UNA RAPIDA ANALISI DEI PROGRAMMI E DEGLI ALGORITMI PER LA COMPRESSIONE DEI TESTI

Andrea Schiavinato

10 marzo 2008

Breve presentazione del lavoro

In questo lavoro verranno provati i quattro metodi per la compressione dei dati presentati nel corso di Teoria dell'Informazione: il metodo di Huffman, l'algoritmo "LZ" di Lempel e Ziv (chiamato anche LZ77), l'algoritmo LZW (Lempel, Ziv e Welch) ed infine di quello di Burrows e Weeler. Le prove consisteranno nel vedere quanto comprime ogni algoritmo ma, piuttosto che adottare un approccio "agonistico" e di stilare una classifica degli algoritmi in base al tasso di compressione che riescono a raggiungere, cercherò di trovare pregi e difetti di ognuno di loro. Nonostante il tasso di compressione rimanga il parametro fondamentale, se opportuno considererò anche altre caratteristiche degli algoritmi come la velocità o la memoria consumata, senza però la pretesa di essere molto precisi. Poi, alla fine, organizzerò quanto ottenuto in una tabella e cercherò di individuare, se esistono, delle condizioni che rendono un metodo più appropriato di un altro.

Gli algoritmi verranno provati solo su testi, ovvero file contenenti caratteri che una persona può interpretare in qualche modo semplicemente aprendoli con un editor di testo. Non è una cosa molto restrittiva, nel senso che il non considerare immagini o contenuti di altro genere renda questo lavoro incompleto. In molte applicazioni vi è la necessità di trattare solo informazioni di tipo testuale, inoltre per la compressione di immagini o altre informazioni di tipo multimediale vi sono dei metodo appositi. Questa limitazione ci da l'opportunità di fare un'analisi abbastanza approfondita in questo dominio, nonostante questo lavoro non abbia molte pretese. Cercherò di individuare delle categorie di testi e poi andrò a testare gli algoritmi su un rappresentate di ogni categoria. Si potrà vedere se alcuni compressori sono più adatti ad alcune categorie oppure, cosa altrettanto interessante, individuare delle categorie di testi più facili da comprimere. Io, ad esempio, credo che un file contenente codice Java sia più facile da comprimere di un testo normale, in quanto contiene più parole ripetute.

All'inizio verrà effettuata un'amplia ma rapida e non esaustiva ricerca degli algoritmi, programmi per la compressione e test simili presenti. Lo scopo principale è quello di rispondere alla domanda "quali algoritmi è meglio implementare e per quali possono essere usate delle implementazioni di altri?" Alcuni sono già implementati in maniera efficiente in programmi o librerie che è facile usare. Un esempio significativo è l'algoritmo LZW implementato nel programma compress, che ora è possibile usare liberamente [1]. Qualora decida di implementare un algoritmo cercherò di farlo in modo da creare un programma orientato più alla semplicità ed alla comprensibilità che all'efficienza. Un altro scopo di questa ricerca iniziale è scoprire che algoritmi nuovi ci sono, se ci sono delle varianti e se c'è un programma più efficiente che potrebbe essere utilizzato come termine di paragone.

[1] Fino all'anno 2003 l'algoritmo LZW era protetto da un brevetto

Questo lavoro è organizzato nei seguenti punti:

1. Per prima cosa svolgerò un rapida analisi degli algoritmi, dei programmi di compressione e dei test simili presenti e stabilirò quali algoritmi implementare.

2. Sceglierò i testi da utilizzare come input agli algoritmi e definirò in maniera precisa la struttura dei test

3. Eseguirò dei test per l'algoritmo di Huffman, e succeccivamente per l'algoritmo LZ, LZW e Burrows-Weeler. Infine ripeterò il test con il programma PAQ, che usa un metodo di compressione differente.

4. Confronterò gli algoritmi e discuterò i risultati principali.

Varianti degli algoritmi noti, nuovi algoritmi e programmi per la compressione

La mia impressione è che nel campo dei compressori general-porpouse o di quelli orientati alla compressione dei testi attualmente non vi sia un grandissimo interesse. Nonostante ciò esiste una grande quantità di varianti degli algoritmi citati e qualche nuovo metodo di compressione. Alcune varianti sono molto usate e vengono impiegate nei programmi di compressione destinati ad un uso comune, altre per ora non vengono molto usate nella pratica. Di seguito fornirò un'analisi molto veloce di ciò che c'è in giro, alcuni aspetti relativi ai programmi scelti per svolgere i test verranno approfonditi più avanti.

Il metodo di Huffman

Non è quello più efficiente ma è sicuramente quello più citato e più conosciuto. Ho visto che esistono numerose piccole varianti (come l'algoritmo che produce codici canonici o codici di lunghezza massima prefissata) che spesso vengono impiegate assieme ad altri algoritmi. Spesso, infatti, la codifica di Huffman rappresenta il passo finale di un programma di compressione e viene utilizzata per rappresentare in modo ancor più efficace l'output di altri algoritmi. Ho deciso di implementarlo perché è un algoritmo semplice e perché non ci sono programmi standard in circolazione basati su questo metodo. In questo modo c'è anche la possibilità di capire meglio le parti che possono essere implementate in maniera diversa e più efficiente. Per l'implementazione userò il linguaggio Java, perché per conto mio permette di scrivere programmi più adatti alla riusabilità, all'analisi ed alla modifica di C. Inoltre, come già detto, non è l'obiettivo di questo lavoro analizzare gli algoritmi per la loro efficienza.

LZ

Anche l'algoritmo di Lempel e Ziv, al quale si fa riferimento con i nomi LZ o LZ77, ha avuto un ruolo molto importante nell'ambito della compressione dei dati: è quello sul quale si basa il diffusissimo formato di file ZIP nato assieme al programma PKZIP. Questo non si basa sull'algoritmo LZ standard, ma in una sua variante chiamata "Deflate". Essa viene usata anche nel programma GZip, anche questo molto famoso, e nel formato di immagini PNG. Oltre a Deflate esistono numerose altre varianti usate nella pratica, come LZX, usato in alcune formati di file della Microsoft oppure LZMA, usato in 7Zip. Dato il grande numero di varianti e di programmi che fanno uso di questo algoritmo verranno scelti due delegati per questa categoria: GZip e WinRAR, che rappresentano anche i due formati più usati per la compressione (rispettivamente ZIP e RAR). In particolare i risultati ottenuti da WinRAR verranno utilizzati in ogni prova come valori di riferimento, per permettere un confronto più semplice fra le prestazioni dei diversi algoritmi. Verrà anche implementata una versione standard dell'algoritmo per comprenderne meglio alcune caratteristiche e per confrontarla con le implementazioni commerciali.

LZW

Si tratta di un'implementazione dell'algoritmo LZ88, pubblicato dalla coppia Lempel e Ziv l'anno dopo la pubblicazione del loro precedente lavoro. É stato implementato nel programma compress e nel programma ARC[1], ma attualmente non viene molto usato a causa di ragioni legali e tecniche[2]. Nonostante ciò compress è facilmente reperibile in Internet e verrà quindi usato nelle prove. Due usi rilevanti di LZW sono nel formato di immagini GIF e nel programma Adobe Acrobat.

Burrows-Wheeler

Nemmeno questo metodo ha avuto un successo paragonabile a quello di LZ, tuttavia viene correntemente usato nel programma commerciale Bzip2 (con l'aggiunta di varie ottimizzazioni). Un professore universitario, Matt Mahoney, che svolge anche delle ricerche nel campo della compressione dei dati, ha sviluppato un programma molto efficiente che implementa questo algoritmo ed egli sostiene che offra prestazioni superiori rispetto a BZip2. Dato che quest'ultimo programma è molto ben fatto verrà usato come rappresentante assieme a BZip2. Altri programmi che fanno uso di questo algoritmo ma che non verranno considerati sono GRZipII, sbc e dark.

Gli altri:

Esistono poi una serie di altri algoritmi per la compressione sui quali non ho indagato più di tanto ma che usano approcci differenti (e che si possono combinare). Uno molto importante si chiama "Prediction by Partial Matching" e consiste nel manterere un modello statistico del testo con lo scopo di assegnare una probabilità al prossimo simbolo in input. Questo, e la sua probabiltà, verranno poi codificati usando la tecnica "Aritmetic Coding", che consiste nell'associare un numero in virgola mobile all'intero file, costruito combinando le probabilità dei simboli letti in modo da richiedere meno bit per essere rappresentato se le probabilità sono alte. Un'altra tecnica di cui fornirò solo il nome ma che vale la pena citare è la "Context Tree Weighting".

Questi ultimi metodi permettono di comprimere maggiormente rispetto agli approcci visti sopra, ed esiste un programma per la compressione genral-porpouse abbastanza famoso, PAQ, che offre prestazioni considerevoli: compare sempre in prima posizione nei test sulla compressione presenti in Internet. Lo svantaggio, che ne diminuisce un po' l'utilità, è che si tratta di un programma molto esigente in termini di memoria e di tempo di calcolo. I testi verranno compressi anche con PAQ.

Altri test sugli algoritmi di compressione

Ho trovato anche un sacco di altri test che hanno come oggetto la comparazione delle prestazioni degli algoritmi di compressione. Sono tutti più o meno simili, si differiscono più che altro per la varietà di materiali compressi, il numero di programmi testati, il numero di metriche adottate (alcuni considerano solo il rapporto di compressione, altri anche la velocità e la memoria consumata) e la frequenza con cui vengono aggiornati. Di seguito riporto quelli che ho ritenuto più interessanti

Large Text Compression Benchmark (Matt Mahoney)

Si tratta di un test completo dal punto di vista dei programmi testati e molto aggiornato. E focalizzato sulla compressone dei testi: vengono compressi i primi 100 o 1000 MB di Wikipedia sotto forma di un unico file XML. Come metrica per la valutazione viene considerata la dimensione del file compresso assieme alla dimensione del programma necessario alla decompressione. Per ogni programma viene specificato anche il tempo impiegato per la compressione e la decompressione, la memoria usata ed alcune note che indicano fra le altre cose l'algoritmo usato. La motivazione che sta alla base del test è quella di incoraggiare le ricerche nel campo del'intelligenza artificiale e dell'elaborazione del linguaggio naturale: viene detto che la compressione dei testi sia strettamente legata a queste discipline. Questo test è molto affine a

"The Hutter Prize" (50'000 euro Prize for Compressing Human Knowledge) [3], nel quale viene messo in palio un premio in denaro a chi propone un programma più efficace per la compressione dei primi 100 MB di Wikipedia di quello attualmente in prima posizione. Si trova in [4].

Squxe Archivers Chart

I risultati si possono trovare all'indirizzio [5] e a mio avviso si tratta di un test semplice ma ben fatto. Vengono provati vari programmi su diversi tipi di dati (testi, file eseguibili, immagini, eccetera) e viene misurato il rapporto di compressione e le velocità di compressione e di decompressione (in megabyte al secondo). I risultato sono abbastanza aggiornati ma non come nel test di Mahoney: la pagina che si trova alla data attuale e stata aggiornata nel mese di novembre 2007, mentre quelle di Matt solo qualche giorno fa. Questi sono però proposti un una forma molto pulita e comprensibile . E' proprio questo il motivo per il quale ritengo opportuno citare questo test: si tratta di quello presentato in forma migliore.

Squeeze Chart - Compressing Text in Different Languages (Stephan Busch)

Si tratta di un test più particolare degli altri: si comprime lo stesso testo (la bibbia) in varie lingue. E interessante perché è il primo test che considera le differenze fra le lingue. E'anche molto completo, in quanto sono considerate ben 33 lingue! Si osserva che la bibbia in greco ed in thai (una lingua parlata in Tailandia e nel nord della Malesia) sono quelle compresse con un fattore più elevato, mentre quella compressa meno è quella in croato. La cosa interessante è che questi risultati sembrano indipendenti dal programma per la compressione usato. Le pecche di questo test a mio avviso sono la presentazione dei risultati, un po' confusa e difficile da capire, e l'insieme dei programmi provati, che non è molto completo Mancano ad esempio gzip e compress che secondo me sono dei importanti termini di paragone. Si trova all'url [6].

Archive Comparison Test (ACT)

Un test interessante in quanto contiene molti dati, ma sembra sia stato abbandonato: l'ultimo aggiornamento è avvenuto nell'anno 2002, dieci anni dopo la sua nascita. I test sono fatti bene, presentati in forma chiara e completa, sono stati testati molti programmi e per ogni programma viene riportata un gran quantità di informazioni. L'url è [7].

Una cosa comune ai vari test è che il metodo che risulta più efficiente per la compressione è quello "Prediction by Partial Matching", che ottiene

prestazioni del 4-5% superiori rispetto a WinRAR (con prestazioni intendo il rapporto di compressione).

