“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.

Sommario

[2 Strutture dati utilizzate 1](#_Toc484961934)

[2.1 Per la creazione del grafo 1](#_Toc484961935)

[2.2 per la visita del grafo 2](#_Toc484961936)

[2.3 strutture di supporto 2](#_Toc484961937)

[3 ricerca del percorso minimo 2](#_Toc484961938)

[3.1 percorso minimo in un grafo privo di nodi alla stessa altezza 2](#_Toc484961939)

[3.2 Dettagli implementativi 2](#_Toc484961940)

[3.2.1 Funzione uphillVisit 3](#_Toc484961941)

[3.2.2 Funzione DFSVisitUphillList 3](#_Toc484961942)

[3.2.3 Funzione printPath 3](#_Toc484961943)

[3.2.4 Funzione pathGenerator 4](#_Toc484961944)

[3.2.5 Funzione pathExtender 4](#_Toc484961945)

[3.2.6 Funzione initializeVisit: 4](#_Toc484961946)

[3.2.7 Funzione hasDuplicates 4](#_Toc484961947)

[3.3 ricerca del percorso minimo in un grafo avente due nodi alla stessa altezza 5](#_Toc484961948)

[3.4 dettagli implementativi 5](#_Toc484961949)

[3.4.1 Algoritmo di Dijkstra 5](#_Toc484961950)

[3.5 algoritmo loseWeightPathPrinter 5](#_Toc484961951)

[4 manuale utente 6](#_Toc484961952)

# Strutture dati utilizzate

## Per la creazione del grafo

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

* 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 alla struttura Node.
* La struttura Node, la quale contiene una variabile float che indica l’altezza del nodo e un puntatore alla struttura edge.
* La struttura Edge, la quale contiene un intero che indica il nodo di destinazione dell’arco, una variabile float che indica il peso dell’arco e un puntatore al successivo elemento della lista di adiacenza.

## 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.

## strutture di supporto

Sono state inoltre utilizzate due strutture di supporto:

* 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 due nodi aventi stessa altezza.

# 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.

## 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 è possibile generare un ordinamento topologico per ogni sottografo e settare la distanza minima dalla sorgente procedendo per topologia.

## Dettagli implementativi

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

### 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.

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 : nel vettore dist di Visit sarà contenuto il percorso minimo

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)

Complessità in termini di spazio : Utilizza un riferimento alla struttura visit e una lista puntata

### 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

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.

Complessità in termini di tempo: O(v+e)

Complessità in termini di spazio: utilizza una lista puntata

### 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)  
Complessità in termini di spazio: Non utilizza strutture aggiuntive

### 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. s: nodo sorgente da cui parte la visita

Complessità in termini di tempo: O(v)  
Complessità in termini di spazio: Utilizza una lista puntata

### 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: Non utilizza strutture aggiuntive

### 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

### Funzione hasDuplicates

Funzione che controlla se nel grafo vi siano due nodi aventi la stessa altezza.

Elenco dei parametri:

1. g: grafo sui cui bisogna controllare che non vi siano nodi aventi la stessa altezza

Post condizione: restituisce un intero che sarà uguale a 0 se la funzione non contiene duplicati e 1 altrimenti.

Complessità in termini di tempo: O(V)

## ricerca del percorso minimo in un grafo avente due nodi alla stessa altezza

Per risolvere il secondo punto del problema invece, poiché è possibile che vi siano due nodi aventi la stessa altezza, non è stato possibile utilizzare l’ordinamento topologico per visitare il grafo, si è quindi utilizzato l’algoritmo di Dijkstra per la ricerca dei percorsi minimi utilizzando come sorgente prima casa di Matteo e poi la destinazione, procedendo successivamente ad unire le due visite per trovare il percorso minimo da percorrere.

## dettagli implementativi

### 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(vlog(v)+e)

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

## algoritmo loseWeightPathPrinter

La funzione si occupa di determinare se il grafo contenga nodi aventi la stessa altezza e quindi di utilizzare il giusto algoritmo di visita. Si occuperà successivamente di unire i risultati ottenuti dalla visita che ha come origine la sorgente e da quella che ha come origine la destinazione. Restituisce infine il percorso che Matteo dovrà percorrere per arrivare in università e un messaggio di errore nel caso in cui nessun percorso del grafo rispetti le proprietà richieste.

Elenco dei parametri:

1. g: grafo su cui viene effettuata la visita
2. s: sorgente da cui parte la visita
3. d:nodo di destinazione della visita

Logica dell’algoritmo:  
Inizialmente l’algoritmo richiama la funzione HasDulicates per controllare se nel grafo vi siano nodi aventi la stessa altezza, se la funzione restituisce 1 allora chiama la funzione di Dijkstra dando come sorgente prima s e poi d. Se invece la funzione restituisce 0 allora chiama la funzione che effettua la visita utilizzando l’ordinamento topologico, dando come sorgente prima s e poi d.

Successivamente controlla che vi sia un percorso che raggiunge la sorgente e la destinazione, e se esso rispetta le proprietà richieste lo stampa, altrimenti stampa un messaggio di errore.

