Esercizio F: Componenti connesse e MST

Giovanni Stefanini - 6182949

Maggio 2021

Introduzione

Analisi delle performance di un'implementazione in Python della struttura dati Union-Find e degli algoritmi per il calcolo delle componenti connesse e di un albero di connessione minimo di un grafo.

1 Union-Find

La **Union-Find** è una struttura dati in grado di gestire una collezione $S = \{S_1, S_2, ..., S_k\}$ di **insiemi disgiunti dinamici**. Le operazioni consentite sono le seguenti:

`MakeSet(x)`: crea un nuovo insieme $S_i = \{x\}$ e aggiunge S_i a S
`Union(x, y)`: aggiunge S_x a S_y e distrugge S_x (o viceversa)
`FindSet(x)`: ritorna il rappresentante (la chiave) dell'insieme che contiene x

Il tempo di esecuzione per `MakeSet(x)` è $O(1)$, in quanto la funzione deve solo aggiornare dei puntatori. Diversamente accade per `FindSet(x)`, che potrebbe dover scorrere tutti gli elementi di S prima di trovare x (o restituire un NIL). `Union(x, y)` è invece ancora più costosa in quanto, nel caso peggiore, potrebbe dover unire tutti gli elementi in S, per un tempo di esecuzione quadratico $O(n^2)$.

2 Componenti connesse

Le **componenti connesse** di un grafo $G = (V, E)$ partizionano i vertici V in classi di equivalenza secondo la relazione "è raggiungibile da", ovvero per il grafo G i vertici u e v sono nella stessa componente connessa se e solo se c'è un cammino tra di loro. L'algoritmo (in pseudocodice) per il loro calcolo è:

```
ConnectedComponents(G)
   for ogni vertice v ∈ G.V
      MakeSet(v)
   for ogni arco (u,v) ∈ G.E
      if FindSet(u) ≠ FindSet(v)
         Union(u,v)
```

3 Alberi di connessione minimi (MST)

Gli alberi di connessione minimi, o **MST** (*minimum spanning trees*), sono alberi (ovvero sottoinsiemi degli archi di un grafo $G = (V, E)$) che connettono tutti i vertici minimizzando il costo totale degli archi stessi (nei grafi pesati). Ogni MST di G (ce ne possono essere più di uno) ha sempre $|V| - 1$ archi e non contiene cicli (dato che si tratta di un albero, grafo aciclico non orientato per definizione). Per trovare un MST di G si può utilizzare la struttura dati Union-Find in combinazione con uno tra gli algoritmi di Kruskal o Prim (entrambi con tempi di

esecuzione dipendenti dall'implementazione di Union-Find). In questo esperimento faremo uso soltanto dell'algoritmo di Kruskal, qui illustrato in pseudocodice:

```
MST-Kruskal(G, w)
   A = ∅
   for ogni vertice v ∈ G.V
      MakeSet(v)
   ordina gli archi di G.E in ordine non decrescente rispetto al peso w
   for ogni arco (u, v) ∈ G.E preso in ordine di peso non decrescente
      if FindSet(u) ≠ FindSet(v)
         A ← A ∪ {(u, v)}
         Union(u,v)
   return A
```

4 Specifiche tecniche

Per la creazione del programma nel linguaggio Python sono state utilizzate le seguenti classi:

-unionFind(): Ai fini dell'esperimento, la struttura dati Union-Find è stata implementata come un dizionario S in cui ogni chiave è costituita da un vertice del grafo, ed i suoi relativi valori sono gli archi che partono/entrano in esso (il grafo non è orientato). Tre funzioni, `makeSet(x)`, `union(x, y)` e `findSet(x)` (quest'ultima ritorna la chiave di x oppure `None`), implementano nella struttura S gli algoritmi sopra presentati in pseudocodice.

-node(k): Classe ausiliaria per la rappresentazione dei vertici; contiene soltanto la chiave k del vertice.

-edge(u, v, w): Classe ausiliaria per la rappresentazione degli archi (u, v), contenente i nodi u e v ed il peso w dell'arco (sempre 1 in grafi non pesati).

