LAPORAN TUGAS BESAR 1

IF2211 STRATEGI ALGORITMA

**Aplikasi “Mini Winzip” untuk Pemampatan dan Penirmampataan *File* dengan**

**Algoritma Huffman (Aplikasi Algoritma *Greedy*)**



Johan - 13514026

Kharis Isriyanto - 13514064

Kevin Supendi - 13514094

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2016

**BAB I**

**DESKRIPSI MASALAH**

Dalam komunikasi data, pesan yang dikirim seringkali ukurannya sangat besar sehingga waktu pengirimannya lama. Begitu juga dengan penyimpanan data, arsip yang berukuran besar membutuhkan ruang penyimpanan yang besar. Kedua masalah ini dapat diatasi dengan mengkodekan pesan atau isi arsip sesingkat mungkin, sehingga waktu pengiriman pesan relatif cepat dan ruang penyimpanan yang dibutuhkan juga sedikit. Cara pengkodean seperti ini disebut kompresi (pemampatan) data dan pemulihan data tersebut kembali seperti aslinya disebut dekompresi (penirmampatan). Salah satu cara pemampatan dan nirmampat data ini adalah dengan menggunakan kode Huffman.

Pada tugas pertama Strategi Algoritma ini, anda diminta membuat aplikasi “*mini WinZip*” yang mengimplementasikan algoritma Huffman untuk memampatkan dan menirmampatkan data. Algoritma Huffman menggunakan prinsip *greedy* dalam pembentukan kode Huffman. Program anda harus dapat memampatkan semua jenis data baik berupa teks, gambar, suara, dan video dan Anda harus mampu mengembalikan data yang sudah dikompres tersebut ke bentuk asalnya (dekompresi). Sebagai contoh, jika *executable file* dimampatkan (misalnya notepad.exe), maka program notepade.exe tersebut harus dapat dijalankan kembali.

Program yang Anda buat selain mampu memampatkan dan menirmampatkan data juga harus dapat menunjukkan perubahan hasil kompresi tersebut melalui nisbah pemampatannya. Nisbah (*ratio*) pemampatan dhitung dengan rumus:

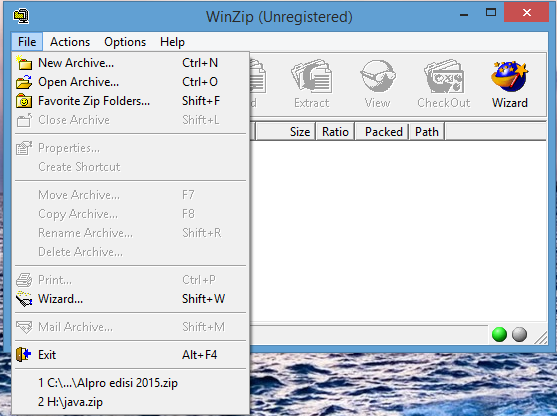
*P* = Ukuran *file* hasil pemampatan/Ukuran *file* sebelum dimampatkan × 100%

Yang berarti ukuran *file* menjadi *P* (dalam persentase) dari ukuran semula.

Waktu yang dibutuhkan untuk memampatkan dan menirmampatkan data juga dicatat. Selain itu, program harus dapat memampatkan banyak *file* sekaligus, dan ketika penirmampatan harus dapat mengembalikannya menjadi *file-file* semula dengan nama sama. Hitung juga entropi pesan dan rata-rata panjang bit setiap simbol di dalam pesan (file). Lihat Lampiran untuk menghitung entropi.

**Spesifikasi program :**

1. Program mampu memampatkan berkas (*file*) berjenis apapun secara tepat dan benar.
2. Program mampu memampatkan banyak file sekaligus menjadi satu berkas mampat.
3. Program mampu menirmampatkan data yang sebelumnya telah dikompres dengan tepat dan benar.
4. Program harus mampu menampilkan proses pemampatan dan nirmampat tersebut (misal dalam *progressive bar*).
5. Program juga harus manampilkan statistik berupa: lama waktu pemampatan (dan penirmampatan), ukuran file semula, ukuran file setelah dimampatkan, nisbah pemampatan, entropi, rata-rata bit/simbol, dll.
6. Antarmuka program sebaiknya kompatibel seperti *Winzip*.



