,} \\
 g(i) &= \text{ongkos mencapai simpul tujuan dari simpul } i
 \end{aligned}$$

Simpul berikutnya yang dipilih untuk di ekspansi adalah simpul yang memiliki c minimum. [1]

Simpul-E adalah simpul yang memiliki minimum nilai, dimana nilai tersebut digunakan untuk mengurutkan pencarian simbol berikutnya yang akan dipilih untuk dilakukan ekspansi. Strategi memilih simpul-E seperti ini dinamakan strategi pencarian berdasarkan biaya terkecil (*least cost search*).

Prinsip dari algoritma *branch and bound* ini adalah:

1. Masukkan simpul akar ke dalam antrian Q. Jika simpul akar adalah simpul solusi (goal node), 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 terdapat beberapa simpul i yang memenuhi, pilih satu secara sembarang.
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.

[3]

2.2.1. TEKNIK BRANCH AND BOUND

Ada beberapa teknik yang digunakan dalam algoritma *Branch and Bound*, antara lain:

1. *FIFO Branch and Bound*

Teknik yang menggunakan bantuan queue untuk perhitungan secara *First in First Out*.

2. *LIFO Branch and Bound*

Teknik yang menggunakan bantuan stack untuk perhitungan *Last in First Out*.

3. *Least Cost Branch and Bound*

Teknik yang akan menghitung *cost* setiap *node*. *Node* yang memiliki *cost* yang paling kecil kemungkinan paling besar memiliki kesempatan menjadi calon solusi.

[4]

2.2.2. PENGHENTIAN PERCABANGAN (*FATHOMING*)

Pencabangan atau pencarian solusi pada suatu masalah dihentikan jika:

1. Semua variable keputusan yang seharusnya bernilai *integer* sudah bernilai *integer*.
2. *Infeasible* atau tidak mempunyai daerah layak.
3. Pada masalah maksimisasi, penghentian pencabangan pada suatu sub masalah dilakukan jika batas atas dari

sub masalah tersebut tidak lebih besar atau sama dengan batas bawah.

4. Sedangkan pada masalah minimisasi penghentian pencabangan pada suatu sub masalah dilakukan jika batas bawah tidak lebih lebih kecil atau sama dengan batas atas.

[4]

2.2.3. KONDISI OPTIMAL

Algoritma *Branch and Bound* mencapai kondisi yang optimal jika:

1. Tidak ada lagi submasalah yang diperlukan lagi, sehingga solusi optimal sudah diperoleh.
2. Pada masalah maksimisasi solusi optimal merupakan solusi submasalah yang saat ini menjadi batas bawah (*lower bound*).
3. Pada masalah minimisasi solusi optimal merupakan solusi submasalah yang saat ini menjadi batas atas (*upper bound*).

[4]

2.2.4. CONTOH KASUS

Diberikan sebuah kasus untuk menentukan profit maksimal jika diketahui kapasitas bobot sebesar 11 dengan object, bobot, dan profit sebagai berikut.

Tabel 4 Objek, bobot, dan profit untuk contoh kasus branch and bound

Objek	Bobot	Profit
1	2	6
2	1	1
3	5	18
4	6	22
5	7	28

Penyelesaian untuk kasus tersebut dengan metode *branch and bound* adalah sebagai berikut.

Langkah pertama adalah melakukan proses pengurutan data berdasarkan density (profit/bobot) yang paling besar sehingga data menjadi :

Tabel 5 Data terurut untuk contoh kasus dengan metode Branch and Bound

Objek	Bobot	Profit	Density (Profit/Bobot)
5	7	28	4
4	6	22	3,67
3	5	18	3,6
1	2	6	3
2	1	1	1

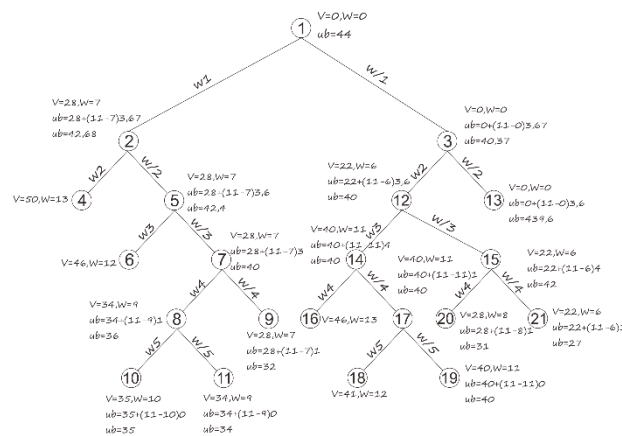
Digunakan perhitungan untuk teknik *bounding* sebagai berikut.

