TUGAS AKHIR – KI141502

IMPLEMENTASI ALGORITMA PENCARIAN K SOLUSI TERBAIK UNTUK PERMASALAHAN KNAPSACK

Indra Saputra
NRP 5111 100 066

Dosen Pembimbing 1
Ahmad Saikhu, S.Si., M.T.

Dosen Pembimbing 2
Rully Soelaiman, S.Kom., M.Kom.

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya 2015

UNDERGRADUATE THESIS – KI141502

IMPLEMENTATION OF FINDING K BEST SOLUTIONS ALGORITHM FOR KNAPSACK PROBLEM

Indra Saputra
NRP. 5111 100 066

Supervisor 1
Ahmad Saikhu, S.Si., M.T.

Supervisor 2
Rully Soelaiman, S.Kom., M.Kom.

DEPARTMENT OF INFORMATICS
Faculty of Information Technology
Institut Teknologi Sepuluh Nopember
Surabaya 2015

LEMBAR PENGESAHAN

IMPLEMENTASI ALGORITMA PENCARIAN K SOLUSI TERBAIK UNTUK PERMASALAHAN KNAPSACK

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Dasar dan Terapan Komputasi
Program Studi S-1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh:

INDRA SAPUTRA
NRP 5111100066

Disetujui oleh Dosen Pembimbing Tugas Akhir:

Ahmad Saikhu, S.Si., M.T. .....
NIP 197107182006041001 (Pembimbing I)

Rully Soelaiman, S.Kom., M.Kom. .....
NIP 197002131994021001 (Pembimbing II)

SURABAYA
JUNI 2015

IMPLEMENTASI ALGORITMA PENCARIAN K SOLUSI TERBAIK UNTUK PERMASALAHAN KNAPSACK

Nama	**:**	**Indra Saputra**
NRP	**:**	**5111100066**
Jurusan	**:**	**Teknik Informatika** **Fakultas Teknologi Informasi**
Dosen Pembimbing I	**:**	**Ahmad Saikhu, S.Si., M.T.**
Dosen Pembimbing II	**:**	**Rully Soelaiman, S.Kom., M.Kom.**

ABSTRAK

Permasalahan knapsack adalah permasalahan klasik yang ada dalam bidang algoritma. Biasanya, dalam permasalahan knapsack, tujuan penyelesaian hanyalah mencari sebuah solusi terbaik dari semua kombinasi yang ada. Namun, pada kenyataannya, seringkali dibutuhkan beberapa solusi terbaik yang dapat digunakan sebagai rekomendasi.

Pencarian beberapa solusi terbaik pada permasalahan knapsack masih jarang diteliti. Beberapa algoritma yang diusulkan terbukti dapat mencari solusi terbaik tetapi memiliki waktu eksekusi yang sangat lambat. Beberapa lainnya memiliki waktu eksekusi yang cepat tetapi tidak dapat dipastikan bahwa hasilnya valid.

Tugas Akhir ini mengimplementasi pencarian beberapa solusi terbaik pada permasalahan knapsack dengan metode Branch and Bound. Metode Branch and Bound dipilih karena terbukti mampu menyelesaikan permasalahan kombinasi dengan waktu eksekusi yang cepat.

Hasil uji coba menunjukkan bahwa metode Branch and Bound mampu menyelesaikan permasalahan yang diajukan. Hasil uji coba kebenaran membuktikan bahwa metode Branch and Bound mampu memberikan hasil yang valid. Hasil uji coba kinerja

membuktikan bahwa metode Branch and Bound mampu menyelesaikan 7 dari 8 kelas dataset yang tersedia.

Kata kunci: Branch and Bound, K Solusi Terbaik, Permasalahan Knapsack.

IMPLEMENTATION OF FINDING K BEST SOLUTIONS ALGORITHM FOR KNAPSACK PROBLEM

Name	**:**	**Indra Saputra**
NRP	**:**	**5111100066**
Department	**:**	**Department of Informatics** **Faculty of Information Technology**
Supervisor 1	**:**	**Ahmad Saikhu, S.Si., M.T.**
Supervisor 2	**:**	**Rully Soelaiman, S.Kom., M.Kom.**

ABSTRACT

Knapsack problem is a classic problem in the field of algorithm. Usually, in the knapsack problem, the main purpose is determining the best solution of all possible solutions. However, in fact, it needs more than one best solution that can be used as a recommendation.

Determining several best solutions of knapsack problem is still rarely discovered. Some proposed algorithms proved to be able to determine some best solutions but the execution time was very slow. The others have a rapid execution time but they can't be guaranted that the result are valid.

This undergraduate thesis implements the Branch and Bound method to determine several best solutions of knapsack problem. Branch and Bound is chosen because it was proved to be able to solve combinational problem in no time.

The trial result shows that Branch and Bound method is able to solve the proposed problem. The trial result of correctness demonstrates that the Branch and Bound method is able to provide a valid result. The trial result of performance demonstrates that the Branch and Bound method is able to solve 7 out of 8 classes of datasets.

Keyword: Branch and Bound, K Best Solutions, Knapsack Problem.

KATA PENGANTAR

بِسۡمِ ٱللَّهِ ٱلرَّحۡمَٰنِ ٱلرَّحِيمِ

Puji syukur penulis panjatkan kepada Tuhan Yang Maha Esa yang telah memberikan rahmat dan karunia-Nya sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul:

IMPLEMENTASI ALGORITMA PENCARIAN K SOLUSI TERBAIK UNTUK PERMASALAHAN KNAPSACK

Tugas Akhir ini diajukan untuk memenuhi salah satu syarat memperoleh gelar Sarjana Komputer di Jurusan Teknik Informatika, Fakultas Teknologi Informasi, Institut Teknologi Sepuluh Nopember.

Penulis menyampaikan terima kasih kepada semua pihak yang telah memberikan dukungan, baik secara langsung maupun tidak langsung, selama penulis menyelesaikan Tugas Akhir. Tanpa mengurangi rasa hormat kepada pihak lain, penulis ingin mengucapkan terima kasih kepada:

1. Tuhan Yang Maha Esa yang telah memberikan rahmat dan karunia-Nya sehingga penulis dapat menyelesaikan Tugas Akhir ini dengan baik.
2. Orang tua, adik, dan semua keluarga serta kerabat, terima kasih atas do'a dan bantuan moral serta material selama penulis belajar di Teknik Informatika ITS.
3. Ibu Dr. Eng. Nanik Suciati, S.Kom., M.Kom. selaku Ketua Jurusan Teknik Informatika, FTIf, ITS.
4. Bapak Radityo Anggoro, S.Kom., M.Sc. selaku koordinator Tugas Akhir.
5. Bapak Ahmad Saikhu, S.Si., M.T. selaku Dosen Pembimbing I yang telah memberikan dukungan dan bimbingan.

6. Bapak Rully Soelaiman, S.Kom., M.Kom. selaku Dosen Pembimbing II yang telah memberikan dukungan, bimbingan, waktu untuk berdiskusi, saran, kritik, dan berbagai pengalaman hidup yang sangat memotivasi dan membuat penulis menjadi pribadi yang lebih baik.
7. Bapak Dwi Sunaryono, S.Kom., M.Kom. dan Bapak Daniel Oranova Siahaan, S.Kom., M.Sc., PD.Eng. selaku dosen wali penulis.
8. Bapak dan Ibu dosen Jurusan Teknik Informatika ITS yang telah memberikan ilmu selama perkuliahan.
9. Seluruh staf dan karyawan Jurusan Teknik Informatika ITS atas bantuan yang diberikan selama perkuliahan.
10. Rekan-rekan Kader Terbaik Bangsa, TC Hura-Hura, panitia NLC Schematics 2013, administrator dan pengguna Laboratorium Algoritma dan Pemrograman, serta angkatan 2011 Teknik Informatika ITS yang telah memberikan pengetahuan, ilmu, hiburan, dukungan, masukan, motivasi, cinta, dan kasih sayang kepada penulis.
11. Seluruh pihak yang tidak bisa penulis sebutkan satu persatu yang telah banyak memberikan dukungan selama ini.

Penulis mohon maaf apabila terdapat kekurangan dalam Tugas Akhir ini. Kritik dan saran selalu penulis harapkan untuk perbaikan dan pembelajaran di kemudian hari. Semoga Tugas Akhir ini dapat memberikan manfaat sebanyak-banyaknya.

Surabaya, Juni 2015

Penulis

DAFTAR ISI

DAFTAR TABEL

DAFTAR KODE SUMBER

DAFTAR GAMBAR

BAB I
PENDAHULUAN

1.1. Latar Belakang

Permasalahan knapsack adalah permasalahan klasik yang ada di bidang algoritma. Pada permasalahan knapsack, terdapat beberapa buah *item* dengan nilai, berat, dan jumlah pengambilan tertentu. Di samping itu, terdapat juga sebuah knapsack dengan kapasitas tertentu yang dapat memuat *item-item* tersebut.

Permasalahan knapsack yang sering muncul adalah permasalahan knapsack 0-1. Pada permasalahan knapsack 0-1, setiap *item* hanya dapat diambil maksimal sekali saja. Sehingga, hanya ada pilihan mengambil atau tidak mengambil pada permasalahan knapsack 0-1.

Biasanya, permasalahan yang muncul adalah bagaimana cara mencari kombinasi *item* terbaik sehingga *item-item* yang diambil memiliki jumlah nilai yang maksimal dan berat total tidak melebihi kapasitas knapsack. Permasalahan ini dapat diselesaikan dengan berbagai metode, seperti *dynamic programming* dan *branch and bound*. Namun, solusi yang ditemukan hanya fokus pada sebuah solusi terbaik saja.

Padahal, dalam dunia nyata seringkali diperlukan solusi lain selain solusi terbaik. Sebagai contoh, diperlukan beberapa solusi lain yang dapat digunakan sebagai rekomendasi apabila ternyata solusi terbaik bukanlah solusi yang diinginkan. Untuk mengatasi hal ini, cara yang paling mudah adalah dengan membangkitkan semua solusi yang mungkin. Namun, cara ini sangat tidak efisien mengingat keterbatasan waktu dan memori.

Tugas akhir ini akan mengimplementasi pencarian k solusi terbaik pada permasalahan knapsack. Pembangkitan k solusi terbaik menggunakan metode *Branch and Bound* yang memiliki kompleksitas waktu cukup cepat bahkan untuk data dengan jumlah banyak.

1.2. Rumusan Masalah

Berikut adalah beberapa rumusan masalah yang diangkat dalam tugas akhir ini:

1. Bagaimana desain algoritma untuk mencari *k* solusi terbaik pada permasalahan knapsack?
2. Bagaimana implementasi algoritma berdasarkan desain yang telah dilakukan?
3. Bagaimana uji coba untuk mengetahui kebenaran dan kinerja dari implementasi yang telah dilakukan?

1.3. Batasan Masalah

Berikut adalah beberapa hal yang menjadi batasan masalah dalam tugas akhir ini:

1. Implementasi algoritma menggunakan bahasa pemrograman C++ dengan bantuan IDE Code::Blocks 13.12.
2. Pembuktian kebenaran hasil implementasi akan dibandingkan dengan metode *complete search*.
3. Uji kecepatan algoritma menggunakan dataset Pisinger dengan mengujinya pada delapan kelas data yang tersedia.
4. Jumlah *item* maksimal yang digunakan untuk uji kebenaran sebanyak 25.
5. Jumlah *item* maksimal yang digunakan untuk uji kecepatan sebanyak 200.
6. Nilai dan berat *item* maksimal adalah 10000.

