Componenti Connesse e MST di un grafo

di Pagliocca Marco

1 Introduzione

I grafi sono strutture dati molto importanti perché permettono di rappresentare la relazione vigente tra elementi generici. Mediante nodi e archi sono determinate modalitá rappresentative intuitive e riproducibili in via grafica. Talvolta peró sono richieste informazioni piú criptiche, non recuperabili con una semplice lettura. A tal proposito si rende necessario assumere un rigore matematico su cui implementare determinati algoritmi che prelevano informazioni in modo conveniente.

Componenti Connesse Dato un grafo G=(V,E), una **componente connessa** é un insieme *massimale* di vertici C⊆V tale che per ogni coppia di nodi esiste un *cammino* che li collega.
Per via grafica trovare le componenti connesse (CC) é molto semplice, ma non sempre si ha questo tipo di rappresentazione, per cui é necessario utilizzare un algoritmo in grado di determinarle in ogni circostanza.
Il generico procedimento consiste nella 'lettura' di tutti gli archi del grafo e l'inserimento, in una lista comune, dei nodi che per via diretta o indiretta sono collegati, cioé esiste un cammino.
Una possibile implementazione utilizza la struttura dati **Union-Find**, cioé una collezione di insiemi dinamici, ciascuno identificato da un *rappresentante*, cioé un elemento dell'insieme che rimane fisso. Vi si puó operare in 3 modi:

MakeSet(x): aggiunge alla collezione un nuovo insieme creato a partire dall'elemento x;

FindSet(x): restituisce il rappresentante dell'insieme di x;

Union(x,y): aggiunge alla collezione un nuovo insieme contenente gli elementi degli insiemi di x e y, e poi elimina gli insiemi d'origine.

Alberi di Connessione Minimi Un problema ben piú articolato é trovare il cosiddetto **Albero di Connessione Minimo**, o MST, di un grafo pesato G, ovvero un albero T in cui la somma dei pesi dei suoi archi:

$$W(T) = \sum_{u,v \in T} w(u,v)$$

sia minimo e connetta tutti i nodi.
Il generico procedimento risolutivo consiste nella creazione di un insieme di archi A, inizialmente vuoto, dove viene inserito uno alla volta un arco in modo tale che sia conservata la seguente *Invariante di Ciclo*:

"Se A é un sottoinsieme di qualche MST, l'arco (u,v) é ***sicuro*** *per A se e solo se A∪(u,v) é sottoinsieme di un qualche MST."*

É dunque sufficiente aggiungere N-1 archi sicuri per ottenere un MST, dove N é il numero di nodi.

Teorema: Sia A un sottoinsieme di qualche MST, (S,V-S) un *taglio* che rispetta A e (u,v) un arco *leggero* che *attraversa* (S,V-S). Allora (u,v) é sicuro per A.

Ancora una volta, una possibile implementazione dell'algoritmo utilizza lo Union-Find, dove gli insiemi disgiunti contengono i vertici connessi da archi sicuri. Ordinando gli archi rispetto al peso non decrescente, mediante FindSet si verifica che gli estremi dell'arco non facciano giá parte dello stesso insieme, dopodiché si chiama Union.
Sebbene faccia uso dello Union-Find, la risoluzione del problema dipende dagli archi dell'MST e non dai nodi collegati. A tal proposito dopo ogni chiamata a Union, l'insieme A viene aggiornato con l'arco sicuro appena trovato.
Questo algoritmo prende il nome **KruscalMST**.

2 Implementazione su Python

La maggiore difficoltá implementativa su *Python 3.6* é la scelta di un'adeguata struttura dati Union-Find, in modo tale da servire bene sia per l'algoritmo delle Componenti Connesse sia per KruscalMST.
In particolare, gli insiemi dinamici sono stati realizzati come liste concatenate

