Elaborazione di Dati Multimediali

Cristian Mercadante
UNIMORE | ULTIMO AGGIORNAMENTO: 12/06/2019

Lezione 5

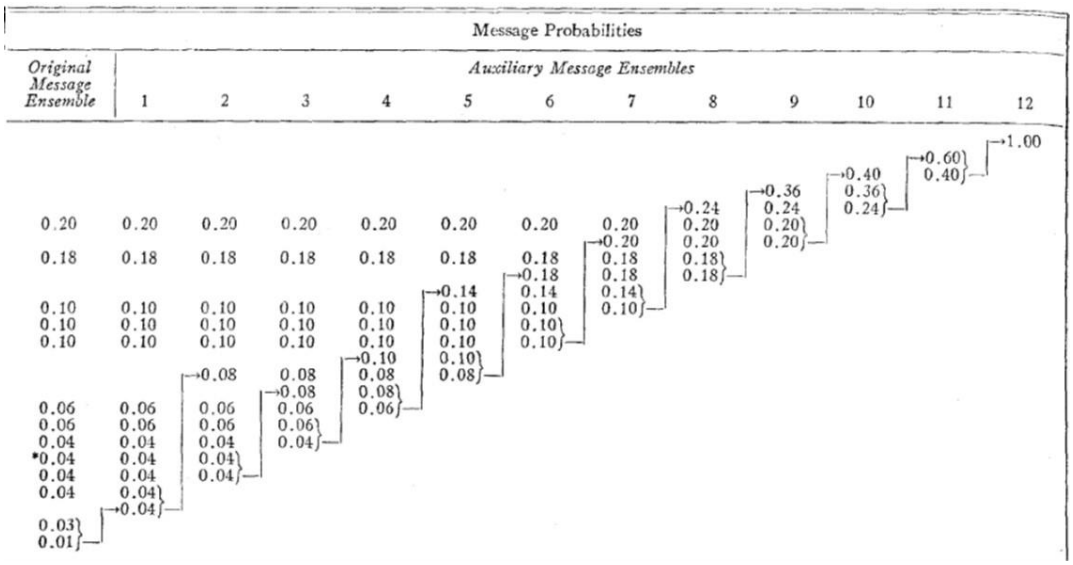
Mercoledì 6 marzo 2019

Se definiamo una codifica come l'insieme di coppie che a ogni simbolo associa una sequenza di bit (parola di bit), questo produce come effetto il poter calcolare una certa lunghezza media di una codifica. Se assumiamo che questa associazione sia fatta in maniera intera, ovvero a ogni simbolo si associa una parola con un numero di bit intero (un certo numero di bit per ogni simbolo), a questo punto l'analisi che abbiamo fatto sta in piedi. Esistono modalità di codifica che sono più sofisticate (es. codifica aritmetica) che spingono ulteriormente, permettono di comprimere un po' di più.

Torniamo al caso facile: associazioni simbolo-codice. Il codice è un certo numero intero di 0 e 1. Supponiamo che la sorgente sia senza memoria (indipendenza sei simboli): siamo esattamente nel caso della teoria di Shannon.

Codifica di Huffman

Nel 52 David Huffman ha pubblicato un'idea che è diventata un riferimento per la codifica ottima. Ha mostrato che utilizzando l'algoritmo che lui presentava si ottiene una codifica a lunghezza variabile ottima. Quindi se abbiamo una distribuzione di probabilità e usiamo la codifica di Huffman, non esiste altra codifica che abbia lunghezza media inferiore. Nella codifica JPEG, le immagini al loro interno usano la codifica di Huffman. I codici generati dall'algoritmo di Huffman sono codici prefix-free.



La tecnica è semplice. Vediamo un esempio: abbiamo un messaggio con un certo insieme di simboli, che non è importante, perché quello che conta sono le probabilità di ogni simbolo. Abbiamo 13 simboli la cui probabilità va da 1% a 20%. Prendiamo un insieme di simboli di cui sappiamo la distribuzione di probabilità (dobbiamo saperla, non è dinamica: contiamo il numero di occorrenze di un simbolo per il numero totale di occorrenze). Ordiniamo i simboli a seconda delle probabilità e si comincia considerando i due elementi meno probabili. Quanti bit mi servono per distinguere fra due cose? 1 bit. L'algoritmo di Huffman costruisce un albero di decisione che, sulla base di un bit, mi permette di scegliere una strada o l'altra. Usiamo quindi singoli bit per distinguere dei gruppi, ma quali gruppi distinguiamo? Dove vogliamo mettere più bit? Sui simboli meno probabili. Per cui, una volta ordinati i simboli a seconda della probabilità, consideriamo l'insieme dei simboli costituito dal simbolo che ha probabilità 1% e il simbolo che ha probabilità 3%. Qual è la probabilità di incontrare uno o l'altro simbolo? 4%. Quindi reinseriamo questo gruppo come fosse un unico simbolo all'interno della sequenza (e ci vorrà un bit per distinguere fra uno e l'altro). Il reinserimento deve far sì che la sequenza sia comunque ordinata per probabilità. Da 13 elementi siamo passati a 12. Per andare alla fine dovremo fare 13 passaggi, ovvero tanti quanti sono i simboli. Quindi:

1. Ordiniamo in ordine decrescente per probabilità dei simboli.
2. Prendiamo le 2 probabilità più basse. 2 perché con 1 bit posso distinguere due simboli.
3. Sommiamo le probabilità costruendo un nuovo gruppo.
4. Reinseriamo nella sequenza.
5. Ricominciamo finché non otteniamo probabilità 100%.

Non abbiamo fatto altro che costruire un albero binario, chiamato **albero di Huffman**, che può esser percorso usando 0 o 1 per andare a destra o a sinistra. Quindi basta assegnare 0 e 1 a destra o sinistra. Ipotizziamo che 0 sia per il ramo meno probabile, mentre 1 sia per il ramo più probabile. Vado avanti percorrendo l'albero aggiungendo bit 0/1 a destra. Non è importante scegliere 0/1 per destra o sinistra: l'importante è che chi codifica e chi decodifica usi la stessa regola. Se ho 13 simboli, ho 2^{13} codifiche di Huffman possibili. Questa codifica è ottima: Huffman l'ha dimostrato. Notiamo che non succede mai che un codice che ha probabilità più alta di un altro abbia un codice più lungo di uno con probabilità più bassa. È evidente che per ottenere un risultato uguale all'entropia, c'è un problema: stiamo usando la stessa quantità di bit per simboli che hanno probabilità diverse.

Notiamo che nel rappresentare simboli più probabili, usiamo meno bit di quelli necessari secondo il calco dell'entropia. Pertanto, in altri casi useremo più bit di quelli necessari secondo l'entropia, quindi complessivamente peggiorerà: non riusciremo ad arrivare all'entropia. Quindi con l'algoritmo di Huffman ci siamo avvicinati paurosamente all'entropia.

Simbolo	p(x)	$-p(x)*\log_2(p(x))$	Codice	L(c(x))	p(x)*L(c(x))
a	0,20	0,46	01	2	0,40
b	0,18	0,45	111	3	0,54
c	0,10	0,33	100	3	0,30
d	0,10	0,33	001	3	0,30
e	0,10	0,33	000	3	0,30
f	0,06	0,24	1010	4	0,24
g	0,06	0,24	11011	5	0,30
h	0,04	0,19	11010	5	0,20
i	0,04	0,19	10111	5	0,20
j	0,04	0,19	10110	5	0,20
k	0,04	0,19	11001	5	0,20
l	0,03	0,15	110001	6	0,18
m	0,01	0,07	110000	6	0,06
H(S) = 3,35			L_{avg} = 3,42		

Abbiamo un insieme di simboli, abbiamo ottenuto i codici da associare. Supponiamo che questa inizi con 'cfamc'. Dovrebbe essere: '100101001110000100110001'. Questo è lo stream di bit che otteniamo, scritto su file in binario. Otterremo una sequenza grande 3 byte. Sarebbe '94 E1 31' in esadecimale. Nota bene: il nostro caso è fortunato perché abbiamo esattamente 3 byte, quindi non ci preoccupiamo di quello che succede se mancano bit. Usando delle opportune operazioni bit a bit, possiamo tirare fuori dai singoli byte i corrispondenti bit. Leggo il primo bit: 1. È uno dei nostri simboli? No. Leggiamo il secondo bit: 0. Potrebbe essere uno dei simboli? Sì perché ne abbiamo uno di lunghezza 2, ma non è quello. Quindi vado avanti e trovo 100 = c. Non esiste nessun codice di lunghezza inferiore che è prefisso di un codice più lungo, perché la codifica è prefix-free. Quindi sono certo che 100 = c. E così via. Questo meccanismo di decodifica funziona con qualsiasi codifica prefix-free.

Se noi abbiamo l'albero di Huffman in memoria, e soprattutto se esiste una regola per andare a destra o a sinistra a seconda degli 0/1, possiamo interpretare la sequenza di bit come un percorso su un albero binario. La decodifica ha complessità lineare nel numero di livelli dell'albero di Huffman, ovvero nel numero di bit massimo. Sia un modo che l'altro sono lentissimi: si può fare di meglio. Per fare una decodifica più efficiente si possono utilizzare delle tabelle e alberi di tabelle.

