RELAZIONE PROGETTO 2

Luca Santopadre: 0257118 Michele Salvatori: 0253519

hasCycleUF ->

Struttura dati utilizzata: *QuickFindBalanced.*
L'algoritmo memorizza in un albero T *QuickFind con bilanciamento sulle union* il grafo G ed effettua per ogni arco *(u,v)* restituito dall'iteratore:

- 2 ***findRoot*** su T, una per vertice dell'arco, e restituisce ***true*** se le radici dei due vertici sono uguali (ciò prova l'esistenza di un ciclo in G)

- altrimenti esegue una ***union(u,v)*** su T

hasCycleDFS ->

Dato un *grafo G connesso*, è necessaria solamente una visita DFS, che può partire da un qualsiasi nodo, per segnalare o meno l'eventuale presenza di un *arco all'indietro* e quindi l'esistenza di un ciclo nel grafo G.

edgeIterator ->

Implementato in *hasCycleUF* permette di scorrere la lista di archi escludendo quelli già marcati. Infatti, essendo G un grafo non orientato nell'inserimento di un nuovo arco verrà inserito anche l'arco opposto.
A tale scopo nel metodo ***next()*** dell'iterator è presente un controllo sull'arco da restituire e la marcatura in una lista di tale arco.

graphUtility ->

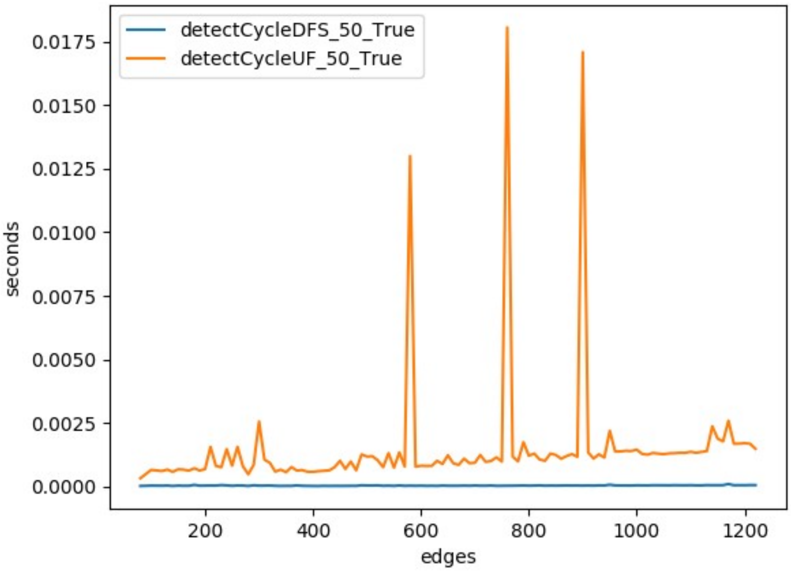
createGraph: realizza con e senza cicli. Nel primo caso il ciclo sarà presente in una posizione randomica del grafo. Ciò permette di effettuare test più veritieri.

Essendo createGraph un generatore di grafi totalmente randomici, la funzione *isConnected* verifica se un grafo è connesso o meno. Eseguendo infatti una semplice DFS da un qualsiasi nodo si può facilmente controllare se la lista dei nodi visitati dalla visita in profondità contiene o meno tutti i nodi del grafo. Se ciò non accade il grafo non sarà connesso e quindi verrà scartato nella fase di testing.

ANALISI SPERIMENTALE

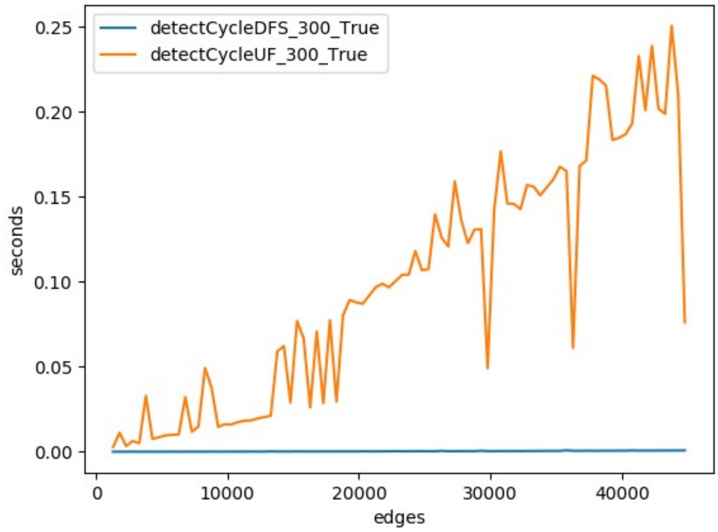
Tutti i test sono stati effettuati su una macchina dotata di sistema operativo Linux e SSD.
Nella visualizzazione grafica dei risultati il tempo è espresso in secondi. Sull'asse delle ascisse troviamo la dimensione del grafo.
I test sono stati effettuati solamente su grafi connessi, tenendo costante il numero dei nodi e facendo variare progressivamente il numero di archi, ovvero il numero di cicli presenti all'interno.
Tutti i risultati sono comunque reperibili, sia in forma grafica che in formato *.csv*
nella directory: (…/results).
A tale scopo è stato utilizzato un *decorator* (*timer* in …./demo/test.py).

