Progetto 2: Relazione

•Introduzione

Tutti i test sono stati eseguiti su una macchina con processore Intel® Core™ i7-8550U e 8 GB di memoria RAM, con kernel Linux stable (4.20.2), tramite l'ambiente di sviluppo PyCharm di JetBrains.

Il codice è stato caricato su GitHub al link https://github.com/Lorenzoval/final-algoritmi

La commit history è stata mantenuta il più pulita possibile, in modo tale da poter osservare con chiarezza ogni singola modifica effettuata. Il codice riguardante il progetto si trova nella cartella project, fatta eccezione per una singola modifica al codice dei grafi implementati con matrici di adiacenza (commit 7bd599b). Tutto il codice è commentato e molti dettagli spiegati nei commenti e nelle docstrings sono stati omessi dalla relazione.

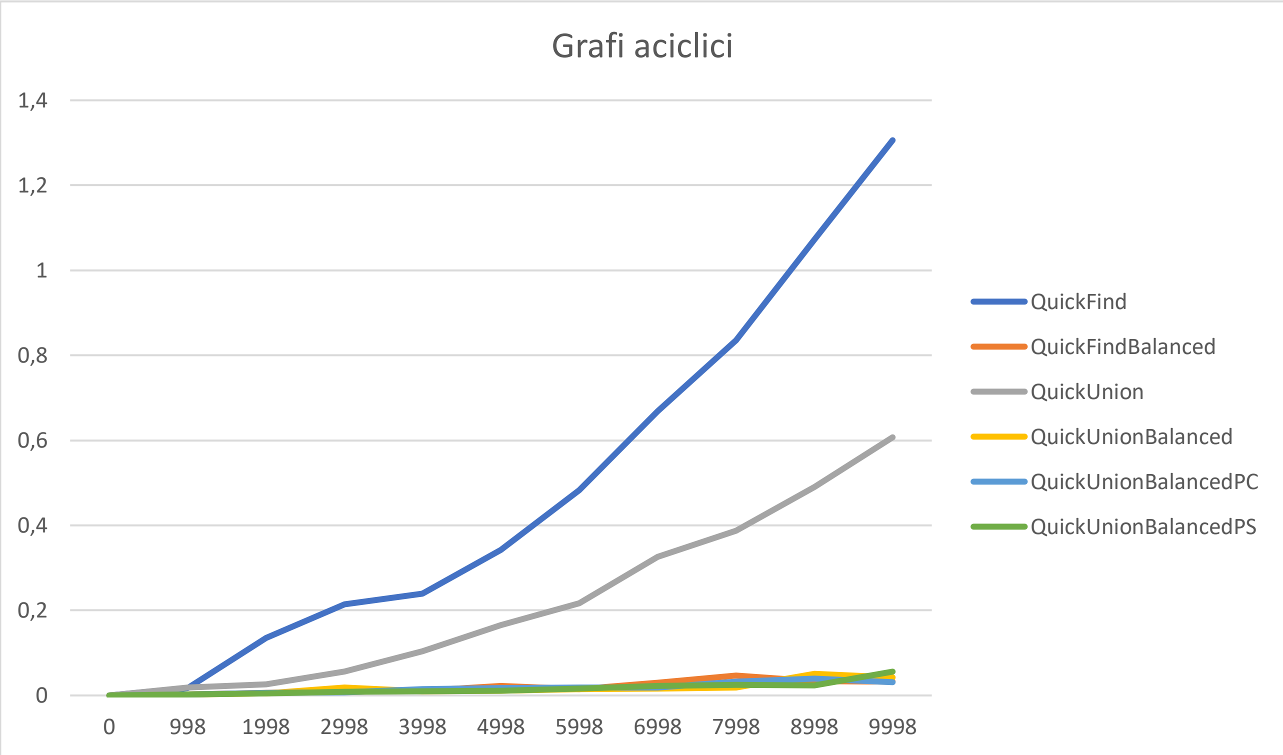
I grafici sono stati realizzati con Excel, sull'asse delle ascisse c'è il numero di archi (contando l'arco diretto da x a y e quello diretto da y a x come due archi diversi per ogni x e y) del grafo in input, su quello delle ordinate c'è il tempo di esecuzione dell'algoritmo in secondi. Tutti i grafi generati casualmente hanno numero di archi variabile, nei grafici che riguardano grafi random i valori presenti sull'asse x rappresentano quindi il numero approssimato di archi.

