RELAZIONE

LORENZO BUOMPANE
S247756

POLITECNICO
DI TORINO

Algoritmi e Programmazione
Appello del 28/01/2020 - Prova di programmazione (18 punti)

Analisi e commento

Analisi delle specifiche

Il problema ci propone un grafo `G=(V,E)` non orientato, semplice e connesso. Si riceve il nome del file, con la descrizione del grafo `G`, come argomento tramite riga di comando. Il programma deve inizialmente leggere da tastiera un intero `k≥0` e visualizzare l'elenco degli vertici del **k-core**, se esiste, ovvero quel sottografo massimale di vertici tale che ognuno di essi abbia grado `≥k`.
Successivamente, dopo aver letto da tastiera un intero `j≥1`, si verifichi se `G` sia **j-edge-connected**, ovvero se esiste un insieme di archi di cardinalità j in grado di sconnetterlo. Dopo aver escluso l'esistenza di insiemi di archi di cardinalità `<j` in grado di sconnetterlo, si visualizzi almeno un sottoinsieme di archi che lo sconnette.
Nel programma sono state utilizzate funzioni di *"librerie standard"*, apportando delle piccole modifiche per completare la struttura dati lievemente modificata.

Struttura dati

L'organizzazione dei dati acquisiti tramite lettura da file vengono organizzati in:

- Un grafo realizzato come ADT di I classe (*Graph*), in cui vengono memorizzati il numero di vertici `V`, il numero di archi `E`, la matrice delle adiacenze `**madj` (utilizzata per rappresentare gli archi del grafo: per ogni coppia `(i,j)` di vertici che crea un arco viene salvato 1 in `madj[i][j]` (e `madj[j][i]`, poichè il grafo non è orientato), altrimenti resta 0 indicando la non presenza di un arco tra i due vertici), una tabella di simboli `tab` e due vettori di interi `*vett` e `*grado`. Si può notare la presenza dei due vettori che vanno a modificare la struttura dati standard.

`int`	`V`
`int`	`E`
`int`	`**madj`
`ST`	`tab`
`int`	`*vett`
`int`	`*grado`

- Una tabella di simboli che fornisce una relazione chiave-indice. Anch'essa realizzata coma ADT di I classe (*ST*), presenta al suo interno una matrice di caratteri `**a`, su ogni riga viene salvata la chiave del vertice, che viene quindi associato all'indice di quella riga. Inoltre ci sono due valori interi `maxN` (indica le righe della matrice) e `N` che indica il numero di vertici presenti nella tabella.

`char`	`**a`
`int`	`maxN`
`int`	`N`

La matrice delle adiacenze di `G` ha dimensione `(VxV)`, poiché ogni vertice potrebbe avere un arco che collega ogni altro vertice. Essa viene inizializzata a 0. Nel momento dell'acquisizione degli archi, vengono lette da file coppie di chiavi, si cerca il loro indice mediante la tabella di simboli ed infine cambio a 1 il valore di `madj[i][j]` (e `madj[j][i]`), con `i` e `j` gli indici dei due vertici. Il vettore `*vett` ha dimensione `V` e viene inizializzato a 1: indica lo stato di preso (0 non preso). Altrettanto `*grado` ha dimensione `V` e viene inizializzato a 0: vettore che indica il grado del vertice. Esso viene incrementato ogni volta che un arco viene inserito nel grafo. La matrice `**a` presente nella tabella di simboli ha dimensione `(VxMAXC)` (`MAXC` variabile globale che indica i caratteri massimi della chiave di un vertice, compreso il fine stringa).

Strategia risolutiva

Acquisizione del file

Il nome del file passato come argomento sulla riga di comando presenta un formato standard con il numero di vertici V sulla prima riga, seguito da V righe con il nome dei vertici ed infine un numero non definito di coppie di vertici rappresentanti gli archi del grafo.

