1 Codici e linguaggi

Come studiare l'algoritmo di Huffman

Mauro Torelli – a. a. 2003-2004

Note al corso di *Algoritmi e strutture di dati*

1.1 Il problema

Abbiamo un file che occupa 2 megabyte (MB) e vogliamo trasferirlo su un dischetto da 1.44 MB: cosa facciamo? Usiamo un programma di *compressione*, che riduca l'occupazione di memoria del nostro file. Come *funziona* un programma di compressione dei dati, quali sono le *idee* su cui è basato, come vengono realizzate in maniera *efficiente*, quale *interfaccia* è più opportuno fornire all'utente? Ecco alcune possibili domande da porsi: un corso di *algoritmi* offre qualche risposta alle due centrali e tipicamente presenta l'***algoritmo di Huffman*** (per algoritmi diversi, prodotti commerciali e *molto* altro si può vedere per esempio la pagina http://www.datacompression.info/).

Lo scopo di questa breve nota è quello di svolgere alcune considerazioni su questo algoritmo, da cui trarre spunto anche per introdurre, in una seconda nota, gli *alberi* in maniera non convenzionale, ma utile per chiarire i diversi tipi di alberi e le loro relazioni. Tra gli altri obiettivi, la nota intende esemplificare alcuni punti che uno studente consapevole dovrebbe considerare durante lo studio, anche se non sono sempre riportati esplicitamente nel testo. In ogni caso, questa nota *non sostituisce* affatto il testo, ma lo accompagna come spiegazione e complemento.

Il nostro libro di testo, *Introduzione agli algoritmi*, Cormen et al., presenta i *codici di Huffman* al paragrafo 17.3, dando per scontato che uno studente abbia un'idea, magari vaga, di cosa sia un ***codice***. In effetti, si suppone che tutti conoscano il *codice ASCII* (si veda, per esempio, http://www.telacommunications.com/nutshell/ascii.htm: il "telacommunications" con la "a" è corretto!) o il più moderno *Unicode* (http://www.unicode.org/).

Tutti questi codici hanno *lunghezza fissa*: la codifica ASCII del carattere "a" = 97 = 0110 0001 è lunga quanto quella della "z" = 122 = 0111 1010 (anche se le lunghezze delle codifiche in decimale sono diverse). Ma, normalmente, in un testo italiano, ci sono molte più "a" che "z", e allora un'ovvia *idea* per risparmiare spazio o memoria è quella di usare codifiche *brevi* per i caratteri *frequenti*, e più *lunghe* per i caratteri *poco* frequenti.

Un ipotetico *studente frettoloso* potrebbe dire: “ho capito tutto. Prendo l’esempio del libro e lo codifico a modo mio. $a = 0$, $b = 00$, $c = 01$, $d = 1$, $e = 10$, $f = 11$: a e d sono i caratteri più frequenti, e sono a posto. Perché il libro spaccia quell’altro codice come ottimo? Il mio è meglio!”.

Non dovreste avere difficoltà a scoprire cosa non funziona nel codice dello studente. Potrebbe essere un po’ più difficile definire con precisione quali devono essere i requisiti di un (buon) codice, quelli che di fatto

autorizzano a chiamare "codice" un insieme... già, un insieme di che cosa? Un insieme di *parole* o di *stringhe* non necessariamente binarie (abbiamo già visto la codifica ASCII in decimale). Ora, un insieme di parole costituisce un *linguaggio* e la *teoria dei linguaggi* fornisce ottimi strumenti per trattare questi insiemi: cenni su tale argomento si trovano nel testo al capitolo 34 e al §36.1.

Vedremo, da un lato, che non c'è nulla di difficile nei rudimenti della teoria dei linguaggi, dall'altro, che la manipolazione di oggetti semplici come le parole, per lo più binarie, ci fornirà strumenti *chiari* e *concisi* per scoprire nuovi oggetti o dimostrare loro proprietà. Per esempio, in un'altra nota mostreremo inattese *corrispondenze* tra linguaggi e alberi.

