Relazione seconda prova pratica in itinere

Ingegneria degli Algoritmi 2018/2019

Giovanni Pica
Amedeo Maria Paniccia
Francesco Legrottaglie

1 Scelte Implementative

Dato un grafo G connesso, non orientato e non pesato, abbiamo implementato due algoritmi per determinare se G ha almeno un ciclo:

- **hasCycleUF** prende come Input il grafo G (rappresentato tramite liste di adiacenza), il numero di Nodi di G ("N") con cui effettueremo il **MakeSet** e "uf"; la **UnionFind** da noi scelta.
  Dai test effettuati, prendendo in considerazione tutte le tipologie di UnionFind, abbiamo deciso di prendere la **QuickUnion.**
  Riguardo l'implementazione di questo algoritmo è stato preso in considerazione lo pseudocodice presente nella traccia, aggiungendo degli accorgimenti.
  Siccome il grafo non è orientato, c'è il rischio che l'algoritmo prenda come ciclo una **ClosedWalk** (partiamo da un vertice, attraverso vertici consecutivi si va a finire sul vertice originario).

- **hasCycleDFS(G)** usa l'approccio della visita in profondità (**DFS**) dove salviamo in tre strutture dati i nodi **aperti, chiusi** e visitati. I nodi aperti sono stati salvati in una **pila**, i secondi in una **lista**, e gli ultimi in un **dizionario.**
  Questo algoritmo consiste nel partire da una radice "Root ID" e procede visitando nodi di figlio in figlio (nodi adiacenti ad esso).
  Se il figlio non è stato visitato allora viene aggiunto al dizionario precedente e alla pila, altrimenti, se il figlio della "Root ID" è già presente nella pila, abbiamo un ciclo. Inoltre abbiamo aggiunto una funzione nella classe **PilaArrayList** chiamata **inPila**, per verificare se un elemento è presente nella pila.

- **algoritmo crea grafi con almeno un ciclo** collega tutti i nodi in ordine crescente (0 - 1,1 - 2 ecc.), prende un vertice random e lo collega con l'ultimo nodo (naturalmente il vertice randomico non può essere l'ultimo).

- **algoritmo crea grafi senza cicli**, semplicemente collega tutti i nodi in ordine crescente.

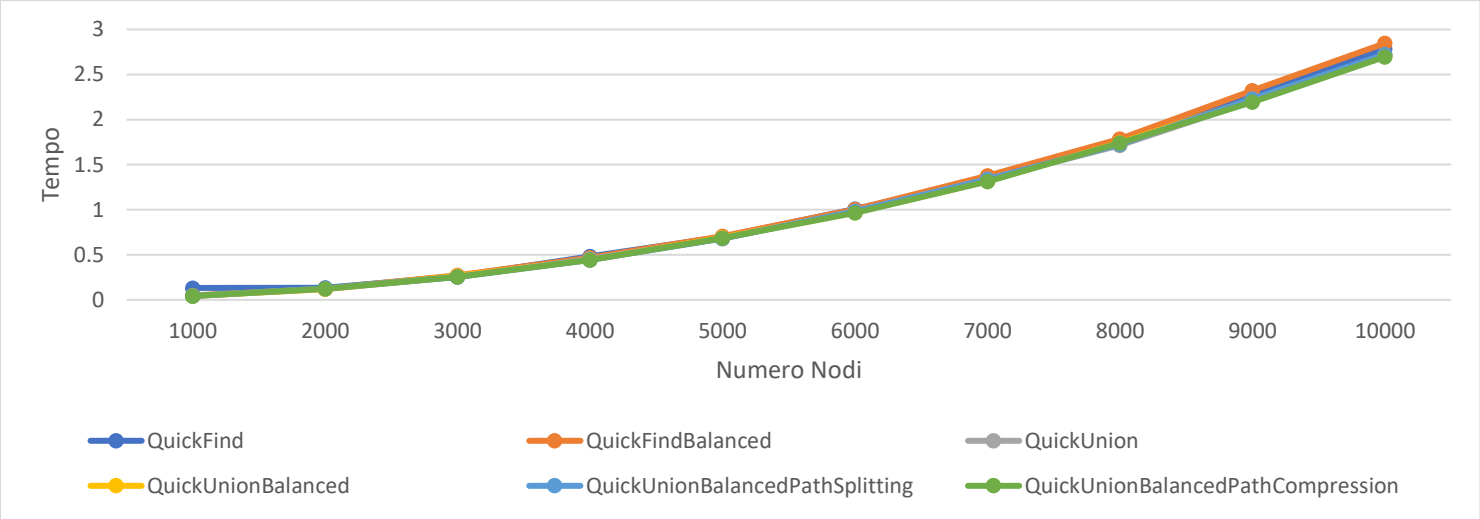
- **Iteratore,** che serve a scandire gli archi è¨ una classe dove si inizializza una lista (lista di archi) e poi si ha __**iter**__ e __**next**__ e quest'ultimo ci servirà per scorrere tutti gli elementi della lista.

