Percorsi nell'Arcipelago

Traccia 1 – Quesito 2

0124002062
Caruso Denny

Indice

Descrizione problema

La traccia del quesito numero due introduce:

- Un arcipelago denominato Grapha-Nui
- Una regina di codesto arcipelago
- Una particolare problematica che può essere ricondotta al "problema dei cammini massimi da sorgente unica"

Informalmente, il quesito chiede di andare a calcolare i percorsi che massimizzano la soddisfazione dei turisti, partendo da una generica isola verso tutte le altre isole dell'arcipelago. Questo al fine di conoscere quali collegamenti devono essere pubblicizzati maggiormente.
Senza ombra di dubbio, possiamo vedere l'arcipelago come un grafo, le isole come i vertici del grafo e i collegamenti fra di esse come gli archi del grafo.

Inoltre, la traccia fornisce i seguenti indizi:

- L'espressione "direzione del collegamento" fa intuire che il grafo è orientato
- La qualità del collegamento è il peso del singolo arco considerato (anche negativo)
- Il grafo è aciclico

Possiamo quindi definire formalmente il problema nel seguente modo:

Sia G un grafo orientato, aciclico e pesato G = (V, E) (anche noto come DAG (Directed Acyclic Graph)), con una funzione peso $\omega: E \to \mathbb{Z}$ che associa agli archi, dei pesi sotto forma di numeri interi relativi.
Sia $\omega(p)$ il peso del cammino $p = \langle v_0, v_1, \dots, v_k \rangle$, la somma dei pesi degli archi che lo compongono. Quest'ultimo è indicato anche come segue:

$$\omega(p) = \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$$

Si definisce il peso di un cammino massimo $\delta(u, v)$ da u a v come segue:

$$\delta(u,v) = \begin{cases} \max\{\, \omega(p) : u \rightsquigarrow v \,\} & \textit{se esiste un cammino da } u \textit{ a } v \\ -\infty & \textit{negli altri casi} \end{cases}$$

Un cammino massimo dal vertice u al vertice v è definito come un cammino qualsiasi p con peso:
$\omega(p) = \delta(u, v)$.

In generale il problema dei cammini massimi da sorgente unica è il seguente:
Sia G un grafo G = (V, E), trovare un cammino massimo che va da un dato vertice sorgente $s \in V$ a ciascun vertice $v \in V$. Cioè per ogni vertice, va trovato fra tutti i cammini che partono da s quello la cui somma dei pesi degli archi coinvolti, è massima.

Il problema dei cammini massimi però nella sua formulazione generale, qualora vi fossero cicli di peso positivo, è un problema NP-hard. Nel nostro caso siccome il grafo è aciclico, allora il problema non è NP-hard.

Inoltre, è necessario notare che il problema dei cammini massimi in un DAG, presenta la sottostruttura ottima. Cioè sottocammini di cammini massimi sono cammini massimi. Formalmente:

Sia G un grafo orientato, aciclico e pesato G = (V, E) con una funzione peso $\omega: E \rightarrow \mathbb{Z}$ e sia $p = \langle v_0, v_1, \ldots, v_k \rangle$ un cammino massimo dal vertice v_0 al vertice v_k e, per qualsiasi i e j tali che $0 \leq i \leq j \leq k$, sia $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ il sottocammino di p dal vertice v_i al vertice v_j. Allora p_{ij} è un cammino massimo da v_i a v_j.

La dimostrazione segue dalla scomposizione del cammino massimo p_{ij} in due sottocammini p_{ik} e p_{kj}. Siccome p_{ij} è un cammino massimo, allora anche p_{ik} e p_{kj}, sono cammini massimi. Se supponiamo per assurdo che uno dei due non lo sia, allora esisterebbe un sottocammino che abbia un peso totale maggiore rispetto al peso del sottocammino considerato. A questo punto utilizzando la tecnica del "cut and paste", creiamo un nuovo cammino p'_{ij} dato da p'_{ik} e p_{kj} (o simmetricamente da p_{ik} e p'_{kj}).
Questo nuovo cammino avrà un costo maggiore di p_{ij} e quindi risulta essere un assurdo, perché avevamo ipotizzato p_{ij} come cammino massimo.
Di conseguenza non esiste p'_{ik} (o simmetricamente p'_{kj}), non esiste p'_{ij} e quindi p_{ik} e p_{kj} sono a loro volta cammini massimi. Il problema gode quindi della sottostruttura ottima.

La presenza della sottostruttura ottima, ci permette di risolvere il problema mediante un'applicazione dell'ordinamento topologico dei vertici del DAG. Come vedremo, l'approccio utilizzato è quello di calcolare l'ordinamento topologico dei vertici del DAG e rilassare di volta in volta gli archi uscenti da ogni vertice (il vertice è preso seguendo l'ordinamento di cui sopra).

La descrizione dell'algoritmo, l'analisi e la correttezza, viene presa in considerazione nella sezione "Descrizione Algoritmo".

Descrizione strutture dati

La struttura dati utilizzata consiste in un grafo orientato, aciclico e pesato G = (V, E) con una funzione peso $\omega: E \rightarrow \mathbb{Z}$. Il grafo è definito dalla coppia (V, E) dove V è l'insieme dei vertici ed E è l'insieme degli archi.
Il grafo è rappresentato mediante liste di adiacenza, cioè ogni vertice v possiede un array all'interno del quale sono presenti i vertici del grafo verso i quali ha un collegamento in uscita (o arco uscente). Nel nostro caso, gli archi sono coppie ordinate di vertici dal momento che il grafo è orientato.