1.4. Tujuan

Tujuan pengerjaan tugas akhir ini adalah untuk mengimplementasikan algoritma untuk mencari *k* solusi terbaik pada permasalahan knapsack dan mengetahui kebenaran serta hasil kinerja hasil implementasi.

1.5. Manfaat

Manfaat yang diharapkan dari hasil tugas akhir ini adalah memberikan pengetahuan baru tentang beberapa solusi terbaik pada permasalahan kombinasi, terutama pada permasalahan knapsack. Beberapa solusi terbaik yang dicari diharapkan mampu menjadi rekomendasi pemilihan bagi masyarakat.

1.6. Metodologi

Berikut adalah beberapa tahap yang dilakukan selama proses pengerjaan tugas akhir ini:

1. Studi literatur
   Tahap ini merupakan proses mengerti dan memahami mengenai solusi untuk permasalahan pencarian k solusi terbaik pada permasalahan knapsack. Literatur yang digunakan berupa *paper*, artikel di internet, serta video.
2. Desain
   Tahap ini merupakan proses melakukan desain metode untuk mencari k solusi terbaik pada permasalahan knapsack.
3. Implementasi
   Tahap ini merupakan proses transformasi desain atau rancangan menjadi aplikasi yang dapat mencari k solusi terbaik pada permasalahan knapsack.
4. Uji coba dan evaluasi
   Tahap ini merupakan proses pembuktian kebenaran dan pengukuran kinerja dari hasil implementasi. Uji coba kebenaran dan pengukuran kinerja dilakukan terhadap beberapa dataset.
5. Penyusunan buku tugas akhir
   Tahap ini merupakan penulisan laporan akhir tugas akhir yang berisi dokumentasi seluruh kegiatan pengerjaan tugas akhir dari awal sampai akhir.

1.7. Sistematika Penulisan

Berikut adalah sistematika penulisan tugas akhir ini:

1. BAB I: PENDAHULUAN
   Bab ini berisi tentang latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat, metodologi, dan sistematika penulisan buku.
2. BAB II: TINJAUAN PUSTAKA
   Bab ini berisi tentang dasar teori yang digunakan untuk menyelesaikan permasalahan yang diajukan.
3. BAB III: DESAIN
   Bab ini berisi tentang desain algoritma dan struktur data yang digunakan untuk menyelesaikan permasalahan.
4. BAB IV: IMPLEMENTASI
   Bab ini berisi tentang implementasi algoritma dan struktur data yang digunakan untuk menyelesaikan permasalahan.
5. BAB V: UJI COBA DAN EVALUASI
   Bab ini berisi tentang uji coba dan evaluasi dari hasil implementasi yang telah dilakukan.
6. BAB VI: KESIMPULAN DAN SARAN
   Bab ini berisi tentang kesimpulan pengerjaan tugas akhir dan saran untuk pengembangan selanjutnya.

BAB II
TINJAUAN PUSTAKA

Bab ini berisi tentang dasar teori yang digunakan untuk menyelesaikan masalah yang ada di tugas akhir ini.

2.1. Permasalahan Knapsack

Permasalahan knapsack adalah suatu permasalahan klasik yang ada dalam bidang algoritma. Permasalahan knapsack merupakan salah satu jenis permasalahan kombinasi. Pada permasalahan knapsack, terdapat m buah *item*. Setiap *item* i memiliki nilai v_i dan berat w_i. Setiap *item* hanya memiliki pilihan untuk diambil sekali atau tidak diambil sama sekali. Selain itu, terdapat beban maksimal W yang menandakan daya tampung knapsack.

Tujuan dari permasalahan knapsack adalah mencari kombinasi pengambilan *item* dimana nilai total *item* yang diambil diusahakan semaksimal mungkin dan berat total *item* tidak melebihi daya tampung knapsack.

Secara matematis, permasalahan knapsack dituliskan sebagai berikut:

maksimalkan $\sum_{i=1}^{m} v_i x_i$ (1)

dengan batasan $\sum_{i=1}^{m} w_i x_i \leq W$ (2)

Nilai masing-masing $v_i \geq 0$ dan nilai masing-masing $w_i > 0$. Sedangkan x_i hanya bernilai 0 atau 1 dan i adalah integer dari 1 sampai m.

Apabila x_i bernilai 0, maka *item* ke-i tidak diambil. Sebaliknya, apabila x_i bernilai 1, maka *item* ke-i diambil. Kondisi ini seringkali disebut permasalahan knapsack 0-1.

Tabel 2.1 menyajikan empat contoh item dengan nilai dan berat masing-masing.

Tabel 2.1 Contoh *item* beserta nilai dan berat masing-masing

No.	Nilai	Berat
1	30	5
2	10	5
3	45	3
4	45	9

2.2. K Solusi Terbaik pada Permasalahan Knapsack

Umumnya, tujuan penyelesaian masalah dari permasalahan kombinasi seperti permasalahan knapsack adalah pencarian sebuah kombinasi terbaik. Artinya, pencarian hanya difokuskan pada sebuah solusi terbaik dari berbagai kombinasi solusi yang ada. Pada pencarian *k* solusi terbaik, pencarian tidak hanya difokuskan pada sebuah solusi saja, melainkan dilakukan pencarian dan pemilihan untuk mendapatkan solusi terbaik sebanyak *k* buah.

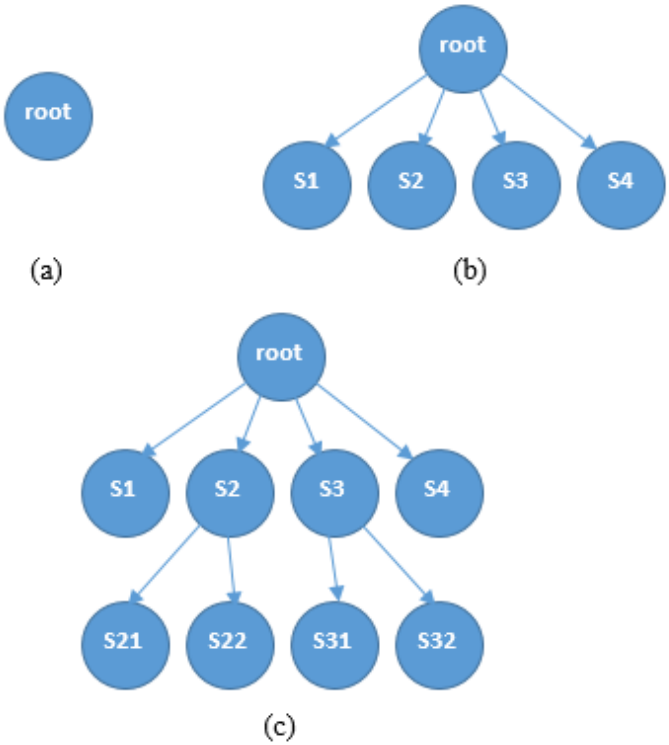
2.3. Algoritma Branch and Bound

Algoritma Branch and Bound (B&B) adalah salah satu algoritma yang sering digunakan untuk menyelesaikan permasalahan kombinasi. Pada dasarnya, algoritma B&B mencari semua kemungkinan solusi dari suatu permasalahan. Namun, hal ini sangat tidak mungkin dilakukan karena solusi yang ada berkembang secara polinomial. Oleh karena itu, fungsi *bound* digunakan untuk membuat algoritma ini hanya menyusuri solusi yang mungkin.

Awalnya, sebuah solusi dibangkitkan. Dari sebuah solusi ini, algoritma B&B akan membangkitkan solusi lain yang mungkin. Kemudian, algoritma B&B akan menelusuri solusi lain yang telah dibangkitkan. Lalu, dari solusi tersebut, dibangkitkan lagi solusi lain yang mungkin. Begitu seterusnya hingga ditemukan solusi yang diminta. Proses pembangkitan solusi lain dan melakukan

proses penelusuran ini disebut *branching*. Gambar 2.1 menunjukkan proses pembangkitan solusi.

Pada Gambar 2.1(a), solusi awal dibangkitkan. Gambar 2.1(b) menunjukkan bahwa solusi awal membangkitkan empat solusi lain. Kemudian, Gambar 2.1(c) menunjukkan solusi S2 dan S3 membangkitkan solusi lain, sedangkan solusi S1 dan S4 tidak membangkitkan solusi lain. Hal ini dapat terjadi karena anak solusi dari S1 dan S4 tidak memenuhi kriteria atau fungsi *bound* untuk dibangkitkan atau dengan kata lain S1 dan S4 tidak menyediakan solusi yang optimal.



Gambar 2.1 Ilustrasi Pencarian Solusi Algoritma B&B

Seperti terlihat pada Gambar 2.1, bentuk hasil pencarian algoritma B&B menyerupai *tree*. Setiap solusi dapat direpresentasikan sebagai *node* pada *tree*. Solusi awal yang terbentuk merupakan *root* dari semua solusi yang ada.

2.4. Algoritma Branch and Bound pada Knapsack

Penerapan algoritma B&B pada permasalahan knapsack dapat dilakukan dengan beberapa tahapan berikut:

1. *Item* harus terurut secara *non-increasing* berdasarkan nilai rasio. Nilai rasio adalah perbandingan antara nilai *item* dengan berat *item*. Secara matematis, dapat dituliskan sebagai berikut:
   $v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_m/w_m$
2. Terdapat kumpulan solusi awal sebanyak k buah dengan nilai solusi nol, yang menandakan bahwa tidak ada *item* yang diambil pada kumpulan solusi tersebut ($S_t = \emptyset$).
3. Nilai suatu solusi (S_v) adalah jumlah nilai *item* yang diambil secara utuh (x_i bernilai 1).
4. Berat suatu solusi (S_w) adalah jumlah berat *item* yang diambil secara utuh (x_i bernilai 1).
5. S_t adalah kumpulan *item* yang diambil pada solusi.
6. Setiap solusi memiliki nilai *heuristic* (S_h). Nilai *heuristic* adalah perkiraan nilai maksimal suatu solusi dengan tidak memedulikan pengambilan diskrit. Artinya, suatu *item* dapat diambil meskipun tidak diambil secara utuh (x_i dapat bernilai pecahan antara 0 dan 1).
7. Nilai *heuristic* suatu solusi dapat dihitung dengan cara berikut:
   - $r =$ *item* terakhir yang diproses pada solusi tersebut
   - $L = W - S_w$

- $S_h = \sum_{j=r+1}^{m} \min\{((L - \sum_{i=r+1}^{j-1} w_i x_i)/ w_j), 1\} v_j$
- $S_h = S_v + S_h$

8. Solusi pertama (*root*) adalah dengan tidak mengambil *item* sama sekali tetapi tetap menghitung nilai *heuristic*. Solusi ini dimasukkan ke dalam kumpulan calon solusi.
9. Apabila masih ada calon solusi di kumpulan calon solusi, sebuah solusi dengan nilai *heuristic* terbesar diambil. Kemudian, pembangkitan solusi lain dilakukan dengan dua cara:
    - Mengambil *item* ke $r + 1$ dari solusi terambil.
    - Tidak mengambil *item* ke $r + 1$ dari solusi terambil.
10. Kedua cara tersebut akan menghasilkan dua solusi baru. Kedua solusi yang telah didapat dibandingkan dengan solusi ke-*k*. Apabila solusi baru memiliki nilai solusi yang lebih baik, maka solusi baru dimasukkan ke dalam kumpulan solusi, kemudian solusi ke-*k* yang lama dibuang dan kumpulan solusi terbaru diurutkan agar kumpulan solusi terurut dari solusi terbaik pertama hingga solusi terbaik ke-*k*.
11. Apabila solusi baru yang dibangkitkan memiliki nilai *heuristic* yang lebih baik daripada solusi ke-*k* terbaru, maka solusi tersebut dimasukkan ke dalam kumpulan calon solusi. Hal ini bertujuan agar solusi-solusi yang dibangkitkan selanjutnya memiliki nilai yang lebih baik daripada solusi sebelumnya.
12. Langkah 9-11 diulangi hingga tidak ada lagi solusi dalam kumpulan calon solusi.

