Nome: Alberto Biliotti
Matricola: 7109894

Data security and privacy
Terza parte esercizi

---

1 Esercizi di programmazione

1.1 Esercizio 3.2

Ho implementato la soluzione descritta di seguito nel file "huffman.py" allegato.
Per iniziare ho definito un metodo "trovaMinimiUnisci" che, dato un dizionario che associa ad ogni simbolo la sua probabilità di occorrenza, fonde i due simboli meno probabili in un unico simbolo, assegnandogli probabilità pari alla somma di quella dei due simboli appena uniti, eliminando poi tali simboli dal dizionario, restituendo infine il dizionario così aggiornato e la probabilità del nuovo simbolo. Ho poi implementato il primo passo dell'algoritmo di Huffman nel metodo "primoPassoHuffman" che, dato in input il dizionario con le probabilità osservate delle singole lettere, restituisce una tupla di due elementi, che possono essere a loro volta tuple di elementi oppure lettere dell'alfabeto. Tale tupla tiene conto delle unioni dei simboli effettuate grazie al metodo appena visto, infatti ogni coppia rappresenta l'unione dei simboli meno probabili ad ogni iterazione di un ciclo while, che termina quando la somma delle probabilità dei termini che ho appena unito è pari ad 1.
Posso quindi implementare anche il secondo passo dell'algoritmo che, presa in input la tupla di due elementi ricavata al passo precedente ed un dizionario vuoto, concatena i simboli a destra ed a sinistra della tupla con 1 e 0. Nel caso poi che uno di questi simboli corrisponda a sua volta ad una tupla, riapplico il metodo ricorsivamente su ciascuno dei due simboli, fino a che non ottengo il dizionario contenente la codeword associata a ciascuna lettera dell'alfabeto.
A questo punto posso implementare un altro metodo, chiamato "comprimiHuffman", che, presa in ingresso una qualsiasi stringa di testo, dopo aver ricavato il dizionario contenente le probabilità osservate di ciascuna lettera, comprimerà la stringa passata in ingresso sfruttando i due metodi implementati in precedenza, ottenendo così il dizionario che associa ad ogni lettera la relativa codeword, che userò per comprimere la stringa secondo la codifica appena ottenuta, che restituirò insieme al dizionario della codifica che ho usato per comprimere la stringa.
Infine ho implementato un metodo "decomprimiPrefixFree" che, presa in in input una qualsiasi stringa compressa con un codice prefix-free ed il relativo dizionario contenente le codeword con cui è stata compressa la stringa, scambia per ogni entry del dizionario la chiave con il contenuto di tale entry, per poi decomprimere la stringa andando semplicemente a controllare in successione i bit finché non incontro una sequenza di bit che è presente tra le chiavi del dizionario appena generato; la relativa lettera verrà quindi concatenata alla parte di stringa già decompressa. Ripeterò lo stesso procedimento sulla parte rimanente della stringa compressa finché non otterrò la stringa decompressa alla fine dell'esecuzione.

1.2 Esercizio 3.4

La soluzione di questo esercizio, implementata in Python, è contenuta nel file "codiciLZ.py" in allegato.
Per prima cosa ho creato due metodi, "associaletterabin" e "associanumbin" che convertono rispettivamente le lettere dell'alfabeto e gli interi nella loro rappresentazione binaria, in particolare ho considerato le lettere dell'alfabeto inglese, con la lettera "a" a cui ho assegnato valore 0, mentre, per quanto riguarda il metodo per rappresentare gli interi, ho inserito un altro valore in input che indica il numero di bit su

