VIZ - Adaptive Huffman Coding (FGK)

Denys Vitali Cristian Spozio

2018-01-19

Contents

Chapter 1

Struttura del codice sorgente

Introduzione

Progetto

Il presente progetto è stato sviluppato presso la Scuola Universitaria della Svizzera Italiana (SUPSI) quale progetto semestrale del corso di *Algoritmi e Strutture Dati (C02008)*.

Specifiche

Le specifiche da rispettare sono le seguenti:

```
I progetti sono svolti a gruppi di 2 studenti. Il contributo di ogni
studente deve essere chiaramente identificato sia nel codice che
nelle presentazioni (Studente A, Studente B).
Ogni progetto ha una fase di analisi e una realizzativa. La fase di
analisi comprende lo studio di documentazione, una fase esplorativa
dove diverse soluzioni vengono abbozzate e messe a confronto, e la
selezione delle soluzioni definitive.
Il programma deve essere scritto in C standard ISO/IEC 9899:1999.
Nessun estensione particolare del linguaggio è consentita.
L'implementazione deve essere verificabile. Il programma DEVE sia
comprimere che decomprimere in due operazioni ben distinte.
```

Esecuzione del programma

```
Il programma deve essere invocabile da linea di commando con la
seguente sintassi:
programma opzioni file-input file-output
dove opzioni deve essere:
-c
per comprimere
-d
per decomprimere.
Altre opzioni sono permesse,
ma non devono essere necessarie per l'esecuzione del programma.
Una volta lanciato, il programma non deve necessitare più nessuna
interazione con l'utente.
Il programma verrà testato in Linux nel modo seguente:
programma -c file.input out.compresso
programma -d out.compresso file.decompresso
diff file.input file.decompresso
```

Autori

Il codice e la documentazione, sono stati realizzati nel periodo 2017-2018 da Cristian Spozio e Denys Vitali.

Algoritmo

Il progetto si basa sull'algoritmo di Huffman Adattivo (FGK - Faller-Gallager-Knuth).
Questo algoritmo si distingue dal Huffman "statico" in quanto l'albero viene generato dinamicamente con l'arrivo di nuovi byte.

Grazie alla dinamicità dell'algoritmo è possibile implementare una compressione di uno stream di dati, senza conoscerne a priori il suo totale contenuto (stream video, audio o dati).

Lo scopo dell'algoritmo è (ovviamente) quello di ridurre la quantità di dati da trasmettere / salvare, sfruttando le ripetizioni di byte.
L'algoritmo risulta essere molto efficiente nella compressione di file con molte ripetizioni (come ad esempio testo), mentre risulta essere poco performante quando i dati hanno un alta entropia.

L'algoritmo si basa sulla proprietà di fratellanza. L'albero generato dovrà sempre rispettare queste due condizioni:

1. Tutti i nodi dell'albero, ad eccezione della radice, devono avere un fratello.
2. I nodi possono essere elencati, da sinistra a destra e dal basso verso l'alto, in ordine di peso.

In questo esempio mostriamo un albero creato dalla codifica della parola "foobar".

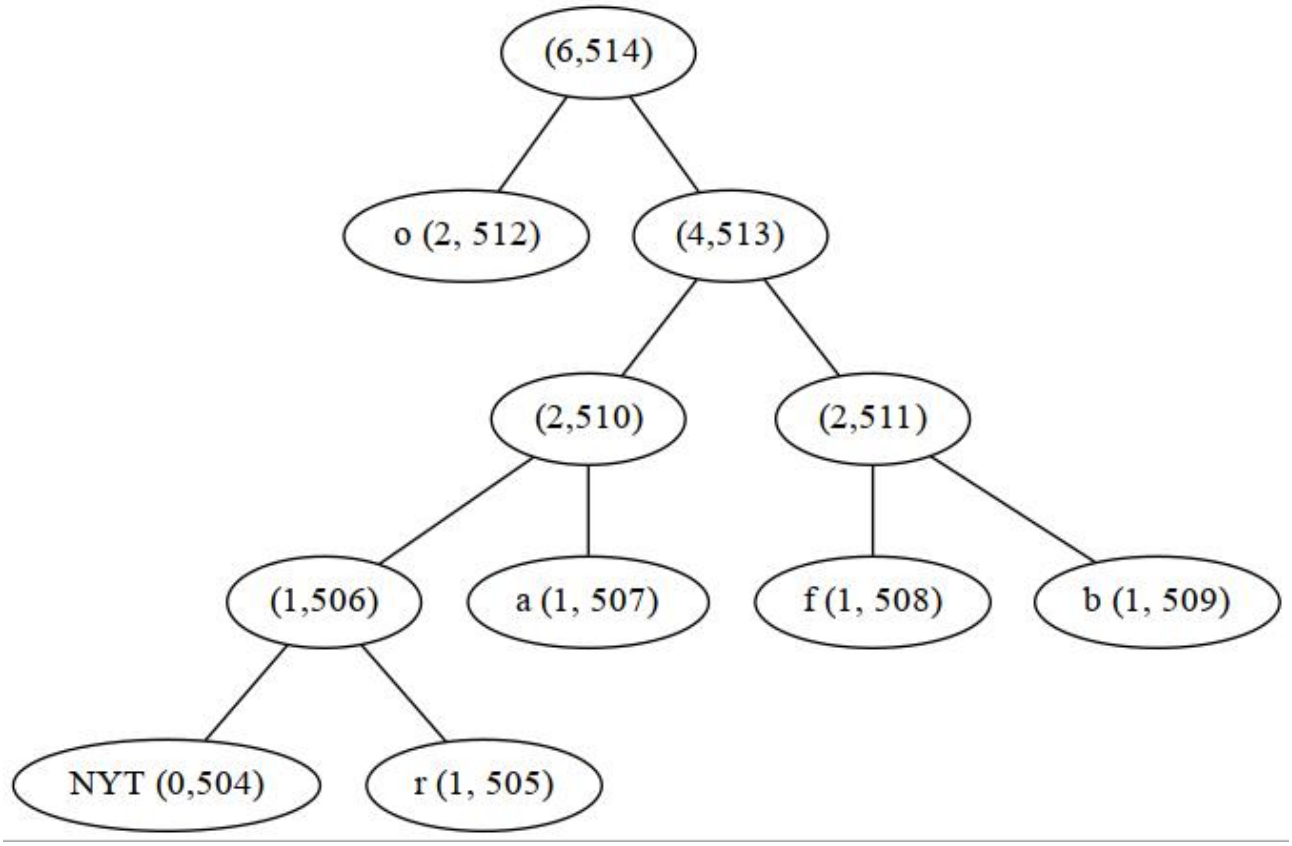

Figure 1.1: Albero di "foobar"

L'albero viene creato a partire da un nodo chiamato NYT (Not-Yet-Translated, "non ancora tradotto") di peso nullo.
Questo nodo è estremamente utile in quanto nella fase di codifica e decodifica il suo percorso (nell'esempio 1000, ossia Radice -> Destra -> Sinistra -> Sinistra -> Sinistra) ci indicherà che i prossimi 8 bit saranno quelli di un nuovo carattere.

Creazione di un albero

L'algoritmo comincia con un albero iniziale contenente un solo nodo, il NYT.

Figure 1.2: Albero vuoto

All'arrivo di un nuovo carattere (per esempio "a"), il compressore verifica se questo è già presente nell'albero. In caso contrario fa nascere dal NYT un sottoalbero con a sinistra il nuovo NYT ed a destra il nuovo elemento, come in figura.

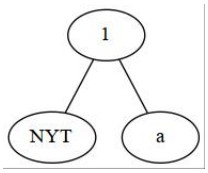

Figure 1.3: Albero di "a"

Essendo la proprietà di fratellanza rispettata, possiamo procedere all'aggiunta di elementi successivi.
All'arrivo di un'altro carattere (per esempio "b"), il compressore riesegue il controllo di esistenza e fa nascere un nuovo sottoalbero dal NYT precedente.

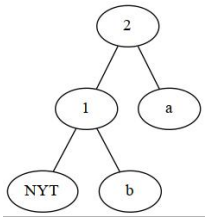

Figure 1.4: Albero di "ab"

Anche in questo caso, prima dell'aggiornamento dei pesi non è necessario effettuare nessuno scambio in quanto la proprietà di fratellanza è rispettata.

All'arrivo di un terzo carattere ("c"), la situazione cambia.

In questo caso il nodo interno che in precedenza aveva peso 1 deve essere scambiato con "a" in quanto questo incrementerà di valore ed andrà a rompere la proprietà di fratellanza. La tecnica alla base dello scambio è quella di andare a scambiare il nodo al quale verrà incrementato il peso con l'ultimo del suo peso e così via per il *parent* che acquisirà.

La proprietà di fratellanza è così rispettata, ed il compressore può quindi continuare a creare l'albero in modo dinamico.

Codifica

Compressore e decompressore concordano sul medoto di indirizzamento dei nodi quale l'invio del bit 1 se, nel percorso per raggiungere il nodo, si va verso il *child* di destra oppure del bit 0 nel caso di quello di sinistra. Questa scelta è effettuata puramente a livello di codice (non varia al variare del file). Durante l'aggiunta degli elementi nell'albero, il compressore prepara i dati da inviare. All'arrivo di un carattere non ancora visto viene inviato il percorso del NYT (nell'ultimo albero 100), e gli 8 bit del byte relativo al carattere. Nel caso in cui l'elemento è già stato visto, viene inviata la sua posizione nell'albero, facendo così risparmiare molti bit in caso di ripetizioni continue (ad esempio, per rappresentare "a" ci basterà inviare "0").

Al termine della compressione, nel caso in cui il byte non sia completato (lavorando con i bit potrebbe capitarci di non avere un bitstream divisibile per 8), la nostra implementazione invia il percorso del NYT, ma dato che gli

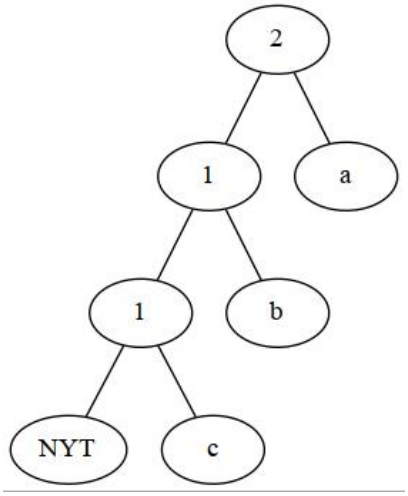

Figure 1.5: Albero di “abc”

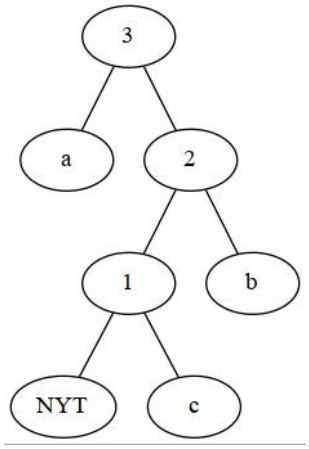

Figure 1.6: Albero corretto di “abc”

8 bit successivi non esistono (perché appunto lo stream termina), il decompressore capirà che il suo lavoro è terminato.