Berikut akan ditunjukkan ilustrasi mengenai jalannya algoritma B&B pada pencarian *k* solusi terbaik pada permasalahan knapsack. Pada ilustrasi, nilai W adalah 15 dan k adalah 4. Tabel

2.2 menunjukkan susunan *item* pada Tabel 2.1 yang telah diurutkan secara *non-increasing* berdasarakan rasio *item*.

Tabel 2.2 Urutan *item* berdasarkan rasio

No	Nilai	Berat	Rasio
1	45	3	15
2	30	5	6
3	45	9	5
4	10	5	2

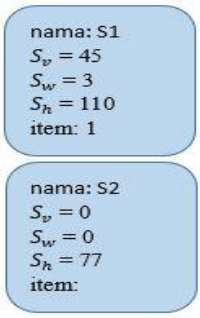
Kemudian, Gambar 2.2 mengilustrasikan pembangkitan solusi awal (*root*), beserta isi dari kumpulan solusi dan kumpulan calon solusi awal.

Gambar 2.2 Ilustrasi Solusi Awal

Setelah pembangkitan solusi awal, sesuai dengan langkah 9-11, maka *root* yang bertindak sebagai satu-satunya solusi di kumpulan calon solusi diambil dan digunakan sebagai dasar untuk membangkitkan S1 dan S2. Perlu diketahui bahwa nilai r pada *root* adalah *null*, sehingga *item* $r + 1$ adalah *item* pertama. S1 dibangkitkan dengan cara mengambil *item* pertama, sedangkan S2 dibangkitkan dengan cara tidak mengambil *item* pertama. Ilustrasi langkah ini dapat dilihat pada Gambar 2.3.

Lalu, S1 yang berada di urutan teratas kumpulan calon solusi diambil dan dilakukan hal yang sama, yaitu sesuai dengan langkah

9-11, untuk membangkitkan solusi S3 dan S4. Nilai r, baik pada S3 maupun S4 adalah 1 karena *item* terakhir yang diproses oleh kedua calon solusi tersebut adalah *item* pertama. Sehingga, nilai $r + 1$ pada S3 dan S4 adalah 2 atau dengan kata lain S3 dan S4 akan memproses *item* kedua. Ilustrasi langkah ini dapat dilihat pada Gambar 2.4.



(a)

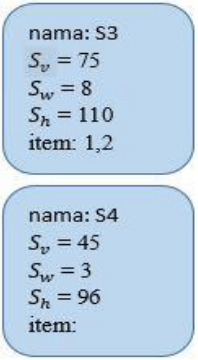
Kumpulan Solusi	
No	Nilai Solusi
1	45
2	0
3	0
4	0

(b)

No	Kumpulan Calon Solusi
1	S1
2	S2

(c)

Gambar 2.3 Ilustrasi Pembangkitan S1 dan S2



(a)

Kumpulan Solusi	
No	Nilai Solusi
1	75
2	45
3	0
4	0

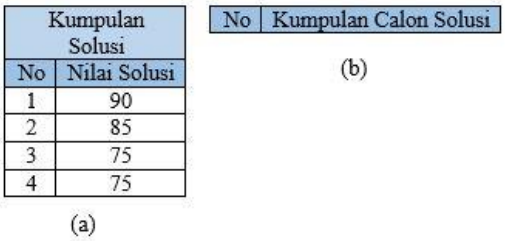
(b)

No	Kumpulan Calon Solusi
1	S3
2	S2
3	S4

(c)

Gambar 2.4 Ilustrasi Pembangkitan S3 dan S4

Selanjutnya, S3 diambil dari kumpulan calon solusi untuk membangkitkan calon solusi lain. Proses ini dilanjutkan hingga tidak ada lagi calon solusi pada kumpulan calon solusi. Gambar 2.5 dan Tabel 2.3 menunjukkan hasil akhir jalannya algoritma B&B pada pencarian *k* solusi terbaik pada permasalahan knapsack. Nomor *item* terambil yang digunakan pada Tabel 2.3 adalah nomor *item* setelah *item* diurutkan.



Kumpulan Solusi	
No	Nilai Solusi
1	90
2	85
3	75
4	75

(a)

No	Kumpulan Calon Solusi

(b)

Gambar 2.5 Hasil Akhir Pencarian Solusi Sebanyak k

Tabel 2.3 Hasil Akhir Pencarian k Solusi Terbaik

No.	Nilai	Berat	Item Terambil
1	90	12	1 dan 3
2	85	13	1, 2, dan 4
3	75	8	1 dan 2
4	75	14	2 dan 3

2.5. Dataset Pisinger

Dataset Pisinger adalah dataset khusus permasalahan knapsack. Dataset Pisinger dibangkitkan dengan menggunakan pembangkit Pisinger yang dapat diunduh di http://www.diku.dk/~pisinger/codes.hml. Dataset Pisinger memiliki delapan kelas data. Berikut adalah penjelasan dari delapan kelas data yang ada di dataset Pisinger.

1. *Uncorrelated instance*
   Nilai dan berat *item* diacak pada interval $[1, R]$.
2. *Weakly correlated instance*
   Berat *item* diacak pada interval $[1, R]$ dan nilai *item* diacak pada interval $[w_i - R/10, w_i + R/10]$.
3. *Strongly correlated instance*
   Berat *item* diacak pada interval $[1, R]$ dan nilai *item* ditetapkan pada nilai $w_i + R/10$.
4. *Inverse strongly correlated instance*
   Nilai *item* diacak pada interval $[1, R]$ dan berat *item* ditetapkan pada nilai $v_i + R/10$.
5. *Almost strongly correlated*
   Berat *item* diacak pada interval $[1, R]$ dan nilai *item* diacak pada interval $[w_i + R/10 - R/500, w_i + R/10 + R/500]$.
6. *Subset-sum*
   Berat *item* diacak pada interval $[1, R]$ dan nilai *item* ditetapkan pada nilai $v_i = w_i$.
7. *Even-odd subset sum*
   Berat *item* diacak pada interval $[1, R]$ dan bernilai genap, $v_i = w_i$, serta W bernilai ganjil.
8. *Even-odd strongly correlated*
   Berat *item* diacak pada interval $[1, R]$, $v_i = w_i + R/10$, serta W bernilai ganjil.

Pada pembangkit Pisinger, banyaknya *item*, kelas data yang dipilih, dan nilai R dapat diatur sendiri. Nilai R yang digunakan biasanya bernilai 1000 dan 10000. Pembangkit Pisinger akan membangkitkan dataset ke suatu *file*. Dataset ini nantinya digunakan untuk menguji kecepatan algoritma B&B. Tabel 2.3 menunjukkan contoh dataset untuk tiap kelas.

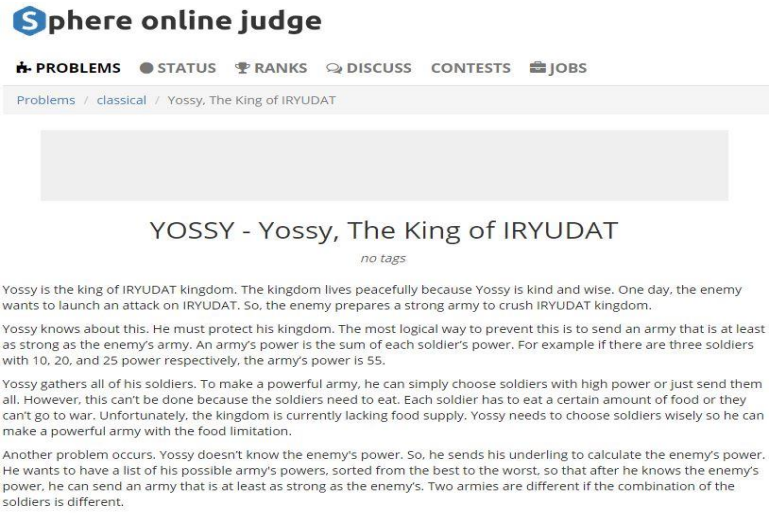
Tabel 2.4 Contoh Dataset untuk Setiap Kelas

Kelas	R=1000		R=10000	
	Nilai	Berat	Nilai	Berat
1	119	1	7331	2123
2	933	896	9191	8305
3	232	132	2022	1022
4	119	219	8730	9730
5	1101	1000	9468	8463
6	119	119	6119	6119
7	132	132	6132	6132
8	924	824	4554	3554

2.6. Permasalahan *Yossy, The King of IRYUDAT*

Pencarian k solusi terbaik pada permasalahan knapsack dapat juga dilihat di situs *online judge*, SPOJ. Di SPOJ, permasalahan yang berkaitan memiliki kode YOSSY, dengan judul permasalahan *Yossy, The King of IRYUDAT*. Gambar 2.6 menunjukkan tampilan permasalahan di situs SPOJ.

Pada permasalahan tersebut, terdapat beberapa orang prajurit dengan tingkat kekuatan dan jumlah kapasitas makanan yang harus dipenuhi. Prajurit-prajurit tersebut akan dikelompokkan sehingga membentuk pasukan. Pasukan memiliki tingkat kekuatan dan kapasitas makanan sesuai dengan penjumlahan dari kekuatan dan kapasitas makanan masing-masing prajurit. Untuk membentuk pasukan, harus diperhatikan kapasitas maksimal makanan karena terdapat batas maksimal kapasitas makanan yang diberikan. Tujuan dari permasalahan tersebut adalah menampilkan jumlah kekuatan k pasukan dari yang terbesar ke yang terkecil, dimana pasukan satu dengan yang lain dikatakan berbeda apabila terdapat setidaknya satu komposisi prajurit yang berbeda.



Gambar 2.6 Permasalahan Yossy, The King of IRYUDAT

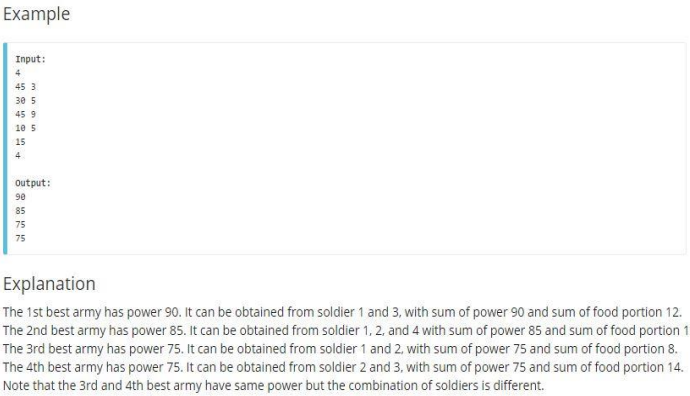
Format masukan dari permasalahan tersebut adalah sebagai berikut. Baris pertama adalah sebuah bilangan N yang menandakan jumlah prajurit. N baris berikutnya terdiri dari dua buah bilangan P dan F yang menandakan kekuatan dan kapasitas makanan masing-masing prajurit. Baris selanjutnya terdapat bilangan S yang menandakan kapasitas makanan maksimal. Baris terakhir terdapat K, yaitu jumlah kombinasi pasukan yang harus ditampilkan, terurut dari pasukan terbaik ke terburuk.

