Traccia 1

Quesito 2

Descrizione del problema:

Il problema posto dal quesito 2 è quello di calcolare il percorso che massimizza la soddisfazione dei turisti che visitano l'arcipelago di Grapha-Nui , partendo da una generica isola verso tutte le altre isole.

L'intero arcipelago si può assimilare ad un grafo le cui isole ed i collegamenti fra queste sono rappresentati dai nodi e dagli archi appartenenti al grafo.

Per quanto riguarda le caratteristiche del grafo si evince che questo è orientato, aciclico e pesato, dove il peso (che può essere anche negativo) indica il grado di soddisfazione del turista che percorre il collegamento fra due isole.

Il risultato consiste nella ricerca di un cammino massimo all'interno di un grafo, ossia la sequenza di nodi la cui somma dei pesi degli archi è massima.

Da evidenziare che essendo il grafo aciclico, il problema ha una sottostruttura ottima, di conseguenza non è NP-hard.

Possiamo affermare che, dato un grafo orientato, pesato e aciclico G(V,E) ed una funzione peso $w:E \rightarrow \mathbb{R}$, posto $p = (v_1,\dots,v_h)$ un cammino massimo dal vertice v_1 al vertice v_h, $\forall i,j$ tale che $1 \leq i \leq j \leq h$, considerato $p_{i,j}$ un sottocammino di p tra v_i e v_j, allora $p_{i,j}$ è un cammino massimo tra v_i e v_j .

Per dimostrare questo supponiamo che $p_{i,j}$ sia un cammino massimo di un grafo tra il vertice v_i e v_j. Spezziamo $p_{i,j}$ in due sottocammini $p_{i,k}$ e $p_{k+1,j}$ compresi rispettivamente tra i vertici v_i e v_k e v_{k+1} e v_j . Per la proprietà della sottostruttura ottima, se $p_{i,j}$ è un cammino massimo allora $p_{i,k}$ e $p_{k+1,j}$ devono essere a loro volta cammini massimi compresi tra i rispettivi vertici. Ragionando per assurdo neghiamo che $p_{i,k}$ non sia il cammino massimo compreso tra i vertici v_i e v_k , per cui esisterà un altro cammino $*p_{i,k}$ esistente tra questi due vertici, che abbia un peso totale maggiore di $p_{i,k}$. Detto ciò andando ad aggiungere $*p_{i,k}$ a $p_{k+1,j}$ verrà creato un nuovo cammino massimo $*p_{i,j}$ tale che questo abbia un peso maggiore di $p_{i,j}$, ma ciò ci pone davanti ad un assurdo perché contraddice l'assunzione fatta in precedenza sul fatto che $p_{i,j}$ sia il cammino massimo del grafo in questione, l'assurdo deriva dalla negazione dell'ipotesi. Questo implica che i due sottocammini $p_{i,k}$ e $p_{k+1,j}$ del cammino massimo di un grafo sono cammini massimi tra i vertici v_i e v_k e v_{k+1} e v_j, dimostrando quindi che il problema in questione gode della proprietà della sottostruttura ottima.

Descrizione struttura dati:

La struttura dati usata è un grafo G, ovvero una coppia ordinata G=(V,E) di insiemi, con V insieme dei nodi ed E insieme degli archi, tali che gli elementi di E siano coppie di elementi di (w,v) di nodi (w,v ∈ V), per cui

si può affermare che E ⊆ V x V. All'interno di un grafo diretto o orientato gli archi sono coppie ordinate (v,w) di nodi mentre all'interno di un grafo indiretto o non orientato gli archi sono coppie non ordinate (v,w) di nodi.

La traccia afferma che in input vengono specificati la direzione ed un valore (anche negativo) che misura la qualità del collegamento per ogni isola, da ciò si deduce che il grafo in questione è un grafo orientato e pesato.

Un grafo è pesato quando esiste attraverso una funzione peso w:E→ $\mathbb{R}$ che associa ad ogni arco un peso, rappresentato da un numero reale.

La traccia indica che il grafo in cui si andrà ad operare è aciclico.

Il metodo di rappresentazione scelto per il grafo è quello delle liste di adiacenza. Delle operazioni "canoniche" fatte sui grafi per risolvere il problema viene applicata la DFS per realizzare l'ordinamento topologico, cosa possibile visto che il grafo che stiamo trattando è un DAG(Grafo diretto e aciclico), alla quale viene aggiunta una variante di Bellman-Ford, ovvero l'algoritmo usato per calcolare i cammini minimi all'interno di un grafo.