Decodifica

La decodifica funziona in modo pressoché identico: all'arrivo di un nuovo byte, questo viene analizzato ed i percorsi presenti sotto forma di bit vengono verificati con l'albero del decompressore. All'arrivo del percorso del NYT, il decompressore si occupa di interpretare i prossimi 8 bit come un byte, e procede quindi all'aggiunta nel suo albero dell'elemento ed all'output su file dello stesso.
Nel caso in cui il percorso verificato non è quello del NYT, il decompressore aggiunge l'elemento ottenuto dal percorso all'albero e manda in output il carattere associato.

In questo modo, compressore e decompressore hanno sempre il medesimo albero, e possono quindi condividersi percorsi brevi per rappresentare byte frequenti.

Ulteriori informazioni

Purtroppo l'algoritmo non è ben documentato, non esistono RFC o documenti che lo descrivono in modo ottimale ed in rete è presente molto poco a riguardo. Esiste però una visualizzazione grafica molto ben realizzata che vale la pena di consultare: Visualizing Adaptive Huffman Coding, di Ben Tanen - COMP-150.
Con l'ausilio di una visualizzazione grafica dell'algoritmo, questo diventa di facile comprensione e può dunque essere sviluppato in maniera più efficace.

Licenza

Suddivisione dei file

La scelta che è stata fatta per la suddivisione dei file è stata quella di avere:
- 1 file per la definizione della licenza: LICENSE
- 1 file per la gestione dei test: minunit.h
- 2 file per la gestione della compilazione: Makefile e defines.h
- 3 file per la gestione della stampa e lo stile di essa: colors.h e console.h/c
- 5 file per codifica e decodifica di Huffman Adattivo: huffman.h/c, main.c, utilities.h/c

Scopo di ogni file

LICENSE

LICENSE è un file di testo nel quale viene semplicemente definita la licenza del software.

minunit.h

minunit.h è un minimal unit testing framework basato su MinUnit. Si differenzia dall'originale semplicemente per l'integrazione di tags per i test.
Il file contiene le seguenti funzioni: * mu_assert * mu_run_test * mu_tag

che sono ripettivamente le funzioni di messaggio, esecuzione ed etichettamento dei test.

Makefile

Makefile è un file necessario per semplificare la compilazione del codice.

defines.h

defines.h è un header file contenente vari flags necessari alla compilazione del codice quali DEBUG / TEST / RELEASE mode, versione, fallback per commit versioning. Sono inoltre presenti scelte arbitrarie di progetto (come il NYT number ed il Magic Number).

colors.h

colors.h è un header file che definisce stile e colori da utilizzare nella stampa.

console.{c,h}

colors.h e *colors.c* sono rispettivamente header file e file di codice contenenti definizioni delle funzioni di stampa a video presenti nel codice.

huffman.{c,h}

huffman.h e *huffman.c* sono rispettivamente header file e file di codice contenenti definizioni delle funzione atte alla gestione dell'albero presente nell'algoritmo.

main.c

main.c è il file principale dove sono definite e implementate le funzioni contenenti le operazioni quali lettura e scrittura su file.

utilities.{c,h}

utilities.h e *utilities.c* sono rispettivamente header file e file di codice contenenti funzioni di varia utilità utilizzate nel progetto.

Chapter 2

Strutture dati utilizzate

Node

Node è una `struct` definita come segue:

```
typedef struct Node{
    int node_number;
    int weight;
    int element;
    struct Node* left;
    struct Node* right;
    struct Node* parent;
} Node;
```

La quale identifica ogni nodo dell'albero con i rispettivi attributi (*node_ number*, *weight* e *element*). Sono inoltre presenti i puntatori ai vari nodi limitrofi quali il *parent* e i *child*

HuffmanTree

HuffmanTree è una `struct` definita come segue:

```
typedef struct{
    Node* root;
    Node* tree[HUFFMAN_ARRAY_SIZE]; // 514
    Node* element_array[HUFFMAN_SYMBOLS];
    Node* nyt;

    char* output;
    int output_length;

    char* partial_output;
    int partial_output_length;

    int elements;
    unsigned int mode;
    unsigned char mask;
    unsigned short decoder_last_chunk;
    int decoder_byte;
    int buffer;
} HuffmanTree;
```

La quale identifica l'albero con i suoi attributi (*root* e *nyt*). Contiene inoltre un *array* utilizzato per avere un accesso più veloce ai nodi dell'albero ed il rispettivo numero (*tree* ed *elements*). Le componenti: *output*, *partial_ output* ,*output_ length*, *partial_ output_ length* sono necessarie per le stampe su file e a video.

Il campo *mode* è necessario in quanto alcune delle funzioni ausiliarie (come per esempio `add_new_element`) hanno comportamenti differenti in caso di compressione o decompressione.
La componente *mask* è utilizzata per scrivere i byte bit per bit.

L'array `element_array` contiene i puntatori ai nodi, ordinati per carattere. È così possibile ottenere il puntatore al nodo desiderato semplicemente utilizzando la sintassi `ht->element_array[carattere]`. Questo array rende molto veloce la funzione di ricerca dei nodi (`find_nodes`) che è chiamata ad ogni chiamata di `add_new_element`. L'elemento NYT sarà sempre in ulitma posizione (256).

Chapter 3

Istruzioni per l'utilizzo

Compilazione

Nota: Per maggiori opzioni di compilazione, consultare il file README.md presente nella root del progetto

Per compilare il progetto è sufficiente utilizzare il comando:

`make release`

Verrà così generata la versione di release (senza debugging symbols ed ottimizzata), dal nome `viz-release`.
Per altri target (come `debug` o `test` si consulti il README.md)

Esecuzione

Compressione

La sintassi di compressione è la seguente:

`./viz-release -c input output.viz`

dove l'argomento `-c` serve per eseguire il programma in compressione, `input` è un file di input con una qualsiasi estensione e `output.viz` è il nome del file che si vorrà avere in output.
È inoltre possibile specificare il flag `-f` per forzare la sovrascrittura del file `output.viz` nel caso in cui questo sia già presente.

Nota: È *consigliato* (ma non necessario) mantenere l'estensione `.viz` per riconoscere visualmente quali file sono stati compressi.

Nota 2: La sintassi di compressione è stata **imposta** dai requisiti del progetto. Questa sintassi è opposta alla sintassi classica di *qualsiasi altro compressore* presente in UNIX. Per ripristinare il comportamento e mantenere la consistenza che UNIX necessita, si setti il flag `INVERTED_COMPRESSION_FLAG` in `defines.h` a `0`.

Decompressione

La sintessi per la decompressione è invece la seguente:

`./viz-release -d input.viz`

dove l'argomento `-d` serve per eseguire il programma in decompressione e `input.viz` è il file di input (generato precedentemente con il flag `-c`) da decomprimere.

Chapter 4

Performance

Velocità

Senza `ht->element_array`

Table 4.1: Compressione e Decompressione a confronto

Nome file	Dim (kB)	Dim compr. (kB)	Comp. rate	C [s]	D [s]	C [kB/s]	D [kB/s]
bible.txt	4347	2518	-72.67%	7.15	7.68	608.0	566.0
adaptivehuffman.txt	8	5	-64.80%	0.02	0.02	377.5	377.5
100000.txt	575	256	-124.91%	0.51	0.6	1127.6	958.5
sinusoide.txt	6	4	-52.48%	0.02	0.02	304.7	304.7
immagine.tiff	3274	3170	-3.29%	3.85	3.83	850.5	854.9
alice.txt	164	96	-70.50%	0.3	0.32	545.3	511.2
32k_random	32	32	0.05%	0.28	0.12	114.3	266.7
32k_ff	32	4	-697.47%	0.01	0.02	3200.0	1600.0
It Is Wednesday My Dudes.mp4	703	703	0.01%	1.2	0.72	585.8	976.3
04.bin	1024	1024	0.00%	7.25	3.71	141.2	276.0
denys.gif	25825	25646	-0.70%	29.63	31.1	871.6	830.4

Con l'utilizzo `ht->element_array`

Table 4.2: Prestazioni con l'utilizzo di ht->elements_array

Nome file	O. Size [kB]	C. Size [kB]	Rate	C [s]	D [s]	C [kB/s]	D [kB/s]
more-a-3.txt	0.1	0.0	67.9%	0.0	0.0	0.0	0.0
bibbia-lf.txt	4519.2	2565.8	43.2%	1.1	1.3	4304.0	3372.5
abcdefghi.txt	0.0	0.0	-190.0%	0.0	0.0	0.0	0.0
bible.txt	4347.0	2517.5	42.1%	1.0	1.3	4435.8	3343.9
more-a-4.txt	0.1	0.0	68.2%	0.0	0.0	0.0	0.0
more-a.txt	0.1	0.0	70.2%	0.0	0.0	0.0	0.0
aa.txt	0.0	0.0	-400.0%	0.0	0.0	0.0	0.0
1000.txt	3.8	1.6	57.0%	0.0	0.0	0.0	0.0
abcdefghij.txt	0.0	0.0	-181.8%	0.0	0.0	0.0	0.0
adaptivehuffman.txt	7.5	4.6	39.3%	0.0	0.0	0.0	0.0
ripetizioni.txt	0.0	0.0	-163.6%	0.0	0.0	0.0	0.0
more-a-2.txt	0.1	0.0	67.6%	0.0	0.0	0.0	0.0
a.txt	0.1	0.0	69.4%	0.0	0.0	0.0	0.0
fitnessgram.txt	0.8	0.5	36.5%	0.0	0.0	0.0	0.0
100000.txt	575.1	255.7	55.5%	0.1	0.1	8215.6	5228.1
sinusoide.txt	6.1	4.0	34.4%	0.0	0.0	0.0	0.0