Format keluaran dari permasalahan tersebut adalah menampilkan K buah kekuatan pasukan dari yang terbaik ke terburuk. Gambar 2.7 menunjukkan contoh masukan dan keluaran.

Adapun batasan pada permasalahan tersebut, sebagai berikut:

1. $1 \leq N \leq 2000$.
2. $1 \leq P \leq 9999$.
3. $1 \leq F \leq 9999$.
4. $1 \leq S \leq 999999$.

5. $1 \leq K \leq 40$.
6. Lingkungan penilaian Intel Pentium G860 3 GHz.
7. Batasan waktu 0.1 - 0.15 detik.
8. Batasan memori 1536 MB.



Example

```
Input:
4
45 3
30 5
45 9
10 5
15
4

Output:
90
85
75
75
```

Explanation

The 1st best army has power 90. It can be obtained from soldier 1 and 3, with sum of power 90 and sum of food portion 12.
The 2nd best army has power 85. It can be obtained from soldier 1, 2, and 4 with sum of power 85 and sum of food portion 13.
The 3rd best army has power 75. It can be obtained from soldier 1 and 2, with sum of power 75 and sum of food portion 8.
The 4th best army has power 75. It can be obtained from soldier 2 and 3, with sum of power 75 and sum of food portion 14.
Note that the 3rd and 4th best army have same power but the combination of soldiers is different.

Gambar 2.7 Contoh Masukan dan Keluaran Permasalahan Yossy, The King of IRYUDAT

2.7. Metode Complete Search

Metode *complete search* merupakan suatu metode eksak yang dapat menentukan k solusi terbaik pada permasalahan kombinasi, termasuk permasalahan knapsack. Metode ini mencari semua kemungkinan kombinasi dari suatu permasalahan. Sehingga, wajar apabila metode ini selalu dapat dipastikan menemukan beberapa buah solusi pada permasalahan kombinasi. Namun, metode ini memiliki kelemahan, yaitu waktu eksekusi yang sangat lama. Untuk permasalahan knapsack, metode ini membutuhkan kompleksitas waktu (2^n) dimana n adalah banyak *item*. Untuk nilai $n \leq 20$, metode ini masih memungkinkan untuk menyelesaikan permasalahan. Untuk $n > 20$, metode ini sudah tidak dapat diandalkan lagi.

BAB III
DESAIN

Bab ini berisi desain algoritma B&B yang digunakan untuk menyelesaikan permasalahan pada tugas akhir ini.

3.1. Deskripsi Umum Sistem

Sistem akan menerima masukan berupa banyak *item* (m), nilai (v_i) dan berat (w_i) masing-masing *item*, kapasitas maksimal knapsack (W), dan nilai k, yaitu banyaknya solusi terbaik yang dicari. Kemudian, dilakukan *preprocess* untuk mengurutkan *item* secara *non-increasing* berdasarkan rasio. *Preprocess* juga digunakan sebagai pembangkit solusi awal atau *root*. Selanjutnya, proses pencarian k solusi terbaik dilakukan. Terakhir, k solusi terbaik yang telah ditemukan ditampilkan hasilnya. Gambar 3.1 menunjukkan *pseudocode* fungsi Main.

```
Main()
1.   input m
2.   for i=1 to m
3.      input v_i, w_i
4.   input W
5.   input k
6.   Preprocess()
7.   FindKthBestSolutions()
8.   output all solutions
```

Gambar 3.1 Pseudocode Fungsi Main

3.2. Desain Algoritma

Terdapat dua fungsi utama dalam sistem yang dikembangkan, yaitu Preprocess dan FindKthBestSolutions. Berikut adalah penjelasan masing-masing fungsi utama yang digunakan dalam sistem.

3.2.1. Desain Fungsi Preprocess

Fungsi Preprocess digunakan untuk mengurutkan *item* secara *non-increasing* berdasarkan rasio dan membangkitkan solusi awal atau *root*. *Pseudocode* fungsi Preprocess ditunjukkan pada Gambar 3.2.

```
Preprocess()
1.   Sort(items)
2.   for i=1 to number of solutions
3.      S_{i_v} = 0
4.      S_{i_w} = 0
5.      S_{i_h} = 0
6.      S_{i_t} = ∅
7.   h = 0
8.   u = 0
9.   for i=1 to m
10.     if h + w_i ≤ W
11.        h = h + w_i
12.        u = u + v_i
13.     else
14.        u = u + ((W-h)/ w_i)*v_i
15.        break
16.  R_h = u
17.  R_v = 0
18.  R_w = 0
19.  R_t = ∅
20.  P = ∅
21.  Enqueue(P,R)
```

Gambar 3.2 Desain Fungsi Preprocess

Baris 1 pada Gambar 3.2 digunakan untuk mengurutkan *item* secara *non-increasing*. Baris 2 hingga 6 digunakan untuk membuat solusi awal sebanyak k buah. Baris 7 melakukan inisialiasi awal berat solusi. Baris 8 melakukan inisiasi awal nilai *heuristic* solusi. Baris 9 sampai 15 melakukan perhitungan nilai *heuristic* solusi awal atau *root*. Pada baris 10 sampai 12, apabila berat solusi *root*

ditambah berat *item* ke-i tidak melebihi batas kapasitas knapsack, maka nilai dan berat *item* ke-i diambil seutuhnya. Sebaliknya, pada baris 13 sampai 15, apabila berat solusi *root* ditambah berat *item* ke-i melebihi batas kapasitas knapsack, nilai *item* ke-i diambil secara parsial dan berat *item* ke-i tidak dimasukkan perhitungan. Kemudian, baris 16 sampai 19 merupakan pembangkitan solusi *root*. Pada baris 19 dapat dilihat bahwa solusi *root* tidak memiliki himpunan *item* yang diambil karena pada solusi *root* fokus perhitungan hanya pada nilai *heuristic* saja. Selanjutnya, baris 20 sampai 21 adalah inisiasi *priority queue* dan memasukkan solusi *root* ke *priority queue*.

3.2.2. Desain Fungsi FindKthBestSolutions

Fungsi FindKthBestSolutions merupakan fungsi utama dalam sistem yang dikembangkan. Fungsi ini bertujuan untuk mencari k solusi terbaik yang diinginkan. *Pseudocode* fungsi FindKthBestSolutions ditunjukkan pada Gambar 3.3.

Selama *priority queue* masih memiliki calon solusi, sistem akan melakukan hal-hal berikut. Pada baris 2, calon solusi dengan nilai *heuristic* tertinggi akan diambil dari *priority queue*. Selanjutnya, pada baris 3, apabila nilai *heuristic* solusi yang diambil lebih baik daripada nilai solusi ke-k, perhitungan dilanjutkan. Baris 4 berguna untuk mengambil *item* terakhir yang diproses oleh solusi terambil. Baris 5 hingga 12 adalah proses pembangkitan solusi baru dengan cara mengambil *item* ke-$r + 1$. Baris 5 berfungsi untuk membuat solusi baru berdasarkan solusi terambil. Fungsi GenerateTakenItemSolution dapat dilihat di Gambar 3.4. Pada fungsi GenerateTakenItemSolution, solusi baru dibangkitkan dengan cara mengambil *item* ke $r + 1$. Selanjutnya, pada baris 6, apabila berat solusi yang baru dibangkitkan kurang dari atau sama dengan berat maksimal knapsack dan nilai solusi yang baru dibangkitkan lebih besar daripada solusi ke-k, maka

solusi tersebut dicek apakah sudah berada di kumpulan k solusi terbaik (baris 7). Fungsi IsInKthBestSolutions dapat dilihat di Gambar 3.6. Jika solusi tersebut belum berada di kumpulan k solusi terbaik, maka solusi tersebut dimasukkan ke kumpulan k solusi terbaik (baris 8). Pada baris 9, kumpulan k solusi terbaik diurutkan lagi secara *non-increasing* berdasarkan nilai solusi. Proses yang dilakukan di baris 8 mengakibatkan kumpulan solusi berjumlah $k + 1$. Oleh karena itu, pada baris 10, solusi terburuk akan dihapus dari kumpulan solusi, sehingga kumpulan solusi hanya memiliki k kumpulan solusi.

```
/*
 * P is priority queue made in Preprocess
 * S is all current k best solutions
 */
FindKthBestSolutions(P,S)
```

```
1.   while P ≠ ∅
2.      C = ExtractMax(P)
3.      if C_h > S_{k_v}
4.         r = last item taken in S_{k_t}
5.         T1 = GenerateTakenItemSolution(C,r+1)
6.         if T1_w ≤ W and T1_v > S_{k_v}
7.            if not IsInKthBestSolutions(S,T1)
8.               Insert(S,T1)
9.               Sort(S)
10.              ExtractMin(S)
11.        if T1_h > S_{k_v}
12.           Enqueue(P,T1)
13.        T2 = GenerateNonTakenItemSolution(C,r+1)
14.        if T2_w ≤ W and T2_v > S_{k_v}
15.           if not IsInKthBestSolutions(S,T2)
16.              Insert(S,T2)
17.              Sort(S)
18.              ExtractMin(S)
19.        if T2_h > S_{k_v}
20.           Enqueue(P,T2)
```

Gambar 3.3 Pseudocode Fungsi FindKthBestSolutions

Baris 13 hingga 20 adalah proses pembangkitan solusi dengan cara tidak mengambil *item* ke-$r + 1$. Proses ini sangat mirip dengan proses pembangkitan solusi dengan cara mengambil *item* ke-$r + 1$. Bedanya, pada proses ini bukan fungsi GenerateTakenItemSolution yang dijalankan, melainkan GenerateNonTakenItemSolution. Fungsi tersebut dapat dilihat pada Gambar 3.5.

```
/*
 * C is solution taken from priority queue
 * z is the index of item r+1
 */
GenerateTakenItemSolution(C,z)
```

```
1.   T_v = C_v + v_z
2.   T_w = C_w + w_z
3.   Insert(T_t,z)
4.   CalculateHeuristicValue(T,z)
5.   return T
```

Gambar 3.4 Pseudocode Fungsi GenerateTakenItemSolution

Gambar 3.4 menunjukkan isi dari fungsi GenerateTakenItemSolution. Baris 1 merupakan penghitungan nilai solusi yang didapat dari nilai solusi terambil ditambah dengan nilai *item* ke-z. Baris 2 merupakan penghitungan berat solusi yang didapat dari berat solusi terambil ditambah dengan berat *item* ke-z. Baris 3 memasukkan *item* ke-z sebagai *item* terakhir yang diproses pada solusi yang akan dibangkitkan. Baris 4 merupakan penghitungan nilai *heuristic* solusi baru. Fungsi CalculateHeuristicValue dapat dilihat pada Gambar 3.7. Lalu, pada baris 5, fungsi membangkitkan sebuah solusi baru dan mengembalikannya ke fungsi pemanggil.

