UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di Ingegneria Elettrica, Elettronica e Informatica

Corso di Laurea Magistrale in Ingegneria Informatica

Andrea Arciprete

ELABORATO FINALE

Node Pruning using Geometric Deep Learning to accelerate the process of finding all maximum cliques of a graph

TESI DI LAUREA

RELATORE

Prof. Giuseppe Mangioni

CORRELATORI

Prof.ssa Vincenza Carchiolo

Ing. Marco Grassia

ANNO ACCADEMICO 2020-2021

Ringraziamenti

In primo luogo, desidererei ringraziare il Prof. Giuseppe Mangioni, la Prof.ssa Vincenza Carchiolo e l'Ing. Marco Grassia per avermi dato la possibilità di approfondire una delle tematiche più interessanti degli ultimi anni, ossia il Geometric Deep Learning.

A posteriori, dopo aver concluso il lavoro, ritengo che la tematica affrontata nel presente lavoro di tesi si è rivelata la migliore che io potessi scegliere in quanto mi ha consentito di consolidare ancora di più le mie conoscenze pregresse riguardanti il Machine Learning e il Deep Learning e, soprattutto, mi ha permesso di approfondire un'ulteriore evoluzione del Deep Learning, ossia il Geometric Deep Learning dandomi, dunque, la possibilità di acquisire una conoscenza ancora più dettagliata circa le varie tecniche di Machine Learning attualmente esistenti.

Di conseguenza, è doveroso ringraziare i relatori sopra citati sia per avermi dato la possibilità, grazie al presente lavoro di tesi, di comprendere che l'ambito del Machine Learning è appassionante e che, dunque, rappresenta un valido sbocco lavorativo sia per essere stati sempre disponibili a risolvere i problemi che si sono verificati durante la stesura della tesi sia per avermi fornito dei consigli molto preziosi atti ad accrescere la qualità del lavoro svolto.

In secondo luogo, desidererei ringraziare una persona per la quale non basterebbe nemmeno un capitolo, ossia Daniela. Grazie a lei ho acquisito autostima, mi sono migliorato sempre di più, ho superato momenti che definire difficili è dir poco e, soprattutto, ho compreso che lavorare in team è una delle cose più belle che mi potessero capitare.

Noi due siamo la prova inconfutabile del fatto che lavorare insieme, aiutarsi sempre e affrontare tutto insieme consente realmente di dividere i compiti e moltiplicare il successo.

Lei c'è sempre stata, mi ha sempre supportato e anche sopportato, ha fatto sì che i pomeriggi di studio che sembrava non finissero mai, in realtà, una fine ce l'avessero.

Questi cinque anni davvero intensi, provanti sia da un punto di vista fisico che mentale sono convinto che non li avrei mai superati se non ci fosse stata lei a supportarmi.

Se ho raggiunto questo livello di apprendimento, se ho acquisito un numero di CFU che sembrava non finissero mai è, soprattutto, merito suo. Di conseguenza, per tutte le cose che ho detto e anche per quelle per cui non ho proferito parola, sono sicuro che, all'interno del mio cuore, ci sarà sempre un posto speciale riservato appositamente per lei.

In conclusione, desidererei ringraziare i miei amici, i miei colleghi e la mia squadra di calcio *smdg*, una seconda famiglia che nel corso di questi anni mi ha fornito quella dose di spensieratezza che mi è stata davvero utile per affrontare questo percorso universitario.

Un ringraziamento particolare va ai miei genitori che nel corso di questi anni hanno fatto non pochi sacrifici affinché io potessi concludere con successo il mio percorso di studi.

Infine, un ringraziamento va a me stesso per il fatto di non essermi mai arreso nonostante le innumerevoli difficoltà che ho incontrato lungo il cammino, per il fatto di averci sempre creduto e di essere riuscito a trasformare i momenti in cui ero sul punto di mollare in momenti in cui ho continuato a spingere di più per raggiungere l'obiettivo.

Indice

Capitolo 1

Introduzione

"Io penso che il prossimo secolo sarà il secolo della complessità"

Stephen Hawking

Oggigiorno basta guardarsi intorno per realizzare che siamo circondati da sistemi che definire complessi è alquanto riduttivo. Consideriamo, ad esempio, la società di tutti i giorni in cui abbiamo miliardi di persone che interagiscono tra loro oppure le infrastrutture di comunicazione costituite da miliardi di smartphone, computer, ecc. che comunicano tra loro.
Quelli appena citati sono esempi di sistemi che vengono definiti complessi.
Un altro esempio di sistema estremamente complesso è il nostro cervello, il quale è costituito da miliardi di neuroni interconnessi tra loro e ci consente di ragionare e comprendere gli aspetti del mondo che ci circonda.
Dato il ruolo estremamente importante che i sistemi complessi ricoprono nella vita di tutti i giorni, riuscire a studiarli e a formulare una descrizione matematica di essi è estremamente importante e, soprattutto, costituisce una delle maggiori sfide del ventunesimo secolo.
Al fine di fronteggiare le sfide appena citate e risolvere una serie di problemi del mondo reale, all'inizio del ventunesimo secolo, è nata una nuova disciplina denominata Network Science [1], la quale ha come obiettivo lo studio delle cosiddette **reti complesse (complex networks)**, utilizzate per modellare i sistemi complessi.
Spesso, però, tali **reti** sono talmente grandi da rendere una serie di problemi di Network Science (come quello della determinazione delle clique di dimensione massima di un grafo) computazionalmente non fattibili a causa dei tempi di computazione estremamente elevati richiesti per la loro risoluzione.
Un modo attraverso il quale è possibile ridurre i tempi di computazione di questi problemi al fine di risolverli in tempi ragionevoli consiste nell'utilizzare il Machine Learning o, ancora meglio, una sua evoluzione: il Geometric Deep Learning.
Tra i vari problemi di Network Science che è possibile risolvere in tempi ragionevoli grazie all'ausilio del Machine Learning o del Geometric Deep Learning è possibile citare il problema della **determinazione delle clique di dimensione massima di un grafo**, di cui ci siamo occupati all'interno del presente lavoro di tesi.

1.1 Obiettivo e Contenuti

Lo scopo della tesi è presentare un modello di Geometric Deep Learning che, dato un grafo *x* in ingresso, in uscita, per ogni singolo nodo del grafo *x*, predice se quel nodo appartiene ad una clique di dimensione massima oppure no. Sfruttando il modello che presenteremo all'interno del presente lavoro di tesi riusciremo ad attuare quello che viene chiamato **node pruning**. In particolare, una volta fornito un grafo *x* in ingresso al nostro modello ricaveremo le predizioni prodotte in uscita da esso al fine di attuare un'operazione di **node pruning** volta a rimuovere i nodi presenti all'interno del grafo *x* che secondo quanto predetto dal modello non fanno parte di alcuna clique di dimensione massima.

Operando in tal modo, a partire dal grafo *x*, ricaveremo un grafo *y* che, rispetto al grafo *x*, è di dimensione ridotta in quanto contiene un numero inferiore di nodi e archi.

Dopo aver fatto ciò, piuttosto che applicare il Solver (algoritmo in grado di determinare le clique di dimensione massima di un grafo) direttamente al grafo originario *x*, il che farebbe lievitare enormemente i tempi di computazione richiesti per la risoluzione del problema, applicheremo il Solver al grafo *y*, ossia al grafo di dimensione ridotta risultante dall'operazione di node pruning attuata sul grafo originario *x*.

Così facendo, dal momento che il Solver non verrà applicato al grafo originario *x* (il quale può essere di dimensioni significative) ma verrà applicato al grafo di dimensioni ridotte *y*, le clique di dimensione massima verranno determinate in tempi significativamente inferiori rispetto a quelli che sarebbero necessari se applicassimo il Solver direttamente al grafo originario *x*.

Operando in questo modo tenteremo di rispondere positivamente alla seguente domanda:

- È possibile accelerare la risoluzione del problema di tipo NP-Hard: **determinazione delle clique di dimensione massima di un grafo** mantenendo dei livelli di accuratezza accettabili?

Per quanto concerne i contenuti:

Capitolo 2: all'interno di questo capitolo verrà data una panoramica della Network Science e di come i sistemi complessi vengono modellati. Inoltre, parleremo dei grafi, delle caratteristiche che li accomunano, del concetto di **clique** e, infine, approfondiremo nel dettaglio il problema che vogliamo risolvere in tempi ragionevoli, ossia il problema della **determinazione delle clique di dimensione massima** di un grafo e vedremo quali algoritmi (Solver) possono essere utilizzati per risolvere tale problema.

Capitolo 3: esso si prefigge di descrivere teoricamente il concetto di Machine Learning, il quale, come detto, è di fondamentale importanza per rendere computazionalmente fattibili i problemi di tipo NP-Hard. Inoltre, verrà data una descrizione del Deep Learning e del Geometric Deep Learning, il quale, di fatto, costituisce un'evoluzione del Deep Learning in quanto può essere visto come un Deep Learning applicato ai grafi.

Capitolo 4: questo capitolo rappresenta un punto focale della tesi in quanto al suo interno verrà messa in risalto la strategia che è stata adottata per ridurre i tempi di computazione necessari per risolvere il problema della determinazione delle clique di dimensione massima di un grafo. Nello specifico, presenteremo il modello di Geometric Deep Learning utilizzato, le features utilizzate e il procedimento che abbiamo adottato per ridurre i tempi di computazione del problema appena citato.

Capitolo 5: questo capitolo costituisce la parte conclusiva della tesi in cui verrà evidenziato il dataset utilizzato, i risultati che sono stati raggiunti e le conclusioni.

1.2 Progetti Correlati

La riduzione dei tempi di computazione necessari per la risoluzione del problema della determinazione delle clique di dimensione massima di un grafo è un tema molto attuale anche se, in realtà, già nel passato ci sono stati alcuni tentativi di risolvere lo stesso problema utilizzando tecniche diverse da quelle utilizzate nel presente lavoro di tesi. Questi tentativi verranno messi in risalto all'interno di questo paragrafo.

Il punto da cui occorre sicuramente partire ogni qualvolta si vogliono ridurre i tempi di risoluzione del problema della determinazione delle clique di dimensione massima di un grafo è il lavoro svolto da *Lauri et al*.

Essi hanno proposto un framework di Machine Learning attraverso il quale è possibile ridurre i tempi di risoluzione del problema della determinazione delle clique di dimensione massima di un grafo [2]. Quella adottata da *Lauri et al.* ha il pregio di essere una delle prime soluzioni attraverso la quale è possibile raggiungere l'obiettivo sopracitato garantendo, al tempo stesso, dei livelli adeguati di accuratezza. Purtroppo, però, questa soluzione presenta delle limitazioni:

1. Il training del modello di Machine Learning è stato effettuato utilizzando grafi provenienti tutti dallo stesso dominio.
2. È stato adottato il pruning single-stage.

Al fine di superare le limitazioni appena citate, *Grassia et al.* hanno proposto un modello di Machine Learning più avanzato rispetto a quello proposto da *Lauri et al.*, il quale è stato allenato considerando grafi appartenenti a una varietà di domini differenti e utilizzando il pruning multi-stage [3].

Il presente lavoro di tesi, di fatto, rappresenta un'ulteriore evoluzione di quanto fatto da *Grassia et al.* in quanto piuttosto che utilizzare un modello di Machine Learning classico, il quale è adatto per i dati Euclidei e non per i grafi, è stato utilizzato un modello di Geometric Deep Learning, il quale è adatto per i grafi.

Capitolo 2

Network Science e Grafi

2.1 Network Science

La definizione di Network Science viene data per la prima volta nel 2005 quando il National Research Council la definisce come una scienza che si occupa “dello studio di come rappresentare i fenomeni fisici, biologici e sociali mediante delle reti al fine di realizzare dei modelli predittivi di questi fenomeni” [4].

Dal momento che ciò che accomuna le diverse reti in grado di rappresentare i fenomeni appena citati è la complessità possiamo affermare che la Network Science focalizza la propria attenzione sullo studio delle cosiddette **reti complesse (complex networks)**, le quali vengono utilizzate per modellare i cosiddetti sistemi complessi.

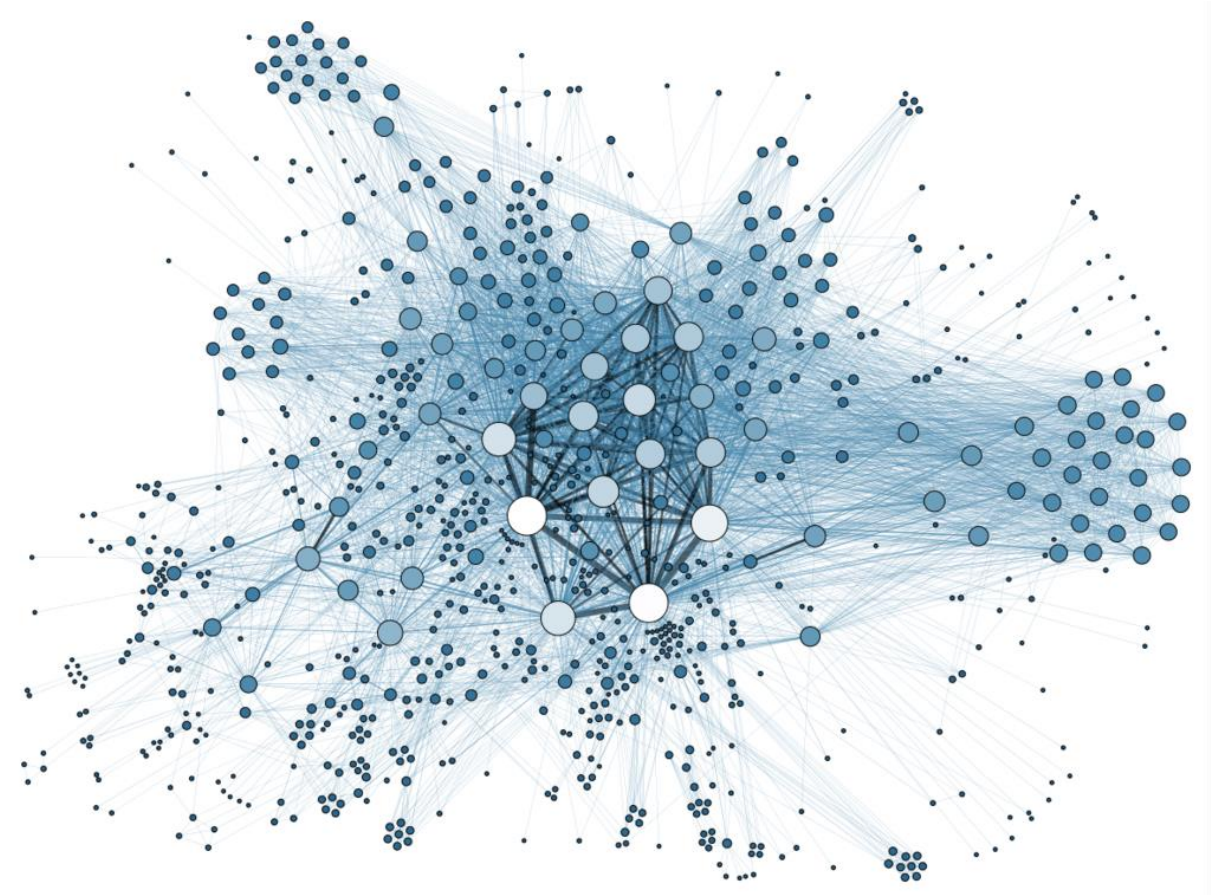


Figura 2.1.1: Esempio di rete complessa.

In maniera più precisa possiamo affermare che le **reti complesse (complex networks)** consentono di codificare i componenti di un sistema complesso e le interazioni che sussistono tra di essi.

La società in cui viviamo è permeata da una vasta gamma di reti complesse, tra le quali è possibile citare:

- Le reti in grado di codificare le interazioni tra i geni, le proteine e i metaboliti.
- Le reti neurali, le quali sono costituite da miliardi di neuroni interconnessi tra di loro.
- Le reti sociali in grado di codificare tutte le relazioni di amicizia, familiari e professionali che costituiscono le fondamenta della società in cui viviamo.
- Le reti di comunicazione, le quali descrivono le interazioni tra i dispositivi che interagiscono tra loro per mezzo di collegamenti cablati o wireless.
- Le cosiddette *power grids*, ossia le reti costituite da generatori e linee di trasmissione.
- Le reti di commercio, le quali consentono di comprendere come le persone si scambiano tra loro beni e servizi.

In aggiunta a quanto affermato finora possiamo affermare che la Network Science è emersa tantissimo in questi ultimi dieci anni per due ragioni che andremo ad evidenziare.

In primo luogo, è grazie alla diffusione delle cosiddette *network maps* che la Network Science è diventata molto popolare. Al fine di descrivere in maniera dettagliata il comportamento di un sistema complesso costituito da miliardi di componenti che interagiscono fra loro è necessario avere a disposizione una mappa di tale sistema. Ad esempio, se consideriamo un sistema complesso come quello modellato da una rete sociale un esempio di mappa che dobbiamo assolutamente conoscere è quella contenente le persone che fanno parte della rete sociale e le relazioni che sussistono tra di esse.

Grazie all'avvento di Internet e allo sviluppo tecnologico abbiamo a disposizione una grandissima varietà di dataset che ci consentono di costruire le mappe di cui abbiamo bisogno per comprendere approfonditamente il comportamento di questi sistemi complessi.

In secondo luogo, la Network Science è diventata molto popolare in questi ultimi anni anche perché si è visto che prendendo in considerazione architetture di reti appartenenti a domini differenti, comunque, esse presentano delle caratteristiche comuni. La conseguenza di ciò è che è possibile applicare una stessa famiglia di tools, algoritmi e soluzioni ad un'ampia varietà di domini differenti [1].

2.2 Reti e Teoria dei Grafi

Una rete, la quale, come detto, consente di rappresentare un sistema complesso può essere definita come un catalogo contenente tutti i componenti di un sistema complesso che vengono chiamati **nodi** o **vertici** e le relazioni che sussistono tra di essi che vengono chiamati **link** o **archi**. Nella letteratura scientifica i termini: **rete** e **grafo** vengono utilizzati in maniera interscambiabile. Solitamente, quando si parla di reti i componenti di una rete vengono chiamati nodi e le relazioni che sussistono tra di essi vengono chiamati link mentre quando si parla di grafi i componenti vengono chiamati vertici mentre le relazioni che sussistono tra di essi prendono il nome di archi. Come detto, però, questi termini possono essere usati in maniera interscambiabile. Allo scopo di essere chiari, premettiamo che da ora in poi useremo sempre il termine grafo ed in particolare parleremo di nodi e di archi.

2.2.1 Teoria dei Grafi

Un grafo è una struttura relazionale composta da un insieme finito di oggetti denominati **nodi** o **vertici** e da un insieme di relazioni tra coppie di oggetti dette **archi**.
La definizione di grafo ha, da un punto di vista strettamente matematico, delle notazioni e tra le possibili notazioni utilizzeremo la seguente:

$$G(N,A)$$

dove N indica l'insieme dei nodi del grafo G e A indica l'insieme degli archi del grafo G.

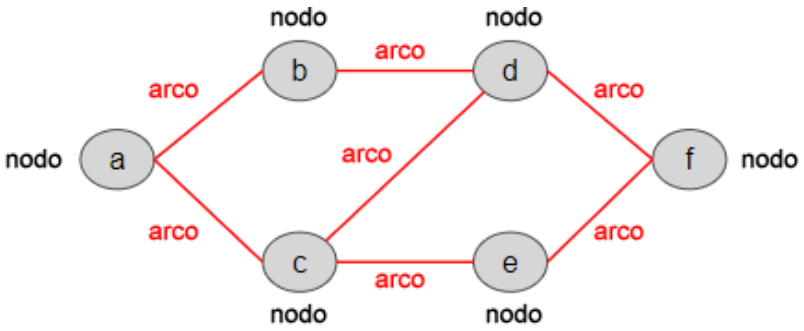


Figura 2.2.1.1: Esempio di grafo.

L'importanza che assume questa struttura logica è da attribuire alla constatazione che più di una "struttura reale" può essere schematizzata utilizzando i grafi, da una rete stradale (dove i nodi sono gli incroci e gli archi le strade) ad un programma di calcolo (dove i nodi rappresentano le istruzioni ed esiste un arco tra due nodi se le relative istruzioni possono essere eseguite in successione) oppure una struttura dati (dove i nodi rappresentano i dati semplici e gli archi i legami tra i diversi dati realizzabili, ad esempio, tramite puntatori).

L’origine storica della teoria dei grafi è generalmente fatta risalire ad un lavoro sviluppato da Eulero nel 1736 in cui veniva data una risposta ad un famoso quesito matematico noto come **problema dei ponti di Königsberg**.

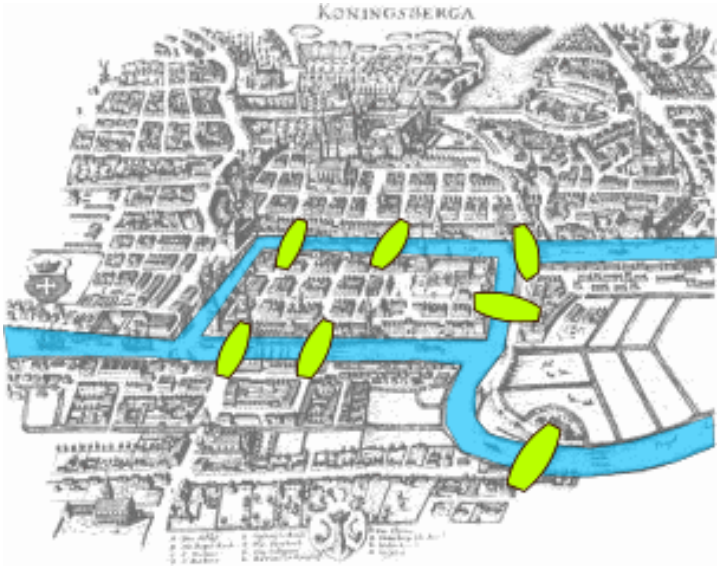


Figura 2.2.1.2: Mappa di Königsberg ai tempi di Eulero.

Il problema risolto da Eulero è stato proposto nel seguente modo:

“*Come in ogni cittadina tedesca, era uso anche a Königsberg che alla domenica gli abitanti facessero la loro passeggiata per le vie della città; era possibile progettare un percorso che permettesse a ciascuno di partire da casa e tornarvi dopo aver attraversato ciascun ponte una ed una sola volta?*”

Al fine di risolvere il problema Eulero trasformò la mappa di Königsberg in un grafo semplicemente sostituendo ogni singola riva del fiume e isola presente nella mappa con un nodo ed ogni ponte con un arco. Dopo aver operato tale trasformazione, Eulero ha osservato che il problema aveva soluzione soltanto nei casi in cui ogni nodo avesse grado pari e non ammetteva soluzione nel caso in cui almeno un nodo avesse un grado dispari. In virtù di queste considerazioni Eulero enunciò il suo **teorema**:

“*Condizione necessaria e sufficiente affinché un grafo sia percorribile completamente partendo da un nodo e ritondandovi passando una volta solamente per ciascun arco è che esista un percorso fra ogni coppia di nodi e che ogni nodo sia toccato da un numero pari di archi*”

A causa del fatto che il teorema di Eulero non viene soddisfatto il problema dei ponti di Königsberg non ha soluzione.

