Componenti connesse e MST

Edoardo Branchi

February 2021

Contents

1 Introduzione

In questa relazione tratteremo lo studio necessario per generare in maniera casuale dei grafi non orientati, sia pesati che non pesati.

Il grafico sarà mostrato sia graficamente che testualmente, il codice produrrà anche la matrice corrispondente che successivamente sarà utilizzata per l'algoritmo di Prim e il calcolo dell'MST, infine sarà proposto uno studio sui tempi e le risorse necessarie per l'esecuzione.

2 Algoritmi utilizzati

2.1 Generazione dei grafi

2.1.1 Grafo casuale non pesato

Il codice che segue riporta lo snippet di codice necessario per la generazione di un grafo non pesato e non orientato.

Chiamando la funzione si ottiene un grafo con numero di nodi tra 2000 e 3000. Gli archi sono anch'essi generati in maniera casuale con il 60 % di possibilità [2] che una volta generati i nodi si crei un arco fra loro. Una volta eseguito il codice otteniamo a schermo un plot del grafo generato e la corrispondente matrice di adiacenza sottoforma di DataFrame Pandas.

```
1    def simple_graph_generation():
2        start_time = time.time()
3        time.sleep(1)
4        labels = string.ascii_lowercase  # prendo le lettere dell'
     alfabeto
5        label_array1 = list(labels)  # le metto nell'array nomi
     nodi
6        label_array = [''.join(comb) for comb in product(
     label_array1, repeat=3)] #generazione variabili per la df
7
8        nodes_number = random.randint(2000,3000)  # genero il
     numero di nodi
9        del label_array[nodes_number:]  # cancello dall'array le
     lettere che non mi servono
10
11       adjacency_matrix = np.zeros((nodes_number, nodes_number))
     # riempio la matrice con 0
12       adjacency_matrix[np.random.rand(*adjacency_matrix.shape) <
     0.6] = 1  # ce' la prob del 20% che uno 0 diventi un uno
13       adjacency_matrix = (adjacency_matrix + adjacency_matrix.T)
     / 2 #matrice simmentrica
14       df = pd.DataFrame(data=adjacency_matrix)  # converto la
     matrice in un dataframe
15       df = df.astype(int)  # faccio diventare il df int
16       df.columns = ['{}'.format(c) for c in label_array]  #
     associo le lettere alle colonne
17       df.index = ['{}'.format(c) for c in label_array]  # associo
      le lettere alle righe
18       np.fill_diagonal(df.values, 0)  # imposto la diagonale a 0
19       exec_time = time.time() - start_time
20       exec_time= exec_time % 1
21       exec_time= f'{exec_time:.10f}'
22       return exec_time, nodes_number
```

[2] E' possibile modificare questo parametro alla riga 11 del codice

	a	b	c	d	e	f	g	h	i	j	k	l
a	0	0	0	1	0	0	0	1	0	0	0	0
b	0	0	1	0	1	0	0	0	0	1	0	0
c	0	1	0	1	1	0	0	0	1	1	0	0
d	1	0	1	0	1	1	0	0	1	0	0	0
e	0	1	1	1	0	0	1	0	1	0	0	1
f	0	0	0	1	0	0	0	1	0	1	1	1
g	0	0	0	0	1	0	0	1	0	1	0	0
h	1	0	0	0	0	1	1	0	0	0	0	0
i	0	0	1	1	1	0	0	0	0	0	0	0
j	0	1	1	0	0	1	1	0	0	0	0	0
k	0	0	0	0	0	1	0	0	0	0	0	0
l	0	0	0	0	1	1	0	0	0	0	0	0

Figure 1: Esempio di un grafo non pesato generato dal codice precedente, i nodi sono stati ridotti per facilitare la visualizzazione