La struttura del test

Come già detto i test che verranno svolti consistono nel comprimere file di testo mediante diversi programmi di compressione, con lo scopo di vedere quali comprimano maggiormente in funzione dell'input. Con file di testo faccio riferimento in generale a qualsiasi file contenenti caratteri (che siamo in grado di elaborare con un editor di testo) e non solo a testi scritti in una lingua parlata. Nelle prove considererò quindi testi in lingua italiana di vario genere, programmi espressi in un linguaggio di programmazione imperativo (Java) ed in uno dichiarativo (Prolog), oltre ad una pagina web (sempre in italiano), un file di log ed il file XML che contiene i primi 100 MB di Wikipedia usato nel "Hutter Prize". Questo potrà essere usato come "collegamento" fra il mio test ed il test di Mahoney e per provare gli algoritmi che implementerò anche su di un file di dimensioni considerevoli. L'elenco dei file scelti si trova nella tabella 1

Sebbene sia sufficiente usare la dimensione dei file compressi come metrica di confronto, userò anche un valore da 0 a 1 per avere un'indicazione più immediata sulla capacità di compressione, calcolato come: $min(1-$dimensione file compresso/dimensione file da comprimere$, 0)$ e che chiamerò "indice di prestazione". Questo indica la percentuale di bytes di un file tolti dal processo di compressione, quindi a valori più prossimi a uno corrispondono risultati migliori. I risultati verranno riportati in forma tabellare ed in forma grafica e in ogni prova verranno affiancati a quelli ottenuti con WinRAR, che verrà quindi usato come termine di confronto. Proporrò poi un breve commento e se ci sarà l'occasione analizzerò alcune caratteristiche degli algoritmi diverse dalla semplice capacità di compressione, anche se in maniera informale e non troppo precisa. Nell'ultima parte di questo lavoro, come già detto, ci sarà una tabella che riassume i risultati ottenuti.

Il computer usato per le prove dispone di un processore Intel Core Duo 2 Ghz, di 2GB di RAM, Windows Vista e Linux Ubuntu 7.10.

L'algoritmo di Huffman

Alcune caratteristiche

L'algoritmo di Huffman è un algoritmo Blocco-Lunghezza variabile. Richiede due scansioni dell'input per comprimerlo: la prima per ricavare delle informazioni sui simboli usati che verranno usare per la costruzione di un codice, la seconda per la codifica vera a propria. Il processo di codifi-

Descrizione/Fonte/Note	Dimensione (byte)
Testo generico in inglese Il file enwik8, usato anche nel “Hutter Prize”	100.000.000
Legge italiana www.parlamento.it Legge 31 dicembre 1996, n. 675	66.592
Testo tecnico La tesi che ho fatto per la laurea triennale (in formato latex)	186.297
Testo letterario www.e-text.it Alessandro Manzoni, tutte le poesie	200.689
Testo per bambini www.lagirandola.it La favola di Cappuccetto Rosso	6.932
Programma JAVA Il file soregente della classe Collections, tratto dale API standard di Java	139.175
Programma PROLOG www.cs.ccu.edu.tw/ dan/prologProgs.html Un piccolo sistema esperto	8.460
Pagina HTML www.libero.it La home page del portale libero, solo il codice HTML	92.537
Messaggio su un forum con risposte forum.kataweb.it Una discussione nell’ambito dell’arte	12.031
Articolo generico tratto da sito web www.gazzetta.it Un articolo di carattere sportivo	2.929
File di log Un file di log generato da Windows Vista	195.092

Tabella 1: Elenco dei file usati per il test

ca/decodifica è molto veloce, però è richiesta la presenza di un "dizionario" dei codici.

Le varianti

Esistono molte modificazioni di questo algoritmo, alcune delle quali molto citate ed usate praticamente. Si può, per esempio, definire un algoritmo che genera solo codici canonici, ovvero con queste caratteristiche:

1. Tutte le parole di codice di una determinata lunghezza, se ordinate in base al simbolo che rappresentano, hanno valori consecutivi (nel senso lessicografico)

2. Le parole di codice più corte precedono lessicograficamente quelle più lunghe (ad esempio 010, 0010, 0011 va bene)

Una possibile vantaggio di un codice di questo tipo è che può essere rappresentato fornendo solo una lista di lunghezze. Esistono anche algoritmi per la generazione di codici di Huffman con una lunghezza massima prefissata, algoritmi adattativi, che non richiedono due passate dell'input ma adattano un albero di codifica iniziale, fissato a priori, in base ai dati correnti, man mano che vengono letti. La versione adattativa ha il grande vantaggio che può essere usata anche per stream di dati di lunghezza non nota a priori (ad esempio dati letti da una tastiera). Un'altra piccola variante velocizza la decompressione usando delle tabelle di lookup, al posto dell'albero di decodifica.

I programmi

I primi programmi per la compressione si basavano proprio sull'algoritmo di Huffman: compact è stato uno dei primi compressori per Unix ed era basato sulla versione adattativa, SQ era un programma per DOS che usava la versione "normale". Le prestazioni dei due programmi, nonostante questa differenza, erano molto simili [1]. Non è stato possibile recuperare nessuno dei due programmi per confrontarne le prestazioni con i loro successori.

Implemetazione

Ho codificato l'algoritmo di Huffman in Java, cercando di creare del codice facile da capire, riutilizzabile e modificabile. Per questi obiettivi e per la non necessità di ottimizzare l'uso delle risorse, ho implementato la versione standard dell'algoritmo, senza le ottimizzazioni presentate quando ho parlato delle varianti. Ora descriverò brevemente l'organizzazione logica del programma, il formato dell'output prodotto, le strutture dati usate ed i principali metodi.

Il programma (che ho chiamato "ComprimiH"), è ripartito in cinque classi. Tre classi sono di carattere generale e possono essere riutilizzate in altri progetti: BitsInputStream e BitsOuputStrem, che permettono di scrivere una quantità desiderata di bit in un file, Code che rappresenta una parola di codice (ovvero una sequenza lunga a piacere di bit). Le rimanenti due classi (ComprimiH e DecompirmiH) sono dedicate una alla compressione di un file ed una alla decompressione. Esse contengono il metodo Main e possono quindi essere eseguite nel seguente modo: "$javaComprimiH < nomedelfiledacomprimere > \{v|d|vd\}$" oppure "$javaDecomprimiH < nomedelfiledadecomprimere > \{v|d|vd\}$". Quindi se volete più informazioni a video usate l'opzione "v" e se volete vedere anche il dizionario generato usate l'opzione "d".

Il file prodotto in output da ComprimiH contiene nei primi 4 byte la lunghezza del file originale (in byte), seguito dal dizionario (una sequenza di elementi di lunghezza variabile, contenenti 8 bit per indicare il carattere al quale viene associato il codice, 8 bit per indicare la lunghezza del codice ed i bit del codice), e dal file compresso.

Le strutture dati usate nel processo di compressione sono:

- Una tabella hash F1 che contiene le frequenze dei simboli del file considerato. I valori memorizzati sono di tipo int gli indici di tipo char. Questa ovviamente viene deallocata una volta creato il codice. Ho usato una tabella Hash, e non un semplice array, perché in questo modo posso gestire anche i caratteri unicode rappresentati con sedici bit, senza usare troppa memoria, inoltre posso usare direttamente dati di tipo char come indici.

- Una PriorityQueue F usata nel processo di costruzione del codice. I suoi nodi sono del tipo (carattere, filgio1, figlio2, frequenza) e verrà usata per la costruzione di un albero che verrà usato poi per la generazione del codice. La PriorityQueue è una struttura che fa parte delle API standard di Java ed implementa uno Heap.

- Un'altra tabella hash che rappresenta il dizionario che associa a dati di tipo char un elemento di tipo Code

Tutte hanno una visibilità globale.

Nel processo di decompressione, invece, viene invece usato un albero binario (chiamato codes) i cui elementi sono triple (up, down, char). Up e Down, come si può intuire, sono dei collegamenti ai figli. Ho deciso di usare questo nome per rievocare l'aspetto visivo di un albero di decodifica, nei quali i due figli di un nodo vengono collocati a in alto a sinistra ed in basso a sinistra rispetto al nodo padre.

Le funzioni usate per la compressione e decompressione vengono descritte rispettivamente nelle figure 1 e 2.

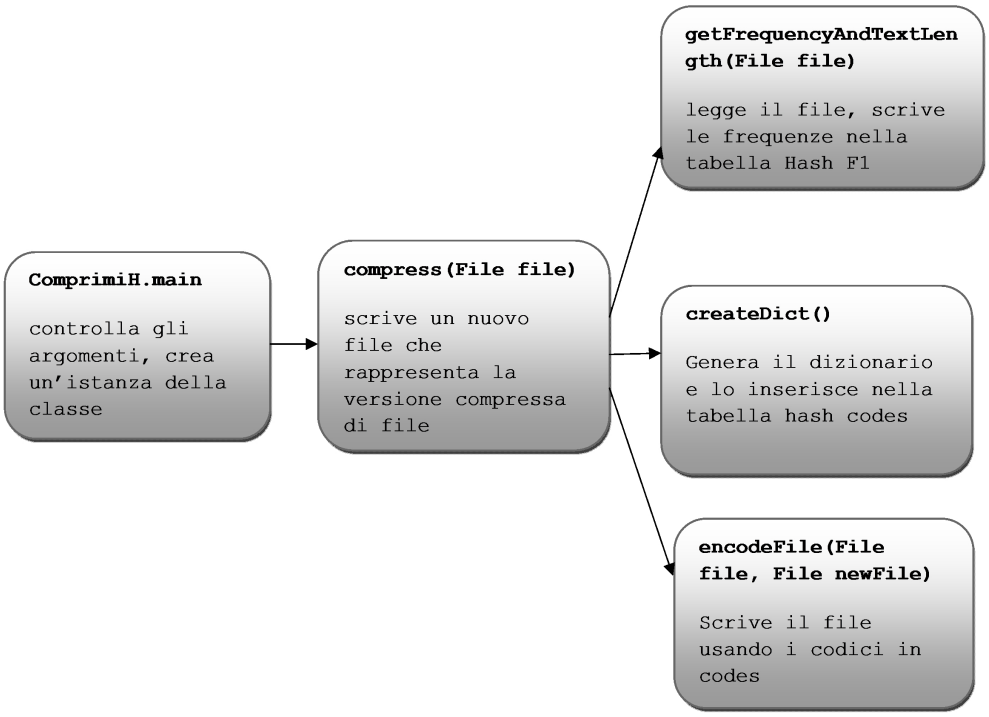

Figura 1: Le funzioni usate nella classe ComprimiH.java per codificare un file mediante l'algoritmo di Huffman

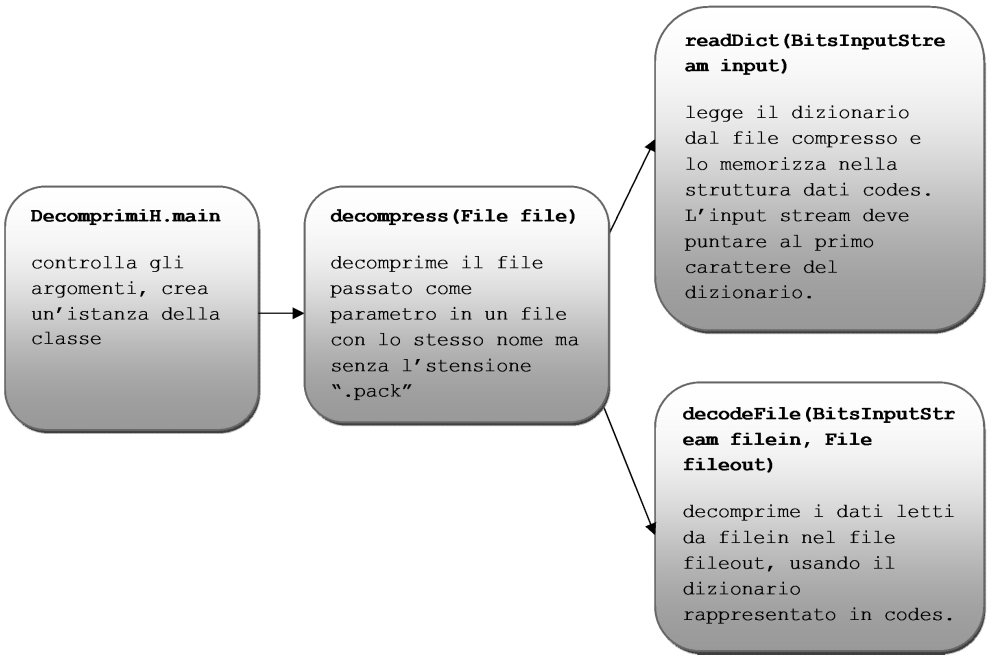

Figura 2: Le funzioni usate in DecomprimiH.java per decomprimere un file compresso con il metodo di Huffman

I risultati

In tabella 2 sono riportati i risultati ottenuti da questo algoritmo. Nel grafico in figura 3 viene proposta una loro rappresentazione grafica ed è anche possibile confrontare i risultati con le prestazione ottenute da WinRar, il programma scelto come termine di paragone in quanto usato piuttosto comunemente. Per quanto riguarda la velocità di esecuzione ci sono voluti due minuti per comprimere il file enwik8 (122 secondi per la precisione), contro i 49 secondi richiesti da WinRar.

