Relazione di Algoritmi e programmazione (prova da 18 punti)

Matricola s256811 – MASCITTI KEVIN

Strutture dati

Ho utilizzato un ADT di 1 classe per definire il tipo Graph, un quasi ADT per definire il tipo Edge, un ADT di 1 classe per definire la tabella di simboli ST (campo tab della struct graph). Le strutture utilizzate sono standard, ad eccezione della struct graph in cui ho inserito un campo aggiuntivo int *vertici, un vettore di V posizioni allocato dinamicamente in GRAPHinit, utile per contrassegnare se il vertice i-esimo fa parte del grafo oppure è stato rimosso. Esso è dunque inizializzato a 1 all'interno della funzione GRAPHinit poiché, prima di chiamare la funzione Kcore, tutti i vertici dati dal file fanno parte del grafo.

Algoritmi utilizzati

Per la **prima richiesta** del programma ho utilizzato la funzione Kcore. Essa alloca un vettore di interi "gradi" che in ogni casella conterrà il grado del vertice i-esimo calcolato attraverso la funzione calcolaGradi, che conta per ogni vertice il numero di vertici a cui esso è collegato (si considerano solo i vertici ancora esistenti nel grafo). Inoltre Kcore alloca un vettore di archi removedEdges, inizializzato per ogni posizione ad archi fittizi da -1 a -1. In seguito procedo iterativamente (finché esistono vertici nel grafo e finché esiste anche un solo vertice con grado < k nel grafo) rimuovendo i vertici con grado < k e aggiornando il vettore gradi. Per rimuovere eventuali vertici, chiamo la funzione removeV che analizza ogni vertice esistente per cui, se esso ha grado < k, rimuove dal grafo il vertice stesso e i relativi archi ad esso incidenti attraverso la funzione rimuoviArchi, la quale rimuove gli archi interessati dal grafo e li inserisce nel vettore di archi removeEdges. Al termine dell'iterazione, se esistono ancora vertici nel grafo, il grafo rimanente è detto k-core del grafo originario e si stampano i vertici ancora esistenti, altrimenti non è un k-core. Infine si ripristina il grafo originale re-inserendo gli archi rimossi e si libera la memoria allocata.

Per la **seconda richiesta** del problema ho utilizzato la funzione EDGEconnected e ho diviso il procedimento di verifica di esclusione di insiemi di cardinalità i < j in grado sconnettere il grafo e quello di individuazione di un sottoinsieme di archi di cardinalità j. Ho utilizzato rispettivamente la funzione EDGEverify per il primo sottopunto e la funzione EDGEfind per il secondo. Entrambe si basano sull'algoritmo delle combinazioni semplici. Nella funzione EDGEconnected, dopo aver inizializzato i vettori di archi sol e a (vettore di archi del grafo), ho chiamato la funzione ricorsiva EDGEverify per i crescenti da 0 a j-1 per verificare che effettivamente il grafo non fosse i-edge-connected per i < j. EDGEverify ritorna 0 se la verifica non termina correttamente ed è stato individuato un insieme di archi a cardinalità i < j tali per cui il grafo risulta sconnesso; ritorna 1 altrimenti. Se EDGEverify ritorna 1, si chiama la funzione EDGEfind che ha il compito di individuare un insieme di archi a cardinalità j tale per cui il grafo risulti sconnesso. Se lo trova, essa ritorna 1 e il grafo è j-edge-connected, altrimenti ritorna 0 e il grafo sarà eventualmente j-edge-connected per valori di j maggiori di quelli passati alla funzione.