•Stesura del codice e primi test pratici

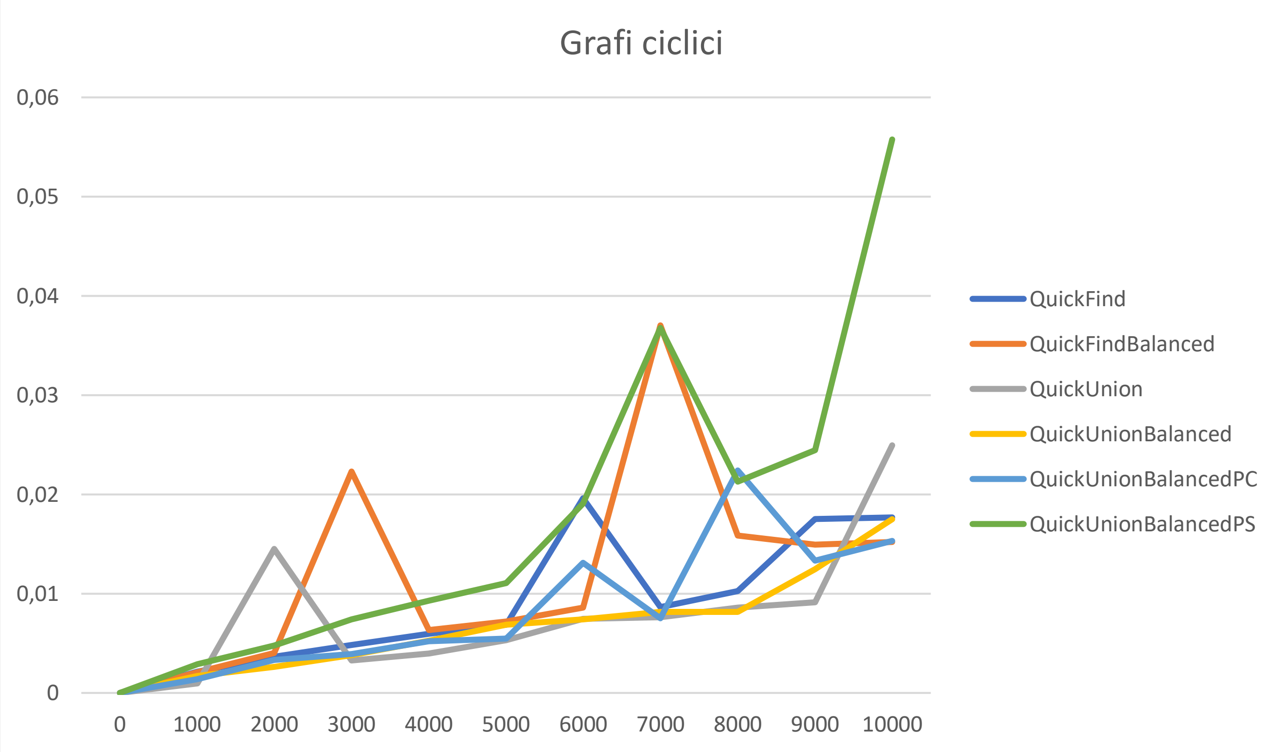
Per prima cosa sono stati implementati alcuni algoritmi per generare grafi non orientati, connessi e non pesati (commit e2e04a9, 313e58a) che fossero aciclici (`generateAcyclicGraph`), ciclici (`generateCyclicGraphNaive`), casuali (`generateRandomGraph`) o completi (`generateCompleteGraph`). Tutti gli algoritmi restituiscono il numero di archi del grafo, contando la coppia (x, y), (y, x) come un singolo arco per ogni x e y. Il generatore di grafi casuali sfrutta un generatore di interi pseudo-random che si basa su una variabile aleatoria binomiale (quindi discreta) tale che la media dei numeri generati (n * p) sia 1. La funzione `random.binomial(n,p)` è stata presa dalla libreria NumPy; è stato preferito l'utilizzo di questa all'uso di `random.randint(a,b)` per mantenere il numero di archi contenuto. Per ogni nodo viene generato un numero di archi random verso i nodi già collegati. Al crescere del numero dei nodi decresce la probabilità di ottenere una sequenza di n uno, quindi decresce la probabilità che il grafo generato sia aciclico. La formula `max(1,np.random.binomial(tempLen,1/tempLen))` garantisce che i numeri generati siano compresi tra 1 e tempLen, evitando quindi lo 0.

In seguito è stato implementato in Python lo pseudocodice dell'algoritmo `hasCycleUF` (commit 4de0826), con una modifica finalizzata a testare il tempo di esecuzione dell'algoritmo con diverse implementazioni di `UnionFind`. Gli id dei nodi vengono richiesti con `G.nodes.keys()` poiché se fossero stati richiesti con `G.getNodes()` (che restituisce `list(self.nodes.values())`), in seguito sarebbe stato necessario accederne all'id eseguendo un'operazione ridondante. Una volta inseriti nella struttura dati `UnionFind`, è possibile accedere agli id dei nodi con `uf.nodes[edge.head]` poiché questi provengono da un dominio totalmente ordinato. I test di `hasCycleUF` con diverse implementazioni di `UnionFind` sono stati eseguiti nel file project/demoHasCycleUF.py (che contiene anche il decorator `writeResults`) producendo i seguenti risultati:

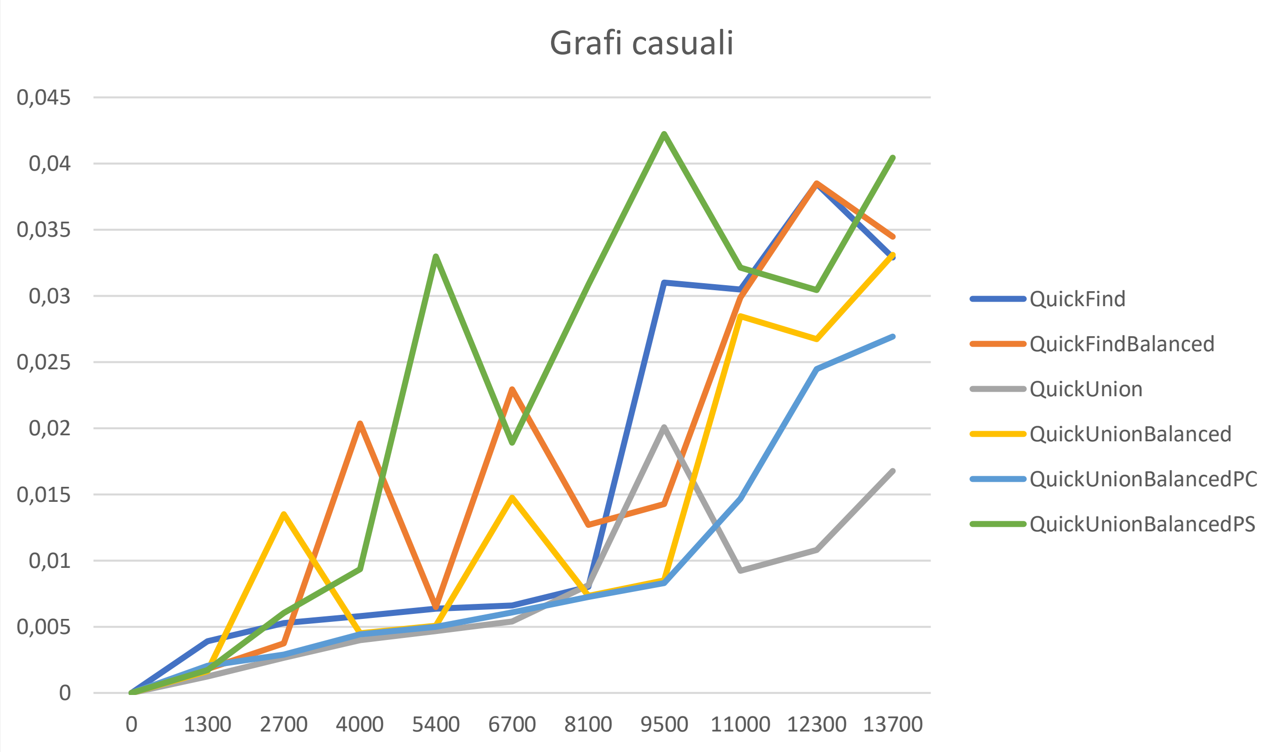
Acyclic	0	998	1998	2998	3998	4998	5998	6998	7998	8998	9998
QF	0	0,017854	0,135705	0,213937	0,240093	0,342052	0,482863	0,668091	0,835409	1,072597	1,306111
QFB	0	0,002639	0,005554	0,008692	0,010808	0,022187	0,015484	0,029745	0,046822	0,033621	0,031950
QU	0	0,018976	0,026122	0,056406	0,104547	0,165029	0,216430	0,325867	0,387415	0,490929	0,607284
QUB	0	0,002272	0,004673	0,018940	0,009330	0,010957	0,014385	0,015681	0,018133	0,050887	0,042875
QUBPC	0	0,002800	0,005696	0,007576	0,014606	0,017728	0,018996	0,018598	0,032808	0,039583	0,031627
QUBPS	0	0,002559	0,005355	0,008515	0,009476	0,011060	0,016174	0,022202	0,024517	0,023867	0,056069