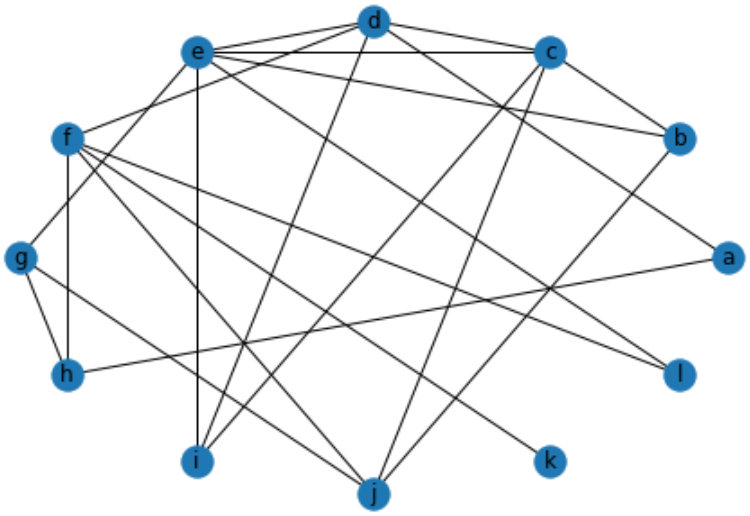

Figure 2: Esempio di un grafo non pesato generato dalla matrice in figura 1,i nodi sono stati ridotti per facilitare la visualizzazione

2.1.2 Grafo casuale pesato

Il seguente snippet genera un grafo pesato seguendo la logica del precedente con alcune differenze.

La principale differenza che possiamo osservare è che per generare un grafo pesato la matrice di adiacenza non conterrà solo 0 e 1 ma un numero intero compreso 1 e 10 (che sarà il peso dell'arco fra i due nodi considerati) con il 70 % di possibilità che esso sia uno 0 [3] e che quindi due nodi non siano connessi.

Nelle righe 32-34 sostituisco i valori 0 nella matrice di adiacenza con 99 [4] per comodità nella successiva esecuzione dell'algoritmo di Prim.

```
1     def weighted_graph_generation(graph_only):
2         start_time = time.time()
3         time.sleep(1)
4         nodes_number = random.randint(2000, 3000) # generazione
    numero di nodi casuali
5         labels = string.ascii_lowercase #prendo le lettere dell'
    alfabeto
6         label_array1 = list(labels)#le metto nell'array
7         label_array = [''.join(comb) for comb in product(
    label_array1, repeat=3)]
8
9
10
11        del label_array[nodes_number:] #cancello le lettere che non
     mi servono
12        adjacency_matrix = np.random.randint(1, 10, (nodes_number,
    nodes_number)) #popolo la matrice con n casuali da 1 a 10
13        adjacency_matrix[np.random.rand(*adjacency_matrix.shape) <
    0.7] = 0 #0.7 di probabilita' che un arco diventi 0
14        adjacency_matrix = (adjacency_matrix + adjacency_matrix.T)
    / 2 #rendo la matrice simmetrica
15        df = pd.DataFrame(data=adjacency_matrix) #trasformo la
    matrice in un df
16        df = df.astype(int) #casto a int la df
17        df.columns = ['{}'.format(c) for c in label_array] #assegno
     una lettera ad aogni colonna
18        df.index = ['{}'.format(c) for c in label_array] #assegno
    una lettera ad ogni riga
19        np.fill_diagonal(df.values, 0) #porto la diagonale a 0
20
21        if (graph_only==False):
22            create_spanning_tree(df,nodes_number)
23
24        exec_time = time.time() - start_time
25        exec_time = exec_time % 1
26        exec_time = f'{exec_time:.10f}'
27        return exec_time, nodes_number
```

[3] Doveroso far notare che la logica è speculare rispetto al grafo non pesato

[4] 99 è usato come valore placeholder in quanto non è possibile usare np.nan in una matrice di interi ma solo in una matrice di float

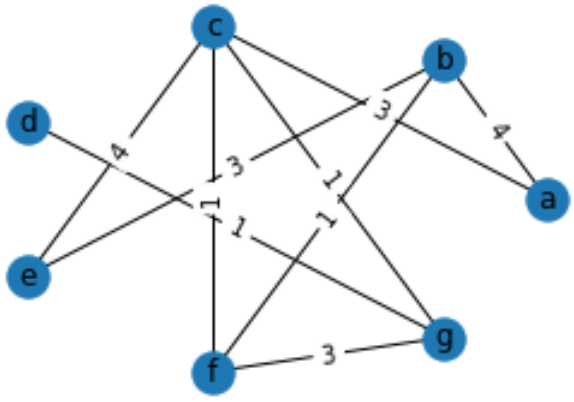

Figure 3: Esempio di un grafo pesato generato dal codice precedente

Matrice di adiacenza del grafo:

	a	b	c	d	e	f	g
a	0	4	3	0	0	0	0
b	4	0	0	0	3	1	0
c	3	0	0	0	4	1	1
d	0	0	0	0	0	0	1
e	0	3	4	0	0	0	0
f	0	1	1	0	0	0	3
g	0	0	1	1	0	3	0

