Seconda prova pratica in itinere

Ingegneria degli Algoritmi 2018/2019

Progetto : **2** Tutor di riferimento : **Ovidiu Daniel Barba**

Studente : Simone Festa, matricola **0251679**

Creazione del grafo

Per la creazione di un grafo viene implementata la funzione createGraph(), avente come argomenti il numero di nodi ed un flag che indica la presenza (o assenza) di un ciclo. Vengono quindi aggiunti gli indici dei nodi in una lista, la quale verrà mescolata per evitare di avere sempre uno stesso grafo lineare. Ogni nodo viene collegato al nodo successivo, eseguendo anche l'operazione contraria, poiché viene richiesto che il grafo sia non orientato. Nel caso si volesse utilizzare un grafo ciclico, basterà collegare il primo elemento e l'ultimo elemento del grafo, creando così un grafo ciclico. I grafi sono rappresentati come liste di adiacenza.

```
Adjacency Lists:
0:[4]
1:[5, 3]
2:[3, 4]
3:[1, 2]
4:[2, 0]
5:[1]
```

Algoritmo hasCycleUF

Per implementare tale algoritmo è importante osservare che lavorare con archi non orientati può rendere più difficile il lavoro. Infatti, data la richiesta di implementazione con UnionFind, troveremo sempre un ciclo, poiché ad esempio, un semplice grafo " A <———>B", viene visto come un arco da A a B e viceversa.
E' allora importante copiare solo uno di questi due archi. Nell'algoritmo è presente un controllo che inserisce in una lista edgesOriented (cioè archi orientati) gli archi non ancora analizzati e che non abbiano l'arco complementare già presente. Per ogni arco, viene eseguito un confronto con gli archi presenti in edgesOriented : l'arco non viene inserito se è già presente, o è presente il complementare.
Otteniamo quindi una nuova lista avente archi idonei alle operazioni con UnionFind.
Altro step importante è la scelta dell'implementazione di UnionFind.
Nella tabella sono presenti i confronti eseguiti su vari numeri di nodi:

Grafo ACICLICO	N = 100	N = 400	N = 1000
QuickFind(), NO ciclo	0.0050640106201171875	0.03885984420776367	0.24623299598693848
QuickFind(), NO ciclo	0.0027132034301757812	0.03797626495361328	0.23825621604919434
QuickFindBalanced(), NO ciclo	0.0031290054321289062	0.03870677947998047	0.24768996238708496
QuickUnionBalanced(), NO ciclo	0.0055999755859375	0.03915691375732422	0.23922481727600098
Grafo CICLICO	**N = 100**	**N = 400**	**N = 1000**
QuickUnion(), SI ciclo	0.003003835678100586	0.04405713081359863	0.24794602394104004
QuickFind(), SI ciclo	0.0028069019317626953	0.04111886024475098	0.2475738525390625
QuickFindBalanced(), SI ciclo	0.003072023391723633	0.038166046142578125	0.2511909008026123
QuickUnionBalanced(), SI ciclo	0.004094839096069336	0.038021087646484375	0.2782158851623535

Come si evince dai risultati, l'implementazione della QuickFind() risulta più efficiente della quickUnion().
Le euristiche per l'ottimizzazione di QuickFind e QuickUnion non apportano miglioramenti.
Implementeremo allora il QuickFind().
Ottenuta una lista di archi non orientati, eseguiamo per ciascun arco una makeSet. Iterando tali archi, preleviamo i nodi che compongono l'arco (identificati da tail e head). Se l'operazione di find eseguita sugli elementi identificati da tail e head restituisce vero come risultato, allora è presente un ciclo. Se ciò non viene verificato, allora l'algoritmo eseguirà una operazione di union, procedendo con l'arco successivo.
hasCyleUF implementa inoltre un iterator per scandire gli archi del grafo. E' infatti definita la classe EdgeIter(), la quale, avente una lista con ciascun elemento definito da un identificativo (il quale parte da 0) prima verifica se siamo arrivati a fine lista, con annesso stop del ciclo; altrimenti prende il valore nella lista attuale, sul quale verrà eseguito il comando "return" successivamente, mentre viene incrementato l'identificatore per la prossima iterazione.