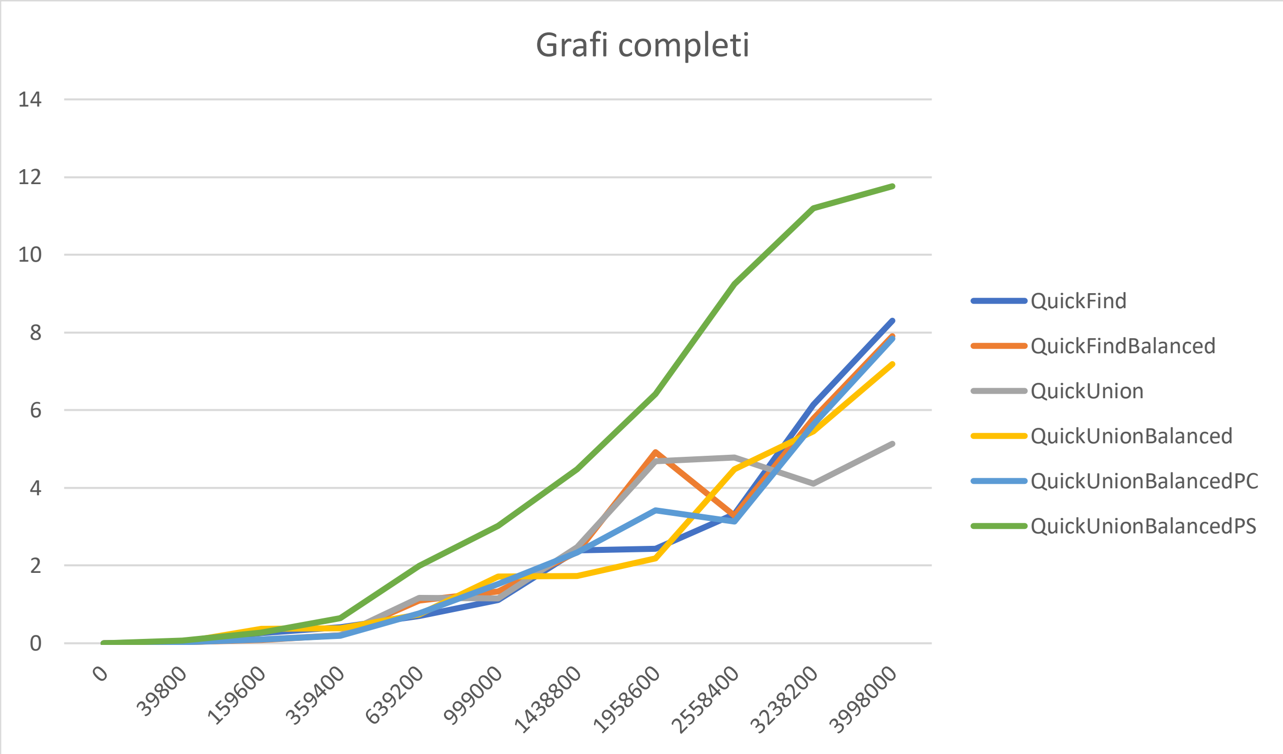
Cyclic	0	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
QF	0	0,002126	0,003629	0,004829	0,005962	0,006887	0,019609	0,008648	0,010252	0,017505	0,017699
QFB	0	0,002111	0,004033	0,022302	0,006311	0,007217	0,008595	0,037031	0,015865	0,014957	0,015186
QU	0	0,000941	0,014532	0,003256	0,003970	0,005328	0,007450	0,007645	0,008593	0,009116	0,024946
QUB	0	0,001709	0,002635	0,003831	0,005244	0,006879	0,007394	0,008176	0,008187	0,012469	0,017501
QUBPC	0	0,001409	0,003322	0,003936	0,005228	0,005468	0,013111	0,007540	0,022413	0,013342	0,015340
QUBPS	0	0,002878	0,004803	0,007397	0,009301	0,011074	0,019077	0,036772	0,021262	0,024463	0,055761



Random	0	1300	2700	4000	5400	6700	8100	9500	11000	12300	13700
QF	0	0,003919	0,005283	0,005805	0,006379	0,006626	0,008035	0,031025	0,030478	0,038473	0,032883
QFB	0	0,001827	0,003729	0,020347	0,006489	0,022929	0,012698	0,014269	0,029869	0,038504	0,034461
QU	0	0,001245	0,002642	0,003983	0,004688	0,005400	0,008129	0,020075	0,009217	0,010796	0,016761
QUB	0	0,001630	0,013484	0,004526	0,005080	0,014736	0,007334	0,008522	0,028476	0,026735	0,033102
QUBPC	0	0,002058	0,002910	0,004439	0,004990	0,006083	0,007238	0,008289	0,014655	0,024484	0,026925
QUBPS	0	0,001721	0,006036	0,009357	0,032976	0,018912	0,030829	0,042232	0,032125	0,030441	0,040455