(*LinkedList*) con una testa, che corrisponde al rappresentante dell'insieme, e una coda, alla quale sono aggiunti i nuovi elementi. Ogni nodo presenta tre attributi: la chiave di riconoscimento (*data*), il puntatore al nodo successivo (*next*) e il puntatore al rappresentante della lista appartenente(*delegate*).
MakeSet crea una nuova lista e la inizializza tramite la funzione di inserimento di LinkedList, la quale aggiorna il valore del rappresentante. Dopodiché la aggiunge alla collezione.
Un secondo attributo della Union-Find, un dizionario di N elementi, é utilizzato per recuperare con tempo costante $\theta(1)$ il rappresentante di ogni nodo in **FindSet**.
Infine **Union** trova le liste di appartenenza dei due nodi e inserisce gli elementi di quella piú corta nella lista piú lunga, aggiornando il rappresentante sia nel nodo che nel dizionario. Il tempo di esecuzione dell'*euristica dell'unione pesata* é O(NlgN), in quanto il numero totale di operazioni é $\theta(N)$.

I grafi in genere possono essere implementati in due differenti modi:

- **Lista di Adiacenza**: un vettore Adj di N liste dove Adj[u] contiene tutti i nodi v tali che (u,v)∈E, con E l'insieme degli archi del grafo;
- **Matrice di Adiacenza**: una matrice NxN i cui elementi sono

$$a_{ij} = \begin{cases} 1 & \text{se(i,j)}\in\text{E} \\ 0 & \text{altrimenti} \end{cases}$$

Differiscono sia per lo spazio di memorizzazione e sia per il tempo di esecuzione per prelevare informazioni.
Inoltre per i grafi pesati, ogni arco ha un peso. In una matrice di adiacenza un valore diverso da 0 indica non solo la presenza dell'arco, ma anche il suo peso. Questa modalitá si presta bene per i problemi da risolvere. Di fatto, la creazione di un grafo richiede il numero di nodi (*numNodes*) e la probabilitá di avere un arco tra ogni coppia, dunque un valore tra 0 e 100. Mediante una funzione pseudo-casuale (*random()*) viene scelto il peso di ogni arco precedentemente creato.

L'algoritmo delle componenti connesse non richiede implementazioni particolari per le euristiche interne.

Quello di Kruscal, invece, necessita l'implementazione di una classe (*Arrow*) contenente una coppia di valori come attributi. Questo permette di memorizzare in A direttamente oggetti di tipo Arco, in modo da mostrare in modo diretto gli archi sicuri ottenuti dall'algoritmo. Infine, per ordinare gli archi rispetto al peso viene usata una funzione (*arcOrd*) che recupera dalla matrice di adiacenza un dizionario le cui chiavi sono archi e i valori sono i pesi. L'ordinamento in senso decrescente avviene mediante la funzione *sorted()*, nativa di Python, la quale restituisce una lista, convertita nuovamente in dizionario tramite *dict()*.

3 Gli esperimenti

Gli esperimenti eseguiti vogliono mettere in luce la correttezza e i tempi di esecuzione di *ConnectedComponents* e *KruscalMST*.
A tal proposito verranno creati grafi casuali di dimensione e probabilitá di presenza di archi crescenti, partendo da un grafo a 6 nodi e col 10% di probabilitá. L'incremento di entrambi é in funzione del valore attuale:

- N = N + int(N/4)
- prob = prob + (100 - prob)/10

Questo criterio permette di avere grafi con numero di nodi e archi non troppo elevati, in modo da facilitare la lettura dei dati e la visione dei grafici.
Inoltre coi primi test sará possibile osservare almeno due componenti connesse. I test continueranno ad essere eseguiti fin quando la probabilitá degli archi non supera il 75%. Dopo tale soglia si ottengo grafici potenzialmente troppo fitti per servire ad un'analisi utile.
La rappresentazione grafica del grafo, prima e dopo l'esecuzione, viene effettuata mediante il pacchetto *networkx* di Python. Tuttavia, a causa dei rallentamenti, nonché distorsione dei dati, dovuti al suo utilizzo, i grafici degli MST verranno mostrati solamente alla fine dell'esecuzione dei test. A tal proposito é sufficiente memorizzare in un vettore i risultati ottenuti di volta in volta.

Misurazione temporale: Il timer viene fatto partire un attimo prima della chiamata all'algoritmo e fermato un attimo dopo la terminazione.