- **decorator**, prende spunto da timer.py però come ***args** avrà il numero di nodi del grafo, stamperà il numero di archi presenti e le tempistiche dei due algoritmi.

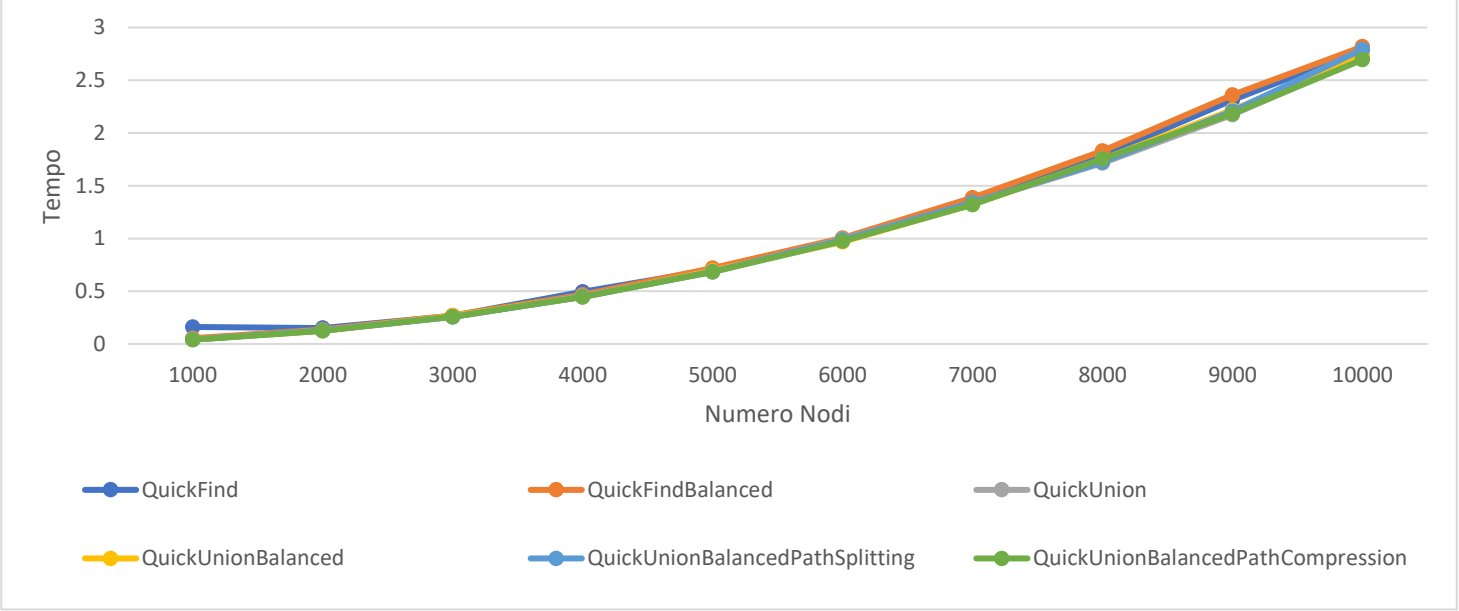
- I primi 5 algoritmi sono presenti nella cartella graph, nel file algoritmi.py
- Il decorator è presente nella cartella decorator