Algoritmo hasCycleDFS

In questo algoritmo deve essere implementata una visita in profondità DFS per verificare la presenza di un ciclo. Per realizzare ciò, basti sapere che con una visita DFS si conferma la presenza di un ciclo se è presente un arco all'indietro.
La visita in profondità utilizza una Stack, con operazioni di push e pop. In particolare, per ogni nodo vengono ricercati ed aggiunti alla Stack i nodi adiacenti. Notiamo che, eseguendo la DFS applicata al grafo ciclico creato in GraphAdjacencyList, la lista dfs_nodes inserirà due volte lo stesso nodo. Questo perché, dati ad esempio 3 vertici A,B,C collegati, partendo dal vertice A, esso viene aggiunto all'elenco dei nodi esplorati, per poi cercare nodi adiacenti (B e C). Nella stack avremo in testa B, che verrà esplorato, e troverà come nodi adiacenti A (già esplorato) e C (non ancora esplorato, poiché nella Stack non ha priorità massima). Allora nella Stack aggiungeremo due volte il nodo C, che avrà tutti i nodi adiacenti esplorati (quindi non vengono aggiunti altri elementi) e l'unica operazione da eseguire sarà 'pop', rimuovendo l'elemento dalla Stack ed aggiungendolo alla lista dfs. Consapevoli di ciò, basta allora verificare che la presenza di un ciclo mediamente DFS è dettata dalla presenza di un doppio elemento nella lista dfs. hasCycleDFS restituirà la presenza di un ciclo se, aggiungendo un elemento all lista dfs, questo sarà già presente, altrimenti lo aggiungerà procedendo con l'algoritmo.

```
[0, 3, 5, 1, 4, 2]
```

