

“Lose Weight” project

Libreria per il progetto sui grafi sviluppata dal gruppo 11 composto da Francesco Borrelli, Alessandro Grieco e Camilla Zampella durante il corso di Laboratorio di Algoritmi e Strutture Dati dell’ a.a. 2017/2018.

1 SOMMARIO

2	Strutture dati utilizzate	1
2.1	Per la creazione del grafo	1
2.2	Per la visita del grafo	2
2.3	Strutture di supporto	2
2.4	Variabile d’errore	3
3	Ricerca del percorso minimo	3
3.1	Percorso minimo in un grafo privo di nodi alla stessa altezza	4
3.2	Dettagli implementativi	4
3.2.1	Funzione uphillVisit	4
3.2.2	Funzione DFSVisitUphillList	4
3.2.3	Funzione printPath	5
3.2.4	Funzione pathGenerator	5
3.2.5	Funzione pathExtender	5
3.2.6	Funzione initializeVisit:	6
3.3	Ricerca del percorso minimo in un grafo avente due nodi alla stessa altezza	6
3.4	Dettagli implementativi	6
3.4.1	Algoritmo di Dijkstra	6
3.5	Algoritmo loseWeightPathPrinter	6
4	Manuale utente	7

2 STRUTTURE DATI UTILIZZATE

2.1 PER LA CREAZIONE DEL GRAFO

Per creare il grafo sono state utilizzate le seguenti tre strutture:

“Lose Weight” project

```
typedef struct edge{
    int k;
    float weight;
    struct edge* next;
} edge;

typedef struct node{
    float height;
    edge* adj;
} node;

typedef struct graph{
    int n;
    node* nodes;
} graph;
```

- La struttura Graph , la quale contiene al suo interno una variabile intera che indica il numero di nodi da cui è formato il grafo e un puntatore ad un array di elementi di tipo Node.
- La struttura Node rappresentante le intersezioni stradali, la quale contiene una variabile di tipo float che indica l’elevazione dell’intersezione rispetto al livello del mare (altezza) e una lista puntata di elementi di tipo Edge.
- La struttura Edge che rappresenta i segmenti stradali, la quale contiene un intero che indica il nodo di destinazione dell’arco, una variabile float che indica la distanza in metri (peso dell’arco) dalla sorgente alla destinazione e un puntatore al successivo elemento della lista di adiacenza.

2.2 PER LA VISITA DEL GRAFO

Si è utilizzata la struttura Visit, la quale contiene un array per la memorizzazione dei colori, un array per la memorizzazione dei predecessori e un array di float per la memorizzazione delle distanze.

```
typedef struct visit{
    int* col;
    int* pred;
    float* dist;
} visit;
```

2.3 STRUTTURE DI SUPPORTO

Sono state inoltre utilizzate due strutture di supporto:

“Lose Weight” project

```
typedef struct list{
    int k;
    struct list* next;
} list;

typedef struct elem{
    float priority;
    int id;
} elem;

typedef struct{
    int size;
    int allocated;
    elem *data;
    int *pos;
}heap;
```

- Una lista puntata al fine di rappresentare una serie di nodi ordinati.
- Una coda a priorità di supporto all’algoritmo di Dijkstra per i percorsi minimi in caso in cui vi siano almeno due nodi aventi stessa altezza.

2.4 VARIABILE D’ERRORE

Al fine di poter gestire al meglio gli errori provocati da un uso incorretto delle funzioni della libreria “graph” è utilizzata la variabile intera GRAPH_ERROR. Questa variabile può assumere diversi valori in base ad i casi commentati nel seguente codice

```
int GRAPH_ERROR=0;
/* -----ERRORS LEGEND-----
-1 : IT'S IMPOSSIBLE TO PERFORM THE FUNCTION ON A NOT ALLOCATED GRAPH
-2 : IT'S IMPOSSIBLE TO ALLOCATE MEMORY
-4 : THERE IS AT LEAST ONE NOT VALID PARAMETER
-5 : SUPPORT DATA STRUCTURE ERROR
-6 : NO TOPOLOGICAL ORDER FOUND
*/
```

3 RICERCA DEL PERCORSO MINIMO

La traccia richiedeva di trovare il percorso minimo in un grafo tale da permettere a Matteo di percorrere la prima parte di strada in salita e la seconda parte in discesa, prima in un grafo in cui non potessero esservi due punti alla stessa altezza e poi in un grafo in cui possano esservi nodi alla stessa altezza. Per risolvere i punti del problema sono state proposte le seguenti soluzioni.