Nome file	O. Size [kB]	C. Size [kB]	Rate	C [s]	D [s]	C [kB/s]	D [kB/s]
bg.jpg	7851.5	7845.1	0.1%	6.7	7.3	1175.4	1081.5
compress.gif	1126.2	1125.9	0.0%	1.0	1.1	1161.0	1072.5
mans-not-hot.png	72.2	72.3	-0.0%	0.1	0.0	1031.9	0.0
nyny-denvit.jpg	1993.4	1993.4	-0.0%	1.8	0.0	1107.4	0.0
denys.gif	25825.1	25646.0	0.7%	21.9	23.9	1177.6	1081.0
jenkins.svg	26.8	16.6	38.1%	0.0	0.0	2677.4	0.0
immagine.tiff	3274.4	3170.0	3.2%	2.7	2.9	1226.4	1121.4
ff_ff_ff	0.0	0.0	-433.3%	0.0	0.0	0.0	0.0
empty	0.0	0.0	0.0%	0.0	0.0	0.0	0.0
alice.txt	163.6	95.9	41.3%	0.1	0.1	3271.8	3271.8
32k_random	32.0	32.0	-0.0%	0.0	0.0	1600.0	0.0
32k_ff	32.0	4.0	87.5%	0.0	0.0	0.0	0.0
20170217.mp3	8154.4	8082.1	0.9%	6.8	7.6	1200.9	1078.6
bbc-radio1-ibiza-1.mp3	581.6	580.0	0.3%	0.5	0.5	1187.0	1077.1
It Is Wednesday My Dudes.ogg	113.4	106.8	5.8%	0.1	0.1	1417.1	1259.7
Never Gonna Give You Up.flac	150269.1	150269.1	-0.0%	130.8	0.1	1148.5	1001793.9
file-2.bin	0.0	0.0	-123.1%	0.0	0.0	0.0	0.0
file-1.bin	0.0	0.1	-44.4%	0.0	0.0	0.0	0.0
file-3.bin	0.0	0.1	-44.4%	0.0	0.0	0.0	0.0
DejaVu.webm	9809.6	9798.8	0.1%	8.6	9.4	1139.3	1042.5
TavInterFlip.wmv	14865.0	14840.1	0.2%	13.0	14.1	1146.1	1056.5
Wednesday.mp4	702.9	702.9	-0.0%	0.6	0.0	1098.3	0.0
07.bin	0.0	0.0	-60.0%	0.0	0.0	0.0	0.0
04.bin	0.0	0.0	-60.0%	0.0	0.0	0.0	0.0
08.bin	0.0	0.0	-60.0%	0.0	0.0	0.0	0.0
06.bin	0.0	0.0	-60.0%	0.0	0.0	0.0	0.0
10.bin	0.0	0.0	-60.0%	0.0	0.0	0.0	0.0
01.bin	0.0	0.0	-60.0%	0.0	0.0	0.0	0.0
09.bin	0.0	0.0	-60.0%	0.0	0.0	0.0	0.0
03.bin	0.0	0.0	-60.0%	0.0	0.0	0.0	0.0
02.bin	0.0	0.0	-60.0%	0.0	0.0	0.0	0.0
05.bin	0.0	0.0	-60.0%	0.0	0.0	0.0	0.0
07.bin	0.0	0.0	-240.0%	0.0	0.0	0.0	0.0
04.bin	0.0	0.0	-240.0%	0.0	0.0	0.0	0.0
08.bin	0.0	0.0	-240.0%	0.0	0.0	0.0	0.0
06.bin	0.0	0.0	-240.0%	0.0	0.0	0.0	0.0
10.bin	0.0	0.0	-240.0%	0.0	0.0	0.0	0.0
01.bin	0.0	0.0	-240.0%	0.0	0.0	0.0	0.0
09.bin	0.0	0.0	-240.0%	0.0	0.0	0.0	0.0
03.bin	0.0	0.0	-240.0%	0.0	0.0	0.0	0.0
02.bin	0.0	0.0	-240.0%	0.0	0.0	0.0	0.0
05.bin	0.0	0.0	-240.0%	0.0	0.0	0.0	0.0
07.bin	1024.0	1024.0	-0.0%	0.9	0.0	1177.0	0.0
04.bin	1024.0	1024.0	-0.0%	0.9	0.0	1113.0	0.0
08.bin	1024.0	1024.0	-0.0%	0.9	0.0	1150.6	0.0
06.bin	1024.0	1024.0	-0.0%	0.9	0.0	1150.6	0.0
10.bin	1024.0	1024.0	-0.0%	0.9	0.0	1177.0	0.0
01.bin	1024.0	1024.0	-0.0%	0.9	0.0	1177.0	0.0
09.bin	1024.0	1024.0	-0.0%	0.9	0.0	1177.0	0.0
03.bin	1024.0	1024.0	-0.0%	0.9	0.0	1150.6	0.0
02.bin	1024.0	1024.0	-0.0%	0.9	0.0	1177.0	0.0
05.bin	1024.0	1024.0	-0.0%	0.9	0.0	1177.0	0.0
07.bin	10.0	10.0	-0.1%	0.0	0.0	1000.0	0.0
04.bin	10.0	10.0	-0.1%	0.0	0.0	0.0	0.0
08.bin	10.0	10.0	-0.1%	0.0	0.0	0.0	0.0
06.bin	10.0	10.0	-0.1%	0.0	0.0	0.0	0.0
10.bin	10.0	10.0	-0.1%	0.0	0.0	0.0	0.0
01.bin	10.0	10.0	-0.1%	0.0	0.0	1000.0	0.0
09.bin	10.0	10.0	-0.1%	0.0	0.0	0.0	0.0

Nome file	O. Size [kB]	C. Size [kB]	Rate	C [s]	D [s]	C [kB/s]	D [kB/s]
03.bin	10.0	10.0	-0.1%	0.0	0.0	0.0	0.0
02.bin	10.0	10.0	-0.1%	0.0	0.0	0.0	0.0
05.bin	10.0	10.0	-0.1%	0.0	0.0	0.0	0.0
07.bin	2.0	2.0	-0.6%	0.0	0.0	0.0	0.0
04.bin	2.0	2.0	-0.6%	0.0	0.0	0.0	0.0
08.bin	2.0	2.0	-0.6%	0.0	0.0	0.0	0.0
06.bin	2.0	2.0	-0.6%	0.0	0.0	0.0	0.0
10.bin	2.0	2.0	-0.6%	0.0	0.0	0.0	0.0
01.bin	2.0	2.0	-0.6%	0.0	0.0	0.0	0.0
09.bin	2.0	2.0	-0.6%	0.0	0.0	0.0	0.0
03.bin	2.0	2.0	-0.6%	0.0	0.0	0.0	0.0
02.bin	2.0	2.0	-0.6%	0.0	0.0	0.0	0.0
05.bin	2.0	2.0	-0.6%	0.0	0.0	0.0	0.0
07.bin	1.0	1.0	-1.2%	0.0	0.0	0.0	0.0
04.bin	1.0	1.0	-1.2%	0.0	0.0	0.0	0.0
08.bin	1.0	1.0	-1.2%	0.0	0.0	0.0	0.0
06.bin	1.0	1.0	-1.2%	0.0	0.0	0.0	0.0
10.bin	1.0	1.0	-1.2%	0.0	0.0	0.0	0.0
01.bin	1.0	1.0	-1.2%	0.0	0.0	0.0	0.0
09.bin	1.0	1.0	-1.2%	0.0	0.0	0.0	0.0
03.bin	1.0	1.0	-1.2%	0.0	0.0	0.0	0.0
02.bin	1.0	1.0	-1.2%	0.0	0.0	0.0	0.0
05.bin	1.0	1.0	-1.2%	0.0	0.0	0.0	0.0

Spiegazione

L'algoritmo risulta essere molto efficiente nel caso di file lunghi e con molte ripetizioni. Questa condizione è spesso verificata in file testuali, file bitmap oppure in sequenze di numeri.
Sfortunatamente il rateo di compressione non è proprio soddisfacente nel caso in cui il file ha un alta entropia (ossia le sue frequenze sono pressoché uniformi).