Gambar 3.5 menunjukkan isi dari fungsi GenerateNonTakenItemSolution. Baris 1 merupakan penghitungan nilai solusi yang didapat langsung dari nilai solusi terambil. Baris 2 merupakan penghitungan berat solusi yang

didapat langsung dari berat solusi terambil. Baris 3 merupakan penghitungan nilai *heuristic* solusi baru. Lalu, pada baris 4, fungsi membangkitkan sebuah solusi baru dan mengembalikannya ke fungsi pemanggil.

```
/*
 * C is solution taken from priority queue
 * z is the index of item r+1
 */
GenerateNonTakenItemSolution(C,z)
1.   T_v = C_v
2.   T_w = C_w
3.   CalculateHeuristicValue(T,z)
4.   return T
```

Gambar 3.5 Pseudocode Fungsi GenerateNonTakenItemSolution

```
/*
 * S is all current k best solutions
 * T is candidate solution
 */
IsInKthBestSolution(S,T)
1.   for i=1 to k
2.      if ∀T_t == ∀S_{i_t}
3.         return true
4.   return false
```

Gambar 3.6 Pseudocode Fungsi IsInKthBestSolutions

Gambar 3.6 menunjukkan isi fungsi IsInKthBestSolutions. Baris 1 berguna untuk mengecek keseluruhan solusi. Pada baris 2, apabila solusi ke-i memiliki kombinasi *item* yang sama dengan solusi yang akan dibangkitkan, maka solusi yang akan dibangkitkan telah ada di kumpulan k solusi terbaik. Oleh karena itu, sistem akan mengembalikan nilai *false* (baris 3). Sebaliknya, apabila solusi yang akan dibangkitkan belum ada di kumpulan k solusi terbaik, maka sistem mengembalikan nilai *true* (baris 4).

```
/*
 * T is candidate solution
 * z is the index of item r+1
 */
CalculateHeuristicValue(T,z)
```

```
1.   h = T_w
2.   if T_w > W
3.      T_h = 0
4.   else
5.      T_h = T_v
6.      for i=z+1 to m
7.         if h + w_i ≤ W
8.            h = h + w_i
9.            T_h = T_h + v_i
10.        else
11.           T_h = T_h + ((W-h)/ w_i)*v_i
12.           break
```

Gambar 3.7 Pseudocode Fungsi CalculateHeuristicValue

Gambar 3.7 menunjukkan isi fungsi CalculateHeuristicValue. Baris 1 digunakan untuk menyimpan berat solusi saat ini. Apabila berat solusi sudah melebihi kapasitas knapsack, perhitungan tidak perlu dilanjutkan (baris 2). Baris 4 memastikan bahwa solusi masih memiliki sisa berat untuk menampung *item*-*item* selanjutnya. Baris 5 menginisasi nilai *heuristic* solusi dengan nilai solusi. Baris 6 melakukan proses perulangan pengambilan *item* dari *item* ke-$z+1$ hingga *item* terakhir. Apabila berat saat ini ditambah berat *item* ke-i masih mencukupi, maka berat saat ini diperbarui dan nilai *item* ke-i ditambahkan ke nilai *heuristic* (baris 8 dan 9). Namun, apabila berat saat ini ditambah berat *item* ke-i tidak mencukupi, maka *item* ke-i diambil secara parsial dan nilainya ditambahkan ke nilai *heuristic* selaras dengan persentase berat yang tersisa dibanding kapasitas total knapsack (baris 11), lalu proses berhenti (baris 12).

3.3. Desain Struktur Data

Dalam sistem yang dikembangkan, untuk menyimpan *item* dan mengaksesnya secara langsung, diperlukan struktur data yang dapat melakukan *insert* dan *random access*. Struktur data seperti ini dapat dipenuhi oleh *list*. *List* mendukung penambahan elemen dan pengaksesan elemen pada posisi tertentu secara langsung. Struktur data ini juga dapat digunakan untuk menyimpan kumpulan solusi.

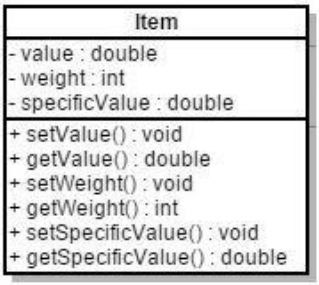
Selain *list*, sistem juga memerlukan suatu struktur data yang mampu secara otomatis mengurutkan elemen di dalamnya sesuai dengan prioritas elemen. Contoh penggunaan struktur data ini ada pada penyimpanan dan pengaksesan kumpulan calon solusi. Kumpulan calon solusi yang disimpan harus terurut dari solusi yang memiliki nilai *heuristic* terbaik hingga terburuk. Struktur data seperti ini dapat dipenuhi oleh *priority queue*.

3.4. Desain Kelas

Ada dua kelas yang dibuat dalam sistem. Pertama adalah kelas Item dan kedua adalah kelas Solution. Berikut adalah penjelasan masing-masing kelas yang dibuat.

3.4.1. Desain Kelas Item

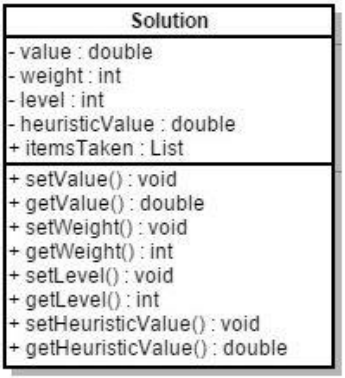
Kelas Item adalah suatu kelas yang digunakan untuk menyimpan *item*. Pada kelas Item, terdapat tiga atribut, yaitu *value*, *weight*, dan *specificValue*. Masing-masing atribut berfungsi untuk menyimpan nilai, berat, dan rasio *item*. Untuk mengakses atribut tersebut, masing-masing atribut memiliki *setter* dan *getter* sendiri. Gambar 3.8 menunjukkan desain kelas Item.



Gambar 3.8 Desain Kelas Item

3.4.2. Desain Kelas Solution

Kelas Solution adalah suatu kelas yang digunakan untuk menyimpan solusi yang telah dibangkitkan. Pada kelas Solution, terdapat lima atribut, yaitu *value, weight, level, heuristicValue,* dan *itemsTaken.* Masing-masing atribut berfungsi untuk menyimpan nilai, berat, indeks *item* terakhir yang diproses, nilai *heuristic*, dan kumpulan *item* yang telah diproses. Masing-masing atribut juga memiliki *setter* dan *getter*. Gambar 3.9 menunjukkan desain kelas Solution.



Gambar 3.9 Desain Kelas Solution

3.4. Desain Metode Complete Search

Metode *complete search* digunakan untuk membandingkan kebenaran metode yang diusulkan. Gambar 3.10 dan 3.11 menunjukkan *pseudocode* metode *complete search*.

```
Main()
1.   input m
2.   for i=1 to m
3.      input v_i, w_i
4.   input W
5.   input k
6.   S = ∅
7.   CompleteSearch(1)
8.   Sort(S)
9.   output all solutions
```

Gambar 3.10 Desain Fungsi Main pada Complete Search

```
/*
 * idx is the index of the processed item
 */
CompleteSearch(idx)
1.   if idx == m+1
2.      tv = 0
3.      tw = 0
4.      for i=1 to m
5.         tv += v_i * state_i
6.         tw += w_i * state_i
7.      if tw ≤ W
8.         Insert(S,tv)
9.      return
10.  for i=0 to 1
11.     state_idx = i
12.     CompleteSearch(idx+1)
```

Gambar 3.11 Desain Fungsi CompleteSearch

Gambar 3.10 menunjukkan *pseudocode* fungsi Main pada metode *complete search*. Sistem akan menerima masukan berupa

banyak *item* (m), nilai (v_i) dan berat (w_i) masing-masing *item*, kapasitas maksimal knapsack (W), dan nilai k, yaitu banyaknya solusi terbaik yang dicari. Kemudian, dilakukan inisiasi kumpulan solusi awal (S). Selanjutnya, fungsi CompleteSearch dipanggil untuk mencari semua kombinasi. Setelah itu, kumpulan solusi diurutkan dari nilai tertinggi ke nilai terendah dan hasilnya ditampilkan.

Gambar 3.11 menunjukkan *pseudocode* fungsi CompleteSearch. Pada baris 1, sistem mencari tahu apakah indeks sudah melewati jumlah *item*. Baris 2 dan 3 merupakan inisiasi awal nilai dan berat solusi. Pada baris 4-6, sistem melakukan penghitungan nilai dan berat solusi. Pada baris 7-8, apabila berat solusi tidak melebihi kapasitas maksimal, maka nilai solusi dimasukkan ke kumpulan solusi. Baris 10-12 merupakan pemanggilan kembali fungsi CompleteSearch untuk menemukan kombinasi lainnya.

BAB IV
IMPLEMENTASI

Bab ini berisi implementasi dari desain algoritma, struktur data, dan kelas yang digunakan untuk menyelesaikan permasalahan pada tugas akhir ini.

4.1. Lingkungan Implementasi

Lingkungan implementasi dalam pengerjaan tugas akhir ini terdiri dari perangkat keras dan perangkat lunak. Berikut adalah detil lingkungan implementasi yang digunakan.

1. Perangkat Keras
   Processor Intel® Core™ i5-2430M CPU @ 2.40GHz
   RAM 4.00 GB
   64-bit Operating System, x-64-based processor
2. Perangkat Lunak
   Sistem operasi Windows 8.1
   Integrated Development Environment Code::Blocks 13.12

4.2. Implementasi Kelas Item

Kelas Item diimplementasikan sesuai dengan desain yang ditunjukkan pada Gambar 3.8. Kode Sumber 4.1 dan Kode Sumber 4.2 menunjukkan implementasi kelas Item. Pada kelas Item, terdapat juga dua macam *constructor* yang digunakan untuk menginisiasi *item*.

```
1.   class Item{
2.   private:
3.      double value;
4.      int weight;
5.      double specificValue;
```

Kode Sumber 4.1 Implementasi Kelas Item (1)

```
6.   public:
7.      Item(){}
8.      Item(double _value, int _weight, double
     _specificValue){
9.         value = _value;
10.        weight = _weight;
11.        specificValue = _specificValue;
12.     }
13.     void setValue(double _value){
14.        value = _value;
15.     }
16.     double getValue(){
17.        return value;
18.     }
19.     void setWeight(int _weight){
20.        weight = _weight;
21.     }
22.     int getWeight(){
23.        return weight;
24.     }
25.     void setSpecificValue(double
     _specificValue){
26.        specificValue = _specificValue;
27.     }
28.     double getSpecificValue(){
29.        return specificValue;
30.     }
31.  };
```

Kode Sumber 4.2 Implementasi Kelas Item (2)

4.3. Implementasi Kelas Solution

Kelas Solution diimplementasikan sesuai dengan desain yang ditunjukkan pada Gambar 3.9. Kode Sumber 4.3 dan Kode Sumber 4.4 menunjukkan implementasi kelas Solution. Pada kelas Solution, terdapat tiga macam *constructor* yang digunakan untuk menginisiasi solusi.

```
1.   class Solution{
2.   private:
3.      double value;
4.      int weight;
5.      int level;
6.      double heuristicValue;
7.   public:
8.      vector<int> itemsTaken;
9.      Solution(){
10.        value = 0;
11.        weight = 0;
12.        level = -1;
13.        heuristicValue = 0;
14.        itemsTaken = vector<int>();
15.     }
16.     Solution(double _value, int _weight, int
     _level, double _heuristicValue){
17.        value = _value;
18.        weight = _weight;
19.        level = _level;
20.        heuristicValue = _heuristicValue;
21.        itemsTaken = vector<int>();
22.     }
23.     Solution(double _value, int _weight, int
     _level, double _heuristicValue, vector<int>
     _itemsTaken){
24.        value = _value;
25.        weight = _weight;
26.        level = _level;
27.        heuristicValue = _heuristicValue;
28.        itemsTaken = _itemsTaken;
29.     }
30.     void setValue(double _value){
31.        value = _value;
32.     }
```