$$ub_i = profit_i + (kapasitas - bobot_i)(density_{i+1}) \quad (5)$$

Untuk node awal, digunakan profit dan bobot sebesar 0 dan perhitungan *bound* sebagai berikut.

$$ub_{awal} = 0 + (11 - 0)(4) = 44 \quad (6)$$

Perhitungan *bound* dilakukan disetiap iterasi sehingga diperoleh :



Gambar 6 Solusi menggunakan branch and bound

Dengan menggunakan strategi algoritma *Branch and Bound*, diperoleh satu objek dengan bobot 6 dan profit 22 dan satu objk dengan bobot 5 dan profit 18, sehingga solusi optimal adalah total profil maksimal sebesar 40.

Penyelesaian contoh kasus tersebut menggunakan strategi algoritma *Brute Force* adalah sebagai berikut.

Tabel 6 Penyelesaian dengan strategi algoritma Brute Force

Object	Bobot	Profit
{0}	0	0
{1}	2	6
{2}	1	1
{3}	5	18
{4}	6	22

{5}	7	28
{1,2}	3	7
{1,3}	7	24
{1,4}	8	28
{1,5}	9	34
{2,3}	6	19
{2,4}	7	23
{2,5}	8	29
{3,4}	11	40
{3,5}	12	46
{4,5}	13	50
{1,2,3}	8	25
{1,2,4}	9	29
{1,2,5}	10	35
{1,3,4}	13	46
{1,3,5}	14	52
{1,4,5}	15	56
{2,3,4}	12	41
{2,3,5}	13	47
{2,4,5}	14	51
{3,4,5}	18	68
{1,2,3,4}	14	47
{1,2,3,5}	15	53
{1,2,4,5}	16	57
{1,3,4,5}	20	74
{2,3,4,5}	19	69
{1,2,3,4,5}	21	75

Dengan membandingkan antara hasil dari strategi algoritma *Brute Force* dan hasil dari strategi algoritma *Branch and Bound*, dapat disimpulkan bahwa strategi algoritma *Branch and Bound* optimal.

2.2.5. RANCANGAN PROGRAM

Rancangan program untuk menyelesaikan permasalahan 0/1 knapsack dengan menggunakan algoritma *Branch and Bound* adalah sebagai berikut.

```
long knapsack(long W, Objek arr[], long n)
{
    bubblesort(arr, n);
    queue<Node> Q;
    Node u, v;
    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);
    long maxprofit = 0;
    while (!Q.empty())
    {
        u = Q.front();
        Q.pop();

        if (u.level == -1)
        {
            v.level = 0;
        }
        if (u.level == n-1)
        {
            continue;
        }
        v.level = u.level + 1;
        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].profit;
        if (v.weight <= W && v.profit > maxprofit)
        {
            maxprofit = v.profit;
        }
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxprofit)
        {
            Q.push(v);
        }
        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxprofit)
        {
            Q.push(v);
        }
    }
    return maxprofit;
}
```

Gambar 7 Rancangan program dengan algoritma branch and bound

Karena perulangan menggunakan perulangan *while* dan akan berhenti ketika *queue* tidak memiliki *element*, maka kompleksitas untuk algoritma tersebut adalah $O(n)$.

Rancangan program tersebut memiliki *abstract data type* dan fungsi untuk membagi profit dan bobot sebagai berikut.

```
#include <iostream>
#include <queue>

using namespace std;

struct Objek
{
    long weight, profit;
};

struct Node
{
    long weight, profit, level, bound;
};

double pperw(Objek a)
{
    return (double) a.profit/a.weight;
}
```