Come fa il JPEG a trovare la distribuzione? Prendiamo tanti file che vogliamo rappresentare (es. immagini, se vogliamo comprimere immagini). Si fa una stima statica a priori dei simboli e si comprime, assumendo che questa stima valga anche nel caso particolare. Altrimenti mi baso solo sul caso particolare e trasmetto anche la tabella coi codici. Oppure esiste una variante dell'algoritmo di Huffman chiamata *Dynamic Huffman Coding*: ogni modifica altera il codice di Huffman a seconda dei simboli che leggo.

Simbolo	N° bit	Cod. canonico 1	Cod. canonico 2
a	2	00	11
b	3	010	101
c	3	011	100
d	3	100	011
e	3	101	010
f	4	1100	0011
g	5	11010	00101
h	5	11011	00100
i	5	11100	00011
j	5	11101	00010
k	5	11110	00001
l	6	111110	000001
m	6	111111	000000

La codifica di Huffman non è unica. Cosa influenza l'entropia? La lunghezza dei codici: quindi se cambio i codici, le lunghezze sono le stesse. A parità di lunghezza quindi qualunque codice di Huffman è equivalente. Fra tutti i possibili codici di Huffman ci sono **codici canonici**. Si ottengono con un procedimento facile, meccanico, ripetibile, a partire solo dalle lunghezze dei simboli. Osservando l'esempio: parto con 0 e aggiungo 1. Se il codice non è abbastanza lungo, aggiungo zeri (shift a sinistra). Nell'avanzare seguo il binario. Mantengo la proprietà del prefix-free. Codici così costruiti finiscono tutti con '111…111'. Posso anche invertire il procedimento, ovvero parto da 1 e decremento di 1. L'unica informazione necessaria per costruire i codici canonici sono solo le lunghezze.

Lezione 6

Lunedì 11 marzo 2019 – Laboratorio

Rifacciamo un paio di considerazioni sull'espressione v3 = v2 * 7. È un expression statement che dobbiamo valutare, con due operatori su cui abbiamo fatto overload. Vediamo col debugger cosa succede: prima il costruttore di copia, ma poi viene fatta un'altra copia, un assegnamento e una distruzione. Basterebbe invece una copia e un distruttore. Quindi la bellezza del codice C++ che abbiamo scritto si porta dietro della lentezza, che alla fine si farà sentire. Quindi una volta non si utilizzava questa cosa. Dal C++ 11 esiste una soluzione a questo.

Quando andiamo a creare una copia del nostro vettore dobbiamo distinguere due casi. Con vector v3 = v facciamo una definizione, quindi quell'uguale è un'inizializzazione. Quindi qui viene chiamato il costruttore di copia. I dati di v devono rimanere validi. Quando invece ho un'expression statement come v3 = v2 * 7, il valore di v2*7 non deve per forza rimanere valido, infatti viene

invocato il distruttore. Quindi abbiamo due casi: l'oggetto da cui copiamo resta valido, oppure l'oggetto da cui copiamo viene distrutto, quindi non è necessario mantenerne l'identità. Fino al C++ 11 non c'era differenza. Ma il compilatore sa che l'oggetto è temporaneo o no. Hanno introdotto perciò due concetti diversi. L'rvalue è il valore che può stare a destra dell'uguale. In vector v3 = v, v è anche un lvalue, perché può stare anche a sinistra dell'uguale. Quindi si è introdotto il concetto di **reference ad rvalue**. Proviamo a fare vector v4 = v2 *7. Dobbiamo cambiare l'operatore *. Intanto possiamo fare un costruttore che rovina i dati, ovvero che faccia sì che i dati non vengano copiati, ma diventino di un altro vettore: freghiamo i dati da un altro vettore, tanto a lui non servono più, perché verrà distrutto. Il problema è che se libero i dati di un vettore, li perde anche quell'altro. Quindi basta cambiare il puntatore ai dati del vettore a cui abbiamo rubato con *nullptr*. Però abbiamo un problema: abbiamo fatto un costruttore di copia che può essere chiamato anche quando non va bene. Vogliamo che questo costruttore possa essere utilizzato solo quando c'è un **rvalue reference**, ovvero un oggetto temporaneo. Quindi come parametro anziché vector& rhs, usiamo vector**&&** rhs. Questo costruttore si chiama **move constructor**. Inoltre, quando assegniamo un vettore a un altro, noi copiamo i suoi dati. Ma se assegniamo un vettore temporaneo a un altro vettore, possiamo rubare i suoi dati, quindi facciamo un operatore di **move assignment**. Ritornare un vector, vuol dire ritornare una copia, quindi deve essere chiamato il costruttore di copia. C'è qualche possibilità che questo venga utilizzato dopo? No perché lo scope della variabile è la funzione. Se invece per qualche ragione ritornassi una reference a un vettore che è stata passata come parametro, non posso dire quale sia il suo scope. Quindi in questi casi verrà sempre chiamato il costruttore di copia.

