A.A. 2018/2019

SECONDO PROGETTO IN ITINERE INGEGNERIA DEGLI ALGORITMI

ALESSANDRO MONTENEGRO 0252225
MARCO CALAVARO 0252844

INDICE

Nella relazione si assume che n sia il numero di vertici del grafo, mentre m il numero degli archi dello stesso.

I test sono stati effettuati su un Dell XPS 15 9560 che monta un Intel Core i7 7700HQ ed esegue Ubuntu Mate 18.04.

SCELTE IMPLEMENTATIVE

Nel codice abbiamo scelto di rappresentare i grafi mediante liste di adiacenza in quanto, per le operazioni richieste, l'implementazione tramite liste di archi o matrici, sia di adiacenza che incidenza, faceva peggiorare notevolmente il tempo di esecuzione. Ad esempio, per la sola visita DFS, su cui si basa uno degli algoritmi di ricerca del ciclo, con la nostra scelta si ottiene un tempo, nel caso peggiore, di O(m+n), mentre con le altre due opzioni avremmo ottenuto rispettivamente O(m*n) e $O(n^2)$.

Per poter implementare le funzioni *hasCycleDFS* e *hasCycleUF* abbiamo sfruttato il codice visto a lezione, più specificatamente, ci siamo serviti delle classi *GraphAdjacencyList* e *QuickFindBalanced*. In particolare, abbiamo scelto di implementare la struttura *Union Find* con il *Quick Find* con bilanciamento poiché nel codice vengono richieste più operazioni *Find*, che con la scelta fatta richiedono O(1), mentre le *Union*, in numero inferiore, richiedono O(logn) in virtù del bilanciamento. Inoltre, abbiamo apportato delle modifiche alle suddette classi che spiegheremo nelle prossime righe, insieme alle funzioni che abbiamo creato per lo svolgimento del progetto.

MODIFICHE GRAPH ADJACENCY LIST

insertEdgeNO(self, tail, head, weight=None)

Questa funzione consente l'inserimento di un arco in un grafo non orientato. La sua creazione è stata necessaria poiché per l'inserimento di un arco, è stato definito nella classe il metodo *insertEdge(self, tail, head, weight=None)* utile per l'implementazione di un grafo non orientato, infatti, nella lista di adiacenza di *tail* veniva aggiunto il nodo *head*, mentre la lista di questo non veniva toccata. La nostra funzione, del tutto identica a quella già presente nella classe, aggiunge anche *tail* nella lista di adiacenza di *head*, così da ottenere un grafo non orientato.

trovaArchi(self)

La procedura si differenzia dalla funzione preesistente *getEdges* in quanto restituisce la lista degli archi del grafo, non come oggetti *Edge*, bensì come tuple in cui il primo valore risulta essere la *head* dell'arco, il secondo la *tail*. La funzione, creata per lavorare al meglio con grafi non orientati, inserisce nelle suddette tuple gli *id* dei nodi.

MODIFICHE UNION FIND

L'unica modifica attuata nella classe *QuickFindBalanced* consiste nell'aver aggiunto un *return son* nel metodo *makeSet*.

CREAZIONE DEL GRAFO

makeGraph(n, bool)

La funzione serve a creare un grafo, rappresentato mediante liste di adiacenza, con n nodi. Se il valore di *bool* risulta 1, allora il grafo avrà un ciclo, se invece *bool* è 0 il grafo creato sarà aciclico. Nella prima parte del codice viene creato il grafo vuoto *graph*, successivamente vengono inseriti tutti i nodi, utilizzando il metodo *insertNode* della classe *GraphAdjacencyList,* che poi vengono collegati in modo sequenziale tramite un ciclo for e la funzione, prima definita, *insertEdgeNO*. Il risultato sarà un grafo connesso, non orientato, non pesato e aciclico. A questo punto viene effettuato il controllo su *bool*, se esso è 1 allora viene generato randomicamente un intero grazie al quale si può selezionare, dalla lista di nodi, un vertice che verrà in fine collegato con un altro nodo distante di un livello. Per come è definita la funzione fino a questo punto, così facendo si genera un ciclo. Con una piccola modifica, scegliendo un intero randomico, potevamo inserire più cicli nel grafo, tuttavia, abbiamo scelto di inserirne uno solo per verificare la correttezza degli algoritmi, in quanto, ai fini del lavoro svolto ne bastava uno.