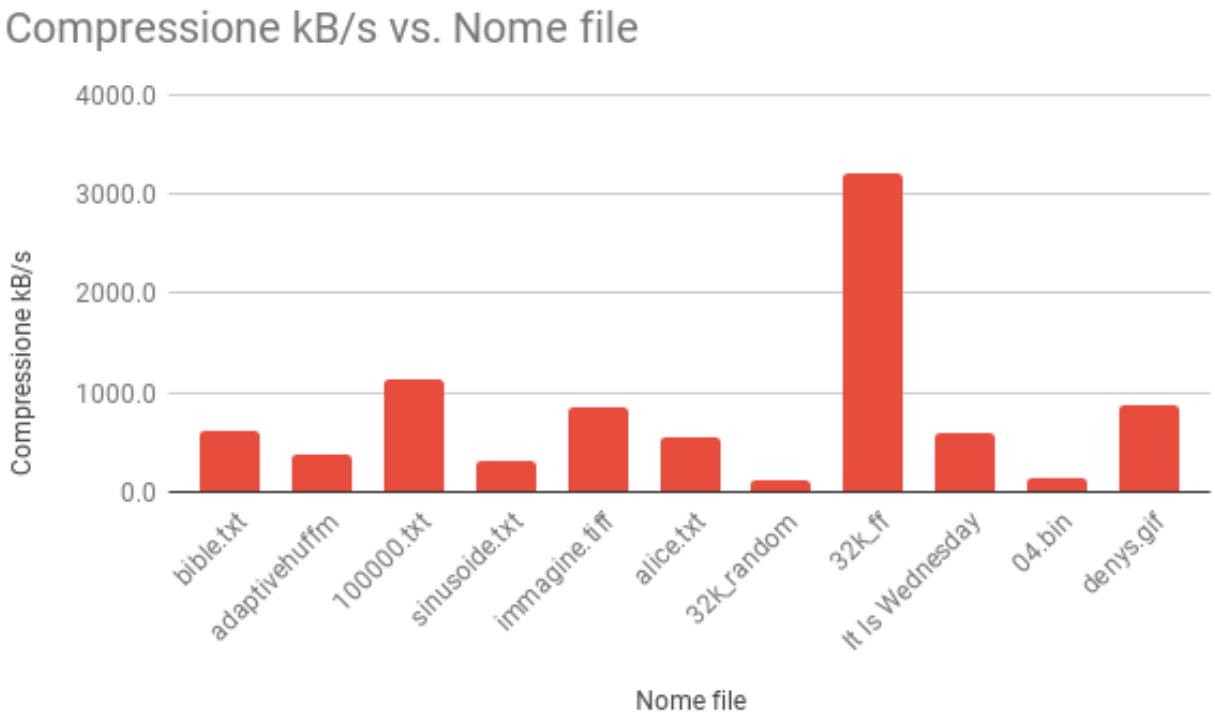

Figure 4.1: Compressione kB/s per file

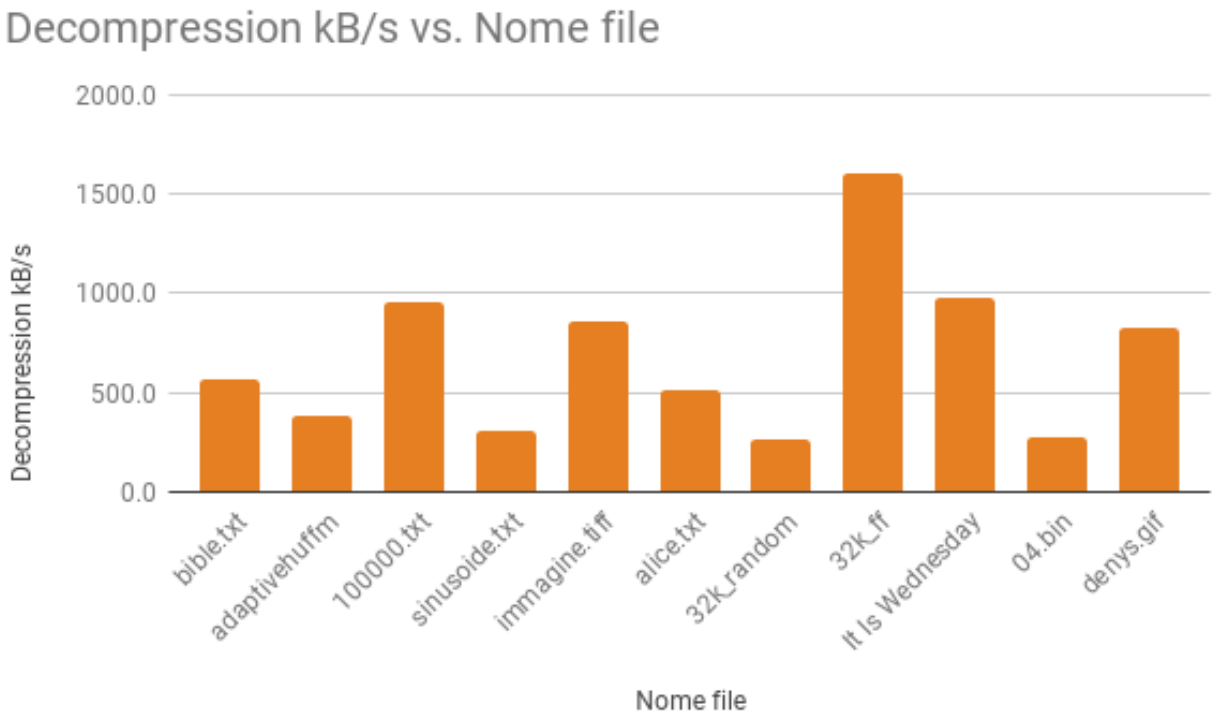

Figure 4.2: Decompressione in kB/s per file

Consumo di memoria

Compressione

Il compressore ha un consumo di memoria lineare di circa 50KiB, indipendente dalla dimensione del file.
Nell'esempio qui sotto è stata compressa un immagine da 2MB tramite il comando `make massif_release_intense_c`.

Decompressione

Il decompressore ha un consumo di memoria lineare, intorno ai 100KiB per qualsiasi tipo di file.
Nell'esempio qui sotto è stata utilizzata un file compresso di 2MB. L'esempio è riproducibile su un qualisasi altro sistema con il comando `make massif_release_intense_d`.

Code coverage

Con l'ausilio della code coverage fornita da Callgrind, abbiamo potuto monitorare quale funzione era chiamata più di frequente e quindi necessitava maggiore ottimizzazione.

Compressione

Durante la fase di compressione la funzione chiamata più spesso risulta essere `add_new_element`. È quindi importante che questa sia molto efficiente al fine di ridurre al minimo i tempi di compressione.

Decompressione

In decompressione la funzione chiamata più spesso è `decode_byte`, che a sua volta chiama `add_new_element`.
Per questo motivo è molto importante che le due funzioni siano più efficienti possibili, al fine di avere tempi di decompressione minimi.