Entrambe le funzioni con le relative varianti verranno spiegate successivamente.

Formato dati in input/output:

L'input del programma è rappresentato da un file di testo contenente nella prima riga due interi separati da uno spazio ossia il numero N delle isole (numerate da 0 ad N-1) ed il numero P dei ponti.

Le successive P righe contengono ciascuna tre numeri I1, I2 e Q dove Q è la qualità del collegamento dall'isola I1 a I2.

Su questi input vengono fatte due assunzioni:

- $2 \leq N \leq 1000$
- $1 \leq P \leq 10000$

Gli altri input sono costituiti dalle decisioni immesse da tastiera da parte dell'utente.

L'output del programma è rappresentato da un grafo diretto e aciclico, all'interno del quale può essere ricavato il percorso massimo da un nodo *s* sorgente scelto dall'utente a tutti gli altri nodi *u* all'interno del grafo.

Verrà stampato a video la richiesta del programma nei confronti dell'utente di scegliere uno tra gli N nodi come sorgente *s* e verrà stampato il peso totale del percorso massimo tra *s* e tutti gli altri nodi *u* del grafo.

Descrizione algoritmo:

Il problema ha una sottostruttura ottima la quale ci garantisce che, calcolato il cammino massimo tra la sorgente *s* ed un nodo *u* allora i cammini calcolati *s* ed i predecessori di *u* saranno a loro volta massimi.

Arrivati al calcolo del cammino massimo di *u*, dobbiamo assicurarci di aver analizzato e determinato il percorso massimo di tutti i predecessori di *u*.

Dobbiamo garantire che calcolato il valore massimo del percorso di un nodo *u*, questo non cambi successivamente.

Per assicurarci ciò, visto che il grafo considerato è un DAG, possiamo realizzare l'ordinamento topologico sul grafo.

Un ordinamento topologico di DAG G=(V,E) è un ordinamento lineare di tutti i suoi vertici tale che se G contiene un arco *(u,v)* allora *u* appare prima di *v* nell'ordinamento. In questo modo arrivati al nodo *v*, avremo calcolato tutti i percorsi che portano da un nodo sorgente *s* a *v*, e scegliendo quello con valore massimo saremo sicuri che successivamente questo non cambierà.

Per effettuare l'ordinamento topologico viene adoperata una DFS, ovvero una visita in profondità, nella quale utilizzeremo uno stack alla fine della DFS_visit per memorizzare l'ordine di precedenza fra i nodi.

I parametri π (padre), d (inizio visita) e f (fine visita) del generico nodo *v* analizzato non vengono definiti nella DFS, dato che non servono, tuttavia saranno valorizzati durante il calcolo del cammino massimo.

Una volta completato l'ordinamento topologico e riempito lo stack, si estraggono tutti i nodi.

Ad ogni nodo estratto si passa a scorre la sua lista di adiacenze, ovvero tutti i nodi al quale il nodo è collegato, e si procede a rilassarli migliorando il cammino massimo del nodo considerato trovato fino a quel punto, passando per un nodo v della sua lista di adiacenza ed aggiornando in questo caso π e *d*.

In questo modo verrà assegnato ad ogni nodo il valore massimo del cammino per arrivare da *s* a quel nodo e si determinerà il padre di questo in modo che risalendo la sequenza dei padri fino ad *s* si possa ottenere la sequenza di nodi facente parte di quel cammino massimo.

Di seguito lo pseudo-codice rappresentante l'algoritmo con le relative spiegazioni.

1. **DFS (G)**
2. **S ← ∅ //viene allocato lo stack che conterrà l'ordinamento topologico**
3. **for each u ∈ V[G]**
4. **color[u] ← WHITE**
5. **for each u ∈ V[G]**

```
6.      if color[u] = WHITE
7.          DFS_visit(u, S)
8.  return S


9.  DFS_visit(u, S)
10. color[u] ← GRAY
11. for each v ∈ adj[u]
12.     if color[v] = WHITE
13.         DFS_visit(v, S)
14. color[u] ← BLACK
15. push(S, u)

16. MaxPathWay(s, G)
17. Inizialized_single_source(s, G)
18. S ← DFS(G)
19. while S ≠ ∅
20.     u = top[S]
21.     pop(S)
22.     for each v ∈ adj[u]
23.         relax(u, v)

24. Inizialized_single_source(s, G)
25. for each u ∈ V[G]
26.      d[u] ← -∞
27.      π[u] ← NIL
28. d[s] ← 0
29. π[u] ← NIL

30. relax(u, v)
31. if d[v] < w(u+v) + d[u]
32.     d[v] ← w(u+v) + d[u]
33.     π[v] ← u
```

Dalla riga 1 alla riga 8 viene illustrato il funzionamento della DFS ovvero della visita in ampiezza, la quale riceve come input il grafo preso in questione.