Al fine di affrontare i problemi relativi alla teoria dei grafi, è necessario disporre di alcune definizioni di base.

GRAFO ORIENTATO E NON ORIENTATO

In generale, gli archi rappresentano una relazione fra una coppia di nodi; se tale coppia è ordinata, cioè se gli archi hanno una **testa** o **nodo di arrivo** ed una **coda** o **nodo di partenza**, il grafo si dice **orientato**, dove **(i,j)** indica un arco diretto dal nodo **i** al nodo **j**.

In un grafo orientato, presa la coppia di nodi **(i,j)**, **i** è detto **predecessore** di **j** e **j** è detto **successore** di **i**.

Figura 2.2.1.3: Esempio di arco orientato.

Nel caso in cui tutte le coppie di nodi presenti nel grafo non siano ordinate il grafo si dice **non orientato**.

MULTIGRAFO E GRAFO SEMPLICE

Un grafo nel quale esista più di un arco tra almeno una coppia di nodi è detto **multigrafo**, altrimenti si parla di **grafo semplice**.

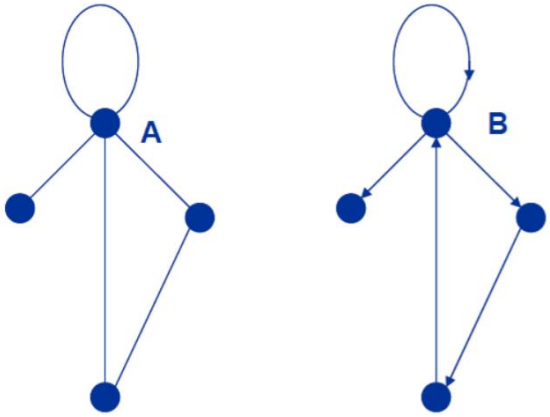


Figura 2.2.1.4: Esempio di grafo semplice non orientato (A) e grafo semplice orientato (B).

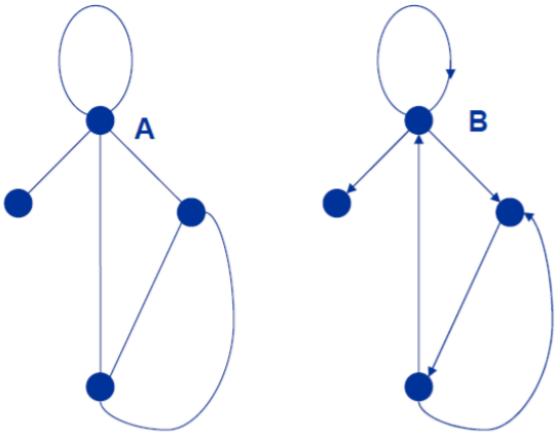


Figura 2.2.1.5: Esempio di multigrafo non orientato (A) e multigrafo orientato (B).

NODI ADIACENTI, ARCHI ADIACENTI E CAMMINO

Un nodo di un grafo non orientato si dice **adiacente** ad un altro nodo del grafo se esiste un arco che li congiunge; se un nodo non ha nodi adiacenti è detto **nodo isolato**.

Definito che due archi si dicono **adiacenti** se esiste un nodo che è estremo per entrambi, in un *grafo non orientato* è detto **cammino** un insieme di archi $\mathbf{(a_1,a_2,\ldots,a_n)}$ tale per cui a_i e a_{i+1} sono adiacenti.

In un *grafo orientato* esistono due tipi di cammino:

- **cammino non orientato**: non pone vincoli all'orientamento degli archi.
- **cammino**: richiede che la sequenza di archi sia tale che la testa di un arco coincida con la coda del successivo.

Se un cammino non passa mai due volte per lo stesso nodo è detto **cammino elementare**, se invece non passa mai due volte per lo stesso arco è detto **cammino semplice**.

DAL GRAFO CONNESSO AL GRAFO COMPLETO

Un grafo tale per cui per ogni coppia di nodi esiste un cammino che li unisce è detto **grafo connesso**. Nel caso di grafi orientati, se esiste un cammino per ogni coppia di nodi si dice che il **grafo è fortemente connesso**, mentre se esiste solo un cammino non orientato fra ogni coppia di nodi il **grafo è debolmente connesso**.

Un grafo (orientato o non orientato) si dice **completo** se per ogni coppia di nodi esiste un arco che li congiunge.

GRAFO PESATO E GRAFO NON PESATO

Un grafo si dice **pesato** se ad ogni singolo arco del grafo viene associato un peso, mentre si dice **non pesato** se ad ogni singolo arco del grafo viene associato un peso unitario [5].

GRADO

Nel caso di grafo **non orientato** si definisce **grado** k_i di un nodo i il numero totale di archi che incidono su tale nodo i. Ad esempio, se consideriamo un grafo delle chiamate che un certo numero di persone hanno fatto tra loro possiamo affermare che il grado di un certo nodo è il numero di chiamate che quella determinata persona ha fatto e ha ricevuto da un insieme di altre persone. Sempre nel caso di grafo **non orientato** il numero totale di archi L può essere calcolato tramite la seguente formula:

$$L = \frac{1}{2}\sum_{i=1}^{N} k_i$$

In pratica, dopo aver calcolato il grado di ogni singolo nodo del grafo non orientato, basta sommare i vari gradi e infine dividere il risultato per 2. Così facendo otteniamo il numero totale di archi presenti nel grafo non orientato.

Nel caso di grafo **orientato** occorre distinguere tra **grado in entrata (incoming degree)** k_i^{in} di un nodo *i* e **grado in uscita (outgoing degree)** k_i^{out} di un nodo *i*.

Il primo rappresenta il numero di archi che puntano al nodo *i*, mentre il secondo rappresenta il numero di archi che partono dal nodo *i*.

Sempre nel caso di grafo **orientato** il **grado totale** k_i di un nodo *i* è dato dalla seguente formula:

$$k_i = k_i^{in} + k_i^{out}$$

GRADO MEDIO

In un grafo **non orientato** il grado meglio del grafo può essere calcolato tramite la seguente formula:

$$< k > = \frac{1}{N} \sum_{i=1}^{N} k_i$$

dove *N* è il numero di nodi del grafo e k_i è il grado del nodo *i-esimo*.

In un grafo **orientato** il grado medio del grafo può essere calcolato tramite la seguente formula:

$$< k^{in} > = \frac{1}{N} \sum_{i=1}^{N} k_i^{in} = < k^{out} > = \frac{1}{N} \sum_{i=1}^{N} k_i^{out}$$

DISTRIBUZIONE DEI GRADI

La **distribuzione dei gradi (degree distribution)** p_k, è la probabilità normalizzata che un nodo qualsiasi (random) abbia grado *k*. Essa può essere calcolata mediante la seguente formula:

$$p_k = \frac{N_k}{N}$$

dove N_k è il numero di nodi del grafo aventi grado pari a *k* mentre *N* è il numero di nodi del grafo [1].

2.2.2 Metodi di memorizzazione di un grafo

Un grafo *G(N,A)* può essere memorizzato all'interno di un calcolatore in molteplici modi; la scelta del modo è fortemente dipendente dalle caratteristiche che si vogliono evidenziare, dal tipo di utilizzo che si intende fare, dalle dimensioni del grafo da memorizzare (numero di nodi e di archi) e dalla sua densità (numero di archi rispetto al numero di nodi).

Occupiamoci di quattro modalità diverse di memorizzazione:

1. **Matrice di connessione o matrice di adiacenza**
2. **Matrice di incidenza**
3. **Edge List**
4. **Lista di adiacenza**

Di queste, la prima e l'ultima sono le più utilizzate; la prima è preferibile nel caso di grafi molto densi in quanto, in questi casi, rispetto all'ultima, permette una maggiore efficienza ed immediatezza nei calcoli, mentre l'ultima è preferibile nel caso di grafi sparsi, cioè nel caso di grafi in cui il numero di archi è piccolo rispetto al numero di nodi.

La matrice di incidenza, seppur meno efficace degli altri due metodi da un punto di vista computazionale, è preferibile in alcuni problemi di Ricerca Operativa.

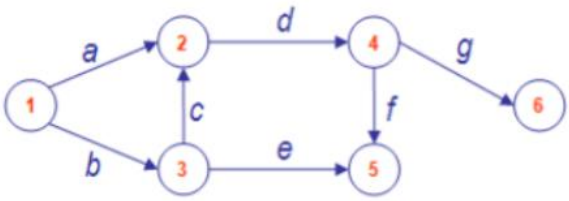
MATRICE DI CONNESSIONE (ADIACENZA)

La **matrice di connessione (adiacenza)** come metodo di memorizzazione si basa sull'utilizzo di una **matrice quadrata *nxn*** (con *n* numero dei nodi del grafo).

Nel caso in cui vogliamo memorizzare un **grafo non pesato** il generico elemento *(i,j)* della matrice di connessione vale 1 se esiste un arco che connette il nodo *i* e il nodo *j*, 0 altrimenti.

Nel caso in cui vogliamo memorizzare un **grafo pesato** il generico elemento *(i,j)* della matrice di connessione assume un valore coincidente con il peso associato all'arco nel caso in cui esiste un arco che connette il nodo *i* e il nodo *j*, 0 altrimenti.

La matrice di connessione è adatta sia per memorizzare grafi orientati sia per memorizzare grafi non orientati, nel caso di *grafi non orientati la matrice sarà simmetrica* (l'elemento *(i,j)* è uguale all'elemento *(j,i)*).



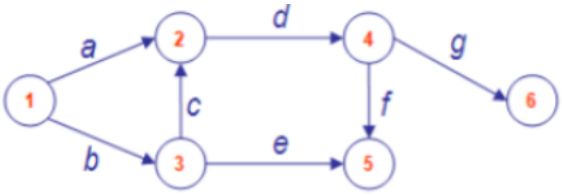
n/n	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	0	1	0	0
3	0	1	0	0	1	0
4	0	0	0	0	1	1
5	0	0	0	0	0	0
6	0	0	0	0	0	0

Figura 2.2.2.1: Esempio di matrice di connessione (adiacenza) nel caso di grafo orientato.

MATRICE DI INCIDENZA

La **matrice di incidenza** si basa sulla memorizzazione di una **matrice *nxm*** (*n* numero di nodi e *m* numero di archi del grafo). In tale matrice, nel caso in cui viene memorizzato un **grafo non pesato** e **non orientato**, il generico elemento *(i,j)* (supponiamo di aver numerato ogni singolo arco *j*) vale 1 se l'arco *j* incide sul nodo *i*, 0 altrimenti.

In tale matrice, nel caso in cui viene memorizzato un **grafo non pesato** e **orientato**, il generico elemento *(i,j)* (supponiamo di avere numerato ogni singolo arco *j*) vale 1 se l'arco *j* esce dal nodo *i*, -1 se l'arco *j* entra nel nodo *i* e 0 se l'arco *j* non incide sul nodo *i*, cioè se il nodo *i* non è un estremo dell'arco *j*.



n/m	a	b	c	d	e	f	g
1	1	1	0	0	0	0	0
2	-1	0	-1	1	0	0	0
3	0	-1	1	0	1	0	0
4	0	0	0	-1	0	1	1
5	0	0	0	0	-1	-1	0
6	0	0	0	0	0	0	-1

Figura 2.2.2.2: Esempio di matrice di incidenza nel caso di grafo orientato.

LISTA DI ADIACENZA

La **lista di adiacenza** è basata sulla memorizzazione di una lista che contiene, per ogni nodo del grafo, l'elenco dei relativi nodi adiacenti. Tale lista richiede un **massimo di (*n*+*m*) elementi** (*n* è il numero di nodi del grafo e *m* è il numero di archi del grafo).

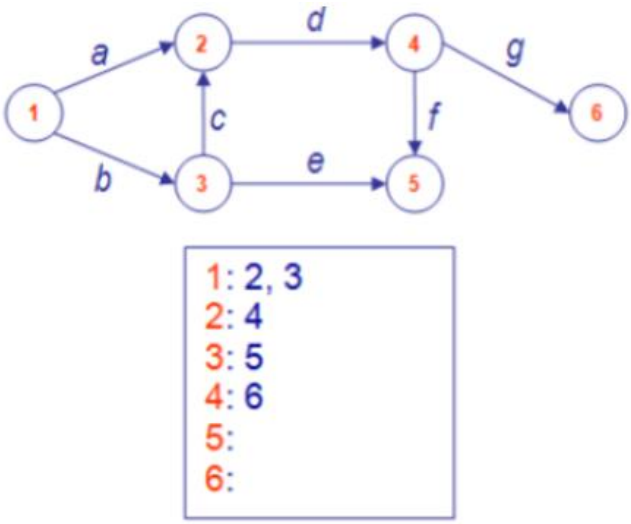


Figura 2.2.2.3: Esempio di lista di adiacenza nel caso di grafo orientato.

EDGE LIST

L'Edge List si basa sull'utilizzo di una lista contenente tutti gli archi che fanno parte del grafo.

Nel caso in cui l'Edge List viene utilizzata per memorizzare un **grafo non pesato** ogni singolo elemento di questa lista contiene una coppia di nodi (nodo di partenza, nodo di arrivo) interconnessa da ogni singolo arco del grafo.

Nel caso in cui l'Edge List viene utilizzata per memorizzare un **grafo pesato** ogni singolo elemento di questa lista contiene una coppia di nodi (nodo di partenza, nodo di arrivo) interconnessa da ogni singolo arco del grafo e il peso associato ad ogni singolo arco del grafo.

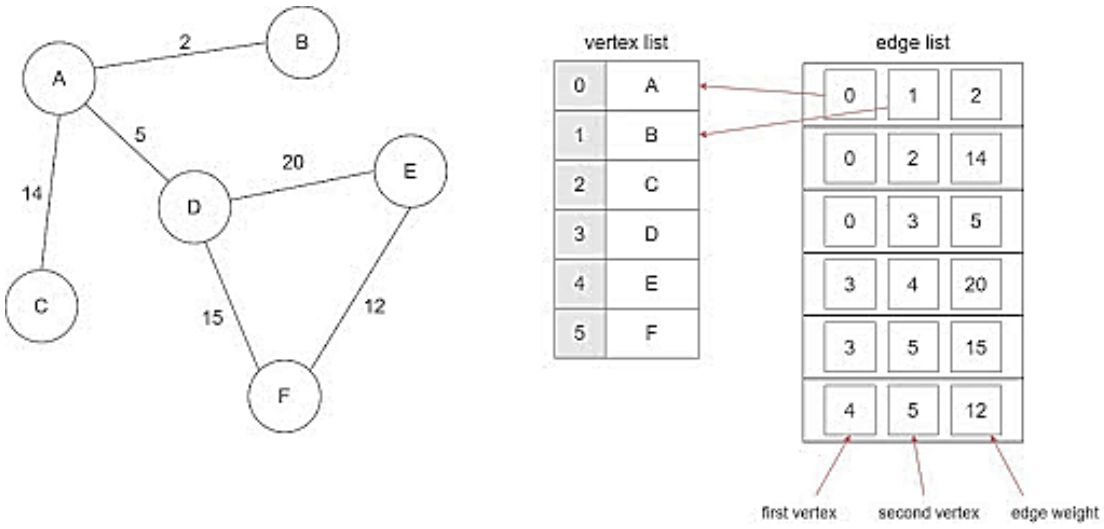


Figura 2.2.2.4: Esempio di Edge List nel caso di grafo non orientato.

2.2.3 Applicazioni Teoria dei grafi

La teoria dei grafi è estremamente importante in quanto può essere applicata in diversi campi. I grafi orientati vengono spesso impiegati per rappresentare le macchine a stati finiti e molti altri formalismi come ad esempio diagrammi di flusso, catene di Markov, ecc.

Ad esempio, nel mondo delle reti, la teoria dei grafi viene associata alla teoria dei **sei gradi di separazione**. Negli anni ’60 del secolo scorso lo psicologo sperimentale Stanley Milgram mise insieme alcuni studi sul fenomeno detto small world nelle reti sociali dell’uomo. Grazie al suo esperimento, Milgram, utilizzando la teoria dei grafi, cercò di stimare la distanza tipica tra due qualunque nodi di una rete sociale come, ad esempio, quella degli attori in maniera tale da dimostrare che, in generale, queste distanze dovevano essere piccole. L’obiettivo di Milgram era quello di provare che la globalizzazione ha il vantaggio di rendere il mondo più piccolo accorciando le distanze.

Un altro problema che viene spesso analizzato utilizzando le tecniche della teoria dei grafi è il problema dei **flussi sui grafi**. In tal caso si ha a disposizione un grafo (rete) e l’obiettivo è quello di studiare i flussi che viaggiano da un nodo all’altro di tale grafo.

Se, ad esempio, consideriamo un grafo rappresentante un sistema stradale, un flusso che può essere studiato è il movimento dei veicoli da un punto all’altro del sistema stradale.

Se consideriamo un grafo rappresentante una rete di computer, un flusso che può essere studiato è il movimento dei pacchetti da un computer all’altro della rete [6].

2.3 Risoluzione dei problemi di tipo NP-Hard sui grafi

Al fine di risolvere una serie di problemi del mondo reale si utilizzano un certo numero di algoritmi che, solitamente, vengono applicati ai grafi (reti) in grado di modellare i fenomeni del mondo reale. Sfortunatamente, la maggior parte di questi problemi, dal momento che per la loro risoluzione richiedono dei tempi di computazione molto elevati, sono di tipo **NP-Hard**. La conseguenza di ciò è che, spesso, o gli algoritmi che vengono utilizzati per risolvere tali problemi riescono a trovare una soluzione ottima dopo un tempo estremamente elevato, o riescono a trovare una soluzione subottima o, addirittura, non riescono nemmeno a trovare una soluzione in quanto tentano di risolvere un problema computazionalmente non fattibile.

Dal momento che la maggior parte degli algoritmi che tentano di risolvere i problemi del mondo reale richiedono dei tempi di risoluzione troppo elevati esiste una strategia per ridurre i tempi di computazione di tali problemi in maniera tale che, dopo aver attuato tale strategia di riduzione, gli algoritmi li possano risolvere in tempi ragionevoli?
Come detto precedentemente, una strategia per rispondere positivamente alla domanda appena posta esiste e consiste nell'utilizzare il Geometric Deep Learning.
Oggigiorno esistono diversi problemi di tipo NP-Hard riguardanti i grafi che è possibile tentare di risolvere, molti dei quali trovano applicazione anche nel mondo reale.
Ad esempio, il problema del **Commesso Viaggiatore** trova applicazione nella logistica, pianificazione, ecc., il problema di **determinazione delle clique di dimensione massima di un grafo** trova applicazione nell'ambito della rilevazione delle cosiddette bot-nets e nella Computer Vision.
Come detto, il tema centrale del presente lavoro di tesi è utilizzare il Geometric Deep Learning al fine di accelerare la risoluzione di un problema di tipo NP-Hard: **determinazione delle clique di dimensione massima** di un grafo.

2.3.1 Maximum Clique Enumeration Problem

Uno dei problemi di tipo NP-Hard che, per la sua risoluzione, richiede dei tempi di computazione estremamente elevati soprattutto quando i grafi sono molto grandi è quello della **determinazione delle clique di dimensione massima di un grafo**.
Una caratteristica del problema appena citato è che trova applicazione all'interno del mondo reale in differenti ambiti che includono: teoria del codice, biologia computazionale e chimica, acquisizione di informazioni, teoria di trasmissione dei segnali, ecc.
Ad esempio, Malod-Dognin, Andonov e Yanev risolvendo il problema sopracitato sono riusciti con successo a scovare una serie di caratteristiche comuni presenti all'interno di strutture di proteine differenti [7].
Riuscire a risolvere il problema sopracitato significa che, dato un grafo, è necessario trovare tutte le clique di dimensione massima presenti all'interno di esso. A questo punto, però, la domanda sorge spontanea, *come può essere definita una clique?*

In un grafo non orientato *G(N,A)* una **clique** può essere definita come un subset $S \subseteq N$ di nodi caratterizzato dal fatto che presa ogni singola coppia di nodi contenuta in questo subset *S*, essi sono adiacenti. In altre parole, dato un grafo *G*, una **clique** di tale grafo è un sottografo costituito da un certo numero di nodi e archi, dove ogni singolo nodo di tale sottografo è connesso tramite un arco con tutti gli altri nodi dello stesso sottografo.
In un grafo non orientato *G(N,A)* si definisce **clique di dimensione massima** la clique avente la dimensione più grande possibile.
Dato il grafo mostrato nella figura sottostante l'unica clique di dimensione massima è quella costituita dai nodi *{1,2,5}*.

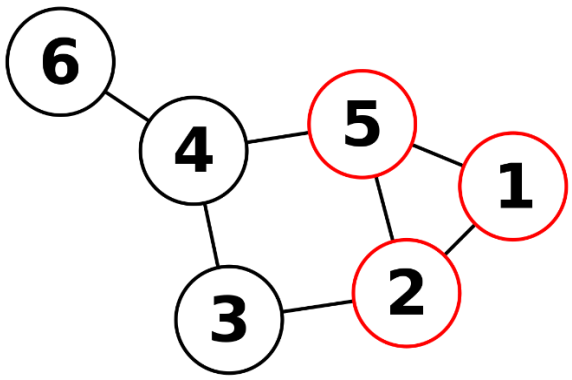


Figura 2.3.1.1: Esempio di clique di dimensione massima.

2.3.2 Solvers for Maximum Clique Enumeration Problem

All'interno di questo paragrafo verrà mostrata una carrellata di algoritmi utilizzati per risolvere il problema della determinazione delle clique di dimensione massima di un grafo.
Attualmente, gli algoritmi che consentono di fare ciò possono essere suddivisi in tre categorie:

- **Solver esatti**: essi sono in grado di trovare una **soluzione ottima** in un tempo che, nel caso peggiore, è esponenziale. Essi solitamente adottano un approccio di tipo *branch and bound*. Tale approccio, innanzitutto, consiste nel prendere l'intero spazio di ricerca e suddividerlo, in maniera iterativa, in un certo numero di sottospazi denominati **branches**. Man mano che viene attuata tale suddivisione, iterazione dopo iterazione, viene attuata un'operazione di **pruning** mediante la quale i branches che non sono utili ai fini della soluzione finale vengono scartati e dunque non vengono più presi in considerazione durante la fase di **branching**. L'operazione tramite la quale viene deciso se un certo branch deve o non deve essere scartato si chiama **bounding**.