In C++ le librerie non hanno estensione, ma si chiamano cstdlib, ecc. con la C davanti. Se ci sono classi che hanno i nomi in conflitto, si definiscono **namespace**. Quindi si scrive namespace nome_ns. e si mette tutto quello che vogliamo all'interno di graffe. Se vogliamo utilizzare cose dentro a quel namespace, scriviamo nome_ns::.

Nella libreria utility esiste al funzione **swap**, che scambia due variabili. Facciamo lo swap fra vettori. Facciamo anche un metodo che, se io sono un vettore, mi scambia con un altro.

Le funzioni possono essere fatte **inline**, ovvero anziché fare le call (che fanno jump e rompono la pipeline dei processori), il codice assembly viene copiato, però il codice diventa lunghissimo. Se mettiamo una in una classe, questa viene fatta inline. Se la mettiamo fuori, diventa non inline (a meno che non lo scriviamo).

Per ora abbiamo usato solo variabili pubbliche. Come si modifica la visibilità degli elementi dall'esterno? Scriviamo private: e public:.

Nelle classi di default tutto è privato, mentre nelle struct di default tutto è pubblico. Possiamo nelle classi definire funzioni **friend** che possono accedere ai dati.

Idioma del C++: tecnica del **copy-and-swap** (https://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom).

L'**RVO** è una tecnica dei compilatori C++.

Quando facciamo una funzione, di default uso *const reference*. Poi guardo: la prima cosa che faccio è una copia? Allora tolgo la reference.

Lezione 7

Martedì 12 marzo 2019 – Laboratorio

Cerchiamo di risolvere un problema che nella scorsa lezione segnava un errore con funzioni definite come friend. Quando definiamo funzioni friend nella classe, sono più protette: possiamo accedervi, ma non vengono trovate nella ricerca dei nomi. In C++ c'è l'**Argument Dependent Lookup (ADL)**: si cerca una funzione non solo nel namespace globale, ma anche nel namespace dei suoi argomenti.

Torniamo al linguaggio C con un progetto chiamato "disegno". Gestiamo un canvas per disegnarci sopra. È una matrice salvata con le righe una dietro all'altra. C'è costruttore, distruttore, operatore new (che alloca e costruisce), operatore delete (distrugge e dealloca), set (scrive carattere), line, rectangle, circle, out (per stampare tutto). Vorremmo passare alla programmazione ad oggetti, creando una gerarchia di oggetti da usare assieme. Costruiamo un oggetto base "shape" ed ereditiamo le proprietà in altre forme. In più vogliamo implementare il polimorfismo: nel chiamare i metodi abbiamo static binding. Noi vorremmo un binding a runtime. Inseriamo nella struct un puntatore a funzione da chiamare. Questo occupa tanta memoria, perché per ogni funzione ho 4 byte. Vorremmo invece una lista di funzioni: **virtual table**. Sono le funzioni collegate a un oggetto. Il puntatore punta a un po' di funzioni che il compilatore saprà in base al problema.

Note sugli esercizi svolti

Abbiamo lavorato su un nuovo progetto in linguaggio C. Il professore ci ha fornito un punto di partenza costituito da due file chiamati "canvas.h" e "canvas.c". Il programma era in grado di utilizzare la linea di comando per stampare delle semplici figure geometriche usando gli ASCII. Abbiamo scritto un main e modellato il tutto con un occhio alla programmazione ad oggetti. Quindi abbiamo creato una struct chiamata "shape", con le relative funzioni di creazione, distruzione e disegno. Il prossimo obiettivo è stato l'implementazione dell'ereditarietà: abbiamo creato altre due strutture chiamate "square" e "circle", che volevamo ereditassero gli attributi da "shape" e che ne avessero anche altri. Queste classi hanno le loro funzioni di creazione, distruzione e disegno, specializzate sui propri attributi. Lo step successivo è l'implementazione del polimorfismo: vorremmo che la stessa funzione chiamata su oggetti di classe differente abbia un comportamento diverso a seconda della classe di questi oggetti. Abbiamo quindi creato una struttura "shape_manager" che potesse tenere e gestire al suo interno diversi tipi di "shape", che fossero "square" o "circle". Questa classe ha metodi per la costruzione, distruzione, aggiunta di "shape" e disegno. Il problema è che i metodi di distruzione e disegno si comportano diversamente a seconda della classe. Abbiamo risolto con due metodi. Il primo consiste nell'aggiungere ai campi della struct tanti puntatori a funzione quante sono le funzioni custom. In questo modo si risolve sì il problema, ma per ogni metodo e per ogni oggetto di una classe aumento la memoria occupata di almeno 4 byte. Il secondo metodo evita questo problema introducendo solo un puntatore all'interno dei campi della struct. Questo campo costituisce una virtual table, ovvero una lista di puntatori a funzione definita per ogni classe. In questo modo mi basta accedere a un elemento della lista per ottenere una funzione.