La memorizzazione del grafo nelle strutture dati *Graph* e *ST* avviene tramite la funzione `GRAPHload`, che riceve come parametri il file di input `*fin` e ritorna il grafo G, inizializzato e con i dati acquisiti.

k-core

La funzione `GRAPHkcore` richiede come parametro il grafo `G` e un intero $k \geq 0$ acquisito da tastiera. Essa controlla il vettore dei gradi ed elimina *"logicamente"* il vertice con grado minore di `k`. Questo controllo viene ripetuto fino a quando i vertici non eliminati hanno tutti grado $\geq k$. La stampa dell'insieme di vertici avviene tramite la funzione `PRINTkcore` solo se è presente un sottografo `k-core` valido, ovvero se non sono stati *"logicamente"* eliminati tutti i vertici. Prima di terminare ripristina i valori nei vettori `*vett` e `*grado`, tramite la funzione `GRAPHrestore`.

j-edge-connected

La funzione `GRAPHjedgeconnected` richiede come parametro il grafo `G` e un intero $j \geq 1$ acquisito da tastiera. Il programma deve capire se è il grafo è connesso o sconnesso togliendo un insieme di archi. Per calcolare tutti gli insiemi di archi è stato utilizzato il modello delle combinazioni semplici, avente come vettore dei valori il vettore di archi creato dalla funzione `GRAPHedges`. La funzione `comb_sempl` crea tutte le combinazioni di archi che verranno eliminate . Per ogni insieme la funzione `CHECKjedgesconnected` controlla se il grafo è connesso o sconnesso. Se è sconnesso il grafo **j-edge-connected** con `j` che è il numero di archi eliminati. Il controllo avviene attraverso la funzione `GRAPHcc` che ritorna la componenti connesse. La funzione `comb_sempl` viene chiamata con il parametro `i` (grandezza dell'insieme di archi da eliminare) con $1 \leq i < j$ per escludere la presenza di un insieme di archi con cardinalità minore di `j` in grado di sconnette il grafo. Se esso non è presente si usa la funzione `comb_sempl` con `j` come grandezza dell'insieme di archi da eliminare. Se esiste stampo l'insieme di archi che vengono rimossi.

Osservazione: è stato scelto il modello delle combinazioni semplici perché l'ordine degli elementi non conta (grafo non orientato), l'elemento non può essere preso più di una volta (non ha senso eliminare due volte lo stesso arco).

Funzioni non presenti nelle librerie standard o modificate

```
Graph GRAPHload(FILE *fin);
```

La funzione `GRAPHload` è una funzione di libreria standard che dichiara una struct grafo `G`. Legge la prima riga del file contenente il numero di vertici e lo salva in una variabile temporanea `V`. Viene poi inizializzata la struct grafo con la funzione `GRAPinit` che chiede come parametro il numero di vertici `V`. La funzione di inizializzazione ritorna il grafo G allocato. Si procede con l'acquisizione da file leggendo `V` vertici, salvandoli nella tabella di simboli per associare ad ogni chiave del vertice un indice. Infine vengono lette le coppie di vertici che formano un arco fino alla fine del file. Per ogni coppia viene cercato l'indice del vertice (`STsearch`) e inserito l'arco nel grafo mediante la `GRAPHinsertE` (troviamo all'interno la funzione `insertE` che mette a 1 la cella della matrice delle adiacenze corrispondente ai due indici). La funzione ritorna il grafo `G`.

```
Graph GRAPHinit(int V);
```

La funzione `GRAPHinit` è la funzione standard con due piccole modifiche per aggiungere l'allocazione e l'inizializzazione dei due vettori `*vett` e `*grado`. Il primo ha dimensione `V` e viene inizializzato a 1 (tutti i vertici sono presenti *"logicamente"* nel grafo). Anche il secondo ha dimensione `V` ma viene inizializzato a 0 (inizialmente i vertici non hanno archi quindi il loro grado è 0, ogni volta che viene inserito un arco i due vertici adiacenti aumenteranno il loro grado di 1).

```
void GRAPHfree(Graph G);
```