Alla riga 2 viene dichiarato lo stack S che conterrà i nodi ordinati in modo lineare.

All'interno del for compreso nelle righe 3 e 4 il colore di tutti i nodi del grafo viene inizializzato a WHITE, per indicare che i nodi non sono stati ancora visitati.

Tra la riga 5 e la riga 7 si scorrono nuovamente tutti i nodi facenti del grafo, su quelli che hanno ancora colore WHITE viene effettuata la DFS_visit funzione alla quale viene passato anche lo stack S.

Alla riga 8 viene ritornato lo stack contenente i nodi provenienti dall'ordinamento topologico.

Dalla riga 9 alla 15 viene illustrato il meccanismo della DFS_visit, la quale riceve in input il nodo da visitare u e lo stack che registrerà l'ordine di precedenza fra i nodi.

Alla riga 10 viene assegnato al nodo u il colore GRAY che denoterà i nodi la cui visita è iniziata.

Tra la riga 11 e la riga 13 si scorre la lista di adiacenze del nodo u e sui nodi v che hanno ancora colore WHITE viene effettuata la DFS_visit funzione alla quale viene passato anche lo stack S, questo per visitare in profondità il grafo.

Alla riga 14 e 15 viene assegnato ad u il colore BLACK cosa che sta a significare che la visita di quel nodo è terminata, infine il nodo viene inserito all'interno di S.

Dalla riga 16 alla 23 viene illustrata la procedura MaxPathWay la quale prende come input il nodo sorgente s e il grafo G. Questa procedura dopo aver ricevuto lo stack contenente i nodi ordinati linearmente, calcolerà il cammino massimo tra s e tutti gli altri nodi del grafo.

Alla riga 17 viene richiamata la funzione inizialized_single_source la quale inizializzerà il parametro π e d di tutti i nodi del grafo meno la radice a -∞ e d a NIL, mentre al parametro π e d della radice viene assegnato rispettivamente NIL e 0.

Alla riga 18 viene dichiarato un nuovo stack S al quale viene assegnato il valore dello stack uscente dalla DFS.

Dalla riga 19 alla riga 23 si entra nel while dal quale si uscirà una volta che lo stack diventerà vuoto, dopodiché si estrae il nodo in cima allo stack che viene subito decrementato.

Nelle ultime due righe vengono analizzati tutti i nodi v della lista di adiacenze del nodo u appena estratto dallo stack, i quali vengono entrambi passati alla funzione relax che migliorerà aumentando se possibile la stima del cammino massimo tra *v* ed *u*. Usciti da tale funzione avremo calcolato completamente il cammino massimo da *s* ad ogni altro nodo del grafo.