- **Solver euristici**: essi, se confrontati con i solver esatti, sono estremamente più veloci in quanto, mentre i solver esatti richiedono tempi esponenziali per determinare una soluzione, quelli euristici richiedono tempi polinomiali per trovare una soluzione. Purtroppo, è vero che i solver euristici impiegano un tempo polinomiale per trovare una soluzione ma, tale soluzione, in generale, non è ottima ma **subottima**. Ciò significa che un solver euristico non è detto che riesca a trovare tutte le clique di dimensione massima di un grafo così come non è nemmeno detto che la dimensione di ogni singola clique trovata dall'algoritmo sia quella reale.
- **Solver domain or application specific**: sono dei solver che, a differenza di quelli precedenti, non sono adatti per un qualsiasi dominio ma possono essere utilizzati solo ed esclusivamente per determinare le clique di dimensione massima di un grafo appartenente ad un **dominio specifico**. La ragione di ciò è da ricercare nel fatto che, essendo questi solver domain specific, riescono a sfruttare una serie di proprietà dei grafi tipici di quel determinato dominio in maniera tale da trovare una soluzione in un tempo estremamente minore rispetto a quello che sarebbe richiesto se venissero utilizzati i solver citati sopra.

In aggiunta a quanto affermato finora possiamo evidenziare alcuni esempi di algoritmi (solver) che effettivamente vengono utilizzati nella pratica.

ALGORITMO BRON-KERBOSCH

In Network Science l'algoritmo Bron-Kerbosch [8], pubblicato nel 1973, viene utilizzato per determinare tutte le clique di dimensione massima di un grafo non orientato.

Esso è un algoritmo di tipo *recursive backtracking* che, nel caso peggiore, è caratterizzato da una complessità computazionale pari a $O(3^{(|V|/3)})$, dove |V| è il numero di nodi del grafo.

Benché ai fini della risoluzione del problema di determinazione delle clique di dimensione massima di un grafo, teoricamente, esistono anche altri algoritmi in grado di offrire prestazioni migliori, all'atto pratico l'algoritmo Bron-Kerbosch viene utilizzato moltissimo in quanto, rispetto alle alternative, si è dimostrato più efficiente.

Sempre nel 1973, in aggiunta alla versione base, è stata pubblicata un'altra versione di questo algoritmo che, rispetto alla versione base, prevede l'utilizzo di un *pivot*.

Nel corso degli anni sono state rilasciate diverse versioni di algoritmi che, in qualche modo, rappresentano un'evoluzione dell'algoritmo base. Ad esempio, la libreria **igraph** [9] (utilizzata nel presente lavoro di tesi) ha implementato un algoritmo in grado di determinare le clique di dimensione massima di un grafo che, di fatto, è una variante dell'algoritmo Bron-Kerbosch.

Tale variante è caratterizzata da una complessità computazionale pari a $O(d * |V| * 3^{(d/3)})$ dove |V| rappresenta il numero di nodi del grafo e *d* rappresenta la cosiddetta degenerazione del grafo [10].

FAST MAX-CLIQUE FINDER

Fast Max-Clique Finder implementa sia un solver esatto sia un solver euristico definiti all'interno di [11], che, secondo quanto affermato dagli autori, è adatto in tutte quelle situazioni in cui dobbiamo determinare le clique di dimensione massima di un grafo estremamente grande.

Per quanto concerne il solver esatto, possiamo affermare che esso adotta una strategia nota con il nome di *branch and bound* e, in particolare, implementa diverse strategie di *pruning* tra le quali ne possiamo citare una veramente innovativa che consiste nell'eliminare quel sottospazio di ricerca costituito da tutti i nodi che hanno un grado inferiore rispetto alla dimensione della clique di dimensione massima trovata dall'algoritmo. Ovviamente, la complessità computazionale di tale solver è esponenziale.

Per quanto concerne il solver euristico, possiamo affermare che esso, iterazione dopo iterazione, prende in considerazione solo ed esclusivamente quei nodi che con un'elevata probabilità possono appartenere a qualche clique di dimensione massima, ossia quei nodi che sono caratterizzati dall'avere il grado massimo. La complessità computazionale di tale solver è pari a $O(|V| * \Delta^2)$ dove Δ è il grado massimo all'interno del grafo.

CLIQUER

Cliquer [12] è un'implementazione molto popolare dell'algoritmo definito da Östergård nel 2002 [13], il quale adotta la strategia del *branch and bound*. Dal momento che le performance di tale algoritmo dipendono dall'ordinamento dei nodi, sempre nel 2002 Östergård ha definito anche un solver euristico che adotta una strategia denominata **vertex colouring**.

EmMCE

EmMCE (External-Memory algorithm for Maximal Clique Enumeration) [14] è un algoritmo che utilizza la memoria esterna (memoria di massa) per memorizzare quelle reti (grafi) che non possono essere allocate all'interno della memoria RAM a causa delle dimensioni eccessive. La caratteristica degna di nota di questo algoritmo è che esso riesce a risolvere il problema di determinazione delle clique di dimensione massima di un grafo utilizzando la computazione parallela.

Capitolo 3

Verso il Geometric Deep Learning

3.1 Machine Learning

Il Machine Learning è un approccio di Intelligenza Artificiale mediante il quale i computer riescono ad apprendere dai dati. In particolare, esso ingloba una serie di metodi volti ad estrapolare dei pattern insiti nei dati al fine di utilizzarli per diversi task, come ad esempio la predizione di dati futuri, ecc. [15].

Il Machine Learning prevede che i dati vengano organizzati in un set di **osservazioni (esempi)**, ognuna delle quali è caratterizzata da un certo numero di attributi denominati **features**. L'insieme costituito da tutte le osservazioni prende il nome di **dataset**, il quale, solitamente, viene rappresentato tramite una formulazione matriciale, dove ogni singola riga di questa matrice è un'**osservazione** e ogni singola colonna di tale matrice è una **feature**.

Nella parte iniziale di questo paragrafo abbiamo detto che il Machine Learning è una tecnica tramite la quale i computer riescono ad apprendere dai dati ma, effettivamente, *quando si dice che un computer apprende dai dati?*

T. Mitchell [16] afferma che un computer riesce ad apprendere dai dati se le sue performance migliorano con l'esperienza data una classe di tasks e una metrica di misurazione delle performance.

3.4.1 Tasks

Solitamente il Machine Learning viene utilizzato per attuare i seguenti tasks [17]:

- **Classificazione**: si parla di classificazione ogni qualvolta abbiamo a che fare con differenti gruppi di dati, ciascuno dei quali appartenente ad una classe differente e quando dato un input vogliamo sapere a quale delle N classi questo dato in input appartiene. Nel caso della classificazione il codominio della funzione che vogliamo approssimare è costituito da un numero discreto e finito di valori. Questi possibili valori rappresentano tutte le possibili categorie a cui può appartenere il dato che viene fornito in input. Un esempio di classificatore è quello che permette di distinguere se un'immagine contiene un gatto o un cane.

In tal caso, la classificazione consiste nel dare un'immagine in input al classificatore in maniera tale che esso ci possa dire se l'immagine fornitale contiene un gatto o un cane.

- **Regressione**: in tal caso, a differenza della classificazione, abbiamo un unico gruppo di dati e quello che vogliamo fare è approssimare la distribuzione dei dati utilizzando una funzione adeguata per lo scopo che si vuole raggiungere. Abbiamo bisogno di approssimare la distribuzione dei dati e dunque di utilizzare la regressione tutte le volte in cui abbiamo bisogno di effettuare delle predizioni. Nel caso della regressione il codominio della funzione che approssima la distribuzione dei dati è costituito da un numero infinito di valori, da tutti i numeri reali.
- **Rilevazione di anomalie**

3.1.2 Strategie di apprendimento

Nell'ambito del Machine Learning sono 3 le tipologie di apprendimento attraverso le quali un modello può imparare dai dati:

- **Apprendimento supervisionato**: in questo caso il modello di Machine Learning viene allenato utilizzando un certo numero di coppie: **osservazione** x_i e **target** y_i.
  Nel caso di apprendimento supervisionato, il task che vogliamo attuare è il seguente:
  *Dato un **training set** di N coppie osservazione-target* $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ *dove ogni* y_i *è stato generato da una funzione sconosciuta* $y = f(x)$*, trovare una funzione h che approssima la vera funzione f.*
  Il vantaggio di questa tipologia di apprendimento (che abbiamo utilizzato nel presente lavoro di tesi) è che essa permette il monitoraggio della qualità del training e, soprattutto, è molto comune e studiata.
  Lo svantaggio di questa tipologia di apprendimento è che, purtroppo, si può verificare l'**overfitting**. Si verifica tale fenomeno tutte le volte in cui il modello ha imparato dal training set talmente bene che ogni qualvolta gli forniamo i dati che ha già visto, ossia gli forniamo dei dati appartenenti al training set fornisce delle risposte sensate, ma talmente male che ogni qualvolta gli forniamo dei dati che non ha mai visto fornisce delle risposte totalmente sbagliate, delle uscite totalmente insensate.

Un altro problema che abbiamo con l'apprendimento supervisionato è che, purtroppo, affinché sia possibile applicarlo dobbiamo necessariamente conoscere anche i targets, i vari y_i e questo non è sempre possibile.

A volte collezionare i targets y_i può essere molto costoso, molto dispendioso. In tutti quei casi in cui non riusciamo a collezionare i targets y_i purtroppo non possiamo utilizzare questa tipologia di apprendimento.

- **Apprendimento non supervisionato**: in questo caso il modello di Machine Learning viene allenato utilizzando solo ed esclusivamente le osservazioni in ingresso x_i.

  Il vantaggio di questa tipologia di apprendimento è che essa consente di scoprire delle dinamiche nascoste che si celano dietro i dati e che sono sconosciute persino agli esperti. Un altro grande vantaggio dell'apprendimento non supervisionato è che, in questo caso, per poter utilizzare questa tipologia di apprendimento, non è necessario conoscere i targets y_i.

  Lo svantaggio di questa tipologia di apprendimento è che, a causa del fatto che non conosciamo i targets y_i, è estremamente complesso riuscire a monitorare la qualità del training.

- **Apprendimento per rinforzo**: esso è totalmente differente rispetto alle tipologie di apprendimento viste precedentemente in quanto, a differenza di quanto accadeva nelle altre due tipologie, in questo caso non abbiamo alcun **dataset**.

  L'apprendimento per rinforzo funziona così: la macchina riceve in input un **obiettivo da raggiungere**. Inizialmente, la macchina conosce l'obiettivo ma non sa assolutamente cosa deve fare per raggiungerlo in quanto rispetto alle tipologie di apprendimento viste prima in questo caso non abbiamo alcun **dataset**. Dunque, l'apprendimento per rinforzo è una tipologia di apprendimento in cui la macchina riesce ad imparare soltanto grazie all'esperienza, soltanto grazie ad una serie di tentativi andati male e grazie ad una serie di tentativi andati bene. In questo caso, **è con l'esperienza, è commettendo errori che la macchina riesce ad imparare e a crearsi una conoscenza**.

  L'idea che sta alla base dell'apprendimento per rinforzo è quella di assegnare alla macchina una ricompensa ogni qualvolta essa si comporta in un modo corretto tale da avvicinarla all'obiettivo da raggiungere che gli è stato fornito in ingresso e una penalizzazione ogni qualvolta essa si comporta in un modo scorretto tale da allontanarla dall'obiettivo da raggiungere.

Tutto ciò viene fatto in maniera tale che la macchina, man mano che riceve delle ricompense e delle penalità, riesce ad apprendere quali sono i comportamenti virtuosi, ossia quei comportamenti che può tranquillamente tenere dal momento che ha ricevuto una ricompensa e quali sono i comportamenti non virtuosi, ossia quei comportamenti che non deve assolutamente tenere dal momento che ha ricevuto delle penalità. Questa è l'idea che sta alla base dell'apprendimento per rinforzo.

3.1.3 Metriche di valutazione delle Performance

Al fine di valutare correttamente le performance di un algoritmo di Machine Learning è necessario utilizzare delle metriche di valutazione delle performance. Solitamente quando si parla di performance si fa riferimento alle **performance di generalizzazione**, ossia a quanto l'algoritmo di Machine Learning riesce a generalizzare e dunque a funzionare correttamente anche quando gli vengono forniti dei dati che non ha mai visto prima d'ora.

Ovviamente, esistono tantissime metriche di valutazione delle performance che è possibile utilizzare. La scelta di quale metrica utilizzare dipende dal dominio in cui ci troviamo ad operare in quanto ci sono delle metriche che sono adatte per alcuni domini e altre metriche che, invece, sono adatte per altri domini [18].

Ad esempio, nel caso in cui il task che vogliamo attuare è la **classificazione** come metrica di valutazione delle performance è possibile utilizzare o l'**accuratezza** oppure la **matrice di confusione**, la quale contiene un certo numero di elementi, dove ogni singolo elemento a_{ij} contiene il numero di osservazioni appartenenti alla classe reale (target) j che il modello (classificatore) ha classificato come appartenenti alla classe i.

3.1.4 Training, Validation e Testing

Nell'ambito del Machine Learning un obiettivo fondamentale da perseguire è fare in modo che i modelli riescano ad apprendere dai dati.

È degno di nota affermare che la fase durante la quale un modello di Machine Learning apprende dai dati prende il nome di **fase di training**. Durante questa fase, iterazione dopo iterazione, viene attuata un'operazione di **ottimizzazione** mediante la quale, al fine di migliorare l'apprendimento del modello, vengono aggiornati i **parametri (pesi)** *w* del modello in maniera tale da minimizzare una funzione denominata **loss function**.

Occorre sottolineare che la **loss function** è totalmente differente rispetto all'**accuratezza** in quanto, mentre la loss function viene utilizzata ai fini dell'ottimizzazione, l'accuratezza viene utilizzata al fine di rendere comprensibile ad un essere umano come il modello sta migliorando il suo apprendimento. Le **loss function** più utilizzate sono:

- **Mean Squared Error (MSE)**: viene utilizzata ogni qualvolta il task da attuare è la **regressione**. Il MSE viene definito così:

$$L\,(w) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

  dove N è il numero di osservazioni presenti all'interno del dataset, y_i è il valore predetto dal modello e $\hat{y}_i$ è il valore corretto dell'osservazione *i-esima* presente all'interno del dataset.

- **Negative Log-Likelihood (NLL)**: viene utilizzato ogni qualvolta il task da attuare è la **classificazione**. Il NLL viene definito così:

$$L\,(w) = -\sum_{i=1}^{N}\log(P(y_i|\,x_i))$$

  dove N è il numero di osservazioni presenti all'interno del dataset e $P(y_i|\,x_i)$ rappresenta la probabilità che data un'osservazione x_i il modello la predica come appartenente alla classe reale di appartenenza y_i.

Dal momento che i modelli di Machine Learning sono complessi e il dataset, spesso, è estremamente grande sia il processo di **ottimizzazione** che prevede di calcolare il gradiente della Loss Function rispetto a tutti i parametri del modello al fine di ricavare tutti i parametri del modello tali per cui il gradiente della Loss Function si annulla (condizione che consente di minimizzare la Loss Function) sia la strategia di provare tutte le possibili combinazioni dei parametri del modello al fine di determinare le combinazioni ottimali che minimizzano la Loss Function **non sono computazionalmente fattibili**.

Dal momento che le due strategie sopra citate non possono essere utilizzate per ottimizzare i parametri del modello la strategia più comune che viene utilizzata è quella che prende il nome di **esplorazione iterativa**. Essa, iterazione dopo iterazione, prevede un aggiornamento **graduale** dei parametri del modello in maniera tale da ottenere il minimo della Loss Function con una buona approssimazione. Purtroppo, questa soluzione garantisce soltanto un minimo locale e non un minimo globale. Ciò significa che aggiornando **gradualmente** i parametri del modello noi possiamo giungere ad un minimo ma, questo non è detto che sia il minimo globale in quanto potrebbero esistere altri minimi che sono ancora più minimi rispetto a quello in cui siamo giunti. Ciò nonostante, questa soluzione viene adottata tantissimo in quanto se il modello che stiamo utilizzando è adatto per il task che vogliamo attuare c'è la speranza che dopo un certo numero di iterazioni il modello riesca ad apprendere e a **generalizzare** correttamente.

Supponendo di utilizzare tale soluzione ci chiediamo: *iterazione dopo iterazione come viene effettuato l'aggiornamento graduale dei parametri del modello?*

I parametri del modello vengono aggiornati utilizzando la regola della **discesa del gradiente**:

$$w(t+1) = w(t) + \Delta w$$

In pratica, il valore di w, ossia dei parametri del modello all'iterazione $t+1$ è pari al valore di w, ossia dei parametri del modello all'iterazione t più Δw.

A questo punto la domanda sorge spontanea, *il valore di* Δw *ad ogni singola iterazione come viene calcolato?* Esso viene calcolato in maniera diversa a seconda dell'algoritmo utilizzato tra i tre disponibili:

1. **Batch Gradient Descent**
2. **Stochastic Gradient Descent**
3. **Mini-batch Gradient Descent**

BATCH GRADIENT DESCENT

In tal caso, il valore di Δw ad ogni singola iterazione viene calcolato applicando la seguente formula:

$$\Delta w = -\eta \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L}{\partial \mathrm{w}}(x_i, y_i)$$

dove η è il **learning rate**, un parametro fondamentale per impostare la velocità di apprendimento del modello, N è il numero di osservazioni appartenenti al dataset e $\frac{\partial L}{\partial \mathrm{w}}(x_i, y_i)$ è il gradiente della Loss Function rispetto a tutti i parametri del modello calcolato utilizzando una determinata coppia: osservazione x_i, target y_i.

In questo caso è opportuno sottolineare il fatto che, ad ogni singola iterazione, i parametri del modello vengono aggiornati una volta soltanto utilizzando l'intero dataset.

STOCHASTIC GRADIENT DESCENT (SGD)

In tal caso, all'interno di ogni singola iterazione, viene calcolato il valore di Δw tante volte quante sono le osservazioni presenti all'interno del dataset utilizzando la formula sottostante:

$$\Delta w = -\eta \frac{\partial L}{\partial \mathrm{w}}(x_i, y_i)$$

dove η è il **learning rate**, un parametro fondamentale per impostare la velocità di apprendimento del modello, *N* è il numero di osservazioni appartenenti al dataset e $\frac{\partial L}{\partial \mathrm{w}}(x_i, y_i)$ è il gradiente della Loss Function rispetto a tutti i parametri del modello calcolato utilizzando una determinata coppia: osservazione x_i, target y_i.

In questo caso è opportuno sottolineare il fatto che, ad ogni singola iterazione, dal momento che vengono calcolati un numero di Δw pari al numero di osservazioni presenti nel dataset i parametri del modello vengono aggiornati un numero di volte pari al numero di osservazioni appartenenti al dataset.

MINI-BATCH GRADIENT DESCENT

In tal caso, si prende il dataset costituito da un numero *N* di coppie osservazione x_i, target y_i e lo si splitta in un numero di mini-batch pari a *N/B*, ognuno dei quali contiene un numero *B* di coppie osservazione x_i, target y_i.

Dopo aver fatto ciò, all'interno di ogni singola iterazione, si prende ogni singolo minibatch e lo si usa per aggiornare tutti i parametri del modello utilizzando la formula seguente:

$$\Delta w = -\eta \frac{1}{B} \sum_{i=1}^{B} \frac{\partial L}{\partial \mathrm{w}}(x_i, y_i)$$

È degno di nota sottolineare che, a prescindere dall'algoritmo utilizzato, in ciascuna delle formule che abbiamo mostrato è presente un segno meno.

Il fatto che sia presente tale segno è di fondamentale importanza in quanto, come detto, ci consente di effettuare un aggiornamento dei parametri del modello muovendoci in una direzione che è opposta rispetto a quella di massimo incremento della Loss Function.

È opportuno che ciò accada in quanto, iterazione dopo iterazione, vogliamo che i parametri del modello vengano aggiornati in maniera tale da minimizzare la Loss Function in quanto più la Loss Function è vicina allo zero più il modello apprende bene dai dati.

Al fine di completare il discorso sull'aggiornamento dei parametri di un modello resta da capire come, iterazione dopo iterazione, viene effettuato il calcolo del **gradiente della Loss Function rispetto a tutti i parametri *w* del modello**.

Si può affermare che, iterazione dopo iterazione, il gradiente della Loss Function rispetto a tutti i parametri del modello, ossia il termine $\frac{\partial L}{\partial \mathrm{w}}$ viene calcolato utilizzando l'**algoritmo di backpropagation**.

Nella parte iniziale di questo paragrafo abbiamo affermato che la fase durante la quale un modello di Machine Learning apprende dai dati prende il nome di **fase di training**. A questo punto, però, la domanda sorge spontanea, *durante la fase di training al fine di allenare il modello viene utilizzato l'intero dataset oppure una sua parte?*

Prima di rispondere alla domanda facciamo una premessa. Un modello di Machine Learning viene allenato al fine di minimizzare il cosiddetto **errore di generalizzazione**. Ciò significa che i parametri del modello, iterazione dopo iterazione, vengono aggiornati in maniera tale che il modello diventi sempre più bravo a generalizzare e dunque a funzionare bene anche quando gli vengono fornite delle osservazioni che non ha mai visto prima d'ora.

Di conseguenza, se utilizzassimo l'intero dataset per effettuare il training non potremmo in alcun modo valutare le performance di generalizzazione del modello in quanto non avremmo a disposizione nemmeno un'osservazione che il modello non ha mai visto prima d'ora. Dunque, è evidente che di tutto il dataset che abbiamo a disposizione ne dobbiamo prendere soltanto una parte da utilizzare per il training, mentre la restante, dal momento che contiene delle osservazioni che il modello non ha mai visto prima d'ora, verrà utilizzata per valutare le performance di generalizzazione. Da quanto appena affermato si evince che prima ancora di iniziare la fase di training è opportuno splittare il dataset in due parti: **training set** e **testing set** ma, purtroppo, fare ciò non è sufficiente in quanto, come detto precedentemente, si può verificare l'**overfitting** in virtù del quale il modello funziona benissimo ogni qualvolta gli forniamo delle osservazioni che ha già visto e funziona malissimo ogni qualvolta gli forniamo delle osservazioni che non ha mai visto prima d'ora.

Al fine di rilevare l'**overfitting** e dunque stimare correttamente gli **iperparametri** del modello come, ad esempio, il **numero di epoche in cui arrestare il training** o il **learning rate** il dataset, in realtà, viene splittato in **training set**, **validation set** e **testing set**.

In sintesi, dopo aver costruito il modello di Machine Learning e prima ancora di iniziare la fase di training quello che dobbiamo fare è splittare il dataset in tre parti:

- **Training set**: esso verrà utilizzato durante la fase di **training** per stimare i **parametri** del modello al fine di minimizzare la Loss Function.
- **Validation set**: esso verrà utilizzato durante la fase di **validation** per stimare gli **iperparametri** ottimali del modello in maniera tale da evitare che si verifichi l'**overfitting**.
- **Testing set**: esso verrà utilizzato durante la fase di **testing** per valutare le performance di generalizzazione del modello. Durante la fase di testing al modello vengono fornite delle osservazioni che non ha mai visto prima d'ora in maniera tale da valutare se il modello riesce a generalizzare bene oppure no.