nodes:50 cycle:True		
edges	hasCycleUF	hasCycleDFS
80	0,0003216	0,0000193
100	0,0006509	0,0000329
150	0,0006750	0,0000336
250	0,0008156	0,0000262
400	0,0005744	0,0000193
500	0,0011773	0,0000391
600	0,0008152	0,0000308
700	0,0009332	0,0000379
800	0,0012088	0,0000310
900	0,0170853	0,0000329
1000	0,0014508	0,0000451
1100	0,0013652	0,0000503
1200	0,0017028	0,0000489



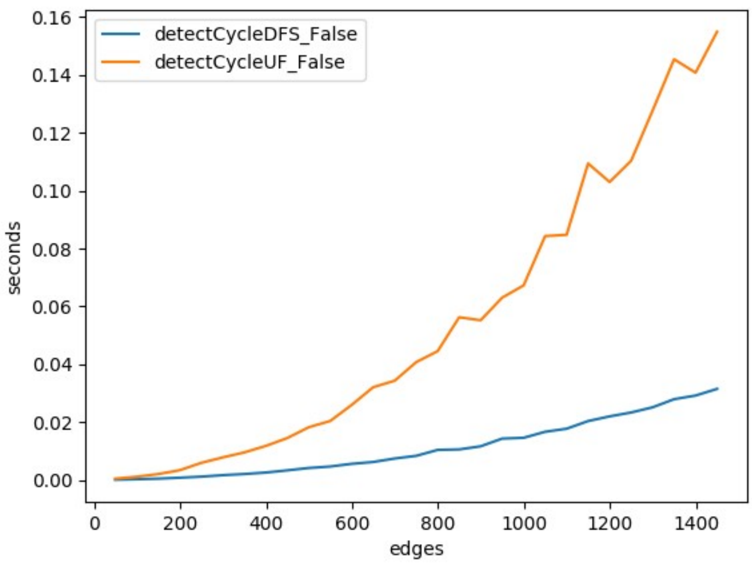
1. *Confronto dei tempi di esecuzione dei due algoritmi al variare del numero di archi su un grafo di 50 nodi contenente cicli.*

Nodes:300 cycle:True		
edges	hasCycleUF	hasCycleDFS
1300	0,0027125	0,0000255
4300	0,0075941	0,0000334
8300	0,0493279	0,0000827
10300	0,0161068	0,0001087
14300	0,0621555	0,0001485
18300	0,0295818	0,0002375
20300	0,0870156	0,0002959
25300	0,1073680	0,0003290
30300	0,1423876	0,0004041
35300	0,1676161	0,0005388
38300	0,2188740	0,0006087
40300	0,1867206	0,0006542
44300	0,2098978	0,0007775



2. Confronto dei tempi di esecuzione dei due algoritmi al variare del numero di archi su un grafo di 300 nodi contenente cicli.

cycle:False		
edges	hasCycleUF	hasCycleDFS
49	0,0004966	0,0001369
99	0,0011559	0,0003111
199	0,0034220	0,0008209
299	0,0078444	0,0017114
399	0,0118067	0,0026081
499	0,0182612	0,0041769
599	0,0259891	0,0056212
699	0,0343511	0,0074596
899	0,0552311	0,0116823
999	0,0672877	0,0146122
1199	0,1030066	0,0220177
1299	0,1276512	0,0251448
1399	0,1407945	0,0292146



3. Confronto dei tempi di esecuzione dei due algoritmi al variare del numero di nodi, su un grafo connesso da n-1 archi, senza cicli.

CONSIDERAZIONI FINALI

La scelta di implementare una UF non risulta molto efficiente perché l’algoritmo scandisce ogni arco del grafo G, dove il numero degli archi ,può essere nel caso peggiore m= O(n^2). La presenza dei “*picchi*” nei grafici è dovuta alla posizione del ciclo nel grafo che è totalmente casuale. Quindi dato che l’algoritmo non fa una selezione “intelligente” degli archi, nel caso peggiore, va a scandire un numero m (molto maggiore di n) di archi.
Tuttavia si registra un *andamento crescente* dei tempi di esecuzione.

Mentre eseguendo una semplice visita in profondità su un grafo connesso non orientato abbiamo la certezza che l’albero ritornato dalla DFS ha al più n-1 archi. Quindi nel caso peggiore ci fermeremo dopo aver scandito *n* archi e trovare un ciclo, oppure scandire n-1 archi, n nodi ed essere sicuri che G sia aciclico.