ALGORITMI PER VERIFICARE LA PRESENZA DEL CICLO

hasCycleDFS(graph)

Questa è la prima funzione che consente di verificare che in un grafo non orientato, passato come parametro, ci sia un ciclo o meno. Essa è una modifica della visita DFS definita nella classe *GraphBase*, infatti è molto simile, ma presenta alcune varianti. Innanzitutto, non ritorna la lista *dfs_nodes*, ma un valore booleano, più specificatamente *True* se è presente un ciclo nel grafo, *False* altrimenti. Oltretutto, tale funzione differisce dall'algoritmo DFS anche per il fatto che gli viene passato come parametro direttamente tutto il grafo, scegliendo nelle prime righe di codice il nodo sorgete da cui partire con la visita casualmente. Inoltre, nel ciclo *while* abbiamo definito la variabile di controllo *k*, essa viene inizializzata a 0 nel ciclo e può raggiungere come valore critico 2, in tal caso nel grafo è presente un ciclo. Tale modifica è dovuta alla creazione del metodo per inserire un arco non orientato di cui abbiamo parlato sopra. Infatti, è del tutto normale che nelle liste di adiacenza di due nodi adiacenti siano presenti entrambi, viene trovato un ciclo quando esiste anche un altro nodo adiacente ad uno che è già adiacente con un altro. Il valore di controllo *k*, viene incrementato di un'unità quando l'algoritmo trova la presenza di un nodo già esplorato nella lista di adiacenza di un altro vertice, se viene incrementato ulteriormente assume il valore 2 e vuol dire che è presente un ciclo nel grafo per i motivi di cui sopra.

hasCycleUF(graph)

La funzione *hasCycleUF* utilizzata per trovare il ciclo all'interno del grafo G passato come parametro sfrutta una struttura dati di tipo *UnionFind*. In particolare, *QuickFindBalanced* che viene inizializzata con il nome *uf*. In seguito, vengono create una lista e un dizionario per poter mantenere traccia dei nuovi nodi e archi, infatti, in un certo senso il grafo dovrà essere tradotto nella suddetta struttura *uf*. Fatto ciò, mediante un *iterator*, così come richiesto dalla consegna, vengono scanditi gli *edges* 'tradotti'

del grafo controllando che i due nodi appartengano allo stesso insieme, ciò indicherebbe la presenza di un ciclo e l'algoritmo restituirebbe *True*, oppure, in caso contrario, si esegue una union tra i due elementi dell'arco. Completando la scansione degli archi senza trovare uguaglianze viene restituito *False* in quanto non è stato trovato alcun ciclo.