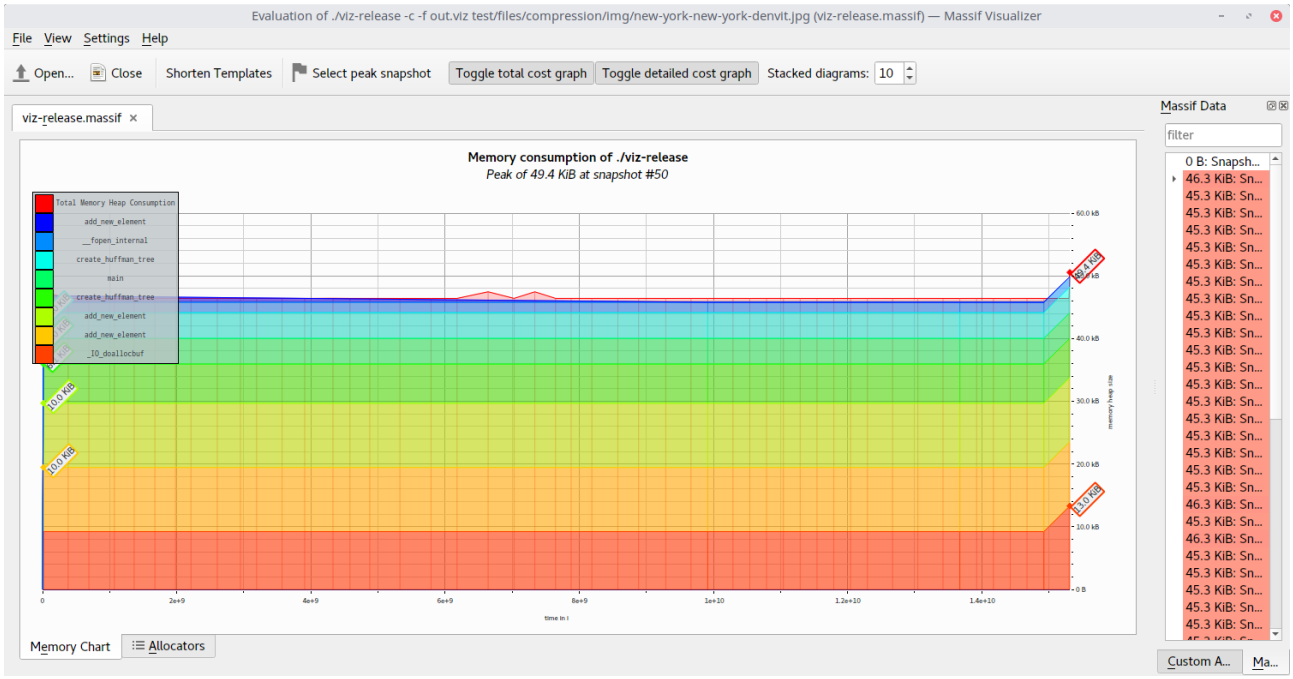

Figure 4.3: Risultato di `make massif_release_intense_c`

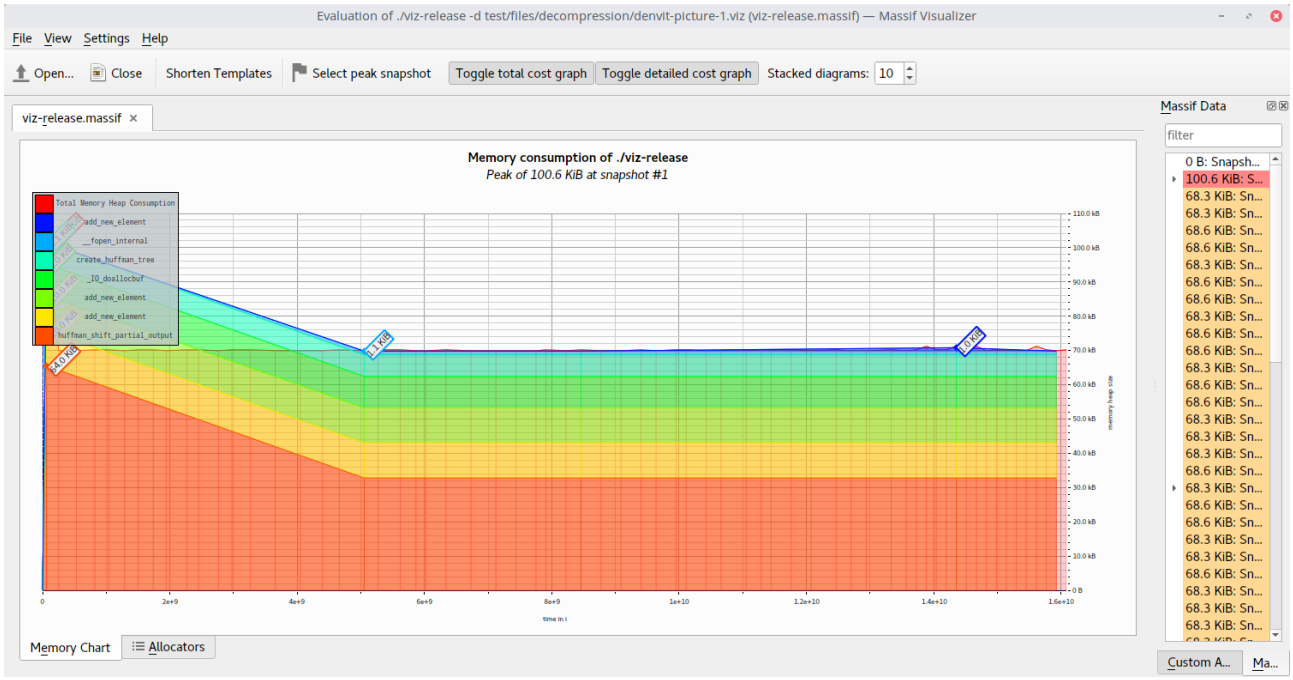

Figure 4.4: Risultato di `make massif_release_intense_d`

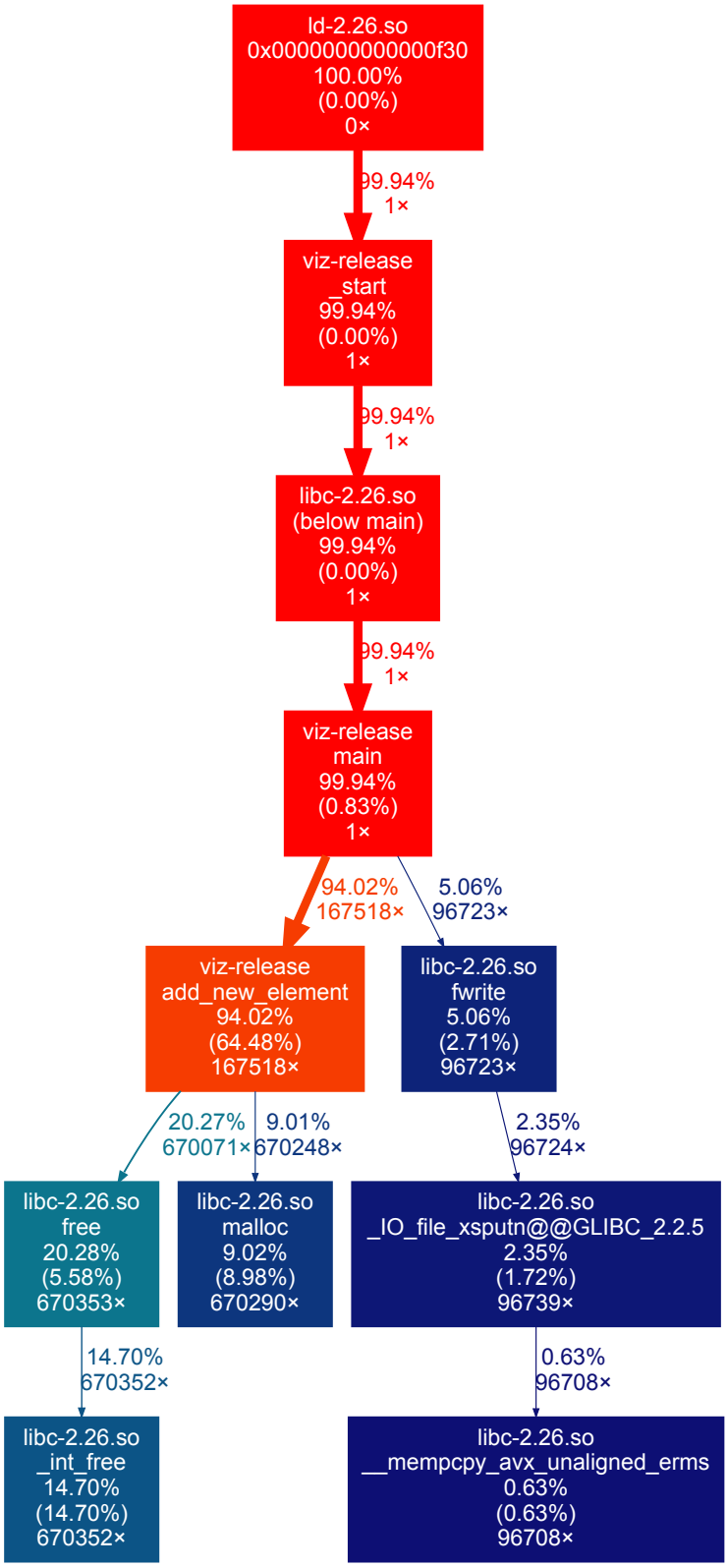

Figure 4.5: Risultato di `make callgrind_release_c`

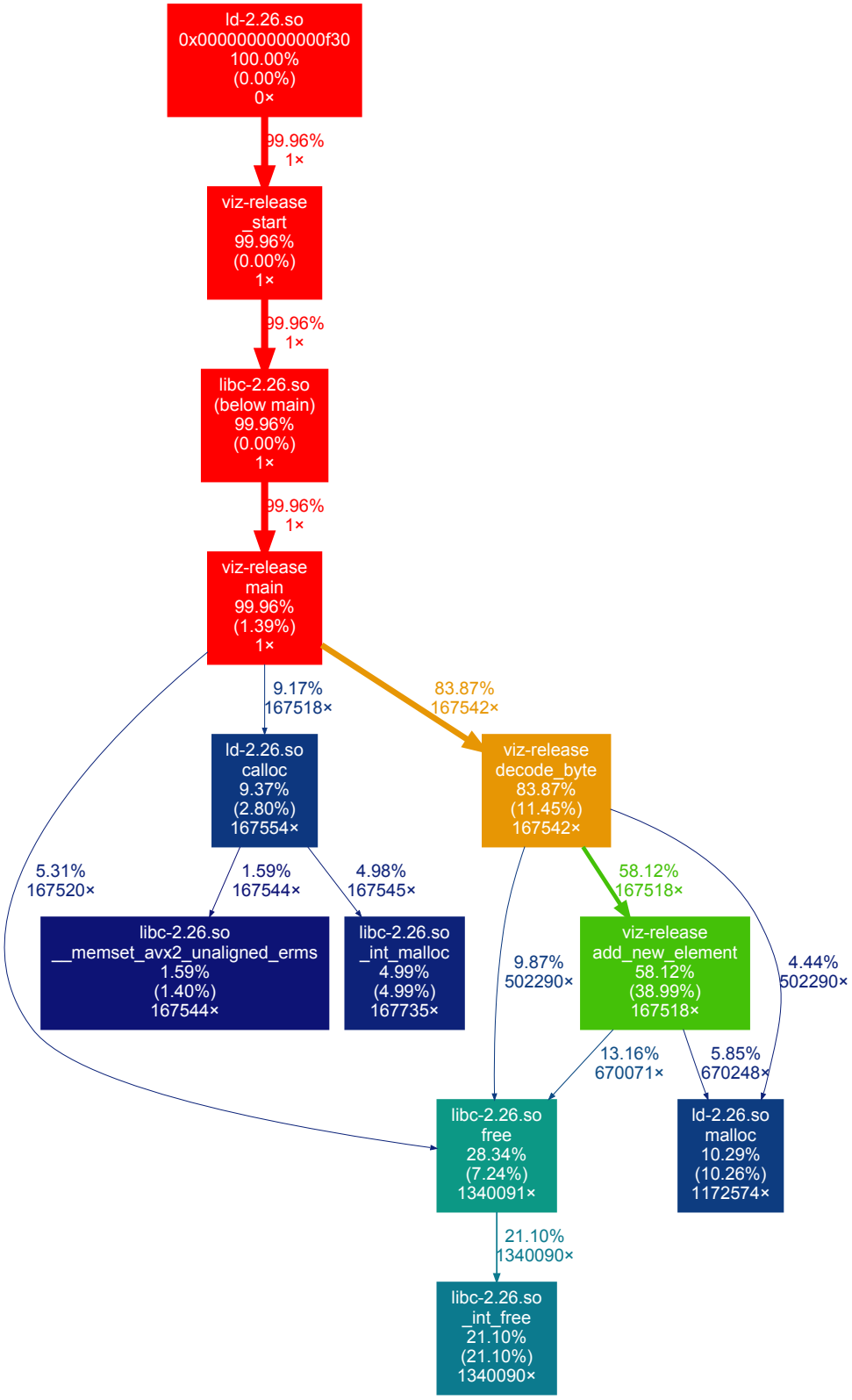

Figure 4.6: Risultato di `make callgrind_release_d`

Valgrind

Di seguito viene illustrato il trend di Valgrind nel corso delle build.

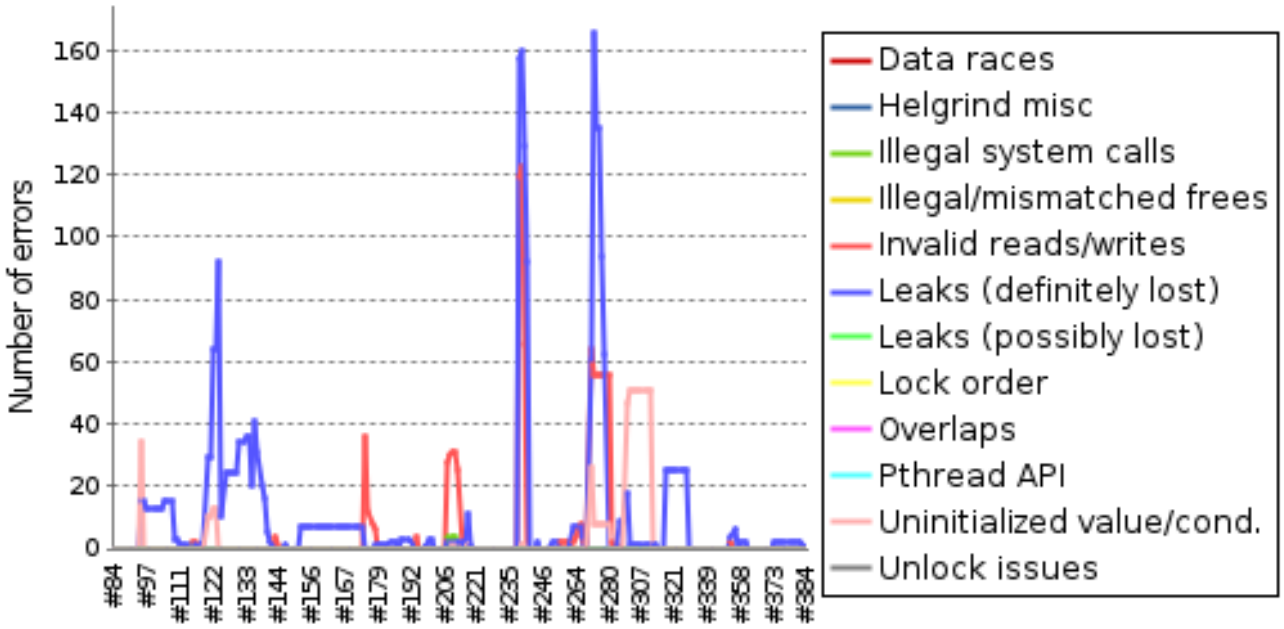

Figure 4.7: Trend di Valgrind

Chapter 5

Descrizione del codice

main.c

Il codice seguente ci permette, nel caso ci sia un singolo parametro all'avvio, di mostrare l'utilizzo del software:

```
if (argc == 1) {
    usage();
    return 0;
}
```

Utilizzando il flag `-v` possiamo ricevere informazioni riguardanti la versione del software:

```
if (strcmp(argv[1], "-v") == 0) {
    version();
    return 0;
}
```

Utilizzando il flag `-c` entriamo in modalità compressione ed andiamo a controllare se ci sono flag aggiuntive viene settata l'apposito flag. Nel caso non siano presenti 4 parametri e questo non sia dovuto per l'inserimento di una flag il software mostrerà come essere utilizzato correttamente e si bloccherà.

```
if (strcmp(argv[1], "-c") == 0) {
    int flag_overwrite = 0;
    int flag_shift = 0;

    if (strncmp(argv[2], "-", 1) == 0) {
        if (strcmp(argv[2], "-f") == 0) {
            flag_overwrite = 1;
            flag_shift++;
        }
    }
    if (argc != 4 && flag_shift == 0) {
        usage();
        return 1;
    }

    debug("Compression");
}
```

huffmantree.c

`Node* highest_numbered_node(HuffmanTree* ht, Node* node)`

La funzione `highest_numbered_node` ci ritorna il nodo con `node_number` maggiore (ossia quello più "in alto" nell'albero) dal peso pari al peso del nodo passato come parametro. Sono state effettuate scelte come far partire il ciclo dal risultato dell'espressione `node->node_number+1` e l'utilizzo di un array all'interno dell `HuffmanTree` per agevolare la ricerca. (`time(array_search) < time(recursive_on_tree_search)`)

`void swap_nodes(HuffmanTree* ht, Node* node, Node* node2)`

Effettua lo swap di due nodi, sia nell'albero (scambio fra puntatori) che nell'array (`ht->tree[]`)

`void node_positioner(HuffmanTree* ht, Node* target)`

La funzione `node_positioner` è il core dell'algoritmo ed è quella che ci permette, come da nome, di posizionare il nodo nella posizione corretta all'interno dell'albero. Ricevendo un nodo in entrata va semplicemente a controllare se il nodo corrisponde a quello di `node_number` più alto restituito da `highest_numbered_node`, effettuato il controllo decide se è necessario lo scambio o semplicemente l'incremento del peso.