1.2 Codici

Cominciamo col vedere quali requisiti debba avere un *codice*. Prendiamo l'esempio del testo (§17.3): abbiamo un insieme A di *caratteri* da codificare, nel nostro caso $A = \{a, b, c, d, e, f\}$, e poi abbiamo un insieme C di parole di codice, con cui vogliamo rappresentare gli elementi di A. Lo studente frettoloso propone $C = \{0, 00, 01, 1, 10, 11\}$ e la *corrispondenza* $a = 0$, $b = 00$, $c = 01$, $d = 1$, $e = 10$, $f = 11$.

Che cosa non funziona? Il fatto, per esempio, che $ad = c = 01$: codifichiamo allo stesso modo le due parole diverse ad e c, e quindi, nella *decodifica*, quando troviamo la parola 01 non sappiamo se corrisponde a ad oppure a c: una situazione ovviamente *ambigua* e inaccettabile.

Il codice non funziona – anzi, diciamo che l'insieme proposto *non è un codice* – se possiamo scrivere *equazioni* tra parole diverse, come $ad = c$, che introducono ambiguità. Al contrario, si può chiamare ***codice*** un qualunque insieme di parole per il quale non valga alcuna equazione. Naturalmente, sono sempre possibili le *identità*, come $ad = ad$, ma queste sono inevitabile e *banali* (in inglese *trivial*: non traducetemelo con *triviale*, per piacere…).

Il libro parla però di codici *prefissi*, lasciando intendere che questi siano sempre dei codici (ossia non ambigui). La dimostrazione è semplice e per qualche aspetto istruttiva, quindi la riportiamo, dopo aver chiarito la nozione di *prefisso*.

Immagino che tutti sappiano cos'è un prefisso (anche gli autori del libro lo immaginano, e perciò non lo spiegano). Però molti studenti, all'esame, non sono capaci di darne una definizione, men che meno una definizione *formale*, che pure è tra le definizioni formali più semplici a cui si possa pensare. Provate a dare una definizione, prima di leggere il prossimo capoverso.

La parola x è un ***prefisso*** della parola y se $y = x\,z$: informalmente, x è un prefisso di y se la parola y *comincia* con la parola x (anche nel testo a pagina 806).

Un ***codice prefisso***, come spiegato nel testo, è un codice in cui *nessuna parola è prefisso di un'altra*. Questa condizione assicura che l'insieme di parole è un codice: lo dimostriamo *per assurdo*. Supponiamo che l'insieme *non* sia un codice, e quindi esista un'equazione del tipo $x\,y\,\ldots\,z = u\,v\,\ldots\,w$, dove $x, y, \ldots, z, u,$

$v, \ldots, w$ sono parole del codice. Se fosse $x = u$ potremmo eliminare x e u ottenendo ancora un'equazione e continuare eventualmente a eliminare coppie di parole eguali: supponiamo quindi $x \neq u$. Ma allora, se x e u sono parole *diverse* con la *stessa lunghezza*, come può valere l'uguaglianza? L'unica possibilità è che x e u abbiano *lunghezze diverse ma comincino allo stesso modo,* ossia che una parola sia *prefisso* dell'altra. Se questo non può succedere non può esserci neppure l'uguaglianza, e quindi l'insieme di parole "senza prefissi" è un codice.

Attenzione! La condizione (di non avere prefissi) è ***sufficiente*** ma non ***necessaria***. Il modo più semplice di mostrare che qualcosa non è vero è di esibire, quando possibile, un ***controesempio***. Qui è facile: quello che abbiam detto per i prefissi lo possiamo dire, simmetricamente, per i suffissi, e un *codice suffisso* (ossia *senza* suffissi) può non essere prefisso (ossia può ammettere parole che *sono* prefissi di altre), come per esempio $C = \{0, 01, 11\}$: 0 è prefisso di 01, ma non ci sono suffissi. Anche $C = \{00, 001, 011, 01, 11\}$ è un codice, pur contenendo 00, prefisso di 001, e 11, suffisso di 011. Per la dimostrazione si può vedere il libro *Algebraic Combinatorics on Words*, (esempio 6.1.5) disponibile in rete all'URL (occhio ai trattini!) www-igm.univ-mlv.fr/~berstel/Lothaire/

Attenzione di nuovo!! Non tutto quello che esiste in rete è di qualità paragonabile a quella di questo libro, scritto da specialisti! Diffidate del materiale in rete se non conoscete gli autori (un laureando copiò pedissequamente delle note alquanto approssimative scritte da uno studente di liceo – che correttamente si dichiarava tale!) e comunque citate sempre le vostre fonti, sia per dare il dovuto riconoscimento, sia per scaricarvi di qualche responsabilità (per esempio, non ho controllato la dimostrazione dell'ultimo esempio sopra riportato…).