Le misurazioni vengono eseguite 5 volte per grafo, in modo da ottenere un valore medio piú preciso. Tuttavia i tempi intermedi non verranno memorizzati, facendo riferimento alla sola media aritmetica. Tali valori sono recuperabili dai file *pickle* "*CCTime.p*" e "*MSTTime.p*" allegati.

Specifiche del calcolatore: gli esperimenti vengono eseguiti su un *ASUS Laptop X541UV* a 8GB di memoria RAM e con processore Intel Core i7-6500U a 2.50GHz di frequenza. L'architettura a 64-bit ospita come Sistema Operativo Windows 10 Home.

4 Risultati dei Test

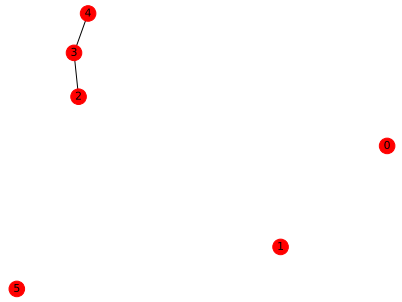
Il numero di dati prodotti dagli esperimenti é molto consistente: non sará possibile includerli tutti, grafici compresi, per cui, per una visione globale, si rimanda agli allegati (*Output.txt*). Le rappresentazioni dei grafi creati sono denominati "NXXPYY", dove XX é il numero di nodi e YY la probabilitá troncata al valore intero. Gli alberi MST, invece, sono denominati "NXXMST". Entrambi hanno estensione *.eps*.

4.1 Componenti Connesse

Come da previsione l'unico grafo che presenta un numero di CC superiore ad uno é il primo, quello con 6 nodi e 10% di probabilitá che vi siano archi:

13	0	0	0	0	0
0	51	0	0	0	0
0	0	0	0	0	0
0	0	74	0	61	0
0	0	0	15	0	0
0	0	0	0	0	0

Matrice di Adiacenza 6x6



Grafo a 6 nodi relativo alla matrice di fianco.

L'output delle componenti connesse é dunque:

List number 1	0		
List number 2	1		
List number 3	3	2	4
List number 4	5		

In effetti all'aumentare del numero di archi é molto probabile che ogni coppia di nodi siano collegati da un qualche cammino. Per questo motivo tutti gli altri grafi ottenuti hanno solo una componente connessa (vedi 1 e 2).

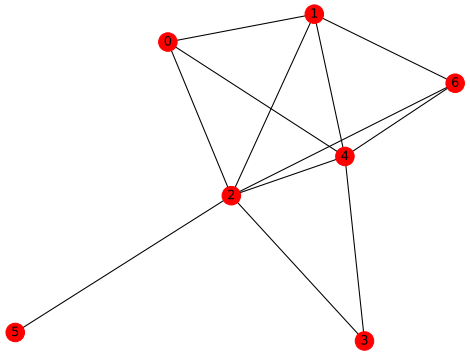


Figure 1: Grafo a 7 nodi.

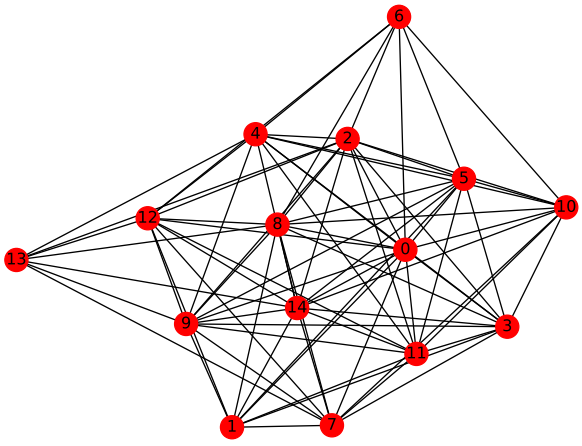


Figure 2: Grafo a 15 nodi

4.2 Alberi di Connessione Minimi

Tuttavia questo genere di grafi sono molto interessanti dal punto di vista del Minimum Spanning Tree: si veda gli alberi di connessione minimi ottenuti in 3 e 4. Gli archi dai quali sono formati sono raccolti nella tabella 4.2.