TESTING

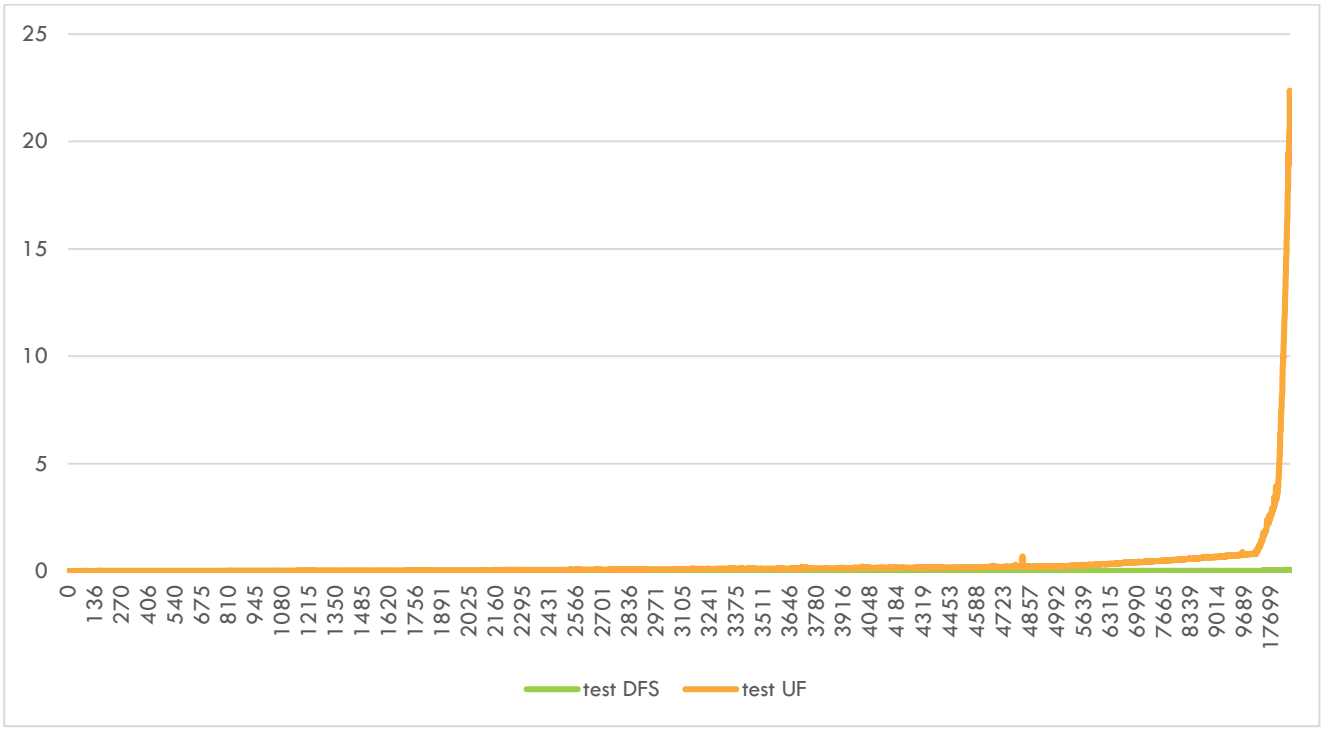
Nella fase di testing ci siamo serviti di alcune funzioni da noi definite: *listSomma, test_and_csv*, *testUF* e *testDFS*.