Se qualcuno non è troppo soddisfatto della formalizzazione fin qui introdotta… ha ragione. Ci sono molte lacune, ma abbiamo cercato di procedere con gradualità, un passo alla volta.

Vediamo almeno un paio di queste lacune. La parola 01 è un prefisso di 01? Ovvero, vale ancora $y = x\,z$ quando $y = x$? Possiamo ammettere una parola z "inesistente"? No, se non esiste, sì, se esiste ma è… *vuota*. Nell'equazione $ad = c$, cos'è esattamente ad? Certo non è un prodotto di numeri, dato che si tratta di parole: ma ha senso un prodotto tra parole? Bisogna formalizzare un po' di più e meglio: il piccolo sforzo sarà compensato da maggiore concisione (quindi sarà più facile capire e ricordare) e maggiore chiarezza (idem).

1.3 Linguaggi

Cominciamo dall'ultima "lacuna" vista precedentemente. Se dobbiamo parlare di *operazioni* tra parole, forse è opportuno ricorrere alle consuete *strutture algebriche*, che forniscono un contesto ben conosciuto e formalizzato. Prima ancora, converrà osservare che le parole non sono *atomi* indivisibili, ma sono "stringhe" o meglio liste di simboli.

Partiamo allora da un insieme S di *simboli*, non vuoto e in genere *finito* (lo chiameremo spesso ***alfabeto***), e da un'*operazione* $\bullet$, detta ***prodotto di concatenazione***, che appunto concatena due o più simboli di S a

formare quella che chiameremo una ***parola*** sull'alfabeto S. Per analogia con quanto si fa nei linguaggi comuni (e anche con l'operazione di prodotto ordinaria), l'operazione $\bullet$ di concatenazione spesso non è indicata da alcun simbolo: i simboli concatenati sono semplicemente scritti di seguito. Chiameremo ***lunghezza*** di una parola il numero di simboli che la compongono, e denoteremo con $|p|$ la lunghezza di una parola p.

Esempio 1. Sia $S = \{0, 1\}$. Il prodotto $0 \bullet 1 = 01$ è una parola, come pure $0 \bullet 0 = 00$ e $0 \bullet 1 \bullet 0 = 010$. Le parole 01 e 00 hanno lunghezza 2, mentre $|010| = 3$.

Siamo sicuri che $(0 \bullet 1) \bullet 0 = 0 \bullet (1 \bullet 0)$ e quindi che possiamo omettere le parentesi? Sì, perché non c'è altro modo di concatenare: l'operazione è *associativa*. È invece ovvio che $0 \bullet 1 = 01 \neq 10 = 1 \bullet 0$, quindi l'operazione di concatenazione *non è commutativa*.

Talvolta si indica la lunghezza di una parola p con $\lg(p)$. Menzioniamo questo fatto per notare l'*analogia con il logaritmo*: l'equazione $\lg(p \bullet q) = \lg(p) + \lg(q)$ vale per entrambi i casi. Inoltre, la lunghezza della rappresentazione binaria di un intero positivo n è circa $\lg n$, e precisamente $\lfloor \lg n \rfloor + 1 = \lceil \lg(n+1) \rceil$ (in questa frase lg denota il logaritmo in base 2). Ma la notazione $|p|$ è più concisa e la preferiamo.

Un ***linguaggio*** è semplicemente un insieme di parole. È naturale *estendere alle parole il prodotto di concatenazione*: il risultato è ancora una parola.

Esempio 2. Sia $S = \{0, 1\}$ come prima: $L = \{00, 01\}$ è un linguaggio su S, e $00 \bullet 01 = 0001$ è una nuova parola, diversa da $01 \bullet 00 = 0100$.