Kode Sumber 4.3 Implementasi Kelas Solution (1)

```
33.     double getValue(){
34.        return value;
35.     }
36.     void setWeight(int _weight){
37.        weight = _weight;
38.     }
39.     int getWeight(){
40.        return weight;
41.     }
42.     void setLevel(int _level){
43.        level = _level;
44.     }
45.     int getLevel(){
46.        return level;
47.     }
48.     void setHeuristicValue(double
     _heuristicValue){
49.        heuristicValue = _heuristicValue;
50.     }
51.     double getHeuristicValue(){
52.        return heuristicValue;
53.     }
54.  };
```

Kode Sumber 4.4 Implementasi Kelas Solution (2)

4.4. Implementasi Fungsi Main

Fungsi Main merupakan fungsi utama yang ada pada sistem. Fungsi ini merupakan fungsi yang pertama kali dipanggil. Semua variabel utama akan dideklarasikan sebelum fungsi Main. Hal ini dilakukan agar variabel bersifat global, sehingga tidak perlu melakukan pengiriman variabel ke fungsi lain. Implementasi fungsi Main dapat dilihat di Kode Sumber 4.5. Baris 1 hingga 7 merupakan deklarasi variabel yang digunakan secara global dalam sistem. Baris 9 hingga 25 merupakan isi dari fungsi Main. Pada baris 7, terdapat sebuah kelas baru yang bernama CompareSolution

yang digunakan sebagai komparasi solusi pada *priority queue*. Kelas ini dapat dilihat pada Kode Sumber 4.6.

```
1.   Item items[MAX_ITEM];
2.   int maxItem;
3.   int knapsackWeight;
4.   int numberOfSolution;
5.   Solution *initSolution, *kthSolution;
6.   vector<Solution*> solutionNodes;
7.   priority_queue<Solution*, vector<Solution*>,
     CompareSolution> solutions;
8.
9.   int main(){
10.     int weight, level;
11.     double value;
12.     scanf("%d", &maxItem);
13.     for(int i=0; i<maxItem; i++){
14.        scanf("%lf %d", &value, &weight);
15.        items[i] = Item(value, weight,
     value/weight);
16.     }
17.     scanf("%d", &knapsackWeight);
18.     scanf("%d", &numberOfSolution);
19.     Preprocess();
20.     FindKthBestSolutions();
21.     for(int i=0; i<numberOfSolution; i++){
22.        printf("%.0lf\n",solutionNodes[i]-
     >getValue());
23.     }
24.     return 0;
25.  }
```

Kode Sumber 4.5 Implementasi Fungsi Main

Kelas CompareSolution pada fungsi Main digunakan untuk membandingkan solusi yang ada pada *priority queue*. Kelas ini harus dibuat agar solusi yang ada pada *priority queue* dapat terurut secara *non-increasing* berdasarkan nilai *heuristic* solusi.

```
1.   class CompareSolution{
2.   public:
3.      bool operator()(Solution* firstSolution,
     Solution* secondSolution){
4.         return (firstSolution-
     >getHeuristicValue() < secondSolution-
     >getHeuristicValue())
5.      }
6.   };
```

Kode Sumber 4.6 Implementasi Kelas CompareSolution

4.5. Implementasi Fungsi Preprocess

Kode Sumber 4.7 menunjukkan implementasi fungsi Preprocess seperti pada Gambar 3.2. Pada fungsi Preprocess, terdapat fungsi lain yang dipanggil, yaitu compareItem, createEmptyNode dan createRoot Node. Fungsi compareItem digunakan untuk membandingkan *item* agar terurut secara *non-increasing* berdasarkan rasio. Fungsi compareItem dapat dilihat pada Kode Sumber 4.8. Fungsi createEmptyNode berguna untuk membuat sebuah solusi kosong. Fungsi createEmptyNode dapat dilihat pada Kode Sumber 4.9. Fungsi createRootNode berguna untuk membangkitkan solusi awal atau *root*. Fungsi createRootNode dapat dilihat pada Kode Sumber 4.10.

```
1.   void Preprocess(){
2.      sort(items, items+maxItem, compareItem);
3.      for(int i=0; i<numberOfSolution; i++){
4.
     solutionNodes.push_back(createEmptyNode());
5.      }
6.      kthSolution = createEmptyNode();
7.      initSolution = createRootNode();
8.      solutions.push(initSolution);
9.   }
```

Kode Sumber 4.7 Implementasi Fungsi Preprocess

```
1.  bool compareItem(Item firstItem, Item
    secondItem){
2.     return (firstItem.getSpecificValue() >
    secondItem.getSpecificValue());
3.  }
```

Kode Sumber 4.8 Implementasi Fungsi CompareItem

```
1.  Solution *createEmptyNode(){
2.     solution *emptyNode = new Solution();
3.     return emptyNode;
4.  }
```

Kode Sumber 4.9 Implementasi Fungsi CreateEmptyNode

```
1.  Solution *createRootNode(){
2.     int nodeWeight = 0;
3.     double upperBound = 0.0;
4.     for(int i=0; i<maxItem; i++){
5.        if(nodeWeight + items[i].getWeight() <=
    knapsackWeight){
6.           upperBound += items[i].getValue();
7.           nodeWeight += items[i].getWeight();
8.        }
9.        else{
10.          upperBound += ((double)
    (knapsackWeight-nodeWeight) /
    items[i].getWeight()) * items[i].getValue();
11.          break;
12.       }
13.    }
14.    return new Solution(0,0,-1,upperBound);
15. }
```

Kode Sumber 4.10 Implementasi Fungsi CreateRootNode

4.6. Implementasi Fungsi FindKthBestSolutions

Fungsi FindKthBestSolutions merupakan fungsi utama pencarian *k* solusi terbaik pada permasalahan knapsack. Fungsi ini dapat dilihat pada Kode Sumber 4.11, Kode Sumber 4.12, dan

Kode Sumber 4.13. Ada dua proses utama dalam fungsi ini, yaitu pembangkitan solusi dengan cara mengambil *item* ke-$r + 1$ (baris 7 hingga 21) dan dengan tidak mengambil *item* ke-$r + 1$ (baris 22 hingga 36). Solusi yang terbentuk akan dibandingkan. Jika, solusi yang dibangkitkan memiliki nilai solusi yang lebih baik daripada nilai solusi ke-k, maka solusi tersebut dimasukkan ke dalam kumpulan solusi (baris 11 dan 26). Kemudian, kumpulan solusi diurutkan kembali secara *non-increasing* (baris 12 dan 27). Terakhir, solusi ke-$k + 1$ dibuang (baris 13 dan 28).

```
1.   void FindKthBestSolutions(){
2.      while(!solutions.empty()){
3.         Solution *currentSolution =
     solutions.top();
4.         solutions.pop();
5.         if(currentSolution->getHeuristicValue()
     > kthSolution->getValue()){
6.            level = currentSolution-
     >getLevel()+1;
7.            if(level < maxItem){
8.               Solution *candidateSolution =
     GenerateTakenItemSolution();
9.               if(candidateSolution->getWeight()
     <= knapsackWeight && candidateSolution-
     >getValue() > kthSolution->getValue()){
10.
     if(!IsInKthBestSolutions(candidateSolution)){
11.                  solutionNodes.push_back(new
     Solution(candidateSolution->getValue(),
     candidateSolution->getWeight(),
     candidateSolution->getLevel(),
     candidateSolution->getHeuristicValue(),
     canditateSolution->itemsTaken));
12.                  sort(solutionNodes.begin(),
     solutionNodes.end(), compareKthSolution);
```

Kode Sumber 4.11 Implementasi Fungsi FindKthBestSolutions (1)

```
13.                     solutionNodes.pop_back();
14.                     delete(kthSolution);
15.                     kthSolution = new
    Solution(solutionNodes[numberOfSolution-1]-
    >getValue(), solutionNodes[numberOfSolution-
    1]->getWeight(),
    solutionNodes[numberOfSolution-1]->getLevel(),
    solutionNodes[numberOfSolution-1]-
    >getHeuristicValue());
16.                 }
17.             }
18.             if(candidateSolution-
    >getHeuristicValue() > kthSolution-
    >getValue()){
19.
    solutions.push(candidateSolution);
20.             }
21.          }
22.          if(level < maxItem){
23.             Solution *candidateSolution =
    GenerateNonTakenItemSolution();
24.             if(candidateSolution->getWeight()
    <= knapsackWeight && candidateSolution-
    >getValue() > kthSolution->getValue()){
25.
    if(!IsInKthBestSolutions(candidateSolution)){
26.                     solutionNodes.push_back(new
    Solution(candidateSolution->getValue(),
    candidateSolution->getWeight(),
    candidateSolution->getLevel(),
    candidateSolution->getHeuristicValue(),
    candidateSolution->itemsTaken));
27.                     sort(solutionNodes.begin(),
    solutionNodes.end(), compareKthSolution);
28.                     solutionNodes.pop_back();
29.                     delete(kthSolution);
```

Kode Sumber 4.12 Implementasi Fungsi FindKthBestSolutions (2)

```
30.                     kthSolution = new
     Solution(solutionNodes[numberOfSolution-1]-
     >getValue(), solutionNodes[numberOfSolution-
     1]->getWeight(),
     solutionNodes[numberOfSolution-1]->getLevel(),
     solutionNodes[numberOfSolution-1]-
     >getHeuristicValue());
31.                 }
32.             }
33.             if(candidateSolution-
     >getHeuristicValue() > kthSolution-
     >getValue()){
34.
     solutions.push(candidateSolution);
35.             }
36.           }
37.         }
38.     }
39. }
```

Kode Sumber 4.13 Implementasi Fungsi FindKthBestSolutions (3)

4.7. Implementasi Fungsi GenerateTakenItemSolution

Fungsi GenerateTakenItemSolution diimplementasikan sesuai dengan *pseudocode* yang ada pada Gambar 3.4. Kode Sumber 4.14 dan 4.15 merupakan hasil implementasi fungsi GenerateTakenItemSolution.

```
1.   Solution *generateTakenItemSolution(Solution
     *currentSolution, int level){
2.      Solution *candidateSolution = new
     Solution();
3.      candidateSolution->itemsTaken =
     currentSolution->itemsTaken;
```

Kode Sumber 4.14 Implementasi Fungsi GenerateTakenItemSolution (1)

```
4.      candidateSolution-
     >setValue(currentSolution->getValue() +
     items[level].getValue());
5.      candidateSolution-
     >setWeight(currentSolution->getWeight() +
     items[level].getWeight());
6.      candidateSolution->setLevel(level);
7.      candidateSolution-
     >itemsTaken.push_back(level);
8.      calculateHeuristicValue(candidateSolution);
9.      return candidateSolution;
10.  }
```

Kode Sumber 4.15 Implementasi Fungsi GenerateTakenItemSolution (2)