Le funzioni EDGEverify e EDGEfind sono analoghe ed eseguono delle combinazioni semplici sugli archi. In particolare, EDGEverify genera degli insiemi a cardinalità i e ricorre per k+1 finché k non è maggiore uguale alla cardinalità i (condizione di terminazione). All'interno della condizione di terminazione, vengono rimossi gli archi del vettore sol, si verifica che il grafo risultante sia connesso e, se non lo è, si ritorna 0 alla funzione EDGEconnected, altrimenti si continua in EDGEverify reinserendo gli archi del vettore sol nel grafo e ritornando 1 alla chiamata ricorsiva, fino a terminare tutte le combinazioni possibili. Qualora nella condizione di terminazione venga trovato un insieme di archi a cardinalità i < j, è stato scelto di non reinserirli nel grafo in quanto inutile ai fini dell'esercizio dato che, se questa condizione è verificata, non devo svolgere nessun'altra operazione di verifica in seguito sul grafo e il programma termina dopo aver eventualmente stampato dei messaggi a video. Si poteva ovviamente scegliere di reinserirli prima di ritornare il valore 0, per ripristinare il grafo originale.

La funzione EDGEfind è uguale a quella appena descritta ma opera su cardinalità j e ritorna dei valori diversi. Se nella condizione di terminazione individua un insieme di archi a cardinalità j tale che, rimuovendoli, il grafo sia sconnesso, allora li stampa e ritorna 1 a EDGEconnected interrompendo le altre eventuali ricorsioni, altrimenti ritorna 0 e si continua ricorsivamente a cercare un insieme sol che soddisfi la condizione. Se alla fine di tutte le chiamate ricorsive non viene trovato nessun insieme sol idoneo, la funzione ritorna 0 a EDGEconnected. Anche in questa funzione vale la stessa cosa sottolineata in precedenza riguardo gli archi rimossi dal grafo e non reinseriti.

Ho implementato le funzioni più semplici che avevo tralasciato per concentrarmi sugli algoritmi più complessi. In particolare, ho sviluppato le funzioni:

- *min*: trova il minimo di un vettore;
- *emptyV*: ritorna 1 se il grafo modificato non ha più vertici "disponibili";
- *ripristina*: ri-inizializza il vettore G->vertici a 1 per segnalare che tutti i vertici tornano a far parte del grafo e reinserisce gli archi rimossi presenti nel removedEdges;
- *EDGEisnull*: ritorna 1 se l'arco passato è fittizio (e.v=e.w=-1);
- *findempty*: ritorna la posizione del vettore removedEdges in cui c'è un arco fittizio e ritorna -1 se il vettore di archi è pieno (non è possibile che lo sia a meno di qualche errore, perché il vettore removedEdges può contenere al massimo G->E archi, ovvero il numero massimo di archi che è possibile rimuovere da G).

Differenze tra compito cartaceo e codice allegato

- Nel cartaceo ho abbozzato un main negli ultimi minuti, ma nell'allegato l'ho rifatto completamente.
- Nella funzione Kcore: ho inizializzato il vettore di archi removedEdges (per poi poter effettuare il controllo di EDGEisnull); ho inserito nel while la condizione di esistenza di almeno un vertice all'interno del grafo; ho inserito i parametri nella funzione removeV perché l'avevo dimenticato per la fretta; ho inserito le free.
- In calcolaGradi ho inserito, subito dopo la chiamata del primo ciclo for, l'inizializzazione di gradi[i] a 0.
- Nella versione cartacea, nel prototipo della funzione removeV, avevo dimenticato di inserire "int *gradi".
- Nel codice allegato, nella funzione rimuoviArchi, dopo G->madj[id][j]=0, ho aggiunto anche G->madj[j][id]=0 (distrattamente non ricordavo che il grafo fosse non orientato), ma ho scelto di non modificare G->E decrementandolo, perché in altre funzioni mi servirà il suo valore effettivo.
- Nel cartaceo, nel prototipo delle funzioni EDGEverify e EDGEfind (analoghe come già spiegato nella sezione *Algoritmi*) ho dimenticato di inserire il parametro int start e ho fatto confusione con gli indici del ciclo for perché preso dall'ansia e dalla fretta di dover consegnare. Nel codice allegato, ho corretto questo e l'indice del vettore a, subito dopo la chiamata del for. Nella condizione terminale della EDGEverify ho inoltre modificato la printf, incompleta nel cartaceo.