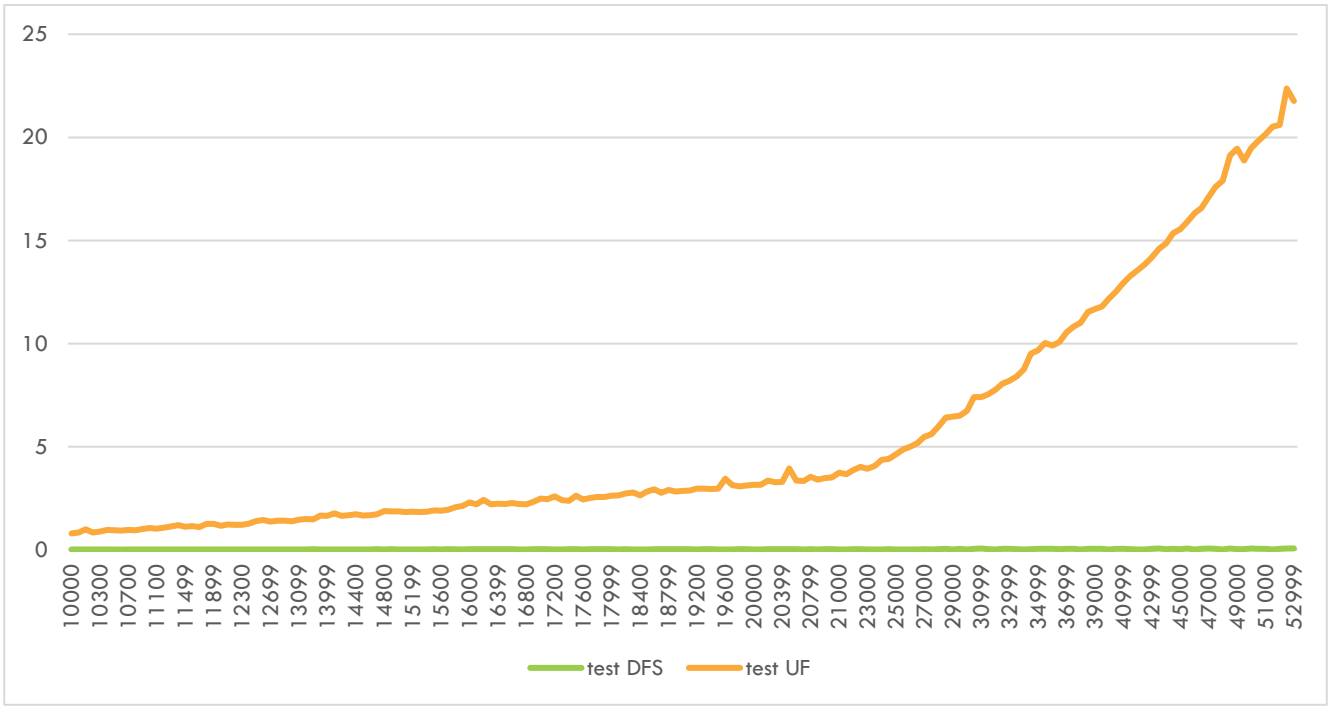
La funzione *listSomma* prende in input una lista e ne restituisce la somma dei valori, verrà impiegata nei test in quanto faremo ripetere la funzione da analizzare diverse volte registrandone i tempi di esecuzione su un dato input per poi andarne a fare una media, così da ottenere una misura più indicativa.

La procedura *test_and_csv* è in realtà un *decorator*. Esso fa ripetere più volte la funzione su un dato input, misurandone sempre il tempo di esecuzione che ogni volta andrà ad arricchire la lista, inizialmente vuota, *time_list*. Una volta terminato il ciclo *for* viene calcolato il tempo medio sfruttando la funzione prima illustrata. Infine, si scrive il risultato su un file *.csv*, avente come nome il nome stesso della funzione e nel formato '*numero_archi, tempo*'. Il *decorator* appena descritto è stato creato per le funzioni *testUF* e *testDFS* che prendono in input il grafo su cui eseguire la ricerca del ciclo, il numero degli archi (args[1]) e il numero di ripetizioni da effettuare della funzione (args[2]), ed eseguono rispettivamente *hasCycleUF* e *hasCycleDFS*.

Nella *main* si fa variare *n*, il numero dei nodi di cui sarà composto il grafo preso in esame, mediante un ciclo *for* al cui interno ad ogni passo viene creato il grafo *g*, che conterrà casualmente un arco in virtù della funzione *randint*, su cui verrà poi effettuato il test di *hasCycleUF* e *hasCycleDFS*. Fino a 5000 nodi abbiamo tenuto il passo di incremento della funzione *range* a 1 e il valore di *repeat* a 5, in seguito, fino ad n=10'000 abbiamo modificato l'incremento a 5, successivamente esso è stato tenuto a 100 fino a 20'000 nodi e a 500 fino a 50'000 vertici, diminuendo però il valore di *repeat* a 2.

GRAFICI





Dai seguenti grafici si evince che l'algoritmo di ricerca del grafo più efficiente è *hasCycleDFS*. La prima differenza sostanziale appare quando il numero di archi del grafo è circa 10'000.