-randomGraph(n, min, max, w): Classe che genera un grafo casuale, con la possibilità di scegliere il numero di nodi n ed il numero minimo (min) e massimo (max) di archi. Dopo alcuni controlli sui valori ricevuti in input di min e max viene generato il numero k di archi da creare, compreso tra min e $n \cdot max$ (in modo che k sia almeno 0 ed al più $n \cdot n$); un ciclo `while` genera i k archi aggiungendoli alla matrice di adiacenza $adjMatrix$ del grafo, che era stata precedentemente inizializzata a tutti zeri. Se il grafo non è pesato ($w = False$), allora gli archi aggiunti avranno tutti peso pari a 1; altrimenti, se $w = True$, gli archi avranno un peso scelto casualmente tra 1 e 15 (valori arbitrari). In seguito viene creata la lista dei nodi *nodes* e quella degli archi *edges*, quest'ultima analizzando la presenza degli archi in $adjMatrix$. Due funzioni aggiuntive, `MSTKruskal()` e `connectedComponents()` implementano sul grafo gli omonimi algoritmi, seguendo letteralmente lo pseudocodice sopra illustrato.

Il programma principale di Test (*Test3ASD.py*) crea, in cicli `for`, grafi casuali pesati e non, con un numero di nodi sempre maggiore (fino ad un massimo di $n = 450$, con step di 30), e calcola per ognuno di essi le componenti connesse ed un albero minimo di connessione tramite le funzioni `connectedComponents()` e `MSTKruskal()` di `randomGraph(n, min, max, w)`. Per ogni iterazione dei suddetti algoritmi viene salvato il tempo di esecuzione in array ausiliari, ai fini di tracciare successivamente i grafici e le tabelle *tabella-GrafiNONPesati.html* e *tabella-GrafiPesati.html* (che elencano i risultati), utili per l'analisi delle performance.

Le **specifiche hardware e software** della macchina utilizzata per eseguire i test sono:

Scheda madre: X580VD Scheda di base ASUSTeK COMPUTER INC.
CPU: Intel Core i7-7700HQ CPU - 2.80GHz, 2808 Mhz, 4 core, 8 processori logici
RAM: 16 GB
SSD: SanDisck SD8SN8U128G1002 - 120 GB
HDD: TOSHIBA MQ04ABF100 - 1 TB

SO: Microsoft Windows 10 Home
IDE: JupyterLab 2.2.6

Prestazioni attese

Considerato il tempo di esecuzione $O(n^2)$ di `Union(x, y)`, che rende quadratici gli algoritmi per il calcolo delle componenti connesse e di un albero minimo di connessione, ci si aspetta che gli andamenti, all'aumentare dei nodi, possano essere sempre più simili a una funzione quadratica (i.e. una parabola), e che i tempi di esecuzione siano quindi sempre maggiori.

5 Analisi dei risultati

Nodi	Tempo CC (ms)	Tempo Kruskal (ms)
0	0.004	0.004
30	0.292	0.323
60	0.956	0.929
90	4.021	4.177
120	20.703	22.394
150	50.093	52.614
180	41.535	43.211
210	52.091	54.73
240	206.276	213.286
270	231.237	241.664
300	120.985	120.002
330	92.962	96.32
360	350.024	368.922
390	534.263	546.624
420	427.962	435.129
449	886.945	901.34

Figura 1: Tabella del tempo t (s) delle CC e di MST-Kruskal per alcuni grafi non pesati G

GRAFI NON PESATI: Osservando la prima tabella (fig.1) e i due grafici (fig.2), riferiti ai grafi non pesati, è abbastanza chiara la conferma delle aspettative riguardo i tempi di esecuzione: `union(x, y)` richiede proprio $O(n^2)$, portando ad avere, a meno delle imperfezioni dovute ad un numero crescente di nodi ed uno sempre maggiore (ma comunque variabile) di archi, un andamento del tempo che possiamo immaginare sia una curva quadratica o, per meglio dire, una parabola. È interessante, inoltre, notare un aspetto non molto evidente, ma molto importante: la colonna sulla destra, riferita ai tempi di esecuzione di `MSTKruskal()`, è quasi perfettamente identica a quella a sinistra, riferita ai tempi di `connectedComponents()`.
Questo fenomeno non è affatto casuale, in quanto si è continuamente ripresentato durante i test condotti (non solo quindi nell'istanza da cui sono state tratte le figure); esso è dovuto al fatto che i due algoritmi analizzati sono estremamente simili, al punto da far coincidere, nella pratica, i loro tempi di esecuzione. Tuttavia, studiando la tabella, si nota come `MSTKruskal()` ha sempre un tempo lievemente superiore rispetto a `connectedComponents()`, dovuto alle poche righe di codice aggiuntive che questo algoritmo richiede.