Lezione 8

Mercoledì 13 marzo 2019 – Laboratorio

Il C++ utilizza il meccanismo delle virtual table per implementare il polimorfismo. "Vtbl" è un puntatore a puntatori a funzioni che non hanno parametri e ritornano void. Nel costruttore c'è un'operazione in più che richiede che per ogni tipo di oggetto un qualche cosa che venga associata alla virtual table. Una shape_vtbl è un array di puntatori a funzioni che ritornano void. Questa è dichiarata nel costruttore di shape e definita in un'altra zona. Per fortuna abbiamo dei linguaggi che implementano tutto ciò, però se vogliamo ottimizzare una funzione, bisogna conoscere questo. Passiamo ora al C++. Ogni metodo è definito nel namespace dell'oggetto. Esistono **ereditarietà** pubblica, protected e private: meglio public. Quando eredito pubblico tutto quello che era privato, rimane privato. Per tutte le shape devo chiamare la delete di quella shape. Bisogna introdurre anche un meccanismo per dire che una classe supporta il **polimorfismo**: l'hanno fatto per evitare di avere una virtual table per ogni struct. Si usa la keyword "virtual": vuol dire che quel metodo viene aggiunto alla tabella dei **metodi virtuali**. Per le funzioni che fanno **override** si usa la keyword "override" per essere sicuri (è facoltativo). **Ricapitolando**: definisco i metodi. Se voglio poter usare una gerarchia di classi metto il virtual. Attenzione: il distruttore deve essere sempre virtuale. Indico override su tutte le funzioni che sovrascrivo e per ereditare metto ":" di fianco al nome della struct.

Canvas perché è allocato dinamicamente? Non ci piace, quindi può essere sullo stack: si dice **variabile automatica** se è allocata sullo stack. Siccome è automatica, la variabile viene distrutta da sola. Stessa cosa per shape_manager. Se un costruttore non vuole parametri, quando scrivo struct_manager sm(), le parentesi non le metto perché altrimenti il compilatore pensa che sia una funzione che ritorna shape_manager: crede che sia una dichiarazione. Quindi non metto le tonde.

La nostra shape vorremmo fosse un **oggetto astratto**: vogliamo che fornisca un'**interfaccia** comune a tutti quelli che ereditano. Le interface non esistono: in C++ possiamo farle trasformando un metodo da "virtual" a "**pure virtual**", cioè non esistente. Come si fa? Metto un puntatore a null nella virtual table. Si fa mettendo "= 0". Non si può istanziare un oggetto di una classe che ha almeno un metodo di tipo pure virtual.

Torniamo a ordina_int. Vogliamo portare dentro alla classe la funzione "compare_int". Però bisogna trovare un modo per dire che la funzione è di classe. Si usa la keyword "**static**", per dire che la funzione fa solo parte della classe. Volgiamo trasformare un vettore in un vettore generico. Si usa un **template**: diciamo che vector è un template per generare altre classi. Si usano "<>". Scrivendo template<typename T>, diciamo che quando costruiremo un vettore useremo il tipo T. I dati sono di tipo T. **Attenzione**: la memcpy va bene se i tipi sono semplici. Quando facciamo tipi generici, è meglio fare l'operazione a mano. È possibile importare dai namespace indicando "**using**". In generale, è meglio nei template usare meno operatori diversi possibile (ad esempio solo il > e non il <). Se abbiamo una variabile che viene inizializzata, è possibile specificare che una variabile è "**auto**": diventerà del tipo che gli stiamo assegnando.

La libreria standard di C++ ha dei concetti che dobbiamo conoscere. Come si scorre un vettore? Ciclo for con accesso diretto. Se volessimo usare una lista doppiamente concatenata (per fare inserimenti nel mezzo ad esempio), nella libreria standard hanno inserito i **container**, che sono template che servono a gestire i dati in memoria in maniera opportuna. Il **vector** è il vettore