LZW

Linea di comando per l'esecuzione

Versione a indici fissi:

gcc LZW_IF.c

Compressione: ./a.out –c *nomeFileDaComprimere nomeFileCompresso*
Decompressione: ./a.out –d *nomeFileCompresso nomeFileDecompresso*

Versione a indici variabili:

gcc LZW_IV.c -lm

Compressione: ./a.out –c *nomeFileDaComprimere nomeFileCompresso*
Decompressione: ./a.out –d *nomeFileCompresso nomeFileDecompresso*

Introduzione

Lzw è un algoritmo di compressione che sfrutta un dizionario. Il nostro compito è stato quello di realizzarlo nella versione base a indici fissi ed una più complessa a indici variabili. Lo scopo è quello di analizzare pro e contro di ciascuna versione, individuando inoltre le tempistiche e le prestazioni anche in correlazione con il tipo di file da elaborare.
Il concetto fondamentale di questo algoritmo si basa sull'invio(scrittura su file) di indici corrispondenti a sequenze di byte (parole) del dizionario, aggiungendone di nuove durante l'esecuzione del programma.
Il decompressore, leggendo gli indici delle parole e aggiornando il dizionario passo dopo passo come faceva precedentemente il compressore, è in grado di ricostruire il file originale.

Procedura dei lavori

Nel codice è scritto che la versione a indici fissi è stata sviluppata da Agnola Tommaso mentre quella a indici variabili da Sala Nicholas; tuttavia è giusto specificare che le due versioni sono state curate da entrambi i componenti del gruppo, soprattuto nel momento in cui si sono riscontrati problemi.
Dopo una prima parte di analisi ci siamo concentrati prevalentemente sulla versione a indici fissi: tale scelta è dovuta al fatto che volevamo fin da subito ottenere una versione del programma funzionante da cui partire per miglioramenti e sviluppi futuri. Una volta ultimata la prima versione siamo poi passati a quella ad indici variabili, trovandoci a buon punto dato che entrambe condividono alcune funzioni e concetti base.

Idee di base

Versione a indici fissi

L'idea base di questa versione è quella di salvare la corrispondenza numerica della parola sempre con 16 bit. Con tale scelta si possono inviare indici da 0 a 65535. Come è intuibile esistono file grandi a tal punto da necessitare un' aggiunta al dizionario di più di 65535 parole; la soluzione adottata per risolvere tale problema è quella di eliminare il dizionario una volta raggiunto l'indice 65535, sia in compressione che in decompressione. Tale cancellazione è un'operazione relativamente costosa per motivi di strutture dati che verranno presentate in seguito quindi i 16 bit scelti sono un compromesso per una compressione performante e tempistiche adeguate.

Versione a indici variabili

La versione a indici variabili, nasce dall'esigenza di risparmiare bit per ottimizzare la compressione dei file, quindi salvare gli indici su file anche con un numero di bit non multiplo di 8. Questo e' possibile tramite un buffer che viene caricato con gli indici e poi scaricato una volta pieno sul file compresso. Anche in questa versione abbiamo previsto una cancellazione del dizionario raggiunto un limite prestabilito.

Versioni intermedie

Prima di iniziare l'effettiva programmazione ci siamo concentrati sulla progettazione del codice, stabilendo delle idee e delle procedure base che avremmo poi dovuto implementare. Come spesso capita queste idee sono poi cambiate col passare del tempo per esigenze di prestazione o perché presentavano dei difetti concettuali; le vogliamo in ogni caso presentare perché secondo noi, anche se nel progetto finale non sono presenti, ne hanno costituito le basi e hanno contribuito in modo attivo o passivo al risultato finale.
La scelta delle strutture dati da utilizzare è stata fin dall'inizio individuata come principale indice che determina la velocità della compressione e della decompressione.
La principale funzione eseguita nella compressione è la ricerca di un una parola all'interno del dizionario ottenendone così l'indice mentre in decompressione la funzione principale è la ricerca di una parola dato l'indice.
Entrambi i processi, come è logico pensare, condividono l'aggiunta di una parola al dizionario e la cancellazione del dizionario.

Dizionario come lista

La prima idea è stata quella di sviluppare una struttura parola che contenesse un puntatore alla parola successiva ottenendo così una lista.

```
typedef struct elemento{
   int indice;
   carattereD *parola;
   struct elemento *punE;
}elementoD;
```

Per le parole invece si è invece pensato a una struttura simile chiamata carattere che punta al carattere successivo.

```
typedef struct carattere{
   unsigned char carattere;
   struct carattere *punC;
}carattereD;
```

Una volta ultimato abbiamo notato che le tempistiche erano considerevolmente alte. Analizzando nel dettaglio la struttura abbiamo capito che la ricerca comportava uno scorrimento della lista e un confronto fra singole parole con un costi computazionali relativamente alti.
Anche la decompressione sfruttava tale struttura, scelta non consona dato che tale processo svolge azioni completalmente diverse, date da esigenze diverse.
Come soluzione a tali problemi abbiamo deciso di modificare la struttura dizionario introducendo l'utilizzo di un'array (in fase di decompressione).