Nell'interfaccia privata di Graph, sono presenti i seguenti attributi:

- Puntatore a un array di puntatori a vertici
- Puntatore a un array di puntatori ad archi
- Tempo per segnare inizio e fine visita di un nodo
- Puntatore a un array di stringhe, dove ogni stringa contiene il percorso con la massima qualità tra il nodo sorgente e un nodo destinazione del grafo

Nell'interfaccia privata di Graph, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- Setters degli attributi di Graph
- resetVerticesProperties(), un metodo per resettare le proprietà dei vertici del grafo (colore, tempo inizio visita, distanza di cammino massimo, ...)
- getTopologicalOrderStack() e relativa visita DFS, metodi per ottenere un ordinamento topologico dei vertici del grafo
- relax(...) per "rilassare" un determinato arco
- getMaxCostPathToDestination(...) per ottenere il cammino massimo da un determinato vertice sorgente a un determinato vertice destinazione

Nell'interfaccia pubblica di Graph, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- Getters degli attributi di Graph
- addVertex(...) e addEdge(...) per aggiungere rispettivamente un vertice e un arco al grafo G
- getMaxCostPathsFromSource(...) per ottenere il cammino massimo da un determinato vertice sorgente verso tutti gli altri vertici del grafo

Il singolo vertice è rappresentato da una serie di attributi e metodi presenti nella relativa classe Vertex. Nell'interfaccia privata del Vertice, sono presenti i seguenti attributi:

- ID, codice identificativo univoco per ogni vertice
- Puntatore al padre
- d, una stima del cammino massimo per raggiungere quel determinato vertice a partire da una sorgente scelta. Nel momento in cui il vertice viene considerato secondo l'ordinamento topologico, quest'ultimo avrà ottenuto il peso del cammino massimo definitivo (in base alla proprietà del limite inferiore dal momento che stiamo considerando cammini "massimi").
- dTime e fTime, che indicano rispettivamente tempo di inizio e fine visita del vertice
- Dati satellite di tipo T
- Lista di adiacenza del vertice rappresentata da un puntatore a una lista di puntatori ad Arco (Edge)
- Colore, utile proprietà sfruttata durante le visite. In base al colore associato al vertice, si riesce a determinare se il vertice durante una generica visita non è stato ancora visitato, o è iniziata la visita ma non è ancora terminata, o è terminata la visita

Nell'interfaccia privata del Vertice, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- get del puntatore alla lista di puntatori ad Arco che rappresenta la lista di adiacenza del vertice
- resetVertex() per resettare gli attributi del vertice
- addEdgeToAdjacencyList(...), per aggiungere un arco alla lista di adiacenza del vertice sul quale viene invocato

Nell'interfaccia pubblica del Vertice, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- Getters degli attributi del vertice

Il singolo arco è rappresentato da una serie di attributi e metodi presenti nella relativa classe. Nell'interfaccia privata dell'Arco, sono presenti i seguenti attributi:

- ID, codice identificativo univoco per ogni arco
- Puntatore al vertice sorgente
- Puntatore al vertice destinazione
- Peso dell'arco

Nell'interfaccia privata dell'Arco, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- Setters degli attributi dell'Arco

Nell'interfaccia pubblica dell'Arco, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- Getters degli attributi dell'arco

Per quanto riguarda i distruttori, il loro unico ruolo è quello di deallocare la memoria dinamicamente allocata precedentemente per vertici, archi e liste di adiacenza.

Per quanto riguarda i costruttori di Edge, Vertex e Graph non hanno un particolare comportamento da segnalare, oltre a quello dell'assegnazione e inizializzazione degli attributi. Invece, il costruttore di Archipelago si occupa anche dell'apertura di uno stream per la lettura da file e la creazione del grafo con le varie isole e collegamenti.

Infine, come già accennato, vi è Archipelago che rappresenta il nostro Arcipelago: "Grapha-Nui". Nell'interfaccia privata dell'Arcipelago, sono presenti i seguenti attributi:

- Un puntatore a un Graph
- Un puntatore a un vertice sorgente dal quale andare a calcolare i cammini "massimi"
- Una stringa per memorizzare il path (o eventualmente solo il nome) del file di input
- Un puntatore a uno stream di input per gestire la lettura, l'apertura e la chiusura del file

Nell'interfaccia privata dell'Arcipelago, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- Setters e Getters degli attributi
- printError(...), per stampare gli errori a run-time generati dall'utente
- openInputFileStream() e closeInputFileStream() rispettivamente per aprire e chiudere lo stream di input
- buildGraph() per avviare la costruzione del grafo in base ai dati contenuti nel file di input
- checkSource(...) per verificare che l'isola sorgente scelta dall'utente sia opportuna

Nell'interfaccia pubblica dell'Arcipelago, sono presenti i seguenti metodi (per semplicità vengono omessi i parametri):

- addIsland(...) e addBridge(...) per aggiungere rispettivamente un'isola e un collegamento all'arcipelago, cioè rispettivamente un vertice e un arco
- calculateMaxCostPaths() per calcolare i cammini dal costo massimo da sorgente unica nell'arcipelago. Se non è possibile calcolarli, restituisce FALSE. Altrimenti, restituisce TRUE. Questo dipende rispettivamente dalla presenza o meno di cicli di peso positivo.
- printMaxCostPaths() per stampare tutti i cammini "massimi"
- chooseSource(...) per permettere all'utente di scegliere un'isola sorgente a partire dalla quale calcolare i cammini "massimi"
- isFileStreamOpen() per verificare se attualmente lo stream in comunicazione con il file di input, è aperto o meno

Formato dati in input/output

In input è assegnato un file di testo contenente nel primo rigo due interi separati da uno spazio: il numero **N** delle isole (numerate da 0 ad N-1) ed il numero **P** dei ponti. I successivi P righi contengono ciascuno tre numeri I_1, I_2 e Q per indicare che è possibile raggiungere l'isola I_2 dall'isola I_1 dove Q rappresenta la qualità del collegamento.