Figure 4: Matrice di adiacenza corrispondente al grafico riportato in figura 3

2.2 Calcolo MST

Per il calcolo dell'MST è stato scelto di utilizzare l'algoritmo di Prim utilizzando una coda min-heap.

Prima dell'esecuzione vera e propria dell' algoritmo è necessaria una pulizia del dataframe che la funzione riceve in ingresso. Tale funzione permette di eliminare dalla mia tabella e di conseguenza dal mio i grafo i nodi "isolati" ovvero i nodi che non essendo connessi con nessun altro nodo hanno nella matrice di adiacenza la loro riga e la loro colonna composta da solo 0.

Alla riga 9 chiamo l'algoritmo di Prim.

L'algoritmo prende in ingresso il DF e il nodo di partenza[5] La prima parte (righe 2-9) si occupa di creare il mio heap con la funzione set() che mi terrà i nodi nell' heap nel giusto ordine e defaultdict() in modo tale da evitare di avere errori sulle chiavi. Utile anche notare che gli archi con costo pari a 99 non vengono presi in considerazione.

La seconda parte si occupa di ciclare sulla lista dei vertici, prendendo il primo elemento della lista, se il nodo verso cui sto andando non è stato visitato lo aggiungo alla lista "visited" e aggiungo il nodo attuale all'mst. Successivamente ciclo sul nodo di arrivo, se uno di quelli a cui è connesso non è stato visitato lo aggiungo all'heap e parto con la successiva iterazione del while.

L'algoritmo termina quando non ho più nodi da visitare.

```
1  def create_spanning_tree(df,nodes_number):
2      start_time = time.time()
3      print(len(df.index))
4      print("mst inizio")
5      for col in df.columns:      #sostituisco gli zeri della matrice
        con archi di peso 99
6          val = 99
7          df[col] = df[col].replace(0, val)
8      for col in df.columns:  # Loop sulle colonne del df
9          if len(df[col].unique()) == 1:  # trovo i valori unici
       nelle colonne, se ==1 allora la colonna ha tutti valori uguali
10             df.drop([col], axis=1, inplace=True)  # elimino la
       colonna, in quanto sarebbe un nodo isolato
11     df = df[df.std(axis=1) > 0] # elimino anche la riga del nodo
       isolato
12     graph = df.to_dict() #converto la df in un dizionario
13     starting_vertex = 'aaa' #chiamo prim
14
15
16     mst = defaultdict(set) #creo un defaultdict per evitare che mi
       siano lanciati errori "keyerror"
17     visited = set([starting_vertex]) #chiamo set() che mi tiene
       ordinati i nodi visitati
```

[5] In questo caso è assunto AAA come nodo di partenza

```
    #aggiunge i nodi connessi al mio vertice di partenza a edges
    try:
        edges = [ (cost, starting_vertex, to) for to, cost in graph
    [starting_vertex].items() if cost != 99]
        heapq.heapify(edges)
        #trasforma edges in un heap mettendo al primo posto della
    lista il nodo con minor costo connesso

        while edges:      #ciclo sui vertici
            cost, frm, to = heapq.heappop(edges) #pop(peso,partenza
    ,arrivo) del primo elemento dell'heap
            if to not in visited: #se non ho visitato l'elemento
    verso cui sto andanto
                visited.add(to)    #lo aggiungo alla lista visited
                mst[frm].add(to)   #aggiungo il nodo di partenza
    all' mst
                for to_next, cost in graph[to].items(): #ciclo sul
    nodo di arrivo
                    if to_next not in visited: #se uno dei nodi
    connessi al mio nodo di arrivo non e' stato visitato
                        heapq.heappush(edges, (cost, to, to_next))
    #lo aggiungo all'heap
        print("mst fine")
        exec_time = time.time() - start_time
        exec_time = f'{exec_time:.10f}'
        result=[exec_time,nodes_number]
        result_mst = open('result_mst.txt', 'a')
        result_mst.writelines(str(result) + "\n")
        result_mst.close()
        exec_time=0
        nodes_number=0

    except Exception as e:
        print(e)
```