I simboli di S sono parole? Sono *indistinguibili*, se non concettualmente, dalle parole di lunghezza 1. A questo punto possiamo considerare anche S come un linguaggio, costituito da tutte le parole di lunghezza 1. Quali sono invece le parole di lunghezza 0? Una parola di lunghezza 0 per definizione non ha simboli di S, dunque non si può differenziare tramite S e sarà pertanto *unica*; si pone poi il problema di visualizzarla in qualche modo. Useremo il simbolo ε (talvolta si preferisce λ) per indicare questa ***parola vuota***: il simbolo ε *non* deve far parte dei simboli di S, altrimenti denoterebbe una parola di lunghezza 1!

A questo punto possiamo *formalizzare* la nozione di codice prefisso: C è un codice prefisso *sse*[1] $x, y \in C$ implica $y \neq xz$ con $z \neq \varepsilon$.

Una *convenzione* comoda e abbastanza diffusa, cui cercheremo di attenerci, prevede che, dovendo usare dei simboli per indicare intere parole di un linguaggio, si usino le lettere *in fondo* all'alfabeto (per esempio da p a z) per denotare *parole* di lunghezza arbitraria, quelle *all'inizio* per denotare invece *simboli* di S, ovvero parole di lunghezza 1.

Con un'ulteriore estensione della nozione di prodotto possiamo ora moltiplicare due (o più) linguaggi: *il* ***prodotto di due linguaggi*** *è l'insieme dei prodotti delle parole del primo per quelle del secondo*:

[1] La parola *sse* è un'abbreviazione molto usata per indicare "*se e solo se*", ossia una *definizione*, come qui, oppure una *condizione necessaria e sufficiente.*

$L \bullet L' := \{x \bullet y : x \in L, y \in L'\}$. Ricordiamo che questa notazione si legge: "il linguaggio prodotto $L \bullet L'$ è costituito *per definizione* (simbolo ":=") dai prodotti $x \bullet y$ di una parola x di L per una parola y di L'".

Esempio 3. Siano $L = \{0, 01\}$ e $L' = \{0, 10\}$: il linguaggio $L \bullet L'$ è l'insieme $\{00, 010, 0110\}$.

Il prodotto di linguaggi *non* è il *prodotto cartesiano* dei due insiemi, ossia l'insieme delle *coppie*: nell'esempio precedente, le due coppie (0, 10) e (01, 0) sono distinte, mentre la parola ottenuta dal prodotto di concatenazione, 010, è unica.

Che cos'è $S \bullet S$, che naturalmente denoteremo con S^2? È l'insieme delle parole ottenute concatenando un simbolo di S con... un simbolo di S, ossia l'insieme delle parole composte da 2 simboli di S, e dunque è l'insieme delle parole su S di *lunghezza* 2. Analogamente, S^n denoterà l'insieme delle parole di *lunghezza* n e, in particolare, $S^0 := \{\varepsilon\}$.

Ci serve ora una notazione per indicare *tutte* le parole su S, ossia $\bigcup_{n=0}^{\infty} S^n$. La notazione universalmente accettata è S^*, un simbolo proposto dal logico *Kleene*: S^* denota dunque l'insieme di *tutte le parole* sull'alfabeto S, *inclusa la parola vuota*. L'operazione che costruisce S^* da S è talvolta chiamata *chiusura di Kleene*. Se vogliamo *escludere* la parola vuota scriveremo invece S^+: dunque per definizione $S^+ := \bigcup_{n=1}^{\infty} S^n$. Possiamo anche scrivere $S^* = S^+ \bigcup S^0$. Abbiamo dato per scontato che l'*unione* di più linguaggi è l'unione degli insiemi di parole che li compongono.

Naturalmente S^* e S^+ sono linguaggi *infiniti*, contenenti infinite parole; invece, per ogni $n \geq 0$, S^n è *finito*, se l'alfabeto S è finito. Ai teorici piace chiamare S^* il ***monoide libero*** *generato da S*. Infatti, la parola vuota ε gioca il ruolo di *unità* per il prodotto di concatenazione: presa una qualunque parola p, si ha $p\varepsilon = \varepsilon p = p$. Dunque la struttura algebrica $(S^*; \bullet, \varepsilon)$ è un *monoide non commutativo* (salvo un caso particolarissimo in cui è *banalmente* commutativo: quale?). Infine, S^* è l'insieme di tutti i prodotti di elementi di S, ossia è *generato* da S, senza alcun *vincolo* (espresso da equazioni) che non sia banale: l'aggettivo "libero" esprime proprio questa condizione.