Le assunzioni che è possibile considerare sui seguenti input sono le seguenti:

- $2 \leq N \leq 1000$
- $1 \leq P \leq 10000$
- $Q_i \in \mathbb{Z}$ per ogni collegamento

Altri dati di input forniti al programma consistono nelle varie scelte effettuate dall'utente mediante menù. Il grafo fornito in input è orientato, pesato e aciclico.

In output il programma stampa i cammini massimi dalla sorgente scelta dall'utente. È presente sia il cammino dal vertice sorgente al vertice destinazione (cioè i nodi che vengono attraversati), sia il peso totale del cammino poc'anzi descritto (somma dei pesi degli archi coinvolti).
Qualora non esista un cammino dal vertice sorgente al vertice destinazione, il programma stamperà che il cammino per raggiungere quella destinazione ha un peso totale pari a -∞.

Descrizione algoritmo

Per la risoluzione del problema, assumiamo che il file contenente i dati di input sopra descritti sia già stato aperto, letto e preso in considerazione. Di conseguenza abbiamo già a disposizione il grafo orientato, aciclico e pesato G = (V, E) sul quale operare.

Rilassando gli archi di un grafo orientato, aciclico e pesato G = (V, E) secondo un ordine topologico dei suoi vertici, è possibile calcolare i cammini massimi da sorgente unica. I cammini massimi sono sempre ben definiti in un DAG; e indipendentemente dal vertice dal quale inizia la visita DFS per generare l'ordinamento topologico, al termine questo ordinamento in un DAG risulta essere sempre lo stesso.

L'algoritmo inizia ordinando topologicamente il DAG per imporre un ordinamento lineare ai vertici. Un ordinamento topologico di un DAG, 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. Se esiste un cammino dal vertice u al vertice v, allora u precede v nell'ordine topologico.
Questo approccio è possibile grazie alla sottostruttura ottima che il problema presenta. Infatti, se è stato trovato il cammino massimo da s a u nel DAG, allora è già stato trovato anche il cammino massimo da s a tutti i nodi interposti fra s e u.

Per ottenere l'ordinamento topologico dei nodi, viene usata una variante della visita DFS in cui la variazione da apportare è l'aggiunta di un'istruzione push su uno stack ausiliario che verrà usato per conservare i vertici la cui visita è terminata (ordine di precedenza fra i nodi).

L'ordine col quale successivamente verranno estratti i vertici dallo stack, è l'ordinamento topologico dei vertici del grafo.

A questo punto, effettuiamo l'estrazione dei vertici dallo stack e per ognuno di essi vengono rilassati tutti gli archi che escono dal vertice. Il processo di rilassare un arco, consiste nel migliorare il cammino massimo di un nodo v trovato fino a quel punto passando per un nodo u ed aggiornando in questo caso $v.\pi$ e $v.d$. Cioè, considerando che ci sia un certo valore per $v.d$ (ottenuto in precedenza), ci chiediamo se percorrendo il cammino che passa per u più il peso dell'arco (u, v), mi porta ad ottenere una stima inferiore di cammino massimo migliore.

Un passo di rilassamento è effettuato dal metodo relax(...) visto prima e può aumentare la stima del cammino massimo $v.d$ ed in tal caso aggiornare $v.\pi$.

Infine, si cerca di rilassare ulteriormente ogni arco. In questo modo, qualora dovesse essere possibile farlo, significa che vi è un ciclo di peso positivo e quindi non è possibile determinare i cammini massimi. In tal caso, l'algoritmo restituisce FALSE, TRUE altrimenti. Nell'ipotesi di DAG del problema, l'algoritmo restituisce sempre TRUE.

Inoltre, l'algoritmo chiama un metodo che permette di salvare nell'array di stringhe associato al grafo, una stringa formattata per ogni vertice contenente il cammino massimo verso quel vertice a partire dalla sorgente scelta. Le stringhe potranno poi essere stampate in output. Il contenuto della singola stringa è descritto nella sezione "Formato dati in input/output".

Alla fine dell'esecuzione del metodo, ogni vertice del grafo avrà il suo valore di peso del cammino massimo. Il cammino massimo per ogni vertice può essere ottenuto seguendo i puntatori ai padri a ritroso. In questo modo, si può ottenere il cammino massimo che va dalla sorgente s a ogni vertice di G. Questo è l'approccio usato per memorizzare le stringhe nell'apposito array di cui abbiamo parlato poc'anzi.

È interessante notare che facendo questo tipo di visita DFS per ottenere l'ordinamento topologico, i vertici che inserisco per prima nello stack, sono quelli che usciranno per ultimi. In questo caso quelli che usciranno per ultimi, sono i vertici da visitare per ultimi.

Inoltre, nell'ipotesi di DAG, è interessante notare che un vertice u che si trova nello stack (le cui estrazioni ripetute forniscono l'ordinamento topologico), può avere degli archi uscenti solo verso i vertici che si trovano al di sotto di esso all'interno dello stack. Cioè u non ha archi uscenti verso i vertici che sono stati inseriti dopo di lui all'interno dello stack.

