

Relazione Progetto Ingegneria degli Algoritmi 2019:

Francesco Lasco
Giorgio Liberatore

January 14, 2019

Abstract

Analisi sull'implementazione di un algoritmo per la visita dei grafi in base al peso di ogni vertice v , e delle conseguenti modifiche alle strutture di dati preesistenti utilizzate.

1 Introduzione

Lo svolgimento della traccia prevedeva l'implementazione dell'algoritmo `visitaInPriorità` (`prioritySearch` nel nostro codice Python) e la relativa modifica delle strutture di dati predisposte a gestire i Grafi.

2 Idea

Basandoci sull'algoritmo `visitaGenerica`, l'algoritmo `visitaInPriorità` parte a visitare dal vertice più pesante, v , del grafo \mathbf{G} e mantiene in una coda con priorità tutti i vertici adiacenti all'ultimo vertice visitato (esclusi i vertici già chiusi). l'algoritmo prosegue nella visita scegliendo il prossimo nodo con priorità più alta nella coda, visitandolo, rimuovendolo dalla coda e appendendo alla coda i suoi nodi adiacenti.

l'algoritmo termina allo svuotarsi della coda.

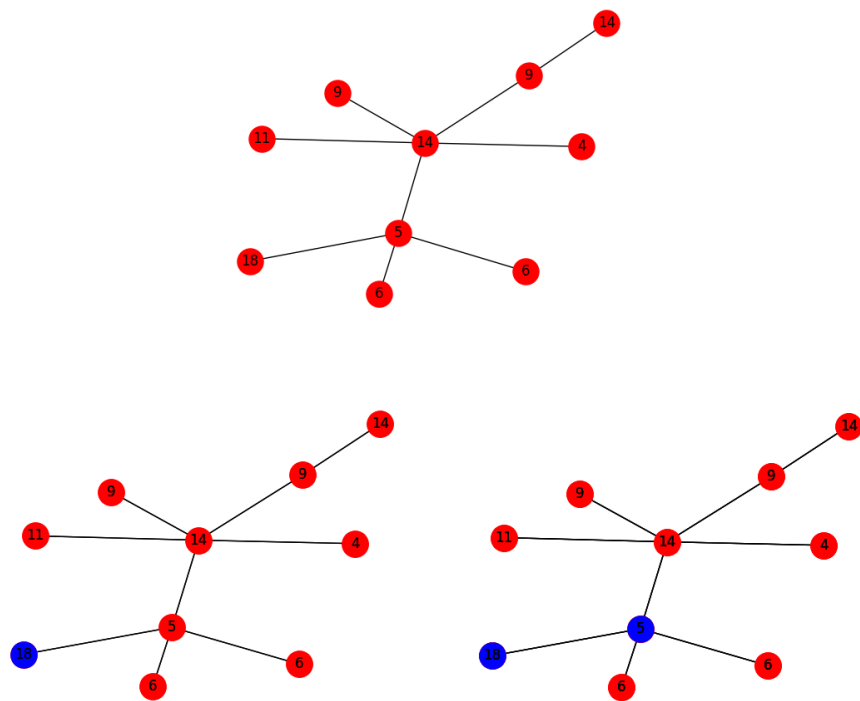
2.1 illustrazione dell'esecuzione di visitaInPriorità