Dizionario come array (fase di decompressione)

```
typedef struct carattere{
        unsigned char carattere;
        struct carattere *punC;
}carattereD;
…
carattereD *dizionario[dim];
```

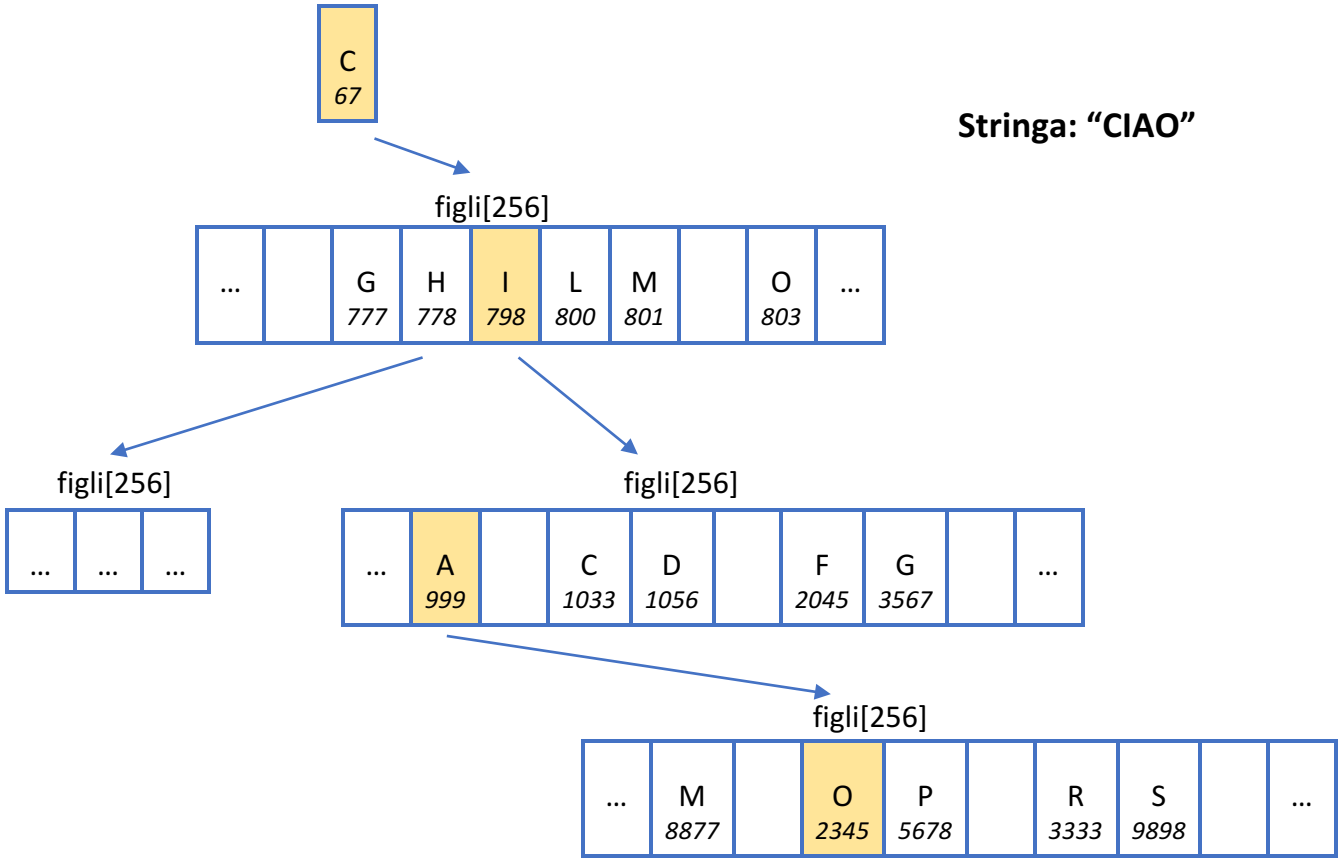
Con l'adozione di un array è stato possibile introdurre l'accesso diretto in decompressione, migliorandone considerevolmente le prestazioni. Con questa versione, una volta letto un indice in decompressione, per risalire alla parola bastava accedere alla posizione identificata dall'indice all'interno del dizionario.
In compressione invece, la ricerca di una parola all'interno del dizionario per risalire all'indice comportava ancora tempi troppo elevati. Siamo così giunti alla versione finale.