Per concludere il discorso sul Machine Learning possiamo affermare che, al fine di ridurre al minimo la probabilità che si verifichi l'**overfitting**, spesso, piuttosto che utilizzare le Loss Function viste nei paragrafi precedenti, si utilizza una **Loss Function regolarizzata** definita dalla seguente formula:

$$L_r\ (w) = L(w) + \ \lambda * R(w)$$

dove λ è il **termine di regolarizzazione** mediante il quale, a seconda del valore che assume, è possibile ridurre le capacità di apprendimento del modello in maniera tale che il processo di training del modello sia più complesso e dunque il modello non vada in overfitting.
Solitamente, dal momento che, come regolarizzazione, si utilizza quella di tipo $\mathbf{L_2}$ il termine *R(w)* può essere definito come segue:

$$R(w) = \frac{1}{2}\ ||w||_2^2$$

3.2 Deep Learning

Un metodo di Machine Learning molto promettente è quello che prende il nome di **Deep Learning**, il quale è una tipologia di **Representation Learning** che sta diventando molto popolare negli ultimi anni grazie alla capacità di garantire un livello di performance estremamente elevato per l'attuazione dei tasks più comuni.
Quando parliamo di **Representation Learning** ci riferiamo al problema di apprendere una rappresentazione dei dati che abbiamo a disposizione che è indipendente dallo specifico task che vogliamo attuare [17].
L'idea che sta alla base del Deep Learning è quella di apprendere delle rappresentazioni molto complesse utilizzando una gerarchia di livelli. Un aspetto importante da sottolineare è il fatto che più procediamo in profondità con i livelli e più ogni singolo livello, facendo uso delle features prodotte in uscita dal livello precedente, riesce ad apprendere delle features sempre più complesse.

3.2.1 Percettrone Multistrato

Il percettrone multistrato (MLP) è la **rete neurale** più generica che possiamo prendere in considerazione. **Una rete neurale è un modello computazionale basato su un grafo di neuroni artificiali interconnessi**.

Quando le connessioni sono acicliche, ovvero quando ciascun livello dipende solo dal livello precedente si parla di **rete feedforward**, la quale è conveniente da un punto di vista computazionale (in quanto per calcolare le uscite dei neuroni presenti in un certo Layer è necessario calcolare soltanto le uscite prodotte dai neuroni presenti nel Layer precedente).

Il **percettrone multistrato** è un esempio di rete neurale di tipo **feedforward**.

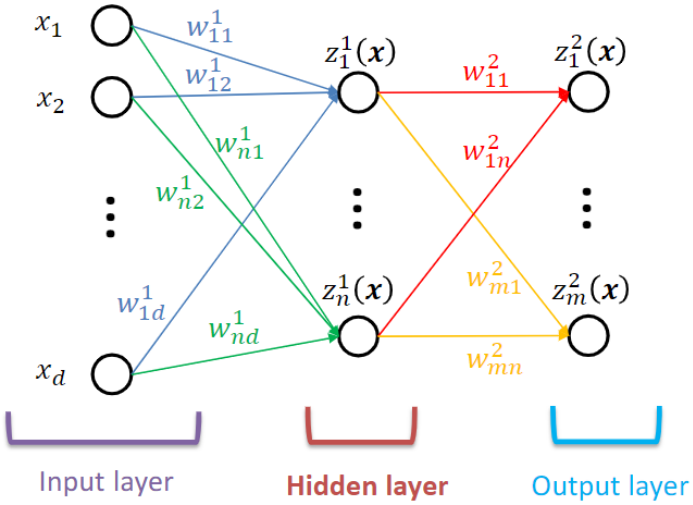


Figura 3.2.1.1: Esempio di rete neurale percettrone multistrato.

Un percettrone multistrato è caratterizzato da un **Input Layer** che è costituito dall'insieme delle features x_i di una determinata osservazione (nell'Input Layer non abbiamo neuroni), da un certo numero di **Hidden Layers**, in ciascuno dei quali abbiamo un certo numero di neuroni e da un **Output Layer** in cui abbiamo un certo numero di neuroni.

La cosa importante da sottolineare è che ogni singolo neurone presente in ogni singolo Hidden Layer è caratterizzato da una funzione di attivazione non lineare mentre ogni singolo neurone presente nell'Output Layer **non** è caratterizzato da nessuna funzione di attivazione.

Il percettrone multistrato viene utilizzato tantissimo in quanto è stato dimostrato che è in grado di approssimare **qualsiasi** funzione non lineare con precisione arbitraria se ha **almeno** un Hidden Layer (e abbastanza neuroni al suo interno). Per questo motivo il percettrone multistrato è detto **approssimatore universale**.

A questo punto vediamo il funzionamento di un percettrone multistrato supponendo che vi sia un solo Hidden Layer.

Per prima cosa viene calcolato il prodotto tra ogni singolo peso $w_{1,i}^1$ e il relativo ingresso x_i.

Questa operazione viene fatta per tutti gli ingressi. Dopodiché, viene calcolata la sommatoria tra tutti questi prodotti $w_{1,i}^1 x_i$ e il bias b_1^1 (è uno scalare). Il risultato di questa operazione viene mandato in ingresso ad una **funzione di attivazione** φ che solitamente è non lineare.

Il valore che viene generato in seguito all'applicazione della **funzione di attivazione** φ è proprio il valore $z_1^1(x)$ che viene prodotto in uscita dal primo neurone presente nell'Hidden Layer.

Dopodiché, viene calcolato il prodotto tra ogni singolo peso $w_{2,i}^{1}$ e il relativo ingresso x_i.
Questa operazione viene fatta per tutti gli ingressi. Dopodiché, viene calcolata la sommatoria tra tutti questi prodotti $w_{2,i}^{1}x_i$ e il bias b_2^1 (è uno scalare). Il risultato di questa operazione viene mandato in ingresso ad una **funzione di attivazione** φ che solitamente è non lineare.
Il valore che viene generato in seguito all'applicazione della **funzione di attivazione** φ è proprio il valore $z_2^1(x)$ che viene prodotto in uscita dal secondo neurone presente nell'Hidden Layer.
Tramite questo procedimento è possibile generare l'uscita di ogni singolo neurone presente nell'Hidden Layer.
Dopodiché, viene calcolato il prodotto tra ogni singolo peso $w_{1,i}^{2}$ e la relativa uscita $z_i^1(x)$.
Questa operazione viene fatta per tutte le uscite generate dai neuroni che si trovano nell'Hidden Layer. Dopodiché, viene calcolata la sommatoria tra tutti questi prodotti $w_{1,i}^{2}z_i^1(x)$ e il bias b_1^2 (è uno scalare). Il risultato di questa operazione è proprio il valore $z_1^2(x)$ che viene prodotto in uscita dal primo neurone presente nell'Output Layer e dunque è proprio la prima uscita della rete neurale.
Questo è il procedimento che deve essere applicato per determinare le diverse uscite dell'Output Layer e dunque della rete neurale.
Da quanto affermato finora è evidente che in un percettrone multistrato **l'uscita di ogni singolo neurone presente in ogni singolo Layer viene calcolata utilizzando le uscite prodotte dal Layer precedente**.
Per effettuare il training di una rete neurale come il percettrone multistrato solitamente viene adottata l'**esplorazione iterativa** descritta nei paragrafi precedenti.

3.2.2 Rete Neurale Convoluzionale

Una **rete neurale convoluzionale** è una rete neurale costituita da un certo numero di layer. Spesso, dal momento che le reti neurali convoluzionali vengono utilizzate per processare immagini, ogni singolo layer convoluzionale riceve in ingresso un'immagine denominata **input feature map**, attua una serie di operazioni di cui parleremo e, infine, in uscita, produce un'immagine chiamata **output feature map**.
All'interno di ogni singolo layer di una CNN vengono attuate tre operazioni:

1. **Convoluzione**
2. **Applicazione della funzione di attivazione**
3. **Pooling**

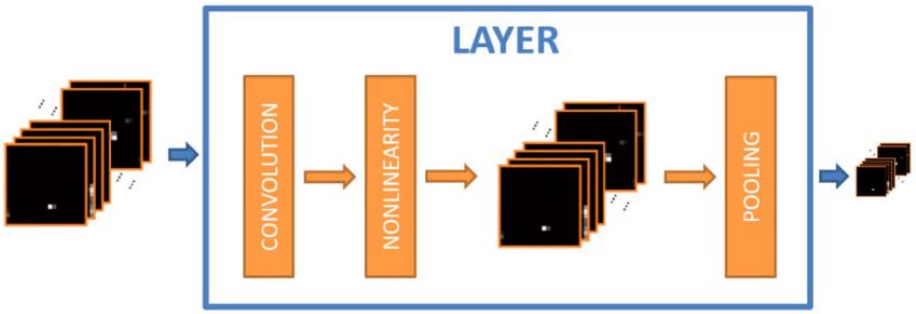


Figura 3.2.2.1: Struttura di un layer di una rete neurale convoluzionale.

La **convoluzione** è l'operazione fondamentale che viene attuata all'interno di un layer di una CNN, cerchiamo di capire come funziona.

Innanzitutto, occorre premettere che la **convoluzione** è un tipo di trasformazione locale che applica un filtro denominato **kernel**, il quale viene definito da una matrice.

Dopo aver premesso ciò, supponiamo di voler applicare la **convoluzione** ad un'immagine rappresentata in scala di grigi, ossia ad un'immagine in bianco e nero. In tal caso, a fronte dell'applicazione della convoluzione si ottiene un'immagine avente un certo numero di pixels dove ogni singolo pixel *i,j* (che è uno scalare) dell'immagine in uscita viene generato calcolando la combinazione lineare (somma dei prodotti) tra gli elementi del filtro (che, come detto, è una matrice) e i corrispondenti pixels (che sono scalari) dell'immagine in ingresso che si sovrappongono al filtro centrato proprio sul pixel *i,j* dell'immagine in ingresso.

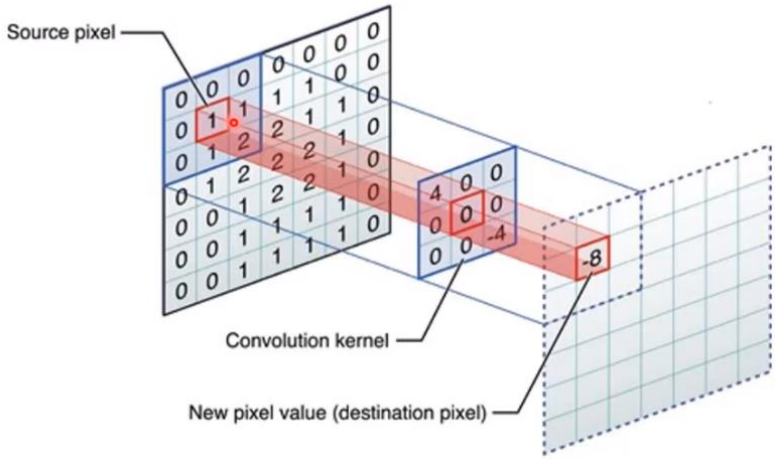


Figura 3.2.2.2: Esempio di applicazione dell'operazione di convoluzione.

Per quanto riguarda la **funzione di attivazione** possiamo affermare che solitamente in una rete neurale convoluzionale si utilizza la funzione di attivazione non lineare **ReLU** (**Rectfied Linear Unit**) che può essere rappresentata in tal modo:

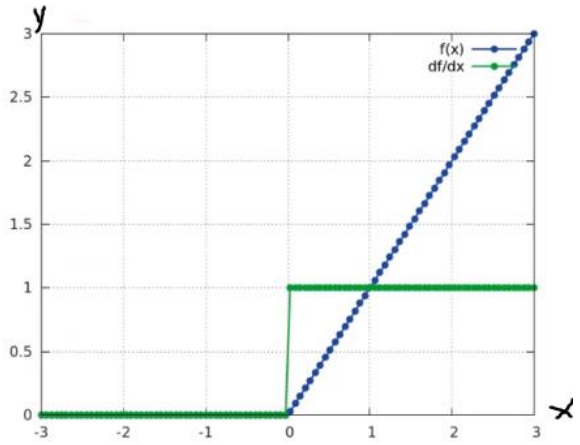


Figura 3.2.2.3: Funzione di attivazione ReLU.

Per concludere il discorso riguardante le operazioni che vengono attuate all'interno di un generico layer di una CNN dobbiamo parlare dell'operazione di **pooling**.

Innanzitutto, prima ancora di applicare tale operazione, la prima cosa che occorre fare è stabilire il valore dell'iperparametro **pool size**. Dal momento che stiamo considerando CNN in grado di processare immagini, specificare l'iperparametro pool size significa specificare quante righe e quante colonne deve avere la **finestrella** da utilizzare per attuare il pooling.

Una volta stabilito il valore di tale iperparametro, data un'immagine in ingresso avente un certo numero di canali, per generare ogni singolo canale *i-esimo* dell'immagine di uscita si procede così: man mano la finestrella la cui dimensione è stata precedentemente stabilita viene fatta scorrere sul canale *i-esimo* dell'immagine in ingresso e, di volta in volta, viene calcolato il valore **massimo** (in quanto stiamo supponendo di attuare un'operazione di **max pooling**) tra i pixels del canale *i-esimo* dell'immagine in ingresso che si sovrappongono a tale finestrella.

Nella figura sottostante, a sinistra viene mostrato un canale di un'immagine in ingresso a cui vogliamo applicare l'operazione di **max pooling** e a destra viene mostrato il canale dell'immagine in uscita ottenuto a seguito dell'applicazione di tale operazione:

Figura 3.2.2.4: Esempio di applicazione dell'operazione di max pooling.

L’operazione di **pooling** non è obbligatoria. Ciò significa che è possibile utilizzare dei layer che fanno uso semplicemente della convoluzione e della funzione di attivazione al fine di produrre un’uscita. Ciò nonostante, in generale, nelle CNN oltre a fare uso della convoluzione e della funzione di attivazione si fa uso anche dell’operazione di pooling in quanto mediante essa è possibile fare in modo che un layer, in uscita, produca un’immagine che è più piccola rispetto all’immagine che gli è stata fornita in ingresso.

In generale, è conveniente attuare un’operazione di **pooling** e dunque ridurre la dimensione dell’immagine che viene fornita in ingresso ad un layer di una CNN per due motivi:

- Man mano che ci spostiamo da un layer a quello successivo quello che accade è che i calcoli si semplificano in quanto, via via, si lavora con immagini sempre più piccole.
- Man mano che andiamo sempre più in profondità con i livelli quello che accade è che ogni singolo neurone non rappresenta semplicemente un pixel ma rappresenta un insieme di pixels e dunque qualcosa di più complesso. Per questo motivo, nei livelli più profondi non abbiamo bisogno di lavorare con immagini aventi risoluzione molto elevata e dunque possiamo tranquillamente ridurre la dimensione.

Per concludere il discorso sulle reti neurali convoluzionali si può affermare che nel caso in cui vogliamo utilizzare una rete neurale convoluzionale per attuare la **classificazione** la rete che deve essere utilizzata è quella che prevede un certo numero di layer convoluzionali e di fully-connected layer proprio come mostrato nella figura sottostante:

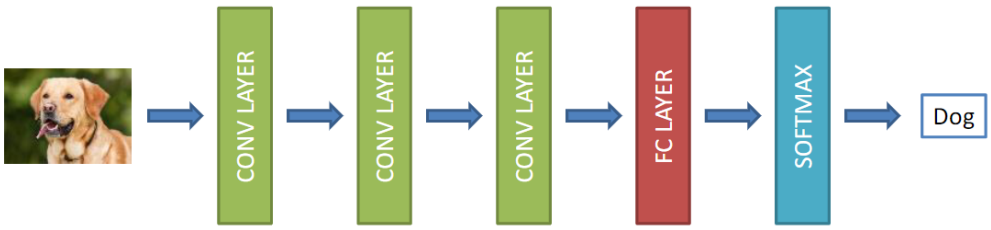


Figura 3.2.2.5: Struttura di una rete neurale convoluzionale.

3.2.3 Rete Neurale Ricorrente

Le reti neurali ricorrenti vengono utilizzate per processare dati sequenziali come:

- Video: un video può essere definito come una sequenza di immagini chiamate **frames**.
- Testo: un testo può essere definito come una sequenza di caratteri oppure come una sequenza di parole.
- Audio.

Nel caso di una RNN alla rete neurale non viene fornita l'intera sequenza tutta in una volta, ma la sequenza le viene fornita man mano con il passare del tempo. Ciò significa che, data una sequenza, in un certo istante di tempo chiamato **step temporale** verrà fornito un certo campione di tale sequenza, nello **step temporale** successivo verrà fornito il campione successivo di tale sequenza, nello **step temporale** ancora successivo verrà fornito il campione ancora successivo di tale sequenza e così via.

Grazie al fatto che i campioni di una certa sequenza non le vengono passati tutti in una volta, ma le vengono passati man mano, **step temporale** dopo **step temporale**, possiamo affermare che la RNN è estremamente vantaggiosa per processare dati sequenziali in quanto riesce a mantenere la **dinamica temporale** insita nella sequenza che, man mano, **step temporale** dopo **step temporale**, campione dopo campione, le viene passata.

TOPOLOGIA DI UNA RNN

A differenza delle reti neurali di tipo feed-forward, una **rete neurale ricorrente (RNN)** è una rete neurale in cui esistono **cicli**.

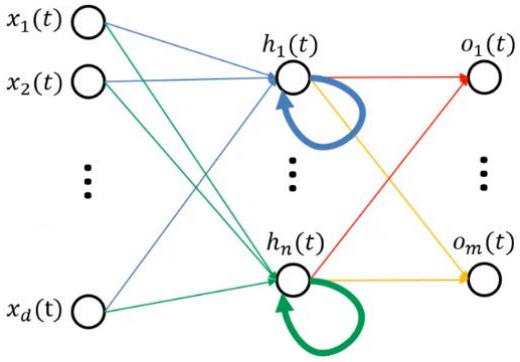


Figura 3.2.3.1: Struttura di una rete neurale ricorrente.

Per comprendere cosa significa commentiamo la figura sovrastante.

Innanzitutto, nel caso più semplice possibile una rete neurale ricorrente è costituita da un Input Layer, da un Recurrent Layer e da un Output Layer.

L'Input Layer contiene le **features del campione** del dato sequenziale che allo **step temporale *t*** viene mandato in ingresso alla rete.

Il Recurrent Layer contiene un certo numero di neuroni, **dove ogni singolo neurone per generare l'uscita all'istante *t* prende in considerazione le features che gli arrivano dall'Input Layer e che sono relative al campione fornito in ingresso alla rete all'istante *t* e prende in considerazione anche le uscite prodotte dai neuroni appartenenti allo stesso Recurrent Layer all'istante *t-1***.

In sintesi, ogni singolo neurone *i-esimo* presente all'interno del Recurrent Layer per generare l'uscita $h_i(t)$ prende in considerazione le features $x_1(t)$, $x_2(t)$, …, $x_d(t)$ presenti all'interno dell'Input Layer e le uscite $h_1(t-1)$, $h_2(t-1)$, …, $h_n(t-1)$ generate allo step temporale precedente *t-1* dai neuroni appartenenti allo stesso Recurrent Layer.

Grazie a questa retroazione ogni singolo neurone appartenente al Recurrent Layer prenderà una decisione e dunque produrrà un'uscita che non è condizionata soltanto dalle uscite che vengono prodotte dal layer precedente ma che è condizionata anche dalla storia passata, ossia dalle uscite prodotte allo step temporale precedente dai neuroni appartenenti allo stesso Recurrent Layer.

L'Output Layer contiene tanti neuroni quante sono le uscite che la rete deve produrre. Ogni singolo neurone *i-esimo* di tale layer riceve le uscite prodotte dal Recurrent Layer e genera l'uscita *i-esima* della rete neurale.

Un modo alternativo per rappresentare una RNN è il seguente:

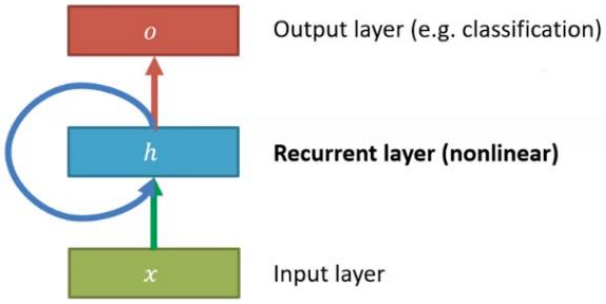


Figura 3.2.3.2: Primo modo alternativo di rappresentare una RNN.

In pratica il Recurrent Layer in ingresso riceve il vettore *x(t)*, ossia il vettore contenente tutte le features del campione che viene mandato in ingresso alla rete all'istante *t* e riceve anche l'uscita *h(t-1)* prodotta dallo stesso Recurrent Layer allo step temporale precedente *t-1* e in uscita produce *h(t)*. Tale uscita viene mandata in ingresso all'Output Layer, il quale, in uscita, produce *o(t)*.

La rappresentazione appena discussa coincide con un'altra rappresentazione chiamata **unrolled visualization**:

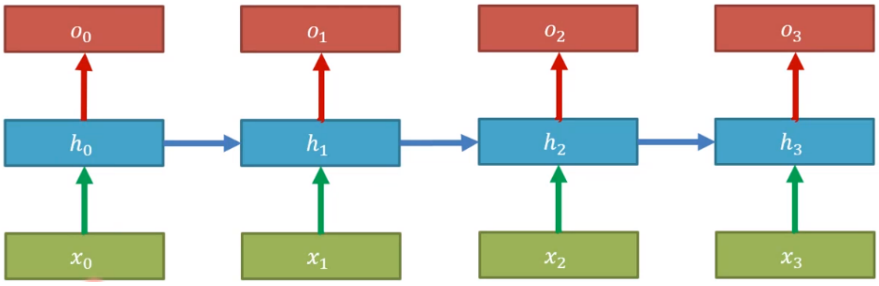


Figura 3.2.3.3: Secondo modo alternativo di rappresentare una RNN.

Questa rappresentazione ci fa vedere come, man mano, step temporale dopo step temporale, viene prodotta l'uscita della rete neurale.

Allo step temporale 0 viene mandato in ingresso alla rete il campione x_0 (è un vettore contenente le features di tale campione). Dopodiché, tale campione (con tutte le sue features) viene mandato in ingresso al Recurrent layer, il quale produrrà l'uscita h_0. Tale uscita va in ingresso all'Output Layer, il quale genera l'uscita o_0.

Allo step temporale 1 viene mandato in ingresso alla rete il campione successivo x_1 (è un vettore contenente le features di tale campione). Dopodiché, il Recurrent layer, in ingresso, riceve il campione x_1 (con tutte le sue features) e l'uscita h_0 generata dallo stesso Recurrent Layer ma allo step temporale precedente e in uscita produce l'uscita h_1. Tale uscita va in ingresso all'Output Layer, il quale genera l'uscita o_1.

Quanto appena affermato vale anche per la generazione delle varie uscite negli step temporali successivi.

DIPENDENZE A LUNGO TERMINE

Un problema tipico delle RNN è quello di riuscire a modellare le cosiddette **dipendenze a lungo termine**.

Le RNN imparano a connettere informazioni passate a informazioni future che vengono estrapolate sulla base del **contesto** in cui si trovano le informazioni passate.