2 Risultati Sperimentali

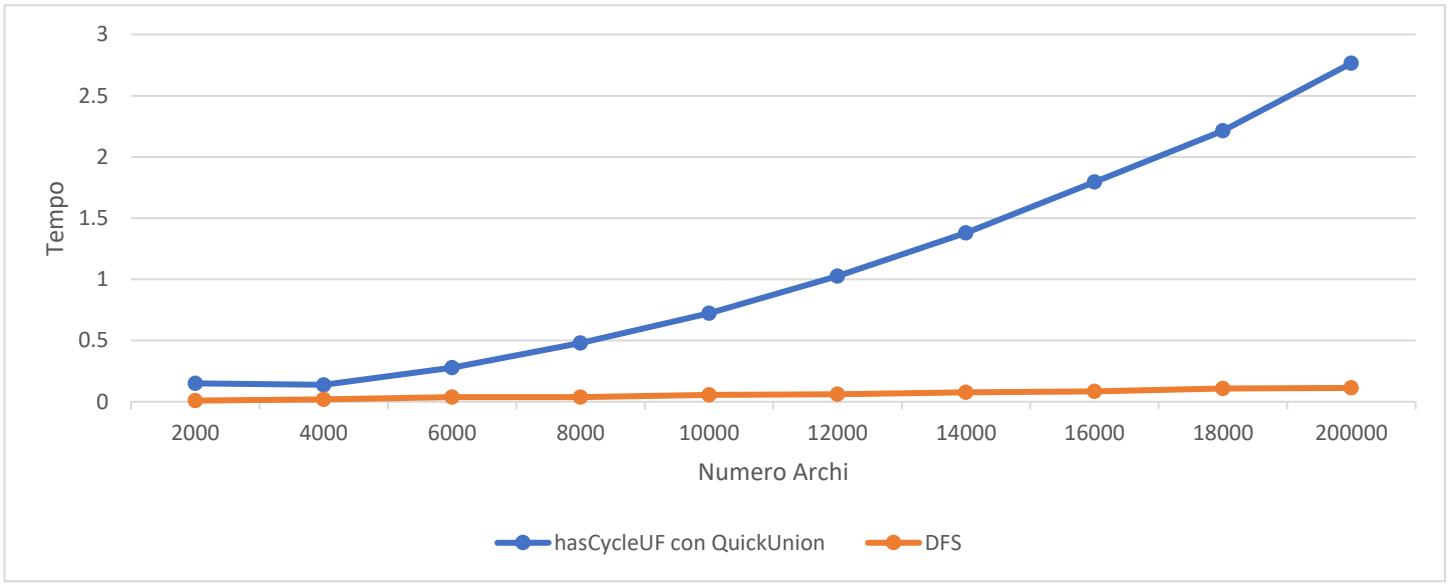
2.1 Confronti UnionFind (grafo con almeno un ciclo)



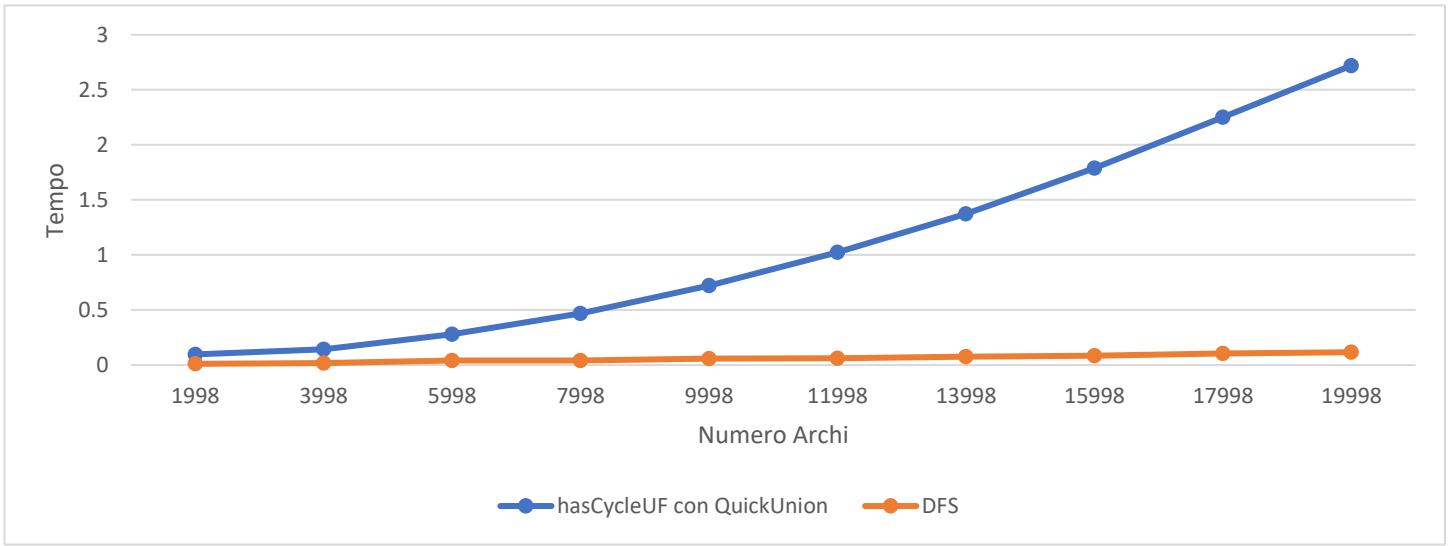
2.2 Confronti UnionFind (grafo senza cicli)