**BAB II**

**DASAR TEORI**

1. **Algoritma Greedy**

Algoritma Greedy adalah salah satu algoritma yang digunakan untuk menyelesaikan maslah optimisasi. Algoritma Greedy akan membentuk solusi langkah per langkah. Terdapat banyak pilihan yang dapat dipilih dalam setiap langkah. Algoritma Greedy melakukan pengambilan keputusan yang terbaik pada setiap langkahnya. Sehingga diharapkan akan didapat solusi yang terbaik pada akhir proses.

Dalam algoritma Greedy, terdapat 5 buah elemen yaitu :

1. Himpunan kandidat, C

Berisi elemen-elemen pembentuk solusi.

1. Himpunan solusi, S

Berisi kandidat yang terpilih sebagai solusi persoalan.

1. Fungsi seleksi (Selection function)

Fungsi yang memilih kandidat yang paling memungkinkan untuk mencapai solusi optimal.

1. Fungsi kelayakan (feasible)

Fungsi yang mengecek apakah suatu kandidat yang sudah dipilih dapat memberikan solusi yang layak atau tidak melanggar syarat

1. Fungsi objektif

Fungsi yang memaksimumkan atau meminimumkan nilai solusi

*Pseudo code* algoritma GreedySumber : Munir, Rinaldi. 2016. Algoritma Greedy. Bahan Kuliah IF2211 Strategi Algoritma. Bandung: Program Studi Teknik Informatika Sekolah Teknik Elektro dan Informatika Institut Teknologi Bandung

Pada setiap langkah, akan terbentuk sebuah solusi lokal. Pada akhir kalang while-do terbentuk sebuah optimum global. Walaupun begitu, kadang solusi yang didpat di akhir bukanlah optimum global, tetapi merupakan sub-optimum atau pseudo optimum. Hal ini dapat terjadi karena algoritma greedy tidak beroperasi secara menyeluruh terhadap semua alternatif solusi yang ada seperti pada metode exhaustive search. Kemungkinan lain adalah terdapatnya beberapa fungsi seleksi yang berbeda, sehingga kita harus memilih fungsi yang tepat jika kita ingin algoritma berjalan dengan benar dan menghasilkan solusi yang paling optimum. Oleh karena itu, pada sebagian masalah algoritma greedy tidak selalu berhasil memberikan solusi yang benar-benar optimum. Algoritma greedy sering berguna untuk menghasilkan solusi yang menghampiri (approximation) optimum, daripada menggunakan algoritma yang lebih rumit untuk menghasilkan solusi yang eksak.

1. **Pohon Biner**

Pohon merupakan salah satu bentuk terapan dari graf tidak berarah yang tidak mengandung sirkuit. Penerapan pohon banyak diterapkan dalam kehidupan sehari-hari seperti silsilah keluarga, bagan pertandingan, dll. Pohon memiliki beberapa terminologi, yakni sebagai berikut:

a. Anak *(children)* merupakan simpul pada bagian di bawah suatu simpul yang lain

b. Orangtua *(parent)* merupakan simpul pada bagian di atas simpul yang lain

c. Akar adalah suatu simpul yang terdapat pada bagian paling atas dari pohon

d. Daun yaitu simpul – simpul yang terletak di ujung paling bawah dari sebuah pohon

e. Aras *(level)* merupakan tingkatan suatu simpul dari akar. Akar memiliki aras = 0.

f. Tinggi *(height)* adalah aras maksimum yang diukur dari akar ke daun terluar.