La mia variante

Dall'idea di avere un dizionario al quale si associa un codice ad ogni lettera in base alla frequenza con la quale è usata, viene abbastanza naturale pensare di assegnare un codice anche alle parole più frequenti. Infatti i testi, in generale, hanno la proprietà di essere suddivisi in parole che sono riconoscibili a livello sintattico e in ogni testo c'è sempre un certo numero di parole ripetute. Nei file XML ci sono i tag, ai quali si potrebbe assegnare un codice, in un programma Java ci saranno le parole chiave, gli identificatori di variabili, i nomi di funzioni, in un testo in italiano ci saranno gli articoli e le preposizioni. Un'altra considerazione è che in una lingua, generalmente, le

	Dim. tot. file compr.	Dizionario	Indice
Testo generico in inglese	63.855.440	708	0,361
Legge italiana	36.735	211	0,448
Testo tecnico	109.377	331	0,413
Testo letterario	109.415	317	0,455
Testo per bambini	4.048	174	0,416
Programma JAVA	7.9561	287	0,428
Programma PROLOG	5.307	223	0,373
Pagina HTML	6.5773	328	0,289
Messaggio su un forum	7.389	243	0,386
Articolo generico da sito web	1.917	216	0,346
File di log	140.222	326	0,281

Tabella 2: I risultati del test per ComprimiH (algoritmo di Huffman). Le dimensioni sono espresse in byte.

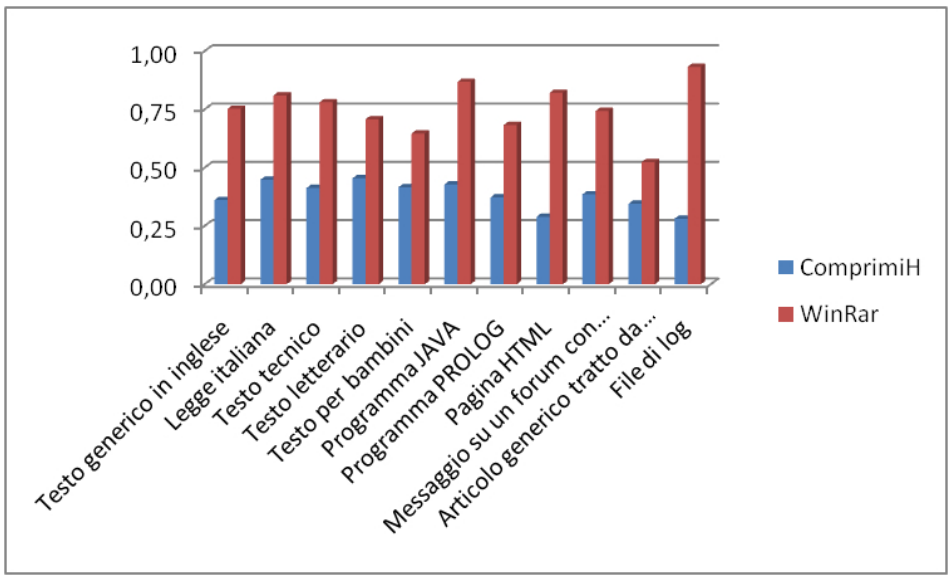

Figura 3: Grafico dei risultati del test per ComprimiH (algoritmo di Huffman)

parole vengono declinate, a seconda del genere, del numero o del caso grammaticale, e i verbi vengono coniugati modificando solo la loro parte finale. Da ciò si può concludere che nel nostro dizionario ha senso memorizzare non parole intere, ma suffissi di parole usati frequentemente.

Ho provato ad eseguire una piccola analisi del file "legge.txt" per avere un'idea concreta su quante parole ripetute ci possano essere in un testo. Su un totale di 18.058 ce ne sono 1.582 diverse. La parola più frequente è "di" (551 occorrenze), seguita da "il" (219 occorrenze) . Nelle prime posizioni troviamo anche parole più lunghe, come "dati" (183 occorrenze) ed "articolo" (114 occorrenze). Dalla tabella 3 osserviamo che il suffisso "person" viene ripetuto 114 volte, "interess" 65 volte.

Parola	Frequenza		Parola	Frequenza
persona	11		interessate	3
personale	11		interessati	7
personali	72		interessato	46
personalita	1		Interesse	8
personalmente	1		Interessi	1
persone	18			

Tabella 3: Esempi di parole con la stessa radice tratte dal file "legge.txt".

Spinto da questi dati e dalla curiosità o quindi creato un algoritmo di Huffman che assegna un codice anche ai suffissi delle parole più usate. Probabilmente le prestazioni non arriveranno ai livelli degli altri algoritmi, però ci sarà sicuramente un miglioramento che forse sarà più evidente nei sorgenti di programmi, nei file XML e simili. Ho anche provato ad eseguire una brevissima ricerca in Internet per vedere se esiste già qualcosa di simile, senza però trovare nulla.

Implementazione

La realizzazione pratica di questa variante richiede l'aggiunta all'implementazione di Huffman vista sopra di alcune operazioni aggiuntive che possono essere individuate nei seguenti punti:

1. Dalla scansione del testo, oltre ad ottenere le frequenze dei caratteri dell'alfabeto utilizzato, occorre anche ottenere una lista di suffissi di parole con le loro frequenze. Ovviamente questi suffissi devono essere usata abbastanza frequentemente nel testo. Inoltre per il calcolo della frequenza dei simboli non bisogna contare le volte nelle quali questi appaiono in uno dei suffissi identificati.

2. Occorre generare un codice, oltre che per i simboli identificati, anche per i suffissi. Inoltre simboli e suffissi devono essere trattati allo stes-

so modo in questa fase, in modo da applicare lo stesso algoritmo di Huffman usato in precedenza su un alfabeto "esteso".

3. Occorre identificare una struttura dati (o più) per la memorizzazione dei codici dei suffissi e per i codici dei simboli che possa essere usata efficacemente in fase di codifica.

4. Occorre codificare il file usando, quando possibile, i codici dei suffissi.

Le aggiunte sopra riportate non sono proprio banali e richiedono un certo numero di considerazioni. Di seguito parlerò solo del metodo usato per la ricerca dei suffissi, di come viene memorizzato il file compresso con il relativo dizionario e della procedura di decompressione. Per un'analisi più approfondita si può vedere il codice sorgente del programma (le classi TextScanner, ComprimiH2, DecomprimiH2) oppure la figura 4, che speiga i compiti delle funzioni impiegate.

L'organizzazione in classi tuttavia segue quella di ComprimiH. Viene solo aggiunta una nuova classe, TextScanner, che svolge i compiti descritti nel primo punto (e le due classi pricipali si chiamano ora ComprimiH2 e DecomprimiH2). Il comportamento esterno di questo programma, compresa la sintassi per eseguirlo, è uguale a quello di ComprimiH.

Ricerca dei suffissi

Una cosa un po' banale, ma che va detta è che una parola viene considerata come una sequenza di caratteri alfanumerici la quale deve essere lunga almeno due caratteri. Qui non ci interessano parole di lunghezza unitaria in quanto sono già rappresentate dai caratteri. Quindi nel nostro programma le parole rilevate non potranno contenere virgole o altri simboli strani e questi si potranno usare come "separatori". Inoltre in un file potrebbero esserci moltissime parole, a differenza dei caratteri che sono in numero finito, per cui non appare molto sicura la scelta di tenere in memoria tutte quelle incontrate. Anche usando meccanismi molto efficenti, se facissimo così la prima scansione del file non avrebbe più complessità lineare.

Ho deciso allora di tenere in un array di dimensione fissata le N parole che, con una certa approssimazione, sono ritenute più frequenti. Per far questo ad ogni parola incontrata, viene associato un punteggio e la frequenza, dove il punteggio tiene conto del numero di occorrenze e dell'età della parola. Tutti i punteggi vengono aggiornati ogni volta che si legge una parola dal file. Quando l'array è pieno e occorre inserire una nuova parola, essa prende il posto di quella con punteggio più basso, che è quindi una apparsa poche volte e non di recente. Per ulteriori dettagli si può vedere il codice del programma, guardando il metodo *updateWordList* della classe *TextScanner*.

Quando si legge una nuova parola questa viene cercata nell'array di cui ho parlato prima. Non si cerca però un match esatto, ma piuttosto una

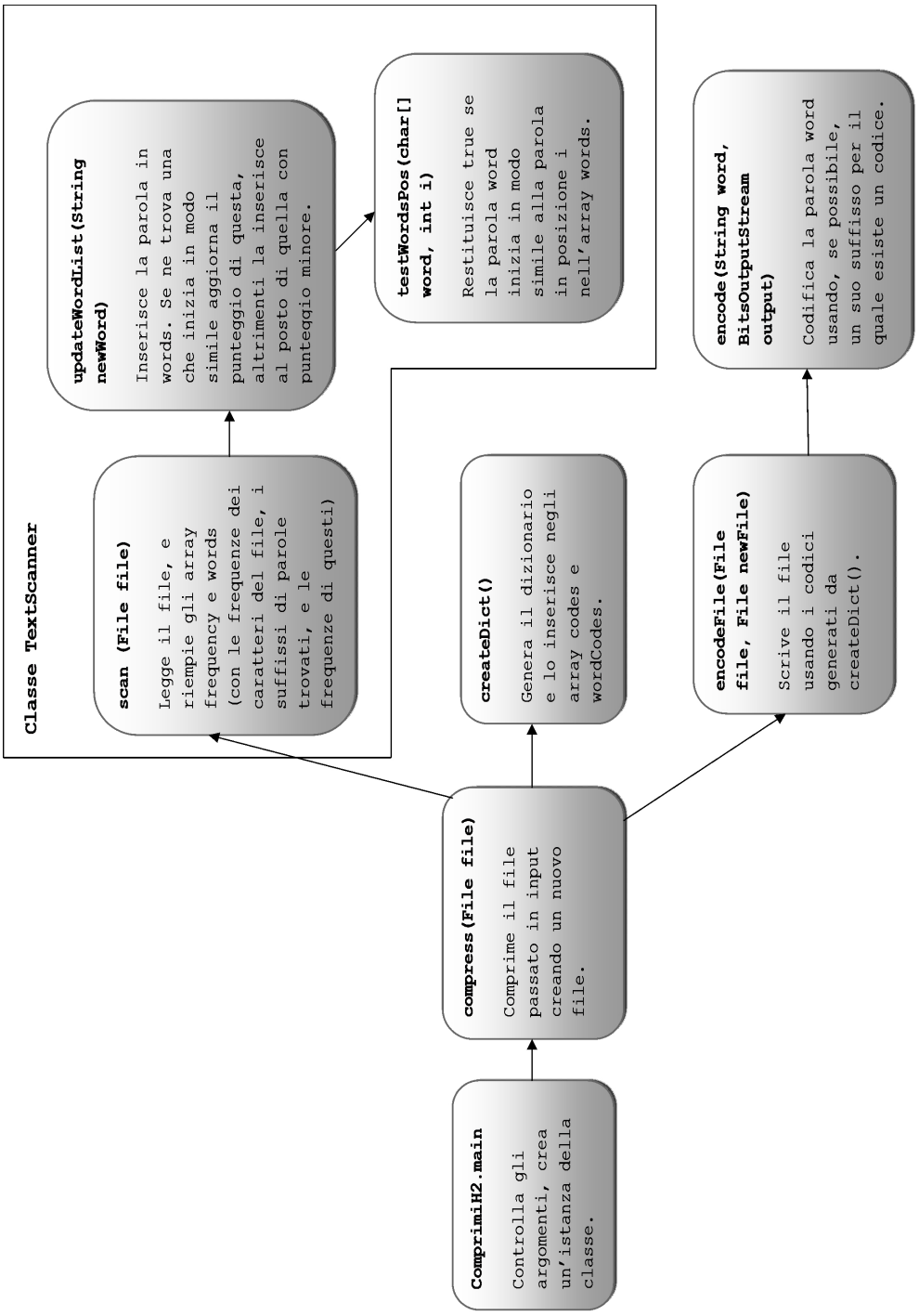

Figura 4: Le funzioni usate nella classe ComprimiH2.java. Sono riporate quelle principali, mentre quelle che fungono solo da supporto sono state omesse.

parola che inizi in modo simile secondo il seguente criterio implementato nel metodo *testWordsPos*:

1. Sia $p1$ la prima parola di lunghezza $l1$, $p2$ la seconda di lunghezza $l2$.
2. Sia j il primo carattere tale che $p1[j] \neq p2[j]$ (se le due parole sono uguali $j = l1 + 1 = l2 + 1$).
3. Se $max(l1, l2) - j \leq 2$ (costante WORD_LENGTH_DIFF_MIN) e $j \geq 2$ (costante WORD-MIN-LENGTH) allora le due parole iniziano in modo simile.