dfs_nodes in caso di grafo ACICLICO

```
[0, 5, 1, 4, 3, 2, 2]
```

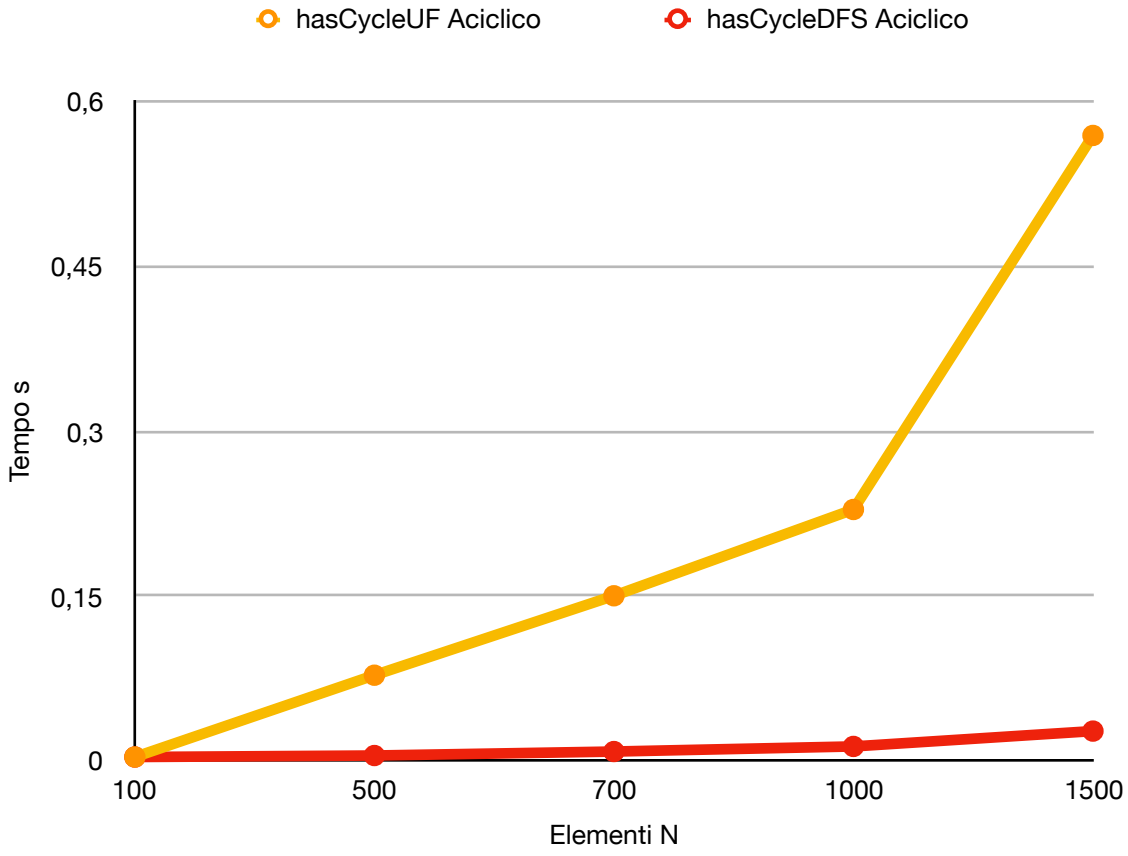
dfs_nodes in caso di grafo CICLICO

ALTRE INFORMAZIONI

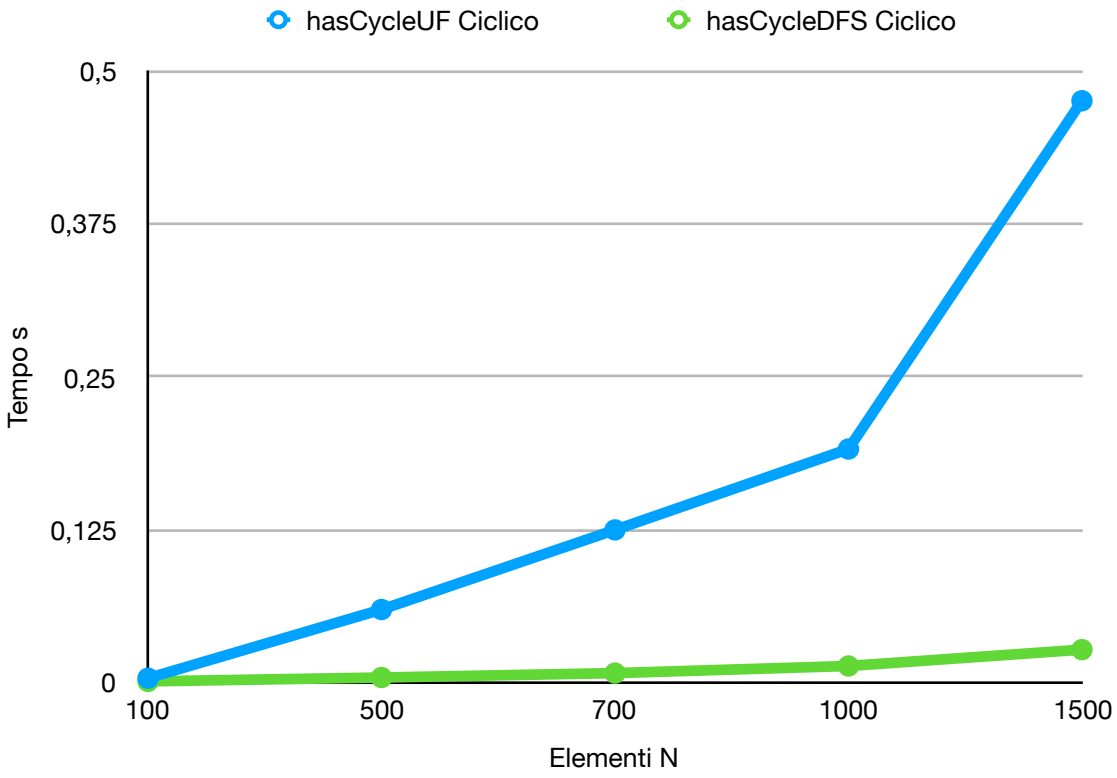
Su entrambi gli algoritmi è stato implementato un decoratore "DecoratoreStampa". Come da richiesta, esso aggiunge in formato n,t (numero archi, tempo) le informazioni sugli esperimenti eseguiti. Viene calcolato per ciascun algoritmo il tempo di esecuzione. Successivamente verrà aperto in 'append' un file (per avere tutti i risultati a disposizione) su cui vengono aggiunte tempistiche ed informazioni sul lavoro. Si conclude chiudendo il file.

