Impariamo a leggere

(i file di testo in C)

Roberto Cordone

11 maggio 2009

Prefazione

Un algoritmo trasforma le istanze di un problema nelle corrispondenti soluzioni. Un'istanza è codificata in una stringa di simboli, per cui ogni algoritmo deve per prima cosa leggere tale stringa e caricarla in memoria nelle apposite strutture di dati, a cui accederà durante l'esecuzione.

Ne consegue che, per poter realizzare un algoritmo attraverso la stesura di un programma, occorre saper leggere i dati da file, nel caso più semplice da file di testo. Purtroppo, molti studenti non sono in grado di farlo e quindi di realizzare algoritmi. Queste note sono dedicate a loro.

Roberto Cordone

File di testo formattati

I file di testo sono il modo più semplice di conservare i dati di un problema. Dato un file di testo, il senso di ogni suo elemento può essere identificato da:

1. la *posizione*;
2. la presenza di un *token* (cioè un simbolo) introduttivo.

Il primo approccio è più semplice, il secondo è più flessibile e rende il file più leggibile da un essere umano.

Supponiamo di voler conservare in un file un grafo completo di 4 nodi $G = (N, A)$ e una funzione di costo associata ai suoi archi, $c : A \to \mathbb{N}$. Una possibile codifica basata sulla posizione è la seguente:

```
4 0 8 1 4 3 0 2 5 2 5 0 9 1 3 4 0
```

dove il primo numero rappresenta il numero dei nodi e i successivi rappresentano i costi degli archi, in ordine lessicografico. Per rendere il file più leggibile da un essere umano, si possono aggiungere degli a capo:

```
4
0 8 1 4
3 0 2 5
2 5 0 9
1 3 4 0
```

Ovviamente, se cambia l'ordine dei numeri, cambia completamente il grafo rappresentato.

Una possibile codifica basata su *token* (per inciso, si tratta di una codifica in formato MathProg) è la seguente:

```
param n := 4 ;

param c :=
[1,1] 0 [2,1] 8 [3,1] 1 [4,1] 4
[1,2] 3 [2,2] 0 [3,2] 2 [4,2] 5
[1,3] 2 [2,3] 5 [3,3] 0 [4,3] 9
[1,4] 1 [2,4] 3 [3,4] 4 [4,4] 0
;
```

dove `n` identifica il numero dei nodi, `c` la funzione di costo e `[i,j]` identifica il costo dell'arco (i, j). L'ordine dei dati è arbitrario, purché ogni *token* preceda il dato corrispondente e il *token* `c` preceda quelli degli archi. Ad esempio, sarebbe corretto, benché molto meno leggibile per un essere umano, scrivere:

```
param c := [1,3] 2 [3,3] 0 [2,3] 5 [4,2] 5 [1,1] 0 [3,1] 1
[2,1] 8 [4,1] 4 [1,2] 3 [2,2] 0 [4,3] 9 [3,2] 2 [1,4] 1
[3,4] 4 [2,4] 3 [4,4] 0 ;

param n := 4 ;
```

Lettura in C di un file basato su *token*

Leggere un file di testo codificato con *token* richiede:

1. di saper distinguere i singoli dati e i singoli *token*;
2. di saper identificare i singoli *token*;
3. di saper caricare i dati in opportune strutture in memoria.

Il linguaggio C offre parecchie procedure per leggere i file di testo e per operare su stringhe di caratteri. Questa ricchezza induce spesso grande confusione negli studenti, che si ingegnano faticosamente e inutilmente a ricostruire procedure che già esistono.

Identificazione del singolo dato o *token*

La quasi totalità delle operazioni necessarie a leggere un file di testo può essere svolta con la funzione *fscanf*[1]. L'istruzione

```
fscanf(f,"%s",s);
```

riconosce nel file `f`, partendo dalla posizione corrente della testina di lettura, una "parola" (sequenza di caratteri racchiusa fra spazi, a capo o tabulazioni) e la copia nella stringa di caratteri `s`.

La stringa `s` deve disporre di spazio sufficiente a contenere l'intera parola. Quindi, deve essere definita staticamente con una lunghezza tale da poter ragionevolmente contenere qualsiasi parola compaia nel file di testo (un'alternativa, in genere inutilmente più complessa, è allocarla dinamicamente).

```
#define WORD_LENGTH 255
...
char s[WORD_LENGTH];
```

[1] La funzione *fscanf* è stata recentemente dichiarata "deprecated", a causa di alcuni bachi nella sua implementazione. Tuttavia, rimane il modo più semplice e comodo di leggere dati da file di testo.

Nei file formattati, una parola è quasi sempre un dato, un *token*, una parola chiave o un segno di interpunzione. Quindi, è un'ottima idea scorrere il file con tale istruzione. Inizialmente escludiamo per semplicità la possibilità che un segno di interpunzione compaia attaccato a un dato o a un *token*, rendendone l'identificazione più difficile (ad esempio `param n:= 4;`, dove `param` è una parola, `n:=` una seconda e `4;` una terza).

