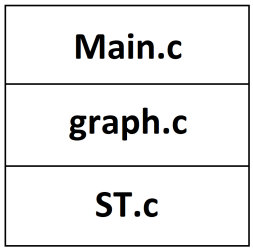
Damiano Da Col 281127

Relazione Lab 9

Il programma è composto da 3 moduli: main.c, graph.c ed ST.c, che rispettivamente rappresentano il client, il modulo per salvare il grafo e quello per salvare la tabella di simboli associata al grafo.

Strutture dati

Il grafo è definito come ADT di I classe con le definizioni del puntatore a grafo e della struct edge degli archi, nell'header graph.h. I dettagli interni del grafo sono definiti in graph.c e sono: la lista delle adiacenze e la struct graph invariata rispetto a quella fornita di libreria.



La tabella di simboli è a sua volta un ADT di I classe con la dichiarazione del puntatore a ST: *tab in ST.h e quella alla vera e propria struttura dati in ST.c. La tabella di simboli in questo caso è una struct che comprende il vettore di stringhe con i nomi dei vertici e un intero per tenere conto del numero effettivo di vertici presenti nel vettore.
il client main.c ha soltanto accesso al puntatore a grafo e coordina le chiamate alle funzioni di graph.c, per eseguire tutte le richieste dell'esercizio.

Scelte algoritmiche

- Individuazione di tutti gli insiemi di archi di cardinalità minima la cui rimozione renda il grafo originale un DAG

Nel caso in cui il grafo non soddisfacesse già i vincoli per essere un DAG, ho risolto un problema di ricerca e ottimizzazione tramite un algoritmo ricorsivo, che si basa sulle combinazioni semplici di tutti gli insiemi possibili di archi su **k** posizioni, con **k** che parte da E-1 (numero di archi - 1) e decresce fino a quando non trovo almeno una soluzione (insieme di archi), il cui grafo sia un DAG; oppure fino a quando arrivo a **k** = V-1 (numero di vertici - 1) sotto il cui valore non avrei più un grafo connesso in nessuna delle soluzioni e quindi nemmeno un DAG.
Quando nella ricorsione, ottengo una soluzione completa (un vettore di **k** archi) entro nella condizione di terminazione, in cui la funzione **GRAPHfromEdges()** costruisce il grafo associato all'insieme di archi della soluzione corrente. Una volta costruito il grafo, viene verificata la condizione per avere un DAG tramite una visita in profondità, operata dalle funzioni di libreria **GRAPHdfs()** e **dfsR()** con le opportune modifiche. Ho rimosso la stampa dei vertici e sfruttato la parte che etichetta ogni arco, per verificare la presenza di archi **Back** e quindi di cicli nel grafo; nel caso siano presenti cicli la soluzione non risulta valida e il grafo non è un DAG, quindi la ricorsione continua fino a quando avrà trovato tutte le soluzioni valide per il **k** più alto possibile (in modo da rispettare la richiesta di rimuovere il minor numero possibile di archi).
La complessità della funzione ricorsiva **comb_sempl()** risulta essere quella di un algoritmo del calcolo combinatorio per enumerare le combinazioni semplici di **n** archi su **k** spazi: $O\binom{n}{k}$.
Una possibile soluzione per ridurre **k** e quindi la complessità data dalle combinazioni semplici, sarebbe stata quella di cercare una soluzione valida, composta dagli archi rimossi piuttosto che una soluzione data dagli archi ancora presenti nel grafo dopo la rimozione.

- Costruzione di un DAG rimuovendo, tra tutti gli insiemi di archi generati al passo precedente, quelli dell'insieme a peso massimo

Ho implementato questa richiesta direttamente nella funzione **comb_sempl()**, che trovata una soluzione valida, chiama la funzione di stampa **printMinEdgeSet()** che visualizza a schermo l'insieme di archi da rimuovere, cercando tra tutti gli archi del grafo iniziale, quelli non presenti nella soluzione data e calcola la somma dei pesi di questi archi. Nel caso questa somma sia maggiore di quella massima attualmente salvata, il valore del massimo e quelli della soluzione massima vengono aggiornati. In questo modo alla fine della ricorsione si avrà il vettore **max_sol** di archi che rappresenta il DAG richiesto.

- Calcolo delle distanze massime da ogni nodo sorgente verso ogni nodo del DAG costruito al passo precedente

Per prima cosa il client chiama la funzione di libreria **DAGts()** che ordina il vettore di vertici in ordine topologico; poi chiama la funzione **GRAPH_nodiSorgente()** che salva in un vettore tutti i nodi sorgente del DAG, e cioè tutti i nodi che hanno *in_degree == 0*. Infine il modulo main.c chiama iterativamente (per ogni nodo sorgente), la funzione **GRAPH_DAGmaxPath()** che calcola e stampa la distanza massima tra un determinato nodo sorgente e tutti i nodi raggiungibili da quest'ultimo. La funzione **GRAPH_DAGmaxPath()** riceve il nodo sorgente e il vettore di vertici in ordine topologico e per tutti i vertici successivi a quello sorgente nel vettore, applica la *relaxation* inversa a tutti i loro archi uscenti. Nel processo, il vettore di interi **d** passa dal contenere il valore di $-\infty$, per ogni vertice che non fosse quello sorgente, a contenere la distanza massima di quel vertice da quello sorgente.
La complessità della funzione di verifica del cammino massimo **GRAPH_DAGmaxPath()**, risulta O(V*E) con **V** numero di vertici ed **E** numero di archi del grafo ed è data dal ciclo **for** su tutti i nodi e dal **for** annidato che itera su un vertice nella lista di adiacenze. Se avessi utilizzato la matrice delle adiacenze, avrei potuto migliorare la complessità di caso peggiore in quanto il **for** annidato sarebbe stato sostituito da un accesso diretto O(1) alla matrice delle adiacenze.