Si nota che il controllo $j \geq 2$ è necessario per garantire di non togliere pezzi ad una parola del dizionario fino a ridurla ad una lunghezza unitaria o addirittura nulla.

Una volta letto tutto il file viene effettuata una "pulizia" dell'array delle parole, nella quale vengono tolte tutte quelle con frequenza inferiore al valore della costante WORD_FREQ_LOWER_LIMIT (che io ho impostato a 3). In questo modo nella fase successiva non vengono generati codici per parole non molto frequenti.

Nell'analisi del testo si usano le seguenti strutture dati:

- Un'array (frequency) che contiene le frequenze dei caratteri del file.
- Un'array (words) di dimensione WORDS_SIZE che contiene le parole più frequenti. I suoi elementi sono quadruple (parola, lunghezza, punti, frequenza).

In tabella 4 troviamo le venti parole più frequenti del file legge.txt, individuate la questo programma. Inoltre, sempre facendo riferimento a questo file, delle 250 parole presenti nell'array dopo l'opreazione di "pulizia" ne rimangono 188[2]

Struttura del file compresso e deompressione

E' molto simile a quella di ComprimiH: viene scritta inizialmente la lunghezza del file in 32 bit; in seguito il dizionario dei simboli, il dizionario delle parole ed il file compresso. Nel dizionario dei simboli viene prima scritta la sua cardinalità (8 bit), poi una sequenza di elementi del tipo: carattere (8 bit), lunghezza codice (8bit), codice. Anche nel dizionario delle parole viene scritta prima la sua cardinalità in 8 bit, seguita da una sequenza di elementi del tipo: lunghezza parola (8 bit), lunghezza codice(8 bit), parola scritta usando i codici associati ai simboli, codice per la parola. Qui il dizionario occupa una parte considerevole di spazio: in media 1,3 Kb contro i 265 byte

[2]E' possibile vedere questi risultati usando l'opzione d di ComprimiH2

Parola/suffisso	Frequenza
di	549
de	307
dell	282
al	199
dat	194
il	172
la	153
per	149
articol	139
cui	138
in	137
trattament	108
da	104
un	90
legg	89
personal	80
Garante	80
le	79
con	73
com	71

Tabella 4: Suffissi ricavati dal programma ComprimiH2 a partire dal file "legge.txt".

di ComprimiH. Si potrebbe ridurre usando, ad esempio, un codice di Huffman fissato a priori per memorizzare le lunghezze e le cardinalità oppurecon i codici di Huffman canonici.

Il programma per la decompressione fa uso di una albero binario molto simile a quello usato in DecomprimiH, ma naturalmente ora i suoi nodi ospiteranno stringhe al posto di caratteri. Ho ritenuto appropriato riportare il pseusodocice del processo di decompressione, che si trova in figura 5, mentre non verrà riportato lo schema che illustra le funzioni usate.

```
Read #characters of the dictionary (8 bytes)
For i=0 to #characters of the dictionary
   read C, L(8 + 8 bytes)
   readDictionaryEntry of length L for char C
Next

Read #words of the dictionary
For i=0 to #words of the dictionary
   read n, L (8 + 8 bytes)
   s = '';
   for j=0 to n
      readHuffmanWord c
      s = s.c
   next
   readDictionaryEntry of length L for string s
next

Count = 0
While count < fileLength
   readHuffmanWord s
   output s
   count = count + length(s)
End while
```

Figura 5: La procedura per decomprimere un file prodotto da ComprimiH2.

I risultati

Si possono vedere nella tabella 5 e nella figura 6. Nella figura vengono riproposte anche le prestazioni di ComprimiH, per avere un confronto diretto fra de due versioni dell'algoritmo. Per comprimere il file enwik8 ci sono voluti 4 minuti e mezzo (279 secondi), circa il doppio del tempo richiesto da ComprimiH.

Breve commento

Dai risultati del test le prestazioni in termini di capacità di compressione dell'algoritmo di Huffman non sembrano essere molto soddisfacenti, soprattutto se confrontate con quelle di WinRar: non si riesce a raggiungere un indice di 0,5 ed il valore medio è 0,381 contro il 0,752 di WinRar. Ciò significa che quando comprimiamo un file usando Huffman, ci dobbiamo aspettare

	Dim. tot. file compr.	Dizionario	Indice
Testo generico in inglese	62.860.364	2.048	0,371
Legge italiana	30.474	1.472	0,542
Testo tecnico	96.818	1.639	0,480
Testo letterario	110.806	1.184	0,448
Testo per bambini	3.894	631	0,438
Programma JAVA	59.897	2.054	0,570
Programma PROLOG	4.888	853	0,422
Pagina HTML	52.419	1.809	0,434
Messaggio su un forum	6.796	900	0,435
Articolo generico da sito web	2.015	369	0,312
File di log	88.095	1.977	0,548

Tabella 5: I risultati del test per ComprimiH2 (variante dell'algoritmo di Huffman). Le dimensioni sono espresse in byte.

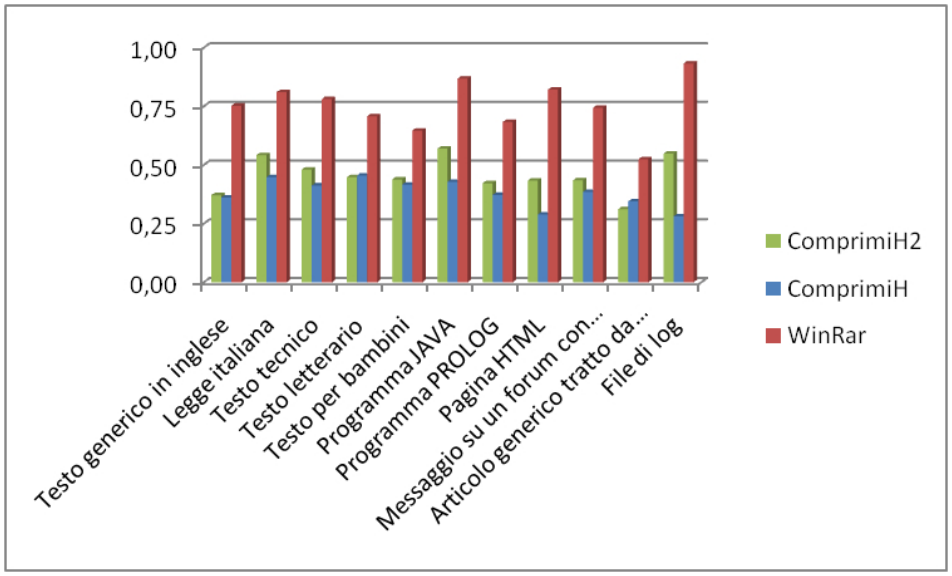

Figura 6: Risultati del test per ComprimiH2 e ComprimiH(algoritmo di Huffman)

di guadagnare solo il 38% dello spazio. Si osserva comunque che la capacità di compressione non varia molto sui diversi tipi di file e che le prestazioni migliori si sono ottenute comprimendo il testo letterario (0,455) mentre quelle peggiori con il file di log (0,281).

La variante, implementata in ComprimiH2, ottiene prestazioni migliori rispetto alla versione originale ma la differenza non è molto grande. Il guadagno maggiore lo si ha per il file di log (si passa da 0,281 a 0,548), ed aumenta in modo significativo anche la compressione del file Java e del file HTML. Le prestazioni peggiorano, anche se in quantità minima, per il testo letterario e per l'articolo generico. In media l'indice di prestazioni è salito a 0,455.

Per quanto riguarda la velocità, si può dire che l'algoritmo non è molto veloce: anche se potrebbe essere implementato in maniera più efficiente (soprattutto la parte di decompressione) sono sempre richieste due "passate" del file in input. Nella compressione e decompressione è richiesta solo la memoria per contenere il dizionario. Infine lo spazio usato per memorizzare il dizionario è spesso trascurabile rispetto alla dimensione totale del file complesso.

Uno degli svantaggi dell'algoritmo di Huffman è che esso non sfrutta le possibili ripetizioni che si possono trovare in un file. Si codifica un byte alla volta e non c'è la possibilità di rappresentare con pochi bit grandi quantità di informazioni ripetute. Questo giustifica in parte le prestazioni scadenti ottenute con il file di log, nel quale vi è una grande quantità di stringhe ripetute, ma vengono anche usati molti simboli diversi. Le prestazioni relative al testo letterario sono invece abbastanza buone: qui non ci sono moltissime stringhe ripetute e vengono usati quasi esclusivamente i simboli che corrispondono a lettere dell'alfabeto italiano. Viene quindi sfruttato il fatto che nella lingua italiana c'è un certo insieme abbastanza ristretto di lettere usate con più frequenza. Un aspetto importante è che le prestazioni non sono molto legate al tipo di file considerato. Questo algoritmo viene spesso usato assieme ad altri ed è abbastanza facile da modificare o adattare a situazioni particolari (come ho fatto in ComprimiH2).

L'algoritmo di Lempel-Ziv

Alcune caratteristiche

E' un algoritmo lunghezza variabile-blocco. Usa i caratteri precedentemente analizzati come dizionario e quindi necessita di un buffer che li mantenga in memoria. Il buffer viene diviso in una parte di ricerca (contenente simboli già elaborati) ed una parte di look-ahead (simboli ancora da elaborare). Usando questi vengono prodotti in output dei codici composti da tre elementi: un puntatore alla parte di ricerca del buffer, un numero di caratteri da copiare ed un carattere del file. Questi codici possono essere usati per

ricostruire il file originale usando delle parti già incontrate precedentemente. La dimensione dei buffer è un parametro importante per le prestazioni di questo algoritmo, inoltre occorre affrontare il problema della ricerca di sottostringhe del buffer che corrispondono con i caratteri successivi in input.

Le varianti

Esistono davvero molte varianti di questo algoritmo: nel libro [8] ne sono presentate sette. Molte, però, consistono solo in piccolo miglioramenti del metodo originale e sono simili fra loro. Moto citata è la "LZSS" in quanto risolve due problemi dell'algoritmo originale, che diventano evidenti una volta implementato l'algoritmo. Il primo è che molto spesso si incontrano codici nei quali i primi due elementi (posizione del buffer di ricerca e lunghezza) sono nulli. Questo capita quando non incontriamo niente nel buffer di ricerca che corrisponda con i prossimi simboli per cui possiamo solo copiare il carattere corrente. La soluzione è quella di avere due tipi di parole di codice: una che rappresenta un carattere, una che rappresenta "cose da copiare". Per distinguere le due parole viene usato un flag. L'altro problema, più evidente e che riguarda il tempo di esecuzione, è la ricerca nel buffer: LZSS prevede la sua gestione mediante BST.

Queste idee vengono usate anche nella variante che sicuramente ha avuto maggior successo: Deflate. Essa, come detto all'inizio, viene usata nei programmi GZip e PkZip, e quest'ultimo è quello che ha introdotto il formato Zip. L'algoritmo, a differenza di molti altri che sono stati brevettati, è di dominio pubblico ed è descritto in [9]. Oltre alle due "modifiche" di LZSS, qui si impiega una codifica di Huffman per rappresentare efficacemente i codici (in effetti in un output di LZSS ci saranno valori di lunghezze e di posizioni usati con maggior frequenza, e che quindi ha senso rappresentare con meno bit). In particolare si usano due alfabeti distinti: uno per i caratteri ed i puntatori, ed uno per le lunghezze. Inoltre il file viene diviso in blocchi ed per ogni blocco vengono usati codici di Huffman differenti. Si capisce bene il funzionamento generale di Deflate guardando l'algoritmo per la decompressione, riportato in figure 7[3].

Un'altra variante interessante è LZX. É molto simile a Deflate, ma prevede l'uso di due codici speciali che rappresentano le ultime posizioni usate nel buffer di ricerca: si è notato che c'è la tendenza di queste posizioni a ripetersi. Come detto all'inizio LZX è usato molto dalla Microsoft, in particolare nei file cab, chm, lit, wim.