Pohon terdiri dari berbagai macam bentuk. Salah satunya ialah pohon biner. Pohon biner adalah suatu pohon yang masing-masing simpul memiliki maksimal 2 buah anak. Pohon Biner ini adalah pohon terurut *(ordered tree)* karena urutan anak dianggap penting serta dibedakannya antara anak kiri *(left child)* dan anak kanan *(right child).*

1. **Kode Huffman**

Kode Huffman ditemukan oleh David A. Huffman, seorang mahasiswa MIT pada tahun 1952. Kode Huffman merupakan penerapan dari algoritma greedy dan pohon. Algoritma Huffman sendiri menggunakan prinsip pengkodean data dalam untaian bit biner, dimana karakter yang sering muncul dikodekan dengan rangkaian bit yang pendek, sedangkan karakter yang jarang muncul dikodekan dengan rangkaian bit yang lebih panjang. Kode Huffman adalah suatu kode prefiks yang menggunakan aplikasi dari pohon biner, kode ini umum digunakan dalam kompresi data yang bersifat *lossless* yang artinya data sebelum dan sesudah proses kompresi tidak mengalami perubahan apapun. Kode Huffman memanfaatkan probabilitas dari kemunculan dari tiap-tiap simbol. Salah satu aplikasi dari kode Huffman adalah kompresi file yang berbasis kode ASCII. Contoh dari kode ASCII untuk beberapa karakter ditunjukkan sebagai berikut.

|  |  |
| --- | --- |
| Simbol | Kode ASCII |
| A | 0100 0001 |
| B | 0100 0010 |
| C | 0100 0011 |
| D | 0100 0100 |

Sehingga rangkaian bit untuk string *‘ABACCDA’* adalah 0100000101000010 0100001101000001 0100000101000011 01000100. Sehingga dibutuhkan 56 bit untuk mereprentasikan 7 karakter. Untuk kompresi menggunakan kode huffman, pertama-tama dibuat terlebih dahulu tabel frekuensi dari string tersebut

|  |  |  |
| --- | --- | --- |
| Simbol | Frekuensi | Peluang |
| A | 3 | 3/7 |
| B | 1 | 1/7 |
| C | 2 | 2/7 |
| D | 1 | 1/7 |

Setelah mendapakan frekuensi dari setiap karakter, maka untuk membuat kode huffman nya, dilakukan langkah-langkah sebagai berikut

1. Pilihlah dua simbol dengan peluang atau frekuensi yang paling kecil, sebagai contoh dari string ‘ABCAACD’, simbol B dan D adalah simbol yang peluangnya terkecil, sehingga gabungkanlah kedua simbol tersebut menjadi suatu simpul orang tua baru dengan simbol baru BD dengan peluangnya 1/7+1/7 = 2/7. Simbol ini diperlakukan seperti simbol-simbol sebelumnya dan digunakan untuk mencari peluang simbol terkecil selanjutnya.
2. Lalu, pilihlah dua simbol dengan peluang terkecil selanjutnya, simbol baru BD juga diperhitungkan. Sebagai contoh ini, kedua simbol tersebut adalah BD dan C, yaitu masing-masing peluangnya adalah 2/7. Sehingga seperti langkah 1, dihasilkanlah simbol baru CBD dengan peluang 4/7.
3. Ulangilah langkah 1 sampai 2 sampai seluruh simbol telah habis dan membentuk simbol ABCD yang memuat semua simbol yang ada. Untuk memastikan apakah pohon Huffman tersebut benar, simbol ABCD haruslah memiliki peluang sebesar 1 (7/7)



1. Jadi, dari pohon akan didapat kode baru

|  |  |
| --- | --- |
| Simbol | Kode ASCII |
| A | 0 |
| B | 110 |
| C | 10 |
| D | 111 |

Sehingga untuk string ‘ABCAACD’ jika diimplementasikan ulang dalam kode Huffman menjadi *0110100010111.* Panjang kode Huffman yang didapatkan adalah 13 bit. Sehingga rasio yang didapatkan adalah 13/56 = 23.2%. Dapat dilihat juga bahwa simbol dengan kemunculan yang paling tinggi memiliki kode Huffman yang paling pendek yaitu simbol A dengan panjang kodenya 1 saja. Sedangkan simbol dengan kemunculannya paling sedikit memiliki kode Huffman yang paling panjang yaitu simbol B dan D yang memiliki panjang kode sebesar 3.

