Bellino Lorenzo Laboratorio 12 matricola n° 24544

Strutture dati utilizzate:

- Grafo rappresentato come lista delle adiacenze con relativa tabella di simboli entrambi rappresentati come ADT di prima classe.
- Matrice sovradimensionata (Edge **tentativi) per contenere gli insiemi di archi che è possibile rimuovere in modo da ottenere un DAG, ogni riga della matrice contiene un insieme di archi per cui la grandezza massima della matrice è N * N; infatti partendo da un grafo con N archi possiamo eliminare al massimo o un insieme unico composto da N archi, oppure N insiemi possibili composti da un solo arco. Tutte le altre soluzioni hanno dimensione minore.

Grafo

`typedef struct node *link;`
`struct node {int v; int wt; link next;};`
`struct graph {int V; int E; link *ladj; ST tab; link z; };`

Tabella di Simboli
`struct symboltable {char **a; int maxN;};`

Procedimento e tecniche algoritmiche impiegate:

Punto 0:

Per quanto riguarda il caricamento dei dati nelle strutture dati, siccome la struttura del file di input è quella standard possiamo ricondurci alle funzioni standard per l'ADT grafo.

Punto 1:

La prima richiesta era quella arrivare alla costruzione di un DAG partendo dal grafo iniziale; per generare l'insieme di archi da eliminare ho applicato un metodo ricorsivo riconducibile alle combinazioni semplici (in quanto non è importante l'ordine in cui gli archi vengono rimossi né poter considerare lo stesso arco più di una volta). Siccome il problema richiedeva di rimuovere uno tra gli insiemi a cardinalità minima, la funzione ricorsiva prova prima a cercare insiemi di cardinalità più bassa, incrementando successivamente nel caso in cui la discesa ricorsiva abbia riportato un risultato negativo.

```
int tentativo_DAG(...){
    int i,j;
    if(pos>=k){
    for(j=0;j<k;j++){
       GRAPH_remove(G,sol[j].v,sol[j].w);
    }
    if(dfs_mod(G)==0){
        for(j=0;j<k;j++){
            tentativi[count][j]=sol[j];
        }
        count++;
    }
    for(j=0;j<k;j++){
         insert_edge(...));
        }
        return count;
    }
    for(i=start;i<n;i++){
        sol[pos]=val[i];
      count=tentativo_DAG(...);
    }
    return count;
```

Una volta generato un insieme di archi generico l'algoritmo prova a rimuoverlo e compie una verifica sulla aciclicità del grafo così ottenuto, per questo controllo ho utilizzato un'altra funzione standard opportunamente modificata per questo compito, la depth-first search (DFS), che ritorna al chiamante il numero di archi di tipo back presenti. Nel caso questo valore fosse zero allora l'insieme viene aggiunto assieme di candidati nella matrice già citata più in alto. In ogni caso l'insieme di archi viene reinserito nel grafo e viene aumentanto un counter per tenere traccia del numero di soluzioni trovate. La ricorsione a questo punto continua fino a che non ritorna il suo risultato alla funzione chiamante. In caso di risposta negativa (se non è stata trovata alcuna soluzione con quella cardinalità) la dimensione viene aumentata e funzione ricorsiva ricomincia. In caso di valore di ritorno diverso da zero il ciclo for si interrompe perché abbiamo soddisfatto la richiesta.

Punto 2:

Ora conosciamo sia il numero di soluzioni candidate sia la loro cardinalità, ovvero conosciamo le dimensioni effettive della matrice. Possiamo agire dunque su di essa per cercare la soluzione a peso massimo e ritornando alla funzione chiamante l'indice della riga della su cui è salvato tale insieme di archi. Possiamo infine rimuovere definitivamente questo insieme dal grafo.

A fianco un estratto della funzione per la ricerca dell'insieme a peso massimo.

```
int best_DAG(...){
    int b_id=0, b_sum=0, sum=0, i,j;
    for(i=0;i<find;i++){
        sum=0;
        for(j=0;j<cardinal;j++){
            sum += tentativi[i][j].wt;
        }
        if(sum>b_sum){
            b_sum=sum;
            b_id=i;
        }
    }
    return b_id;
}
```

Punto 3:

Il terzo punto ci chiede di calcolare i cammini massimi da ogni vertice verso ogni altro. Per arrivare alla soluzione ho deciso di applicare la relaxation inversa.

In quanto siamo arrivati ad ottenere un DAG grazie ai punti precedenti posso proseguire ordinando topologicamente il DAG e poi applicando la relaxation sui vertici seguendo il loro ordine topologico.

```
static int *TS_dag(Graph G){
    int v, time = 0, *pre, *ts;
    pre = malloc(G->V*sizeof(int));
    ts = malloc(G->V*sizeof(int));
    for (v=0; v < G->V; v++){
        pre[v] = -1; ts[v] = -1;
    }
    for (v=0; v < G->V; v++){
        if (pre[v]== -1) TS_dfsR(...);
    }
    return ts;
}

static void TS_dfsR(...){
    link t;
    pre[v] = 0;
    for(t=G->ladj[v];t!=G->z;t=t->next){
        if(pre[t->v]==-1) TS_dfsR(...);
    }
    ts[(*time)++] = v;
}
```

```
static void relaxation(int *ts, Graph G, int id){
    int i,v; link t;
    int *fn = malloc(G->V*sizeof(int));
    int *max_d = malloc(G->V * sizeof(int));
    for(i=0;i<G->V;i++){
        fn[i]=-1;
        max_d[i]=-1;
    }
    fn[id]=id; max_d[id]=0;
        for(i=1;i<=G->V;i++){
        v=ts[(G->V)-i];
        if(fn[v]!=-1){
            for(t=G->ladj[v];t!=G->z;t=t->next){
                if(max_d[v]+t->wt > max_d[t->v]){
                    max_d[t->v]= max_d[v]+t->wt;
                    fn[t->v]=v;
                }
            }
```