cui si desidera ottenere tale rappresentazione (se la lunghezza della rappresentazione binaria è minore di quella richiesta aggiungo dei bit 0 di riempimento finché non ottengo la lunghezza desiderata).
Fatto ciò, posso implementare il metodo per comprimere la stringa passata in input, che si chiamerà "comprimiStringaLZ". Tale metodo definisce un dizionario che contiene inizialmente una entry corrispondente al blocco vuoto ed una corrispondente alla prima lettera della stringa in analisi, insieme ad un contatore del numero di blocchi osservati e alla variabile che conterrà la stringa compressa, che inizializzo con la codifica binaria della prima lettera della stringa (escludo il puntatore iniziale al blocco nullo in quanto sarebbe identico in tutte le stringhe compresse). Il metodo inizia quindi a scorrere la stringa interrompendosi quando incontra un blocco che non era ancora presente nel dizionario, in tal caso aggiorna il numero di blocchi già osservati, per poi inserire nel dizionario la entry associata al nuovo blocco, che contiene il valore del contatore che ho aggiornato prima insieme alla codifica associata al blocco che ottengo concatenando il puntatore in binario al blocco prefisso che ricavo eliminando l'ultima lettera dal blocco in analisi insieme alla rappresentazione binaria di tale lettera. A questo punto concatena la rappresentazione del nuovo blocco appena ottenuta con la parte di stringa già compressa, per poi riprendere a scorrere la stringa finché non incontrerò un nuovo blocco non ancora osservato. Alla fine dell'esecuzione è possibile che sia rimasto un ultimo suffisso della stringa in analisi, di cui però è già presente il corrispondente blocco nel dizionario. In tal caso concateno alla stringa compressa la rappresentazione in binario del puntatore al blocco che ottengo eliminando l'ultima lettera della stringa rimanente e la rappresentazione binaria della lettera stessa.
Per quanto riguarda il metodo di decompressione della stringa, chiamato "decomprimiStringaLZ", definisco anzitutto un contatore che tiene traccia del numero di blocchi visitato, insieme a un dizionario che assocerà ad ogni intero n, rappresentante la n-esima iterazione del ciclo sulla stringa compressa, il relativo blocco che ho analizzato in quella specifica iterazione.
Dopo aver analizzato il primo blocco (corrispondente alla rappresentazione della prima lettera della stringa) come caso a parte, in quanto manca il primo bit 0 rappresentante il puntatore alla stringa vuota, ricavo, per ogni blocco presente nella stringa compressa, prima la stringa binaria contenente il puntatore alla entry del dizionario contenente il prefisso del blocco in analisi, a cui andrò a concatenare la lettera rappresentata nei cinque bit successivi, ottenendo così il blocco decompresso, che andrò a salvare nel dizionario usando come chiave il contatore del numero di esecuzioni eseguite fino ad ora e a concatenare con il resto della stringa già decompressa. Ripeto tale procedimento finché non esaurisco la stringa compressa, ritornando così l'intera stringa decompressa. A questo punto posso confrontare i risultati dei due algoritmi di compressione definiti in questo e nel precedente esercizio: per fare ciò userò un nuovo file, chiamato "confronto.py". In tale file definiremo un metodo per "pulire" una stringa da caratteri non alfabetici e per rendere tutte le lettere minuscole, inoltre sfrutteremo anche il metodo "unidecode" dell'omonima libreria per sostituire eventuali lettere accentate con le relative versioni non accentate.
Ho quindi creato dieci file, ciascuno dei quali contenente testi con più di mille lettere in lingua inglese, in particolare ho utilizzato: il primo capitolo del "Signore degli Anelli", dello "Hobbit", del "Silmarillion" e dei "Figli di Hurin" di Tolkien, il primo capitolo della "Guida Galattica per Autostoppisti" di Douglas Adams, il primo capitolo de "Il Vecchio e il Mare" di Ernest Hemingway, il primo capitolo di "Moby Dick" di Herman Melville, il primo capitolo de "Le Avventure di Tom Sawyer" di Mark Twain, insieme alla dichiarazione di indipendenza degli Stati Uniti d'America e la prima sezione della Costituzione statunitense. Ho quindi ordinato tali file, assegnando a ciascuno un nome pari a "testo" più un numero progressivo da 1 a 10, in modo da poter iterare facilmente su tutti i file in unico ciclo for.
A questo punto, dentro tale ciclo, in ogni iterazione apro un file diverso e ne pulisco il contenuto grazie al metodo implementato in precedenza, per poi comprimere tale stringa pulita utilizzando prima l'algoritmo di Huffman e successivamente quello di Lempel-Ziv.
Fatto ciò, posso testare la correttezza delle implementazioni viste in precedenza controllando che le stringhe decompresse ottenute dall'applicazione dei due metodi siano identiche alla stringa di partenza.
Ora posso quindi ricavare la percentuale di compressione che ottengo da entrambi i metodi, tenendo conto

