**Relazione sul secondo progetto per l’esame di Ingegneria degli Algoritmi**

Fabio Buracchi, 0253822

Danilo D’Amico, 0252956

**PREFAZIONE**

Seguendo le indicazioni riportate nella traccia, è stato svolto il Progetto 1.

Il codice è disponibile su GitHub al seguente link: https://github.com/Zaimokuza/EX2IA19.git

Nella realizzazione del progetto è stato utilizzato esclusivamente codice scritto dagli autori, presente nelle librerie standard di o fornito sul sito GitHub del corso.

Il file costituisce un interessante esempio di casi di utilizzo del codice scritto.

Il progetto è stato sviluppato sulla base dei moduli forniti a lezione contenuti nella cartella . Tra i file presenti nella cartella principale, implementa l’algoritmo come metodo di una classe figlia della classe astratta vista a lezione. L’algoritmo è basato sul codice del metodo , rappresentante una visita generica. La frangia F della visita è una coda con priorità gestita in tramite una classe che ne seleziona un’implementazione tra quelle contenute nella cartella .

costruisce un grafo non orientato, connesso e pesato sui vertici, avente il minimo numero di nodi possibili e lasciando a l’onere di inserire all’occorrenza eventuali archi aggiuntivi.

Tra le classi contenute nella cartella 𝐿𝑖𝑏𝑟𝑒𝑟𝑖𝑒, sono state apportate modifiche solo in PQbinomialHeap.py, nel quale è stato innalzato, nella riga 78, il valore di , di cui si discuterà più avanti nella relazione.

**SCELTE IMPLEMENTATIVE**

Nei paragrafi successivi verrà analizzato nel dettaglio il codice della classe in relazione alla sua classe madre, alla classe e a quello delle strutture dati che implementa.

Essendo figlia di una classe astratta, sono stati implementati tutti i metodi astratti presenti in con l’eccezione di che è stato ignorato in favore di un metodo in grado di gestire un grafo pesato sui vertici tramite una classe figlia di .

Nel costruttore della nuova classe grafo è presente un attributo che gestisce gli archi mediante liste di adiacenza, scelte in quanto consentono la visita generica di un grafo in tempo lineare , in cui rappresenta il numero di archi e il numero di nodi presenti nel grafo. I metodi e aggiungono e sottraggono elementi a tali liste andando a modificare le liste di adiacenza sia della testa che della coda dell’arco. Questa doppia operazione si rende necessaria perché il grafo da gestire non è orientato. Ne consegue che in ogni momento sono presenti nelle liste di adiacenza elementi, che occupano comunque spazio .

Data la natura della gestione degli archi, la classe non possiede un tipo di dato , come richiesto di ritornare dai metodi e , ma si affida a contenenti testa e coda dell’arco. Si noti che l’implementazione di restituisce solamente elementi, inserendo nella lista che viene ritornata soltanto la prima tra le due istanze di uno stesso arco.

Il metodo si occupa di implementare la visita richiesta dalla traccia. Scandisce in la lista dei nodi per trovare quello con priorità maggiore, con il quale inizializzerà la coda con priorità F. Le implementazioni delle code con priorità presenti in mantengono come primo valore la chiave minima presente al loro interno. Dato che, al contrario, il metodo deve estrarre il massimo ad ogni iterazione del suo ciclo , durante l’inserimento in coda invertiamo il segno del peso del nodo, che usiamo come chiave. Il ciclo è scritto seguendo la stessa logica della visita generica presente in : viene iterato finché la frangia F non si svuota, seguendo un sistema di marcatura in nodi , , ed . Un nodo viene aperto quando è estratto come minimo dalla coda con priorità e considerato immediatamente dopo, con il suo inserimento nella lista e nell’insieme . In viene mantenuto l’elenco ordinato dei nodi incontrati compiendo le estrazioni dalla coda, mentre in è mantenuto lo stesso insieme di elementi ma in modo disordinato; questo consente, alla riga 157, di verificare la marcatura di in tempo . Scorrere la lista avrebbe, invece, richiesto un tempo . L’ di riga 155 velocizza l’esecuzione del ciclo nell’evenienza in cui tutti i nodi siano già stati visitati o inseriti nella coda, saltando la ricerca di un nodo adiacente a quello appena estratto dalla coda che possa esservi inserito.