1. **Huffman Adaptif**

Selain metode Huffman biasa yang dijelaskan sebelumnya, ada satu lagi metode Huffman yang digunakan untuk memampatkan *file*. Metode Huffman adaptif ini ditemukan oleh Faller, Gallagher, Knuth, dan kemudian oleh Vitter.[1] Metode Huffman adaptif membaca file hanya sekali. Pohon Huffman diubah dan diperbaharui setiap pembacaan karakter.

Metode Huffman adaptif ini memiliki beberapa kelebihan dibanding metode Huffman biasa, yaitu kecepatan pemampatan karena file hanya dibaca sekali. Adapun kekurangannya yaitu jika ada kesalahan 1 saja di tengah proses pemampatan, maka file akan rusak saat dinirmampatkan.[1]

Prinsip dari metode Huffman Adaptif adalah manipulasi pohon pada saat pembacaan karakter. Pohon mempunyai satu *node* khusus bernama NYT (*Not Yet Transmitted*). Fungsi dari *node* tersebut adalah untuk penandaan karakter yang baru pertama kali muncul. Saat memanipulasi pohon, baik saat menambah karakter atau *update* pohon, pohon yang dibentuk haruslah memenuhi kriteria berikut.

1. Frekuensi atau *weight* pada *node* harus lebih besar dari setiap *node* di bawahnya.
2. Frekuensi atau *weight* pada *node* yang berada pada suatu level harus terurut membesar dari kiri ke kanan.
3. Frekuensi atau *weight* dari suatu *node* yang bukan daun harus sama dengan jumlah *weight* dari *node* anak kiri dan kanannya.

Saat pertama dibuat, pohon Huffman hanya memiliki *node* NYT. Oleh karena itu, program akan membaca karakter atau 8 bit selanjutnya. Pohon akan diperbaharui (*update*)dengan menambahkan *node* baru dengan nilai karakter tersebut dan frekuensi 1. Pembacaan karakter akan menentukan apakah akan dibuat *node* baru atau tidak. Jika karakter yang dibaca belum pernah muncul maka tuliskan kode Huffman untuk *node* NYT diikuti karakter yang dibaca. Jika karakter tersebut sudah pernah muncul, maka tuliskan kode Huffman untuk *node* dengan nilai karakter tersebut. Pada setiap pembacaan pohon harus diperbaharui. Jika ada *node* dengan frekuensi sama dengan atau lebih kecil dengan *node* yang frekuensinya akan ditambahkan dengan order yang lebih besar, tukar (*swap*) *node* yang akan diupdate dengan *node* berfrekuensi sama dengan order yang paling besar. Pastikan *node* yang ditukar bukan *parent* dari *node* tersebut.

**BAB III**

**ANALISIS PEMECAHAN MASALAH**

Pada pembuatan aplikasi “Mini Winzip” ditemukan berbagai masalah yang harus dipecahkan di antaranya pengkodean simbol ke dalam pohon Huffman, penulisan bit 1 atau 0 dan penulisan karakter ke dalam file hasil kompres (encode), pembacaan karakter dari file yang dikompres, dan juga masalah pemampatan banyak file.

1. **Penempatan Karakter pada Pohon Huffman Adaptif**

Kami menggunakan pohon Huffman adaptif untuk melakukan pengkodean pesan. Penempatan karakter yang dimaksud di sini adalah bagaimana saat suatu karakter baru pertama kali dibaca (belum pernah muncul sebelumnya) dan saat *update* pohon saat suatu karakter yang sudah pernah muncul dibaca.

Untuk implementasi pohon kami membuat kelas pohon bernama PenTree yang terdiri dari *pointer* menuju kelas lain di dalamnya yang bernama NodeElmt. Kelas NodeElmt terdiri dari *pointer* menuju PenTree \*Left dan \*Right, *integer* kar dan *long long int weight* sebagai info dari NodeElmt tersebut, yaitu karakter yang muncul dan frekuensinya. Selain itu, PenTree juga mempunyai *pointer* menuju PenTree lain bernama Parent untuk menunjuk *parent* dari suatu *node.* Ada juga variabel *integer* order sebagai penanda untuk membuat pohon Huffman adaptif. Untuk menandai NYT dipakai *node* yang memiliki info kar bernilai 300.