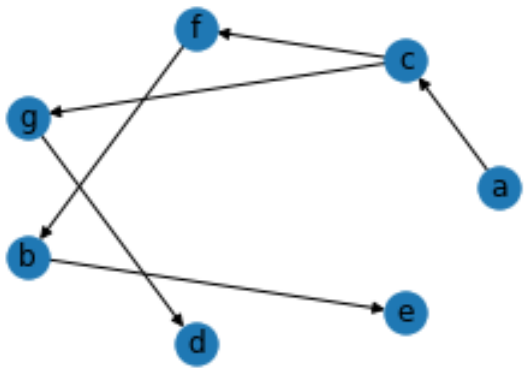

Figure 5: Output grafico dell'MST sul grafo riportato in figura 1 e 2

```
MST:

{'a': {'c'},
 'b': {'e'},
 'c': {'f',
       'g'},
 'f': {'b'},
 'g': {'d'}}
```

Figure 6: Forma testuale dell'MST in figura 3

3 Modalità di esecuzione dei test

I test sono stati eseguiti in maniera equivalente su tutte e 3 le funzioni.

Saranno esaminate le tempistiche per generare 200 grafi,pesati e non pesati, con un numero di nodi compreso fra 2000 e 3000.

L'algoritmo per il calcolo dell'MST prenderà poi in ingresso i grafi pesati generati precedentemente, quindi opererà anch'esso su 200 grafi con un numero di nodi compreso fra 2000 e 3000.

Durante il calcolo di tutti e tre gli algoritmi è stata disattivata qualunque tipo di esecuzione in background ed è solamente stata attivata la scrittura su file per riportare i dati dei test, quindi i risultati sotto riportati fanno rifermento solo all'esecuzione.

Inoltre durante i test di

```
Create_spanning_tree()
```

il tempo è calcolato solo per l'esecuzione dll'algoritmo di Prim e non tiene conto della generazione del grafo. Tutti i test sono stati eseguiti 200 volte ciascuno e in single thread.

4 Risultati attesi

Dal codice di generazione dei grafi mi aspetto un tempo di generazione di $\mathcal{O}(n^2)$ in quanto è necessario generare una matrice $n \times n$ e popolarla con 0 e 1 nel caso del grafo non pesato e con numeri da 1 a 10 nel grafo pesato, in questo caso però probabilmente i tempi saranno leggermente maggiori perchè, per come è stato implementato il codice la matrice è visitata due volte, la prima per popolarla la seconda per cambiare con una certa probabilità i valori in 0.

Inoltre, mi aspetto che all'aumentare del numero di nodi aumenti in maniera lineare anche il tempo necessario per generarli.

Per quanto riguarda il calcolo dell' MST mi aspetto un tempo medio di $\mathcal{O}(V^2)$ visto l'utilizzo della matrice di adiacenza, dove V è il numero dei vertici del grafico preso in considerazione.

Viste queste considerazioni per conoscenza si riportano le caratteristiche del computer su cui sono stati eseguiti i test che seguono:

- Intel Core i7-3610QM @ 2.30Ghz.
- 2x2Gb SODIMM Ram @ 1333 Mhz [6].

Per la parte di calcolo dell'MST esattamente come le generazioni mi aspetto un aumento del tempo necessario al calcolo all' aumentare del numero di nodi ottenuti dalla generazione.