Versione finale

Dalle versioni precedenti abbiamo capito che l'unico problema restante per rendere pressochè ottimale anche la compressione era quello di individuare una struttura dati che consentisse una ricerca veloce di una parola per ottenere l'indice. La scelta è ricaduta su un trie.
Il nostro algoritmo si basa sulla ricerca all'interno del dizionario della corrispondenza più lunga nel file da compilare. Ciò avveniva ricercando nel dizionario delle porzioni incrementali del file da comprimere. Tale ricerca è molto dispendiosa perché come già detto il confronto di tante parole ha costi in termine di tempo molto alti.
Abbiamo così capito che sarebbe stato più veloce sviluppare un trie che ha per nodi un array di 256 elementi (i caratteri del codice ASCII) scorrendolo in verticale in base alle lettere da cercare.

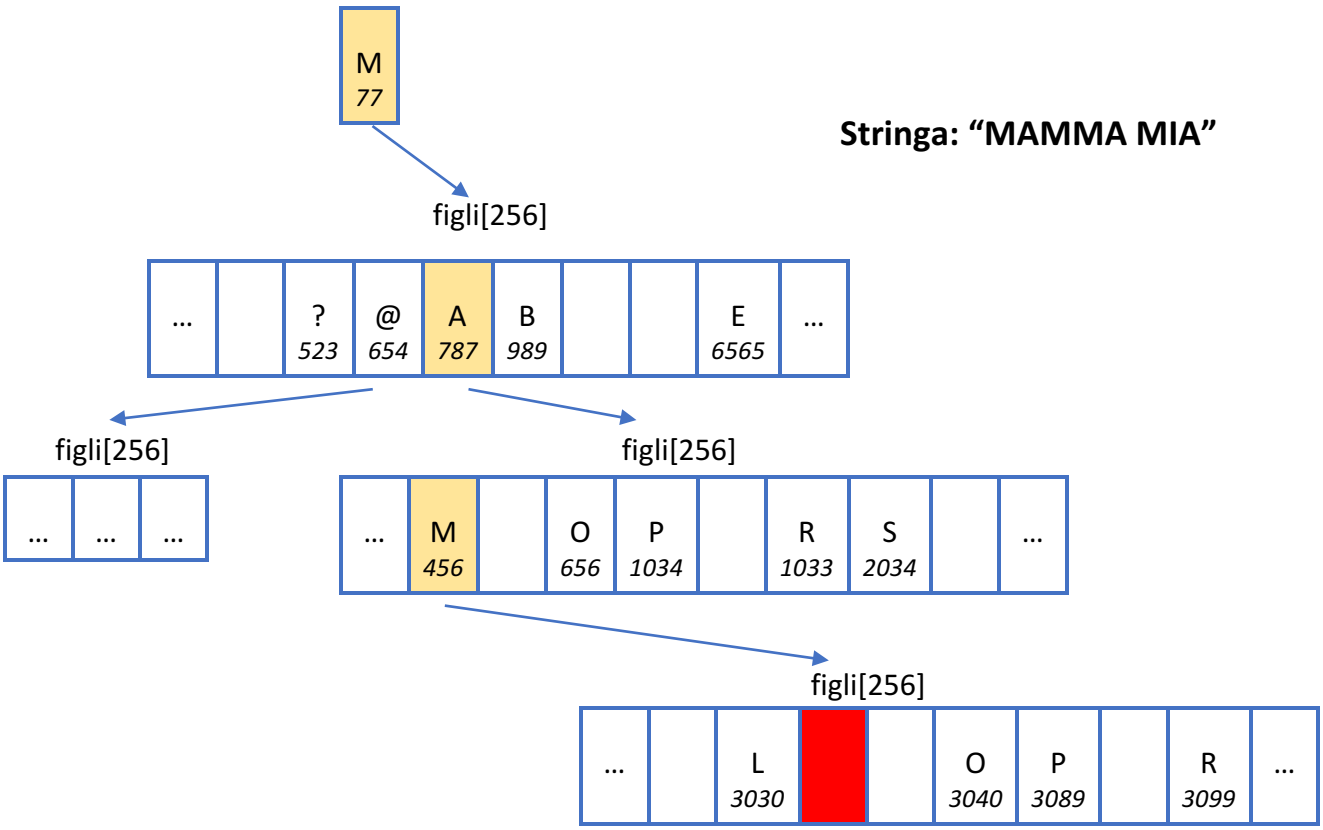
```
typedef struct nodo{
        unsigned int id;
        struct nodo **figli;
}Nodo;
```

Una volta individuata la parola, con il riferimento alla lettera finale possiamo ottenere l'indice univoco (id) che identifica l'intera parola nel dizionario, anche in decompressione.