Masalah selanjutnya yang kami hadapi adalah saat kami membutuhkan *swap* atau pertukaran info dari suatu *node*. Saat *node* tersebut ditukar, anak-anak dari *node* tersebut ditukar, tetapi *parent*nya tetap. Oleh karena itu kami memisahkan *parent* di luar elemen dari NodeElmt.

Saat memakai metode Huffman adaptif, kita memerlukan atribut *order* untuk mengurutkan *node* karena setiap *node* dengan orderlebih besar harus mempunyai weightyang lebih besar. Oleh karena itu, kami membuat satu kelas lagi yang bernama ListOrder. Kelas ini mempunyai *pointer* menuju TreeNode yang bernama item. Fungsinya untuk menunjuk *node* pada pohon. ListOrder juga mempunyai *pointer* menuju ListOrder selanjutnya. ListOrder akan terurut berdasarkan order dari setiap *node*.

1. **Penulisan dan Pembacaan Karakter File Kompres**

Masalah selanjutnya yang kami hadapi adalah penulisan dan pembacaan karakter ke atau dari file hasil kompres. Kita dapat menuliskan karakter yang berukuran 8 bit langsung ke dalam file, atau menuliskan bilangan biner 1 atau 0 yang diwakili oleh satu bit ke dalam file. Akan tetapi, kita tidak dapat langsung menuliskan satu bit langsung ke dalam file karena kita hanya mempunyai fitur menulis atau membaca karakter berukuran 8 bit dari file.

Untuk memecahkan masalah tersebut kami membuat satu kelas khusus untuk pembacaan dan penulisan file hasil pemampatans. Kelas tersebut kami beri nama iofile. Kelas iofile memiliki atribut ifstream dan ofstream untuk membaca atau menulis ke dalam *file*, *unsigned char* c untuk menyimpan bit-bit sementara yang akan ditulis, dan *integer* count untuk menghitung berapa bit yang sudah ada pada *unsigned char c*. Pada proses kompres, *constructor* membuka ofstream untuk menulis ke dalam file. Jika kita ingin menuliskan biner 1 atau 0, ada metode binwriter dengan parameter integer. Integer tersebut disimpan ke dalam *unsigned char c* setelah sebelumnya digeser ke kiri sebanyak satu kali.dan variabel *count* akan bertambah. *Unsigned char c* akan ditulis jika variabel *count*  sudah mencapai 8. Jika sudah ditulis, *unsigned char* akan direset dan variabel *count* diubah menjadi 0.

Jika kita ingin menuliskan karakter langsung ke dalam file kompres, kita harus melihat apakah *unsigned char* c masih kosong atau sudah berisi. Jika masih kosong kita hanya tinggal menuliskan karakter tersebut ke dalam file. Jika tidak kita harus memecah karakter tersebut. Misalnya di *unsigned char* c sudah terdapat 3 angka biner, kita memecah karakter yang akan ditulis menjadi 5 untuk disambung dengan *unsigned char* c dan 3 untuk disimpan kemudian. *Unsigned char* c yang sudah digabung dituliskan ke dalam file dan 3 karakter yang disimpan akan menunggu masukan selanjutnya sampai sudah terdapat 8 bit. Ilustrasi penulisan karakter ke dalam file ditunjukkan pada diagram berikut.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| x | x | x | x | x | 1 | 0 | 1 |

Isi *unsigned char c* dengan count = 3

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| x | x | x | x | 1 | 0 | 1 | 1 |

Isi *unsigned char c* dengan setelah ditambahkan angka biner 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| x | x | X | 1 | 0 | 0 | 1 | 1 |
|  |  |  |  |  |  |  |  |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |

Tabel atas adalah isi *unsigned char c* dengan *count 5*. Tabel bawah adalah karakter yang akan dimasukkan.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
|  |  |  |  |  |  |  |  |
| x | x | x | 0 | 1 | 0 | 0 | 1 |

Tabel atas adalah karakter yang ditulis ke *file* hasil kompres. Tabel bawah adalah *unsigned char c* setelah karakter ditulis.