[3]Se il file è compresso con "codici di Huffman dinamici" viene letto il dizionario dal file, altrimenti si usano dei codici fissati a propri e descritti in [9].

```
do
   read block header from input stream.
   if stored with no compression
      skip any remaining bits in current partially processed byte
      read LEN and NLEN (see next section)
      copy LEN bytes of data to output
   otherwise
      if compressed with dynamic Huffman codes
         read representation of code trees
      loop (until end of block code recognized)
         decode literal/length value from input stream
         if value < 256
            copy value (literal byte) to output stream
         otherwise
            if value = end of block (256)
               break from loop
            otherwise (value = 257..285)
               decode distance from input stream
               move backwards distance bytes in the output
               stream, and copy length bytes from this
               position to the output stream.
      end loop
while not last block
```

Figura 7: La procedura per decomprimere un file compresso con l'algoritmo Deflate.

Programmi

I programmi che ho usato nelle prove sono gzip e WinRAR. Inoltre ho implementato una versione "standard" dell'algoritmo per compararne le prestazioni con Deflate.

Gzip è un programma gratuito (gzip è l'acronimo di GNU Zip) basato sulla variante Deflate. Si può trovare al sito [10] in formato codice sorgente, oppure compilato per ben 23 diverse piattaforme. L'ultima versione è la 1.3.12, rilasciata in Aprile 2007, mentre la prima versione risale all'anno 1992. Il formato di file usato è descritto in [11] e, a differenza dei file Zip, consente di comprimere solo un file alla volta. Nel file compresso, comunque, c'è lo spazio per inserire alcune meta-informazioni come nome del file, timestamp, un codice da 32 bit per la correzione degli errori, un commento ed altro ancora. Si dice sia più lento di compress, ma che offra prestazioni superiori. In particolare testi in inglese o sorgenti vengono tipicamente compressi del 60-70%. Come previsto da Deflate gli offset vengono cercati nei 32Kb precedenti al simbolo corrente, mentre le distanza possono essere lunghe al massimo 258 bit. Il buffer è gestito mediante una tabella hash che per gestire le collisioni usa una strategia chaining. La lunghezza delle liste associate agli elementi è determinata dal livello di compressione, impostabile al momento dell'esecuzione con un valore da 1 a 9. Questo parametro viene anche usato per determinare se fermarsi al primo match sufficientemente lungo trovato, oppure se eseguire una ricerca più completa. Delle indicazioni

più approfondite e ben scritte su questo programma si possono trovare in [12].

WinRAR è un programma commerciale che produce archivi in formato Rar (acronimo di Roshal Archive) oppure in formato Zip. La versione usata nelle prove è la 3.60 beta 3, scaricabile dal sito [13] ed è stata rilasciata a Maggio 2006. Una delle prime versioni (la 1.54) risale invece all'anno 1995. Per quanto riguarda il formato Rar ci sono 6 livelli di compressione a disposizione: da "store", nel quale i file non vengono compressi ma solo archiviati[4], a "best". Viene usato un algoritmo generale basato su LZSS nel quale il buffer di ricerca (chiamato dizionario) può assumere dimensioni da 64Kb a 4MB [14]. Oltre a questo vengono usati altri algoritmi per specifici tipi di dato che vengono attivati solo nelle modalità di compressione più avantzate. C'è un algoritmo per l'audio, uno per le immagini, uno per i testi che fa uso di "Prediction by Partial Matching", una "delta compression" ed uno per i file eseguibili. WinRar sceglie automaticamente quali algoritmi attivare, tuttavia è possibile anche sceglierli manualmente. Queste informazioni sono state tratte dalle guida in linea ma per approfondimenti è disponibile un file "TechNote.txt" che contiene una descrizione di come sono strutturati i file Rar, fornito assieme al programma, oppure sempre sul sito [13] si possono scaricare i file sorgenti di un programma per decomprimere file Rar.

I risultati

Nelle tabelle 6 e 7 sono riportati i risultati ottenuti da gzip e WinRAR. Quest'ultimo è stato provato sia usando il formato Rar che usando il formato Zip, che dovrebbe avere prestazioni in linea con quelle di gzip dato che si usa lo stesso algoritmo e cambia solo l'intestazione del file. In questi programmi è possibile settare dei parametri che incidono sulla capacità di compressione, ed ho cercato di settarli in modo da ottenere le massime prestazioni possibili. Per gzip ho usato l'opzione "-9", mentre in WinRAR il metodo di compressione "best", sia nel caso rar che nel caso zip. La rappresentazione grafica dei risultati verrà proposta più avanti, in modo da includere nel grafico anche i risultati ottenuti dal programma basato su LZ che ho implementato. Gzip è stato il secondo programma più veloce a comprimere il file enwik8: ci sono vuoluti 12 secondi. WinRar, invece, ha impiegato 49 secondi nella compressione Rar, 17 secondi nella compressione Zip.

[4] molti programmi per la compressione permettono di memorizzare più di un file in un unico archivio compresso. Con programmi come compress, gzip o bzip2 è invece possibile comprimere solo un file alla volta

	Dim. file compresso	Indice
Testo generico in inglese	36.445.248	0,636
Legge italiana	17.080	0,744
Testo tecnico	57.352	0,692
Testo letterario	79.409	0,604
Testo per bambini	2.880	0,585
Programma JAVA	24.563	0,824
Programma PROLOG	3.040	0,641
Pagina HTML	20.551	0,778
Messaggio su un forum	3.554	0,705
Articolo generico da sito web	1.527	0,479
File di log	18.577	0,905

Tabella 6: I risultati del test per Gzip(Deflate). Le dimensioni sono espresse in byte.

	Rar		Zip	
	Dim. file	Indice	Dim. file	Indice
Testo generico in inglese	24.747.078	0,753	36.422.375	0,636
Legge italiana	12.626	0,810	17.158	0,742
Testo tecnico	40.799	0,781	57.267	0,693
Testo letterario	58.620	0,708	79.407	0,604
Testo per bambini	2.449	0,647	2.970	0,572
Programma JAVA	18.287	0,869	24.646	0,823
Programma PROLOG	2.680	0,683	3.128	0,630
Pagina HTML	16.529	0,821	20.632	0,777
Messaggio su un forum	3.080	0,744	3.642	0,697
Articolo generico da sito web	1.394	0,524	1.618	0,448
File di log	13.133	0,933	18.668	0,904

Tabella 7: I risultati del test per WinRAR (vari algoritmi per quanto riguarda Rar, Deflate per quanto riguarda Zip). Le dimensioni sono espresse in byte.

Implemetazione

Anche in questo caso ho usato il linguaggio Java ed ho cercato di scrivere del codice puntando alla semplicità ed alla comprensibilità. Ho scelto di implementare la versione base dell'algoritmo per poterla confrontare con la variante Deflate. Inoltre implementandola ho avuto la possibilità di capire l'algoritmo ad un livello più profondo, quali cose possono essere fatte in modo alternativo, quali sono i suoi problemi ed i suoi vantaggi. Ad esempio ho trovato LZ più veloce da implementare rispetto a Huffman, infatti il codice che ho scritto è più corto rispetto a quello di ComprimiH. Questo è dovuto, a mio avviso, al fatto che qui si effettua un'unica "passata" del testo in input e che la procedura per la decompressione è molto simile al quella per la compressione.

Anche qui descriverò brevemente la struttura logica del programma, il formato del file compresso prodotto in output, le strutture dati usate ed i metodi. Altre informazioni si trovano sul codice sorgente (file ComrpimiLZ.java).

Il programma è ripartito in tre classi: quella principale è ComprimiLZ, mentre le due rimanenti supportano la scrittura e la lettura dei file generati da questo algoritmo e sono LZCodeInputStream e LZCodeOutputStream. A causa della similitudine fra compressione e decompressione è stata creata un'unica classe che svolge entrambi i compiti. Questa contiene il metodo main è può quindi essere eseguita lanciando "$javaComprimiLZ < nomedelfile > \{opzioni\}$. Per stabilire se il file è da comprimere oppure da decomprimere viene valutata la sua estensione, come avviene in gzip o in compress: se è ".pack" il file è già stato compresso è quindi va decompattato. Si possono anche specificare delle opzioni in una stringa opzionale, passata come secondo parametro, che può contenere le seguenti lettere: "v", per avere più informazioni stampate a video, "c", per visualizzare a video i codici prodotti, presentati in un formato opportuno, "f" per impostare la modalità veloce. In questa, anziché eseguire una ricerca completa nel buffer di ricerca, ci si ferma appena si trova un match lungo almeno quanto la costante MIN_LUNGH (impostata a 3). Questa modalità serve per ridurre il tempo di calcolo (nella modalità normale l'algoritmo di ricerca impiegato è quadratico). Dalle prove ho notato che più o meno la velocità di compressone raddoppia, ma anche la dimensione del file compresso e quindi spesso le prestazioni sono insoddisfacenti.

I file compressi prodotti sono semplicemente una sequenza di triple costituite da: un puntatore al buffer di ricerca, una lunghezza ed il carattere successivo. Questo è rappresentato in 8 bit, mentre la lunghezza delle prime due componenti è determinata dalla dimensione del buffer.

L'unica struttura dati usata è appunto il buffer (un array chiamato appunto "buffer"), diviso ovviamente in parte di ricerca e parte di look-ahead. Esso viene gestito in modo circolare per rendere semplici le operazioni di

shift: al posto di spostare tutti i caratteri di un certo numero di posizioni si sposta un puntatore che indica qual è il primo elemento. Le funzioni *getEl* e *setEl* permettono di accedere agli elementi dell'array come se questo non fosse gestito in modo circolare, rendendo più chiaro il codice che lo usa. Ci sono cinque variabili usate per la sua gestione:

- *size*: dimensione del buffer
- *middle*: dimensione della parte di ricerca
- *p_end*: puntatore all'ultimo elemento del buffer (il primo si trova quindi in posizione p_end + 1 mod size). Lo shift di p posizioni del buffer consiste nell'effettuare p_end = p_end + p mod size.
- *p_middle*: puntatore al primo elemento della parte di look-ahead del buffer, viene aggiornato come p_end.
- *lastValid*: puntatore usato dopo un'operazione di shift per identificare gli elementi che sono stati "buttati fuori". Viene usato o per scrivere altri caratteri del file in input al posto di questi, oppure per salvarli da qualche parte. Nell'operazione di shift in esso viene copiato il valore di p_end prima che questo venga modificato. Successivamente i dati "buttati fuori" saranno quelli che vanno da p_end a lastValid.

I metodo usati per comprimere un file sono descritti in figura 8

I risultati

In tabella 8 si possono trovare i risultati del test per quanto riguarda ComprimiLZ. Ho impostato la dimensione del buffer a 64Kb, la parte di ricerca era quindi da 32Kb che è la stessa dimensione stabilita da Deflate. Il valore nullo, che si trova nella riga relativa all'articolo generico, indica che quel file non è stato compresso ma addirittura espanso (il file prodotto ha 538 byte in più rispetto al file in input). Nel grafico 9 possiamo vedere i risultati di tutti i programmi testati che usano un algoritmo della famiglia LZ, per WinRar però si sono considerate solo le prestazioni relative al formato Rar, in quanto i risultati ottenuti comprimendo mediante Zip coincidono con quelli di gzip (infatti in entrambi viene usato Deflate). Nel grafico 10 è invece proposto un confronto fra le prestazioni di ComprimiLZ e ComprimiH, che rappresentano le versioni "pure" degli algoritmi di Huffman e di Lempel-Ziv.

Per comprimere il file enwik8 c'è voluta un'ora e mezza, ciò è dovuto alla procedura di ricerca nel buffer che ha complessità quadratica.