La funzione `GRAPHfree` è una funzione di libreria standard a cui viene aggiunta la `free` dei vettori `*vett` e `*grado`.

```
void  insertE(Graph G, Edge e);
```

La funzione `insertE` è una funzione di libreria standard a cui viene aggiunto l'incremento del grado nei due vertici adiacenti.

```
void  removeE(Graph G, Edge e);
```

La funzione `removeE` è una funzione di libreria standard a cui viene aggiunto il decremento del grado nei due vertici adiacenti.

```
void GRAPHkcore(Graph G, int k);
```

La funzione `GRAPHkcore` presenta al suo interno un ciclo `while` in cui si scandisce il vettore dei gradi e per ogni vertice controlla il suo grado (presente nel vettore `*grado`), se esso è minore di `k` allora procede all'eliminazione *"logica"* del vertice in considerazione tramite la funzione `GRAPHkcoreremove` (questo avviene solo se il vertice non è ancora stato eliminato, controllando che il valore della cella del vettore `*vett` in posizione *"indice del vertice"* sia diverso da 0). Il ciclo termina quando la funzione `CHECKkcore` ritorna 1. Finito il ciclo viene controllato un `if` con condizione la funzione `EXISTkcore` che controlla che non siano stati eliminati tutti i vertici. Nel caso la condizione sia verificata stampa l'elenco di vertici presenti nel `k-core`, altrimenti stampa un messaggio di non presenza di un sottografo `k-core`. Prima della terminazione viene richiamata la funzione `GRAPHrestore`.

```
Graph GRAPHkcoreremove(Graph G, int i);
```

La funzione `GRAPHkcoreremove` elimina il vertice `i`-esimo. Imposta a 0 `vett[i]` (vertice eliminato) e a -1 `grado[i]` (vertice eliminato). Controlla, tramite la matrice delle adiacenze, vertici adiacenti al vertice `i`: viene decrementato di 1 il grado del vertice `j` adiacente a `i`. Ritorna il grafo G modificato.

```
int CHECKkcore(Graph G, int k);
```

La funzione `CHECKkcore` ritorna 1 se tutti i vertici non eliminati hanno grado $\geq$ `k`, altrimenti ritorna 0.

```
int EXISTkcore(Graph G);
```

La funzione `EXISTkcore` ritorna 0 se non sono stati eliminati tutti i vertici, altrimenti ritorna 1.

```
void PRINTkcore(Graph G);
```

La funzione `PRINTkcore` stampa i vertici non eliminati (`vett[i]==1`). Stampa la chiave del vertice che recupera dalla tabella di simboli grazie alla funzione `STsearchByIndex`.

```
void GRAPHrestore(Graph G);
```

La funzione `GRAPHrestore` riporta tutto il vettore a 1 `*vett` e a 0 `*grado`. Ricalcola, grazie alla lettura della matrice della adiacenze, i gradi di tutti i vertici.

```
void GRAPHjedgesconnected(Graph G, int j);
```

La funzione `GRAPHjedgeconnected` inizialmente dichiara un intero, `flag`, e inizializzo uguale a 0 (verrà utilizzato per non continuare le combinazioni semplici se si è trovato un insieme che sconnette `G`), alloca un vettore di archi di `E` elementi che viene riempito dalla funzione di libreria `GRAPHedges` e un vettore di archi di dimensione `j` (cardinalità massima dell'insieme di archi che devo controllare). Un `if` controlla che `j` non sia uguale a 1 (mi evita la `printf` inserita nel codice e inutile per `j==1`, poiché non ci sarà da escludere nulla ma solo controllare se è **1-edge-connected**). Un ciclo `for` richiama `comb_sempl` per ogni cardinalità `i` compresa tra 1 e `j` (escluso). Facendo ciò si può escludere o meno la presenza di un insieme di cardinalità minore di `i`, grazie al `flag` passato *by reference*. Se quest'ultimo è 1 l'insieme di archi esiste ed è di cardinalità `i`: stampo l'insieme ed interrompo il ciclo (il `flag` essendo a 1 non controlla le combinazioni con `j` ma stampa a schermo il messaggio che il grafo non è **j-edge-connected**); altrimenti se è 0 la funzione continua e controlla se esiste un insieme di cardinalità `j`. Controllo ulteriormente il `flag`: se 1, stampo l'insieme e termino (**j-edge-connected: SI**); se 0 termino (**j-edge-connected: NO**).

```
void comb_sempl(Graph G, int pos, Edge *val, Edge *remove,
      int E, int j, int start, int *flag);
```