È buona norma verificare ogni volta il valore restituito dalla funzione `fscanf`. Se è andata a buon fine, essa restituisce il numero di elementi riconosciuti e assegnati ad altrettante variabili. Ad esempio, `fscanf(f,¨%s¨,s);` deve restituire sempre il valore 1. In caso contrario, i dati non sono stati riconosciuti, e quindi caricati correttamente. In particolare, il valore `EOF` (costante predefinita di sistema) indica che il file è terminato. Come vedremo, entrambe le condizioni (valore diverso dal numero di argomenti o valore pari a `EOF`) possono essere usate come condizioni di uscita da cicli di lettura.

Le parole chiave e i segni di interpunzione andranno ignorati, ma ovviamente vanno letti, perché il file è una struttura sequenziale. Per farlo, o si leggono e si assegnano a stringhe che poi resteranno inutilizzate, oppure si indica direttamente alla funzione `fscanf` di ignorarle. Nell'istruzione

```
fscanf(f,"%*s %s",s);
```

l'asterisco indica che la prima parola riconosciuta non va assegnata ad alcun argomento, mentre la seconda va regolarmente assegnata alla stringa `s`.

Qualora il dato non sia una stringa, ma un numero intero, si può banalmente convertirlo con l'istruzione

```
sscanf(s,"%d",&i);
```

che traduce la stringa `s` nel numero intero `i`. Se il numero non è intero, ma `long int`, `float` o `double`, la stringa di formato `%d` va sostituita, rispettivamente, con `%ld`, `%f` e `%lf`. In alternativa, si può direttamente leggere il numero da file con l'istruzione `fscanf`, sostituendo la stringa di formato `%s` con una di quelle su elencate.

Scomposizione di una parola in più dati

Alcune parole racchiudono al loro interno più di un dato. Ad esempio, la parola `[1,2]`, che è il *token* introduttivo per il costo dell'arco $(1, 2)$, va scomposta nei segni di interpunzione (parentesi quadre e virgola) e nei dati (i numeri 1 e 2). Si potrebbe fare anche questo con l'istruzione

```
fscanf(f,"[%d,%d]",&i,&j);
```

ma è molto probabile che non funzioni. Il motivo è che la testina di lettura è posizionata inizialmente non sul carattere `[`, ma sullo spazio bianco, la tabulazione o l'a capo che seguono la parola precedente. Quindi, la funzione `fscanf` correttamente non riconosce la stringa di formato e si arresta, restituendo 0. In tale situazione, conviene spezzare in due il problema: prima saltare gli spazi bianchi e caricare l'intera parola `[1,2]` in una stringa `s`, poi interpretare la stringa

```
fscanf(f,"%s",s);
sscanf(s,"[%d,%d]",&i,&j);
```

Ancora più complicato è il caso in cui la stringa non rispetta rigorosamente il formato, ad esempio perché vi sono spazi bianchi distribuiti irregolarmente fra un termine e l'altro. In questo caso, non si può fare affidamento né sulla loro presenza né sulla loro assenza. In tali casi, conviene procedere al riconoscimento sulla base dei segni di interpunzione. La stringa di formato

```
%[^x]
```

rappresenta qualsiasi sequenza di caratteri non appartenenti all'insieme fra parentesi quadre, cioè qualsiasi carattere diverso da x. Quindi, per raggiungere la successiva parentesi quadra `[` basta scrivere

```
fscanf(f,"%[^[]");
```

Per leggere un'intera riga, compreso l'a capo finale, basta scrivere

```
fscanf(f,"%[^\n]\n",s);
```

Si noti che questa istruzione va a capo, ma non assegna l'a capo alla stringa `s`. Invece la funzione `fgets`, che fa la stessa cosa (ed è consigliabile) assegna a `s` anche l'a capo finale.

Identificazione di *token*

Per tener conto del fatto che l'ordine dei dati non è prefissato, occorre una procedura che cerca il *token* all'interno del file, scorrendo le parole una alla volta e riconoscendo se la parola corrente coincide o no col *token* cercato. Supponiamo che il numero dei nodi sia introdotto dal *token* `n`. Ad ogni passo l'istruzione

```
strcmp(s,"n");
```

restituisce valore negativo o positivo se la stringa `s` è lessicograficamente inferiore o superiore a `n`, valore nullo se coincide con `n`.

La seguente funzione determina se il file `fp` (già aperto) contiene o no la stringa `token`.

```
boolean TrovaStringa (FILE* fp, char* token)
{
  int n;
  char s[WORD_LENGTH];

  rewind(fp);
  do n = fscanf(fp,"%s",s);
  while ((n != EOF) && (strcmp(s,token) != 0));

  return ((n == EOF) ? FALSE : TRUE);
}
```

Il comando `rewind` riporta la testina all'inizio del file `fp` (per controllare che il *token* non sia prima della posizione corrente). Il ciclo legge una parola alla volta nella stringa ausiliaria `s` con la funzione `fscanf`, finché non arriva al termine del file (`EOF`) oppure finché la parola corrente `s` non coincide col *token*. Nel primo caso, la funzione restituisce `FALSE`, nel secondo `TRUE`. Si noti che `s` viene allocata staticamente con una lunghezza `WORD_LENGTH`, che si suppone sufficiente a contenere qualsiasi parola del file dato. Ovviamente, il file `fp` deve essere già aperto, le costanti `TRUE`, `FALSE` e `LENGTH` e il tipo `boolean` devono essere stati definiti dall'utente e le librerie `stdio.h` e `string.h` devono essere state incluse per poter accedere alle funzioni utilizzate.