Come visibile dal grafico non sono presenti tutte le lettere nell'array figli: esse vengono aggiunte non appena è necessario seguendo la procedura dettata dall'algoritmo.

Per assegnare l'id alle lettere utilizziamo un contatore globale (maxD) che incrementiamo ad ogni immissione di nuove parole nel dizionario. Nell'esempio presentato sopra, se l'ultima O ha indice 2345 l'intera parola CIAO viene identificata con l'intero non segnato 2345.



In questo esempio viene mostrata la ricerca lettera per lettera che procede fino a quando non viene più individuata la corrispondenza (in questo caso la seconda M non è presente nel sotto-albero della prima M). I passi successivi sono:

1. scrivere sul file compresso l’indice (456) dell’ultima lettera individuata;
2. aggiungere la lettera M nella posizione in cui prima non è stata individuata, con indice maxD;
3. incrementare maxD e ricominciare il processo di ricerca dalla lettera successiva alla corrispondenza trovata (dalla seconda M);

In decompressione invece la procedura è molto più semplice e la struttura è la stessa descritta nel capitolo *Versioni precedenti* nella sezione *Dizionario come array*:

1. viene letto dal file compresso l’indice della parola;
2. si accede in posizione indice nell’array dizionario; identificata la parola e la si scrive su un buffer;
3. si aggiunge al dizionario la parola precedentemente processata seguita dal primo carattere della parola attuale (il decompressore inizierà un passo dopo rispetto al compressore);
4. quando il buffer del punto 2 è pieno lo si scrive sul file;

Il punto 4 si giustifica con il fatto che la scrittura su file ogni volta che si trova una parola rallenterebbe notevolmente l’esecuzione del programma.

Indice	…	*2345*	*2346*	*2347*	…
Parola	…	CIAO	CIAO_	CIAO_M	…

Approfondimento di porzioni di codice

Trattiamo in questa sezione le porzioni di codice più rilevanti dell'algoritmo.

Funzione comprimi (versione a indici fissi e variabili)

Questa è la funzione principale della compressione. Nello specifico viene preso il file a blocchi (array *porzione[]*) e tramite un nodo navigatore *n* inizializzato alle radici del dizionario, si scorre quest'ultimo fino a quando *n* si trova in una posizione==NULL (il byte non è presente nell'albero); in questo caso si è trovata la corrispondenza più lunga presente nel dizionario. Il prossimo passo è andare a inserire nel buffer di scrittura l'indice dell'ultimo valore di n prima di aver assunto il valore NULL (salvato in *x*) ed aggiungere all'albero la nuova parola creata, per poi riportare il nodo navigatore alle radici.

```
void comprimi(char *nomeInput,char *nomeCompresso){
    Nodo **radici=(Nodo **)malloc(256*sizeof(Nodo *));
    inizializzaDizionarioC(radici);

    Nodo **n = radici;

    FILE *file;
    file = fopen(nomeInput, "rb");
    FILE *compresso;
    compresso = fopen(nomeCompresso, "wb");
    if(file){
        unsigned int i=0;
        unsigned int x = 0;
        unsigned char porzione[MAX];
        do{
            i=fread(porzione, sizeof(unsigned char), MAX, file);
            for(int k=0; k<i; k++){
                unsigned char e = porzione[k];

                if(n[e] == NULL){
                  Nodo* aggiunto = (Nodo *)malloc(sizeof(Nodo));
                  aggiunto->figli = (Nodo **)calloc(256,sizeof(Nodo *));
                  aggiunto->id = maxD;

                  n[e] = aggiunto;
                  aggiungiASalva(x, compresso);

                  n = radici;
                  maxD++;
                  k--;
                }else{
                  x = n[e]->id;
                  n = n[e]->figli;
                }

                if(maxD==(LIMITE)){
                    svuotaDizionario1(radici);
                    inizializzaDizionarioC(radici);
                }
            }
        }while(i==MAX);
        if(ftell(file)!=0)
            aggiungiASalva(x, compresso);
        scaricaFinale(compresso);
    }else{    printf("Errore");        }
    fclose(file);
    fclose(compresso);
}
```

Funzione decomprimi (versione a indici fissi)