La funzione `comb_sempl` calcola tutte le combinazioni semplici, salvando nel vettore `*remove` solo quelle che rispettano la condizione `pos==j`, ovvero la cardinalità dell'insieme di archi che si sta valutando. Quando la condizione precedente è valida controlla tramite la funzione `CHECKjedgesconnected` se il grafo è connesso o sconnesso. Nel primo caso imposta `flag` uguale a 1 facendo terminare la funzioni senza valutare le combinazioni successive. Nel secondo caso continuo cercando una combinazione valida (se nessuna combinazione è valida significa che `G` non si sconnette con nessun insieme di archi di quella cardinalità). La funzione è ricorsiva e riesce a calcolare tutti gli insiemi possibili.

```
int CHECKjedgesconnected(Graph G, Edge *remove, int j);
```

La funzione `CHECKjedgesconnected` rimuove dal grafo `G` gli archi presenti nel vettore `*remove`, utilizzando la funzione di libreria standard `GRAPHremoveE` (al suo interno chiama `removeE`). Richiama la funzione `GRAPHcc` che ritorna il numero delle componenti connesse che viene salvato nella variabile `id`. Prima di terminare inserisce nel grafo `G` gli archi che ha eliminato per controllare la connessione o sconnessione del sottografo.

```
int GRAPHcc(Graph G);
```

La funzione `GRAPHcc` è stata opportunamente modificata per renderla compatibile con la rappresentazione con matrice delle adiacenze. È stata aggiunta anche la `free` del vettore `*cc`.

```
void dfsRcc(Graph G, int v, int id, int *cc);
```

La funzione `dfsRcc` è stata opportunamente modificata per renderla compatibile con la rappresentazione con matrice delle adiacenze.

Elenco delle differenze

`main.c`

- Aggiunte delle `printf` per spiegare le funzioni che esegue e, soprattutto, per richiedere i dati da tastiera.
- Aggiunto il controllo sull'apertura del file e sull'allocazione del grafo.
- Aggiunto controllo sulla condizione `k≥0` e `j≥1`.
- Aggiunta la funzione `GRAPHfree`.

`Graph.c`

- Sostituito V con G->V.
- `GRAPHkcore`: aggiunta la chiamata alla funzione di stampa, esistenza e restore del k-core *(erano implementate ma non chiamate)*.
- `CHECKkcore`: aggiunta condizione `G->vett[i]!=0` (in &&).
- `PRINTkcore`: in fase d'esame ho predisposto la stampa con la stampa degli archi e non dei vertici, sostituito `EDGESprint` con una semplice `printf` dei vertici.
- Implementata la funzione `GRAPHjedgesconnected`.

Analisi delle differenze

Durante la prova d'esame sono state richiamate le funzioni di libreria standard indicando in breve quali dovevano essere le modifiche da aggiungere (tutte le differenze riguardano i due vettori `*vett` e `*grado`, per ogni funzione sono descritte a pagina 4). I controlli sull'apertura del file o sull'allocazione sono stati aggiunti durante la trascrizione del codice. Vengono aggiunte le chiamate a funzioni implementate ma che presentavano alcun richiamo nel codice cartaceo, e implementata la funzione j-edge-connected. Infine è stata aggiunta una condizione in un costrutto `if` all'interno di `CHECKkcore`, poiché venivano eliminati vertici già rimossi in precedenza.