Ad esempio, se l'informazione passata è **le nuvole sono nel** la RNN grazie al contesto di tale frase molto probabilmente connetterà tale informazione passata all'informazione futura **cielo**. In questo caso, la RNN riesce facilmente a predire la frase futura in quanto il contesto su cui si basa la rete, nel tempo, è vicinissimo all'informazione futura che la rete deve predire.

Quando il contesto necessario per predire l'informazione futura da connettere a quella passata è troppo lontano nel tempo rispetto all'informazione futura da predire la RNN potrebbe fallire e dunque predire un'informazione scorretta. Se ad esempio l'informazione passata è **Sono cresciuta in Francia. Ho due fratelli. Sto studiando all'università. La mia prima lingua è** dal momento che il contesto **Sono cresciuta in Francia** è troppo distante nel tempo rispetto all'informazione futura da predire, informazione che in questo caso è **francese**, la RNN potrebbe anche non riuscire a predire correttamente l'informazione futura **francese**.

La causa di ciò è da ricondurre a quello che viene chiamato problema del **vanishing gradient**. Esso consiste nel fatto che, man mano che si va avanti con gli **step temporali**, il gradiente della Loss Function relativo ad ingressi che la RNN ha ricevuto in step temporali meno recenti diventa sempre più piccolo fino quasi a svanire del tutto e dunque ad annullarsi. La conseguenza importante di ciò è che, man mano che si va avanti con gli step temporali e dunque nuovi ingressi vengono forniti alla RNN, gli ingressi che la RNN ha ricevuto in step temporali meno recenti non influenzeranno affatto l'uscita generata allo step temporale corrente. Ciò significa che, nel produrre un'uscita allo step temporale corrente, la RNN terrà conto solo ed esclusivamente degli ingressi che le sono stati forniti in step temporali molto recenti, ossia vicini a quello corrente. Tutto ciò fa sì che la RNN tradizionale abbia non poche difficoltà nell'apprendere le **dipendenze a lungo termine**.

Al fine di risolvere il problema delle **dipendenze a lungo termine**, piuttosto che utilizzare una RNN tradizionale come quella di cui abbiamo parlato finora, si utilizza la cosiddetta **LSTM (Long Short Term Memory)**, la cui architettura è quella mostrata nella figura sottostante:

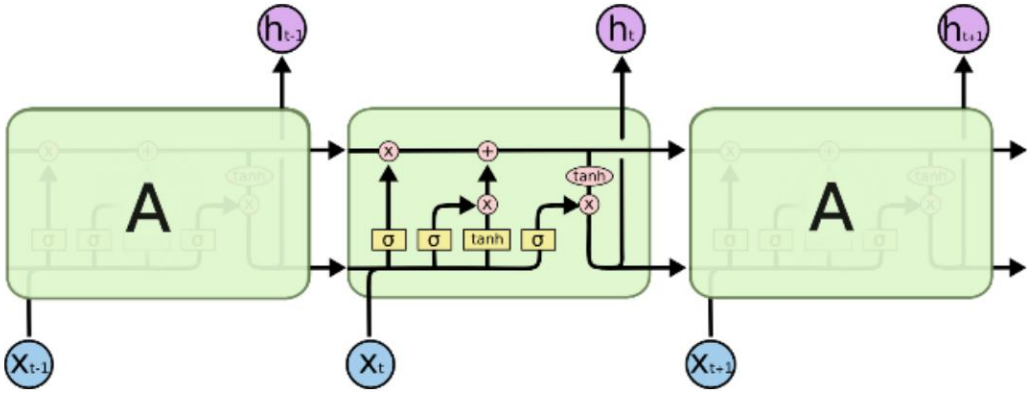


Figura 3.2.3.4: Architettura di una LSTM.

3.3 Geometric Deep Learning

Negli ultimi anni la comunità scientifica ha iniziato ad interessarsi, in maniera molto approfondita, ai cosiddetti **dati non euclidei** come ad esempio i **grafi** in quanto si è visto che, ad esempio, attraverso i grafi è possibile modellare una serie di fenomeni molto complessi del mondo reale che, se opportunamente studiati utilizzando il Machine Learning, possono portare alla risoluzione di molti problemi del mondo reale ancora tutt'oggi irrisolti a causa degli elevatissimi tempi di computazione richiesti per la loro risoluzione.

Purtroppo, però, utilizzare il Machine Learning per realizzare dei modelli al fine di apprendere da dati che sono **non euclidei** non è stato per niente facile in quanto tutti i campi del Machine Learning, incluso il Deep Learning, che negli ultimi anni si sono evoluti tantissimo fino a garantire delle prestazioni inimmaginabili fino a qualche anno fa, possono essere applicati solo ed esclusivamente a **dati euclidei**.

Per questo motivo, negli ultimi anni, si è assistito alla nascita di un nuovo campo del Machine Learning denominato **Geometric Deep Learning**, il cui scopo è quello di realizzare dei modelli in grado di apprendere da **dati non euclidei**.

Fatta questa premessa presentiamo una serie di approcci che è possibile utilizzare per operare con i **dati non euclidei** e una serie di task che è possibile attuare utilizzando il Geometric Deep Learning.

3.3.1 Approccio Spettrale

Un approccio che è possibile adottare per utilizzare i modelli di Deep Learning anche in presenza di dati non euclidei è quello che prevede l'utilizzo di una serie di tecniche prese a prestito dalla **teoria spettrale dei grafi** [19].
Quest'ultima prevede l'utilizzo degli autovalori e autovettori associati alla matrice laplaciana e alla matrice di adiacenza del grafo.
Purtroppo, in presenza di grafi estremamente grandi, calcolare gli autovettori è un'impresa ardua, il che rende tale approccio inutilizzabile nella pratica.

3.3.2 Embedding

Dato un nodo di un grafo l'obiettivo dell'**embedding** è quello di riuscire ad apprendere una rappresentazione compatta di questo nodo in grado di contenere informazioni essenziali quali la posizione del nodo all'interno del grafo e la struttura del grafo locale costituito da tutti i nodi vicini a tale nodo. In altre parole, tramite l'**embedding** vogliamo prendere ogni singolo nodo del grafo originario e lo vogliamo proiettare in uno spazio chiamato **spazio latente** o **embedding space** in maniera tale che le relazioni tra i nodi, raffigurate per mezzo di archi, che sussistono tra i nodi presenti nel **grafo originario** corrispondano a delle relazioni geometriche all'interno dello **spazio latente (embedding space)**.

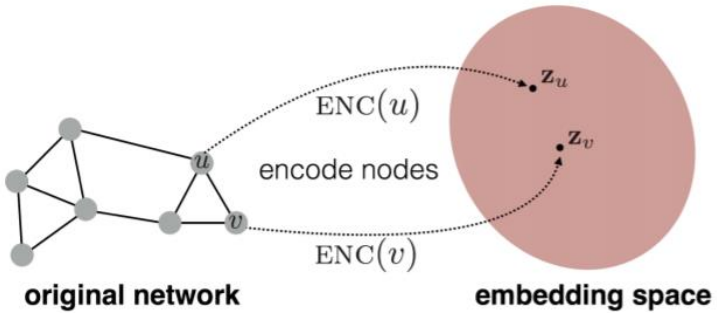


Figura 3.3.2.1: Embedding.

In pratica, se ad esempio all'interno del **grafo originario** abbiamo un nodo *u* e un nodo *v* che sono collegati tramite un arco, tramite l'**embedding** quello che accade è che del nodo *u* viene generata una rappresentazione compatta, del nodo *v* viene generata una rappresentazione compatta in maniera tale che passando dal **grafo originario** all'**embedding space** quella che nel **grafo originario** era una relazione tra il nodo *u* e il nodo *v* espressa tramite un arco nell'**embedding space** diventi una relazione geometrica tra le rappresentazioni di questi nodi.

In un modo ancora differente possiamo affermare che dati due nodi *u* e *v* l'obiettivo dell'**embedding** è quello di determinare una rappresentazione compatta del nodo *u* e del nodo *v* in maniera tale che la **similarità** tra le rappresentazioni z_u e z_v presenti all'interno dell'**embedding space** approssimi il più possibile la **similarità** tra i nodi *u* e *v* presenti all'interno del **grafo originario** [20]:

$$similarity(u, v) \approx z_v^T z_u$$

In altre parole, vogliamo che facendo il prodotto scalare tra le rappresentazioni z_v^T e z_u presenti all'interno dell'**embedding space** venga fuori un numero che sia quanto più simile possibile a quello ottenuto applicando una **funzione di similarità** ai nodi *u* e *v* presenti nel **grafo originario**.

A questo punto vediamo quali sono gli step che occorre seguire per fare ciò:

1. Definire un **encoder**.
2. Definire una **funzione di similarità**.
3. **Ottimizzare** i parametri dell'**encoder** in maniera tale che:

   $$similarity(u, v) \approx z_v^T z_u$$

   la **similarità** tra le rappresentazioni z_u e z_v presenti all'interno dell'**embedding space** approssimi il più possibile la **similarità** tra i nodi *u* e *v* presenti all'interno del **grafo originario**. In altre parole, vogliamo **ottimizzare** i parametri dell'encoder in maniera tale che facendo il prodotto scalare tra le rappresentazioni z_v^T e z_u presenti all'interno dell'**embedding space** venga fuori un numero che sia quanto più simile possibile a quello ottenuto applicando una **funzione di similarità** ai nodi *u* e *v* presenti nel **grafo originario**.

A questo punto vediamo prima l'**encoder** e poi degli esempi di **funzioni di similarità** che possono essere utilizzate.

ENCODER

L'**encoder** riceve in ingresso un nodo *v* appartenente all'insieme *V* (insieme di tutti i nodi del grafo) e restituisce un vettore $z_v \in R^d$, ossia un vettore contenente *d* elementi [21].

In questa trattazione stiamo prendendo in considerazione una tipologia di **encoder** che si basa su un approccio chiamato **shallow embedding**. In virtù di tale approccio l'**encoder**, in realtà, è semplicemente una matrice Z $\in R^{|v| \, x \, d}$ avente un numero di righe pari al numero di nodi del grafo e un numero di colonne pari al numero di elementi della rappresentazione di ogni singolo nodo del grafo.

La conseguenza di ciò è che, ogni qualvolta all'**encoder** viene fornito l'**indice** di un nodo *v*, esso semplicemente restituisce la rappresentazione, ossia la riga della matrice Z corrispondente all'**indice** del nodo *v*.

La cosa importante da sottolineare è che, dal momento che di ogni singolo nodo non stiamo prendendo in considerazione né le features e nemmeno la struttura del grafo locale costituito dai nodi vicini ad ogni singolo nodo, di fatto, per adesso, stiamo attenzionando un encoder molto semplice che si basa sull'approccio *shallow embedding* e che dunque in ingresso riceve semplicemente l'indice del nodo e in uscita produce la rappresentazione compatta di tale nodo.

Successivamente, quando parleremo delle **Graph Neural Network** vedremo che esistono anche degli encoder in grado di generare una rappresentazione compatta di un nodo ricevendo in ingresso, non l'indice del nodo ma qualcosa di molto più complesso, come ad esempio, le features di un nodo e/o la struttura del grafo locale costituito dai nodi vicini a tale nodo.

Per adesso, però, concentriamoci solo ed esclusivamente sull'**encoder** basato sull'approccio **shallow embedding**.

FUNZIONI DI SIMILARITA'

A seconda delle tecniche utilizzate per misurare la similarità tra due nodi esistono diverse **funzioni di similarità**. Le tecniche utilizzate per misurare la similarità tra due nodi sono:

1. **Adjacency-based similarity**
2. **Multi-hop similarity**
3. **Approccio Random Walk**

Iniziamo a parlare della **Adjacency-based similarity**.

In questo caso, per calcolare la **similarità** tra due nodi: *u* e *v* come **funzione di similarità** si utilizza il **peso** dell'arco che collega il nodo *u* e il nodo *v*.

Grazie al fatto che, come funzione di similarità, si utilizza il **peso** dell'arco che collega il nodo *u* e il nodo *v* possiamo affermare che, in questo caso, alla fine l'obiettivo è quello di **ottimizzare** i parametri dell'**encoder** in maniera tale che facendo il prodotto scalare tra le rappresentazioni z_v^T e z_u presenti all'interno dell'**embedding space** venga fuori un numero che sia quanto più simile possibile al **peso** dell'arco che collega il nodo *u* e il nodo *v* all'interno del **grafo originario**.

Per raggiungere questo obiettivo, per prima cosa dobbiamo calcolare la **Loss Function** utilizzando la formula sottostante:

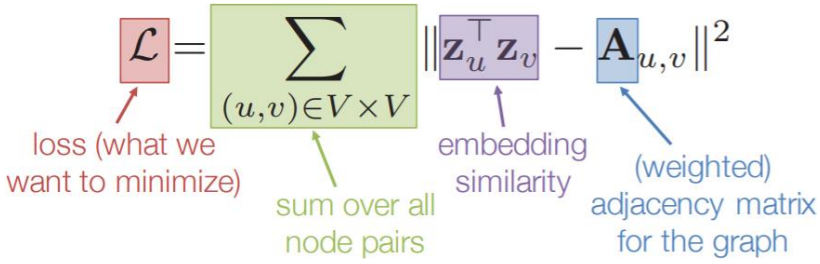


Figura 3.3.2.2: Loss Function nel caso di Adjacency-based similarity.

Per calcolare la **Loss Function** si fa così: si prende ogni singola coppia di nodi *u* e *v*, si passano i nodi *u* e *v* all'encoder, si prelevano le rappresentazioni z_u e z_v presenti all'interno dell'**embedding space**, si calcola il modulo della differenza tra il prodotto z_u^T, z_v e $A_{u,v}$ (elemento (peso) presente all'interno della riga corrispondente al nodo *u* e all'interno della colonna corrispondente al nodo *v* della **matrice di adiacenza**), si eleva tutto al quadrato e, infine, si effettua la somma. Una volta calcolata la **Loss Function** l'obiettivo è quello di trovare la matrice $Z \in R^{|v| \, x \, d}$ dell'**encoder** tale per cui la Loss Function è minima. Per fare ciò come metodo di ottimizzazione si può utilizzare quello che viene chiamato **Stochastic Gradient Descent** (SGD).

Adesso concentriamoci sull'approccio **Random Walk**.
In questo caso, per calcolare la **similarità** tra due nodi: *u* e *v* come **funzione di similarità** si utilizza una **probabilità** e, in particolare, **la probabilità che il nodo *u* e il nodo v facciano parte entrambi di uno stesso cammino casuale del grafo**.
Grazie al fatto che, come funzione di similarità, si utilizza la **probabilità** appena definita possiamo affermare che, in questo caso, alla fine l'obiettivo è quello di **ottimizzare** i parametri dell'**encoder** in maniera tale che facendo il prodotto scalare tra le rappresentazioni z_u^T e z_v presenti all'interno dell'**embedding space** venga fuori un numero che sia quanto più simile possibile alla **probabilità che il nodo *u* e il nodo v facciano parte entrambi di uno stesso cammino casuale del grafo**.
In altre parole, vogliamo **ottimizzare** i parametri dell'**encoder** in maniera tale che:

$$P_r(v|u) = \frac{e^{z_u^T z_v}}{\sum_{n \in V} e^{z_u^T z_n}} \approx z_u^T z_v$$

dove *V* è l'insieme di tutti i nodi del grafo.
Nell'approccio **Random Walk** la **Loss Function** che si utilizza è la seguente:

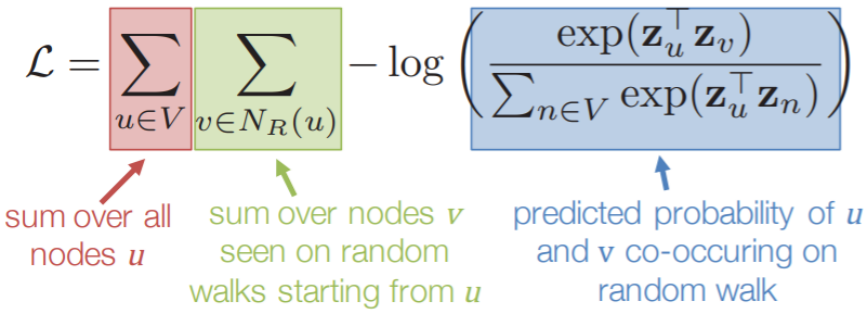


Figura 3.3.2.3: Loss Function nel caso di approccio Random Walk.

Per concludere, i passi da seguire ogni qualvolta si utilizza l'approccio **Random Walk** sono i seguenti:

1. Per ogni singolo nodo *u* del grafo occorre eseguire un algoritmo in grado di determinare un cammino casuale, il quale parte proprio da tale nodo *u* e contiene un certo numero di nodi.
2. Una volta fatto ciò, per ogni singolo nodo *u* del grafo, è necessario memorizzare $N_R(u)$, ossia l'insieme di tutti i nodi facenti parte del cammino casuale (random walk) determinato a partire dal nodo *u*.
3. Infine, dopo aver calcolato la **Loss Function** utilizzando la formula sovrastante, l'obiettivo è quello di trovare la matrice $Z \in R^{|v| \, x \, d}$ dell'**encoder** tale per cui la Loss Function è minima.

3.3.3 Graph Neural Network

Nei paragrafi precedenti sono stati presentati alcuni approcci che possono essere utilizzati per generare una rappresentazione compatta di ogni singolo nodo del grafo. Tali approcci, come detto, si basano sul cosiddetto **shallow embedding** in quanto consentono di generare una rappresentazione (embedding) di un nodo tenendo conto solo ed esclusivamente dell'indice del nodo e trascurando informazioni essenziali come, ad esempio, le features del nodo e la struttura del grafo locale costituito dai nodi vicini a tale nodo. Adesso parleremo delle **Graph Neural Networks**, le quali possono essere utilizzate per generare una rappresentazione (embedding) di ogni singolo nodo del grafo tenendo conto delle features di tale nodo e/o della struttura del grafo locale costituito dai nodi vicini a tale nodo.

Dunque, grazie ad una **Graph Neural Network**, dato un **grafo** *G = (V, E)*, dove *V* è l'insieme di tutti i nodi, *E* è l'insieme di tutti gli archi e data la matrice $X \in R^{d \times |v|}$, ossia la matrice contenente le features di tutti i nodi del grafo, per ogni singolo nodo $v \in V$ del grafo riusciamo a generare una rappresentazione (embedding).

La cosa importante da sottolineare è che la caratteristica principale di una GNN è che essa utilizza un framework chiamato *neural message passing* tramite il quale una serie di vettori di messaggi vengono scambiati tra i nodi e aggiornati utilizzando delle reti neurali.

OVERVIEW DEL FRAMEWORK NEURAL MESSAGE PASSING

Durante ogni singola iterazione *k* in cui avviene lo scambio di messaggi tra i nodi, all'interno di una GNN, accade che per ogni singolo nodo *u* viene effettuato l'aggiornamento di quello che viene chiamato **hidden embedding** $\boldsymbol{h_u^{(k)}}$. Tale aggiornamento viene fatto aggregando i messaggi (informazioni) provenienti dai nodi appartenenti al grafo *N(u)*, ossia al grafo costituito dai nodi vicini al nodo *u*.

Per comprendere il funzionamento supponiamo di considerare il grafo mostrato nella figura sottostante:

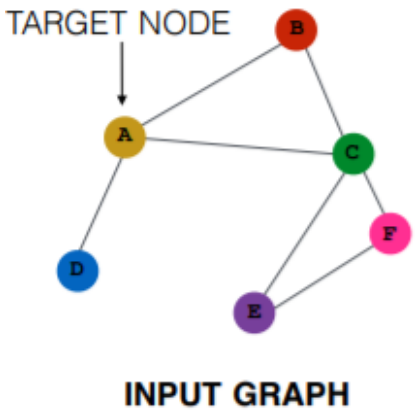


Figura 3.3.3.1: Esempio di grafo non orientato per il Neural Message Passing.

e supponiamo di focalizzare l'attenzione sul nodo A. In base a ciò che abbiamo appena detto, ad ogni singola iterazione *k*, all'interno della GNN accade che per il nodo A viene effettuato l'aggiornamento dell'**hidden embedding** $\boldsymbol{h_A^{(k)}}$ aggregando i messaggi (informazioni) provenienti dai nodi B, C, D appartenenti al grafo *N(A)*, ossia al grafo costituito dai nodi vicini al nodo A:

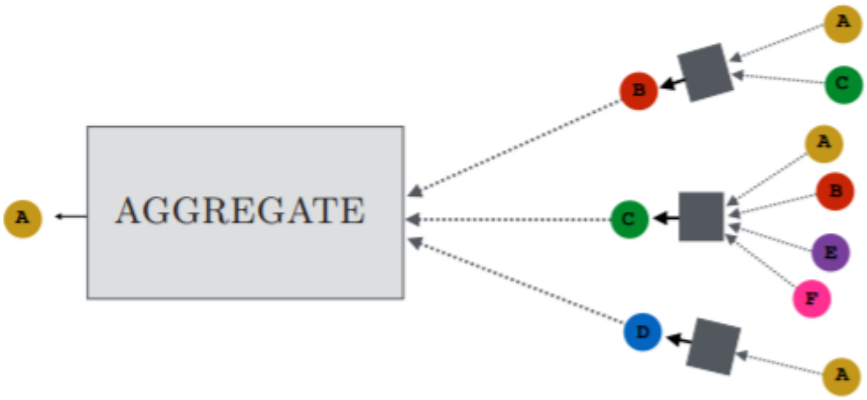


Figura 3.3.3.2: Esempio di applicazione del framework Neural Message Passing.

Ovviamente, ogni singolo nodo vicino al nodo A, affinché possa generare il messaggio (informazione) da inviare al nodo A, a sua volta, aggregherà i messaggi (informazioni) provenienti dai nodi ad esso vicino e così via.

In particolare, il nodo B per generare il messaggio da inviare al nodo A aggregherà i messaggi provenienti dai nodi A e C, il nodo C per generare il messaggio da inviare al nodo A aggregherà i messaggi provenienti dai nodi A, B, E ed F, il nodo D per generare il messaggio da inviare al nodo A considererà solo ed esclusivamente il messaggio proveniente dal nodo A.

Adesso vediamo come da un punto di vista matematico, all'iterazione *k+1*, è possibile effettuare l'aggiornamento dell'**hidden embedding** $\boldsymbol{h_u^{(k+1)}}$ relativo al nodo *u*.

La formula che ci consente di fare ciò è la seguente:

$$\begin{aligned}\mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right) \\ &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right),\end{aligned}$$

In pratica, all'iterazione *k+1* per effettuare l'aggiornamento dell'**hidden embedding** $\boldsymbol{h_u^{(k+1)}}$ relativo al nodo *u* si prende in considerazione l'**hidden embedding** $\boldsymbol{h_u^{(k)}}$ relativo al nodo *u* calcolato all'iterazione precedente *k* e si prende in considerazione anche il messaggio risultante dall'aggregazione dei messaggi che il nodo *u* ha ricevuto dai nodi appartenenti al grafo *N(u)* all'iterazione precedente *k*.

La cosa importante da sottolineare è che le differenti iterazioni in cui avviene lo scambio di messaggi tra i nodi vengono chiamati anche **layers** della GNN.