Lettura e caricamento dei dati in memoria

Supponiamo di voler leggere il seguente file, che descrive un grafo orientato non completo $G = (N, A)$, con una funzione di costo c definita sugli archi e una funzione di peso w definita sui nodi.

```
param n := 5 ;

set A := (1,2) (1,4) (2,5) (3,1) (3,5)
         (4,1) (4,2) (5,1) (5,3) (5,4) ;

param w :=
1 10
2 12
```

```
3  9
4 21
5 15
;

param c :=
[1,2] 1 [1,4] 3 [2,5] 2 [3,1] 8 [3,5]  5
[4,1] 2 [4,2] 4 [5,1] 1 [5,3] 3 [5,4] 10 ;

end;
```

Assumeremo che si disponga di una struttura dati *grafo* definita in modo ragionevole, cioè dotata di procedure per:

- la creazione di un grafo vuoto (`CreaGrafo`);
- l'inserimento di nuovi nodi (`InsNodo`);
- l'inserimento di nuovi archi (dati i nodi estremi) (`InsArco`);
- la ricerca di nodi (dato l'indice) (`TrovaNodo`);
- la ricerca di archi (dati gli indici dei nodi estremi) (`TrovaArco`).

e procederemo nella maniera più semplice e diretta:

1. aprire il file;
2. costruire un grafo vuoto;
3. leggere il numero dei nodi;
4. inserire il numero corretto di nodi nel grafo;
5. leggere i pesi dei nodi: per ciascuno, cercare il nodo nel grafo e assegnargli il peso corretto;
6. leggere gli archi: per ciascuno, cercare i nodi estremi nel grafo e inserire l'arco nel grafo;
7. leggere i costi degli archi: per ciascuno, cercare l'arco nel grafo e assegnargli il costo corretto;
8. chiudere il file;
9. restituire il grafo.

Nel codice seguente per chiarezza si sono eliminati i controlli relativi all'esito delle operazioni di apertura del file (`fopen`), lettura (`fscanf`) e ricerca dei *token* (`TrovaStringa`).

```
grafo *LoadGraph (char* nomefile)
{
  FILE *f;
  grafo *G;
  char s[LENGTH];

  int NumNodi, i, j;
  nodo *N1, *N2;
  arco *A;
  int costo, peso;

  f = fopen(nomefile,"r");
  G = CreaGrafo();

  // Cerca il numero dei nodi (token "n")
  TrovaStringa(f,"n");

  // Ignora il ":=" e legge il numero dei nodi
  fscanf(f,"%*s %d",&NumNodi);

  // Inserisce i nodi nel grafo
  for (i = 1; i <= NumNodi; i++) {
    N1 = CreaNodo(i);
    InsNodo(N1,G);
  }

  // Cerca i pesi dei nodi (token "w")
  TrovaStringa(f,"n");

  // Ignora il ":=" e legge i pesi dei nodi
  // (sfruttando la conoscenza del loro numero)
  fscanf(f,"%*s");
  for (i = 1; i <= NumNodi; i++) {
    fscanf(f,"%d %d",&j,&peso);
    N1 = TrovaNodo(j,G);
    N1.peso = peso;
  }

  // Cerca l'insieme degli archi (token "A") e ignora il ":="
  TrovaStringa(f,"A");
  fscanf(f,"%*s");

  // Scorre gli archi fino al ";" e li inserisce nel grafo
  fscanf(f,"%s",s);
  while (sscanf(s,"(%d,%d)",&i,&j) == 2) {
    N1 = TrovaNodo(i,G);
```

```
    N2 = TrovaNodo(j,G);

    A = CreaArco(N1,N2);
    InsArco(A,G);

    fscanf(f,"%s",s);
  }

  // Cerca i costi degli archi (token "c") e ignora il ":="
  TrovaStringa(f,"c");
  fscanf(f,"%*s");

  // Scorre gli archi fino al ";" e li inserisce nel grafo
  fscanf(f,"%s",s);
  while (sscanf(s,"[%d,%d]",&i,&j) == 2) {
    fscanf(f,"%d",&costo);

    A = TrovaArco(i,j,G);
    A.costo = costo;

    fscanf(f,"%s",s);
  }

  fclose(f);
  return G;
}
```

Il codice precedente può essere reso un po' più efficiente se si può dare per scontato che ogni nodo abbia un peso e ogni arco un costo. In tal caso, infatti, il ciclo di creazione dei nodi potrebbe essere fuso con quello di lettura dei pesi e il ciclo di creazione degli archi con quello di lettura dei costi. Se il grafo non è enorme, la lettura dei dati ha scarso impatto sul tempo di esecuzione del programma, per cui tali ottimizzazioni sono superflue. Conviene però farle se si hanno forti indizi del contrario.