Complessità in termini di tempo:

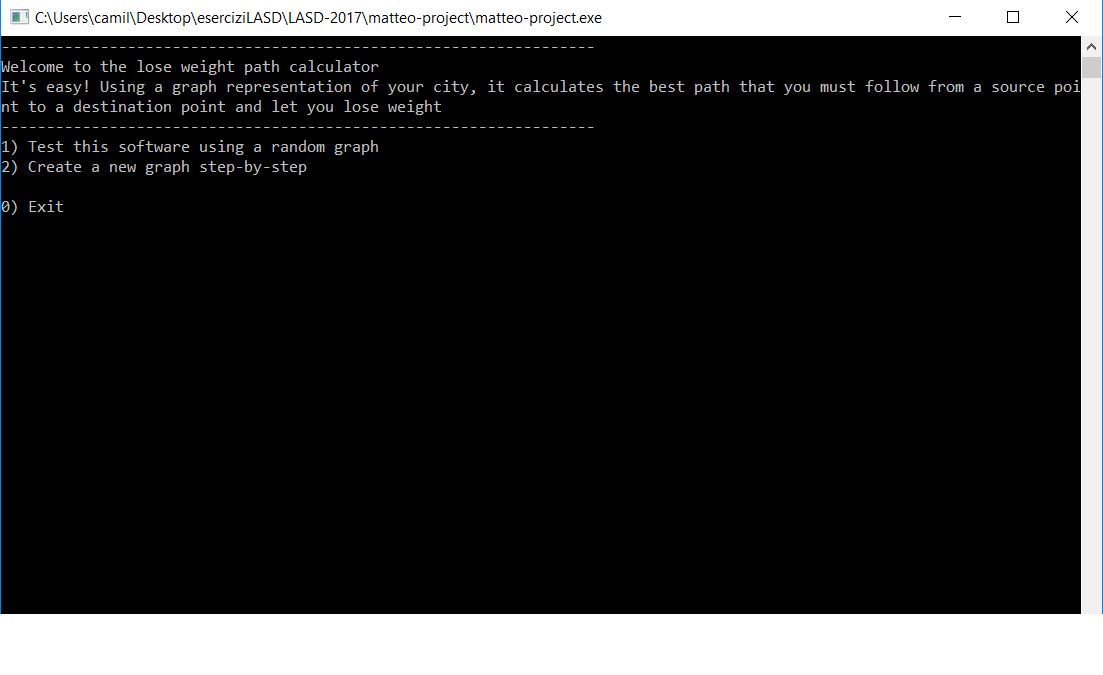
Caso di grafo privo di nodi aventi stessa altezza: O(v+e)

Caso di grafo con nodi aventi la stessa altezza: O(vlog(v)+e)

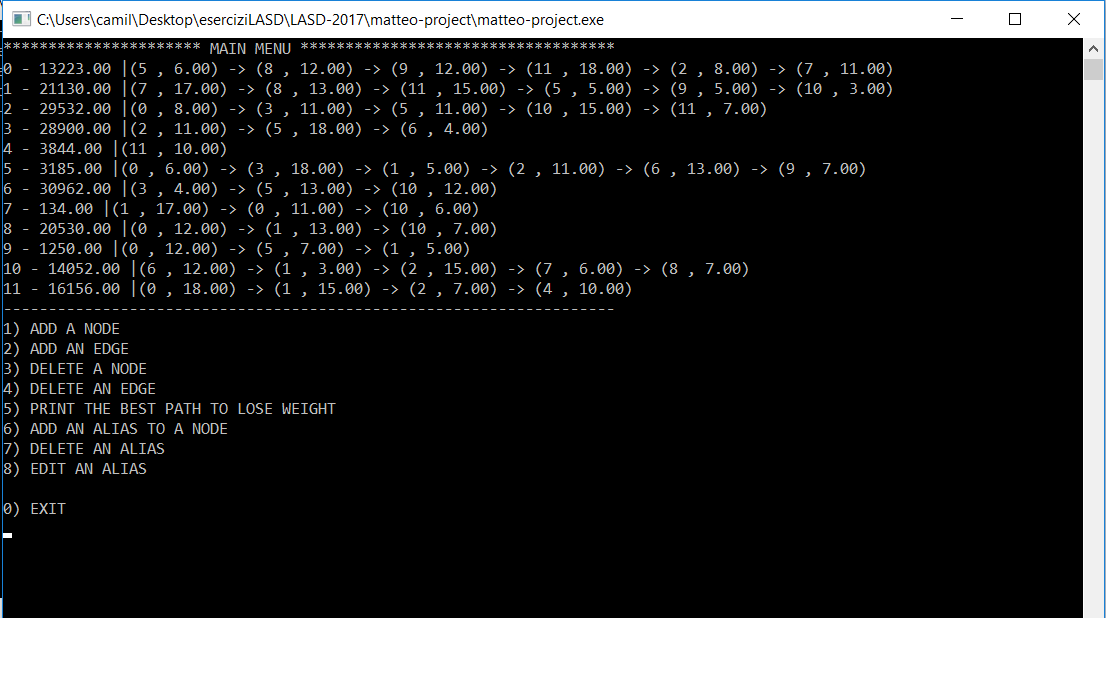
Complessità in termini di spazio:

# 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.



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.



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, e la possibilità di aggiungere un alias ai nodi.

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