Il costo del metodo appena studiato risulta come la somma di operazioni di estrazione del massimo dalla coda con priorità e di fino a operazioni compiute dal ciclo for nell’iterazione dei nodi adiacenti al massimo. In altre parole:

Ne deriva che la scelta della coda con priorità riveste un ruolo non indifferente nel costo asintotico della visita. Segue una tabella che riassume i costi asintotici delle operazioni compiute per l’estrazione del massimo per ognuna delle tre code con priorità presenti in .

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  | = O(log (n)) |
|  |  |  |
|  |  |  |

Il caso ideale studiato per il quale il costo della visita sarebbe stato lineare in è rappresentato dall’evenienza in cui sia che si possano eseguire in tempo ammortizzato . Come si può vedere nei grafici presenti nella sezione Risultati Sperimentali, un buon costo ammortizzato per l’estrazione del massimo è fornito da , il quale, operando su una foresta di alberi anziché su di un singolo albero, esegue le operazioni richieste in un tempo ammortizzato costante.

Una seconda tabella riassume i costi di tutti i metodi implementati nella classe :

|  |  |
| --- | --- |
| **Nome Metodo** | **Costo** |
|  |  |
|  | / |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Con la notazione ci riferiamo al grado del vertice . Tale notazione ci è utile in quanto, quando è possibile scorrere più di una lista, con un confronto tra i gradi delle due eseguito tramite il metodo , si può evitare di studiare la più lunga.

Dedichiamo infine un paragrafo all’implementazione del generatore del grafo G presente in . L’obiettivo è, a partire da un grafo vuoto e pesato sui vertici, riempire G in modo che sia connesso. Dato che in è rilevante eseguire i medesimi esperimenti su grafi connessi che abbiano il minimo numero di archi o il massimo, costruiamo nella funzione un grafo con nodi. Tale risultato viene raggiunto seguendo un algoritmo che mantiene G connesso e al quale, nel passo generico i-esimo, viene aggiunto un nodo collegato tramite un singolo arco ad uno a caso tra i nodi già presenti. Nel caso si voglia eseguire dei test su di un grafo con archi, invece, viene in aiuto di la funzione , che aggiunge a G tutti gli archi possibili non presenti.

È stato incrementato il valore di presente nella libreria PQbinomialHeap.py in quanto una foresta binomiale contenente alberi di altezza massima 32 non è conveniente per gestire code con priorità di grafi aventi 1000 elementi.

**RISULTATI SPERIMENTALI**

Gli esperimenti sono stati condotti utilizzando le ultime versioni disponibili di (versione 2018.3.3) e dell’interprete di (versione 3.7.2).

I test sono il risultato della media di 100 prove eseguite su grafi connessi di 10, 100 e 1000 elementi, aventi ognuno in un primo test il minimo numero di nodi possibili ed in un secondo il massimo.

Le code con priorità considerate sono dei d-Heap con d = 3, 10, 100, un heap binario ed un heap binomiale.

|  |  |
| --- | --- |
| Test su d-Heap con archi | Test su d-Heap con archi |
| Test su code con priorità aventi archi | Test su code con priorità aventi archi |

A livello asintotico i grafici assumono un andamento rispecchiante le considerazioni di carattere teorico effettuate nella sezione precedente della relazione.

Dai test di utilizzo medio emerge che, mentre tendendo a 1000 elementi il 10-heap ha costo inferiore, tra i 10 ed i 100 elementi il 100-heap ha un deciso vantaggio.

Si noti anche come, all’aumentare del numero di archi, aumenta il tempo necessario per la visita con priorità, che tra i due test raddoppia per i d-heap ed aumenta leggermente per l’heap binomiale, che riesce ciò nonostante a mantenere un costo ottimo grazie alla maggior flessibilità della sua struttura dati a foresta.