Commento dei dati sperimentali

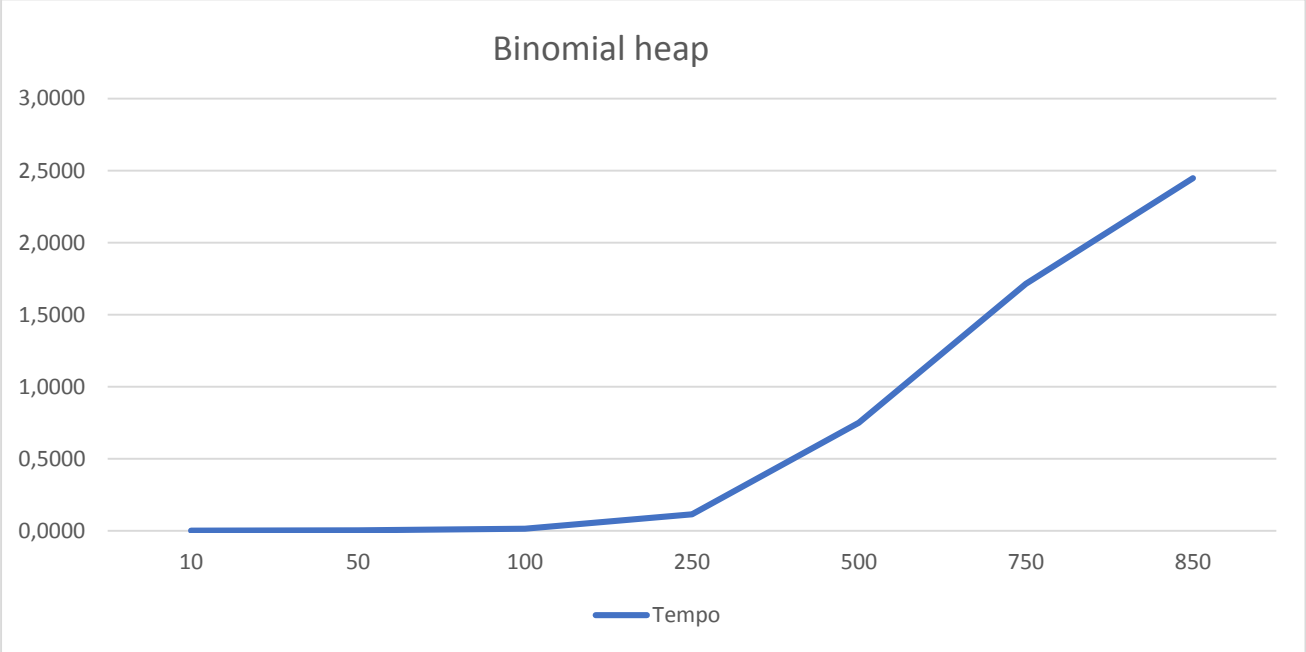
Gli esperimenti sono stato fatti usando un diverso numero di nodi e di archi. Il numero di nodi è stato scelto in maniera arbitraria, mentre il numero di archi per ogni nodo varia dal numero più piccolo possibile al più grande (per la spiegazione della scelta del numero di archi intermedi vedere il documento “Scelte Implementative”). Le code con priorità sono state modificate in modo da avere il nodo con peso massimo come radice dell’albero, e ciò è stato fatto senza cambiare i tempi teorici delle strutture dati che prevedono il minimo come radice.

Analisi dei dati ottenuti

Analizzando il tempo di esecuzione dell’algoritmo priorityVisit usando diverse code con priorità (Binomial Heap, Binary Heap e D-Heap) possiamo notare che il tempo è influenzato non solo da numero di nodi che vengono inseriti all’interno del grafico, ma anche dal numero di archi che possiede.
È evidente infatti che all’aumentare del numero di nodi e di archi, la priorityVisit impiega più tempo per essere completata, dovendo controllare più nodi all’interno della lista di adiacenza e verificare se questi si trovino o meno nella lista dei nodi marcati. Aumentare il numero di archi, invece, significa non avere un gran impatto sull’algoritmo quando il numero di nodi non è molto alto (fino a 250), ma superata una certa soglia, e avendo un numero di archi estremamente alto, si complica notevolmente la lista di adiacenza del grafo, che deve essere in ogni caso costruita e analizzata tutta per controllare se i suoi nodi sono stati o meno marcati precedentemente dalla visita.

Binomial Heap

I binomial Heap sono la struttura dati che impiega più tempo per ultimare la visita in priorità. Ciò è dovuto dal fatto che ogni volta che viene inserito un nuovo nodo al suo interno, se ci sono due alberi B(i) deve provvedere al merge dei due, che risulta comunque essere un’operazione abbastanza dispendiosa di tempo. All’aumentare del numero di nodi e di archi questa differenza di prestazioni risulta essere sempre meno accentuata.



D-Heap

Per quanto riguarda i D-Heap questi sono leggermente più prestanti del binomial heap, poiché non devono fare un merge di alberi, ma ad ogni inserimento deve essere fatto alla peggio un muoviAlto finché il nuovo nodo non incontrerà un nodo “padre” più grande di lui (oppure potrebbe diventare lui la nuova radice della struttura dati), o un muoviBasso quando viene cancellato il minimo . La struttura dati ad albero, però, risulta essere molto più efficiente per operazione di tipo search, rispetto all’insert o al delete. L’insert, infatti, potrebbe modificare di molto la struttura dati e ciò rallenta notevolmente il codice, mentre il search, mettendo una foglia in cima, deve poter scendere nella sua posizione corretta e confrontarsi quindi con i suoi D figli. Aumentando anche il numero D di figli che un nodo può avere non riusciamo ad avere miglioramenti significativi nel tempo di esecuzione del codice, poiché il numero di confronti da fare risulta essere in media il medesimo.