`void huffman_coding_reset_partial_output(HuffmanTree* ht)`

Reimposta il partial_output

`void huffman_coding_bitcheck(HuffmanTree* ht)`

Controlla se il byte è completato ed agisce di conseguenza.

`void huffman_append_partial_path(HuffmanTree* ht, unsigned short* path, int path_size)`

Crea il byte (lo "crafta") sfruttando il percorso dell'elemento già visto.

`void huffman_append_partial_new_element(HuffmanTree* ht, unsigned short* nyt_path, int path_size, char`

Aggiunge un elemento creando il byte con il percorso del NYT ed gli 8 bit del carattere.

`void huffman_partial_final_conversion(HuffmanTree* ht)`

Termina la codifica di Huffman gestendo il padding finale del file.

`void endHuffman(HuffmanTree* ht)`

La funzione si occupa di terminare la codifica di Huffman, chiamando la funzione `huffman_partial_final_conversion`.

`int is_compressor(HuffmanTree* ht)`

Ritorna 1 se `ht` è in modalità di compressione.

`HuffmanTree* add_new_element(HuffmanTree* ht, char c)`

Aggiunge un elemento all'albero - questa funzione è utilizzata sia in compressione che in decompressione. La funzione in modalità di compressione si occupa anche di chiamare opportunamente le funzioni per la creazione del byte (byte crafting).

`unsigned int get_bit(HuffmanTree* ht)`

La funzione ritorna un bit dal `ht->partial_output` usando la maschera specificata da `ht->mask`, e shiftandola opportunamente ad ogni lettura di bit.

```
/* Used only in decompression mode! */
int decode_byte(HuffmanTree* ht)
```

La funzione si occupa di decodificare un byte durante la decompressione.

`int is_nyt(Node *pNode)`

Verifica se il nodo passato come argomento è il NYT.

`int is_leaf(Node *pNode)`

Verifica se il nodo passato come argomento è una foglia.

`int is_internal_node(Node *pNode)`

Verifica se il nodo passato come argomento è un nodo interno (= non una foglia)

`Node* find_node(HuffmanTree* ht, int c)`

La funzione si occupa di cercare il carattere all'interno del `elements_array`.

`Node* create_nyt(int i)`

La funzione si occupa di creare un NYT con Node Number pari a quello passato per argomento.

`Node* create_node(int node_number, int weight, int element, Node *left, Node *right, Node *parent)`

La funzione si occupa di creare un nodo con il *node number*, *peso*, *elemento*, *nodo di sinistra*, *nodo di destra*, *parent* specificato.

`HuffmanTree* create_huffman_tree()`

La funzione si occupa di inizializzare la struttura dati dell'albero di Huffman (utilizzato sia in compressione che in decompressione).

`void free_node(Node *node)`

La funzione si occupa di liberare il nodo dalla memoria.

`void free_huffman(HuffmanTree *ht)`

La funzione si occupa di liberare le risorse utilizzate dall' Huffman Tree.

`unsigned short* node_path(Node* node, int* length)`

Questa funzione si occupa di convertire il percorso di un nodo in un array di `unsigned short*`, di lunghezza `int* length`.

Chapter 6

Procedure di test e problemi noti

Problemi noti

Efficienza dell'algoritmo

Come menzionato precedentemente, l'algoritmo non sembra essere molto efficiente nel comprimere qualsiasi tipo di file. È quindi sconsigliabile quale general-purpose compressor. Il compressore è però molto efficace in caso di documenti strutturati (come per esempio file SVG / HTML / JSON) oppure in documenti testuali (libri o estratti di pagine web).

~~Compressione "lenta"~~

~~Nonostante il nostro algoritmo possa vantare di una velocità di compressione di circa 1208 kB/s (~1.18 MB/s), questo risultato non è ottimale. Con un maggiore lavoro di code optimization è sicuro possibile sfiorare la soglia di 1.5 MB/s.~~

A seguito di un ottimizzazione del codice, possiamo confermare di riuscire a raggiungere picchi di 3MB/s in compressione e decompressione. La media rimane comunque intorno a 1.5MB/s [1].

Test effettuati

test_debug

Testa se il software è in modalità `DEBUG`.

test_create_huffman_tree

Testa se la funzione `create_huffman_tree` viene eseguita correttamente.

test_swap_ht_array

Testa la funzione `swap_nodes` e confronta il risultato con quello aspettato.

test_get_node_level

Testa la funzione `getNodeLevel` e confronta il risultato con quello aspettato.

[1] Intel Core i7-6700HQ @ 8x 3.5GHz, 32GB RAM, x86_64 Linux 4.14.13-1, Spectre patch applied.

test_simple_swap

Testa la funzione `swap_node` applicandola ad un esempio semplice e confronta il risultato con quello aspettato.

test_swap_nodes

Testa la funzione `swap_node` in tutte le sue funzioni e confronta il risultato con quello aspettato.

test_node_path

Testa la funzione `node_path` e confronta il risultato con quello aspettato.

test_huffman_coding

Testa se la codifica di Huffman viene eseguita correttamente

test_huffman_coding_abracadabra

Il test si assicura che l'albero generato dalla codifica di "abracadabra" corrisponda a quello aspettato.

test_huffman_coding_abcbaaa

Il test si assicura che l'albero generato dalla codifica di "abcbaaa" corrisponda a quello aspettato.

test_huffman_coding_bookkeeper

Il test si assicura che l'albero generato dalla codifica di "bookkeeper" corrisponda a quello aspettato.

test_huffman_coding_mississippi

Il test si assicura che l'albero generato dalla codifica di "mississippi" corrisponda a quello aspettato.

test_huffman_coding_engineering

Il test si assicura che l'albero generato dalla codifica di "engineering" corrisponda a quello aspettato.

test_huffman_coding_foobar

Il test si assicura che l'albero generato dalla codifica di "foobar" corrisponda a quello aspettato.

test_huffman_coding_aardvark

Il test si assicura che l'albero generato dalla codifica di "aardvark" corrisponda a quello aspettato.

test_huffman_coding_sleeplessness

Il test si assicura che l'albero generato dalla codifica di "sleeplessness" corrisponda a quello aspettato.

test_bin2byte

Testa la funzione bin2byte che converte una stringa di 0 ed 1 in un byte.

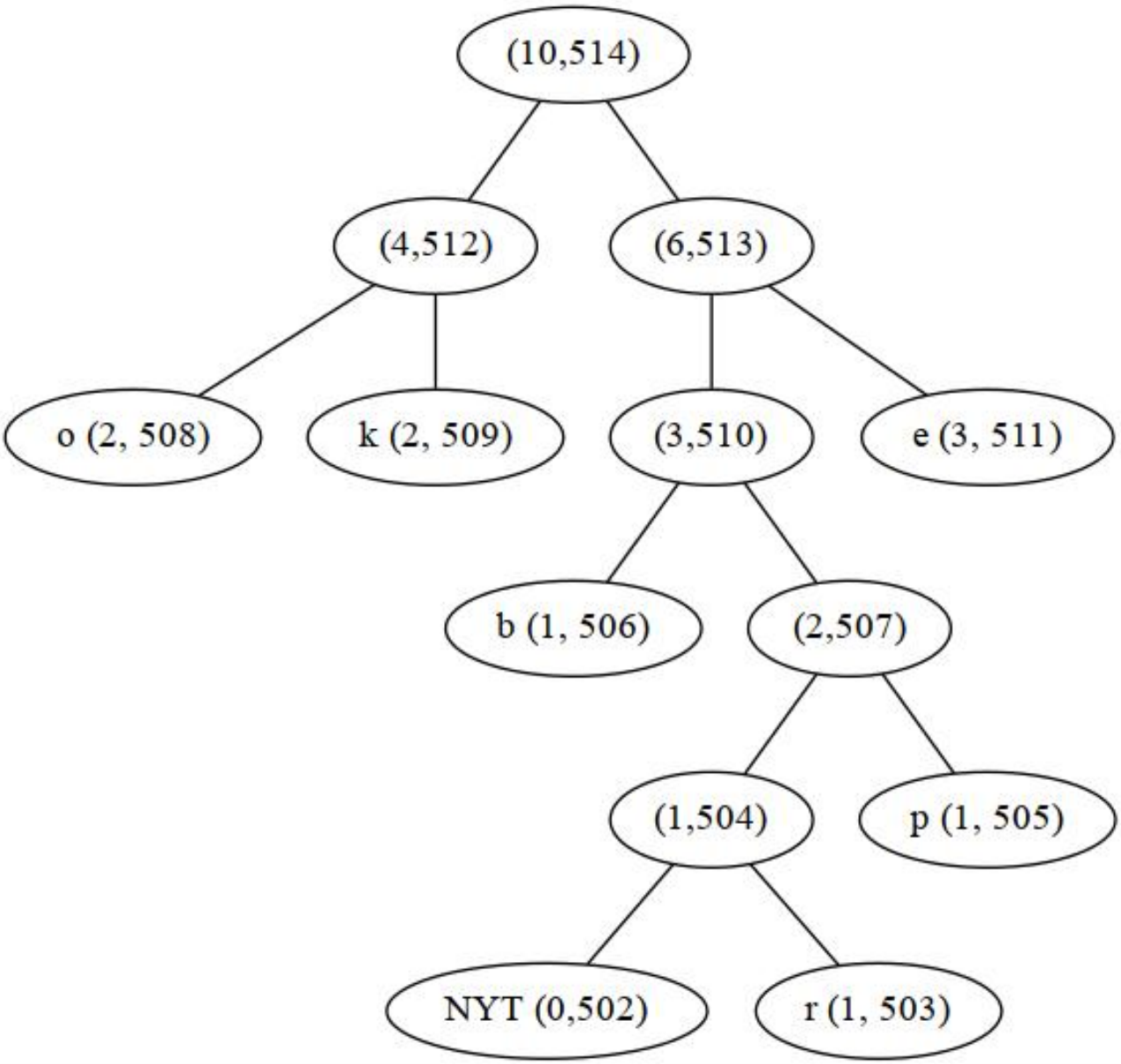

Figure 6.1: Albero risultante per “bookkeeper”