4.8. Implementasi Fungsi GenerateNonTakenItemSolution

Fungsi GenerateNonTakenItemSolution diimplementasikan sesuai dengan *pseudocode* yang ada pada Gambar 3.5. Kode Sumber 4.16 merupakan hasil implementasi fungsi GenerateNonTakenItemSolution.

```
1.   Solution *generateNonTakenItemSolution(
     Solution *currentSolution, int level){
2.      Solution *candidateSolution = new
     Solution();
3.      candidateSolution-
     >setValue(currentSolution->getValue() +
     items[level].getValue());
4.      candidateSolution-
     >setWeight(currentSolution->getWeight() +
     items[level].getWeight());
5.      candidateSolution->setLevel(level);
6.      calculateHeuristicValue(candidateSolution);
7.      return candidateSolution;
8.   }
```

Kode Sumber 4.16 Implementasi Fungsi GenerateNonTakenItemSolution

4.9. Implementasi Fungsi IsInKthBestSolutions

Fungsi IsInKthBestSolutions diimplementasikan sesuai dengan *pseudocode* yang ada pada Gambar 3.6. Kode Sumber 4.17 merupakan hasil implementasi fungsi IsInKthBestSolutions. Fungsi ini mengharuskan suatu solusi untuk dibandingkan dengan semua solusi pada kumpulan *k* solusi terbaik (baris 2). Apabila kombinasi *item* pada suatu solusi sama persis dengan kombinasi *item* pada salah satu solusi pada kumpulan solusi (baris 3 hingga 10), maka solusi tersebut bukanlah solusi baru (baris 11). Sebaliknya, apabila solusi tersebut belum ada di kumpulan solusi terbaik, maka solusi tersebut adalah suatu solusi baru dan perlu diproses lebih lanjut oleh fungsi pemanggil (baris 15).

```
1.   bool isInKBestSolution(Solution *checkedNode){
2.      for(int i=0; i<numberOfSolution; i++){
3.         if(checkedNode->itemsTaken.size() ==
     solutionNodes[i]->itemsTaken.size()){
4.            int counter = 0;
5.            for(int j=0; j<(int)checkedNode-
     >itemsTaken.size(); j++){
6.               if(checkedNode->itemsTaken[j] ==
     solutionNodes[i]->itemsTaken[j]){
7.                  counter++;
8.               }
9.            }
10.           if(counter == checkedNode-
     >itemsTaken.size()){
11.              return true;
12.           }
13.        }
14.     }
15.     return false;
16.  }
```

Kode Sumber 4.17 Implementasi Fungsi IsInKthBestSolutions

4.10. Implementasi Fungsi CalculateHeuristicValue

Fungsi CalculateHeuristicValue diimplementasikan sesuai dengan *pseudocode* yang ada pada Gambar 3.7. Kode Sumber 4.18 merupakan hasil implementasi fungsi CalculateHeuristicValue.

```
1.   void calculateHeuristicValue(Solution
     *solution){
2.      int nodeWeight = solution->getWeight();
3.      int startingNode = solution->getLevel()+1;
4.      if(solution->getWeight() > knapsackWeight){
5.         solution->setHeuristicValue(0.0);
6.      }
7.      else{
8.         solution->setHeuristicValue(solution-
     >getValue());
9.         for(int i=startingNode; i<maxItem; i++){
10.           if(nodeWeight + items[i].getWeight()
     <= knapsackWeight){
11.              solution-
     >setHeuristicValue(solution-
     >getHeuristicValue()+items[i].getValue());
12.              nodeWeight +=
     items[i].getWeight();
13.           }
14.           else{
15.              solution->setHeuristicValue(
     solution->getHeuristicValue() + (((double)(
     knapsackWeight-nodeWeight) /
     items[i].getWeight())*items[i].getValue()));
16.              break;
17.           }
18.        }
19.     }
20.  }
```

Kode Sumber 4.18 Implementasi Fungsi CalculateHeuristicValue

4.11. Implementasi Metode Complete Search

Fungsi Main dan CompleteSearch pada metode *complete search* diimplementasikan sesuai dengan *pseudocode* yang ada pada Gambar 3.10 dan 3.11. Kode Sumber 4.19 dan 4.20 merupakan hasil implementasi fungsi Main dan CompleteSearch.

Pada Kode Sumber 4.19, baris 17, fungsi *sort* pada C++ mengurutkan variabel dari nilai terendah ke tertinggi. Oleh karena itu, pada baris 19, pemanggilan hasil dilakukan dari indeks terakhir agar hasil terurut dari nilai tertinggi ke terendah.

```
1.   int maxItem;
2.   int knapsackWeight;
3.   int numberOfSolution;
4.   int state[MAX_ITEM];
5.   int value[MAX_ITEM];
6.   int weight[MAX_ITEM];
7.   vector<double> solutions;
8.
9.   int main(){
10.     scanf("%d", &maxItem);
11.     for(int i=0; i<maxItem; i++){
12.        scanf("%lf %d", &value[i], &weight[i]);
13.     }
14.     scanf("%d", &knapsackWeight);
15.     scanf("%d", &numberOfSolution);
16.     CompleteSearch(0);
17.     sort(solutions.begin(),solutions.end());
18.     for(int i=0; i<numberOfSolution; i++){
19.        printf("%.0lf\n",
     solutions[solutions.size()-i-1]);
20.     }
21.     return 0;
22.  }
```

Kode Sumber 4.19 Implementasi Fungsi Main pada Complete Search

```
1.   void CompleteSearch(int idx){
2.      if(idx == maxItem){
3.         double tempValue = 0;
4.         int tempWeight = 0;
5.         for(int i=0; i<maxItem; i++){
6.            tempValue += value[i]*state[i];
7.            tempWeight += weight[i]*state[i];
8.         }
9.         if(tempWeight <= knapsackWeight){
10.           solutions.push_back(tempValue);
11.        }
12.        return;
13.     }
14.     for(int i=0; i<=1; i++){
15.        state[idx] = i;
16.        CompleteSearch(idx+1);
17.     }
18.  }
```

Kode Sumber 4.20 Implementasi Fungsi CompleteSearch

BAB V
UJI COBA DAN EVALUASI

Bab ini berisi hasil uji coba dan evaluasi terhadap implementasi algoritma, struktur data, dan kelas yang digunakan untuk menyelesaikan permasalahan pada tugas akhir ini.

5.1. Lingkungan Uji Coba

Lingkungan uji coba dalam pengerjaan tugas akhir ini terdiri dari perangkat keras dan perangkat lunak. Berikut adalah detil lingkungan uji coba yang digunakan.

1. Perangkat Keras
   Processor Intel® Core™ i5-2430M CPU @ 2.40GHz
   RAM 4.00 GB
   64-bit Operating System, x-64-based processor
2. Perangkat Lunak
   Sistem operasi Windows 8.1
   Integrated Development Environment Code::Blocks 13.12

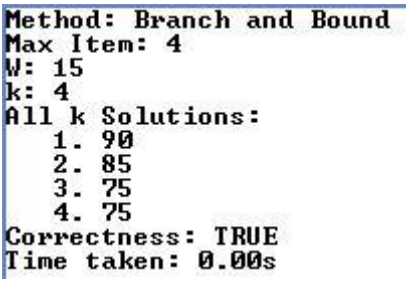
5.2. Skenario Uji Coba

Bagian ini akan menjelaskan mengenai skenario uji coba yang telah dilakukan. Rangkaian uji coba dibagi menjadi dua bagian, yaitu uji coba kebenaran dan uji coba kinerja. Uji coba kebenaran dilakukan untuk menguji kebenaran hasil implementasi. Uji coba kebenaran akan dibandingkan dengan program lain yang serupa. Sedangkan uji coba kinerja dilakukan dengan menggunakan dataset Pisinger.

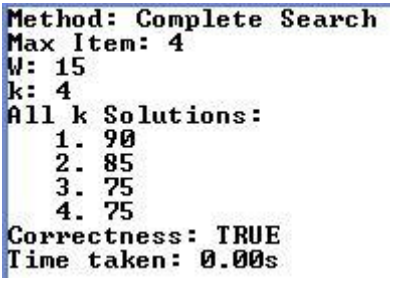
5.2.1. Uji Coba Kebenaran

Uji coba kebenaran dilakukan dengan membandingkan hasil implementasi dengan hasil program lain yang dijamin kebenarannya, yaitu metode *complete search*. Gambar 5.1

menunjukkan salah satu hasil uji coba dengan menggunakan metode B&B dan Gambar 5.2 menunjukkan salah satu hasil uji coba dengan menggunakan metode *complete search*. Kedua gambar hasil uji coba menggunakan data dengan 4 *item*. Nilai dan berat *item* sama seperti pada Tabel 2.1. Nilai *W* sebesar 15 dan nilai *k* sebesar 4. Dapat dilihat pada kedua gambar bahwa hasil yang diberikan sama persis.



Gambar 5.1 Hasil Uji Coba Kebenaran Metode B&B



Gambar 5.2 Hasil Uji Coba Kebenaran Metode Complete Search

Tabel 5.1 menunjukkan perbandingan hasil metode B&B dengan *complete search*. Terdapat sepuluh data uji coba yang digunakan. Masing-masing data memiliki jumlah *item* dan nilai *k* yang berbeda.

Tabel 5.1 Hasil Uji Coba Kebenaran Metode B&B dan Complete Search

No	Jumlah Item	Nilai k	Hasil	
			B&B	Complete Search
1	4	4	True	True
2	5	10	True	True
3	6	10	True	True
4	7	15	True	True
5	8	15	True	True
6	9	20	True	True
7	10	20	True	True
8	15	20	True	True
9	20	40	True	True
10	25	40	True	True

5.2.2. Uji Coba Kinerja

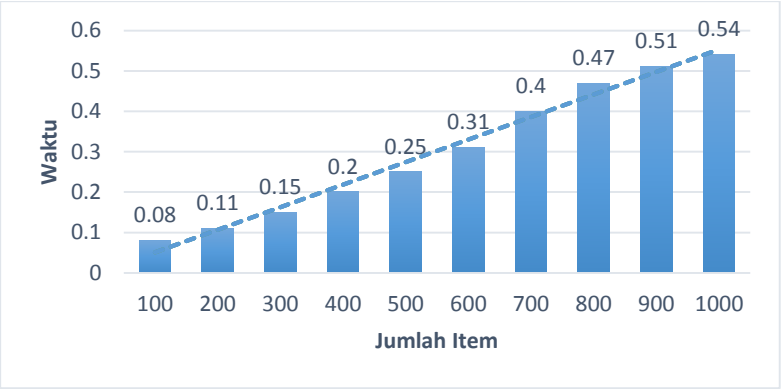
Uji coba kinerja akan dibagi menjadi tiga. Uji coba pertama dilakukan dengan membangkitkan *item* secara acak dengan nilai m yang berbeda. Nilai k akan dibuat tetap untuk semua nilai m. Uji coba kedua dilakukan dengan membangkitkan *item* secara acak dengan nilai k yang berbeda. Untuk setiap nilai k, nilai m akan dibuat sama. Uji coba ketiga dilakukan dengan membangkitkan *item* dengan menggunakan dataset Pisinger. Setiap kelas dataset diuji untuk mengetahui pengaruh perbedaan kelas data terhadap waktu. Nilai m dan k akan dibuat sama pada setiap data yang diuji.