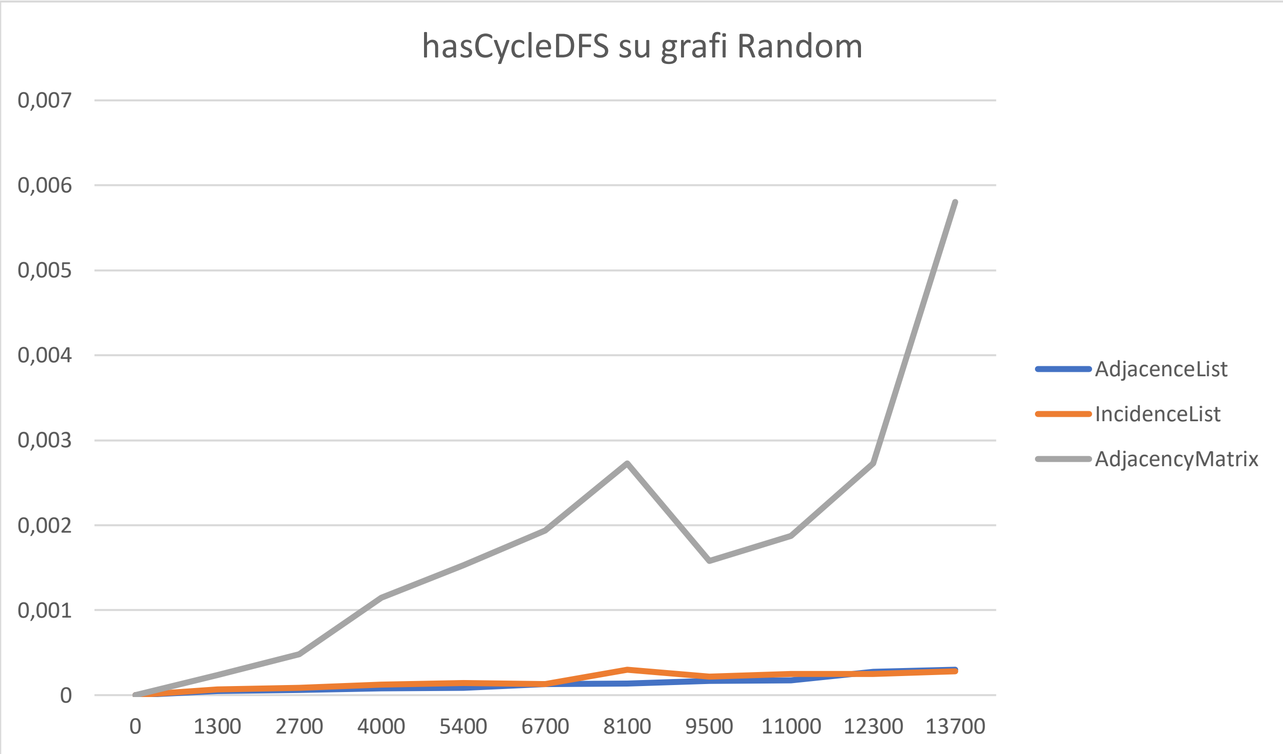
Complete	0	39800	159600	359400	639200	999000	1438800	1958600	2558400	3238200	3998000
QF	0	0,042011	0,255208	0,409865	0,700504	1,110286	2,389528	2,433451	3,321198	6,135385	8,302906
QFB	0	0,023954	0,089024	0,205583	1,096260	1,330973	2,350341	4,914069	3,286918	5,788521	7,904341
QU	0	0,025741	0,095916	0,213555	1,168879	1,150298	2,473910	4,679459	4,788367	4,102511	5,135008
QUB	0	0,032787	0,378461	0,385667	0,740772	1,715333	1,733896	2,181080	4,479575	5,454781	7,186125
QUBPC	0	0,022371	0,091962	0,198014	0,776009	1,528843	2,332118	3,425729	3,134420	5,633955	7,841218
QUBPS	0	0,067346	0,271891	0,646937	1,999339	3,019993	4,485516	6,414840	9,241779	11,19692	11,76380



Come si può vedere, `QuickUnion` e `QuickUnionBalanced` raggiungono mediamente risultati migliori, tuttavia `QuickUnionBalanced` ha performances migliori per grafi aciclici e ciclici, e viene superata di poco (~0,016 s) nei grafi casuali, che rappresentano i casi più significativi dei test. Per le esecuzioni successive è stata quindi scelta `QuickUnionBalanced` come struttura dati (commit 7bab8a9).

Per l’implementazione di `hasCycleDFS` sono state fornite una versione iterativa ed una ricorsiva (commits 897643e, 074a2ae, 676a03a, 1feb309) entrambe basate sul fatto che in grafi aciclici non ci sono archi all’indietro nell’albero DFS. Dalla teoria si sa che la visita DFS richiede meno tempo se il grafo è implementato con liste di adiacenza, ma sono stati eseguiti comunque dei test (in project/demoHasCycleDFS.py) su grafi random per verificare quale fosse l’implementazione migliore:

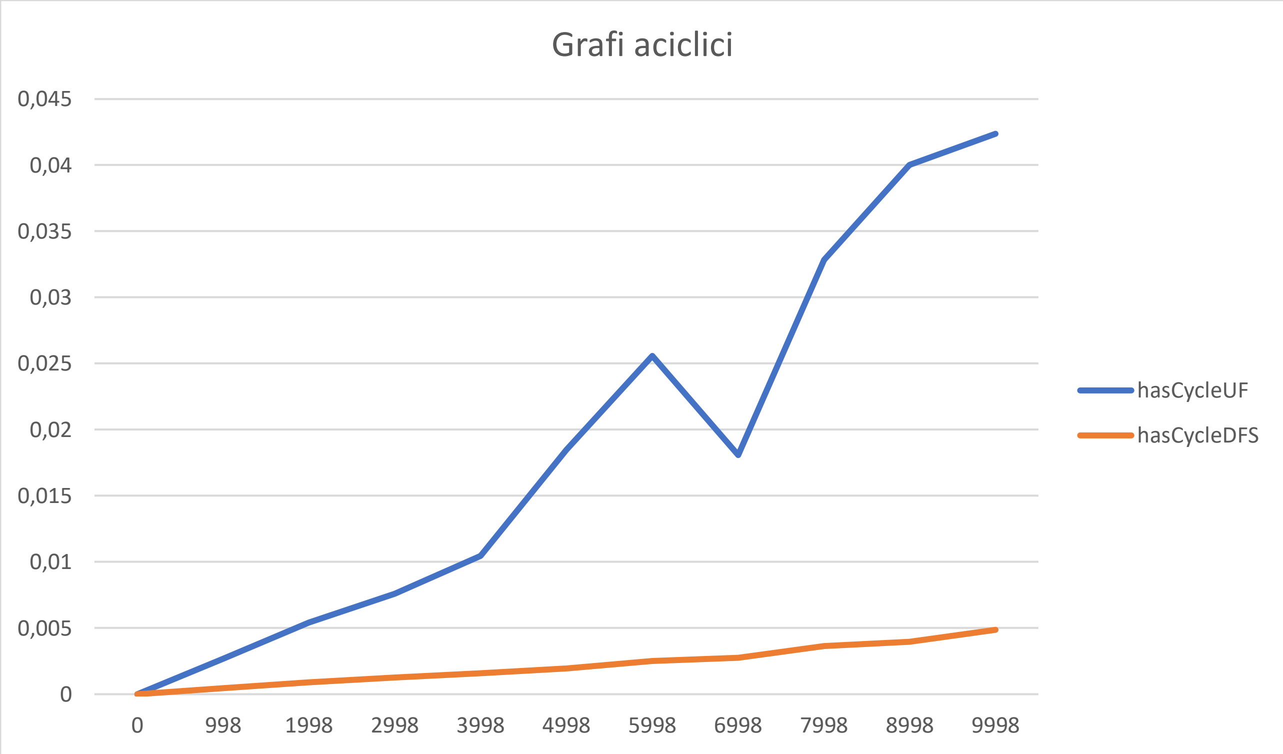
Random	0	1300	2700	4000	5400	6700	8100	9500	11000	12300	13700
AdjList	0	4,63E-05	6,29E-05	8,15E-05	8,92E-05	0,000130	0,000137	0,000165	0,000177	0,000275	0,000299
InList	0	6,84E-05	8,51E-05	0,000121	0,000140	0,000133	0,000299	0,000215	0,000248	0,000252	0,000278
AdjMat	0	0,000235	0,00048	0,001148	0,00153	0,001936	0,002725	0,00158	0,001872	0,00273	0,005804



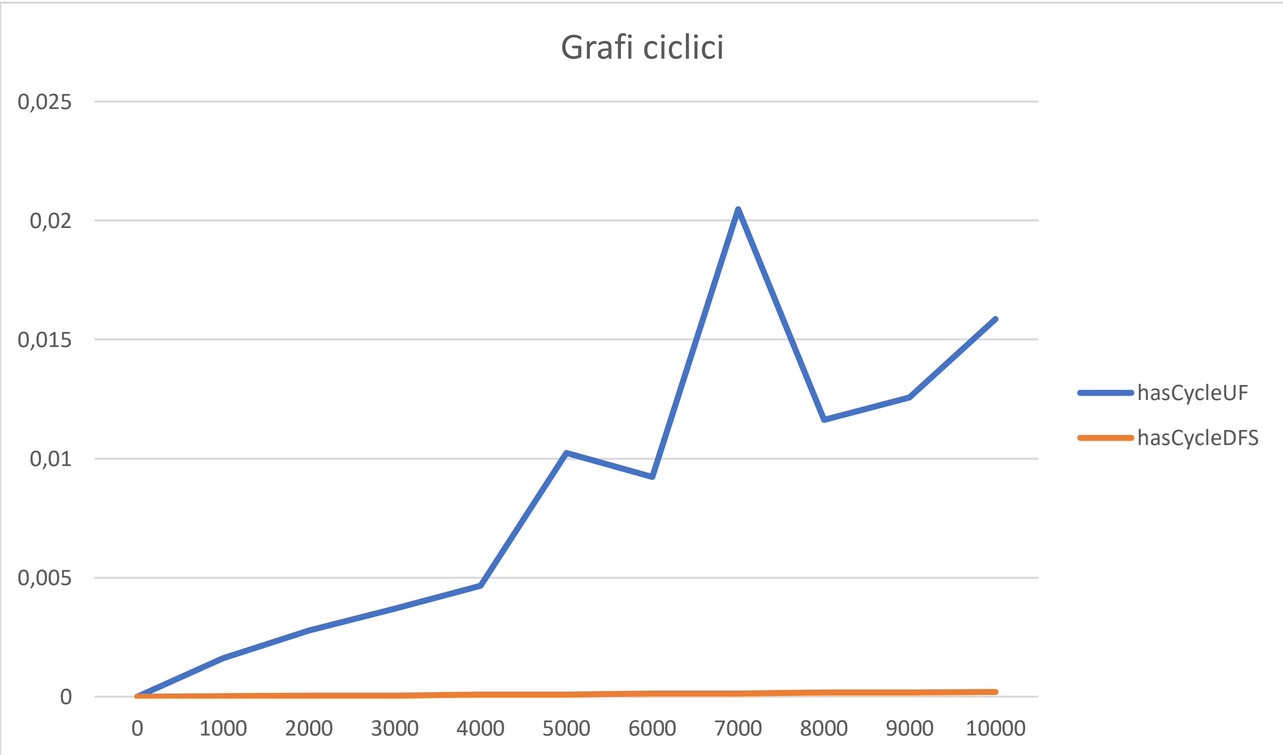
•Successivi test e commento dei risultati