Questo è un estratto del codice della decompressione. Come nella compressione, anche in questo caso viene letto il file a blocchi di grandezza *MAX*; tale blocco viene inserito nell'array porzione[], le cui celle contengono gli indici a 16 bit delle parole del dizionario. Tramite gli indici vengono estratte dall'array dizionario le parole e se ne costruiscono di nuove seguendo la logica dell'algoritmo. Una volta trovata la parola viene inserita in un buffer che verrà scritto sul file finale una volta pieno.
Anche in questo caso, per sincronizzare le operazioni con il compressore, il dizionario viene svuotato una volta inserita la parola con indice LIMITE-1. Tale scelta è dovuta al fatto che come specificato dalla documentazione dell'algoritmo, il decompressore è un passo indietro rispetto al compressore. Per chiarire, l'aggiunta al dizionario in decompressione consiste nell'aggiungere alla parola precedentemente elaborata il primo carattere della corrispondenza attuale. Il decompressore quindi inizia ad aggiungere parole al dizionario un passo dopo rispetto al compressore non avendo, al primo passo, una parola precedente da elaborare; da qui la scelta di LIMITE-1.

```
void deComprimi(char *nomeCompresso,char *nomeDecompresso)
{
    unsigned int dim=LIMITE;
    carattereD *dizionario[dim];
    inizializzaDizionarioD(dizionario);
    FILE *file;
    file = fopen(nomeCompresso, "rb");
    FILE *deCompresso;
    deCompresso = fopen(nomeDecompresso, "wb");
    if(file)
    {
        for(int i=0;i<MAX;i++)
            scaricaO[i]=0;
        int f = 0;
        unsigned int i=0;
        unsigned int cond=0;
        unsigned char c;
        unsigned short porzione[MAX];
        carattereD *precedente = (carattereD *)malloc(sizeof(carattereD));
        carattereD *corrente = (carattereD *)malloc(sizeof(carattereD));
        do{
            i=fread(porzione, 2, MAX, file);
            for(int k=0; k<i; k++)
            {
                corrente = cercaParola(dizionario, porzione[k]);
                if(corrente==NULL)
                {
                    corrente=creaParolaCopia(precedente);
                    aggiungiCarattereAParola(corrente, corrente->carattere);
                }
                if(f)
                {
                    aggiungiCarattereAParola(precedente,corrente->carattere);
                    aggiungiParolaADizionario(dizionario, precedente);
                }
                precedente = creaParolaCopia(corrente);
                while(corrente != NULL)
                {
                    c=corrente->carattere;
                    bufferizza(c,deCompresso);
                    corrente = corrente->punC;
                }
                if((limite+1)==(LIMITE))
                {
                    svuotaDizionario(dizionario);
                    inizializzaDizionarioD(dizionario);
                }
                f=1;
            }
        }while(i==MAX);
        scaricaFinale2(deCompresso);
    }else{  printf("Errore");       }
    fclose(file);
    fclose(deCompresso);
}
```

Funzione decomprimi (versione a indici variabili)

La funzione di decompressione nella versione a indici variabili dell'algoritmo è differente in quanto gli indici andranno estratti rispettando nBit (il numero di bit con cui estrarre l'indice attuale, basato sul massimo indice presente in dizionario). Leggiamo un blocco di file e lo mettiamo in un array *porzione* dove ogni cella corrisponde a 32 bit, ognuno di questi 32 bit andrà a comporre una cella dell'array *salva* e la funzione prelevaIndice provvederà a ritornare l'indice esatto.

```
iP = 0;
i=fread(porzione, 4, MAX, compresso);
aggiungiCompletoASalva(porzione[iP]);//inizializzo salva
do{
        x = prelevaIndice(porzione, &i, compresso); //prelevo l'indice
        identificaEAggiungiD(dizionario,x);

        if(residui == 0){
            scaricaD(prec, deCompresso);
        }else{
            //fprintf(stderr, "residui e' %d",residui);
        }

        if(maxD==(LIMITE))
        {
                svuotaDizionario(dizionario);
                inizializzaDizionarioD(dizionario);
        }
}while(iP < i);
```

Funzione prelevaIndice (versione a indici variabili)

Questa funzione opera su *salva*, un array di short che contiene effettivamente le cifre binarie estratte dal file, tramite degli shift binari e delle somme va a ricostruire l'indice effettivo che il decompressore richiede.

```
unsigned int prelevaIndice(unsigned int porzione[], int *len, FILE *fptr)
{
    int l = *len;
    short c = 0;
    unsigned int r = 0;
    int limite = iS + nBit;

        for(int i=iS;i<limite;i++)
        {
            if(iS<32 && iP<l)
            {
                r = r<<1;
                r += salva[iS];
                iS++;
                c++;
            }
            else if(iP<l) //abbiamo letto tutto l'array salva, dobbiamo ricaricare
            {
                carica(porzione, fptr, len);
                limite = (nBit - c) + 1;
                i = 0;
            }else
            {
                residui = c;//blocco la lettura, e mantengo i dati finora letti in
                break;
            }
        }

    return r;
}
```