[6] Visto la particolare tecnologia delle RAM è doveroso specificare che durante l'esecuzione il computer era alimentato

5 Risultati ottenuti

5.1 Generazione grafo non pesato

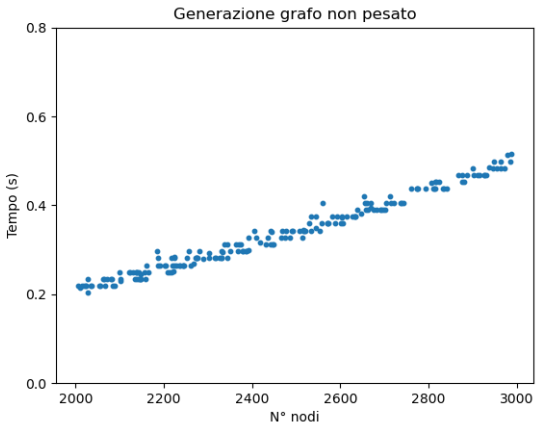

Figure 7: Grafico Scatter che mostra il tempo di generazione dei grafi non pesati in base al numero di nodi

5.2 Generazione grafo pesato

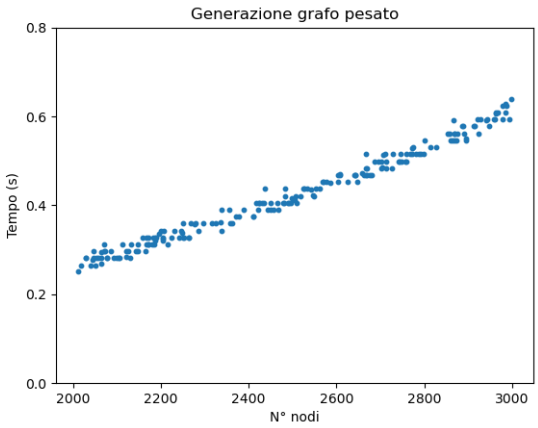

Figure 8: Grafico Scatter che mostra il tempo di generazione dei grafi pesati in base al numero di nodi

5.3 Calcolo MST

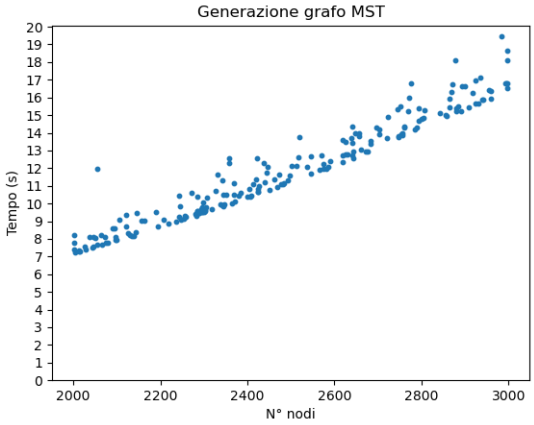

Figure 9: Grafico Scatter che mostra il tempo di calcolo dell'MST in base al numero di nodi

6 Conclusioni

I risultati ottenuti rispecchiano le ipotesi formulate prima di eseguire i test ma è doveroso fare alcune precisazioni.

Nella generazione dei grafi il tempo necessario alla generazione aumenta aumentando il numero di nodi. La generazione dei grafi pesati e non pesati segue lo stesso andamento con una buona consistenza ma nel caso di grafi pesati c'e' un ritardo di circa 0.05 secondi rispetto ai grafi non pesati, la generazione del grafo pesato prima popola la matrice di adiacenza con dei numeri casuali da 1 a 10 e poi la ispeziona nuovamente eliminando con una probabilità del 70% alcuni archi, questa seconda chiamata potrebbe essere la causa del "ritardo" rispetto alla generazione del grafo non pesato.

Per quanto riguarda il calcolo dell'MST il grafico alla figura 9 è molto simile allo stesso grafico per le generazioni dei due grafi, con una scala temporale ben diversa, ma presenta meno compattezza nella distribuzione dei risultati, forse dovuto al fatto che il tempo necessario per la generazione di un grafo da 3000 nodi è molto più lungo di quello necessario per la generazione di uno da 2000 nodi, questa grande escursione temporale porta ad una distribuzione meno uniforme dei risultati, oppure potrebbero essere dati da problemi legati alla macchina su cui è eseguito il programma.

Inoltre il grafico dell'MST conferma l'ipotesi della complessità pari a $\mathcal{O}(V^2)$.