Gambar 8 Abstract data type dan fungsi untuk branch and bound

Dalam algoritma tersebut, digunakan metode *BubbleSort* dengan kompleksitas $O(n^2)$ untuk mengurutkan data berdasarkan nilai profit/bobot paling besar yang diimplementasikan sebagai berikut.

```
Barang* bubblesort(Barang arr[], long n)
{
    bool tuker = true;
    long j = 0;
    while (tukoer)
    {
        tuker = false;
        j++;
        for (long i=0; i<n-j; i++)
        {
            if (pperw(arr[i]) < pperw(arr[i+1]))
            {
                Barang tmp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = tmp;
                tuker = true;
            }
        }
    }
    return arr;
}
```

Gambar 9 Bubblesort untuk branch and bound

Untuk mencari batas atas dan batas bawah untuk mencari solusi yang optimal pada subregion dengan menggunakan teknik bounding, implementasi kodenya adalah sebagai berikut.

```
long bound(Node u, long n, long W, Barang arr[])
{
    if (u.weight >= W)
    {
        return 0;
    }

    long profit_bound = u.profit;
    long j = u.level + 1;
    long totweight = u.weight;

    while ((j < n) && (totweight + arr[j].weight <= W))
    {
        totweight += arr[j].weight;
        profit_bound += arr[j].profit;
        j++;
    }
    if (j < n)
    {
        profit_bound += (W - totweight) * arr[j].profit / arr[j].weight;
    }

    return profit_bound;
}
```

Gambar 10 Teknik bounding untuk branch and bound

Karena perulangan menggunakan perulangan *while* dan akan berhenti ketika nilai *j* lebih kecil dari nilai *n* dan hasil penjumlahan total bobot dengan bobot objek ke *j* lebih kecil atau sama dengan kapasitas, maka kompleksitas untuk teknik *bounding* adalah $O(n)$ yang dapat dibuktikan sebagai berikut.

$$T(n) = \sum_{j=0}^{n-1} 1 \quad (7)$$

$$T(n) = n - 1 - 0 + 1 = n \quad (8)$$

$$T(n) \in O(n) \quad (9)$$

Hasil program berdasarkan contoh kasus adalah sebagai berikut

D:\Programming\C_CS_CPP\Code

```
input :
5
2 6
1 1
5 18
6 22
7 28
11
output :
40
```

Gambar 11 Hasil program dengan metode branch and bound

Dengan menambahkan source code :

```
#include <time.h>

static double diffclock(clock_t clock1, clock_t clock2)
{
    double diffticks=clock1-clock2;
    double diffms=(diffticks)/(CLOCKS_PER_SEC/1000);
    return diffms;
}

int main() {
    int start,duration;

    start=clock();
    //CODE HERE
    duration=clock();
    duration=diffclock(duration, start);
    cout << "durasi: " << duration;
}
```

Gambar 12 Hitung waktu eksekusi Branch and Bound

Sehingga *main* program menjadi :

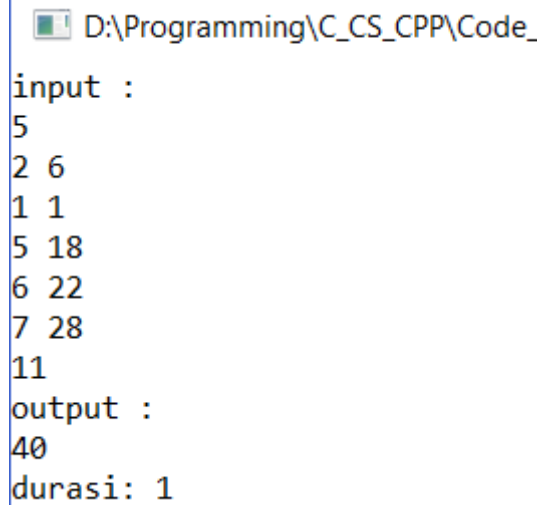
```
int main()
{
    long n,k;
    long* w = NULL;
    long* p = NULL;
    Objek* arr = NULL;

    cout << "input : " << endl;
    cin >> n;
    w = new long[n];
    p = new long[n];
    arr = new Objek[n];
    for (long i=0; i<n; i++)
    {
        cin >> w[i] >> p[i];
        arr[i].weight = w[i];
        arr[i].profit = p[i];
    }
    cin >> k;

    clock_t start=clock();
    cout << "output : " << endl;
    cout << knapsack(k, arr, n) << endl;
    clock_t duration=clock();
    duration=diffclock(duration, start);
    cout << "durasi: " << duration;
    cin.get(); cin.get();
    return 0;
}
```