Breve commento

Se consideriamo la versione "pura" dell'algoritmo LZ, implementata in ComprimiLZ, le prestazioni sono anche superiori alle mie aspettative iniziali: esso

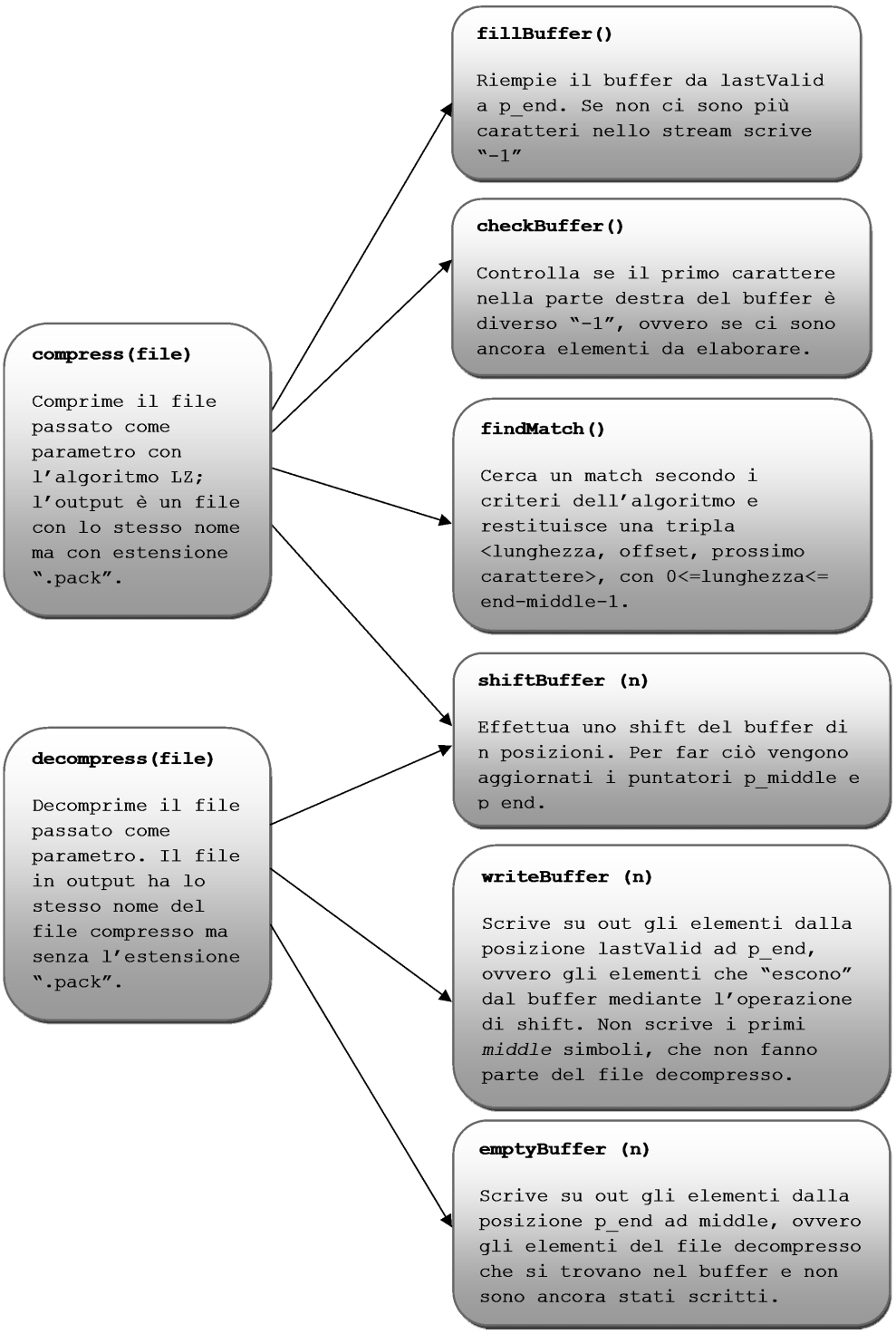

Figura 8: Le funzioni usate nella classe ComprimiLZ.java. Sono riporate quelle principali, mentre quelle che fungono solo da supporto sono state omesse.

	Dim. file compresso	Indice
Testo generico in inglese	67.539.264	0,325
Legge italiana	31.402	0,528
Testo tecnico	107.022	0,426
Testo letterario	142.481	0,290
Testo per bambini	6.184	0,108
Programma JAVA	45.467	0,673
Programma PROLOG	6.626	0,217
Pagina HTML	39.277	0,576
Messaggio su un forum	7.386	0,386
Articolo generico da sito web	3.467	0,000
File di log	34.907	0,821

Tabella 8: I risultati del test per ComprimiLZ (LZ). Le dimensioni sono espresse in byte.

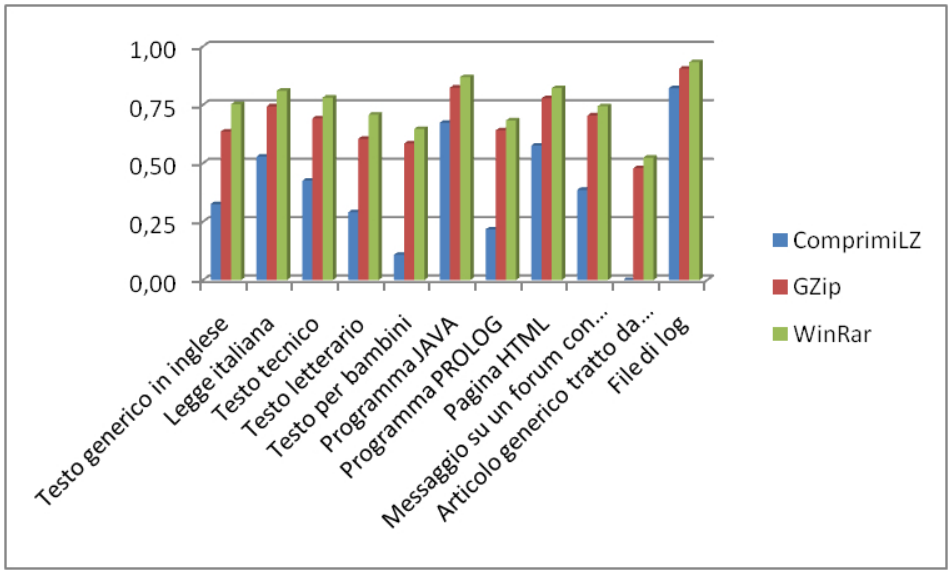

Figura 9: Risultati del test per i programmi gzip (Deflate), WinRar (vari algoritmi) e ComprimiLZ (LZ)

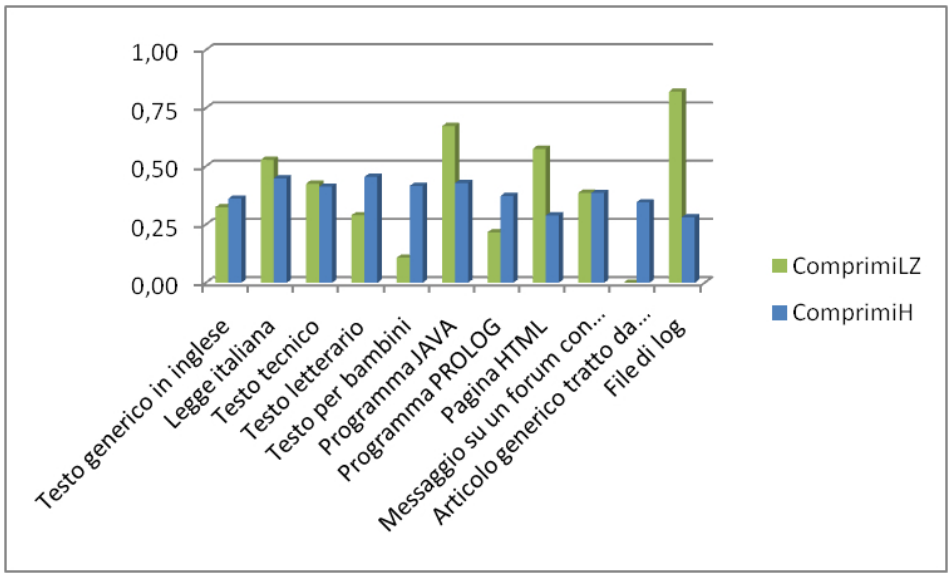

Figura 10: Confronto fra le prestazioni di ComprimiH (Huffman) e ComprimiLZ (LZ)

ha molti punti che potrebbero essere migliorati. In primo luogo la lunghezza delle parole di codice (38 bit, di cui 15 per il puntatore al buffer, 15 per la lunghezza ed 8 per il carattere) sembra eccessiva: nel caso in cui non si trovano sottostringhe nel buffer di ricerca dei prossimi caratteri in input vengono usati 38 bit per rappresentare un singolo carattere. Inoltre appaiono troppi 15 bit per la rappresentare la lunghezza della sottostringa da copiare. Sembra quindi che sia assolutamente necessario usare una codifica più efficiente delle parole di codice, come viene fatto in Deflate. Tuttavia l'algoritmo comprime e in certi casi ottiene anche buone prestazioni. Le migliori le si hanno per il file di log (0,821), ma anche con la legge, la tesi, il file Java ed il file HTML. Quelle peggiori per l'articolo generico (viene addirittura espanso) che comunque è un file molto corto, per la discussione su un forum, per il testo per bambini e per il file enwik8. Se non consideriamo quest'ultimo sembra che la compressione aumenti all'aumentare della dimensione del file in input. Inoltre le prestazioni variano molto in funzione del file considerato (la varianza è di 0,244). L'indice di compressione medio è di 0,395: più di ComprimiH, meno di ComprimiH2.

Deflate, grazie alla codifica più efficiente delle parole di codice, offre una maggio capacità di compressione. L'indice di compressione medio di gzip è 0,690 ed i risultati sembrano essere meno variabili a seconda del file considerato (la varianza è scesa a 0,119). Anche la velocità di compressione è molto buona, grazie alla gestione del buffer mediante una tabella Hash.

Il maggior incremento di prestazioni si ottiene nei file più corti (testo per bambini, articolo generico e programma Prolog).

In conclusione l'algoritmo di Lempel-Ziv offre buone prestazioni sia di compressione che di velocità, a patto che venga implementato in un modo efficiente. In particolare offre prestazioni molto buone se accoppiato con l'algoritmo di Huffman. La logica di funzionamento è abbastanza semplice e sono disponibili anche alcuni circuiti in grado di comprimere usando questo algoritmo. Altri punti di forza sono l'elevata velocità di decompressione e la possibilità di regolare quella di compressione cambiando le modalità di ricerca nel buffer. Uno svantaggio è che, sia per la compressione che per la decompressione, è richiesta molta memoria per il buffer. Inoltre la struttura di questo algoritmo comporta che le prestazioni dipendano da come sono distribuite le stringhe ripetute in un file.

In questa discussione non ho considerato il programma WinRAR in quanto per la compressione dei testi si basa sull'algoritmo "Prediction by Partial Matching" (tuttavia WinRar è stato incluso in questa sezione in quanto l'algoritmo principale è basato su LZ)

L'algoritmo LZW

Alcune caratteristiche

L'algoritmo di Lempel-Ziv e Welch è un algoritmo lunghezza variabile-blocco che costruisce un dinamicamente un dizionario di stringhe incontrate nell'input (chiamato string table). I codici usati per generare il file compresso sono degli indici in questo dizionario. Si tratta di un algoritmo semplice, molto veloce ma che richiede la presenza della string table in memoria . Un parametro fondamentale di questo algoritmo è la dimensione della string table.

Programmi

LZW non viene molto usato al giorno d'oggi (almeno non quanto LZ), sia per motivi legati al brevetto che lo riguardava (ma che ora è scaduto), sia per motivi tecnici ([2], credo si faccia riferimento alle scarse prestazioni). Il programma di riferimento è compress ed è disponibile in rete sotto il nome di "ncompress" (il nome del file da cercare è "ncompress-4.2.4.tar.z"). La versione più recente disponibile è la 4.2.4 e risale all'anno 1992, mentre la prima versione è stata implementata nell'anno 1984. E' possibile scaricarne una versione per Linux ad esempio dal sito [15]. Nelle note allegate al programma si dice che sia molto veloce ma non così efficiente gli algoritmi basati su LZ. In media comprime testi in inglese del 50-60%. A differenza dell'algoritmo LZW "standard", compress genera codici di lunghezza variabile.

Alcune note tecniche: le parole di codice sono costituite inizialmente da 9 bit, quindi il programma gestisce 256 stringhe che occupano le posizioni superiori alla 256 della string table. Quando questa viene riempita si passa a codici da 10 bit, rendendo disponibili altri 512 spazi per memorizzare stringhe dell'input. Si arriva fino a codici da 16 bit (è possibile cambiare questo valore quando viene lanciato il programma, tuttavia 16 bit rimane il massimo consentito). Quando le parole sono da 16 bit e la string table è piena si osserva il rapporto di compressione: se aumenta continuo ad usare la string table esistente, altrimenti la si svuota e si ricomincia daccapo. Per gestire la string table e permettere una ricerca efficiente delle stringhe si usa una tabella hash con strategia open adressing. [Documentazione trovata sul file ncompress-4.2.4.tar.z]

I risultati

L'unica opzioni di compress che è possibile impostare è la dimensione massima della string table, però di default assume già il valore massimo. Per cui non sono state usate opzioni particolari per comprimere usando compress. I risultati sono riportate in tabella 9 ed in figura 11. Per comprimere enwik8 ci sono voluti 6 secondi: è stato il programma più veloce.

	Dim. file compresso	Indice
Testo generico in inglese	46.247.947	0,538
Legge italiana	25.206	0,621
Testo tecnico	73.335	0,606
Testo letterario	83.227	0,585
Testo per bambini	3.548	0,488
Programma JAVA	44.715	0,679
Programma PROLOG	4.322	0,489
Pagina HTML	39.197	0,576
Messaggio su un forum	6.016	0,500
Articolo generico da sito web	1.784	0,391
File di log	48.722	0,750

Tabella 9: I risultati del test per compress (LZW). Le dimensioni sono espresse in byte.