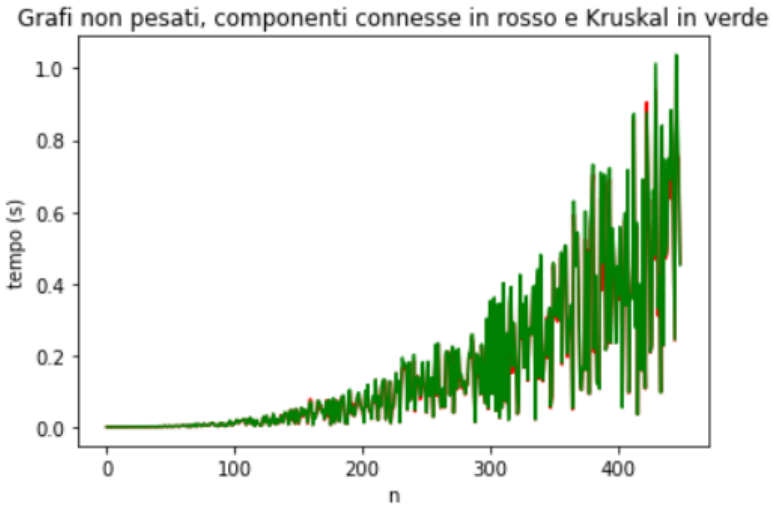

Figura 2: Grafici del tempo t (s) delle CC e di MST-Kruskal per alcuni grafi non pesati

Nodi	Tempo CC (ms)	Tempo Kruskal (ms)
0	0.01	0.006
30	3.691	2.164
60	3.26	3.984
90	2.497	3.254
120	19.292	22.079
150	29.741	35.476
180	7.815	8.992
210	60.403	70.223
240	51.135	59.527
270	149.202	171.773
300	134.213	155.733
330	440.091	518.291
360	557.463	630.014
390	671.214	716.534
420	532.053	590.138
449	1009.269	1123.968

Figura 3: Tabella del tempo t (s) delle CC e di MST-Kruskal per alcuni grafi pesati G

GRAFI PESATI: Proseguendo con i grafi pesati, si osserva (fig.3 e fig.4) che accadono gli stessi fenomeni descritti per i grafi non pesati, sia per quanto riguarda la quadraticità del tempo di esecuzione di `union(x, y)`, sia per quanto riguarda la quasi completa sovrapposizione dei tempi di esecuzione di `connectedComponents()` e `MSTKruskal()`. L'unica differenza, visibile nella rispettiva tabella (fig. 2), consiste nei tempi di esecuzione generalmente maggiori, dovuti all'elaborazione di nodi contenenti più informazioni rispetto ai grafi non pesati e, nel caso del calcolo di un MST, all'ordinamento degli archi (il quale avrà più lavoro da svolgere, dati i pesi diversi).

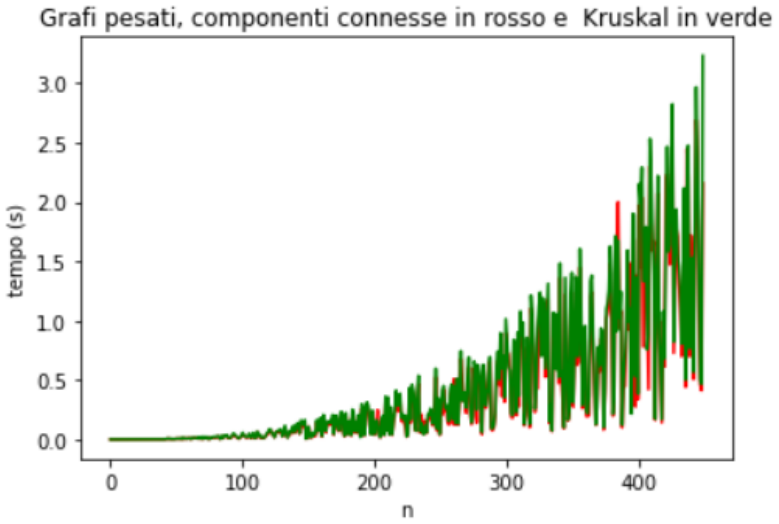

Figura 4: Grafici del tempo t (s) delle CC e di MST-Kruskal per alcuni grafi non pesati

6 Conclusioni

In conclusione, l'implementazione di Union-Find in Python tramite un dizionario S non permette di ottenere tempi di esecuzione migliori di $O(n^2)$ nel caso peggiore, e gli esperimenti condotti sui grafi hanno reso palese questo fatto. In ogni caso, sono stati confermati tramite gli esperimenti condotti i tempi di esecuzione quadratici di Union-Find (e `ConnectedComponents(G)` e `MST-Kruskal(G, w)` di conseguenza), e l'estrema somiglianza tra gli algoritmi trattati.