Ricordiamo che S è l'alfabeto, ossia l'insieme dei simboli. Se invece consideriamo le potenze di un linguaggio L, in generale potremo scrivere delle equazioni, a meno che L non sia un *codice*, come abbiamo visto nel paragrafo precedente. Per esempio, se $L = \{0, 01, 10\}$, L^2 comprende la parola 010, che può essere ottenuta o come $0 \bullet 10$ o come $01 \bullet 0$. Se poniamo $x = 0$, $y = 01$ e $z = 10$ abbiamo il vincolo non banale $x \bullet z = y \bullet x$. Nel monoide libero c'è solo il vincolo banale $a \bullet b \bullet ... \bullet c = a \bullet b \bullet ... \bullet c$.

1.4 Codici di Huffman

Veniamo finalmente ai codici che ci interessano, quelli ottenibili con l'algoritmo di Huffman, pubblicato nel lontano 1952. Il libro dice che tale algoritmo costruisce un codice prefisso *ottimo*. Ormai sappiamo cosa significa "codice prefisso", quindi non resta da chiarire che il termine "ottimo", anche se il significato di

questo termine è quello stesso che ha nel linguaggio comune (inteso con precisione): una soluzione è ottima se è *la migliore possibile*, ossia se non è possibile fare di meglio. Poiché l'obiettivo era quello di risparmiare memoria, un codice ottimo è un codice che usa la quantità di memoria *minima* possibile; si potrebbe anche dimostrare che non solo un codice di Huffman è ottimo *tra quelli prefissi*, ma anche che non vi sono codici che usino meno memoria, anche abbandonando il comodo requisito dell'essere prefissi.

L'ottimalità è dimostrata nel testo, mentre il fatto che si ottenga un codice prefisso è dato per scontato. Ora, uno studente attento dovrebbe, in primo luogo, accorgersi di questo fatto, in secondo luogo dovrebbe riuscire a dimostrare molto facilmente (ecco perché non è detto esplicitamente) che si ottiene comunque un codice prefisso, senza bisogno di verifica (a proposito, quante operazioni occorrono per verificare in maniera ovvia se un codice di n parole è prefisso? Ecco una tipica domanda da corso di algoritmi. Dovendo confrontare *coppie* di parole per stabilire se una è prefisso dell'altra, occorre un numero di confronti dell'ordine di n^2).

In realtà, l'algoritmo di Huffman non costruisce subito un codice, ma costruisce un *albero binario*, una struttura composta da *nodi* e *puntatori* che puntano ad altri nodi o hanno il valore NIL. Ogni nodo ha due puntatori: un puntatore sinistro e uno destro. Il testo (al §5.5.3) dà una definizione ricorsiva e più *astratta* di albero binario (non fa riferimento ad attributi concreti come i puntatori e il valore NIL), e noi daremo più avanti una definizione in termini di linguaggi, che utilizzerà in parte l'approccio seguito per il codice di Huffman. In particolare, gli alberi costruiti con l'algoritmo di Huffman risultano *pienamente binari*: ogni nodo o ha due figli (nodo *interno*) o non ne ha (è una *foglia* o nodo *esterno*; un nodo con un solo figlio potrebbe tranquillamente essere eliminato, ottenendo un codice migliore).

Per ottenere il codice, associamo il simbolo 0 a un puntatore sinistro, e il simbolo 1 a un puntatore destro: seguendo i puntatori dalla radice a un qualsiasi nodo dell'albero si ottiene una parola binaria (ossia sull'alfabeto {0, 1}) e le parole del codice sono quelle ottenute raggiungendo le *foglie* dell'albero.

Poiché le parole del codice sono associate *solo alle foglie* dell'albero, non capita mai che vi sia una parola di codice prefisso di un'altra, perché ciò significherebbe che la parola deriva da un *nodo interno* e non da una foglia! Quindi il codice così ottenuto è "automaticamente" prefisso.

È anche *ottimo*? La dimostrazione del testo è abbastanza lunga e la conclusione che dai due lemmi dimostrati discenda il risultato non è proprio evidente. Proviamo a dare *una dimostrazione* completamente *diversa*.