Funzione identificaEAggiungiD (versione a indici variabili)

Questa funzione viene usata dalla versione a indici variabili. Dato l'indice essa aggiunge al dizionario la nuova parola, e uguaglia *prec* (variabile che identifica la corrispondenza precedente) alla parola appena aggiunta, così che la funzione deComprimi debba solo salvare su file i byte corrispondenti a prec una volta richiamata questa funzione. Caratteristica principale di questa funzione è la gestione dell'arrivo di un indice che corrisponde al prossimo indice da creare, quindi a una parola ancora non presente. Essendo la parola, aggiunta in decompressione, la corrispondenza precedente più il primo carattere della corrispondenza attuale basterà in questo caso richiamare la funzione aggiungiADizionario (dato un carattere lo aggrega alla parola precedente e aggiunge questa parola creata al dizionario) passandogli *prec->carattere* (il primo carattere della corrispondenza precedente).

```
void identificaEAggiungiD(carattereD *dizionario[], unsigned int indice){
        //aggiungo nuova corrispondenza al dizionario e aggiorno prec con la corrispondenza attuale
        if(prec == NULL){
            prec = (carattereD *)malloc(sizeof(carattereD));
            prec = dizionario[indice];
        }else{
            if(indice == maxD){ //siamo nel caso parola creata al momento
                    aggiungiADizionario(dizionario, prec->carattere);
            }else{
                  aggiungiADizionario(dizionario, dizionario[indice]->carattere);
            }

            maxD++;
            aggiornaNBit(1);
            prec = dizionario[indice];
        }
}
```

Funzione inizializzaDizionarioC (versione a indici fissi e variabili)

Questa funzione consente di precaricare il dizionario con i 256 byte di base. Essa viene richiamata all'avvio del processo di compressione e in un momento successivo alla cancellazione del dizionario.

```
void inizializzaDizionarioC(Nodo **radici){
    for(int i=0; i<256; i++)
    {
        radici[i] = (Nodo *)malloc(sizeof(Nodo));
        radici[i]->id = i;
        radici[i]->figli=(Nodo **)calloc(256,sizeof(Nodo *));
    }
    maxD = 256;
}
```

Utilizzo del varInt (versione a indici variabili)

Per garantire anche nella decompressione l'accesso diretto in un array per risalire alla parola dato l'indice, sorge il problema di come specificare la dimensione di tale array. Al decompressore è quindi necessario conoscere l'indice massimo raggiunto dal dizionario in compressione alla fine delle operazioni, così da creare l'array dizionario delle dimensioni adatte. La nostra solzuone è stata quella di scriverlo in testata al file compresso. Dato che tale numero però può variare a seconda del file da comprimere, non si può conoscere in anticipo quanti bit sono necessari per

scriverlo su file. La soluzione da noi adottata è il varInt. All'inizio del file compresso vengono scritti una successione di byte: nei primi 7 bit di ogni byte è contenuto l'effettivo numero mentre l'ultimo bit indica se bisogna leggere il prossimo byte (che conterrà ancora parte del numero) oppure no. Una volta raggiunto lo zero come ottavo bit di un byte, abbiamo deciso di scrivere un ulteriore byte che rappresenta quante volte è stato svuotato il dizionario in fase di compressione.
In seguito inizia il "vero" file compresso.

```
unsigned char indicatore = 0;
for(int y=n-1; y>=0; y--){
    if(y!=0){
        indicatore = 1<<7;
    }
    indicatore += ((maxD & (127 << y*7))>>y*7);
    fwrite(&indicatore, sizeof(unsigned char), 1, compresso);
    indicatore = 0;
}
```

Funzione svuotaDizionario (versione a indici fissi e variabili)

Queste funzioni consentono di svuotare il dizionario richiamando una serie di *free()* in modo ricorsivo sui nodi. Si scorre ogni ramo dell'albero fino a raggiungere un nodo uguale a NULL; a questo punto, ricorsivamente, si libera la memoria occupata dal padre di quest'ultimo. Una volta liberati tutti i nodi dei rami, viene liberata anche la memoria occupata dalle radici.
Tali funzioni vengono utilizzate anche nella versione a indici variabili per principalmente un motivo: quando viene creato l'array dizionario in decompressione si deve specificare una dimensione. Tale dimensione inevitabilmente non può essere infinita quindi si è scelto il massimo numero rappresentabile da un unsigned long (2^32-1). Una volta inserita nel dizionario la parola che ha per indice tale numero viene svuotato il dizionario e ripopolato dei primi 256 caratteri del codice ASCII.

```
void svuotaDiz(Nodo *n){
    if(n!=NULL){
        for(int i=0;i<256;i++){
            if(n->figli[i]!=NULL){
                svuotaDiz(n->figli[i]);
            }
        }
        free(n->figli);
        free(n);
    }
}

void svuotaDizionario1(Nodo **radici)
{
    for(int i=0;i<256;i++){
        svuotaDiz(radici[i]);
    }
}
```