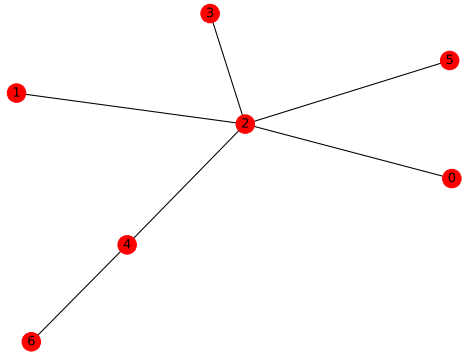


Figure 3: MST del grafo a 7 nodi

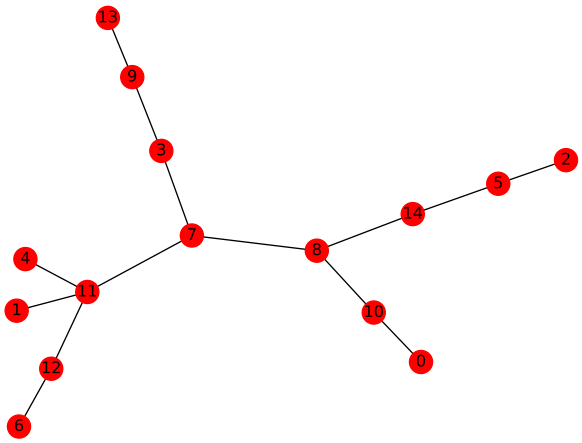


Figure 4: MST del grafo a 15 nodi

Per avere una visione completa sulla correttezza dei risultati si veda anche la matrice di adiacenza del grafo a 7 nodi nella tabella sottostante.

G7	G15
(5 , 2)	(7 , 8)
(2 , 3)	(11 , 4)
(4 , 2)	(7 , 3)
(1 , 2)	(3 , 9)
(4 , 6)	(5 , 14)
(0 , 2)	(6 , 12)
	(8 , 14)
	(5 , 2)
	(7 , 11)
	(10 , 8)
	(11 , 12)
	(10 , 0)
	(1 , 11)
	(13 , 9)

Archi del MST

99	90	56	0	75	0	0
0	0	37	0	76	0	0
0	0	0	7	83	0	84
0	0	48	0	0	0	0
0	0	11	53	0	0	55
0	0	6	0	0	76	0
0	90	0	0	0	0	0

Matrice di Adiacenza del grafo a 7 nodi

Quando un grafo contiene un'unica CC il numero di archi necessari a realizzare il MST é N-1, dove N é il numero di vertici.
La matrice di adiacenza é una modalitá di rappresentazione intuitiva, tuttavia non sempre é conveniente utilizzarla. Di fatto il grafo a 7 vertici contiene soli pochi archi, e la sua matrice é sparsa: si preferisce usare la lista di adiacenza.
Tuttavia gli esperimenti sono effettuati al crescere dei nodi e degli archi, pertanto l'uso delle matrici si rivela giusto.
Si considera dunque l'ultimo grafo realizzato, quello a 63 vertici, di cui, per ragioni di spazio, ne omettiamo la matrice: la probabilitá di creazione archi é oltre il 74%, dunque, mediamente, vengono implementate 3 relazioni ogni 4 coppie di nodi. Di fatto anche la rappresentazione tramite *networkx* (Figura 5) é troppo fitta per distinguere i molteplici archi.
Ció che é interessante osservare sono i tempi di esecuzione, in secondi, raccolti nella seguente tabella:

	Tempo 1	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**
CC	0.005607	0.006779	0.004136	0.004040	0.004173
MST	0.01049	0.01277	0.01638	0.01545	0.01412