Al decorator `writeResults` è stato aggiunto `@wraps` (commit 7fc0dfd), come visto nella documentazione, per far sì che funzioneDecorata.__name__ restituisse il vero nome della funzione, invece di wrapping_function. Una volta completata la stesura del codice, sono stati comparati i tempi di esecuzione di `hasCycleUF` e `hasCycleDFS` in project/demoHasCycle.py.

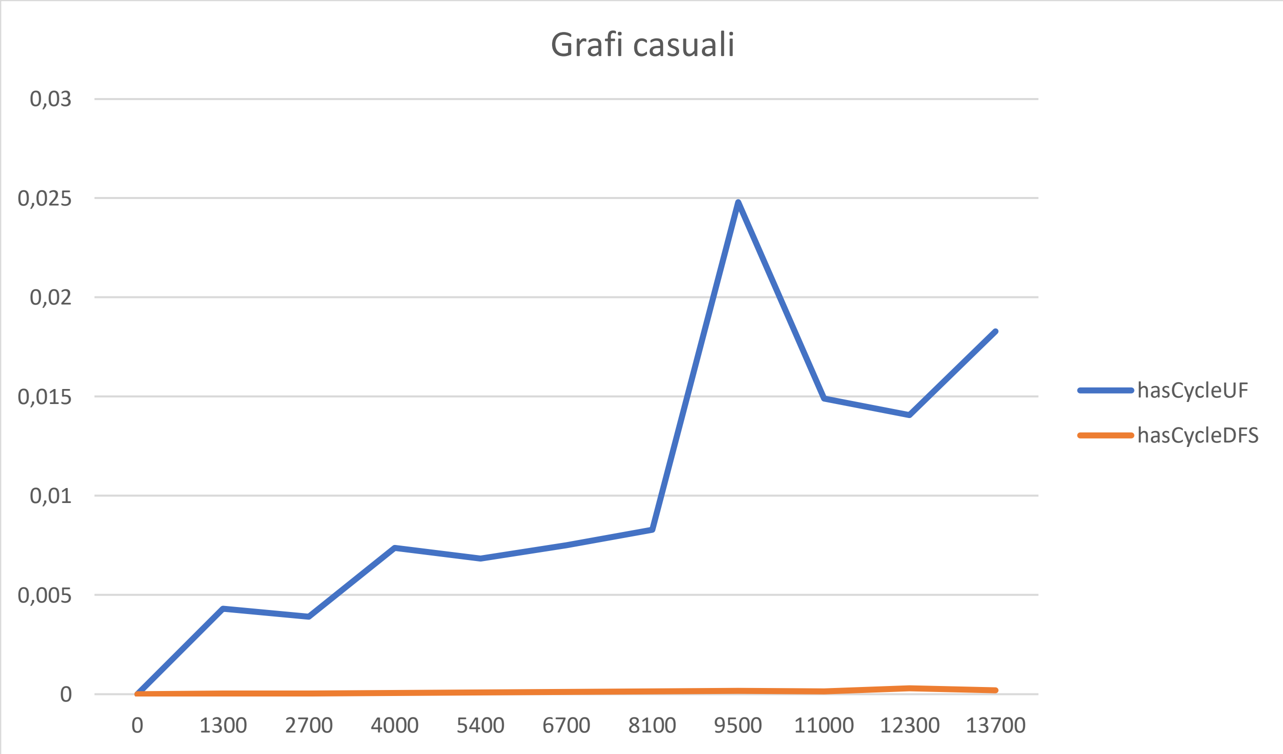
Acyclic	0	998	1998	2998	3998	4998	5998	6998	7998	8998	9998
UF	0	0,002693	0,005416	0,007589	0,010445	0,018482	0,025600	0,018097	0,032824	0,040000	0,042374
DFS	0	0,000463	0,000903	0,001246	0,001584	0,001934	0,002497	0,002759	0,003629	0,003973	0,004868