La correttezza dell'algoritmo è data dal seguente **Teorema**:

Se un grafo orientato pesato G = (V, E) ha sorgente s e nessun ciclo, allora al termine della procedura getMaxCostPathsFromSource(source), $v.d = \delta(s, v)$ per tutti i vertici $v \in V$ e il sottografo dei predecessori G_π è un albero di cammini massimi.

Dimostrazione
Dimostriamo prima che $v.d = \delta(s, v)$ per tutti i vertici $v \in V$ al termine della procedura. Se v non è raggiungibile da s, allora $v.d = \delta(s, v) = -\infty$ per la proprietà dell'assenza di un cammino.
Se invece v è raggiungibile da s, allora esiste un cammino massimo $p = \langle v_0, v_1, \ldots, v_k \rangle$, dove $v_0 = s$ e $v_k = v$. Poiché i vertici vengono elaborati in ordine topologico, gli archi di p vengono rilassati nell'ordine $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$.
La proprietà del rilassamento del cammino implica che $v_i.d = \delta(s, v_i)$ al termine della procedura per i = 0, 1, ... , k. Infine, per la proprietà del sottografo dei predecessori, G_π è un albero di cammini massimi.

La proprietà del rilassamento del cammino coinvolta è la seguente:
Se $p = \langle v_0, v_1, \ldots, v_k \rangle$ è un cammino massimo da $v_0 = s$ a v_k e gli archi di p vengono rilassati nell'ordine $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, allora $v_k.d = \delta(s, v_k)$
Questa proprietà è soddisfatta indipendentemente da altri passi di rilassamento che vengono effettuati, anche se sono interposti fra i rilassamenti degli archi di p. Infatti, non potrebbero far altro che migliorare la stima di cammino massimo.

Lo pseudo-codice seguente implementa la suddetta idea:

```
1.  DFS(G)
2.      S = new stack (∅)
3.      for each u ∈ G.V
4.          u.color = WHITE
5.          u.π = NULL
6.          u.d = -∞
7.          u.dTime =-∞
8.          u.fTime = -∞
9.      for each u ∈ G.V
10.         if u.color == WHITE
11.             DFS_VISIT(u, S)
12.     return S
13.
14. DFS_VISIT(u, S)
15.     u.color = GRAY
16.     u.dTime = time = time + 1
17.     for each v ∈ G.Adj[u]
18.         if v.color == WHITE
19.             v.π = u
20.             DFS_VISIT(v, S)
21.     u.color = BLACK
22.     u.fTime = time = time + 1
23.     push(S, u)
24.
25.
26. getMaxCostPathsFromSource(s, G, w)
27.     stringPaths = ∅
28.     time = 0
29.     S = DFS(G)
30.     s.d = 0
31.     s.π = NULL
32.     while (S ≠ ∅)
33.         u = S.top()
34.         S.pop()
35.         for each v ∈ G.Adj[u]
36.             relax(u, v)
37.
38.     for each arco (u, v) ∈ G.E
39.         if v.d < u.d + w(u, v)
40.             return FALSE
41.
42.     for each v ∈ G.V
43.         getMaxCostPathToDestination(s, v, stringPaths)
44.     return TRUE
```

```
45.  relax(u, v, w)
46.      if v.d < u.d + w(u,v)
47.          v.d = u.d + w(u,v)
48.          v.π = u
49.
50.  getMaxCostPathToDestination(s, v)
51.      S = new stack (∅)
52.      S.push("v.d")
53.
54.      if v.π == NULL OR v.d == −∞
55.          S.push(" → v.ID")
56.      else
57.          while (v.π ≠ NULL)
58.              S.push(" → v.ID")
59.              v = v.π
60.
61.      S.push("s.ID")
62.      while (S ≠ ∅)
63.          pathFromSourceToDestination.append(S.top())
64.          S.pop()
65.
66.      stringPaths = stringPaths ∪ pathFromSourceToDestination
```

Dalla riga 1 alla riga 12 è presente il metodo DFS(G) che fa una visita in profondità nel grafo G.
Nel mentre, memorizza in uno stack ausiliario i nodi la cui visita è stata completata, cioè che sono caratterizzati dal colore BLACK.
Lo stack viene inizializzato come vuoto nella riga 2. Dalla riga 3 a 8, vengono inizializzati gli attributi di ogni vertice nel grafo e successivamente dalla riga 9 alla riga 11, si effettua la DFS_VISIT, passandole il vertice da visitare e lo stack S.
Alla riga 12, viene restituito lo stack S che conterrà i vertici del grafo G. La loro estrazione in successione permette di avere il loro ordinamento topologico.

Dalla riga 14 alla riga 23, è presente la DFS_VISIT(u, S) che imposta il colore del nodo passato come parametro a GRAY, imposta il tempo di inizio visita e successivamente per ogni vertice adiacente al nodo u, se questo è ancora di colore WHITE (cioè ancora non è avvenuta la visita su di esso), allora si imposta u come il padre di v e si procede con la ben nota chiamata ricorsiva sul vertice v considerato.
Dopo che tutti i vertici che vengono raggiunti da u sono stati ispezionati, la riga 21 colora il vertice di BLACK (visita terminata), si imposta il tempo di fine visita e si fa il push sullo stack S del vertice la cui visita è terminata.

Dalla riga 26 alla riga 44, vi è il "cuore" dell'algoritmo. Il metodo inizializza l'array di stringhe formattate contenente i cammini massimi "stringPaths" come un insieme vuoto, imposta il tempo (usato durante le visite) a zero e invoca la DFS per ottenere l'ordinamento topologico poc'anzi descritto dei vertici del grafo G. A questo punto, si imposta la distanza di cammino massimo a 0 e il padre a NULL per il nodo sorgente.