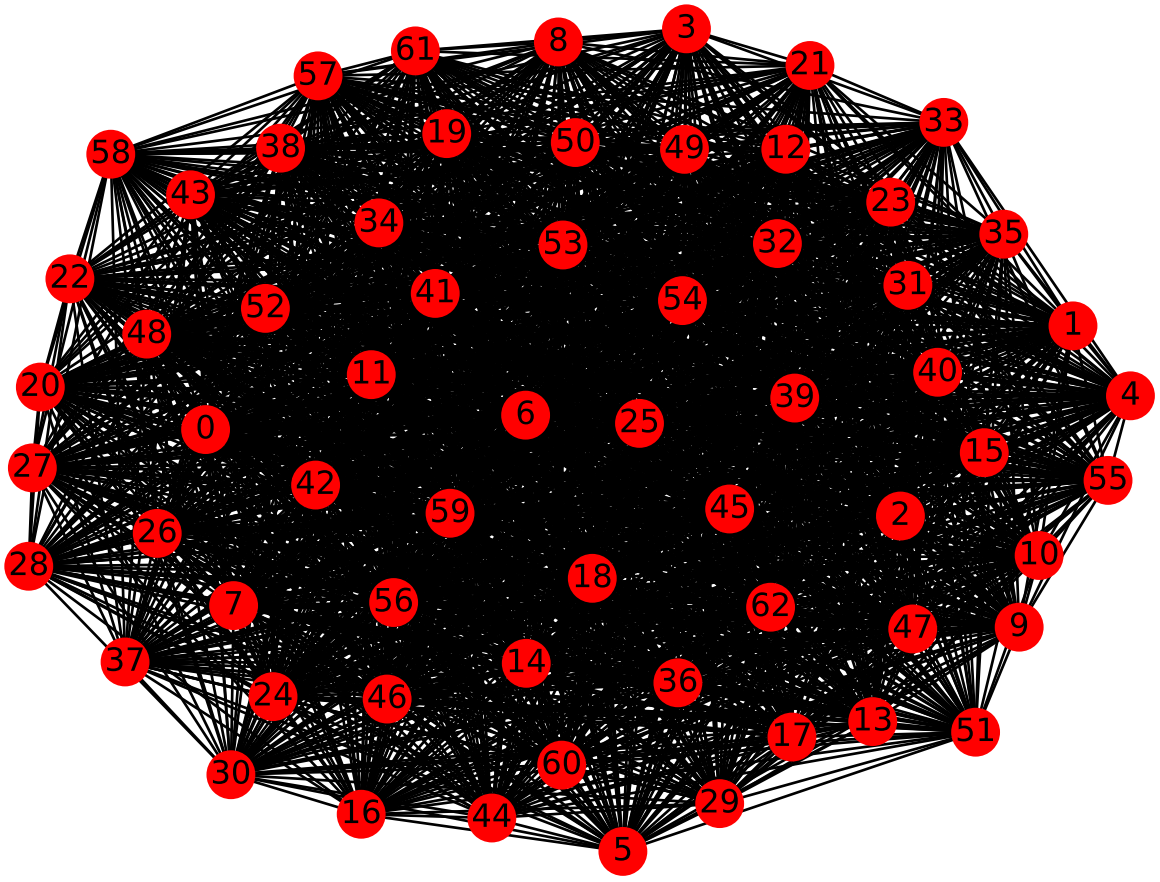


Figure 5: Grafo a 63 nodi

La prima osservazione riguarda la differenza dei tempi tra le due operazioni: KruscalMST impiega almeno un'unitá di grandezza in piú rispetto a CC, concorde con l'analisi eseguita precedentemente. La differenza si accentua all'aumentare dei nodi e degli archi, in quanto KruscalMST deve effettuare l'operazione di unione ad ogni arco che collega vertici che non appartengono allo stesso insieme. Anche CC effettua le operazioni di unione, ma non utilizza tempo computazionale per ordinare gli archi e recuperare le liste da unire dalla collezione di insiemi disgiunti. E poiché dipendono dalla dimensione dell'insieme dei nodi e degli archi, il divario tra i tempi di esecuzione aumenta in modo consistente. Si veda Figura 6.