Gambar 13 Main program Branch and Bound

Waktu eksekusi algoritma *dynamic programming* berdasarkan contoh kasus diatas adalah 1 ms yang dapat dilihat dari gambar berikut.



```

D:\Programming\C_CS_CPP\Code_
input :
5
2 6
1 1
5 18
6 22
7 28
11
output :
40
durasi: 1
  
```

Gambar 14 Waktu eksekusi Branch and Bound

2.3. PERBANDINGAN WAKTU EKSEKUSI DYNAMIC PROGRAMMING DAN BRANCH AND BOUND

Untuk membandingkan waktu eksekusi strategi algoritma *Dynamic Programming* dan *Branch and Bound*, diberikan data masukan sebagai berikut.

```

50
517 417
1034 834
2068 1668
1034 834
165 65
330 130
495 195
176 76
232 132
464 264
928 528
464 264
472 372
944 744
456 356
912 712
232 132
464 264
696 396
139 39
297 197
461 361
922 722
  
```

```

461 361
231 131
462 262
924 524
437 337
874 674
437 337
365 265
365 265
258 158
258 158
304 204
608 408
1216 816
912 612
300 200
600 400
220 120
440 240
660 360
303 203
606 406
606 406
303 203
606 406
909 609
522 422
10121
  
```

Diketahui bahwa baris pertama menunjukkan n buah objek dengan n sama dengan 50, n baris berikutnya menunjukkan bobot dan profit berturut-turut untuk masing-masing objek, dan baris terakhir menunjukkan kapasitas bobot sebesar 10121.

Dengan menggunakan strategi algoritma *Dynamic Programming*, hasil program berdasarkan kasus tersebut adalah sebagai berikut.


```
D:\Programming\Pascal\knapsackdp.exe
462 262
924 524
437 337
874 674
437 337
365 265
365 265
258 158
258 158
304 204
608 408
1216 816
912 612
300 200
600 400
220 120
440 240
660 360
303 203
606 406
606 406
303 203
606 406
909 609
522 422
10121
output :
8020
durasi : 4.00000
```

Gambar 15 Waktu eksekusi dengan Dynamic Programming

Dengan menggunakan strategi algoritma *Branch and Bound*, hasil program berdasarkan kasus tersebut adalah sebagai berikut.

```
D:\Programming\C_CS_CPP\Code_Blocks\KnapsackBnB'
462 262
924 524
437 337
874 674
437 337
365 265
365 265
258 158
258 158
304 204
608 408
1216 816
912 612
300 200
600 400
220 120
440 240
660 360
303 203
606 406
606 406
303 203
606 406
909 609
522 422
10121
output :
8020
durasi: 5
```

Gambar 16 Waktu eksekusi dengan Branch and Bound

Berdasarkan hasil program antara strategi algoritma *Dynamic Programming* dan strategi algoritma *Branch and Bound*, strategi algoritma *Dynamic Programming* menghasilkan waktu eksekusi yang lebih cepat daripada waktu eksekusi strategi algoritma *Branch and Bound*.

3. KESIMPULAN

Kompleksitas strategi algoritma *Branch and Bound* lebih baik daripada strategi algoritma *Dynamic Programming* karena strategi algoritma *Branch and Bound* memiliki kompleksitas $O(n)$, sedangkan strategi algoritma *Dynamic Programming* memiliki kompleksitas $O(n(k+1))$.

Waktu eksekusi strategi algoritma *Dynamic Programming* lebih baik daripada waktu eksekusi strategi algoritma *Branch and Bound* karena strategi algoritma *Branch and Bound* harus dilakukan *sorting*, metode *sorting* yang dipilih adalah *Bubblesort* dengan kompleksitas $O(n^2)$.

REFERENSI

- [1] Munir, Rinaldi, *Strategi Algoritmik*, Bandung, 2007.
- [2] Nur, Hayati, “*Aplikasi Algoritma Branch and Bound Untuk Menyelesaikan Integer Programming*”, *Dinamika Teknik*, Vol. IV, No.1, 2010, 13-23.
- [3] Shena, Anggie, “*Pemecahan Masalah Knapsack Dengan Menggunakan Algoritma Branch and Bound*”, Institut Teknologi Bandung, 2007
- [4] Noviani, Herty, “*Metode Branch & Bound*”, 2015.