A questo punto si procede con la visita in ordine topologico dei nodi, estraendoli man mano dallo stack e cercando di rilassare di volta in volta tutti gli archi uscenti dal nodo u estratto dallo stack S.

Dalla riga 38 alla riga 40, vi è il controllo per la presenza di eventuali cicli di peso positivo (qualora il grafo in input non fosse un DAG).

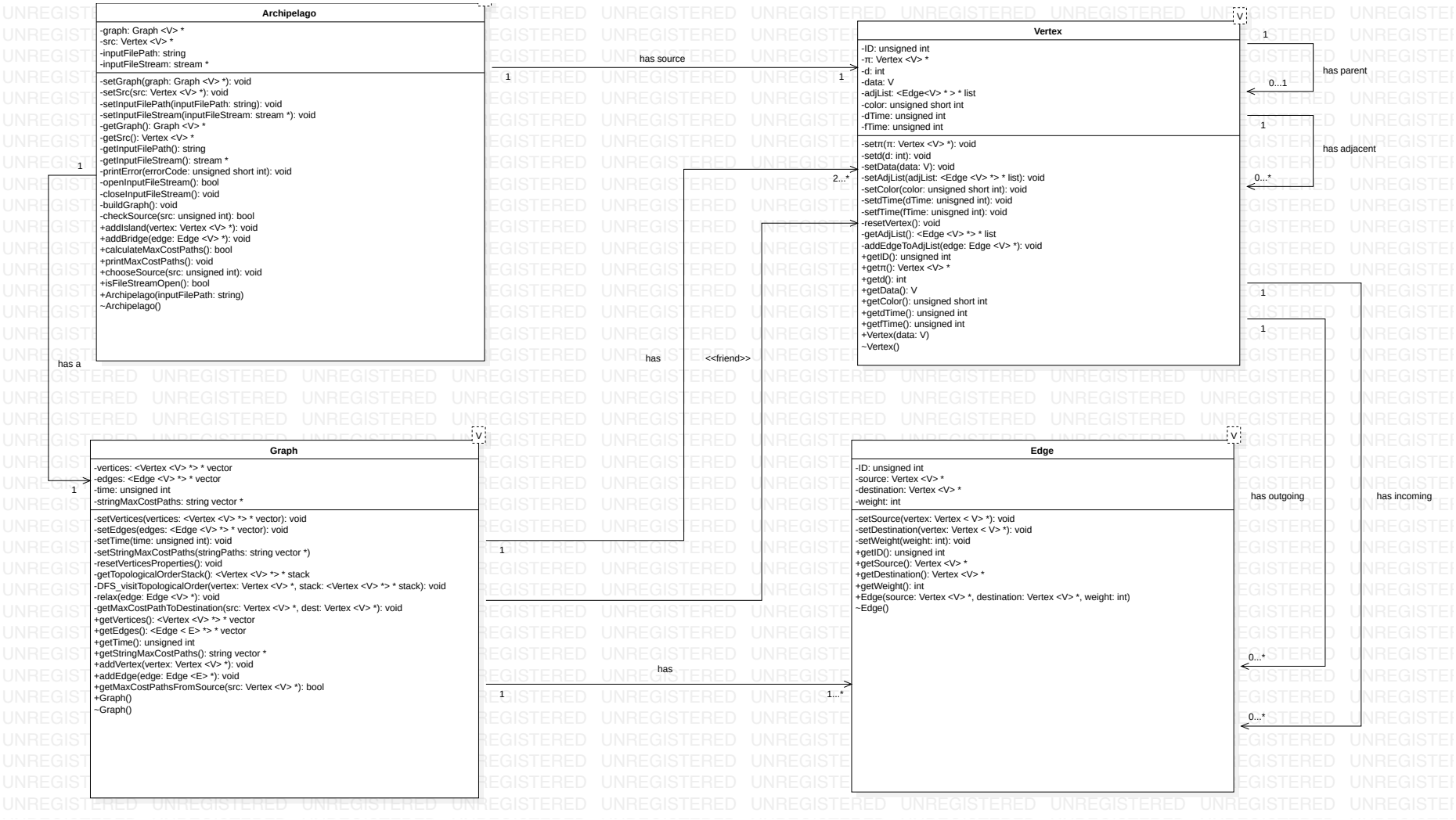
Dalla riga 42 alla riga 44, vi è la chiamata al metodo che permette di riempire l'array di stringhe formattate contenente i cammini massimi da sorgente singola.

Dalla riga 45 alla riga 47 è presente il metodo che cerca di andare a migliorare la stima di cammino massimo per il vertice v. Quest'ultimo permette di andare a massimizzare il peso totale del cammino mediante l'if al suo interno.

Infine, dalla riga 50 alla riga 66, è presente il metodo che permette in maniera iterativa di andare a ripercorrere i puntatori ai padri per ogni nodo e salvare una stringa opportunamente formattata da stampare poi in output all'utente. Per fare ciò si usa uno stack ausiliario e poi con le estrazioni in successione, si va a fare l'append della stringa uscita dallo stack in una stringa locale.
Alla fine, la stringa locale viene memorizzata in un array in cui al termine delle V chiamate saranno presenti tutti i cammini massimi sotto forma di stringhe.

C'è da precisare che nello stack, si inseriscono di volta in volta delle stringhe. Inoltre, era possibile fare a meno dello stack ausiliario, sfruttando un approccio ricorsivo. In questo caso però, ho preferito un approccio iterativo.