3.1 PERCORSO MINIMO IN UN GRAFO PRIVO DI NODI ALLA STESSA ALTEZZA

Per risolvere il primo punto della traccia si è suddiviso il problema in due sottoproblemi: inizialmente si cerca il percorso minimo in salita partendo dalla sorgente, poi si cerca il percorso minimo in salita partendo dalla destinazione e infine si uniscono le due visite.

Poiché ogni visita genera un sottografo orientato aciclico (perché la visita avviene solo in salita), è possibile generare un ordinamento topologico per entrambe le visite. Rilassando gli archi seguendo l'ordinamento si garantisce che ogni volta che un nodo dell'ordinamento viene analizzato è sicuro che sono già stati rilassati tutti i percorsi che portano a lui e che per tanto la sua distanza minima sarà quella corretta.

3.2 DETTAGLI IMPLEMENTATIVI

Di seguito l'implementazione che si adatta alla strategia di risoluzione del problema.

3.2.1 Funzione uphillVisit

È la funzione che sfrutta l'ordinamento topologico per settare le distanze tra i nodi all'interno del vettore dist della struct visit affinché calcoli il percorso minimo.

Elenco dei parametri:

1. g: grafo su cui devo eseguire la visita
2. s: nodo sorgente da cui deve partire la visita

Ritorna un riferimento alla struttura visit.

Pre condizione: la sorgente sia un nodo contenuto all'interno del grafo.

Post condizione: viene ritornato un riferimento alla struttura visit all'interno della quale i dati immessi nei campi rispecchino la visita su descritta.

Logica dell'algoritmo:

Chiama la funzione DFSVisitUphillList per creare l'ordinamento topologico. Successivamente setta le distanze nel vettore delle distanze controllando quali archi forniscono il percorso minimo.

Complessità in termini di tempo : $O(V + E)$ con V numero di vertici del grafo ed E numero degli archi del grafo

Complessità in termini di spazio: $O(V)$ Utilizza un riferimento alla struttura visit e una lista puntata

3.2.2 Funzione DFSVisitUphillList

Funzione che crea l'ordinamento topologico del sottografo al fine di ottenere il percorso minimo

Elenco dei parametri:

1. g: grafo di cui si deve fare l'ordinamento topologico
2. v: riferimento della struttura visit
3. l: riferimento ad una lista puntata
4. s: nodo sorgente da cui parte la visita

“Lose Weight” project

Ritorna una lista contenente l'ordinamento topologico del sottografo.

Pre condizione: la sorgente deve essere un nodo contenuto all'interno del grafo

Post condizione: all'interno di l sarà contenuto l'ordinamento topologico. Se l'ordinamento topologico non esiste, la variabile `GRAPH_ERROR` assumerà il valore -6

Complessità in termini di tempo: $O(V + E)$

Complessità in termini di spazio: $O(V)$

3.2.3 Funzione `printPath`

Funzione che serve a stampare un percorso dopo una visita

Elenco dei parametri:

1. `v`: riferimento della struttura `visit`
2. `s`: nodo sorgente

Complessità in termini di tempo: $O(V)$

3.2.4 Funzione `pathGenerator`

Funzione che si occupa di generare un percorso sotto forma di lista puntata a partire da una visita precedentemente eseguita.

