SEBASTIANO GUZZONE S270855

RELAZIONE APA LAB-13

Descrizione generale e perché ho scelto questo esercizio

Il problema generale include l'ADT grafo con ovviamente alcune varianti che dovranno essere sviluppate. Ho scelto questo esercizio per la relazione perché tutta la descrizione del problema ruota attorno alle ultime lezioni e principalmente alla sola descrizione di grafo.

MAIN

Il main si occupa semplicemente di richiamare le funzioni presentate dall'ADT GRAPH. Da menzionare che non è stato usato un grafo aggiuntivo per la definizione e l'utilizzo del DAG, ma è stato invece modificato il grafo di partenza (se necessario) per renderlo un DAG.

GRAFO

- Ho preferito usare la matrice di adiacenza in quanto l'eliminazione dell'arco nel grafo ha costo o(1). Vengono considerati soltanto gli archi NON NEGATIVI (peso arco>=0).
  Di conseguenza le scelte possibili per la matrice di adiacenza possono essere molteplici: Prima ho implementato una matrice aggiuntiva V*V con valori 1 e 0, poi ho deciso di sfruttare la proprietà degli archi presenti (NON NEGATIVI) per cancellare "logicamente" e non fisicamente un arco semplicemente moltiplicandolo per *-1, cosi da avere il suo opposto negativo e quindi non considerato come arco.

GRAPH.h/.c

- **STRUTTURA** Struttura base con V,E, int **madj e ST tab
- **FUNZIONI** Oltre le funzioni basi, si notano le seguenti funzioni/modifiche:
    - int GRAPHdfs(Graph G, int id)
      diversamente dalla funzione base, sfrutto questa GRAPHdfs come valore utile per il check che verrà usato nella ricorsione per capire se l'insieme di archi di K elementi eliminati "logicamente" è riuscito a rendere il grafo in un DAG

    - void dfsR(Graph G, Edge e, int *time, int *pre, int *post, int *st, int *p_check)
      Il valore di ritorno del GRAPHdfs viene modificato da questa versione particolare del dfsR.

    - void GRAPHdisplay(Graph G)
      Qui per la stampa si controlla che "G->madj[i][j]>=0"

    - int powerset_r(Edge * val, int k, Edge * sol, int j, int pos, int start, Edge * b_sol, int *p_sommabest, Graph G)
      In questa versione sfrutto le combinazioni semplici (archi tra loro diversi, no multigrafo, no ripetizioni, non importante l'ordinamento.

      È chiaro che il powerset, nel caso peggiore, deve analizzare tutto l'insieme delle parti, considerando il numero di elementi in considerazione ('E' numero di archi), il powerset avrà complessità O(2^E).

      Ovviamente dopo aver trovato il primo set di k archi che rende DAG il mio grafo, cerco solo i rimanenti subset da calcolare aventi k archi.
      Ad ogni subset che soddisfa e rende il mio grafo DAG faccio un controllo per verificare che non sia un subset dal peso

maggiore, in tal caso invece sostituisco il precedente subset con il nuovo trovato.

In base al grafo denso o meno ovviamente può essere più veloce l'utilizzo di lista di adiacenza oppure matrice, ma essendo il costo di calcolo dell'insieme delle parti di per se esponenziale, in ogni caso la ricerca di un arco da andare ad inserire nel subset non cambierà la complessità dell'algoritmo.

- void DAGssmp(Graph D)

  Questa funzione inizialmente richiama una visita in profondità, utilizza per ottenere un ordinamento topologico inverso del nostro DAG. Fatto ciò invertiamo l'ordine del vettore di vertici così da avere un ordinamento topologico. A questo punto, essendo il grafo un DAG applicheremo un l'algoritmo per DAG pesati così da considerare ogni vertice in ordinamento topologico e per ogni suoi arco uscente applicheremo una relaxation inversa, così da trovare il massimo.
  La visita in profondità effettuata in questo caso non ha modifiche del codice. Le stime delle distanze iniziali vengono inizializzate a -1 (non può essere scelto 0 data la presenza di archi NON NEGATIVI).

ST.h/.c

1. **STRUTTURA** Implementazione della struttura base senza modifiche di rilievo per il salvataggio dei vertici del grafo.