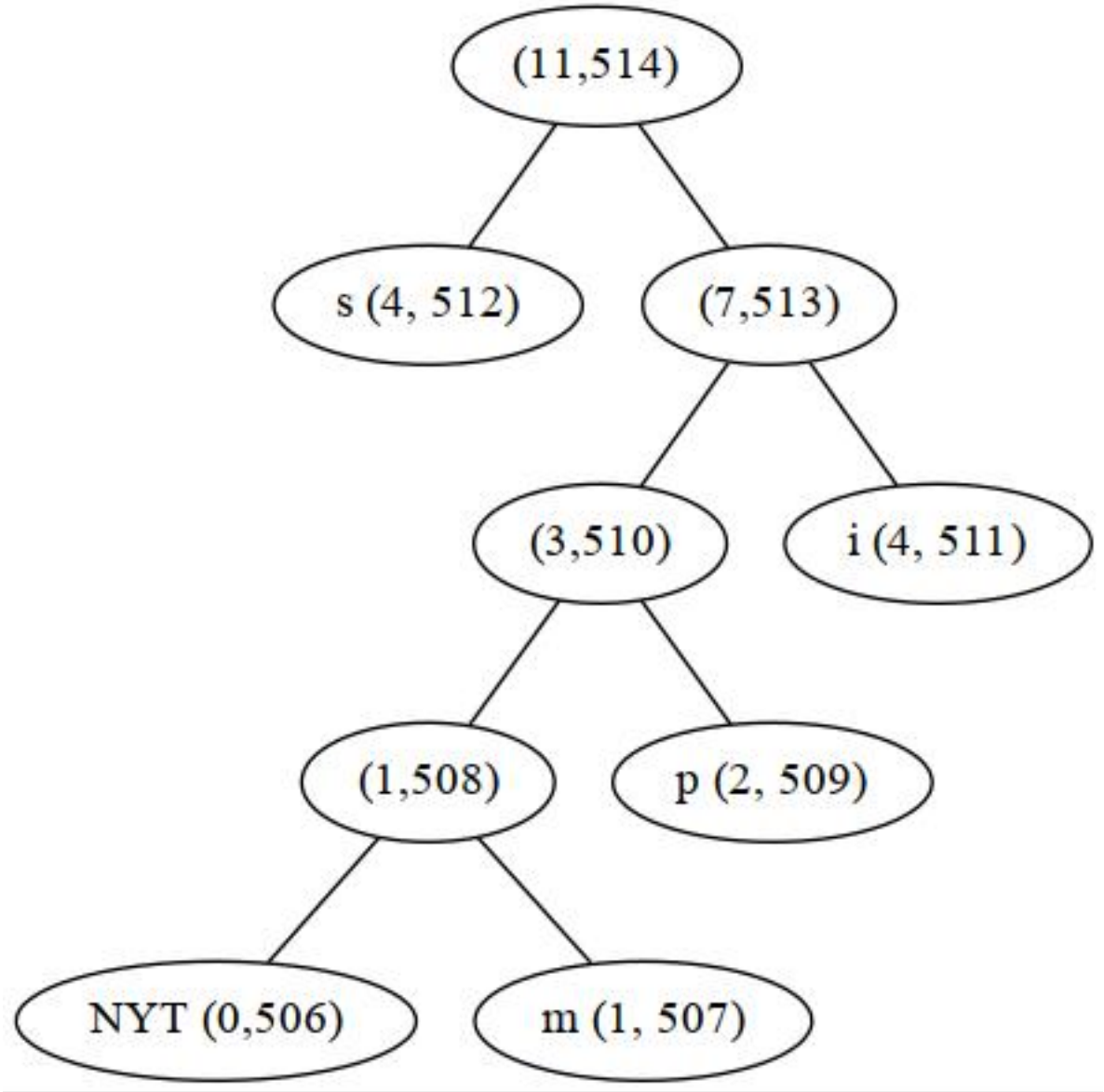

Figure 6.2: Albero risultante per “mississippi”

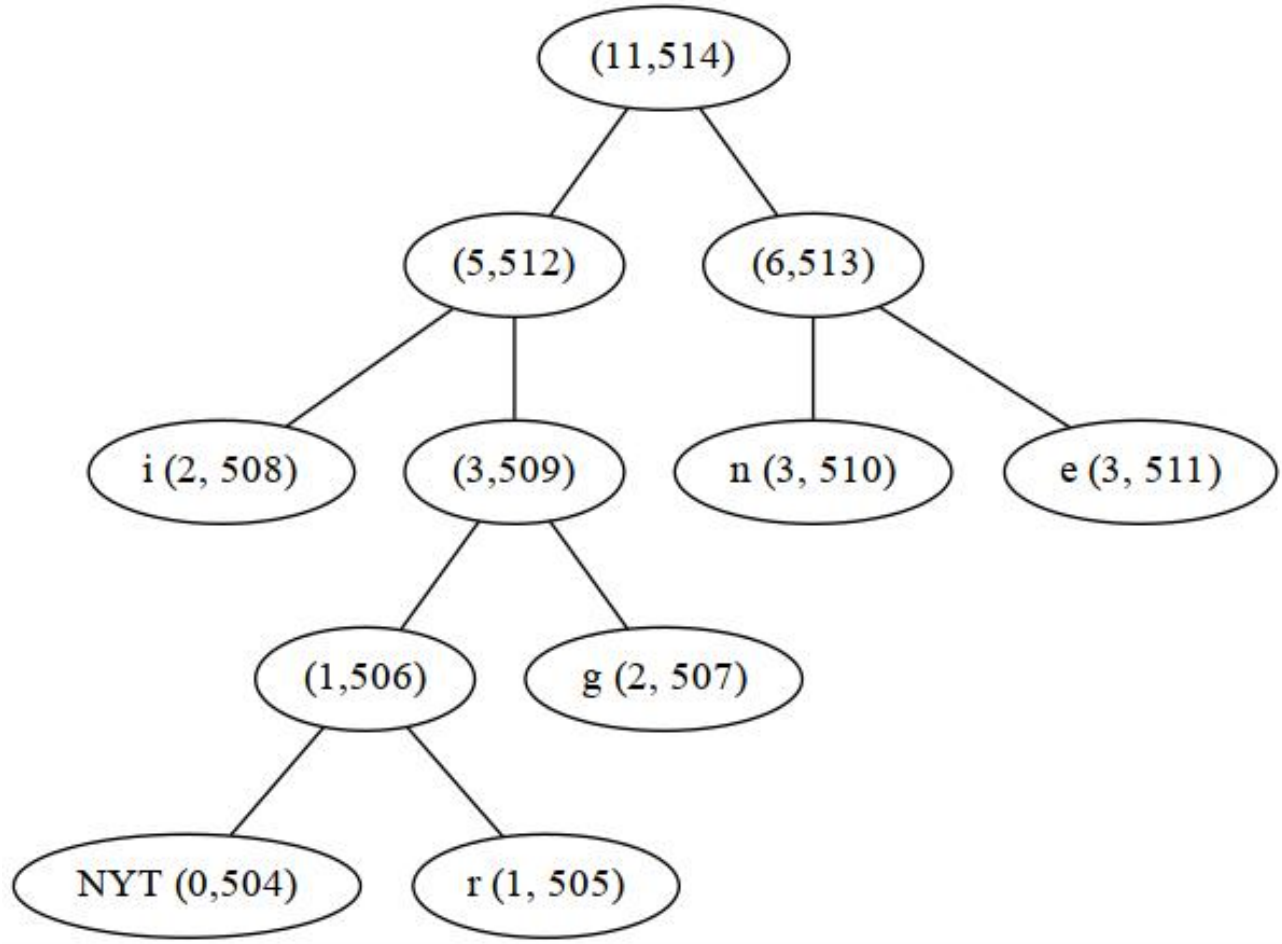

Figure 6.3: Albero risultante per “engineering”

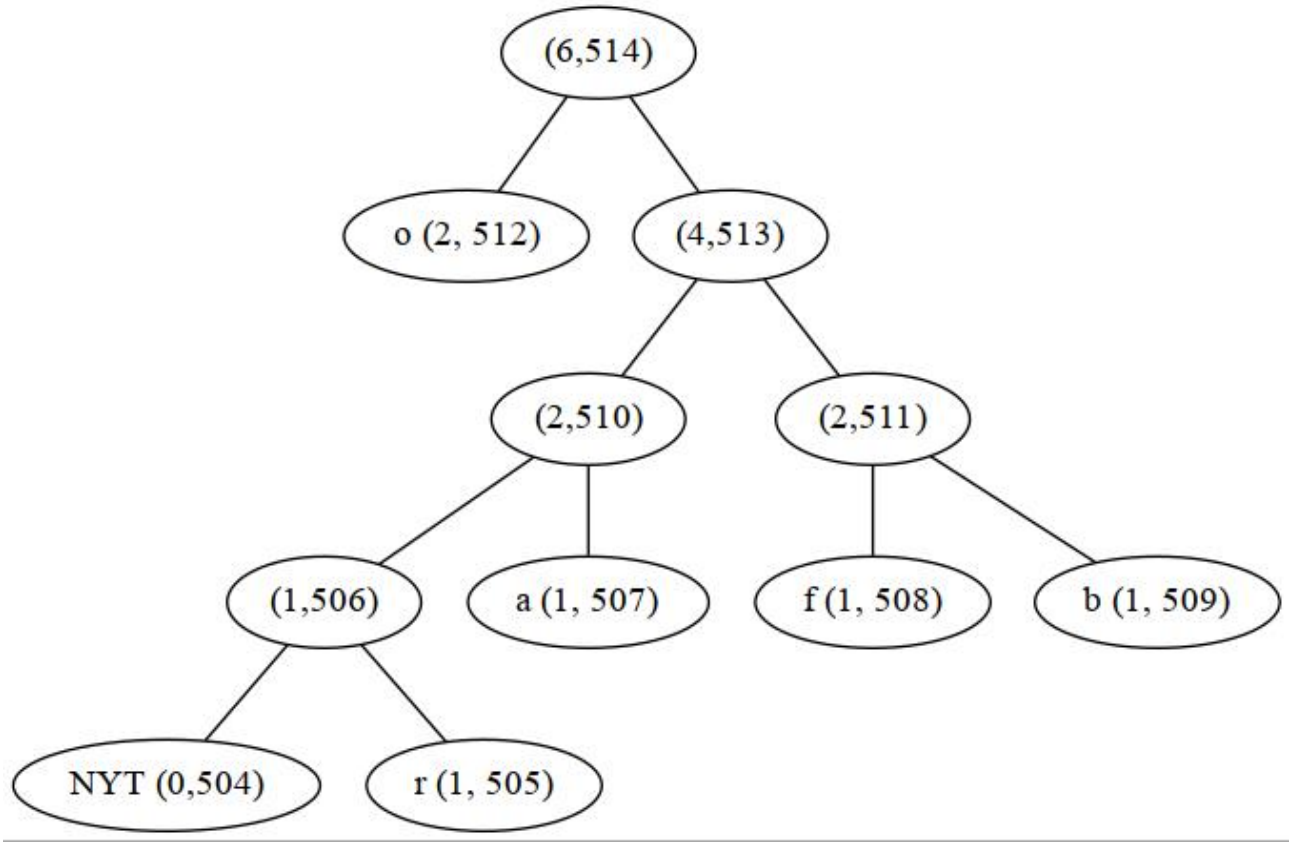

Figure 6.4: Albero risultante per “foobar”

test_bin2byte2

Estende il test di bin2byte per casi più complessi.

