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

Nella realizzazione del progetto è stato utilizzato esclusivamente codice scritto dagli autori, presente nelle librerie standard di $Python$ o fornito sul sito GitHub del corso, con l'eccezione dei package $matplotlib$ e $pandas$ richiesti dal modulo $plotter.py$ per la generazione dei grafici presenti in questa relazione.

Un esempio d'uso della struttura dati creata per la gestione del grafo è contenuta nel file $esempiDiUtilizzo.py$

Verranno utilizzati frequentemente nel resto della relazione m ed n per riferirsi rispettivamente agli archi ed ai vertici di un grafo e ci si riferirà spesso ai due nodi connessi da un arco con gli appellativi di testa e coda.

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

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

SCELTE IMPLEMENTATIVE

Perno del progetto è la classe $CustomGraph$ presente nell'omonimo modulo, sottoclasse della classe $Graph$ presente nella directory $Librerie$. Essendo $CustomGraph$ derivata da una classe astratta sono stati implementati tutti i metodi astratti presenti nella classe base, ad eccezione di $addNode$, ritenuto incompatibile vista l'implementazione della classe derivata $WeightedNode$ figlia della classe $Node$, ritenuta necessaria per adempiere alla richiesta della traccia. In sostituzione al modulo sopra citato è stato inserito un metodo $addWeightedNode$.

La classe $CustomGraph$ gestisce gli archi mediante liste di adiacenza, che consentono l'esecuzione di una visita generica di un grafo in tempo lineare $O(m+n)$.

Come conseguenza diretta i metodi $insertEdge$ e $deleteEdge$ devono modificare sia la lista di adiacenza della testa di un arco che quella della sua coda. Dunque il numero totale di archi presenti nelle liste di adiacenza è esattamente il doppio del numero di archi del grafo, occupando quindi spazio $O(m)$.

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

Il metodo $visitaInPriorità$ implementa la visita richiesta dalla traccia, per poterne trattare adeguatamente è necessario innanzitutto analizzare la classe $PriorityQueue$.

La classe $PriorityQueue$ contenuta nell'omonimo modulo presente nella $root$ della repository permette di inizializzare un'oggetto contenente una coda con priorità. La scelta del tipo di coda è stata resa modulare, difatti attraverso il parametro d'inizializzazione $type$ si può scegliere quale tipo di coda prioritaria utilizzare. È stata utilizzata una classe di enumerazione per poter rendere semplice l'aggiornamento e l'implementazione dei vari tipi di coda prioritaria.

È stato inoltre aumentato il parametro $MAXSIZE$ della classe $PQbinomialHeap$ presente nell'omonima libreria per permettere alla coda di risultare più efficiente nella fase di testing.

Il metodo $visitaInPriorità$ inizializza proprio una frangia F, istanza della classe $PriorityQueue$, e ricerca in $O(n)$ il nodo con peso maggiore nella lista dei nodi da inserire nella coda prioritaria. La visita eredita la logica della visita generica presente in $GraphBase$: un ciclo $while$ viene iterato finché la frangia F non si svuota, seguendo un sistema di marcatura dei nodi.

I nodi estratti dalla coda vengono inseriti nella lista $visited\_nodes$ e nel $set\ explored\_nodes$. Mentre $visited\_nodes$ mantiene l'elenco ordinato dei nodi incontrati, $explored\_nodes$ utilizza un sistema di hashing e permette di verificare la marcatura in tempo di $O(1)$ occupando comunque sia solamente spazio $O(n)$.

Le implementazioni delle code con priorità presenti in $Librerie$ mantengono come primo valore la chiave minima presente al loro interno. Dato che, al contrario, il metodo $visitaInPriorità$ deve estrarre il massimo, durante l'inserimento in coda di un nodo invertiamo il segno del suo peso per non modificare le librerie.

Per migliorare inoltre il tempo medio di esecuzione, l'algoritmo controlla ad ogni iterazione se la lista dei nodi e la frangia contengono il totale dei nodi del grafo, in caso affermativo questo smette di controllare le liste di adiacenza.

Il costo del metodo appena studiato risulta come la somma di n operazioni di estrazione del massimo dalla coda con priorità e di fino a $2m$ operazioni compiute dal ciclo for nell'iterazione dei nodi adiacenti al massimo. In altre parole:

$$O(costoEstrazioneMassimo * n + m)$$

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 $priorityQueues$.