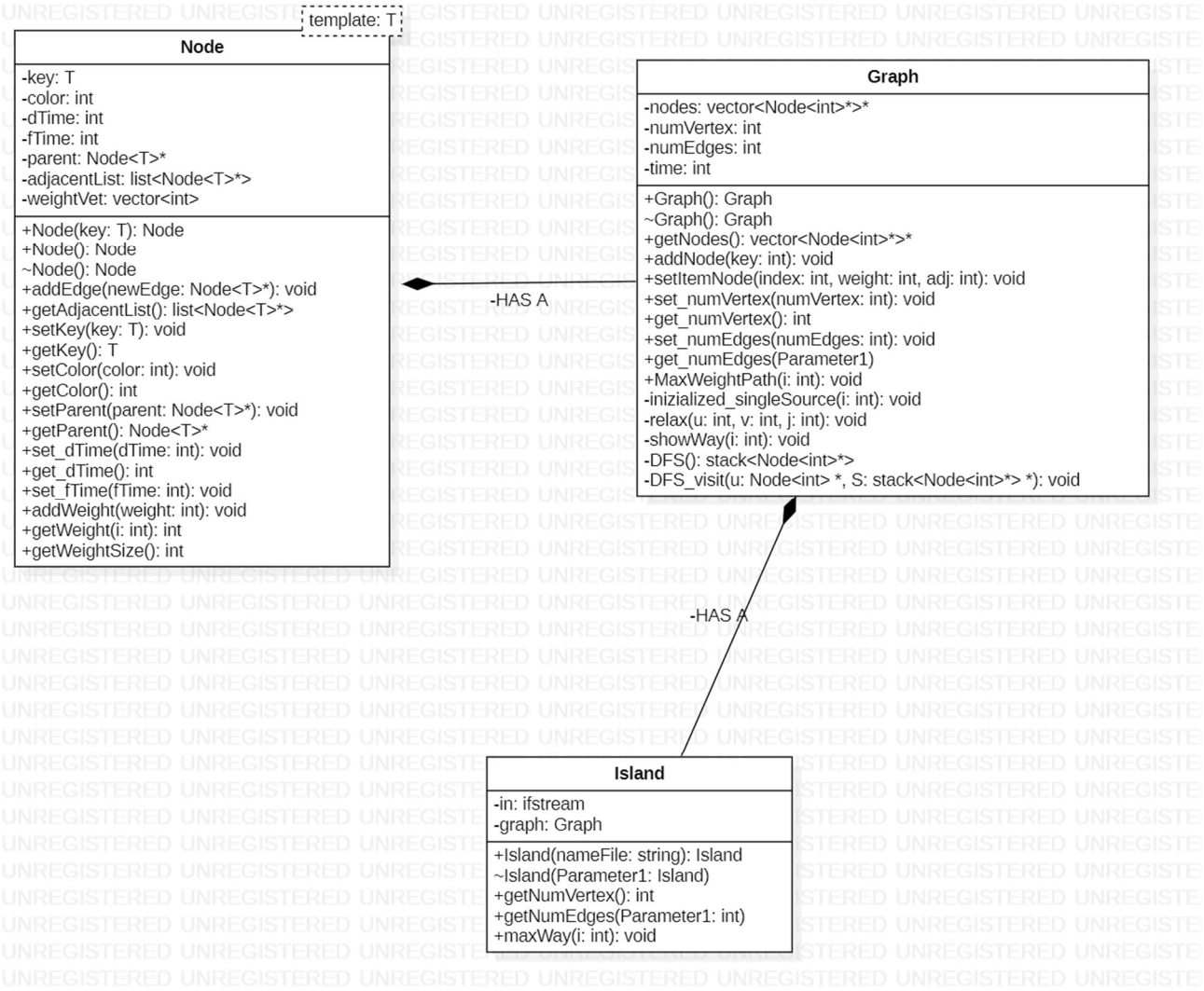
Dalla riga 24 alla 29 la funzione inizialized_single_source che prende in input il grafo e la sorgente inizializza tutti i nodi del grafo. Attraverso un ciclo for che scorre tutti i nodi del grafo ai quali assegna al loro parametro π e d rispettivamente NIL e -∞. Al parametro π e d della sorgente viene assegnato NIL e 0.

Dalla riga 30 alla 33 viene illustrato lo pseudo-codice della funzione relax.

Questa funzione prende in input il nodo *u* appena estratto dallo stack e uno dei nodi appartenente alla sua lista di adiacenza *v*.

Il compito di questa funzione è quella di aggiornare la stima del cammino massimo di *v* che viene salvata in *d*. Se d[v] ovvero la stima del cammino massimo da *s* a *v* presente fino a quel momento è minore della stima di d[u] più il peso dell'arco tra *u* e *v*, *w(u, v),* allora d[v] ← w(u, v) + d[u] e viene salvato come padre di v, π[v], u. Una volta calcolato il massimo cammino del nodo u grazie alla proprietà dell'ordinamento topologico sappiamo che sarà il valore definitivo attribuito al parametro d[u]. Questo sta a significare che ad ogni passo della funzione relax verrà scelto l'arco del nodo il cui peso porterà ad ottenere il cammino massimo da s a quel nodo.

Class diagram:



Studio complessità:

La complessità di tempo di questo algoritmo viene determinato dalle funzioni DFS e MaxPathWay.