2.3 Confronto DFS, hasCycleUF con QuickUnion (grafo con cicli)



2.4 Confronto DFS, hasCycleUF con QuickUnion (grafo senza cicli)



3 Tempistiche inerenti ai grafici precedenti

- **Tempistiche relative al grafico 2.1**

	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
QuickFind	0,131793022	0,134773493	0,262221098	0,483615398	0,696857691	1,009228706	1,349651814	1,776514769	2,287766218	2,785419464
QuickFindBalanced	0,048509598	0,124623299	0,271030664	0,467826366	0,706375122	1,004811287	1,375653505	1,785401344	2,323553085	2,846505642
QuickUnion	0,043002844	0,120037794	0,255735636	0,453282118	0,679644346	0,981092453	1,336371899	1,713355064	2,203688145	2,705368757
QuickUnionBalanced	0,043263912	0,120017052	0,263702154	0,442587614	0,692032099	0,970498562	1,315604925	1,741728306	2,221960783	2,715500355
QuickUnionBalancedPathSplitting	0,043378115	0,128027439	0,256037235	0,442824602	0,681344032	0,983039379	1,337045908	1,724390984	2,23129487	2,723817587
QuickUnionBalancedPathCompression	0,042816162	0,121397734	0,253148556	0,442408562	0,683831692	0,966109514	1,312497377	1,740835667	2,194395304	2,696055651

- **Tempistiche relative al grafico 2.2**

	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
QuickFind	0,161533594	0,15059948	0,268535376	0,493589878	0,704469681	0,995123863	1,363304615	1,78512764	2,31332922	2,790010214
QuickFindBalanced	0,054338932	0,135859013	0,268620968	0,46877265	0,717936039	1,0047791	1,387446165	1,830449104	2,362329245	2,818265438
QuickUnion	0,041215897	0,129968882	0,25598073	0,456439972	0,681909084	0,996486664	1,335317135	1,714062214	2,172535658	2,713886499
QuickUnionBalanced	0,042268991	0,123745203	0,266474962	0,445091486	0,696444035	0,968303442	1,324159145	1,74986434	2,217751741	2,736968994
QuickUnionBalancedPathSplitting	0,041074991	0,127950907	0,258728266	0,445887089	0,685133696	0,987010002	1,340174675	1,723530293	2,207625866	2,792411089
QuickUnionBalancedPathCompression	0,0402987	0,12522912	0,257441759	0,445878267	0,68306303	0,973210573	1,320889235	1,75535512	2,181971073	2,694676399

- **Tempistiche relative al grafico 2.3**

	2000	4000	6000	8000	10000	12000	14000	16000	18000	200000
hasCycleUF con QuickUnion	0,15088105	0,13874125	0,27862191	0,47879267	0,72436309	1,02693343	1,38013577	1,79450083	2,21403289	2,76597857
DFS	0,00983262	0,01913953	0,0390079	0,03879142	0,05539642	0,06138039	0,07814407	0,08570766	0,1087234	0,11325407

- **Tempistiche relative al grafico 2.4**

	1998	3998	5998	7998	9998	11998	13998	15998	17998	19998
hasCycleUF con QuickUnion	0,097025156	0,142501593	0,280448914	0,470202208	0,723112106	1,023376226	1,372478724	1,788155556	2,249778032	2,719797611
DFS	0,011204243	0,019599199	0,040568829	0,042021036	0,059198856	0,063003778	0,077166319	0,084335327	0,10400033	0,116675854

4 Commento dei risultati ottenuti

- Nel primo test (grafico 2.1 e 2.2) abbiamo confrontato tutte le tipologie di unionFind, utlizzando:
  grafo con ciclo e senza ciclo con il numero di nodi che varia da 1000 a 10000.

  - **file:** unionfindcompare.py nella cartella unionfind.

  Da qui si evince come ci sia un netto equilibrio tra tutti i tipi di unionFind, però un piccolo vantaggio (seppur di millesimi di secondi) lo hanno sicuramente le QuickUnion (“normale”, balanced ecc.).

- Nel secondo test (grafico 2.3 e 2.4) abbiamo confrontato hasCycleDFS & hasCycleUF, utilizzando:
  grafo con ciclo e senza ciclo con il numero di archi variabile.

  - **file:** decorator.py nella cartella decorator

  Da quest'ultimo notiamo come hasCycleDFS sia di gran lunga più veloce di hasCycleUF.