	$findMin()$	$deleteMin()$
$D-Heap$	$O(1)$	$O\big(d\log_d(n)\big) = O(\log(n))$
$Binary\ Heap$	$O(1)$	$O(\log(n))$
$Binomial\ Heap$	$O(\log(n))$	$O(\log(n))$

Il caso ideale studiato per il quale il costo della visita sarebbe stato lineare in $O(m+n)$ è rappresentato dall'evenienza in cui sia $findMin()$ che $deleteMin()$ si possano eseguire in tempo ammortizzato $O(1)$. Come si può vedere nei grafici presenti nella sezione Risultati Sperimentali, un buon costo ammortizzato per l'estrazione del massimo è fornito da $Binomial\ Heap$, 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 $CustomGraph$:

Nome Metodo	**Costo**
$numEdges$	$O(1)$
$addNode$	/
$addWeightedNode$	$O(1)$
$deleteNode$	$O(m)$
$getNode$	$O(1)$
$getNodes$	$O(n)$
$insertEdge$	$O(1)$
$deleteEdge$	$O(\sigma(x) + \ \sigma(y))$
$getEdge$	$O(\min\{\sigma(x), \sigma(y)\})$
$getEdges$	$O(m)$
$isAdj$	$O(\min\{\sigma(x), \sigma(y)\})$
$getAdj$	$O(1)$
deg	$O(1)$
$print$	$O(n+m)$

Con la notazione $\sigma(x)$ ci riferiamo al grado del vertice x. 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 deg, si può evitare di studiare la più lunga.

Dedichiamo infine un paragrafo all'implementazione del generatore del grafo G presente in $graphGenerator.py$. L'obiettivo è, a partire da un grafo vuoto e pesato sui vertici, riempire G in modo che sia connesso. Dato che in $test.py$ è rilevante eseguire i medesimi esperimenti su grafi connessi che abbiano il minimo numero di archi o il massimo, costruiamo nella funzione $graphGenerator$ un grafo con $n-1$ 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 il massimo numero di archi, invece, viene in aiuto di $test.py$ la funzione $fillEdges()$, che aggiunge a G tutti gli archi possibili non presenti.

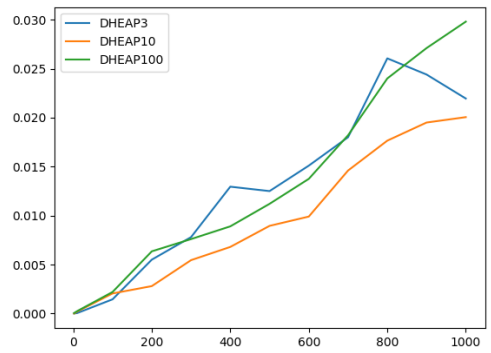
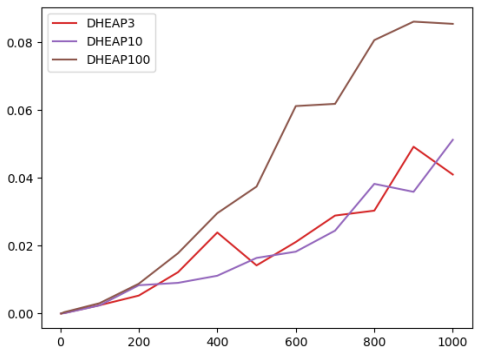
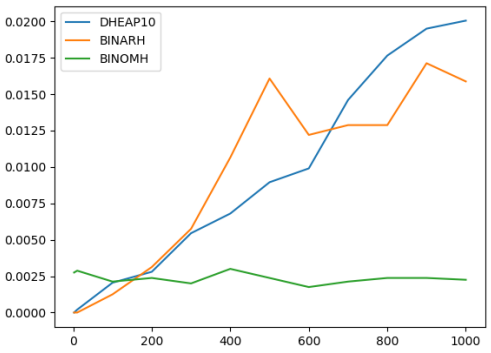
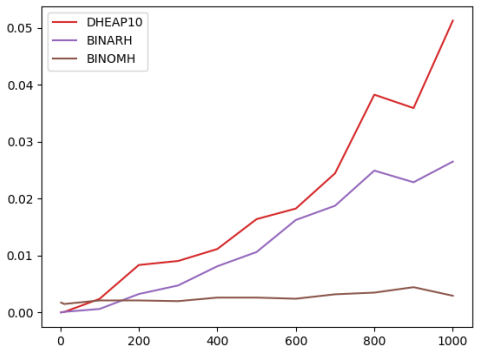
RISULTATI SPERIMENTALI

Gli esperimenti sono stati condotti utilizzando le ultime versioni disponibili di $PyCharm$ (versione 2018.3.3) e dell'interprete di $Python$ (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.

Dai test è emerso che il $D - Heap$ con $d = 10$ poteva essere ritenuto un adeguato rappresentare della sua struttura.

Test su D-Heap con $n - 1$ archi	Test su D-Heap con $\frac{n(n-1)}{2}$ archi
	
Test su code con priorità aventi $n - 1$ archi	**Test su code con priorità aventi $\frac{n(n-1)}{2}$ archi**
	

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

Si noti anche come, all'aumentare del numero di archi, aumenti 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.