test_byte2bin

Testa la funzione byte2bin che converte dei byte in una stringa di 0 ed 1.

test_filename

Testa la funzione `get_filename(char* filepath)`.

test_create_file

Testa la funzione di creazione di file

test_write_to_file

Testa la funzionalità di scrittura su file

test_read_file

Testa la lettura da file

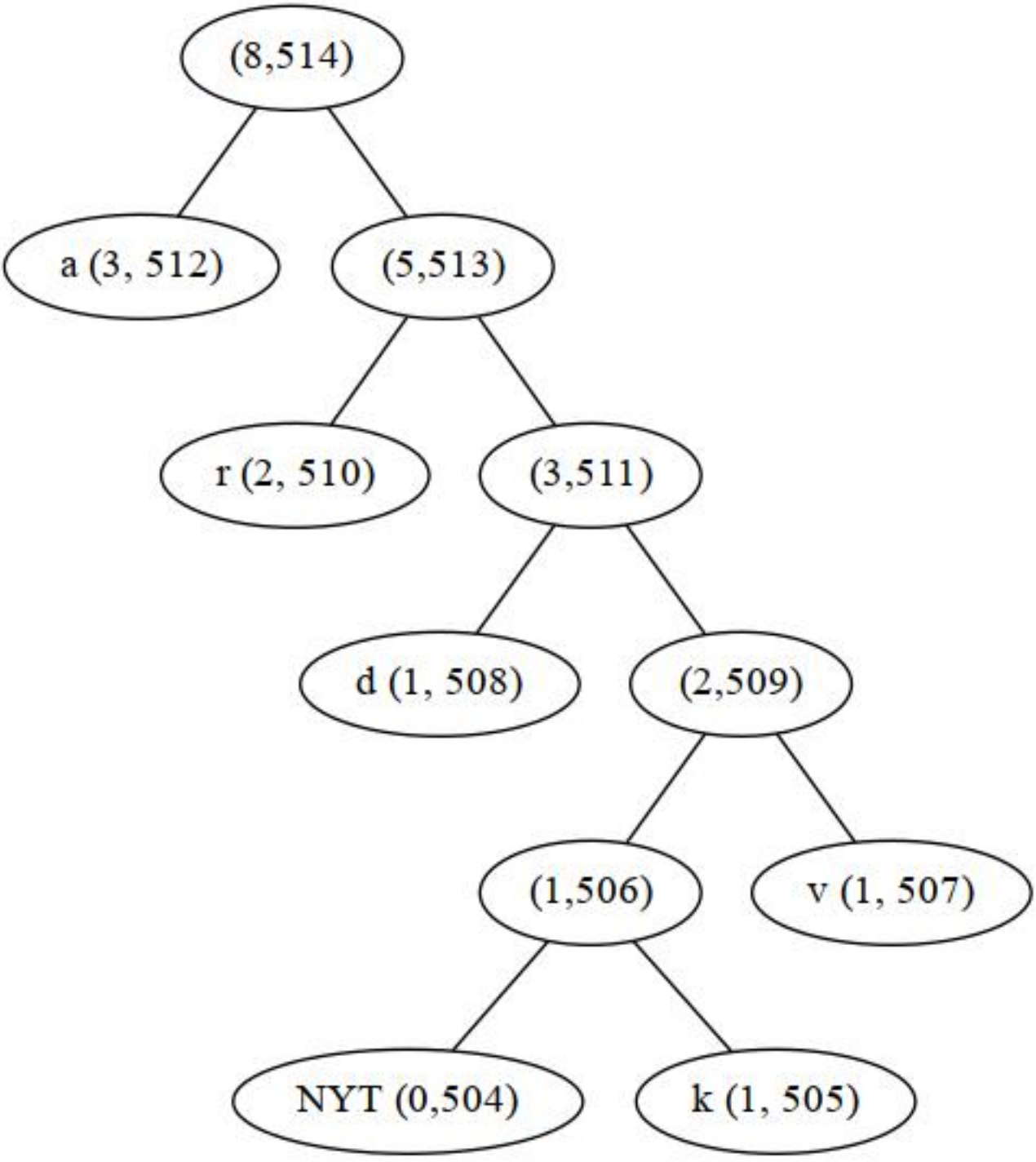

Figure 6.5: Albero risultante per “aardvark”

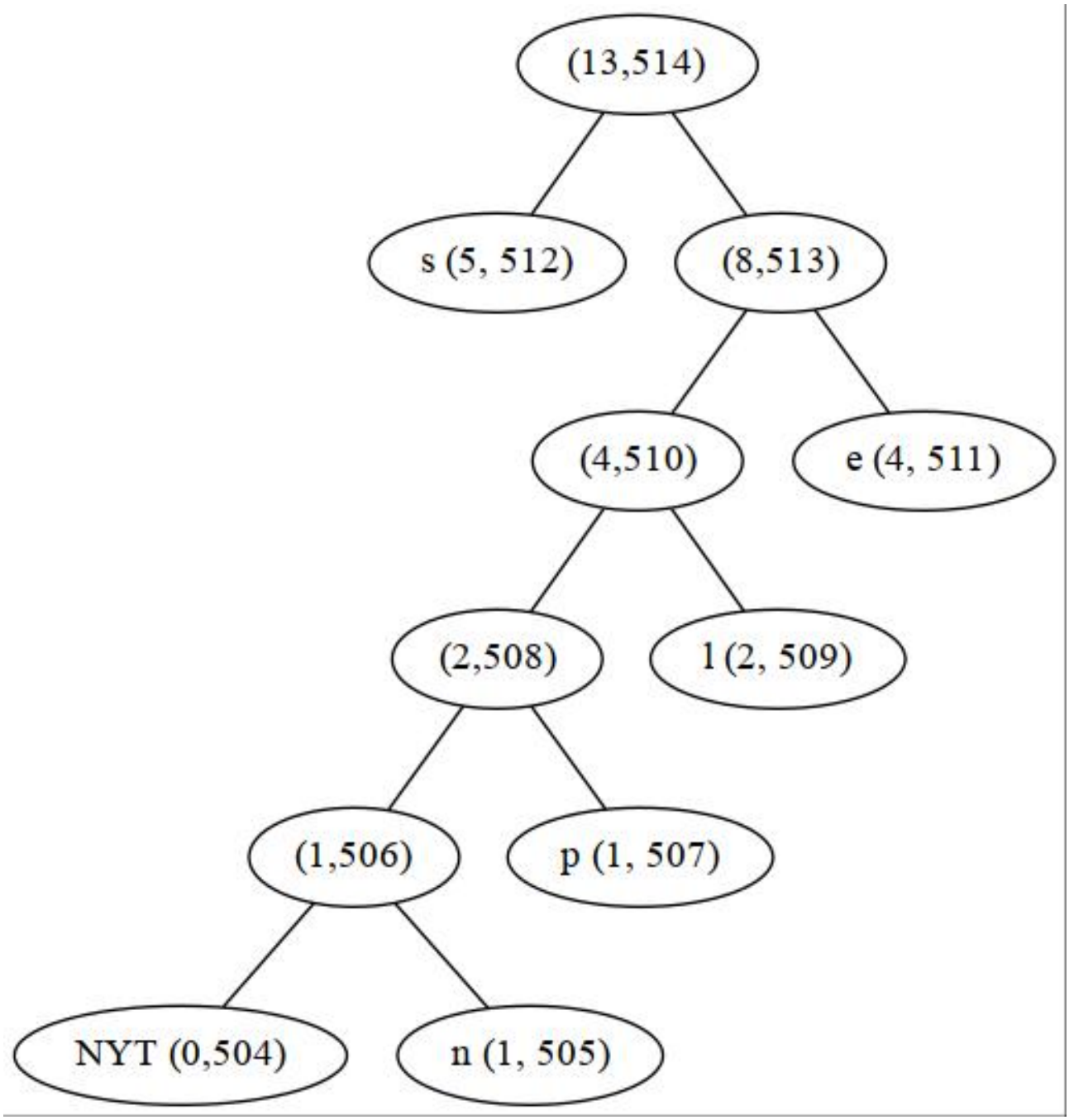

Figure 6.6: sleeplessness

test_file_delete

Testa la funzionalità di eliminazione di file

Chapter 7

Conclusione

Questo progetto ci ha permesso di comprendere al meglio le difficoltà tecniche che possono insorgere durante lo sviluppo di un progetto come quello affrontato. Una gestione efficiente della memoria, l'ottimizzazione del codice ed il debugging intenso ci hanno permesso di raggiungere risultati decenti in termini di velocità di compressione e decompressione.

Il progetto ci ha posto di fronte ad innumerevoli difficoltà che inizialmente non avevamo calcolato. Le nostre abilità di ragionamento e di logica sono state messe alla dura prova.

Durante lo sviluppo del progetto abbiamo anche imparato a lavorare con numerosi strumenti come `git`, `valgrind` (+ memcheck, massif, callgrind), `gdb`, `gcc`, `Makefile`s, Jenkins, Pandoc, Dot, Python, Bash, LaTeX.

Ringraziamenti

Ringraziamo David Huber e Nicola Vermes per i consigli e gli aiuti forniti durante le lezioni di laboratorio.
In modo particolare ringraziamo David Huber per il suo magnifico consiglio nell'uso di Valgrind - lo strumento ci è infatti stato di grandissimo aiuto durante tutto lo sviluppo e ci ha risparmiato molte ore di debugging.

Riferimenti

viz-fgk-compressor on GitHub
viz-fgk-compressor CI on Mastodontico's Jenkins Instance
Presentazione 1
Presentazione 2

Risorse

Adaptive Huffman Coding - FGK - Stringology.org
Adaptive Huffman Coding - Wikipedia
Adaptive Huffman Coding - The Data Compression Guide
Adaptive Huffman Coding - cs.duke.edu
Visualizing Adaptive Huffman Coding - Ben Tanen
Array Implementation for Complete Binary Trees

Enterprise Waterfall