Inoltre, all'iterazione iniziale, ossia all'iterazione *k=0*, l'**hidden embedding** $\boldsymbol{h_u^{(0)}}$ di ogni singolo nodo *u* è pari al vettore contenente tutte le features del nodo *u*.

Dunque, per sintetizzare, possiamo affermare che, all'iterazione 0 l'**hidden embedding** $\boldsymbol{h_u^{(0)}}$ di ogni singolo nodo *u* è pari al vettore contenente tutte le features del nodo *u*.

Dopodiché, ad ogni iterazione *k* viene effettuato l'aggiornamento dell'**hidden embedding** $\boldsymbol{h_u^{(k)}}$ di ogni singolo nodo *u* aggregando i messaggi (informazioni) provenienti dai nodi appartenenti al grafo *N(u)*, ossia al grafo costituito dai nodi vicini al nodo *u*.

Infine, dopo aver eseguito un certo numero di iterazioni l'**hidden embedding** $\boldsymbol{h_u^{(K)}}$ di ogni singolo nodo *u* che viene prodotto al termine dell'ultima iterazione (ultimo layer) *K* rappresenta proprio l'embedding, ossia la rappresentazione z_u generata per il nodo *u*:

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}$$

MOTIVAZIONI

Per quale motivo la GNN si basa sul framework **neural message passing**? Il motivo è molto semplice: man mano che aumenta il numero di iterazioni all'interno della rappresentazione (embedding) di ogni singolo nodo *u* vengono aggiunte delle informazioni calcolate tenendo conto delle informazioni provenienti da nodi che, man mano, sono sempre più distanti dal nodo *u*.

Se vogliamo essere ancora più precisi possiamo affermare che all'iterazione *k=1* all'interno dell'embedding di ogni singolo nodo *u* verranno aggiunte delle informazioni calcolate tenendo conto solo ed esclusivamente delle informazioni provenienti da tutti i nodi vicini al nodo *u* che possono essere raggiunti attraverso un numero di hops pari a 1, all'iterazione *k=2* all'interno dell'embedding di ogni singolo nodo *u* verranno aggiunte delle informazioni calcolate tenendo conto anche delle informazioni provenienti da tutti i nodi vicini al nodo *u* che possono essere raggiunti attraverso un numero di hops pari a 2, all'iterazione *k* all'interno dell'embedding di ogni singolo nodo *u* verranno aggiunte delle informazioni calcolate tenendo conto anche delle informazioni provenienti da tutti i nodi vicini al nodo *u* che possono essere raggiunti attraverso un numero di hops pari a *k*.

GNN: SCENDIAMO NEL DETTAGLIO

Precedentemente abbiamo detto che la GNN si basa sul framework **neural message passing** e abbiamo visto che durante ogni singola iterazione *k* in cui avviene lo scambio di messaggi tra i nodi, all'interno di una GNN, accade che per ogni singolo nodo *u* viene effettuato l'aggiornamento di quello che viene chiamato **hidden embedding** $\boldsymbol{h_u^{(k)}}$ utilizzando una formula in cui figurano due funzioni: UPDATE E AGGREGATE.

Ovviamente, tale formula è totalmente astratta. Di conseguenza, se vogliamo riuscire ad implementare concretamente una GNN quello che dobbiamo fare è trasformare la formula vista, la quale è astratta, in una formula concreta. La formula che, ad ogni singola iterazione *k*, consente di effettuare l'aggiornamento dell'**hidden embedding** $\boldsymbol{h_u^{(k)}}$ relativo ad ogni singolo nodo *u* è la seguente [22]:

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)}\sum_{v\in\mathcal{N}(u)}\mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right)$$

dove $W_{self}^{(k)}$, $W_{neigh}^{(k)}$ sono delle matrici contenenti dei pesi che, man mano, per effetto del training vengono aggiornati, σ è una funzione di attivazione (solitamente si utilizza la **tanh** o la **ReLU**), $b^{(k)}$ è il vettore contenente i bias che, man mano, per effetto del training vengono aggiornati, $h_u^{(k-1)}$ è l'**hidden embedding** relativo al nodo *u* calcolato all'iterazione precedente *k-1*, $h_v^{(k-1)}$ è l'**hidden embedding** relativo al nodo *v* (appartenente al grafo *N(u)*, ossia al grafo costituito dai nodi vicini al nodo *u*) calcolato all'iterazione precedente *k-1* e *N(u)* è il grafo costituito dai nodi vicini al nodo *u*.

Una volta compreso il funzionamento di una GNN possiamo affermare che una Graph Neural Network può essere allenata sia in maniera **non supervisionata** sia in maniera **supervisionata**.

Nel primo caso la rete neurale viene allenata utilizzando solo ed esclusivamente la struttura di un certo numero di grafi, mentre nel secondo caso viene allenata per attuare una serie di tasks di cui parleremo come ad esempio node classification o link prediction.

In aggiunta alla rete neurale vista finora esistono anche altre tipologie di reti neurali che differiscono da quella di cui abbiamo parlato finora per la **funzione di aggregazione** che utilizzano per aggiornare l'**hidden embedding** $\boldsymbol{h_u^{(k)}}$ relativo ad ogni singolo nodo *u* del grafo.

3.3.4 Graph Convolutional Neural Network

Una **Graph Convolutional Neural Network** da un punto di vista architetturale e concettuale è identica alla Graph Neural Network presentata nel paragrafo precedente in quanto anche una GCNN adotta il framework denominato **neural message passing**.

Ma allora, *una GCNN in cosa differisce rispetto ad una GNN?*

Una **Graph Convolutional Neural Network** utilizza una **funzione di aggregazione** per aggiornare l'**hidden embedding** $\boldsymbol{h_u^{(k)}}$ relativo ad ogni singolo nodo *u* del grafo che è differente rispetto a quella utilizzata da una Graph Neural Network.

In particolare, nel caso di una **Graph Convolutional Neural Network**, la formula che, ad ogni singola iterazione *k*, consente di effettuare l'aggiornamento dell'**hidden embedding** $\boldsymbol{h_u^{(k)}}$ relativo ad ogni singolo nodo *u* è la seguente [23]:

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v^{(k-1)}}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right)$$

dove $W^{(k)}$ è una matrice contenente dei pesi che, man mano, per effetto del training vengono aggiornati, σ è una funzione di attivazione (solitamente si utilizza la **tanh** o la **ReLU**), $h_v^{(k-1)}$ è l'**hidden embedding** relativo al nodo *v* (appartenente al grafo *N(u)*, ossia al grafo costituito dai nodi vicini al nodo *u*) calcolato all'iterazione precedente *k*-1, *N(u)* è il grafo costituito dai nodi vicini al nodo *u*, *N(v)* è il grafo costituito dai nodi vicini al nodo *v*, |*N(u)*| è il numero di nodi appartenenti al grafo *N(u)* e |*N(v)*| è il numero di nodi appartenenti al grafo *N(v)*.

3.3.5 GraphSAGE

Una rete **GraphSAGE** da un punto di vista architetturale e concettuale è identica alla Graph Neural Network presentata nei paragrafi precedenti in quanto anche una rete **GraphSAGE** adotta il framework denominato **neural message passing**.

Ma allora, *una rete GraphSAGE in cosa differisce rispetto ad una GNN?*

Una rete **GraphSAGE** utilizza una **funzione di aggregazione** per aggiornare l'**hidden embedding** $\boldsymbol{h_u^{(k)}}$ relativo ad ogni singolo nodo *u* del grafo che è differente rispetto a quella utilizzata da una Graph Neural Network.

In particolare, nel caso di una rete **GraphSAGE**, la formula che, ad ogni singola iterazione *k*, consente di effettuare l'aggiornamento dell'**hidden embedding** $\boldsymbol{h_u^{(k)}}$ relativo ad ogni singolo nodo *u* è la seguente [24]:

$$h_u^{(k)} = \sigma\left(W^{(k)} \cdot CONCAT\left(h_u^{(k-1)}, AGGREGATE^{(k)}\left(\left\{h_v^{(k-1)}, \forall\, v \in N(u)\right\}\right)\right)\right)$$

dove $W^{(k)}$ è una matrice contenente dei pesi che, man mano, per effetto del training vengono aggiornati, σ è una funzione di attivazione (solitamente si utilizza la **tanh** o la **ReLU**), $h_v^{(k-1)}$ è l'**hidden embedding** relativo al nodo *v* (appartenente al grafo *N(u)*, ossia al grafo costituito dai nodi vicini al nodo *u*) calcolato all'iterazione precedente *k*-1, *N(u)* è il grafo costituito dai nodi vicini al nodo *u* e $h_u^{(k-1)}$ è l'**hidden embedding** relativo al nodo *u* calcolato all'iterazione precedente *k-1*.

La formula sovrastante, purtroppo, è astratta, dunque, ancora una volta, se vogliamo implementare GraphSAGE nella pratica dobbiamo passare da questa formula astratta ad una formula concreta. Per fare ciò dobbiamo specificare quale o quali **funzioni di aggregazione (AGGREGATE)** è possibile utilizzare nella pratica. Nel caso di **GraphSAGE** sono tre le possibili **funzioni di aggregazione** che è possibile utilizzare:

- **Mean Aggregate Function**: in tal caso aggregare significa calcolare la media tra le features dei diversi nodi di cui si vuole effettuare l'aggregazione.
- **Pooling Aggregate Function**: in tal caso, dato un certo numero di nodi di cui vogliamo effettuare l'aggregazione, aggregare significa prendere la feature *i-esima* di tali nodi e calcolare il massimo o il minimo, la feature *i+1* di tali nodi e calcolare il massimo o il minimo, la feature *i+2* di tali nodi e calcolare il massimo o il minimo e così via per tutte le altre features di tali nodi.
- **LSTM Aggregate Function**: in tal caso, al fine di effettuare l'aggregazione, viene utilizzata una LSTM (Long Short Term Memory).

Per concludere possiamo affermare che la **funzione di aggregazione** utilizzata da **GraphSAGE** deve essere simmetrica, ossia invariante al variare dell'ordine con il quale i nodi vengono forniti alla rete.

3.3.6 Tasks

All'interno di questo paragrafo presenteremo alcuni esempi di task che possono essere attuati utilizzando il Geometric Deep Learning.

NODE CLASSIFICATION

Supponiamo di avere a disposizione un dataset di un social network contenente milioni di utenti, dove però all'interno di questo dataset sappiamo già che un numero estremamente elevato di utenti, in realtà, sono **bot**. Riuscire ad identificare quali di questi utenti sono dei **bot** può essere estremamente importante. Di conseguenza, potrebbe essere utile creare un modello di Geometric Deep Learning in grado di classificare ogni singolo utente che gli viene fornito come **bot** o come utente reale. Questo è un esempio di task di tipo supervisionato in quanto all'interno del dataset, per ogni singola osservazione (utente) abbiamo anche il corrispondente target che ad esempio sarà 0 se l'utente è reale, 1 se è un bot. Utilizzando questo dataset, si può effettuare il training di un modello in maniera tale che dato un utente il modello ci possa dire se esso è reale oppure è un bot.

Ciò che abbiamo appena visto è un esempio di **node classification**. Solitamente, la node classification è un task di tipo supervisionato in quanto all'interno del dataset, per ogni singola osservazione (utente) abbiamo anche il corrispondente target (se l'utente è un bot o è reale).

LINK PREDICTION

Questo è un altro task estremamente importante. In cosa consiste la **link prediction**? La **link prediction** consiste, come suggerisce anche il termine, nel predire un collegamento tra due nodi. Nell'ambito della **link prediction** abbiamo a disposizione un insieme di nodi V e un insieme di archi E_{train} dove, però, tale insieme di archi E_{train} NON contiene tutti gli archi del grafo ma ne contiene soltanto un sottoinsieme.

In tal caso, l'obiettivo è quello di realizzare un modello in grado di predire quali sono i collegamenti, ossia gli archi mancanti che sono presenti nell'insieme di tutti gli archi del grafo E ma che non sono presenti nell'insieme E_{train} che abbiamo a disposizione.

La **link prediction** viene utilizzata moltissimo per la realizzazione di quella che viene chiamata **content recommendation**:

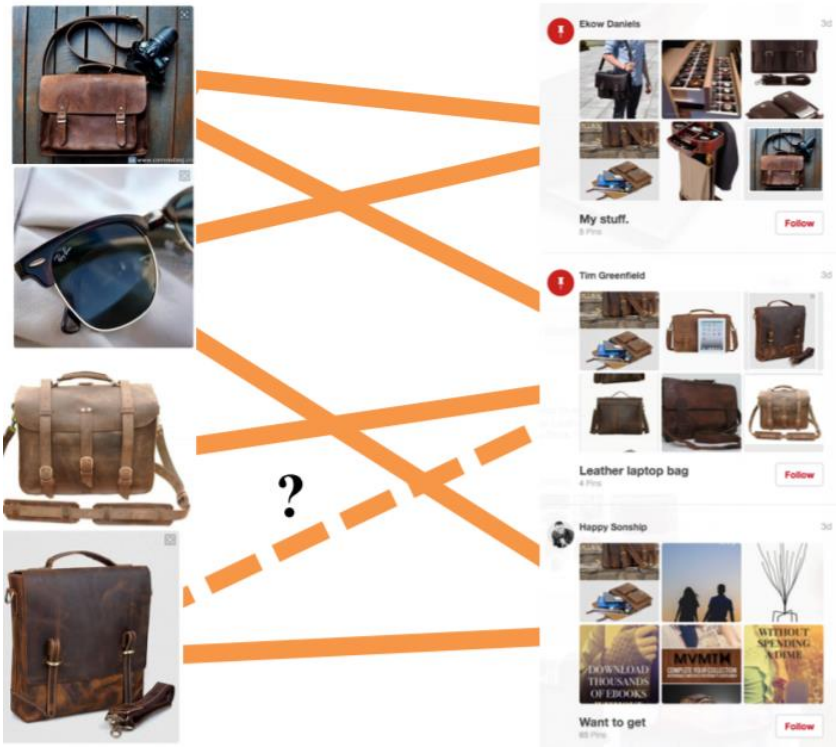


Figura 3.3.6.1: Link Prediction per la Content Recommendation.

I **recommendation system** sono dei sistemi che, ad esempio, sulla base di ciò che una persona ha acquistato su un determinato sito di e-commerce sono in grado di consigliarle una serie di altri articoli come ad esempio: borse, vestiti, ecc.

Capitolo 4

Model Design e Implementazione

4.1 Strategia risolutiva

Nei paragrafi precedenti abbiamo sottolineato il fatto che l'obiettivo del presente lavoro di tesi è la riduzione dei tempi di computazione del problema di **determinazione delle clique di dimensione massima di un grafo**, il quale, come detto, è un problema di tipo **NP-Hard**.
L'obiettivo appena citato può essere raggiunto adottando una strategia risolutiva che prevede i seguenti passi:

1. Allenare un modello di Geometric Deep Learning che, dato un grafo *x* in ingresso, in uscita, per ogni singolo nodo del grafo *x*, con un certo grado di accuratezza predice se quel nodo appartiene ad una clique di dimensione massima oppure no.
2. Sfruttare tale modello (che verrà descritto dettagliatamente all'interno di questo capitolo) per attuare un'operazione di **node pruning** tramite la quale, rispettando fedelmente le predizioni prodotte in uscita dal modello, è possibile eliminare dal grafo *x* tutti i nodi che non fanno parte di alcuna clique di dimensione massima e dunque ricavare un grafo *y* che contiene solo ed esclusivamente i nodi del grafo *x* che il modello predice come appartenenti ad una clique di dimensione massima.
3. Applicare il Solver (algoritmo in grado di determinare tutte le clique di dimensione massima di un grafo) direttamente al grafo *y*, il quale, come detto, rispetto al grafo originario *x* è di dimensione ridotta.

Seguendo questa strategia, dal momento che il Solver non viene applicato al grafo originario *x*, il quale può essere anche di dimensioni significative, ma al grafo di dimensione ridotta *y* riusciamo a ridurre i tempi di risoluzione del problema sopracitato.

4.2 Features utilizzate

Nel paragrafo precedente abbiamo sottolineato il fatto che l'obiettivo fondamentale del presente lavoro di tesi è quello di allenare un modello di Geometric Deep Learning.
Dal momento che stiamo parlando di un modello di Geometric Deep Learning è lecito pensare che a tale modello, in ingresso, debba essere fornito un grafo costituito da un certo numero di nodi e archi.

La cosa estremamente importante da sottolineare è che ogni singolo nodo del grafo che viene fornito in ingresso ad un modello di Geometric Deep Learning è caratterizzato da un certo numero di features.
Nel presente lavoro di tesi, per ogni singolo nodo *x* del grafo, sono state utilizzate 4 features:

- **Grado normalizzato del nodo *x***: è il rapporto tra il grado del nodo *x* (numero di archi incidenti il nodo *x*) e il grado massimo del grafo.
- **Local Clustering Coefficient (LCC) del nodo *x***: indica la frazione di nodi vicini al nodo *x* che formano un triangolo con il nodo *x*.
- **Chi-squared del Grado del nodo *x***
- **Chi-squared del Local Clustering Coefficient del nodo *x***

4.3 Modello utilizzato

All'interno di questo paragrafo presenteremo l'architettura e il funzionamento di una Graph Attention Network e, infine, mostreremo com'è strutturato il modello di Geometric Deep Learning utilizzato nella pratica nel presente lavoro di tesi.

4.3.1 Graph Attention Network

Una Graph Attention Network è un'evoluzione della Graph Neural Network vista nel capitolo precedente in quanto si basa anch'essa sul framework **neural message passing** ma introduce un meccanismo innovativo denominato **meccanismo di attenzione**.
Il **meccanismo di attenzione** è innovativo in quanto consente alla rete neurale di focalizzare l'**attenzione** solo ed esclusivamente su quelle porzioni di grafo che effettivamente sono importanti. Questo meccanismo si sposa benissimo con l'obiettivo che la tesi vuole perseguire in quanto attuare un'operazione di **node pruning** equivale ad eliminare dal grafo tutti quei nodi che non sono ritenuti importanti ai fini della risoluzione del problema. Dunque, il fatto che la rete neurale, grazie al meccanismo di attenzione, riesca ad individuare le porzioni di grafo veramente importanti consente di attuare un'operazione di **pruning** molto più efficiente.

Dato un set di nodi $\Gamma_V = \{v_0, \dots, v_n\} \subset V$ dove V è l'insieme di tutti i nodi del grafo e dato un oggetto target *s* rappresentante una specifica entità (come, ad esempio, un nodo) del grafo si definisce **attenzione** una **funzione** $\Phi'_s : \Gamma_V \rightarrow [0,1]$ che ad ogni singolo nodo *s* presente in Γ_V associa un **coefficiente di attenzione** soddisfacente la seguente relazione:

$$\sum_{i=0}^{|\Gamma_V|} \Phi'_s(v_i) = 1$$

La cosa importante da sottolineare è che esistono due tipologie di meccanismi di attenzione: **attention-based node embedding** e **attention-based graph embedding**.

Dal momento che il modello di Geometric Deep Learning utilizzato all'interno della tesi deve essere in grado di creare una rappresentazione compatta di **ogni singolo nodo** del grafo parleremo dell'**attention-based node embedding**.

Nel caso di **attention-based node embedding** l'oggetto target *s* è un generico nodo v_j del grafo mentre Γ_V è l'insieme di tutti i nodi vicini al nodo v_j. In pratica, come evidenziato nel capitolo precedente, Γ_V è proprio l'insieme $N(v_j)$.

A questo punto vediamo i passi da seguire affinché, all'iterazione *k+1*, possa essere effettuato l'aggiornamento dell'**hidden embedding** $\boldsymbol{x_i^{(k+1)}}$ relativo al nodo *i*:

1. Se con $\left\{x_1^{(k)}, \dots, x_n^{(k)}\right\} \in R^{f_k}$ indichiamo l'insieme costituito da un certo numero di elementi, dove ogni singolo elemento $x_i^{(k)}$ rappresenta l'**input feature vector** del nodo *i* all'iterazione *k*, la prima cosa da fare è applicare una trasformazione lineare all'**input feature vector** $x_i^{(k)}$ di ogni singolo nodo *i* per mezzo di una matrice $W \in R^{f_{k+1} \times f_k}$, la quale contiene un certo numero di pesi determinati durante il processo di training della rete. Applicare tale trasformazione significa che, per l'**input feature vector** $x_i^{(k)}$ di ogni singolo nodo *i*, all'iterazione *k*, viene calcolato il prodotto $Wx_i^{(k)}$.
2. Dopodiché, per ogni singolo nodo *i* del grafo viene calcolato più volte il **coefficiente di attenzione** e_{ij} considerando la prima volta il nodo *i* e se stesso (*j*=*i*) e le successive volte il nodo *i* e i nodi *j* ad esso vicini. Il **coefficiente di attenzione** di un certo nodo *i* è estremamente importante in quanto consente di determinare quanto ogni singolo nodo *j* vicino al nodo *i* è importante ai fini della determinazione dell'**embedding**, ossia della rappresentazione compatta del nodo *i*.

   Il **coefficiente di attenzione** e_{ij} di ogni singolo nodo *i* del grafo viene calcolato utilizzando la seguente formula:

$$e_{ij} = a(Wx_i^{(k)}, Wx_j^{(k)})$$

dove *j* è il generico nodo vicino al nodo *i* e dunque appartenente all'insieme *N(i)* e *a* è il **meccanismo di attenzione** utilizzato.

3. Dopo aver calcolato tutti i **coefficienti di attenzione** e_{ij} per ogni singolo nodo *i* del grafo, a causa del fatto che non tutti i nodi *i* hanno lo stesso numero di nodi vicini *j*, è necessario normalizzarli e dunque calcolare i vari **coefficienti di attenzione normalizzati** a_{ij} per ogni singolo nodo *i* del grafo.

   La formula per calcolare i vari **coefficienti di attenzione normalizzati** a_{ij} è la seguente:

$$a_{ij} = \frac{\exp\left(LeakyReLU\left(e_{ij}\right)\right)}{\sum_{k\epsilon N(i)}(LeakyReLU\left(e_{ik}\right))}$$

   Nel caso specifico in cui il **meccanismo di attenzione** *a* è una rete neurale feedforward parametrizzata da un vettore $a \in R^{2f_{k+1}}$ e avente un solo layer la formula per calcolare i vari **coefficienti di attenzione normalizzati** a_{ij} per ogni singolo nodo *i* del grafo diventa la seguente:

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\boldsymbol{a}^T(Wx_i^{(k)}||Wx_j^{(k)})\right)\right)}{\sum_{k\in\mathcal{N}(i)}\exp\left(\text{LeakyReLU}\left(\boldsymbol{a}^T(Wx_i^{(k)}||Wx_k^{(k)})\right)\right)}$$

   dove || indica l'operazione di concatenazione e *T* indica l'operazione di trasposizione.

4. Infine, è possibile calcolare l'**hidden embedding** $\boldsymbol{x_i^{(k+1)}}$ relativo al nodo *i* all'iterazione *k+1* utilizzando la seguente formula:

$$x_i^{(k+1)} = \xi\left(\sum_{j\in\mathcal{N}(i)}\alpha_{ij}Wx_j^{(k)}\right)$$

   dove ξ è una funzione di attivazione non lineare.