Breve commento

LZW sì è dimostrato un algoritmo molto veloce (6 secondi per comprimere enwik8, è il più veloce) ma con prestazioni inferiori rispetto a gzip: l'indice di compressione mediò è 0,566 (0,124 in meno rispetto a gzip). Il file compresso maggiormente è sempre il file di log (0,75) e mentre i risultati inferiori si

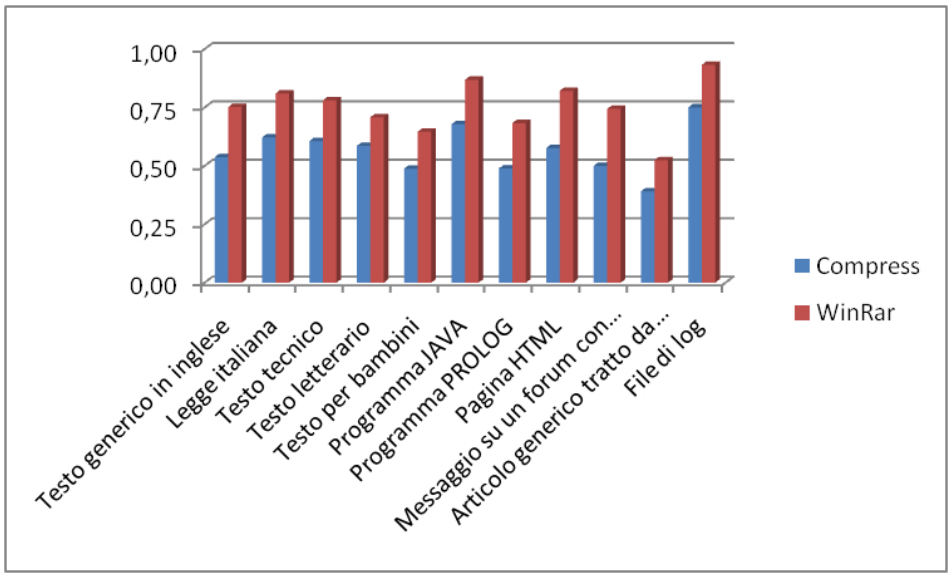

Figura 11: Grafico dei risultati del test per compress(algoritmo LZW)

ottengono con l'articolo generico (0,391), il programma Prolog(0,489) ed il testo per bambini(0,488). I risultati, in compenso, non variano molto a seconda del file considerato: la varianza è di 0,099 (la più bassa dopo le due versioni dell'algoritmo di Huffman).

In conclusione, sebbene LZ possa riconoscere stringhe ripetute solo se queste si trovano in un "introrno" della posizione corrente a causa delle dimensione limitata del buffer di ricerca, la sua capacità di compressione è superiore rispetto a LZW. Il problema qui sembra sia la dimensione limitata della string table, che limita la capacità tenere traccia delle sottostringhe ripetute e quindi di usare pochi bit per rappresentare molta informazione. La velocità, comunque, è doppia rispetto a gzip per cui risulta difficile stabilire quale, fra i due approcci, offra il miglior rapporto capacità di compressione/velocità.

L'algoritmo di Burrows-Wheeler

Alcune caratteristiche

E' un algoritmo che opera su blocchi di file e per questo talvolta viene chiamato "block sorting". I blocchi vengono trasformati, con un processo reversibile, in modo che contengano pochi simboli e che simboli uguali siano il più possibile contigui. Blocchi con queste proprietà possono essere poi compressi efficacemente con l'algoritmo di Huffman. Questo metodo è effi-

cace se i blocchi sono molto grandi, per questo è richiesta molta memoria ed il processo di trasformazione è computazionalmente pesante.

Programmi

Per il test ho usato due programmi basati su questo algoritmo: bzip2 e bbb. Il primo è il più famoso ed usato a livello pratico, il sito di riferimento è [16] e da lì è possibile scaricare l'ultima versione, la 1.04 uscita nel mese di dicembre 2006, per sistemi Windows o Unix. La prima versione di questo programma, invece, risale all'anno 2000, quindi si tratta di una cosa giovane se confrontata con compress o gzip. Nella documentazione allegata al programma si dice che questo offra prestazioni superiori rispetto a loro ma che sia più lento in fase di compressione . Si dice anche che questo algoritmo non funzioni bene con dati random, i quali mediamente vengono espansi con un rapporto di 8,5 bit per byte. La dimensione dei blocchi varia da 100Kb fino a 900Kb e può essere impostata al momento del lancio. Una caratteristica curiosa è che un blocco può essere codificato usando diverse codifiche di Huffman, quando si ritiene opportuno cambiare codifica si scrive una sequenza speciale di bit che identifica il nuovo codice da usare. Per il test verranno usati blocchi da 900Kb.

BBB, acronimo di Big Block BWT, è datato agosto 2006 e può essere scaricato dall'url [17]. Il suo vantaggio è che riesce a gestire blocchi molto più grandi rispetto a bzip2 garantendo quindi migliori capacità di compressione. Nelle prove si sono usati blocchi larghi 100Mb (quindi in grado di contenere interamente ogni file)

I risultati

Dal punto di vista dei parametri che regolano la capacità di compressione BZip2 è molto simile a gzip: anch'esso mette a disposizione 9 livelli di compressione. Anche qui, quindi, si è usato il livello 9. In BBB, invece, la dimensione dei blocchi è impostabile a piacere, per cui ho usato blocchi da 100Mb mediante l'opzione "m100". I risultati si possono vedere nella tabella 10 e nel grafico in figura 12. Bzip2 ha impiegato 24 secondi per comprimere enwik8, mentre bbb un po' meno di quattro minuti (232 secondi).

Breve commento

Si notano indici di compressione migliori rispetto a quelli ottenuti dai programmi basati su Huffman, LZ oppure LZW, tuttavia la velocità di compressione si è dimezzata rispetto a gzip. E' simpatico notare come il tempo impiegato da gzip per comprimere enwik8 sia esattamente il doppio di quello di compress e la metà di quello di bzip2. A parte questo il file compresso maggiormente è sempre il solito file di log (0,92) e quello compresso meno è sempre l'articolo (0,481). Se non consideriamo quest'ultimo, però, i risultati

	BZip2		BBB	
	Dim. file	Indice	Dim. file	Indice
Testo generico in inglese	29.008.758	0,710	24.576.921	0,754
Legge italiana	14.643	0,780	14.551	0,781
Testo tecnico	46.281	0,752	44.954	0,759
Testo letterario	63.737	0,682	59.361	0,704
Testo per bambini	2.663	0,616	2.638	0,619
Programma JAVA	20.983	0,849	21.445	0,846
Programma PROLOG	3.063	0,638	3.091	0,635
Pagina HTML	19.486	0,789	20.723	0,776
Messaggio su un forum	3.897	0,676	3.945	0,672
Articolo generico da sito web	1.519	0,481	1.413	0,518
File di log	15.624	0,920	18.115	0,907

Tabella 10: I risultati del test per BZzip2 e BBB (algoritmo di Burrows-Wheeler). Le dimensioni sono espresse in byte.

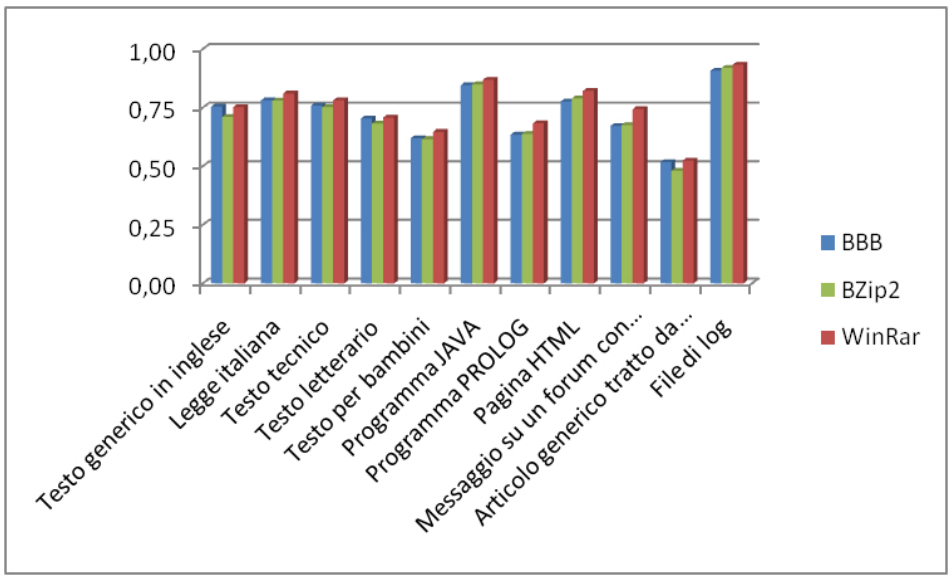

Figura 12: Grafico dei risultati del test di BZip2 e BBB (algoritmo di Burrows-Wheeler)

sono sempre superiori allo 0,6. L'indice di compressione non è molto omogeneo fra i vari file di input: la varianza dei risultati è 0,12, quella maggiore se non calcoliamo ComprimiLZ.

Le prestazioni di bbb e di bzip2 non siano molto differenti, nonostante la diversa dimensione dei blocchi. Si ha un incremento abbastanza significativo per file enwik8 (nel quale bbb riesce a "togliere" 5Mb in più), ed anche nel testo tecnico. In alcuni casi, invece, le prestazioni sono inferiori a bzip2 (ad esempio nel file di log e nella pagina HTML). Comunque l'indice medio di compressioe è di 0,725 contro il 0,718 di bzip2.

Nel complesso l'algoritmo di Burrows-Wheeler è quello che offre prestazioni maggiori rispetto a quelli visti fin'ora, tuttavia occorre considerare il significativo aumento dei tempi richiesti per la compressione rispetto a gzip. Inoltre, sempre nella fase di compressione, è richiesta una quantità di memoria superiore rispetto agli altri: bzip2, usando blocchi da 900Kb, richiede 7,5Mb (ma solo 3,6Mb per la deocmpressione).

Altri algoritmi/programmi di compressione

Il programma PAQ

L'obiettivo di questo lavoro è valutare le quattro famiglie di algoritmi visti sopra,per cui non fornirò una descrizione molto approfondita del programma considerato in questa sezione, PAQ. Comunque fornirò qualche informazione tratta da [2] e [17]. Si basa sul un algoritmo "Context Mixing" (simile al "Prediction by Partial Matching") ed è sviluppato da un gruppo numeroso di persone con in comune l'obiettivo di sviluppare un programma ad alte capacità di compressione, senza preoccuparsi delle prestazioni in termini di velocità e memoria utilizzata. La prima versione di PAQ è stata rilasciata nell'anno 2002, mentre la versione usata è la "8o8", risale ad ottobre 2007 ed è scaricabile gratuitamente dal sito [17]. E' disponibile anche una versione alternativa specializzata per la compressione di testi in lingua inglese ma che fornisce prestazioni inferiori con altri tipi di dati (immagini, ad esempio, ma anche testi in italiano). E' possibile specificare un parametro che influisce sulla capacità di compressione e può spaziare da 1 a 8, in particolare questo incide sulla memoria usata dal programma che varia da 35Mb a 1712Mb (con l'opzione 8).

Vengono impiegati dei metodi speciali per la compressione di testi: essi, prima di essere compressi dall'algoritmo principale, vengono sottoposti ad un processo nel quale si sostituiscono delle parole presenti in un dizionario fissato a priori con dei codici speciali. In aggiunta le lettere maiuscole vengono sostituite dalle corrispondenti minuscole, in modo da considerare uguali anche due parole che si differiscono solo per l'iniziale maiuscola. Caratteri speciali vengono impiegati per ricordarsi quali lettere erano maiuscole.

I risultati

PAQ è stato usato con l'opzione "-8", che è quella che offre miglior capacità di compressione. I risultati sono riportati in tabella 11 e nel grafico in figura 13. Per comprimere il file enwik8 c'è voluta circa un'ora. La versione di PAQ ottimizzata per testi in inglese è in grado di comprimere il file enwik8 a 16.230.028 byte, ottenendo un indice di prestazione pari a 0,837, tuttavia è stata esclusa dal test in quanto offre prestazioni inferiori a PAQ 8o8 sui testi in italiano.

	Dim. file compresso	Indice
Testo generico in inglese	17.904.756	0,821
Legge italiana	11.374	0,829
Testo tecnico	36884	0,802
Testo letterario	52429	0,739
Testo per bambini	2292	0,669
Programma JAVA	13942	0,900
Programma PROLOG	2327	0,725
Pagina HTML	13108	0,858
Messaggio su un forum	2788	0,768
Articolo generico da sito web	1294	0,558
File di log	9111	0,953

Tabella 11: I risultati del test per PAQ 8o8 (Context Mixing). Le dimensioni sono espresse in byte.

