Relazione Progetto 2

Ingegneria degli Algoritmi 2018/2019

Matteo Lungo – 0253524

Scelte Implementative

Per la realizzazione del progetto, la struttura Grafo è stata modificata implementando i metodi **dHeapPrioritySearch** e **binomialHeapPrioritySearch** all'interno della classe **GraphBase** e il parametro **weight** all'interno della classe **Node** (file Graph.py) secondo le istruzioni fornite dal testo. All' interno dei due metodi di visita, il set di nodi da esplorare è stato rimpiazzato rispettivamente con un D-Heap (PQ_DHeap.py) e un Heap Binomiale (PQbinomialHeap.py). I nodi visitati vengono infine mostrati tramite il print della lista **markedNodes**. Per l'inizializzazione e l'estrazione del valore massimo (corrispondente al peso del nodo), le due strutture sono state adeguatamente modificate. In particolare, i metodi modificati sono:

Per D-Heap:

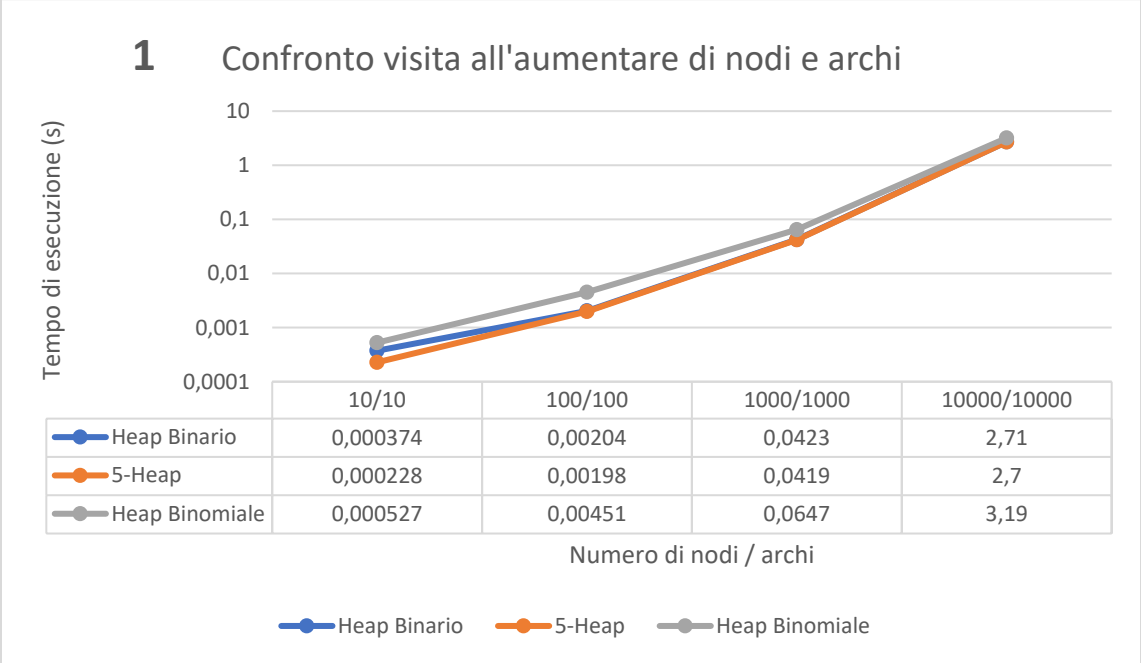
- **minSon → maxSon**
- **findMin → findMax**
- **deleteMin → deleteMax**
- **insert**

Per Heap Binomiale:

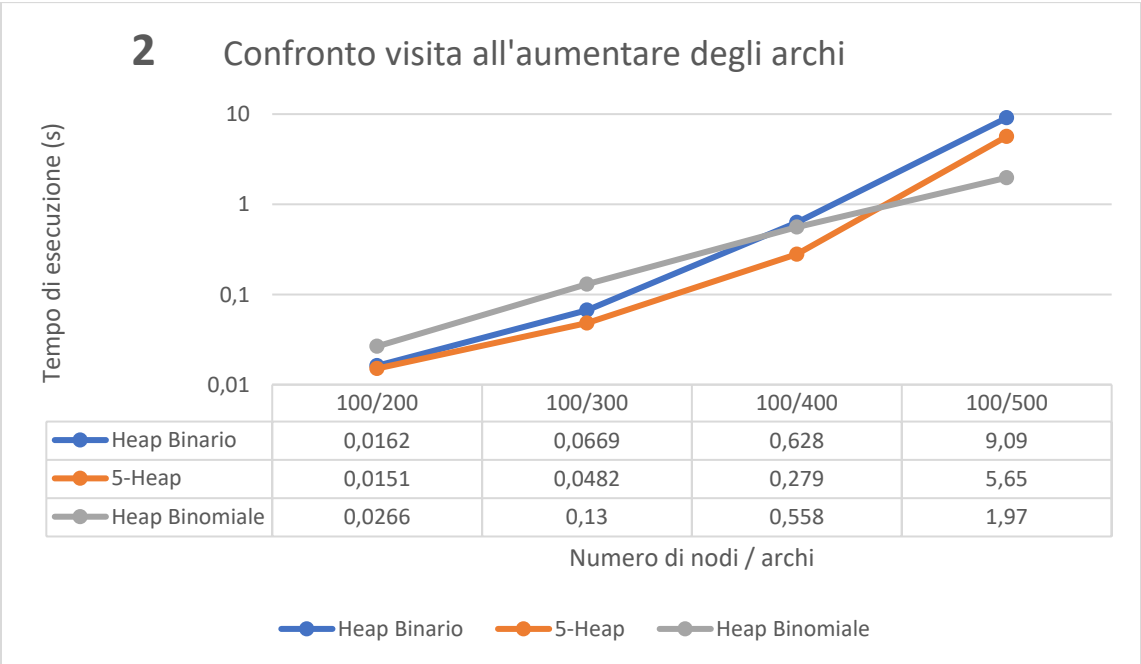
- **findMinIndex → findMaxIndex**
- **findMin → findMax**
- **deleteMin → deleteMax**
- **insert**
- aggiunto metodo **lenght**

Per la generazione del grafo è stato ricostruito il metodo **buildGraph** del **GraphHelper** (file graphHelper.py). Il metodo in questione genera un grafo dal numero di nodi scelto a cui vengono attribuiti valori e pesi casuali. L'algoritmo genera poi il numero di archi scelto (dal valore casuale), assicurandosi che il grafo sia connesso. La stampa del grafo avviene tramite lista d'incidenza (definita nel file Graph_IncidenceList.py), modificata in modo da visualizzare al fianco di ogni nodo il suo peso.

Risultati Sperimentali e Commenti



	10/10	100/100	1000/1000	10000/10000
Heap Binario	0,000374	0,00204	0,0423	2,71
5-Heap	0,000228	0,00198	0,0419	2,7
Heap Binomiale	0,000527	0,00451	0,0647	3,19



	100/200	100/300	100/400	100/500
Heap Binario	0,0162	0,0669	0,628	9,09
5-Heap	0,0151	0,0482	0,279	5,65
Heap Binomiale	0,0266	0,13	0,558	1,97

Per l’analisi dei tempi di esecuzione degli algoritmi di visita è stata utilizzata la funzione PQSearchPerformance (file PQSearchPerformance.py). Il confronto è stato effettuato prima all’aumentare di nodi e archi in numero uguale (Grafico 1) e poi all’aumentare dei soli archi con numero di nodi costante (Grafico 2).

Nel primo caso, i tempi di esecuzione della visita aumentano esponenzialmente all’aumentare del numero di nodi/archi per ognuna delle code con priorità senza presentare troppe differenze (l’heap binomiale si dimostra leggermente più lento).

Nel secondo caso invece, i tempi di esecuzione della visita aumentano esponenzialmente all’aumentare del numero di archi (con numero di nodi costante), presentando valori molto più elevati a parità di archi rispetto al primo caso. I valori riportati per valori di archi grandi sono molto variabili in quanto dipendenti dalla selezione casuale degli archi da inserire (se l’arco già esiste non ne viene creato uno nuovo). In linea di massima però, il 5-Heap si dimostra più prestante dell’heap binario, mentre l’heap binomiale, che inizialmente impiega tempi più elevati, superato il valore di 400 archi (per il caso considerato), si dimostra sempre più prestante dei precedenti due heap.