Untuk masalah pembacaan bit per bit saat penirmampatan mirip dengan masalah penulisan bit per bit. Kelas membaca satu karakter dan disimpan di *unsigned char c.* Jika diambil satu bit, maka *unsigned char c* berkurang satu dan digeser ke kiri. Jika diambil satu karakter, maka diambil 8 bit sekaligus, jika tidak cukup ambil karakter setelahnya dari *file* dan dipecah seperti pada saat memasukkan karakter pada *file*.

1. **Pemampatan *Multiple Files***

Untuk pemampatan *multiple files* masalah yang kami hadapi adalah bagaimana memisahkan kode dari satu *file* dengan *file* yang lain. Pemampatan *file* hanya menuliskan kode karakter berdasarkan pohon Huffman. Hal ini akan menyulitkan kita mengetahui *file* apa saja yang ada di file hasil pemampatan jika *file* tersebut ingin dinirmampatkan.

Akhirnya kami membuat suatu *header*  yang ditulis sebelum proses pemampatan dimulai. Kami menuliskan panjang digit nama file beserta *path*nya. Setelah itu dipisahkan dengan spasi kami menulis nama file beserta *path*nya. Kami menuliskan spasi lagi dan kemudian menuliskan ukuran *file* untuk memberitahu berapa karakter yang harus dibaca untuk penirmampatan. Kami baru menuliskan kode hasil pemampatan setelah dipisahkan dengan spasi. Di akhir setiap pemampatan kode kami menuliskan *unsigned char* bernilai 255 untuk memberitahu bahwa satu *file* sudah selesai dibaca dan karakter-karakter selanjutnya adalah kode pemampatan *file* lain.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

1. **Implementasi**

Implementasi program dibagi menjadi beberapa modul yang masing-masing mempunyai fungsi-fungsinya tersendiri. Modul-modul tersebut akan dijelaskan di bawah ini.

1. Modul PenTree

PenTree adalah sebuah kelas yang digunakan untuk pengimplementasian pohon Huffman. Kelas PenTree ada di modul tree.cpp dan tree.h.

1. Modul ListOrder

ListOrder adalah sebuah kelas yang digunakan untuk menyimpan *node* pohon. ListOrder mengurutkan *node* berdasarkan *weight.*Modul ListOrder ada di modul tree.cpp dan tree.h

1. Modul iofile

Iofile adalah sebuah kelas yang mengurus pembacaan dan penulisan karakter ke dalam *file* hasil pemampatan. Penulisan dan pembacaan *file* pemampatan tidak bisa memakai pembacaan dan penulisan *file* biasa karena secara teknis kita menuliskan bit demi bit. Iofile ada di *file* file.cpp