che la lunghezza in bit della stringa di partenza è pari a cinque volte il numero di lettere che essa contiene, in quanto ogni lettera viene rappresentata con cinque bit nella stringa non compressa.
Osservo quindi che l'algoritmo di Huffman comprime meglio la stringa in sette casi su 10 rispetto all'algoritmo di codifica LZ, mantenendo sempre una percentuale di compressione compresa tra il 15.5% ed il 17.5%. L'algoritmo LZ, invece, ha molta più variabilità andando dal 5.5% di compressione ottenuto per il primo capitolo della "Guida Galattica per Autostoppisti" al 19.31% di compressione per la prima parte della Costituzione Statunitense.
Questa differenza così ampia è spiegata dal fatto che l'algoritmo di codifica di Lempel-Ziv performa meglio quando lavoro con sequenze ridondanti, come accade appunto nella Costituzione degli USA che ha molti termini ed espressioni ripetuti molte volte al suo interno, essendo composta di articoli con formule estremamente ripetitive, oppure con testi estremamente lunghi, come accade per il primo capitolo del "Signore degli Anelli".
L'algoritmo di Huffman restituisce invece sempre un codice perfetto per la stringa in analisi, ovvero un codice che possiede lunghezza media minima; per questo la percentuale di compressione rimane sempre simile, e dipenderà unicamente dall'entropia della distribuzione delle lettere nel testo, in quanto, per il primo teorema di Shannon, la lunghezza media di un codice perfetto è maggiore o uguale dell'entropia di Shannon della distribuzione delle lettere nel testo e minore di tale valore sommato di uno. Avendo usato solo testi in lingua Inglese, si avrà quindi che l'entropia della distribuzione delle lettere dei vari testi non dovrebbe variare molto, e di conseguenza non varierà molto nemmeno la percentuale di compressione dell'algoritmo di Huffman, com'è effettivamente accaduto nel nostro confronto.

2 Esercizio di approfondimento

2.1 Esercizio 2.3

a. Nel caso in cui tutti i piccioni riescono ad arrivare a destinazione senza essere abbattuti, avrei che la capacità del canale è esattamente di 8 bit. Infatti, la matrice del canale non rumoroso corrisponde alla matrice identità, che è debolmente simmetrica (ed è anche simmetrica), quindi posso applicare la proposizione che afferma che la capacità C del canale è pari a:

$$C = log_2|Y| - H(r) \tag{1}$$

Dove Y è l'alfabeto di uscita del canale (ossia i $2^8 = 256$ messaggi componibili con 8 bit) e H(r) è l'entropia di una qualsiasi riga della matrice del canale. Nel nostro caso, l'entropia associata ad una riga della matrice del canale è pari a zero, in quanto ho solo un messaggio possibile che è quello corretto che viene restituito con probabilità pari ad 1. Quindi avrò che la capacità del canale sarà pari a $log_2|Y|$ che è uguale ad 8. Volendo infine ricavare la capacità in bit per ora è sufficiente considerare che in un'ora arrivano a destinazione 12 piccioni, quindi in totale il canale avrà una capacità oraria di $12 * 8 = 96$ bit/ora.

b. Analizziamo ora invece il caso in cui il nemico riesca ad abbattere e sostituire una parte α dei piccioni. Si avrà che in una parte $1-\alpha$ dei casi il piccione non verrà abbattuto, mentre nella restante frazione α dei casi il messaggio verrà estratto casualmente tra tutti i 2^8 messaggi componibili con otto bit. Possiamo osservare come la probabilità che arrivi a destinazione il messaggio corretto è pari ad $1-\alpha$ (probabilità di non abbattimento del piccione) più la probabilità che il piccione venga sì abbattuto, ma il messaggio che porta sostituito con uno identico, che è pari a $\alpha * \frac{1}{2^8}$. La probabilità di ricevere un messaggio coretto sarà quindi pari a $1 - \alpha + \alpha * \frac{1}{2^8}$, ossia $1 - \alpha\frac{2^8-1}{2^8} = 1 - \alpha\frac{255}{256}$, mentre quella che arrivi uno in particolare degli altri 255 messaggi non coretti sarà pari a $\frac{\alpha}{256}$. Di conseguenza la matrice associata al canale, di dimensioni 256X256, avrà sulla diagonale valori pari a $1 - \alpha\frac{255}{256}$ (la diagonale corrisponde al caso in cui in uscita ho lo stesso messaggio che ho ricevuto in entrata nel canale), mentre fuori dalla diagonale tutti valori pari a $\frac{\alpha}{256}$ (corrispondenti ai casi in cui il

messaggio in uscita dal canale è diverso da quello che avevo in ingresso). Tale matrice è debolmente simmetrica (è anche simmetrica), in quanto ogni riga presenta gli stessi elementi ordinati in maniera diversa e la somma degli elementi sulle colonne è costante (ed uguale ad 1). Grazie a questo posso sfruttare la proprietà 1 per ottenere la capacità del canale, che sarà quindi pari a $8 - H(r)$ dove $H(r)$ è l'entropia di una qualsiasi riga (in seguito considereremo la prima riga per semplicità).
Per poter ricavare tale entropia devo però prima dimostrare il seguente risultato:
Data una generica distribuzione $\mathbf{p} = (p_1, ..., p_n)$ si ha che:

$$H(p_1, ..., p_n) = H(p_1) + (1 - p_1)H(\frac{p_2}{1 - p_1}, ..., \frac{p_n}{1 - p_1}) \tag{2}$$