Elenco dei parametri:

1. `g`: grafo su cui si è effettuata la visita
2. `v`: struttura `visit` restituita da una visita precedentemente effettuata
3. `s`: nodo sorgente da cui parte la visita

Complessità in termini di tempo: $O(V)$

Complessità in termini di spazio: $O(V)$

3.2.5 Funzione `pathExtender`

Funzione che si occupa di estendere un percorso precedentemente generato da una visita

Elenco dei parametri:

1. `g`: grafo su cui si è effettuata la visita
2. `path`: lista puntata contenente il percorso precedentemente generato da una visita
3. `v`: riferimento della struttura `visit` contenente i dati ottenuti da una visita precedentemente effettuata
4. `s`: sorgente da cui ha origine la visita

Complessità in termini di tempo: $O(V)$

Complessità in termini di spazio: $O(V)$

“Lose Weight” project

3.2.6 Funzione initializeVisit:

Funzione che si occupa di allocare e inizializzare gli elementi all'interno della struttura visit prima di effettuare una visita del grafo.

Elenco dei parametri:

1. g: grafo su cui verrà eseguita la visita

Post condizione: restituisce un riferimento alla struttura visit inizializzata.

Complessità in termini di tempo : $O(V)$

Complessità in termini di spazio: $O(V)$

3.3 RICERCA DEL PERCORSO MINIMO IN UN GRAFO AVENTE DUE NODI ALLA STESSA ALTEZZA

Per risolvere il secondo punto del problema, è possibile utilizzare l'algoritmo precedente nel caso in cui non vi siano cicli con nodi aventi altezza uguale dato che si può utilizzare l'ordinamento topologico per visitare il grafo. Altrimenti, se nel grafo è presente un ciclo i cui vertici hanno la stessa altezza è necessario utilizzare Dijkstra per poter trovare il percorso minimo.

3.4 DETTAGLI IMPLEMENTATIVI

3.4.1 Algoritmo di Dijkstra

Funzione che consente di trovare il percorso minimo da una sorgente ai nodi in un grafo pesato

Elenco dei parametri:

1. g: grafo su cui viene effettuata la visita
2. s: sorgente da cui parte la visita

L'algoritmo si serve dell'utilizzo di code a priorità al fine di rendere l'estrazione del peso minimo un'operazione costante.

Complessità in termini di tempo: $O(V \log_2(V) + E)$

Complessità in termini di spazio: è necessario l'utilizzo di code a priorità

3.5 ALGORITMO LOSEWEIGHTPATHPRINTER

La funzione si occupa di determinare se esiste un percorso minimo che rispetti i vincoli della traccia (utilizzando la strategia migliore) e di stamparlo. Attraverso la ricerca dell'ordinamento topologico è possibile determinare quale visita effettuare sul grafo.

Elenco dei parametri:

1. g: grafo su cui viene effettuata la visita

“Lose Weight” project

2. s: sorgente da cui parte la visita
3. d: nodo di destinazione della visita

Logica dell’algoritmo:

Inizialmente l’algoritmo richiama la funzione uphillVisit per controllare se è possibile generare un ordinamento topologico da sorgente e destinazione. Se uno degli ordinamenti topologici non è generabile, effettua le visite con l’algoritmo di Dijkstra. Successivamente controlla che vi sia un percorso minimo che raggiunge la sorgente e la destinazione sommando le distanze dei nodi raggiunti da entrambe le visite (viene presa in considerazione la somma minima delle distanze), e infine lo stampa, altrimenti stampa un messaggio di errore.

Complessità in termini di tempo:

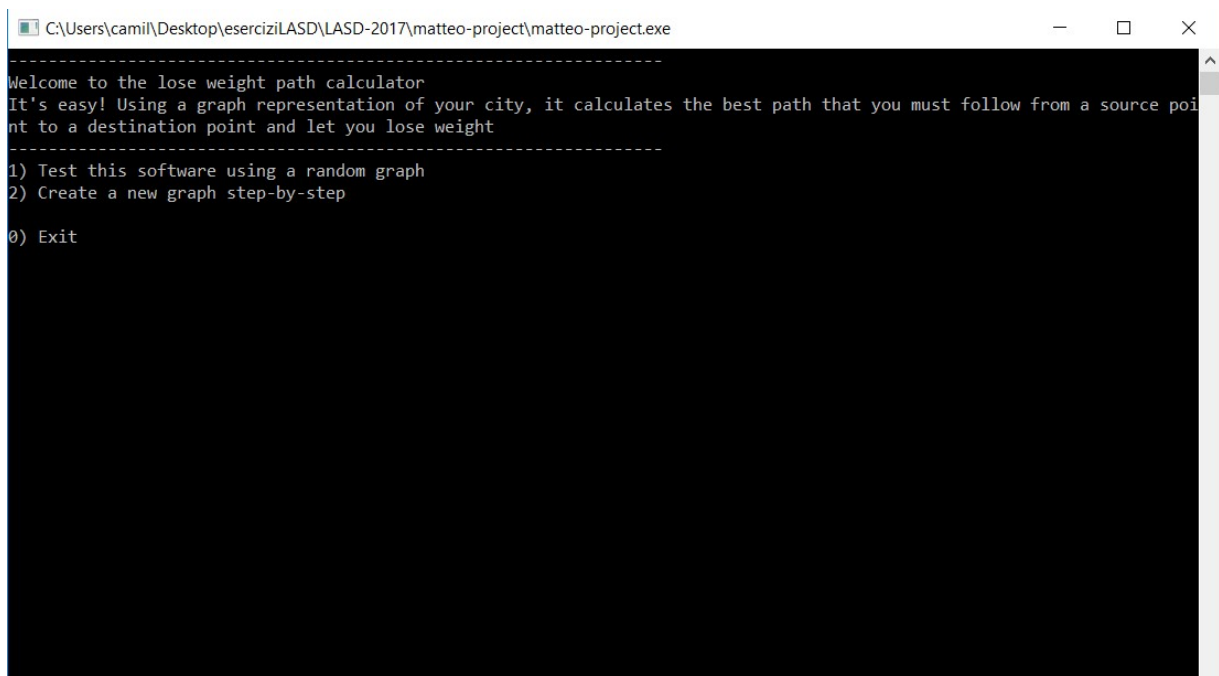
Caso di grafo privo di nodi aventi stessa altezza: $O(V + E)$

Caso di grafo con nodi aventi la stessa altezza in un ciclo: $O(V \log_2(V) + E)$

Complessità in termini di spazio: $O(V)$

4 MANUALE UTENTE

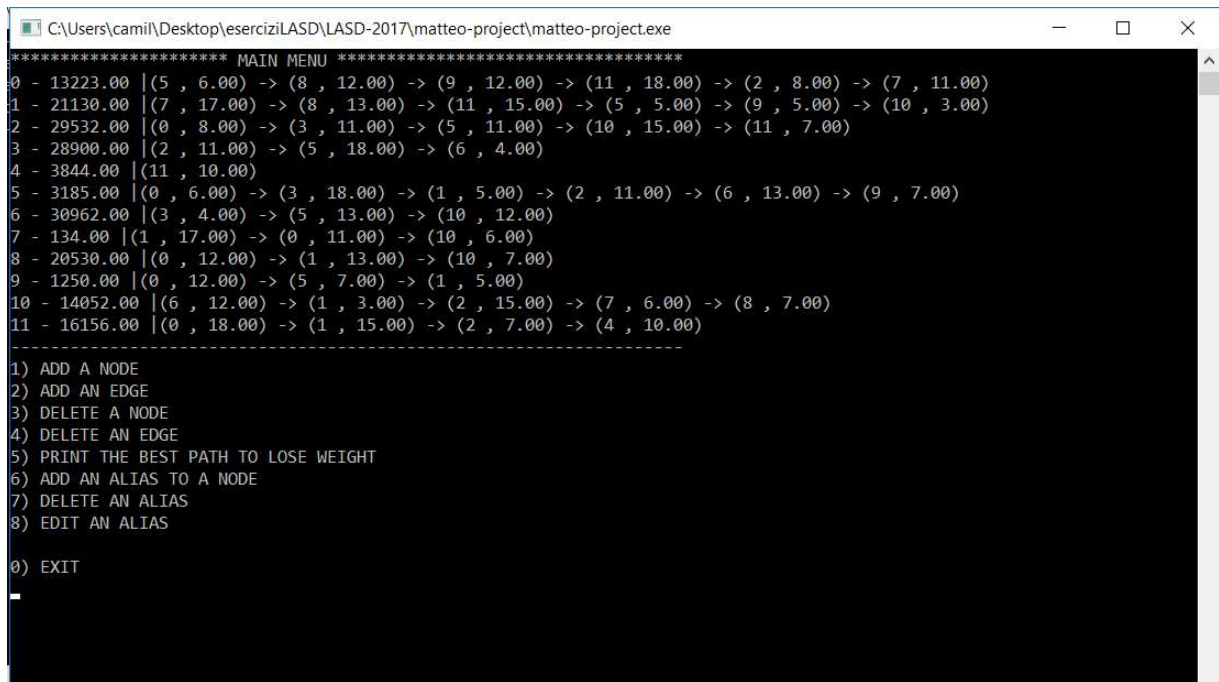
Il software si apre presentando un menù iniziale da cui è possibile scegliere se generare il grafo in maniera randomica o se invece crearne uno immettendo valori scelti dall’utente.



```
C:\Users\camil\Desktop\eserciziLASD\LASD-2017\matteo-project\matteo-project.exe
-----
Welcome to the lose weight path calculator
It's easy! Using a graph representation of your city, it calculates the best path that you must follow from a source point to a destination point and let you lose weight
-----
1) Test this software using a random graph
2) Create a new graph step-by-step
0) Exit
```