Breve commento

PAQ è il programma che offre maggiori capacità di compressione: l'indice di prestazione medio è di 0,784 contro il 0,690 di gzip. La varianza dei risultati (0,112), invece, è in linea con la media. Il file compresso maggiormente è sempre il file di log (si raggiunge 0,953, il valore più alto), mentre quello compresso meno è sempre l'articolo.

Richiede moltissima memoria in confronto agli altri: la quantità minima è 32Mb, ed anche il tempo di esecuzione è notevolmente superiore (se non calcoliamo ComprimiLZ). Ciò rende difficile stabilire se l'alta capacità di comprimere di PAQ giustifichi le alte richiesti di tempo e risorse. Tuttavia una caratteristica a favore di questo programma è che si tratta di un progetto recente e in fase di sviluppo, per cui possiamo aspettarci qualche miglioramento in futuro.

Inoltre occorre considerare anche WinRAR, che fa uso dell'algoritmo "Prediction by Partial Matching", ed è il secondo programma più performante: il risultato medio è di 0,752. I suoi tempi di compressione (49 secondi

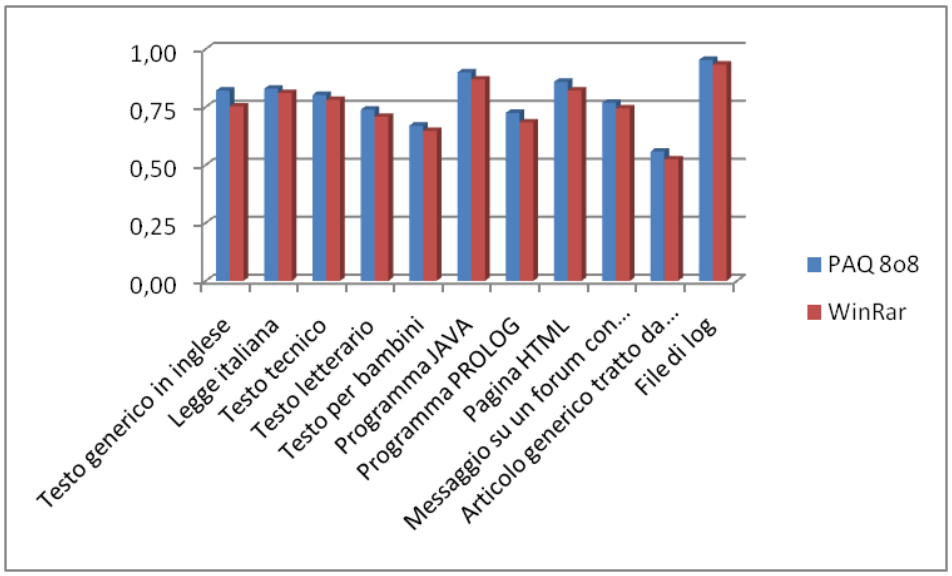

Figura 13: Grafico dei risultati del test per PAQ 8o8 (Context Mixing)

per enwik8) sono superiori alla media degli altri algoritmi considerati, compreso Burrows-Wheeler, però non eccessivi e le prestazioni non sono poi così lontane da quelle di PAQ. WinRar è un programma usato molto comunemente, anzi, quasi certamente si tratta del programma più diffuso fra quelli considerati.

Confronto degli algoritmi e principali risultati

Nonostante ci siano ancora molte cose da vedere e da approfondire, questo lavoro si ferma qui. Ora cercherò di mettere assieme quanto visto sui vari algoritmi e di dare una visione d'insieme dei risultati ottenuti.

Un primo confronto lo si può avere guardando i grafici nelle figure 14 e 15. Il primo riporta le prestazioni relative alla compressione raggiunte dai vari programmi provati, mentre nel secondo troviamo i tempi impiegati per comprimere il file enwik8. Nella tabella 12, ho invece cercato di classificare i programmi secondo alcune caratteristiche che mi sono venute in mente durante lo svolgimento di questo lavoro e che ho ritenuto significative. Per permetterne una visione più immediata ho assegnato dei valori simbolici che vanno da "basso" ad "alto". Questi li ho assegnati mettendo assieme i risultati numerici con le impressioni che ho avuto nello svolgimento dei test. Non si tratta sicuramente di una classificazione completa, manca ad esempio un'analisi del processo di decompressione, tuttavia credo che fornisca una prima caratterizzazione di ciascun algoritmo.

Credo inoltre che l'insieme dei programmi testato sia abbastanza paradigmatico e che copra in modo sufficientemente completo gli algoritmi principali. Tuttavia ci sono moltissimi programmi per la compressione, ciascuno dei quali aggiunge qualche nuova idea all'algoritmo principale sul quale si basa. Ho speso un po' di tempo ad analizzarli e a cercare quelli più significativi, ma ci sono molte cose che ho tralasciato per cui credo che sicuramente ci sia in giro qualche altro programma che sarebbe stato interessante confrontare con quelli visti, ma che non ho considerato.

Interessante è anche il grafico 16 che riporta i valori di compressione minimi, medi e massimi con cui sono stati compressi i vari testi. Si nota, ad esempio, che i file che sono più facili da comprimere sono il file di log, il programma Java e la pagina HTML.

Comunque ho trovato molto interessante ed appagante svolgere questa indagine. Sono consapevole del fatto che non sia molto approfondita e che sicuramente conterrà qualche inesattezza ma spero che, oltre a far conoscere un po' meglio le varie famiglie di algoritmi, saprà trasmettere anche un po' di interesse verso questo settore dell'informatica.

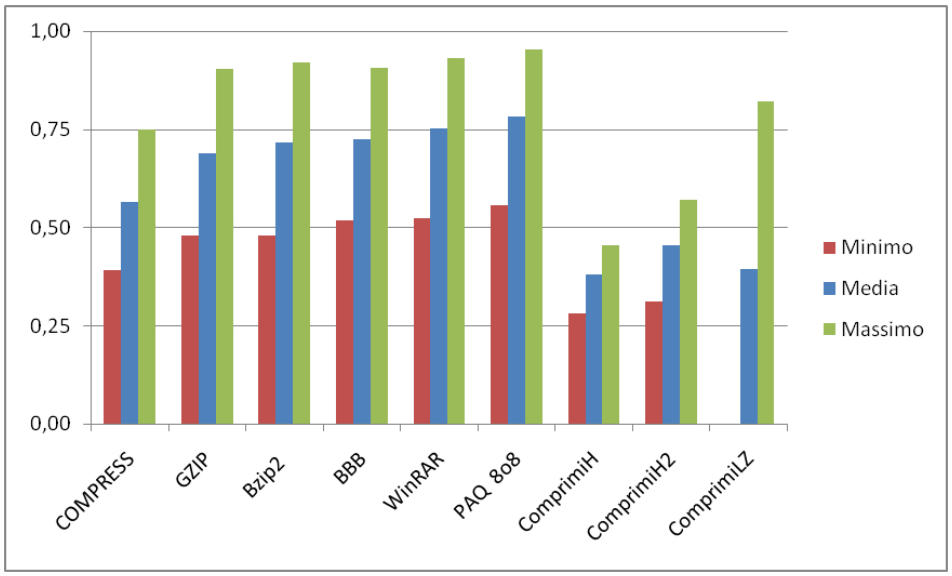

Figura 14: Grafico riassuntivo sugli indici di prestazione raggiunti da ogni programma)

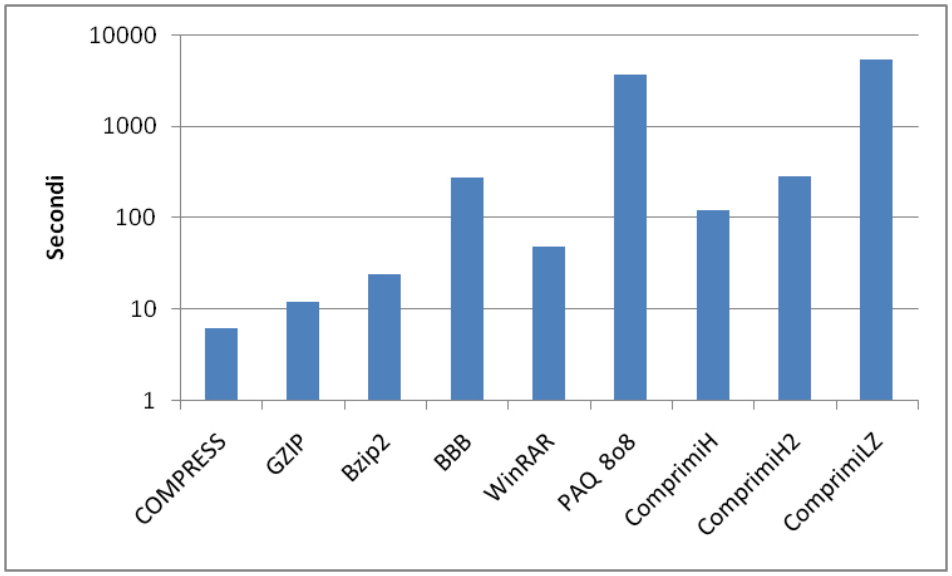

Figura 15: I tempi impiegati dai vari programmi per comprimere il file enwik8

	Capacità di compr.	Varianza dei risultati	Velocità di compr.	Memoria richiesta	Utilizzo attuale
ComprimiH	bassa	bassa	medio-bassa	bassa	alto
ComprimiH2	bassa	bassa	medio-bassa	bassa	
gzip	media	medio-alta	alta	media	alto(ZIP)
WinRAR	alta	media	medio-bassa		alto
ComprimiLZ	bassa	alta	bassa	media	
compress	medio-bassa	medio-bassa	alta	media	basso
bzip2	medio-alta	medio-alta	media	medio-alta	medio
bbb	medio-alta	media	medio-bassa	alta	
PAQ 8o8	alta	media	bassa	alta	

Tabella 12: Alcune caratteristiche dei programma analizzati.

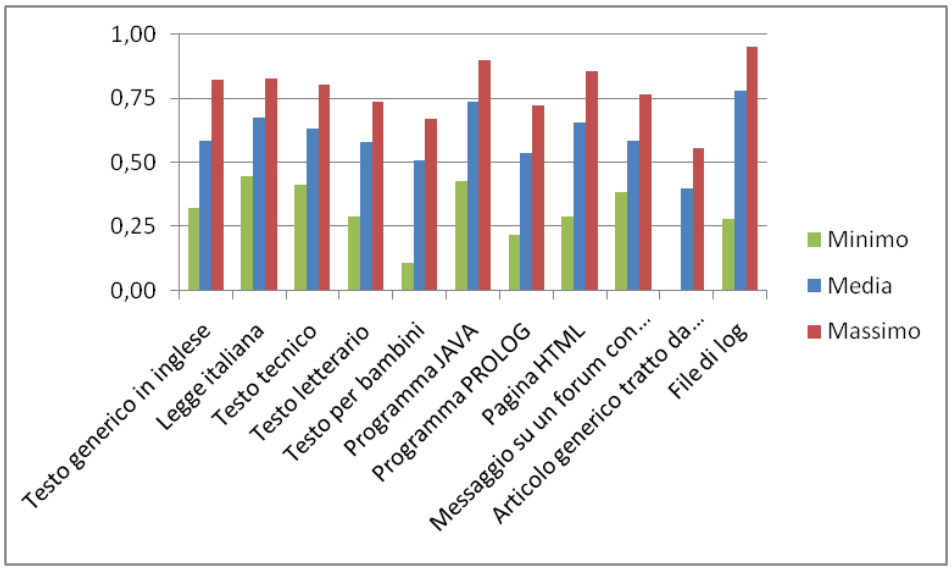

Figura 16: Grafico riassuntivo sugli indici di prestazione raggiunti sui differenti file considerati

Bibliografia

[1] Nark Nelson et al. *The Data Compression Book*. M&T Books, 2nd edition, 1996.

[2] www.wikipedia.com.

[3] http://prize.hutter1.net/.

[4] http://mattmahoney.net/text/text.html.

[5] http://maxcompress.narod.ru/.

[6] http://squeezechart.freehost.ag/bible.html.

[7] http://compression.ca.

[8] David Solomon. *Data Compression (The Complete Reference)*. Springer, 3rd edition, 2004.

[9] *RFC1951: DEFLATE Compressed Data Format Specification version 1.3*, 1996.

[10] www.gzip.org.

[11] *RFC1952: GZIP file format specification version 4.3*, 1996.

[12] www.gzip.org/algorithm.txt.

[13] www.win-rar.com.

[14] www.win-rar.com/index.php?id=24&kb_article_id=43.

[15] http://ncompress.sourceforge.net/.

[16] www.bzip.org.

[17] http://cs.fit.edu/m̃mahoney/compression.