La funzione DFS usata per ottenere l'ordinamento topologico ha un costo θ(V+E), dove V è il numero dei vertici ed E il numero degli archi che corrisponde al numero di archi visitati. A questo va sommato il costo della MaxWeightPath che complessivamente impiega un tempo θ(V+E), derivante dal while presente dalla riga 19 fino alla 23 nel quale si va a richiamare anche la funzione relax.

In totale quindi abbiamo due θ(V+E), per cui a livello asintotico la complessità della ricerca del cammino massimo è θ(V+E).

Per quanto riguarda la complessità di spazio è definita da θ(V+E), ovvero il numero dei nodi e degli archi salvati dal grafo.

Test e risultati:

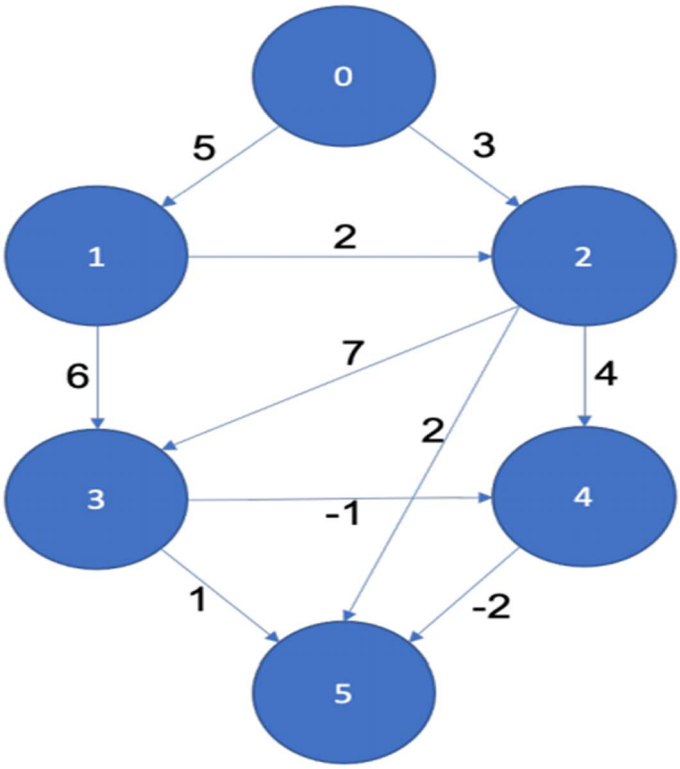
Il programma alla sua apertura avverte l'utente nel caso il file non sia stato letto correttamente.
In caso contrario il programma informerà l'utente che sono stati letti N nodi.
Successivamente all'utente viene richiesto di digitare un numero tra 0 ed N-1, il numero è il numero che indica il nodo scelto dall'utente come sorgente per calcolare il cammino massimo. Dopo che viene effettuata la scelta dell'utente il programma stampa in sequenza tutti i nodi del grafo e affianco il valore del peso del cammino massimo tra la sorgente s ed il nodo in questione. Dopo di che l'utente può scegliere di digitare 0 per uscire dal programma o qualsiasi altro carattere per visualizzare il valore del cammino massimo a partire da una sorgente diversa.
La correttezza del programma e di ciò che si è appena spiegato e illustrato dagli esempi di seguito.

```
Grapha-Nui with 6 islands and 10 bridges was created from the file
Choose an island from 0 to 5 from which routes are calculated that maximize tourist satisfaction 0
Path from island 0 to others which maximaze tourist satisfaction
0 -> 0 0
0 -> 1 5
0 -> 2 7
0 -> 3 14
0 -> 4 13
0 -> 5 15

Type 0 to exit any other key to continue 1
Grapha-Nui with 6 islands and 10 bridges was created from the file
Choose an island from 0 to 5 from which routes are calculated that maximize tourist satisfaction 2
Path from island 2 to others which maximaze tourist satisfaction
2 -> 0 -infty
2 -> 1 -infty
2 -> 2 0
2 -> 3 7
2 -> 4 6
2 -> 5 8

Type 0 to exit any other key to continue 1
Grapha-Nui with 6 islands and 10 bridges was created from the file
Choose an island from 0 to 5 from which routes are calculated that maximize tourist satisfaction 1
Path from island 1 to others which maximaze tourist satisfaction
1 -> 0 -infty
1 -> 1 0
1 -> 2 2
1 -> 3 9
1 -> 4 8
1 -> 5 10

Type 0 to exit any other key to continue 0
```


0
5
3
1
2
2
7
6
4
2
3
4
-1
1
-2
5