Class Diagram



La classe Archipelago ha al suo interno un puntatore a un oggetto di tipo Graph parametrizzato dal parametro V. L'Archipelago ha un Graph perché Archipelago è stata interpretata come la classe "Problema" che per l'appunto risolve il problema dei cammini massimi da sorgente singola su un determinato Graph.

La classe Archipelago ha una relazione di 1...1 con Graph. Infatti, un Archipelago ha un solo Graph al suo interno e viceversa, un Graph è contenuto in un solo Archipelago.
Inoltre, Archipelago ha un vertice sorgente a partire dal quale calcolare i cammini massimi. Quindi Archipelago ha come sorgente un Vertex. La molteplicità anche in questo caso è di 1...1, infatti un Archipelago ha come sorgente un solo Vertex.

La classe Archipelago svolge buona parte del lavoro nel momento in cui viene istanziato un oggetto di tipo Archipelago. Infatti, richiede un inputFilePath cioè "nome.estensione" del file da aprire (es. "input.txt"), avvia lo stream, apre il file e costruisce il grafo. A questo punto l'utente invoca il metodo calculateMaxCostPaths() che procederà (come visto anche in precedenza), a calcolare i cammini massimi da sorgente singola.
Gli altri metodi sono per lo più setters, getters e metodi di gestione dello stream, verifica nodo sorgente, aggiunta di un'isola/collegamento e un ulteriore metodo per la stampa.

La classe Graph è anch'essa parametrizzata dal parametro V. Possiede al suo interno un puntatore ad array di Vertex e un puntatore ad array di Edge. Quindi la relazione con la classe Vertex è di tipo "has" Vertex e la molteplicità è 1 ... 2...*; in quanto le assunzioni del quesito dicono che ci sono almeno due isole. Per la relazione con la classe Edge, è anch'essa di tipo "has" e la molteplicità è di tipo 1 ... 1...* in quanto le assunzioni del quesito dicono che c'è almeno un arco.

È importante precisare che la classe Graph è "friend" di Vertex. Di conseguenza potrà accedere anche ad attributi e metodi privati della classe Vertex. Nonostante ciò, l'utente finale ha modo di accedere solo ai metodi pubblici di Archipelago senza sapere cosa effettivamente accade, né avendo la possibilità di accedere a metodi e attributi di Vertex, Edge o Graph.
L'utente può solo scegliere una nuova sorgente, aggiungere un'isola o un collegamento manualmente, calcolare i cammini massimi, stampare i cammini massimi, verificare che il file stream sia aperto.

Inoltre, è interessante notare che proprio per questo motivo, benché molti metodi ritornino puntatori a dati membro privati della classe, l'incapsulamento e la sicurezza dei dati non è compromessa.

La classe Vertex è anch'essa parametrizzata dal parametro V. Un Vertex ha diversi attributi che lo caratterizzano come è stato menzionato anche in precedenza nella sezione "Descrizione strutture dati". Per quanto riguarda le relazioni, un Vertex può avere 0 o più Vertex presenti nella sua lista di adiacenza. Per cui la relazione è di tipo "has" e la sua molteplicità è 1 ... 0...*.
Inoltre, Vertex può avere un Vertex padre e quindi la relazione è di tipo "has" anche in questo caso e la molteplicità è di tipo 1 ... 0...1.

La classe Vertex può avere sia degli archi uscenti che degli archi entranti in quanto grafo orientato. Cioè un Vertex u nella sua lista di adiacenza può avere dei vertici verso i quali ha degli archi uscenti. Allo stesso tempo altri Vertex possono avere degli archi uscenti verso u. In questo caso però, per u saranno degli archi entranti. In entrambi i casi la relazione è di tipo "has" e la molteplicità è 1 ... 0...*

Infine, abbiamo la classe Edge che rappresenta il singolo Arco. La classe Edge ha un Vertex sorgente, un Vertex destinazione e il peso di tale arco, risultando essere strettamente collegata con la classe Vertex per il discorso poc'anzi fatto su archi uscenti, archi entranti e in quanto Vertex possiede un puntatore a una lista di puntatori a Edge parametrizzati da V.

La classe template V in questo specifico caso non è sfruttata al meglio delle sue potenzialità. Una possibile applicazione generalizzata della classe template V è quella di avere nel file di input, oltre a interi identificanti l'isola, anche dei dati satellite.
Però, per come è stato progettato il metodo buildGraph(), attualmente non è in grado di gestire tale eccezione.

Studio complessità

Analizziamo la complessità di tempo della DFS:

- Il primo ciclo impiega un tempo $\Theta(V)$
- Il secondo ciclo invoca la DFS_VISIT esattamente una volta per ogni vertice e solo se quest'ultimo risulta essere ancora di colore WHITE. Il ciclo della DFS_VISIT viene eseguito E volte in totale.

Di conseguenza la complessità di tempo per ottenere l'ordinamento topologico dei vertici è: $\theta(V + E)$ dove V è il numero dei vertici ed E è il numero degli archi.

Analizziamo la complessità di tempo di getMaxCostPathsFromSource:

- Esegue l'ordinamento topologico dei vertici: $\Theta(V + E)$
- Esegue un ciclo while estraendo di volta in volta un vertice e cercando di rilassare tutti i suoi archi uscenti: $\Theta(V + E)$
- Esegue un ulteriore ciclo for sugli archi alla ricerca di un eventuale ulteriore rilassamento: $\Theta(E)$
- Esegue un ultimo ciclo for sui vertici per andare a costruire l'array di stringhe dei cammini massimi tramite il metodo getMaxCostPathToDestination: $\Theta(V)$

Quindi la complessità di tempo totale a livello asintotico è data da: $\Theta(V + E)$.