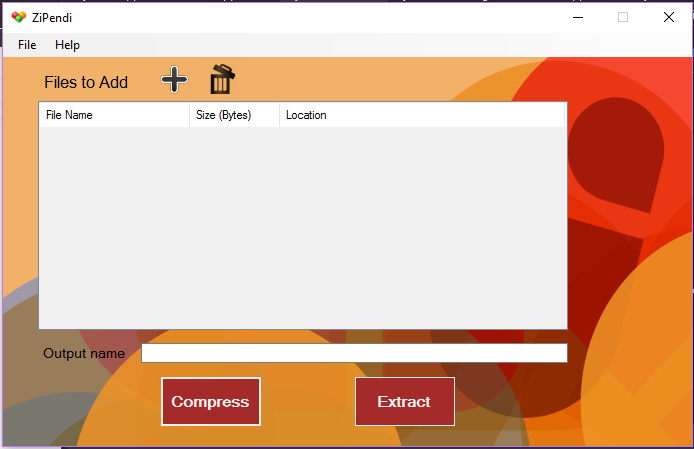
1. Modul Compress

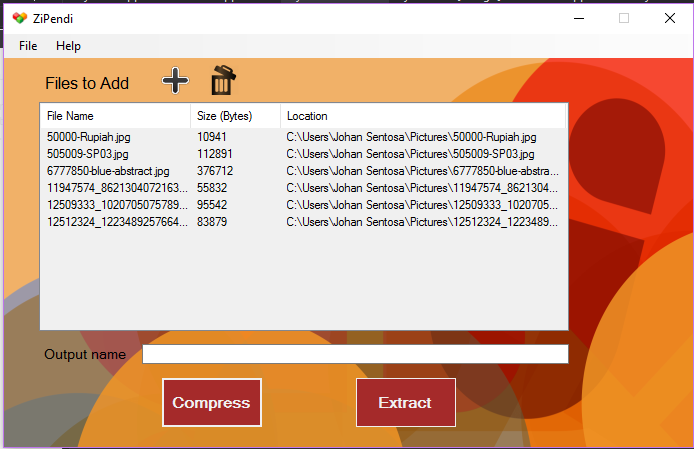
Compress adalah sebuah prosedur untuk memampatkan *file*. Di prosedur ini PenTree dan iofile digunakan bersama-sama untuk memampatkan *file.* Modul ini ada di main.cpp.

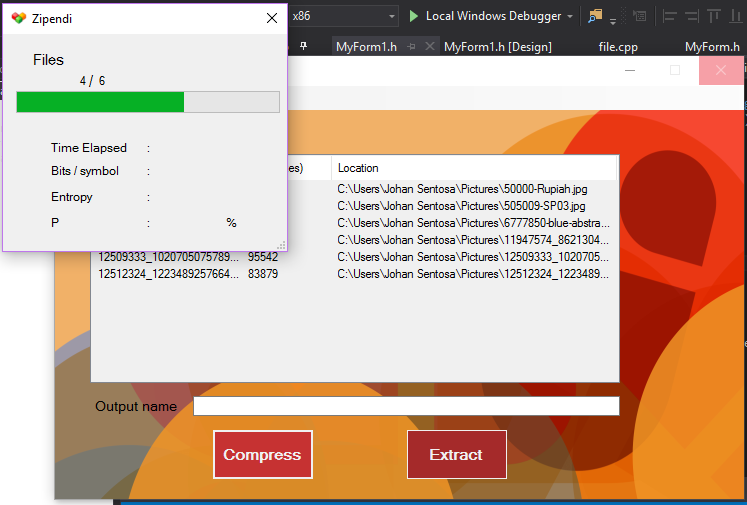
1. Modul Decompress

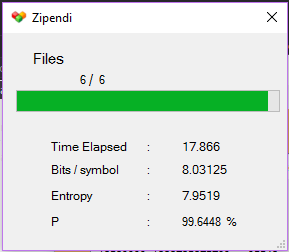
Seperti Compress, Decompress menggunakan PenTree dan iofile secara bersama-sama.Yang membedakan adalah prosedur ini digunakan untuk menirmampatkan *file*. Modul Decompress ada di *file* main.cpp.

1. **Pengujian**









BAB V

KESIMPULAN DAN SARAN

Algoritma *greedy* dapat dimanfaatkan untuk pemampatan *file*. Dalam hal ini algoritma *greedy* diimplementasikan melaluialgoritma Huffman.Algoritma Huffman mengkodekan karakter yang mempunyai frekuensi lebih besar dengan jumlah bit yang lebih sedikit. Sementara karakter yang mempunyai frekuensi kecil dikodekan dengan jumlah bit yang lebih banyak. Hal ini sejalan dengan prinsip algoritma *greedy* yaitu ambil sebanyak*-*banyaknya pada setiap langkah agar tercapai solusi optimum. Solusi optimum yang kita inginkan dalam kasus ini adalah ukuran *file* sekecil mungkin. Hal ini bisa didapat dengan algoritma Huffman. Sementara itu dalam masalah waktu, algoritma Huffman dapat lebih efektif dengan diterapkannya *Adaptive Huffman Coding* yang hanya membaca *file* satu kali saja.

REFERENSI

[1] Low, Jonathan. 2008. “*Adaptive Huffman Coding”.* <https://www.cs.duke.edu/csed/curious/compression/adaptivehuff.html> [diakses pada 15 Februari 2016]