“Lose Weight” project

Una volta selezionata una delle due voci, si aprirà un secondo menù da cui sarà possibile selezionare quale operazione si vuole compiere sul grafo precedentemente creato.



```
C:\Users\camil\Desktop\eserciziLASD\LASD-2017\matteo-project\matteo-project.exe
***** MAIN MENU *****
0 - 13223.00 |(5 , 6.00) -> (8 , 12.00) -> (9 , 12.00) -> (11 , 18.00) -> (2 , 8.00) -> (7 , 11.00)
1 - 21130.00 |(7 , 17.00) -> (8 , 13.00) -> (11 , 15.00) -> (5 , 5.00) -> (9 , 5.00) -> (10 , 3.00)
2 - 29532.00 |(0 , 8.00) -> (3 , 11.00) -> (5 , 11.00) -> (10 , 15.00) -> (11 , 7.00)
3 - 28900.00 |(2 , 11.00) -> (5 , 18.00) -> (6 , 4.00)
4 - 3844.00 |(11 , 10.00)
5 - 3185.00 |(0 , 6.00) -> (3 , 18.00) -> (1 , 5.00) -> (2 , 11.00) -> (6 , 13.00) -> (9 , 7.00)
6 - 30962.00 |(3 , 4.00) -> (5 , 13.00) -> (10 , 12.00)
7 - 134.00 |(1 , 17.00) -> (0 , 11.00) -> (10 , 6.00)
8 - 20530.00 |(0 , 12.00) -> (1 , 13.00) -> (10 , 7.00)
9 - 1250.00 |(0 , 12.00) -> (5 , 7.00) -> (1 , 5.00)
10 - 14052.00 |(6 , 12.00) -> (1 , 3.00) -> (2 , 15.00) -> (7 , 6.00) -> (8 , 7.00)
11 - 16156.00 |(0 , 18.00) -> (1 , 15.00) -> (2 , 7.00) -> (4 , 10.00)
-----
1) ADD A NODE
2) ADD AN EDGE
3) DELETE A NODE
4) DELETE AN EDGE
5) PRINT THE BEST PATH TO LOSE WEIGHT
6) ADD AN ALIAS TO A NODE
7) DELETE AN ALIAS
8) EDIT AN ALIAS
0) EXIT
_
```

Oltre le operazioni classiche quali inserire e cancellare nodi e archi, è stata data la possibilità di cercare il percorso minimo tra due punti che rispetti la proprietà di essere parte in salita e parte in discesa.

Il comando per eseguire la compilazione del progetto è ***gcc graph.c queue.c list.c utils.c main.c***