5.2.2.1. Pengaruh Banyak Item Terhadap Waktu

Kasus uji yang diujikan sebanyak sepuluh kasus. Setiap kasus uji memiliki jumlah *item* bervariasi dari 100 hingga 1000. Setiap kasus uji memiliki nilai k tetap, yaitu 40. Hasil uji coba ditunjukkan pada Tabel 5.2 dan Gambar 5.3

Tabel 5.2 Hasil Uji Coba Pengaruh Banyak Item Terhadap Waktu

No	Jumlah Item	Waktu (detik)
1	100	0.08
2	200	0.11
3	300	0.15
4	400	0.2
5	500	0.25
6	600	0.31
7	700	0.4
8	800	0.47
9	900	0.51
10	1000	0.54



Gambar 5.3 Grafik Hasil Uji Coba Pengaruh Banyak Item Terhadap Waktu

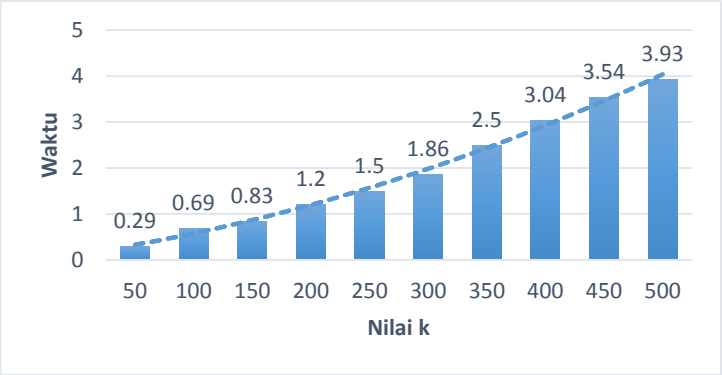
Dari hasil uji coba yang telah dilakukan, dapat dilihat bahwa pertumbuhan waktu yang dibutuhkan program mendekati kurva linier seiring dengan pertumbuhan jumlah item.

5.2.2.2. Pengaruh Nilai K Terhadap Waktu

Kasus uji yang diujikan sebanyak sepuluh kasus. Setiap kasus uji memiliki nilai *k* bervariasi dari 50 hingga 500. Setiap kasus uji memiliki jumlah *item* tetap, yaitu 500. Hasil uji coba ditunjukkan pada Tabel 5.3 dan Gambar 5.4.

Tabel 5.3 Hasil Uji Coba Pengaruh Nilai K Terhadap Waktu

No	Nilai K	Waktu (detik)
1	50	0.29
2	100	0.69
3	150	0.83
4	200	1.2
5	250	1.5
6	300	1.86
7	350	2.5
8	400	3.04
9	450	3.54
10	500	3.93



Gambar 5.4 Grafik Hasil Uji Coba Pengaruh Nilai K Terhadap Waktu

Dari hasil uji coba yang telah dilakukan, dapat dilihat bahwa pertumbuhan waktu yang dibutuhkan program mendekati kurva kuadratik seiring dengan pertumbuhan nilai k.

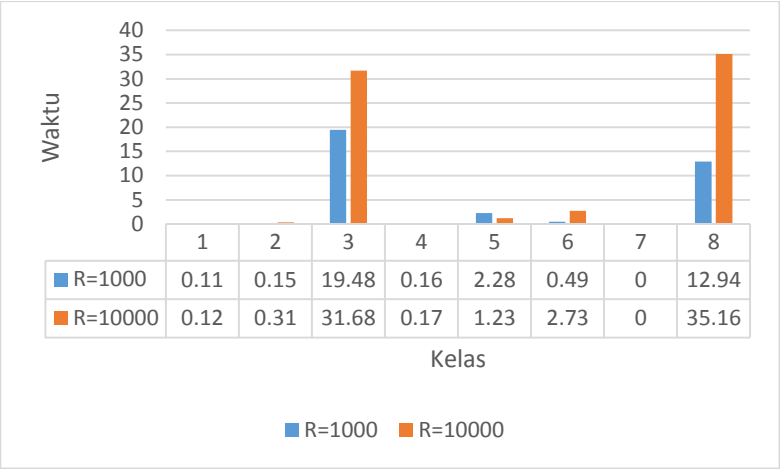
5.2.2.3. Pengaruh Perbedaan Kelas Dataset Terhadap Waktu

Kasus uji yang diujikan sebanyak delapan kasus. Setiap kasus uji merepresentasikan sebuah kelas dataset. Setiap kasus uji memiliki jumlah *item* dan nilai k tetap, yaitu 200 dan 40. Setiap kasus uji duji dengan dua nilai R, yaitu 1000 dan 10000. Setiap kasus uji memiliki batas waktu eksekusi selama 3600 detik. Apabila waktu eksekusi melebihi 3600 detik, maka kasus uji dianggap tidak terselesaikan. Hasil uji coba ditunjukkan pada Tabel 5.4 dan Gambar 5.5.

Secara umum, nilai R yang lebih kecil mengakibatkan waktu eksekusi yang lebih cepat dengan pengecualian pada dataset keenam. Untuk dataset ketujuh, pada kedua nilai R, algoritma hasil implementasi tidak dapat menyelesaikan dataset tersebut.

Tabel 5.4 Hasil Uji Coba Pengaruh Perbedaan Kelas Dataset Terhadap Waktu

No	Kelas Dataset	Waktu (detik)	
		R=1000	R=10000
1	Uncorrelated	0.11	0.12
2	Weakly uncorrelated	0.15	0.31
3	Strongly correlated	19.48	31.68
4	Inverse strongly correlated	0.16	0.17
5	Almost strongly	2.28	1.23
6	Subset-sum	0.49	2.73
7	Even-odd subset sum	∞	∞
8	Even-odd strongly correlated	12.94	35.16



Gambar 5.5 Hasil Uji Coba Pengaruh Kelas Dataset Terhadap Waktu

BAB VI
KESIMPULAN DAN SARAN

Bab ini berisi kesimpulan terhadap hasil uji coba yang yang telah dilakukan. Selain itu, bab ini juga berisi saran mengenai hal-hal yang masih bisa dikembangkan ke depannya.

6.1. Kesimpulan

Dari hasil uji coba yang telah dilakukan terhadap implementasi algoritma B&B yang digunakan untuk memecahkan permasalahan pencarian k solusi terbaik pada permasalahan knapsack, dapat diambil kesimpulan sebagai berikut:

1. Implementasi yang telah dilakukan mampu menyelesaikan permasalahan pencarian k solusi terbaik pada permasalahan knapsack. Hal ini dibuktikan dengan uji kebenaran.
2. Waktu yang dibutuhkan oleh algoritma B&B untuk mencari k solusi terbaik pada permasalahan knapsack tumbuh secara linier terhadap banyak *item*.
3. Waktu yang dibutuhkan oleh algoritma B&B untuk mencari k solusi terbaik pada permasalahan knapsack tumbuh secara kuadratik terhadap nilai k.
4. Algoritma hasil implementasi memiliki kompleksitas waktu $O(nk^2)$ untuk menyelesaikan pencarian k solusi terbaik untuk permasalahan knapsack.
5. Algoritma hasil implementasi mampu menyelesaikan 7 dari 8 kelas dataset yang disediakan.

6.2. Saran

Berikut adalah saran yang dapat dilakukan untuk mengembangkan hasil implementasi solusi pencarian k solusi terbaik pada permasalahan knapsack:

1. Implementasi yang telah dilakukan belum dapat mengatasi dataset kelas 7 pada dataset Pisinger. Perlu dilakukan pengecekan kelas data terlebih dahulu sebelum algoritma utama dijalankan.

BAB VI
KESIMPULAN DAN SARAN

Bab ini berisi kesimpulan terhadap hasil uji coba yang yang telah dilakukan. Selain itu, bab ini juga berisi saran mengenai hal-hal yang masih bisa dikembangkan ke depannya.

6.1. Kesimpulan

Dari hasil uji coba yang telah dilakukan terhadap implementasi algoritma B&B yang digunakan untuk memecahkan permasalahan pencarian k solusi terbaik pada permasalahan knapsack, dapat diambil kesimpulan sebagai berikut:

1. Implementasi yang telah dilakukan mampu menyelesaikan permasalahan pencarian k solusi terbaik pada permasalahan knapsack. Hal ini dibuktikan dengan uji kebenaran.
2. Waktu yang dibutuhkan oleh algoritma B&B untuk mencari k solusi terbaik pada permasalahan knapsack tumbuh secara linier terhadap banyak *item*.
3. Waktu yang dibutuhkan oleh algoritma B&B untuk mencari k solusi terbaik pada permasalahan knapsack tumbuh secara kuadratik terhadap nilai k.
4. Algoritma hasil implementasi memiliki kompleksitas waktu $O(nk^2)$ untuk menyelesaikan pencarian k solusi terbaik untuk permasalahan knapsack.
5. Algoritma hasil implementasi mampu menyelesaikan 7 dari 8 kelas dataset yang disediakan.

6.2. Saran

Berikut adalah saran yang dapat dilakukan untuk mengembangkan hasil implementasi solusi pencarian k solusi terbaik pada permasalahan knapsack:

1. Implementasi yang telah dilakukan belum dapat mengatasi dataset kelas 7 pada dataset Pisinger. Perlu dilakukan pengecekan kelas data terlebih dahulu sebelum algoritma utama dijalankan.

DAFTAR PUSTAKA

[1] Leão, Aline A.S., Cherri, Luiz H., dan Arenales, Marcos N. 2014. "Determining the K-best solutions of knapsack problem". **Computers & Operations Research**. 13, 4:71-82.

[2] Gilmore, P.C., dan Gomory, R.E. 1963. "A Linear Programming Approach to the Cutting Stock Problem-Part II". **Operations Research**.Vol 11 No 6. pp. 863-888.

[3] Clausen, Jens. 1999. "Branch and Bound Algorithms – Principles and Examples". **Operations Research**. Copenhagen, Denmark.

[4] "Dataset Pisinger," [Online]. Available: http://www.diku.dk/~pisinger/codes.hml.

[5] "SPOJ.com – Problem YOSSY," [Online]. Available: http://www.spoj.com/problems/YOSSY/.

BIODATA PENULIS

Penulis bernama lengkap Indra Saputra, lahir di Surabaya pada tanggal 16 April 1993. Penulis adalah anak pertama dari dua bersaudara. Penulis telah menempuh pendidikan formal di TK Baitussalam, Warugunung, Surabaya (1997-1999), SD Negeri 2 Warugunung, Surabaya (1999-2000), SD Negeri 3 Jetis, Mojokerto (2000-2005), SMP Negeri 2 Jetis, Mojokerto (2005-2008), SMA Negeri 1 Sooko, Mojokerto (2008-2011), dan S1 Jurusan Teknik Informatika, Fakultas Teknologi Informasi (FTIf), Institut Teknologi Sepuluh Nopember (ITS) Surabaya (2011-2015). Penulis pernah menjadi finalis GeMasTIK V ITB (2012) pada bidang pengembangan perangkat lunak. Penulis juga pernah menjadi juara I SNITCH FTIf ITS (2013) pada bidang pengembangan permainan. Penulis aktif sebagai anggota Himpunan Mahasiswa Teknik Computer-Informatika (HMTC) ITS sebagai staff PSDM (2012-2013) dan *Steering Committee* (SC) Pengaderan HMTC (2013-2014). Penulis pernah menjadi anggota (2012) dan koordinator (2013) National Logic Competition (NLC) Schematics ITS. Penulis juga pernah menjadi asisten mata kuliah Pemrograman Terstruktur (2012), Algoritma dan Struktur Data (2013), Pemrograman Berorientasi Obyek (2013), dan Perancangan dan Analisis Algoritma (2014) di Jurusan Teknik Informatika FTIf-ITS.