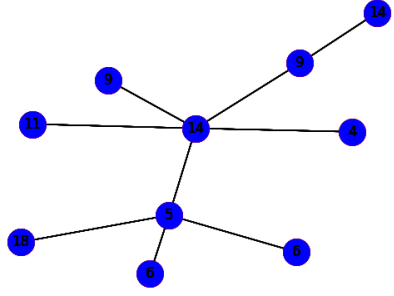
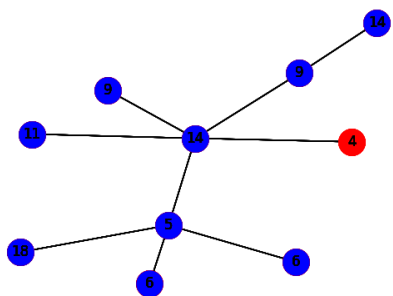
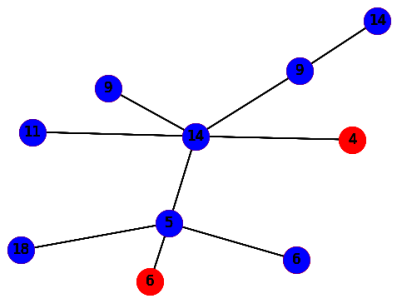
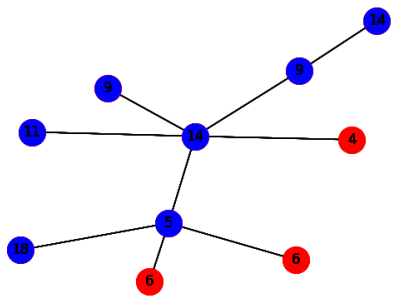
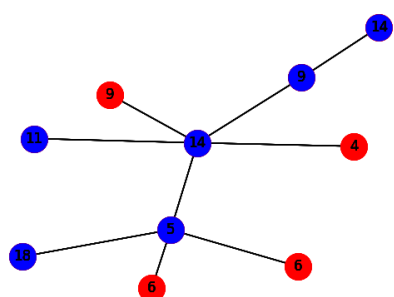
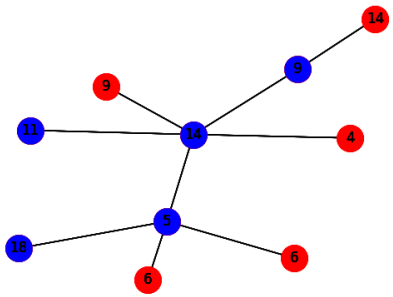
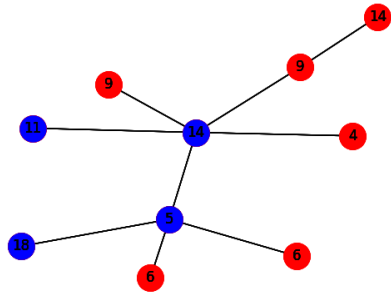
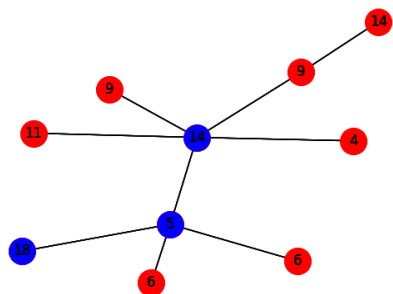
Nodo rosso = nodo non visitato.

Nodo blu = nodo visitato.

Valore nel nodo = Peso vertice.

(grafo generato casualmente, immagini ottenute tramite lo script `showGraph.py`.)





3 Scelte Implementative

3.1 Classi importate da ingegneria-algoritmi-2018

PQ_Dheap; PQbinaryHeap; PQbinomialHeap; Graph; GraphIncidenceList.

3.2 Classi e moduli esterni importati per l'analisi

random; networkx; matplotlib; time.

3.3 Descrizione Del Codice

modifiche alla classe Grafo:

La classe grafo è stata modificata in modo tale da permettere a ogni nodo di contenere oltre alle classiche informazioni (ID, valore) un attributo **weight** per gestire il peso dei vertici, come richiesto da consegna.

Il valore assegnato a weight è un intero non negativo; nei nostri test e nella nostra funzione per costruire grafi pesati sui vertici abbiamo scelto di assegnare il peso di ogni singolo vertice casualmente, prendendolo tra uno e due volte il numero dei vertici.

PRIORITYSEARCH:

```
-prioritySearch_PQbinaryHeap  
-prioritySearch_PQbinomialHeap  
-prioritySearch_PQ_DHeap  
T = exploredNodes  
F = PriorityQueue
```

marcamento vertici = lista **enqueuedVertices**.

Questi metodi implementano la **visitaInpriorità**.

Inizialmente si trova il vertice con peso massimo nel grafo tramite il metodo **maxWeight**, sarà il vertice dal quale far partire la visita, lo si inserisce nella **priorityQueue** con peso negativo (per avere la priorità più alta nella coda basata sul minimo) e nella lista degli ID già visitati. Finché la **priorityQueue** non è vuota si estrae il vertice con priorità massima cancellandolo e inserendolo nella lista dei nodi visitati, successivamente per ogni vertice adiacente viene effettuato un controllo per l'inserimento nella coda. Il risultato della visita ritornato nella lista **exploredNodes**.

La differenza fondamentale tra le tre implementazioni dell'algoritmo sta nella scelta del tipo di coda con priorità da utilizzare.

Analizzeremo più avanti eventuali benefici o svantaggi delle tre possibili scelte.