   Alternativamente, al fine di stabilizzare il processo di training, piuttosto che utilizzare la formula per il calcolo dell'**hidden embedding** $\boldsymbol{x_i^{(k+1)}}$ relativo al nodo *i* all'iterazione *k+1* appena descritta, è possibile utilizzare un meccanismo denominato **multi-head attention**. Tale meccanismo consente di determinare l'**hidden embedding** $\boldsymbol{x_i^{(k+1)}}$ relativo al nodo *i* all'iterazione *k+1* utilizzando la seguente formula:

$$x_i^{(k+1)} = \xi\left(\frac{1}{m}\sum_{p=1}^{m}\sum\nolimits_{j\epsilon N(i)}\alpha_{ij}^m W^m x_j^{(k)}\right)$$

   dove *m* rappresenta il numero di **attention heads** e $x_i^{(k+1)} \in R^{mf_{k+1}}$.

4.3.2 Implementazione modello

Il codice del modello di Geometric Deep Learning utilizzato all'interno del presente lavoro di tesi per accelerare la risoluzione del problema della **determinazione delle clique di dimensione massima di un grafo** è quello mostrato nella figura sottostante:

```
class GAT(nn.Module):

  #INIT
  def __init__(self, h_sizes, out_size, inOutDim,head,layerLin,linear):
    super(GAT, self).__init__()

    torch.manual_seed(0)
    #Hidden Layers GAT
    self.hidden = nn.ModuleList()
    if h_sizes>1:
      for k in range(h_sizes-1):
          self.hidden.append(GATConv(inOutDim[k]*head[k-1] if k!=0 else inOutDim[k] ,inOutDim[k+1],heads=head[k]))
      # LAST LAYER GAT
      self.hidden.append(GATConv(inOutDim[-1]*head[-2], linear[0],heads=1,concat=False))
    else :
      self.hidden.append(GATConv(inOutDim[0]*head[0], linear[0],heads=1,concat=False))

    #FULLY-CONNECTED (LINEAR) LAYERS
    self.fc = nn.ModuleList()
    for k in range(0,layerLin):
        self.fc.append(nn.Linear(linear[k], 1 if k+2>len(linear) else linear[k+1]))

  #FORWARD
  def forward(self, data,device):
    x, edge_index= data.x, data.edge_index
    for layer in self.hidden:
      x = layer(x,edge_index)
      x= F.elu(x)

    for layer in self.fc:
      x=layer(x)
      #x=F.relu(x)

    x=torch.sigmoid(x)

    x = x.view(x.size(0))
    return x
```

Figura 4.3.2.1: Codice del Modello utilizzato all'interno della tesi.

Come si vede dalla figura sovrastante, il codice si compone di due parti:

1. Nella prima parte è presente il metodo **init**, il quale, in qualche modo, può essere paragonato ad un costruttore che viene invocato ogni qualvolta viene creata un'istanza della classe denominata **GAT** e dunque del modello. All'interno del metodo **init** abbiamo specificato la struttura del modello e, in particolare, i **Layer** di cui si compone. Come si vede dalla figura sovrastante il modello si compone di un certo numero di **Layer GAT** e di un certo numero di **Layer Fully-Connected**.

   Diciamo "*un certo numero*" in quanto il codice è stato scritto in maniera tale che, in fase di istanziazione del modello, sia possibile specificare di quanti Layer GAT e Fully-Connected il modello deve essere costituito.

All’interno del modello ogni singolo **Layer GAT** è stato aggiunto creando un’istanza della classe **GATConv**, la quale viene messa a disposizione dalla libreria **Pytorch Geometric** e offre un costruttore che richiede i seguenti parametri di ingresso:

- Il primo parametro consente di specificare il numero di features di ogni singolo nodo che viene mandato in ingresso al Layer GAT.
- Il secondo parametro consente di specificare il numero di features dell’uscita prodotta dal Layer GAT.
- Il terzo parametro consente di specificare il numero di **attention heads** (di cui abbiamo parlato nel paragrafo precedente).

All’interno del modello ogni singolo **Layer Fully-Connected** è stato aggiunto creando un’istanza della classe **Linear** che offre un costruttore richiedente i seguenti parametri di ingresso:

- Il primo parametro consente di specificare il numero di features dell’ingresso che il Layer Fully-Connected riceve.
- Il secondo parametro consente di specificare il numero di features dell’uscita che il Layer Fully-Connected produce.

2. Nella seconda parte è presente il metodo **forward**, il quale viene invocato automaticamente ogni qualvolta al modello viene fornito un ingresso al fine di generare un’uscita. Nel momento in cui tale metodo viene invocato l’ingresso, per prima cosa, attraversa un certo numero (specificato in fase di istanziazione del modello) di **Layer GAT** e di funzioni di attivazione **elu**. Dopodiché, ciò che viene prodotto in uscita dall’ultima funzione di attivazione **elu** attraversa un certo numero (specificato in fase di istanziazione del modello) di **Layer Fully-Connected** e, infine, ciò che viene prodotto in uscita dall’ultimo **Layer Fully-Connected** viene mandato in ingresso ad una funzione di attivazione **sigmoidea** in maniera tale che possa essere generata l’uscita del modello.

Come si vede dalla figura sovrastante lo step finale attraverso il quale il modello riesce a generare l’uscita prevede l’utilizzo di una **funzione di attivazione sigmoidea**.

Per quale motivo abbiamo utilizzato proprio questa funzione di attivazione?

Abbiamo deciso di utilizzare tale funzione di attivazione in quanto essa produce dei numeri compresi tra 0 e 1 che possono essere interpretati in termini probabilistici.

Dunque, grazie al fatto che stiamo utilizzando proprio questa funzione di attivazione è possibile affermare che il modello, in uscita, per ogni singolo nodo del grafo che riceve in ingresso, produce una **probabilità** e, in particolare, la **probabilità che il nodo appartenga ad una clique di dimensione massima**.

La probabilità prodotta in uscita dal modello viene utilizzata durante la **fase di testing** (che approfondiremo) per attuare un'operazione di **node pruning**.

Una volta conclusa la panoramica sul codice, andiamo a presentare qual è il modello effettivo che abbiamo utilizzato.

Il modello di Geometric Deep Learning utilizzato all'interno del presente lavoro di tesi al fine di accelerare mediante **node pruning** la risoluzione del problema di tipo **NP-Hard: determinazione delle clique di dimensione massima di un grafo** è il seguente:

Numero Livelli Gat	Numero Livelli Lineari	Head GAT Layers	Input Features Linear Layers
3	1	[4, 1, 1]	[20]

Figura 4.3.2.2: Modello utilizzato all'interno della tesi.

Esso prevede un numero di **Layer GAT** pari a tre e un numero di **Layer Fully-Connected** pari a uno.

Inoltre, il primo Layer GAT è caratterizzato da un numero di **attention heads** pari a quattro, il secondo è caratterizzato da un numero di **attention heads** pari a uno e il terzo è caratterizzato da un numero di **attention heads** pari a uno.

Infine, il numero di features che l'unico **Layer Fully-Connected** riceve in ingresso è pari a venti. Queste sono le caratteristiche del modello migliore che siamo riusciti a trovare.

Il modello appena descritto è stato allenato in maniera **supervisionata** per un numero di **epoche** pari a **250** utilizzando un **learning rate** pari a $\mathbf{5 * 10^{-4}}$ e l'algoritmo **Stochastic Gradient Descent**.

4.4 Approccio implementativo

All'interno di questo paragrafo descriveremo il flusso logico seguito nel presente lavoro di tesi per realizzare le fasi di **training**, **validation** e **testing** del modello.

4.4.1 Fasi preliminari

In una fase preliminare alle fasi di **training**, **validation** e **testing** del modello sono state attuate le seguenti fasi:

1. **Generazione grafi**: all'interno di questa fase, a partire dai vari file aventi estensione **.ncol**, sono stati generati vari grafi, alcuni da utilizzare per il training, altri da utilizzare per la validation e altri ancora da utilizzare per il testing. Dopo aver creato i vari grafi, sempre all'interno di questa fase, sono state generate anche le quattro features (evidenziate nei paragrafi precedenti) di ogni singolo nodo di ogni grafo appena generato. Infine, allo scopo di generare il target (0 se il **nodo** non appartiene ad una clique di dimensione massima, 1 se vi appartiene) corrispondente ad ogni singolo **nodo** di ogni grafo appena generato è stato utilizzato un Solver in grado di restituire tutte le clique di dimensione massima del grafo a cui è stato applicato.

   Facendo uso delle clique di dimensione massima restituite dal Solver sono stati ricavati i nodi facenti parte di tali clique e quelli non facenti parte di tali clique.

   Ai primi è stato associato un target pari a 1, mentre ai secondi è stato associato un target pari a 0.

   Questa procedura è stata attuata in una fase preliminare al **training**, alla **validation** e al **testing** del modello.
2. **Creazione maschere**: dal momento che per il **testing** non è stato necessario, la creazione della maschere è stata attuata solo ed esclusivamente in una fase antecedente al **training** e alla **validation** del modello. All'interno di questa fase, per ogni singolo grafo da utilizzare per il training e per la validation, è stata generata una maschera contenente un certo numero di valori pari a 0 o 1. In particolare, il numero di valori pari a 1 è tale per cui la metà di questi si riferiscono a dei nodi aventi target pari a 0 e l'altra metà si riferiscono a dei nodi aventi target pari a 1.

   Le maschere create in questa fase sono state utilizzate durante la fase di **training** e **validation** per bilanciare ogni singolo vettore delle **uscite** prodotto dal modello e ogni singolo vettore dei **targets** in maniera tale da calcolare la **Loss** e l'**Accuracy** utilizzando i vettori delle uscite e dei targets **bilanciati**.

   Dal momento che, di base, i grafi da utilizzare per il training e per la validation sono sbilanciati, ossia sono costituiti da un numero di nodi aventi target pari a 0 che è molto maggiore del numero di nodi aventi target pari a 1, se non mascherassimo ogni singolo vettore delle uscite e ogni singolo vettore dei targets, la Loss e l'Accuracy non sarebbero attendibili in quanto inficiate dal fatto che i grafi sono sbilanciati. Ecco perché è stato necessario attuare un'operazione di **mascheramento**.

3. **Generazione dataset**: all'interno di questa fase per ogni singolo grafo da utilizzare per il **training**, **validation** e **testing** sono state generate le matrici: *x* (contenente le features di ogni singolo nodo del grafo), *edge_index* (contenente la struttura del grafo), il vettore *y* (contenente il target (0 o 1) relativo ad ogni singolo nodo del grafo) e la lista *id_nodi* (contenente l'identificativo di ogni singolo nodo del grafo).

4.4.2 Training e Validation

La fase di **training** è stata eseguita per un numero di epoche pari a 250, in ognuna delle quali, per un numero di volte pari al numero di grafi appartenenti al **training set**, sono state attuate le seguenti operazioni:

1. Innanzitutto, in ingresso al modello è stato fornito un oggetto contenente le matrici *x*, *edge_index*, il vettore *y* e la lista *id_nodi* relativi ad un certo grafo del **training set**.
2. Dopodiché, è stata prelevata l'uscita prodotta dal modello, la quale è un vettore contenente tanti elementi quanti sono i **nodi** del grafo che il modello ha ricevuto in ingresso, dove ogni singolo elemento altro non è che la probabilità che quel nodo appartenga ad una clique di dimensione massima.
3. Sia all'uscita generata dal modello sia al vettore dei targets *y* è stata applicata una **maschera** in maniera tale da renderli bilanciati.
4. Utilizzando l'uscita bilanciata e il vettore dei targets *y* bilanciato è stata calcolata la **Loss** e l'**Accuracy**.
5. Dopo aver fatto ciò, è stato calcolato il gradiente della **Loss** rispetto a tutti i parametri del modello e, infine, applicando l'algoritmo **Stochastic Gradient Descent**, sono stati aggiornati i parametri del modello in maniera tale da minimizzare sempre di più la **Loss Function**.

La fase di **validation** è stata eseguita anch'essa per un numero di epoche pari a 250, in ognuna delle quali, per un numero di volte pari al numero di grafi appartenenti al **validation set**, sono state attuate le stesse identiche operazioni viste per la fase di **training**, escludendo le operazioni attuate al passo numero 5.

4.4.3 Testing

In una fase preliminare alla fase di **testing** ad ogni singolo grafo appartenente al **testing set** è stato applicato il Solver in maniera tale da ricavare il tempo richiesto dal Solver per determinare tutte le clique di dimensione massima del grafo e tali clique.

È stato necessario ricavare queste due informazioni in maniera tale da poter valutare, al termine del processo di **node pruning**, quanto accurato è stato tale processo e lo speedup ottenuto attuando l'operazione di **node pruning**.

Dopo aver fatto ciò, è stata attuata una fase di **testing** al variare della soglia e una fase di **testing** fissata una soglia specifica.

È opportuno sottolineare che la soglia è lo strumento attraverso il quale è possibile discriminare i nodi che il modello predice come appartenenti ad una clique di dimensione massima dai nodi che il modello predice come non appartenenti ad una clique di dimensione massima.

TESTING AL VARIARE DELLA SOGLIA

Prima di procedere con tale fase, è stato creato un oggetto **thresholds** contenente le 21 soglie da utilizzare per attuare le varie operazioni di **node pruning**.

Dopodiché, durante la fase di **testing al variare della soglia** sono state attuate le seguenti operazioni tante volte quanti sono i grafi appartenenti al **testing set**:

1. Innanzitutto, in ingresso al modello è stato fornito un oggetto contenente le matrici *x, edge_index,* il vettore *y* e la lista *id_nodi* relativi ad un certo grafo *x* del **testing set**.
2. Dopodiché, è stata prelevata l'uscita prodotta dal modello, la quale è un vettore contenente tanti elementi quanti sono i **nodi** del grafo che il modello ha ricevuto in ingresso, dove ogni singolo elemento altro non è che la probabilità che quel nodo appartenga ad una clique di dimensione massima.
3. A questo punto, sono state attuate le seguenti operazioni tante volte quante sono le soglie contenute all'interno dell'oggetto **thresholds**:
    1. Al grafo *x* del **testing set** è stata applicata una funzione **pruning** che, utilizzando le predizioni generate dal modello e una certa **soglia** contenuta in **thresholds**, non fa altro che eliminare dal grafo originario tutti i nodi che, secondo quanto predetto dal modello, non appartengono ad alcuna clique di dimensione massima.

2. Al grafo restituito dalla funzione **pruning**, ossia al grafo ottenuto a seguito dell'attuazione dell'operazione di **pruning**, è stato applicato il Solver in maniera tale da ricavare il tempo necessario affinché il Solver possa ricavare tutte le clique di dimensione massima di tale grafo e tali clique.
3. Il tempo necessario affinché il Solver possa ricavare tutte le clique di dimensione massima del grafo originario *x* e il tempo necessario affinché il Solver possa ricavare tutte le clique di dimensione massima del grafo ottenuto a seguito del **pruning** sono stati utilizzati per calcolare lo **speedup**, mentre le clique di dimensione massima ottenute applicando il Solver al grafo originario *x* e le clique di dimensione massima ottenute applicando il Solver al grafo ottenuto al seguito del **pruning** sono state utilizzate per calcolare la **similarity**, la quale fornisce un'idea di quanto l'operazione di pruning è stata accurata.

TESTING FISSATA UNA SOGLIA SPECIFICA

Durante la fase di **testing fissata una soglia specifica** sono state attuate le seguenti operazioni tante volte quanti sono i grafi appartenenti al **testing set**:

1. Innanzitutto, in ingresso al modello è stato fornito un oggetto contenente le matrici *x*, *edge_index*, il vettore *y* e la lista *id_nodi* relativi ad un certo grafo *x* del **testing set**.
2. Dopodiché, è stata prelevata l'uscita prodotta dal modello, la quale è un vettore contenente tanti elementi quanti sono i **nodi** del grafo che il modello ha ricevuto in ingresso, dove ogni singolo elemento altro non è che la probabilità che quel nodo appartenga ad una clique di dimensione massima.
3. Al grafo *x* del **testing set** è stata applicata una funzione **pruning** che, utilizzando le predizioni generate dal modello e la **soglia specifica**, non fa altro che eliminare dal grafo originario tutti i nodi che, secondo quanto predetto dal modello, non appartengono ad alcuna clique di dimensione massima.
4. Al grafo restituito dalla funzione **pruning**, ossia al grafo ottenuto a seguito dell'attuazione dell'operazione di **pruning**, è stato applicato il Solver in maniera tale da ricavare il tempo necessario affinché il Solver possa ricavare tutte le clique di dimensione massima di tale grafo e tali clique.

5. Il tempo necessario affinché il Solver possa ricavare tutte le clique di dimensione massima del grafo originario *x* e il tempo necessario affinché il Solver possa ricavare tutte le clique di dimensione massima del grafo ottenuto a seguito del **pruning** sono stati utilizzati per calcolare lo **speedup**, mentre le clique di dimensione massima ottenute applicando il Solver al grafo originario *x* e le clique di dimensione massima ottenute applicando il Solver al grafo ottenuto al seguito del **pruning** sono state utilizzate per calcolare la **similarity**, la quale fornisce un'idea di quanto l'operazione di pruning è stata accurata.
6. Dopodiché, l'uscita prodotta dal modello e il vettore dei targets *y* relativo al grafo *x* sono stati utilizzati per calcolare il **true positive rate**, il **false positive rate** e l'**auc score** (area sotto la curva ROC) relativi a tale grafo.
7. Infine, il numero di nodi presenti nel grafo originario *x* e il numero di nodi presenti nel grafo risultante dall'applicazione dell'operazione di node pruning sono stati utilizzati per calcolare il **node pruning rate** mentre il numero di archi presenti nel grafo originario *x* e il numero di archi presenti nel grafo risultante dall'applicazione dell'operazione di node pruning sono stati utilizzati per calcolare l'**edge pruning rate**.

Capitolo 5

Risultati e Conclusioni

5.1 Dataset utilizzato

Il dataset utilizzato all'interno del presente lavoro di tesi è composto da grafi provenienti dal mondo reale appartenenti ai seguenti domini:

- **Animal Social Network**
- **Biological Network**
- **Collaboration Network**
- **Economic Network**
- **Email Network**
- **Interaction Network**
- **Retweet Network**
- **Social Network**
- **Facebook Network**
- **Technological Network**
- **Web Network**

I grafi appartenenti ai domini appena citati possono essere reperiti da **NetworkRepository.com** [25].

5.1.1 Training set

I grafi appartenenti al **training set** che sono stati utilizzati per effettuare il **training** del modello sono quelli mostrati nella figura sottostante:

Grafo	|V|	|E|	n. ω	ω
bio-CE-GT	924	3 K	2	8
bio-CE-HT	2 K	2 K	1	4
bio-CE-LC	1 K	1 K	2	7
bio-DM-HT	2 K	4 K	2	4
bio-DM-LC	658	1 K	1	5

Grafo	|V|	|E|	n. ω	ω
bio-HS-HT	2 K	13 K	10	38
bio-SC-LC	2 K	20 K	68	29
bio-celegans	453	2 K	6	9
bio-celegans-dir	453	2 K	6	9
bio-diseasome	516	1 K	1	11
bio-dmela	7 K	25 K	1	7
bio-grid-fission-yeast	2 K	25 K	1	12
bio-grid-mouse	1 K	3 K	1	7
bio-grid-plant	1 K	6 K	9	9
bio-grid-worm	3 K	13 K	27	7
bio-yeast	1 K	1 K	1	6
bio-yeast-protein-inter	2 K	4 K	1	6
soc-dolphins	62	159	3	5
soc-firm-hi-tech	36	147	1	6
soc-hamsterster	2 K	16 K	1	25
soc-karate	34	78	2	5
soc-wiki-Vote	889	2 K	1	7
soc-tribes	16	58	3	5
soc-pages-food	620	2 K	3	10
soc-pages-tvshow	4 K	17 K	5	57
socfb-Reed98	962	18 K	6	16
web-EPA	4 K	8 K	47	4
web-edu	3 K	6 K	1	30
web-polblogs	643	2 K	22	9
web-webbase-2001	16 K	25 K	1	33
tech-pgp	11 K	24 K	12	25
tech-routers-rf	2 K	7 K	5	16
rt_uae	5 K	6 K	4	4
rt_tlot	4 K	4 K	1	4
rt_oman	5 K	6 K	6	4
rt_lebanon	4 K	4 K	1	4
rt_damascus	3 K	4 K	11	4
rt_bahrain	5 K	8 K	7	5

Grafo	|V|	|E|	n. ω	ω
rt_assad	2 K	3 K	1	5
rt_alwefaq	4 K	7 K	26	8
ia-infect-dublin	410	3 K	1	16
ia-email-univ	1 K	5 K	1	12
ia-fb-messages	1 K	6 K	8	5
ia-reality	7 K	8 K	5	5
ia-email-EU	32 K	54 K	24	12
email-dnc-corecipient	906	16 K	1	75
email-univ	1 K	5 K	1	12
econ-mahindas	1 K	8 K	106	4
econ-poli-large	16 K	17 K	34	4
econ-poli	4 K	4 K	7	4
ca-Erdos992	6 K	8 K	1	8
ca-GrQc	5 K	14 K	1	44
aves-weaver-social	445	1 K	1	12
aves-thornbill-farine	62	1 K	7	27
aves-wildbird-network-3	126	2 K	2	25
mammalia-voles-rob-trapping	1 K	5 K	2	8
mammalia-voles-bhp-trapping	2 K	5 K	3	8
mhd3200b	3 K	8 K	199	6
sstmodel	3 K	10 K	2387	4
tols4000	4 K	6 K	72	19
as20000102	6 K	13 K	17	10
gene	1 K	2 K	1	13
DD242	1 K	3 K	22	5
cora	3 K	5 K	9	5

Tabella 5.1.1.1: Training set. La colonna **|V|** contiene il numero di nodi di ciascun grafo, la colonna **|E|** contiene il numero di archi di ciascun grafo, la colonna **n. ω** contiene il numero di clique di dimensione massima di ciascun grafo e la colonna **ω** contiene la dimensione di ogni singola clique di dimensione massima di ogni grafo.

5.1.2 Validation set

I grafi appartenenti al **validation set** che sono stati utilizzati per effettuare la fase di **validation** del modello sono quelli mostrati nella figura sottostante:

Grafo	|V|	|E|	n. ω	ω
ca-AstroPh	19 K	198 K	1	57
ca-HepPh	12 K	118 K	1	239
rec-amazon	92 K	126 K	294	5
socfb-nips-ego	3 K	3 K	4	4
soc-pages-media	28 K	206 K	3	31
soc-pages-sport	14 K	87 K	15	29
tech-internet-as	40 K	85 K	12	16
tech-p2p-gnutella	63 K	148 K	16	4
web-frwikinews-user-edits	25 K	194 K	5	7

Tabella 5.1.2.1: Validation set. La colonna **|V|** contiene il numero di nodi di ciascun grafo, la colonna **|E|** contiene il numero di archi di ciascun grafo, la colonna **n. ω** contiene il numero di clique di dimensione massima di ciascun grafo e la colonna **ω** contiene la dimensione di ogni singola clique di dimensione massima di ogni grafo.