Ciò si può dimostrare ricorrendo alla chain rule che afferma che, date due variabili aleatorie discrete X e Y, l'entropia della distribuzione di probabilità congiunta può essere scritta come:

$$H(X, Y) = H(X) + H(Y|X) = H(Y) + H(X|Y) \tag{3}$$

Possiamo quindi definire, una variabile aleatoria X distribuita come $\mathbf{p}$, e una variabile aleatoria ausiliaria Y che assume valore $Y = 1$ se $X = x_1$ e $Y = 0$ altrimenti.
Applicando la chain rule posso quindi scrivere l'entropia della distribuzione congiunta di queste due variabili come $H(X, Y) = H(X) + H(Y|X) = H(Y) + H(X|Y)$. Osservo che $H(Y|X)$ è uguale a zero, in quanto l'informazione su X mi da già la certezza della conoscenza del valore assunto da Y (1 se $X = x_1$, 0 altrimenti).
Quindi posso affermare che:

$$H(X) = H(Y) + H(X|Y) \tag{4}$$

Per definizione di entropia differenziale posso però riscrivere tale formula come $H(X) = H(Y) + p_1 * H(X|Y = 1) + (1 - p_1) * H(X|Y = 0)$. Tale risultato può essere ulteriormente semplificato considerando che $H(X|Y = 1)$ è uguale a zero, in quanto se osservo $Y = 1$ allora saprò già che $X = x_1$ per cui non avrò più alcuna incertezza sul risultato. Posso quindi riscrivere l'equazione come:

$$H(X) = H(Y) + (1 - p_1) * H(X|Y = 0) \tag{5}$$

Ma ciò non è altro che un modo differente per scrivere l'equazione 2 in quanto: la variabile aleatoria X ha distribuzione $\mathbf{p}$ mentre l'entropia di Y corrisponde all'entropia binaria di p_1 in quanto Y possiede distribuzione di Bernoulli con probabilità di successo pari a p_1. Infine notiamo come $H(X|Y = 0)$ sia equivalente a $H(\frac{p_2}{1-p_1}, ..., \frac{p_n}{1-p_1})$, poiché $(\frac{p_2}{1-p_1}, ..., \frac{p_n}{1-p_1})$ corrisponde alla distribuzione di probabilità $\mathbf{p}$ in cui però ho la certezza che non si sia verificato l'evento $X = x_1$ in quanto ho osservato $Y = 0$.
Dimostrato ciò posso quindi calcolare l'entropia associata alla prima riga della matrice del canale (tale entropia è costante per tutte le righe della matrice), ricordando che tale riga è definita come $(1 - \alpha\frac{255}{256}, \frac{\alpha}{255}, ..., \frac{\alpha}{255})$, come:

$$H(r) = B(1 - \alpha\frac{255}{256}) + (\alpha\frac{255}{256})H(\frac{1}{255}, ..., \frac{1}{255}) \tag{6}$$

Dove $B(\lambda) = H(\lambda, 1-\lambda)$ è la funzione di entropia binaria. Notiamo come $H(\frac{1}{255}, ..., \frac{1}{255}) = log_2(255)$ in quanto è l'entropia di una distribuzione uniforme su 255 elementi.

$$H(r) = B(1 - \alpha\frac{255}{256}) + (\alpha\frac{255}{256})log_2(255) \approx B(1 - \alpha\frac{255}{256}) + \alpha * 7.9631 \tag{7}$$

Ora posso quindi ricavare la capacità del canale grazie all'equazione 1 come:

$$C = 8 - B(1 - \alpha\frac{255}{256}) - \alpha * 7.9631 \tag{8}$$

Per ottenere la capacità del canale in bit/ora dovrò infine moltiplicare tale risultato per il numero di piccioni partititi ed arrivati in un ora, ossia 12.
Otterrò quindi che la capacità del canale definito dai piccioni viaggiatori che vengono abbattuti e sostituiti dal nemico con probabilità α sarà pari a $C_{ora} = 96 - 12 * B(1 - \alpha\frac{255}{256}) - \alpha * 95.52$ bit/ora.
Notiamo che con $\alpha = 0$ si ritorna alla situazione vista nel punto a dell'esercizio, con entropia per ora pari a 96 bit/ora.