__init__: Viene chiamata la super, riprendendo il costruttore della classe `GraphIncidenceList`.

isEmpty: Questo metodo viene utilizzato per controllare se il grafo è vuoto.

isFull: Questo metodo è stato implementato per capire se il grafo ha già raggiunto il numero massimo di archi, $(N*N-1)$.

maxWeight: Questo metodo è stato implementato per ritornare una lista con id e peso del vertice con peso massimo. Si scorrono tutti i nodi e viene manenuto un massimo (inizialmente **None**), se il nodo in esame è diverso da **None** e il peso è superiore al massimo, diventa il nuovo massimo.

buildGraph: Questa è la funzione che permette di creare un grafo connesso e pesato sui vertici pronto per eseguire una **visitaInpriorità**. Il grafo avrà $2*(numeroVertici-2)$ archi poich il primo vertice non ne creerà, mentre dal secondo in poi verranno scelti dei nodi a caso a cui aggiungere un arco.

addVertices: Questa funzione permette di aggiungere vertici ad un grafo. Se il grafo è vuoto viene aggiunto un vertice (inizialmente non connesso). Poi vengono aggiunti altri $n-1$ vertici con archi connessi in maniera casuale. Se il grafo non è vuoto viene recuperata la lista di vertici per le connessioni casuali degli archi e per stabilire il peso massimo assegnato ad ogni vertice (tra 1 e $2 * \#(V)$, con V = insieme dei vertici di G). Il grafo viene modificato in-place quindi non ci sono parametri di ritorno.

addEdges: Questa funzione è stata implementata per aggiungere archi in un grafo. I due vertici da collegare vengono scelti tramite due funzioni `lambda`: la prima sceglie un vertice che non ha già esaurito i possibili collegamenti. La seconda sceglie il vertice in modo che non ci siano archi degeneri (da un arco a se stesso) nè un collegamento già esistente.

4 Risultati Sperimentali

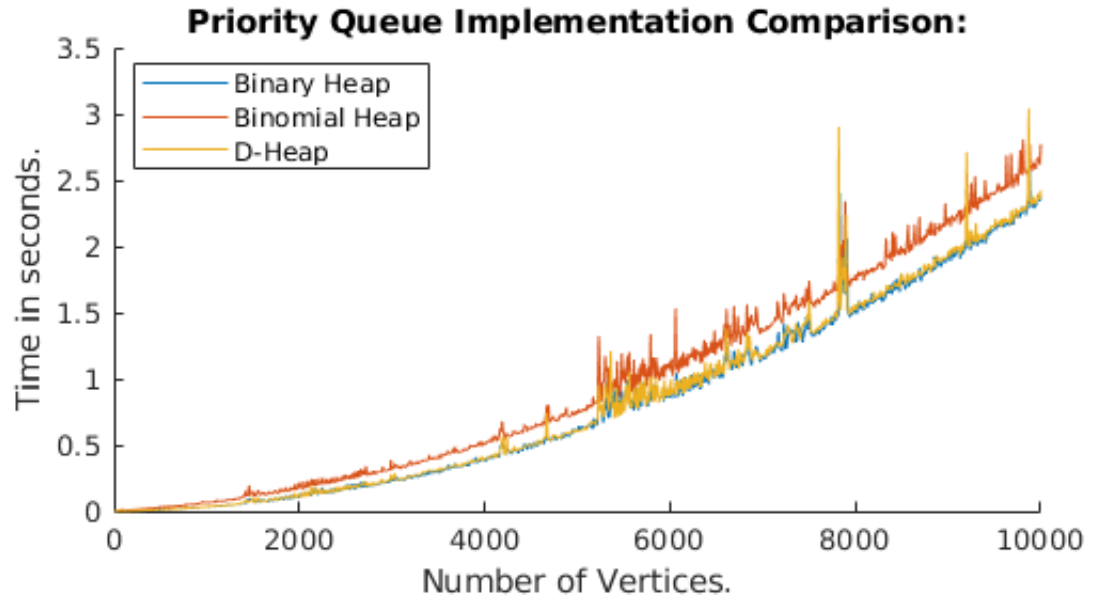


Figure 1: Grafico del tempo necessario per una `prioritySearch` al variare dell'implementazione di `priorityQueue` e al crescere di V .

`prioritySearch` su Grafo con vertici crescenti.