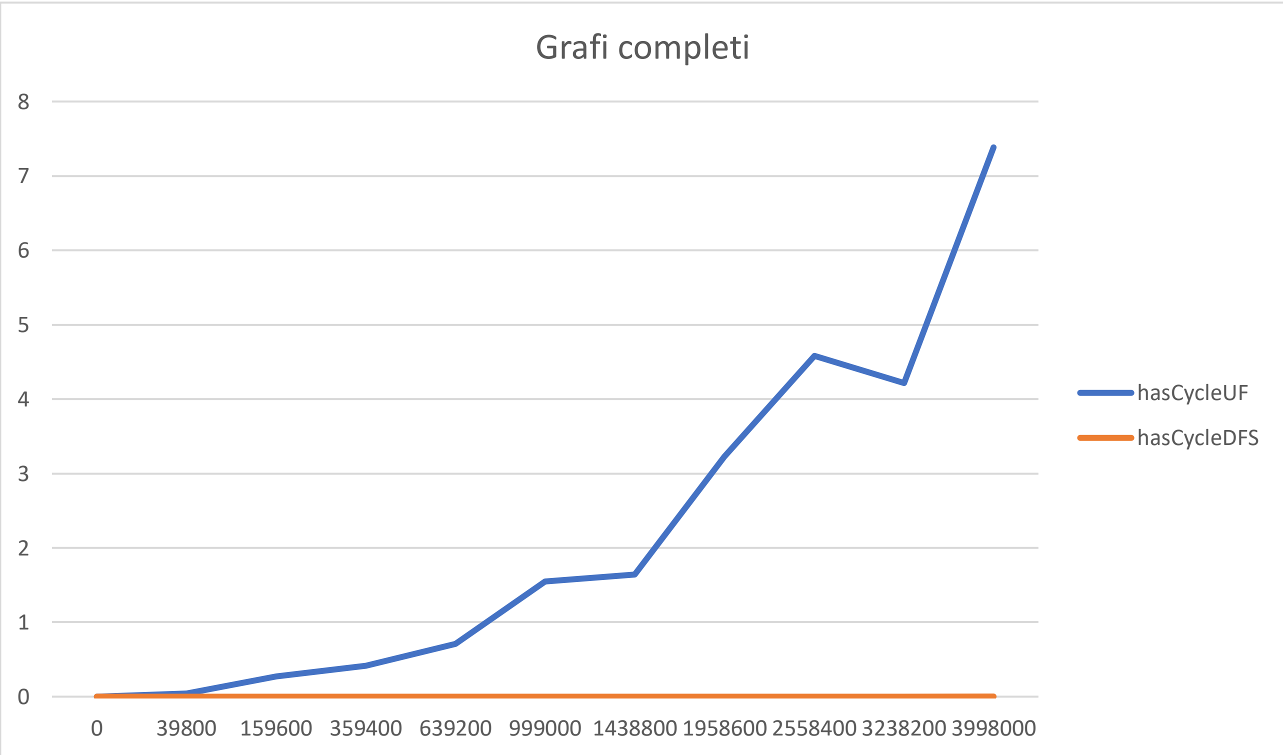
Cyclic	0	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
UF	0	0,001613	0,00278	0,003702	0,004653	0,010247	0,009239	0,020478	0,011634	0,012563	0,015867
DFS	0	2,74E-05	4,12E-05	5,32E-05	8,49E-05	8,15E-05	0,000124	0,000132	0,000174	0,000182	0,000197



Random	0	1300	2700	4000	5400	6700	8100	9500	11000	12300	13700
UF	0	0,004300	0,003894	0,007374	0,006845	0,007504	0,008284	0,024803	0,014914	0,014063	0,018281
DFS	0	3,74E-05	4,41E-05	5,36E-05	8,82E-05	0,000117	0,000138	0,000160	0,000151	0,000300	0,000190



Complete	0	39800	159600	359400	639200	999000	1438800	1958600	2558400	3238200	3998000
UF	0	0,044954	0,269073	0,414955	0,709035	1,545596	1,643993	3,225309	4,580802	4,216956	7,383876
DFS	0	0,000178	0,000274	0,000465	0,000993	0,000820	0,000939	0,001151	0,001272	0,001608	0,001770



Dato che `generateCyclicGraphNaive` crea un ciclo con i primi tre nodi (quindi il numero dei nodi deve essere >= 3), nel caso di grafi ciclici rootId è stato scelto casualmente in `hasCycleDFS`. I test mostrano una notevole differenza nei tempi di esecuzione dei due algoritmi mostrando che `hasCycleDFS` è più efficiente in ogni caso. Anche se il grafo fosse stato implementato diversamente, almeno nel caso di grafi random, `hasCycleDFS` sarebbe più veloce di `hasCycleDFS`. Per risolvere lo stesso problema in un caso pratico reale potrebbe bastare contare gli archi e fermarsi se il contatore diventa > n – 1, con n = numero dei nodi. Infatti, per definizione, un grafo non orientato, connesso ed aciclico è un albero; si può dimostrare per induzione** che un albero ha esattamente n – 1 archi. Tutti gli altri grafi, che avranno un numero di archi compreso tra n e n * (n – 1) / 2, conterranno un ciclo.

**Dimostrazione:

n = 1, banale.

prendendo un albero con n + 1 nodi, si può trovare e rimuovere una foglia (un nodo di grado 1), che esiste poiché il grafo è non orientato, connesso ed aciclico. Rimuovendo la foglia e l’arco che la collega al padre (è uno solo perché il grafo è aciclico) si ottiene un albero di n nodi, che ha n – 1 archi per ipotesi induttiva. Ne consegue che l’albero di prima aveva (n – 1) + 1 = n archi