Per quanto riguarda la complessità di spazio, vi è:

- Lo spazio necessario a memorizzare il grafo con i suoi vertici, archi e liste di adiacenza: $\Theta(V + E)$
- Lo spazio necessario ad allocare uno stack per l'ordinamento topologico e uno stack per costruire le stringhe dei cammini massimi: $\Theta(V)$

Quindi la complessità di spazio totale a livello asintotico è data da: $\Theta(V + E)$.

Test e risultati

Il programma permette all'utente di scegliere l'isola sorgente a partire dalla quale calcolare i cammini massimi da sorgente singola. Se la scelta dell'utente va a buon fine, si procede con l'elaborazione di essi.
Qualora sia possibile determinare i cammini massimi, si procede con la stampa di essi.
Successivamente viene chiesto all'utente se vuole calcolare i cammini massimi sullo stesso Arcipelago, ma cambiando l'isola sorgente.

Il programma termina nel momento in cui l'utente esprime la propria volontà di non voler calcolare altri cammini massimi da una sorgente differente.

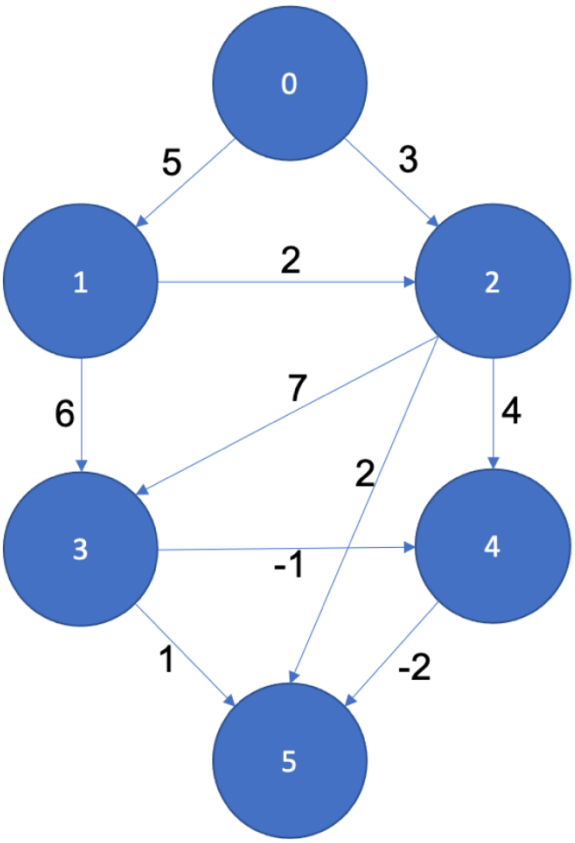
È presente un controllo sull'input inserito dall'utente sia sull'isola scelta, sia sulla scelta che esegue quando gli viene chiesto di continuare o meno con l'esecuzione (scegliendo un'altra isola sorgente).

L'output mostrato qui di seguito è descritto formalmente nella sezione "Formato dati in input/output".

```
Benvenuti nell'Arcipelago di Grapha-Nui...

Le isole e i ponti verranno caricati a breve. Inserisci l'isola a partire
dalla quale vuoi calcolare i percorsi col costo massimo verso tutte le
altre isole dell'arcipelago.

Digita un numero da 0 a (N - 1) dove N è il numero delle isole totali
presenti nel file che hai caricato.

Digita qui: 0
------------------------------
0->0   0


------------------------------
0->1   5


------------------------------
0->1->2   7


------------------------------
0->1->2->3   14


------------------------------
0->1->2->3->4   13


------------------------------
0->1->2->3->5   15




Vuoi partire da un'altra isola? (Y = YES / N = NO).
Scegli: y
```

```
Benvenuti nell'Arcipelago di Grapha-Nui...

Le isole e i ponti verranno caricati a breve. Inserisci l'isola a partire
dalla quale vuoi calcolare i percorsi col costo massimo verso tutte le
altre isole dell'arcipelago.

Digita un numero da 0 a (N - 1) dove N è il numero delle isole totali
presenti nel file che hai caricato.

Digita qui: 1
------------------------------
1->0   -INF


------------------------------
1->1   0


------------------------------
1->2   2


------------------------------
1->2->3   9


------------------------------
1->2->3->4   8


------------------------------
1->2->3->5   10




Vuoi partire da un'altra isola? (Y = YES / N = NO).
Scegli: y
```