Numero Vertici	Tempo <code>binaryHeap</code> (sec.)	Tempo <code>binomialHeap</code> (sec.)	Tempo <code>D-Heap</code>
1.000	0.03451132774353027	0.06687092781066895	0.03332638740539551
2.500	0.16240239143371582	0.27475476264953613	0.1934661865234375
5.000	0.613001823425293	0.7367110252380371	0.5978813171386719
7.500	1.5704772472381592	1.7399401664733887	1.593226432800293
10.000	2.388897657394409	2.7697293758392334	2.424243688583374

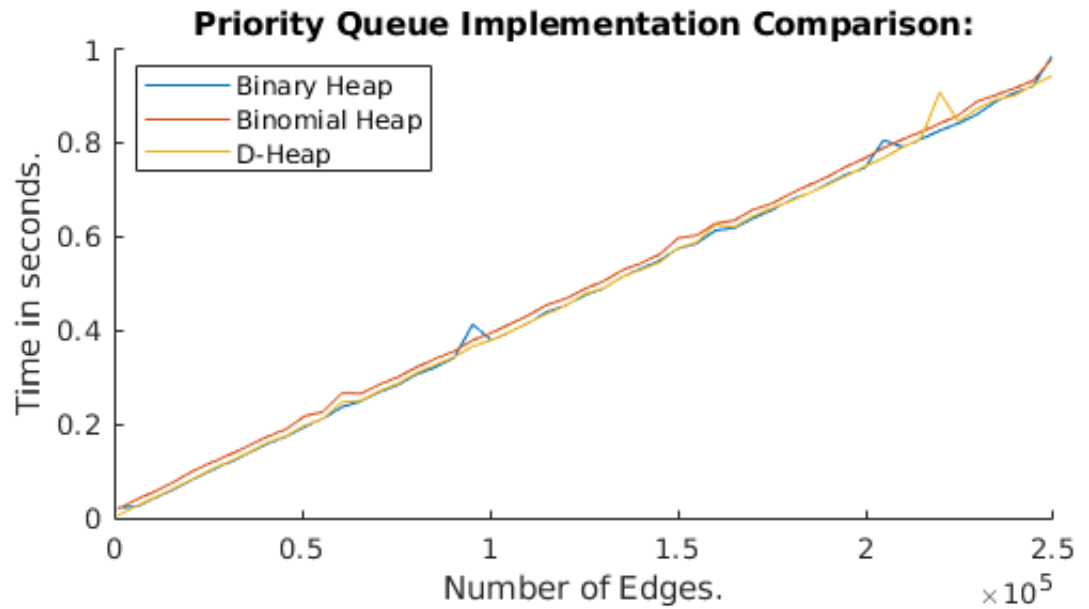


Figure 2: Grafico del tempo necessario per una prioritySearch al variare dell'implementazione di priorityQueue e al crescere di E.

prioritySearch su Grafo di 500 vertici e archi crescenti.

Numero Archi	Tempo binaryHeap (sec.)	Tempo binomialHeap (sec.)	Tempo D-Heap
998	0.024201631546020508	0.021246671676635742	0.007825374603271484
50.698	0.19614839553833008	0.21862459182739258	0.19875836372375488
100.398	0.3802378177642822	0.3956644535064697	0.3807394504547119
125.248	0.4761958122253418	0.48966145515441895	0.47887587547302246
174948	0.6570134162902832	0.6704854965209961	0.6607739925384521
249498	0.9841418266296387	0.9787428379058838	0.9430813789367676

Per i test effettuati si nota che la crescita temporale al crescere del numero di **vertici** è simile per ognuna delle tre possibili implemetazioni di **priritySearch**. È inoltre possibile notare che in caso di picchi, dovuti a particolarità del grafo G, l'implementazione tramite D-Heap (con D pari a 10 per i nostri test), che in media ha prestazioni leggermente migliori rispetto alle altre implementazioni ha una crescita temporale che la porta a superare quella di tutte le altre implementazioni.

Per quanto riguarda l'analisi al crescere del numero di **archi**, notiamo che la crescita temporale è lineare e non ci sono particolari differenze dovute alle scelte implementative della coda con priorità.

5 Conclusione e commento

Le implementazioni delle code con priorità hanno prestazioni simili applicate alla struttura di dati studiata.

Il progetto ci ha dato modo di approfondire l'implementazione dei Grafi e degli algoritmi a loro correlati, le strutture di coda con priorità e le loro possibili applicazioni.