Cognome e nome: Fassio Davide
Matricola: s268519
Laboratorio n° 13

Gerarchia del progetto

Il progetto è suddiviso in 4 moduli:

1. Tabella di simboli (tab_simb.h / tab_simb.c)
2. Matrice (mat.h / mat.c)
3. Lista (lista.h / lista.c)
4. Grafo (grafo.h / grafo.c)

I moduli 1,2,3 sono necessari per il modulo 4.
Le funzionalità del modulo grafo sono poi importate e utilizzate dal client E01.c.

Si analizzano ora le strutture dati e funzionalità di ciascun modulo.

Tabella di simboli

Implementata come ADT di I classe.
L'implementazione interna presenta 4 campi:

1. char **data: Vettore di stringhe di lunghezza nota (len)
2. int cap: Capacità massima della tabella di simboli = numero di righe di data
3. int n: Numero di righe attualmente utilizzate in data
4. int len: Lunghezza di una stringa = numero di colonne in data

La tabella di simboli è ad accesso diretto e non è implementata come tabella di hash.

Presenta le funzioni per allocare la tabella (TS_init) e liberare la memoria (TS_free).
La funzione di inserimento in tabella (TS_insert) controlla se la stringa da inserire è già presente nella tabella, se non lo è allora inserisci.
La funzione TS_getNameByIndex restituisce il puntatore alla stringa all'indirizzo passato senza farne una copia e quindi senza allocare nuova memoria.
La funzione TS_getIndexByName restituisce l'indice a cui è presente la stringa passata, se non è stata trovata allora restituisce -1.

Matrice

Implementata come quasi ADT per permettere ai client di accedere a data direttamente.
È lo stesso fornita una definizione più compatta per il puntatore alla struttura (mat.h, riga 13).
Il numero di righe è rappresentato come long int per permettere di accomodare le combinazioni generate dal modulo grafo.
Le funzioni presenti sono quelle di allocazione (M_init), liberamento della memoria (M_free) e stampa a schermo (M_print).

Lista

Nel modulo lista sono presenti due ADT di I classe: link e Lista.

Il nodo della lista contiene al suo interno due interi, l'id del nodo e il valore dell'arco.
Sono presenti i getter per i campi del nodo (Node_getId, Node_getVal, Node_getNext) e le funzioni per allocazione e inizializzazione (Node_init) e liberamento della memoria (Node_free).

La lista contiene un puntatore alla testa della lista e un contatore degli elementi della lista.
La lista è una lista linkata semplice e non utilizza nodi sentinella.

Sono presenti le funzioni di allocazione (L_init) e liberamento della memoria (L_free) implementato ricorsivamente.
Ci sono due funzioni di push: L_push che alloca e inizializza il nodo e L_pushNode che inserisce il nodo gia creato.
La funzione di estrazione (L_extract) esegue una ricerca in lista e se il nodo è trovato lo estrae dalla lista e restituisce un puntatore ad esso.

Grafo

Implementato come ADT di I classe.
Il grafo viene rappresentato tramite la lista delle adiacenze. La lista delle adiacenze del grafo trasposto viene anche salvata per velocizzare l'esecuzione.
Il vettore varchi è un vettore di archi in cui ogni riga è un tupla (vertice di partenza, vertice di arrivo, peso dell'arco), viene utilizzato per individuare quali archi cancellare con le combinazioni nella funzione GRF_DAGify.
Il vettore ordtop è un vettore in cui i vertici di un DAG sono salvati secondo un ordinamento topologico, viene allocato e inizializzato dalla funzione GRF_printLongestPath.
Viene utilizzata una tabella di simboli per la corrispondenza: nome vertice ⇔ indice.

La funzione GRF_DAGify trasforma un grafo generico in un DAG togliendo il minor numero di archi possibili, se sono possibili più combinazioni elimina quella a peso maggiore.
Inizialmente controlla se il grafo è un DAG, se lo è esce dalla routine (grafo.c, riga 238).
Se non lo è calcola le combinazioni $\binom{numero\ nodi}{i}$ incrementando i fino a che non si ottiene un DAG.
Una volta trovata una combinazione che rende il grafo un DAG continua a cercarne altre con la stessa i. Trovate tutte calcola quello con il peso massimo e la rimuove.

La funzione che permette di identificare un DAG è isDAG.
Viene applicato l'algoritmo di Kosaraju e sappiamo che se un grafo è un DAG allora dopo la seconda vista in profondità risulta:

$$\forall v, post[v] - pre[v] = 1$$

Se questa condizione viene rispetta allora restituisce 1 (vero) altrimenti 0 (falso).

La funzione GRF_printLongestPath stampa a schermo i cammini massimi di ogni vertice partendo da un vertice passato. Il main chiamando la funzione per ogni vertice permette di visualizzare i cammini massimi da ogni vertice a ogni vertice.
Se il grafo non presenta un ordinamento topologico allora viene calcolato tramite una visita in profondita, prendendo i vertici in ordine decrescente di fine tempo di elaborazione (grafo.c, righe da 332 a 355).
Poi viene cambiato il segno dei pesi degli archi (peso = peso * -1), così è possibile applicare l'algoritmo per i cammini minimi sfruttando la programmazione dinamica (grafo.c, righe da 361 a 383). Vengono stampati i risultati e i pesi degli archi vengono ripristinati al valore originale.