Funzione aggiungiCarattereAParola (versione a indici fissi e variabili)

Come precedentemente esposto, in decompressione costruiamo delle parole aggiungendo nuovi caratteri a parole già esistenti nel dizionario. Prima di fare ciò costruiamo una parola copia per non sovrascrivere nel dizionario la parola modificata (viene richiamata la funzione *creaParolaCopia()*). Per aggiungere un carattere ad una parola utilizziamo tale funzione:

```
void aggiungiCarattereAParola(carattereD *p, unsigned char c)
{
    carattereD *prec=p;
    while(p != NULL)
    {
        prec=p;
        p = p->punC;
    }
    p=(carattereD *)malloc(sizeof(carattereD));
    prec->punC=p;
    p->carattere=c;
    p->punC=NULL;
}
```

Essa scorre la parola fino all'ultimo carattere e fa puntare quest'ultimo al carattere da aggiungere alla parola, di fatto creando una parola nuova.

Funzione aggiornaNBit

La versione ad indici variabili necessita di una gestione diversa per il salvataggio degli indici. Abbiamo deciso di creare una variabile nBit che corrisponde al numero di bit necessario per salvare l'indice massimo del dizionario in quel momento, in questo modo sia compressore che decompressore sapranno a quanti bit scrivere/leggere l'indice attualmente in elaborazione. Questa funzione, richiamata ad ogni aggiunta al dizionario in compressione e in decompressione, consente di aggiornare la variabile nBit. Essendo il compressore un passo avanti al decompressore, il decompressore dovrà calcolare nBit in base al massimo del dizionario attuale + 1.

```
void aggiornaNBit(int fase){
        if(fase){
            nBit = (int)(log2(maxD+1)+1);
        }else{
            nBit = (int)(log2(maxD)+1);
        }
}
```

Funzione aggiungiASalva

Funzione che consente di salvare l'indice in compressione, in un array di 32 short corrispondente a un buffer nel quale ogni cella corrisponde a una cifra binaria dell'indice, che ogni volta pieno viene scaricato in un secondo buffer di presalvataggio indici.

```
void aggiungiASalva(unsigned int x, FILE *fptr)
{
    for(int i=1;i<=nBit;i++)
    {
        if(iS<32)//il buffer di salvataggio non e' pieno
        {
            salva[iS] = (x >> (nBit - i)) & 1;
            iS++;
        }
        else {  i--; scarica(fptr); }
    }
}
```

Prestazioni

Nei nostri algoritmi vi sono alcuni fattori modificabili che possono aumentare o diminuire le prestazioni in tempo o rapporto di compressione. Non escludiamo l'adozione di modifiche di questi fattori in determinate situazioni dove si predilige il rapporto di compressione minore oppure altre dove si necessitano tempistiche minori a discapito dell'effettiva compressione.
Nella versione a indici fissi è possibile cambiare il numero di bit scritti per rappresentare un indice. Con tale soluzione si potrebbero inserire più parole all'interno del dizionario e cancellare meno volte il contenuto dello stesso. Sicuramente però il rapporto di compressione aumenterebbe, soprattutto nel file piccoli.
In entrambe le versioni è possibile modificare la grandezza del buffer di lettura. Aumentando tale valore potrebbe velocizzare le operazioni senza grandi svantaggi. Abbiamo deciso comunque di tenerlo a 1MB per questioni di comodità e cercare di non appesantire troppo la memoria.

Versione a indici fissi

Nome file	Dimesione iniziale	Dimensione compresso	Tempo compressione	Tempo decompressione	Rapporto di compressione
32k_ff	32.8 kB	512 byte	0.002 s	0.025 s	1.56 %
32k_random	32.8 kB	55.3 kB	0.050 s	0.023 s	168.59 %
alice.txt	167.5 kB	79 kB	0.047 s	0.017 s	47.16 %
empty	0 byte	0 byte	0.001 s	0.001 s	100.00%
ff_ff_ff	3 byte	4 byte	0.002 s	0.002 s	133.33 %
Immagine.tiff	3.4 MB	4 MB	2.483 s	0.287 s	117.64 %
bitmap.bmp	95.6 MB	3.0 MB	4.681 s	347.633 s	3.13 %
divina.txt	557.2 kB	252.7 kB	0.190 s	0.052 s	45.35 %