CONFRONTO

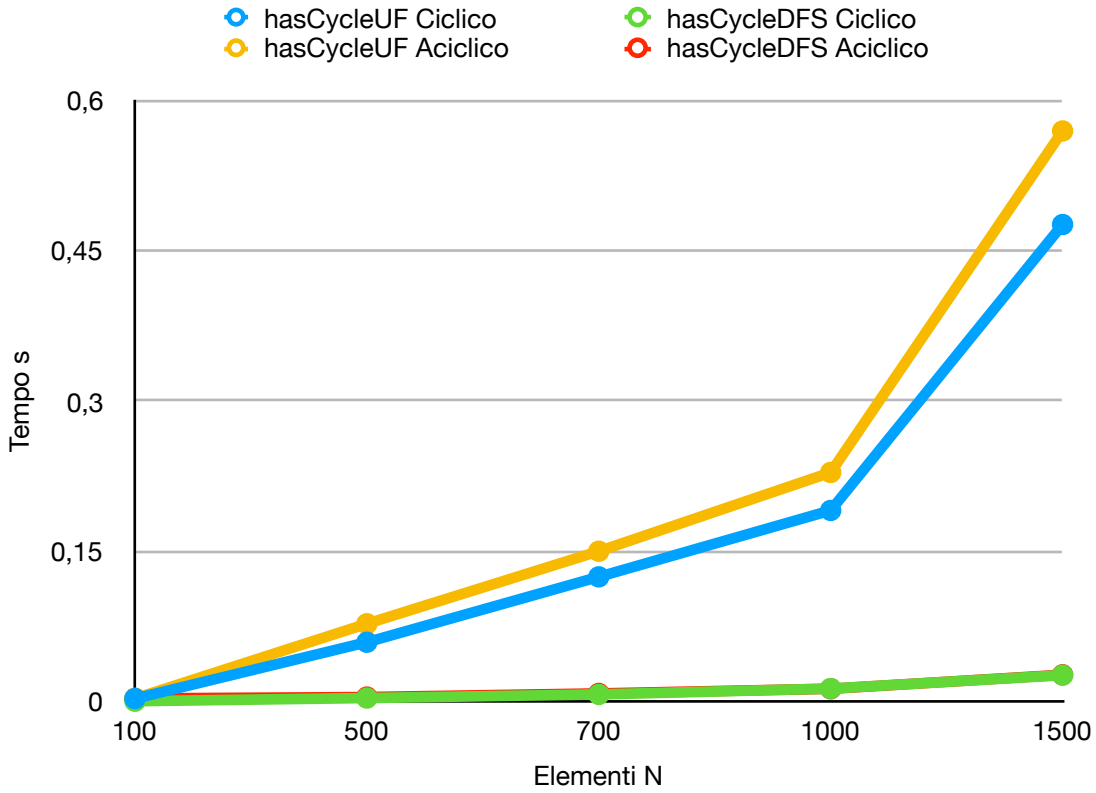
Il confronto risulta favorire l'algoritmo hasCycleDFS, il quale presenta dei tempi di esecuzione davvero ridotti. hasCycleUF non demerita, ma non riesce a raggiungere le tempistiche ottenute con la visita DFS. Probabilmente tale differenza è data dal fatto che hasCycleUF, per come è stato implementato, richiede di trovare archi orientati e di aggiungerli ad una lista, eseguendo confronti che probabilmente potrebbero essere eseguiti in modo più efficiente. Anche considerando le ottimizzazioni possibili, risulta difficile pensare che possa eguagliare hasCycleDFS, il quale ha tempi veramente ridotti e spesso difficilmente recuperabili. Studiando però i grafi, sappiamo bene che l'uso della visita in profondità su liste di adiacenza risulta essere un modo ottimale per la presenza di un ciclo. Di seguito vengono elencati alcuni risultati ottenuti con un grafico esplicativo.



Grafi ACICLICI	N =100	N =500	N = 700	N =1000	N = 1500
hasCycleUF()	0,003301858901977539	0,07782101631164551	0,15018701553344727	0,22880792617797852	0,569598913192749
hasCycleDFS()	0,000370025634765625	0,004642963409423828	0,008270263671875	0,012928962707519531	0,026905059814453125



Grafi CICLICI	N =100	N =500	N = 700	N =1000	N = 1500
hasCycleUF()	0,003125905 99060059	0,059376955 0323486	0,12461781501	0,190913915 634155	0,476238965 988159
hasCycleDFS()	0,000515937 805175781	0,003683090 20996094	0,007287025 45166016	0,013118028 6407471	0,026510953 9031982



Come possiamo osservare, il caso più lento è quando eseguiamo hasCycleUF con un grafo aciclico. Questo è un risultato che poteva essere previsto, poiché avendo implementato la QuickFind(), l'operazione più onerosa è l'esecuzione della Union, che verrà eseguita sempre in caso di grafo senza ciclo.