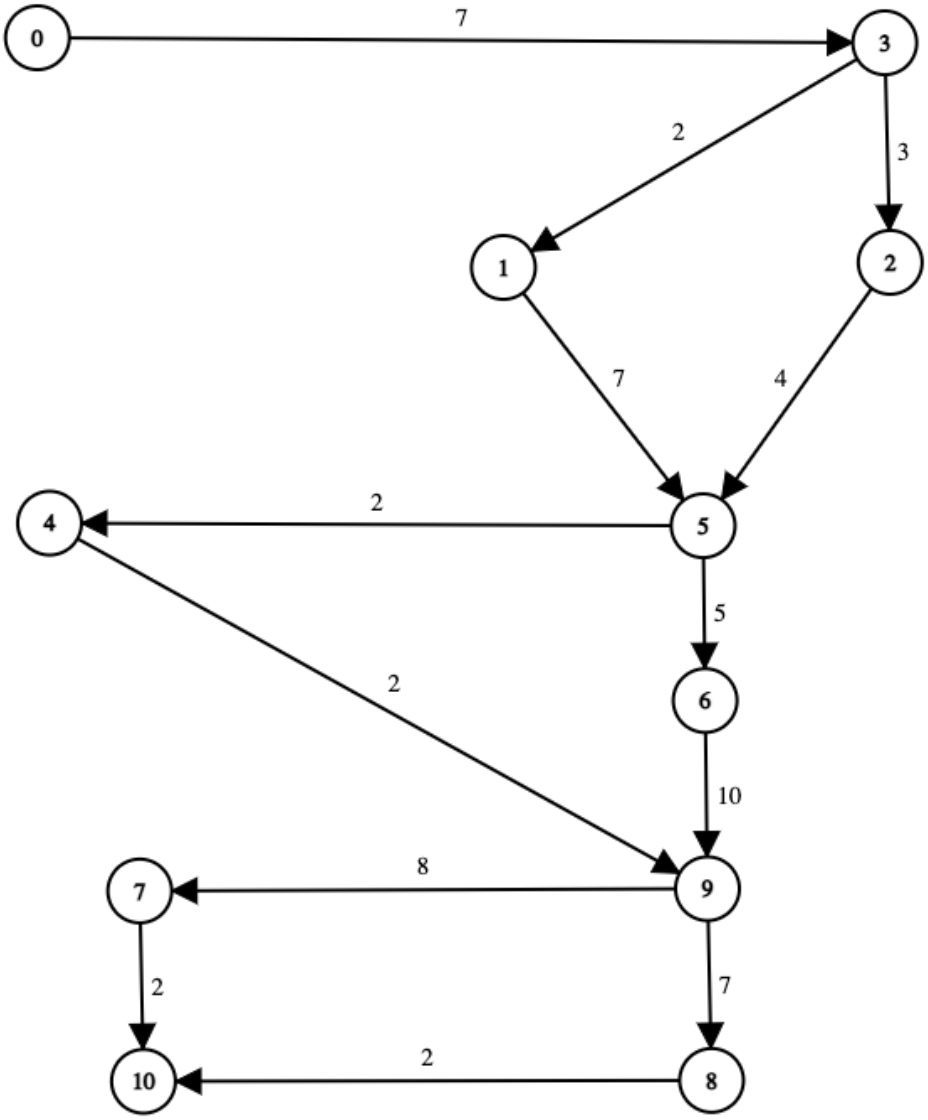
```
Benvenuti nell'Arcipelago di Grapha-Nui...

Le isole e i ponti verranno caricati a breve. Inserisci l'isola a partire
dalla quale vuoi calcolare i percorsi col costo massimo verso tutte le
altre isole dell'arcipelago.

Digita un numero da 0 a (N - 1) dove N è il numero delle isole totali
presenti nel file che hai caricato.

Digita qui: 2
------------------------------
2->0   -INF


------------------------------
2->1   -INF


------------------------------
2->2   0


------------------------------
2->3   7


------------------------------
2->3->4   6


------------------------------
2->3->5   8



Vuoi partire da un'altra isola? (Y = YES / N = NO).
Scegli:
```



```
Benvenuti nell'Arcipelago di Grapha-Nui...

Le isole e i ponti verranno caricati a breve. Inserisci l'isola a partire
dalla quale vuoi calcolare i percorsi col costo massimo verso tutte le
altre isole dell'arcipelago.

Digita un numero da 0 a (N - 1) dove N è il numero delle isole totali
presenti nel file che hai caricato.

Digita qui: 0
-----------------------------
0->0   0


-----------------------------
0->3->1   9


-----------------------------
0->3->2   10


-----------------------------
0->3   7


-----------------------------
0->3->1->5->4   18


-----------------------------
0->3->1->5   16


-----------------------------
0->3->1->5->6   21


-----------------------------
0->3->1->5->6->9->7   39


-----------------------------
0->3->1->5->6->9->8   38


-----------------------------
0->3->1->5->6->9   31


-----------------------------
0->3->1->5->6->9->7->10   41
```

```
Benvenuti nell'Arcipelago di Grapha-Nui...

Le isole e i ponti verranno caricati a breve. Inserisci l'isola a partire
dalla quale vuoi calcolare i percorsi col costo massimo verso tutte le
altre isole dell'arcipelago.

Digita un numero da 0 a (N - 1) dove N è il numero delle isole totali
presenti nel file che hai caricato.

Digita qui: 1
----------------------------
1->0   -INF


----------------------------
1->1   0


----------------------------
1->2   -INF


----------------------------
1->3   -INF


----------------------------
1->5->4   9


----------------------------
1->5   7


----------------------------
1->5->6   12


----------------------------
1->5->6->9->7   30


----------------------------
1->5->6->9->8   29


----------------------------
1->5->6->9   22


----------------------------
1->5->6->9->7->10   32
```

```
Benvenuti nell'Arcipelago di Grapha-Nui...

Le isole e i ponti verranno caricati a breve. Inserisci l'isola a partire
dalla quale vuoi calcolare i percorsi col costo massimo verso tutte le
altre isole dell'arcipelago.

Digita un numero da 0 a (N - 1) dove N è il numero delle isole totali
presenti nel file che hai caricato.

Digita qui: 2
------------------------------
2->0   -INF


------------------------------
2->1   -INF


------------------------------
2->2   0


------------------------------
2->3   -INF


------------------------------
2->5->4   6


------------------------------
2->5   4


------------------------------
2->5->6   9


------------------------------
2->5->6->9->7   27


------------------------------
2->5->6->9->8   26


------------------------------
2->5->6->9   19


------------------------------
2->5->6->9->7->10   29
```


0
3
1
2
4
5
6
9
7
8
10
7
2
3
7
4
2
5
2
10
8
2
7
2