Come visibile dalla tabella alcuni file non vengono compressi. La principale motivazione crediamo sia la struttura dell'algoritmo ed i difetti strutturali dello stesso. Non possiamo escludere il fatto che ci potrebbero essere soluzioni migliori della nostra.
Nei file piccoli, come *ff_ff_ff* la versione a indici fissi deficita in compressione dato che ogni parola viene rappresentata in 16 bit quando in realtà un semplice carattere ne occupa solo 8. Non elaborando molte parole, ogni parola è pressochè un carattere.
Si vedono però i pregi del nostro algoritmo nei file relativamente grandi come *bitmap.bmp*, immagine non precedentemente compressa, dove vi è un fattore di compressione del 3,13% (a parer nostro un ottimo risultato). Le tempistiche molto elevate nella decompressione di questo file sono dovute al fatto che, trovando corrispondenze nel dizionario molto grandi, l'algoritmo è obbligato ad elaborarle rallentando il processo.
La non compressione di *immagine.tiff* è dovuta al fatto che tale immagine è già di per se compressa; non trovando molte corrispondenze nel dizionario verranno scritti molti indici e quindi byte in più nel file compresso.

Versione a indici variabili

Nome file	Dimesione iniziale	Dimensione compresso	Tempo compressione	Tempo decompressione	Rapporto di compressione
32k_ff	32.8 kB	292 byte	0.002 s	0.014 s	0.89 %
32k_random	32.8 kB	48.0 kB	0.036 s	0.008 s	146.34 %
alice.txt	167.5 kB	71.1 kB	0.060 s	0.019 s	42.44 %
empty	0 byte	4 byte	0.001 s	0.0005 s	
ff_ff_ff	3 byte	6 byte	0.001 s	0.001 s	200.00 %
Immagine.tiff	3.4 MB	errore			
bitmap.bmp	95.6 MB	errore			
divina.txt	557.2 kB	229.1 kB	0.183 s	0.064 s	41.11 %

Come visibile dalla tabella in alcuni file le prestazioni sono migliori rispetto alla versione ad indici fissi, data la struttura più avanzata dell'algoritmo.
Purtoppo però abbiamo riscontrato alcuni problemi in compressione che discuteremo in seguito.
Come nel caso precedente, nei file piccoli non avviene una effettiva compressione data la logica dell'algoritmo. Inoltre, a differenza della versione a indici fissi, questa versione include anche un'intestazione la struttura del varInt (descritta nei capitoli precedenti) che aumenta di qualche byte (al massimo 5) la dimensioen del file compresso. In questa versione è presente anche un byte per indicare il numero di volte che il compressore ha svuotato il dizionario.
Nei file grandi invece le tempistiche e i relativi rapporti di compressione sono a nostro parere buoni. Si puo notare come la versione a indici variabili riesca ad comprimere meglio rispetto a quella a indici fissi, tranne in alcuni casi come per il file empty, dove vengono comunque scritti dei byte corrispondenti al varint iniziale e al numero che corrisponde al numero di volte in cui il compressore ha svuotato il dizionario.
Nella prossima sezione discuteremo di eventuali motifiche che potrebbero migliorare le prestazioni che, per motivi di tempo, non siamo riusciti a sviluppare.

Problemi noti

Per problemi di tempo non siamo riusciti a risolvere determinate problematiche già evidenziate in precedenza. Sicuramente vi sono soluzioni ai suddetti problemi.

- Vi sono anche sicuramente modi per velocizzare la decompressione. Abbiamo individuato come fattore principale che determina rallentamenti la costruzione della nuova parola da inserire nel dizionario. Nello specifico, per aggiungere un nuovo carattere ad una parola gia esistente bisogna scorrere quest'ultima carattere per carattere mediante i puntatori e secondo noi tale operazione è dispendiosa. Come miglioria si potrebbe tenere un puntatore che punti all'ultimo elemento della parola così da accedere più velocemente al carattere a cui aggiungere un carattere successivo.
- Come spiegato in precedenza nella sezione del varInt, utilizziamo un solo byte per specificare quante volte è stato svuotato il dizionario in compressione. Il limite di tale soluzione è che si potrà svuotare il dizionario un massimo di 255 volte. Una miglioria potrebbe essere l'utilizzo del varInt anche per questo dato.
- La decompressione di file più o meno grandi (circa 600 kB) nella versione a indici variabili, da un errore di segmentation fault, che non siamo riusciti a risolvere dato che avevamo problemi con priorità maggiore.