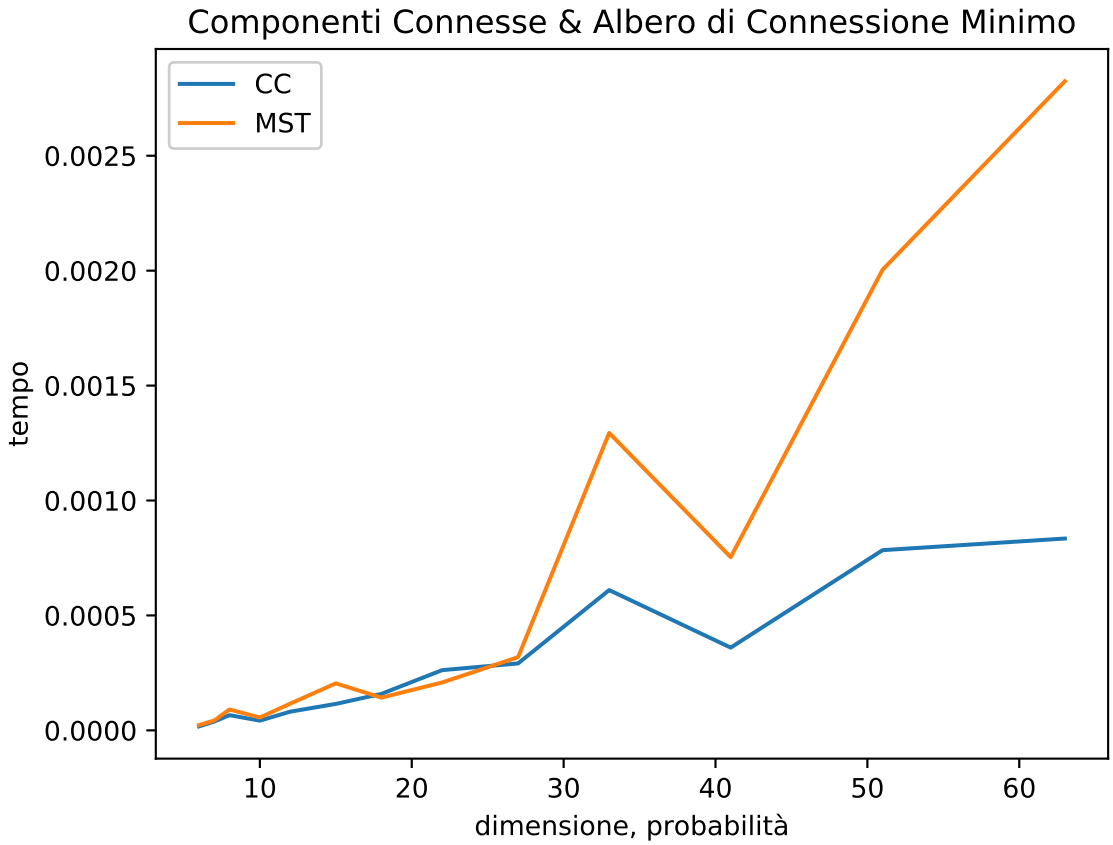


Figure 6: Divergenze temporali tra CC e KruscalMST al variare del numero di nodi e della probabilitá di avere archi nel grafo

	6, 10%	**7, 19%**	**8, 27%**	**10, 34%**	**12, 40%**	**15, 46%**	**18, 52%**
CC	0.00001746	0.0000384	0.00006622	0.00004242	0.00008138	0.0001151	0.0001587
MST	0.00002322	0.00004306	0.00009094	0.00005586	0.0001162	0.0002046	0.0001424
	22, 57%	**27, 61%**	**33, 65%**	**41, 68%**	**51, 71%**	**63, 74%**	**-**
CC	0.000262	0.0002912	0.0006104	0.000360	0.000784	0.000834	-
MST	0.0002084	0.0003184	0.001294	0.0007536	0.002004	0.002824	-

5 Conclusioni

Da quanto visto fino ad ora, la costruzione di grafi casuali permette di capire come operazioni, quali la ricerca delle Componenti Connesse e dell'Albero di Connessione Minimo, dipendano dal numero di nodi e archi. Di fatto rappresentano generici elementi e relazioni tra di essi, per cui ci si aspetta che dati molto connessi richiedano tempi di esecuzione molto maggiori. In particolare, l'implementazione della struttura dati Union-Find é decisiva ai fini della valutazione, poiché quasi tutte le operazioni su cui si basano CC e MST sfruttano metodi in essa definiti.
Per concludere, gli algoritmi che lavorano sui grafi devono essere implementati in modo da effettuare il numero minore possibile di visite ai vertici e/o archi, ovvero alle variabili su cui dipende il reperimento delle informazioni.