Seconda Prova Pratica In Itinere

Ingegneria Algoritmi 2018/2019

°°°

Progetto 2 [Barba]

°°°

RELAZIONE DI:
Adriano Bramucci (Matricola 0240128)
Matteo Conti (Matricola 0244242)
Stefano Picchioni (Matricola 0218935)

Nel Progetto è richiesta l'implementazione di due algoritmi, i quali interagiscono con un grafo G(V, E) connesso, non orientato e non pesato. I due algoritmi devono determinare l'eventuale presenza di un ciclo nel grafo fornito in input.

I due algoritmi verranno chiamati:

- **hasCycleUF**(grafo G) che utilizza la struttura dati *UnionFind* e un *iterator* per scandire gli edge del grafo.

- **hasCycleDFS**(grafo G) che usa l'approccio della visita in profondità (Depth-First-Search).

Per il primo algoritmo possiamo scegliere l'implementazione della *UnionFind*(*QuickFind*, *QuickUnion*, ecc) in maniera opportuna da minimizzare il tempo di esecuzione e rendere efficiente l'algoritmo.
Viene richiesto di effettuare diversi confronti tra gli algoritmi e di verificare i tempi di esecuzione al variare dei grafi forniti in input.

Inoltre, implementare un *decorator* che scrive i dati degli esperimenti su di un file con il formato n, t su ogni riga dove n è il numero di spigoli del grafo e t è il tempo di esecuzione dell'algoritmo.
Viene anche richiesto di realizzare un algoritmo che generi grafi "random" con o senza cicli, da poter usare nella fase di testing.

Funzionamento hasCycleUF

Pseudocodice Algoritmo

```
procedure hasCycleUF(Graph G) →
    bool uf ← UnionFind
        if uf.find(u) == uf.find(v)
           then cycle detected
        else uf.union(u, v)
    return no cycle present
```

Funzionamento: Dato un grafo e una struttura UnionFind **uf** l'algoritmo andrà a scandire tutti gli spigoli e a verificare se l'elemento u si trova nello stesso **set** dell'elemento v. Se la condizione nella if è verificata allora abbiamo trovato un ciclo altrimenti eseguiamo una **union** tra i due set.
Se si conclude la for senza aver trovato due elementi nello stesso set allora il grafo non presenta cicli.

Funzionamento hasCycleDFS

Algoritmo che esegue una visita Depth-First-Search sul grafo in input.

PseudoCodice Algoritmo

```
procedure hasCycleDFS(Graph G) →
    bool chiama la procedura dfs

procedure dfs(vertice v) → bool
    s ← Stack
    s.push(v)
    explored ← v
    while (!s.isEmpty())
       node ← s.pop()
       explored ← node
       aggiungi tutti i nodi adiacenti di node a s
       if ho già visitato il nodo
          return True
    return False
```

Funzionamento: L'algoritmo importante è la visita dfs la quale scandisce tutti i vertici in profondità e li segna come esplorati. Se la dfs si ritrova su di un vertice già esplorato in precedenza ci troviamo davanti ad un ciclo e allora ritorna true, altrimenti termina la dfs e non abbiamo trovato cicli.

Moduli del Progetto

1)Cartella datastruct: Contiene le classi Queue e Stack(da utilizzare con l'algoritmo DFS).

2)Cartella dictionary: Contiene informazioni riguardo strutture dati di tipo Dizionario e Albero.

3)Cartella graph: Contiene classe Graph e tutti i moduli per il corretto funzionamento della struttura dati.

4)Cartella list: Contiene classe per la struttura LinkedList.

5)Cartella unionfind: Contiene classi per le strutture dati UnionFind, QuickUnion e QuickFind.

6)Cartella grafici: Contiene i grafici generati con la libreria pyplot a partire dagli array generati eseguendo il programma testing.py

7)Moduli Importanti

- **Testing.py:** Contiene codice riguardo il testing dei vari algoritmi. Usato per confrontare i tempi di esecuzione e produrre un risultato.

- **ufCheckQuickFind.py:** Contiene gli algoritmi **hasCycleUFQF** e **hasCycleUFQFB**.

- **ufCheckQuickUnion.py** Contiene gli algoritmi **hasCycleUFQU, hasCycleUFQUB, hasCycleUFQUBPC e hasCycleUFQUBPS**.

- **dfsCheck.py:** Contiene l'algoritmo **hasCycleDFS**.
  Nota: La Dfs è una modifica di quella contenuta nel file Graph.py della cartella graph.

- **creaGrafiMan.py:** Contiene modulo per la creazione di grafi in modo manuale che sono stati usati in fase di progettazione.

- **creaGrafiRand.py:** Contiene modulo per la creazione di grafi random che sono stati usati per il testing.

- **decoratorProgetto.py:** Contiene codice per l'implementazione del decorator richiesto nel progetto.

Conclusioni

Come si evince dalle tabelle e dai grafici allegati mettendo a confronto le varie implementazioni di UnionFind sia nei grafi senza ciclo che in quelli contenenti un ciclo asintoticamente risulta essere più veloce la quickUnion bilanciata con path compression mentre fino ad n ~ 300 risulta essere leggermente più veloce la quickFind. Per quanto riguarda il confronto tra l'algoritmo che utilizza la visità in profodità si può vedere che non c'è paragone con l'implementazione che utilizza la struttura unionFind in quanto risulta essere largamente più prestante in grafi di dimensione qualsiasi con e senza ciclo.

Tabella Tempi

N=Size Graph

Algoritmo	N=10	N=100	N=1000
DFS CYCLE	16ms	17ms	51ms
DFS	25ms	27ms	37ms
UFQU CYCLE	16ms	0.6s	92s
UFQU	31ms	0.7s	82s
UFQUB CYCLE	16ms	0.7s	87s
UFQUB	31ms	0.6s	78s
UFQF CYCLE	16ms	0.5s	66s
UFQF	31ms	0.5s	67s
UFQFB CYCLE	16ms	0.5s	66s
UFQFB	31ms	0.5s	65s
UFQUBPC CYCLE	16ms	0.5s	54s
UFQUBPC	31ms	0.5s	55s
UFQUBPS CYCLE	16ms	0.7s	114s
UFQUBPS	31ms	0.7s	116s