Data una lista L di interi positivi (con possibili ripetizioni) chiameremo ***catena di addizioni generata da*** L una lista o sequenza *non decrescente* di interi positivi comprendente L e tale che *ogni elemento non in L sia la somma di due elementi precedenti nella sequenza*. Per evidenziare gli elementi di L nella catena li sottolineeremo.

Per esempio, (5, 9, 12, 13, 16, 45) è una catena che coincide con la lista generatrice L; (5, 9, 12, 13, 14, 16, 45) è una catena generata dalla medesima lista L, con l'aggiunta dell'elemento 14 ottenuto da 5 + 9.

Gli *alberi pienamente binari* che hanno gli elementi di L nelle *foglie* e la somma dei nodi figli in ciascun *nodo interno* corrispondono a *particolari* catene tali che *ogni elemento della catena* (tranne l'ultimo) *viene usato come addendo in una e una sola somma*. In altre parole, possiamo pensare di *eliminare* due elementi di L sostituendoli con la loro somma, poi sostituire altri due elementi (inclusi i nuovi) con la loro somma, e così via. Dopo quanti passi resta *un solo elemento* (di valore pari, ovviamente, alla *somma degli elementi di L*)? Si tratta in sostanza di un *torneo a eliminazione* un po' insolito, poiché il numero di partite necessarie per vincere può variare da giocatore a giocatore: ogni partita elimina un giocatore e se i giocatori sono n, il vincitore si avrà dopo $n - 1$ partite. Sappiamo dunque *quanto sono lunghe queste catene* e anche quanti sono i nodi interni in un albero pienamente binario: *se vi sono n foglie*, *vi sono* $n - 1$ *nodi interni*!

Potremmo chiamare *catene arboree* siffatte catene, o, più spiritosamente… ***liane***. Per esempio, la catena C = (5, 9, 12, 13, 14, 16, 25, 39, 45, 61, 100) è una liana sulla solita lista L, la lista che, guarda caso, contiene i valori di frequenza (in migliaia) dell'esempio del testo per l'algoritmo di Huffman. Abbiamo cercato di indicare con colori in qualche modo analoghi di volta in volta gli addendi e il risultato. Notiamo che occorre "costruire" 25 come 12 + 13, non come 9 + 16, per rispettare il vincolo del monouso: 9 è usato in 14 = 5 + 9 e 16 in 61 = 16 + 45.

In un certo senso, dunque, la liana contiene "meno informazione" di un albero corrispondente, o meglio l'informazione della liana richiede di essere elaborata (determinando le coppie corrette), mentre nell'albero è esplicita ("vediamo" un nodo 25 padre di 12 e 13). Ma se l'obiettivo è quello di arrivare a una dimostrazione più convincente della correttezza dell'algoritmo di Huffman, allora è preferibile *ridurre l'informazione agli elementi essenziali* (i valori da minimizzare) invece di "distrarsi" con elementi accessori (la forma dell'albero).

Il problema risolto dall'algoritmo di Huffman può essere ora riformulato come quello di *determinare una liana ottima,* ossia *avente somma minima* (dove per "somma" intendiamo la somma di tutti gli elementi), data la lista L delle frequenze. È ora evidente che, dovendo utilizzare tutti gli elementi di L e *tutti* quelli via via generati (tranne l'ultimo, corrispondente alla somma delle frequenze), per ottenere la somma minima *basta* sommare di volta in volta i due elementi disponibili (ricordiamo che si possono usare *una volta sola*!) *più piccoli*: se non lo faccio, posso introdurre nella sequenza un elemento più grande del necessario (ma non è detto che la somma sia *maggiore*: quello che mi interessa è il fatto che non può essere *minore*). La liana di Huffman è semplicemente quella in cui la somma dei primi k elementi è minima, per ogni k.

Una liana *ottima* per la lista L del nostro esempio è (5, 9, 12, 13, 14, 16, 25, 30, 45, 55, 100), che è quella ottenuta con l'algoritmo di Huffman. Un'altra è, per esempio, (5, 9, 12, 13, 14, 16, 26, 29, 45, 55, 100): pur essendoci svincolati dalla *forma* dell'albero, non esiste però in generale un'*unica liana ottima*, anche se vi sono *molti* più alberi ottimi e codici ottimi (le etichettature con 0 e 1 sono arbitrarie).