5.1.3 Testing set

I grafi appartenenti al **testing set** che sono stati utilizzati per effettuare il **testing** del modello sono quelli mostrati nella figura sottostante:

Grafo	|V|	|E|	n. ω	ω
ca-citeseer	227 K	814 K	1	87
ca-dblp-2010	226 K	716 K	1	75
ca-dblp-2012	317 K	1 M	1	114
rt-retweet-crawl	1 M	2 M	26	13
rt-higgs	425 K	734 K	7	12
soc-delicious	536 K	1 M	9	21
soc-sign-Slashdot081106	77 K	517 K	52	26

Grafo	|V|	|E|	n. ω	ω
soc-themarker	69 K	1 M	40	22
web-baidu-baike-related	416 K	3 M	2	95
socfb-OR	63 K	817 K	2	30
socfb-wosn-friends	64 K	1 M	2	30
tech-RL-caida	191 K	608 K	31	17

Tabella 5.1.3.1: Testing set. La colonna **|V|** contiene il numero di nodi di ciascun grafo, la colonna **|E|** contiene il numero di archi di ciascun grafo, la colonna **n. ω** contiene il numero di clique di dimensione massima di ciascun grafo e la colonna **ω** contiene la dimensione di ogni singola clique di dimensione massima di ogni grafo.

5.2 Metriche di valutazione delle performance

Le **metriche** utilizzate all'interno di questa tesi al fine di valutare le **performance** raggiunte dal modello in fase di **testing** sono le seguenti:

- **Similarity**: essa consente di determinare quanto l'operazione di **node pruning** è accurata dal momento che esprime il grado di somiglianza tra le clique di dimensione massima determinate a partire dal grafo originario e quelle determinate a partire dal grafo risultante dall'operazione di **node pruning** applicata al grafo originario.

  Supponiamo di indicare con *x* il grafo originario e con *y* il grafo risultante dall'operazione di node pruning applicata al grafo originario *x*.

  Fatta questa premessa, possiamo affermare che la **similarity** viene calcolata così:

  a) **Se** la dimensione di ogni singola clique di dimensione massima ricavata a partire dal grafo *y* **è inferiore** alla dimensione di ogni singola clique di dimensione massima ricavata a partire dal grafo *x* allora la **similarity** è pari allo 0% in quanto per lo scopo che la tesi vuole raggiungere il fatto che l'operazione di **node pruning** comporti una riduzione della dimensione di ogni singola clique di dimensione massima è un qualcosa di estremamente negativo.

b) **Se** la dimensione di ogni singola clique di dimensione massima ricavata a partire dal grafo *y* **è uguale** alla dimensione di ogni singola clique di dimensione massima ricavata a partire dal grafo *x* allora la **similarity** viene calcolata applicando la seguente formula:

$$Similarity = \frac{n_{cap}}{n_{cbp}} * 100$$

dove n_{cbp} indica il numero di clique di dimensione massima del grafo *x* e n_{cap} indica il numero di clique di dimensione massima del grafo *y*.

Ovviamente, la situazione ideale si ha quando la **similarity** è pari al 100% in quanto, in tal caso, vorrebbe dire che l'operazione di **node pruning** non solo ha comportato un miglioramento dei tempi necessari affinché il Solver possa determinare tutte le clique di dimensione massima del grafo ma, inoltre, è stata perfettamente accurata in quanto ha lasciato totalmente inalterati sia il numero di clique di dimensione massima sia la dimensione di ognuna di essa.

- **Speedup**: esso indica il miglioramento ottenuto in termini di riduzione del tempo necessario affinché il Solver possa determinare tutte le clique di dimensione massima del grafo in seguito all'applicazione dell'operazione di **node pruning**.

  Supponiamo di indicare con *x* il grafo originario e con *y* il grafo risultante dall'operazione di node pruning applicata al grafo originario *x*.

  Se, ad esempio, lo **speedup** è pari a 30 significa che in seguito all'attuazione dell'operazione di node pruning, il Solver riesce a determinare le clique di dimensione massima del grafo *y*, impiegando un tempo che è 30 volte più piccolo rispetto al tempo che sarebbe necessario se le clique di dimensione massima venissero determinate a partire dal grafo originario *x*.

  Se con t_{bp} indichiamo il tempo necessario affinché il Solver possa determinare tutte le clique di dimensione massima del grafo originario *x* e con t_{ap} indichiamo il tempo necessario affinché il Solver possa determinare tutte le clique di dimensione massima del grafo *y* allora lo **speedup** può essere calcolato mediante la formula seguente:

$$Speedup = \frac{t_{bp}}{t_{ap}}$$

- **Node Pruning Rate**: esso fornisce un'indicazione sul numero di nodi che vengono eliminati dal grafo originario a seguito dell'attuazione dell'operazione di **node pruning**.

  Supponiamo di indicare con *x* il grafo originario e con *y* il grafo risultante dall'operazione di node pruning applicata al grafo originario *x*.

Se con x_{vcount} indichiamo il numero di nodi del grafo originario *x* e con y_{vcount} indichiamo il numero di nodi del grafo *y* allora il **node pruning rate** può essere calcolato tramite la formula sottostante:

$$Node\ Pruning\ Rate = \frac{x_{vcount} - y_{vcount}}{x_{vcount}} * 100$$

Se, ad esempio, il Node Pruning Rate è pari al 50% significa che in seguito all'attuazione dell'operazione di **node pruning** il grafo *y* ha un numero di nodi che è la metà rispetto al numero di nodi del grafo originario *x*.

- **Edge Pruning Rate**: esso fornisce un'indicazione sul numero di archi che vengono eliminati dal grafo originario a seguito dell'attuazione dell'operazione di **node pruning**.

  Supponiamo di indicare con *x* il grafo originario e con *y* il grafo risultante dall'operazione di node pruning applicata al grafo originario *x*.

  Se con x_{ecount} indichiamo il numero di archi del grafo originario *x* e con y_{ecount} indichiamo il numero di archi del grafo *y* allora l'**edge pruning rate** può essere calcolato tramite la formula sottostante:

$$Edge\ Pruning\ Rate = \frac{x_{ecount} - y_{ecount}}{x_{ecount}} * 100$$

  Se, ad esempio, l'Edge Pruning Rate è pari al 50% significa che in seguito all'attuazione dell'operazione di **node pruning** il grafo *y* ha un numero di archi che è la metà rispetto al numero di archi del grafo originario *x*.

5.3 Risultati

All'interno di questo paragrafo verranno presentati i risultati che sono stati ottenuti al termine della fase di **testing** del modello e, in particolare, i risultati ottenuti facendo variare la soglia e quelli ottenuti fissando la soglia ad un valore specifico.

Infine, verranno mostrate le **Curve ROC** tracciate sempre al termine della fase di **testing** del modello.

5.3.1 Soglia variabile

All'interno di questo paragrafo, per ogni singolo grafo appartenente al **testing set**, verrà mostrato un grafico all'interno del quale sono stati tracciati, **al variare della soglia**, la **similarity** e lo **speedup**.

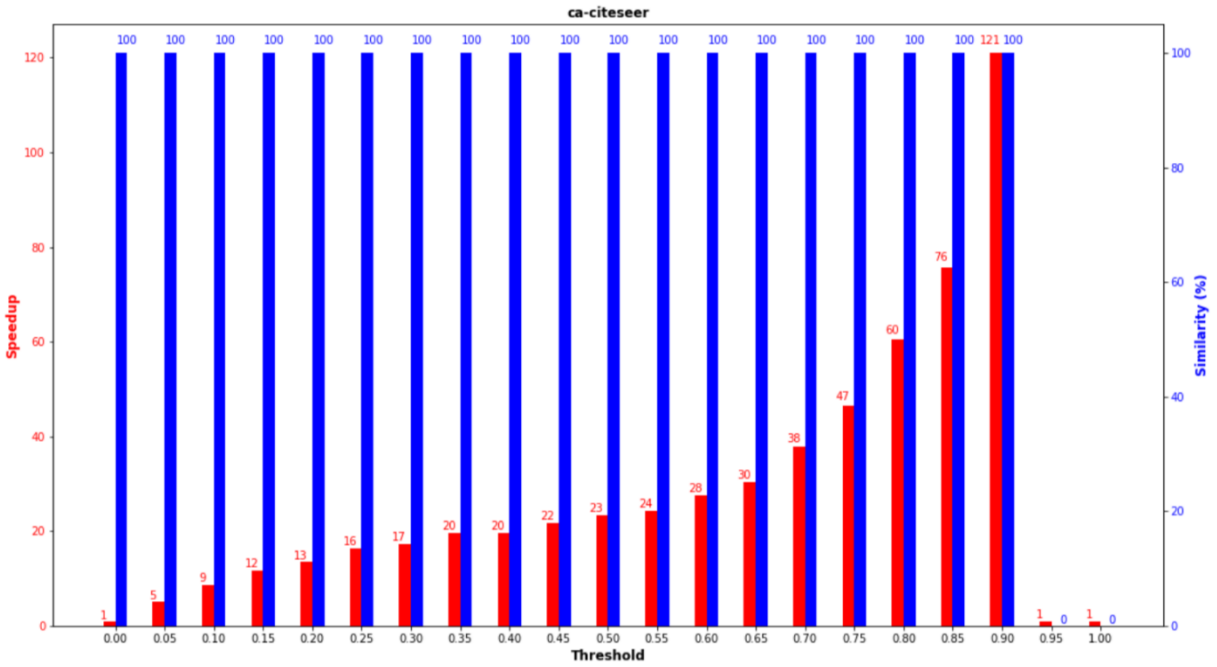


Figura 5.3.1.1: Similarity e Speedup al variare della soglia considerando il grafo ca-citeseer.

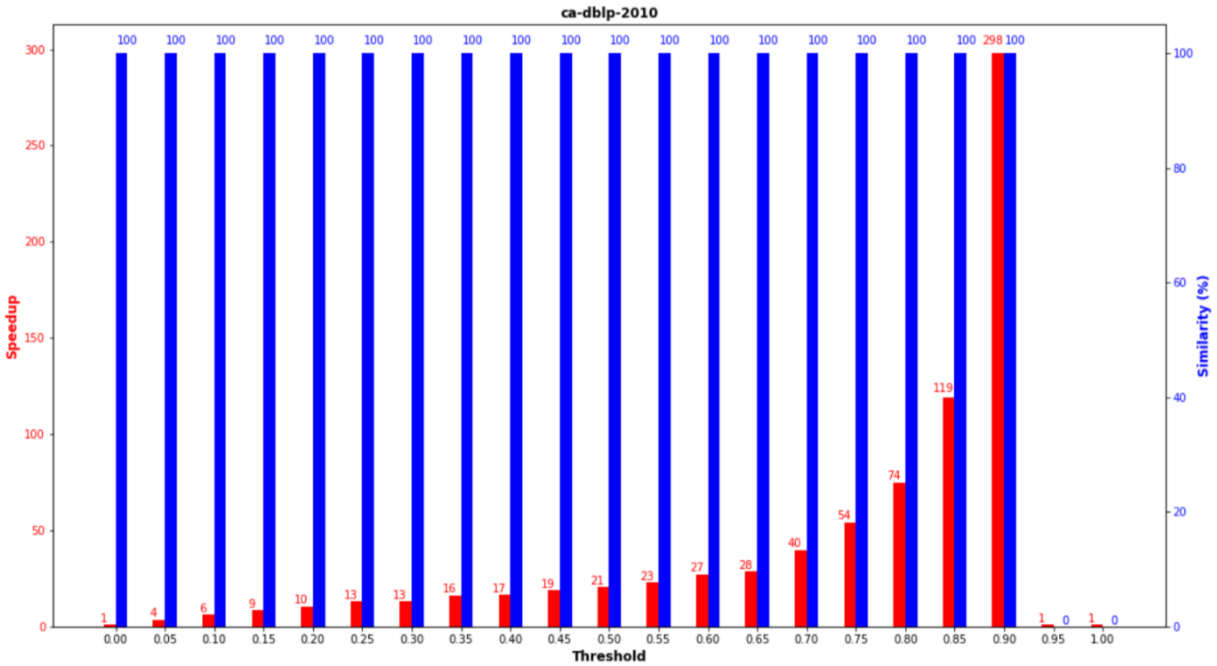


Figura 5.3.1.2: Similarity e Speedup al variare della soglia considerando il grafo ca-dblp-2010.

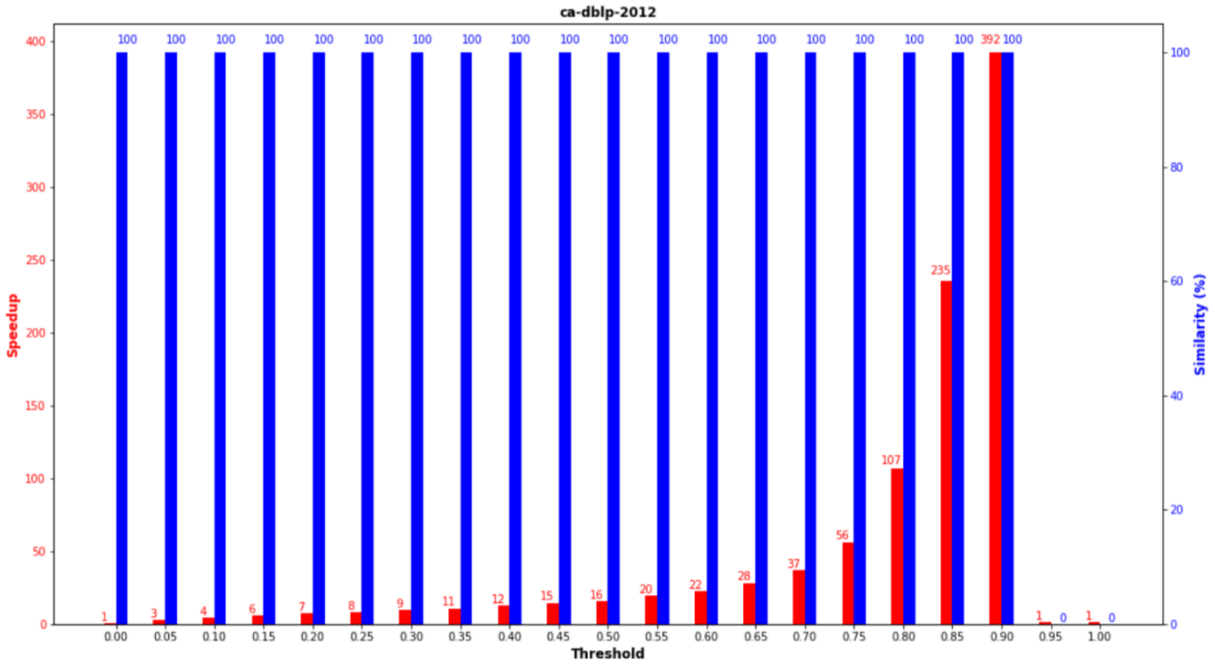


Figura 5.3.1.3: Similarity e Speedup al variare della soglia considerando il grafo ca-dblp-2012.

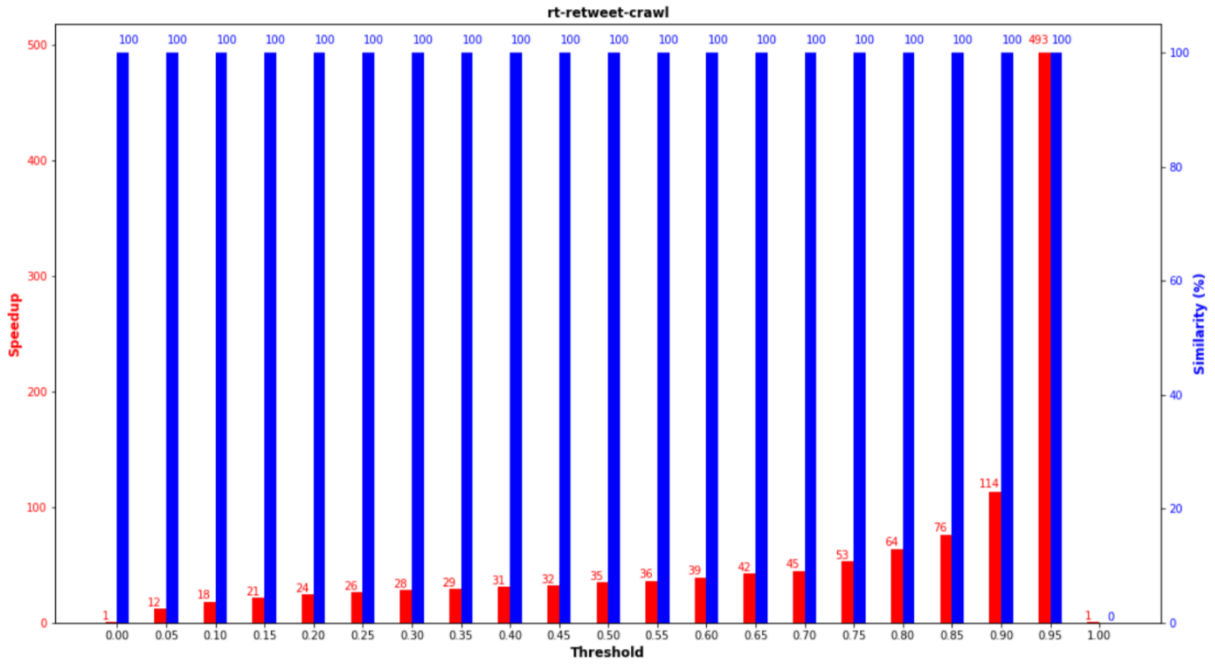


Figura 5.3.1.4: Similarity e Speedup al variare della soglia considerando il grafo rt-retweet-crawl.

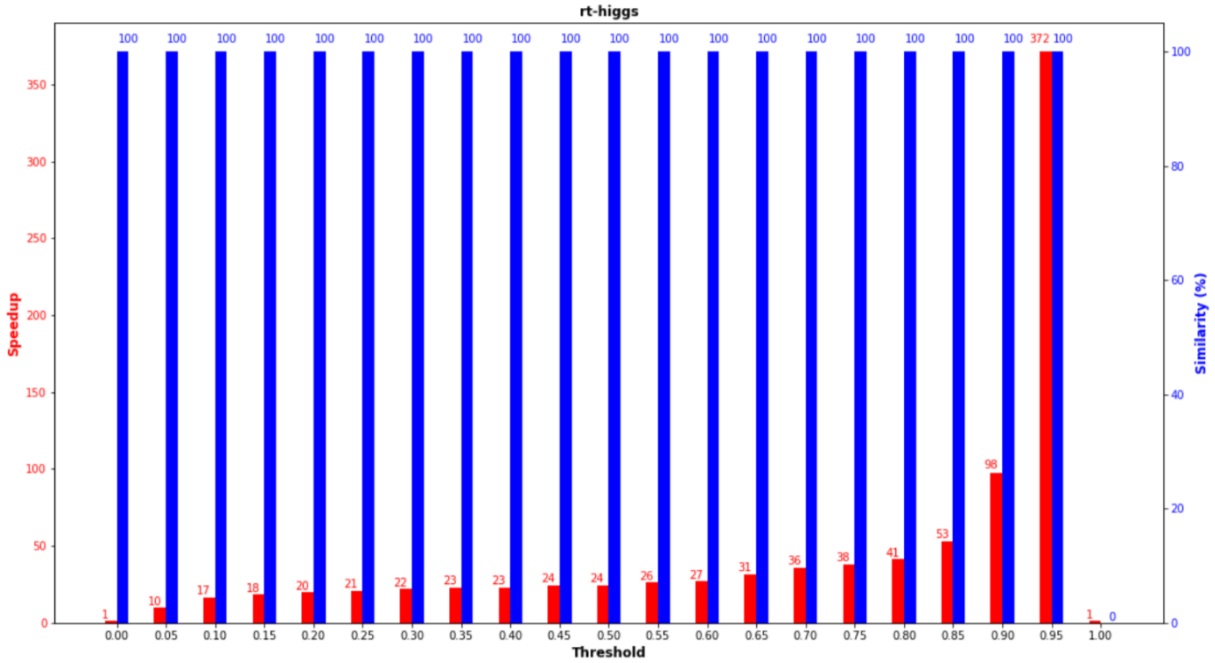


Figura 5.3.1.5: Similarity e Speedup al variare della soglia considerando il grafo rt-higgs.

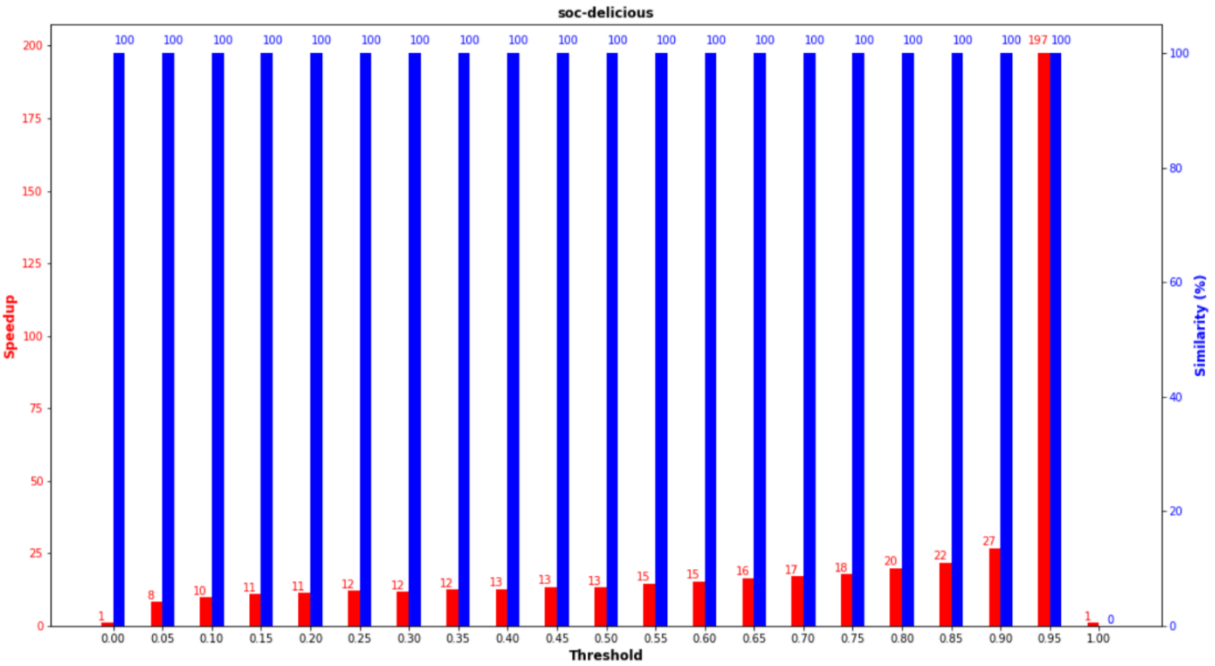


Figura 5.3.1.6: Similarity e Speedup al variare della soglia considerando il grafo soc-delicious.

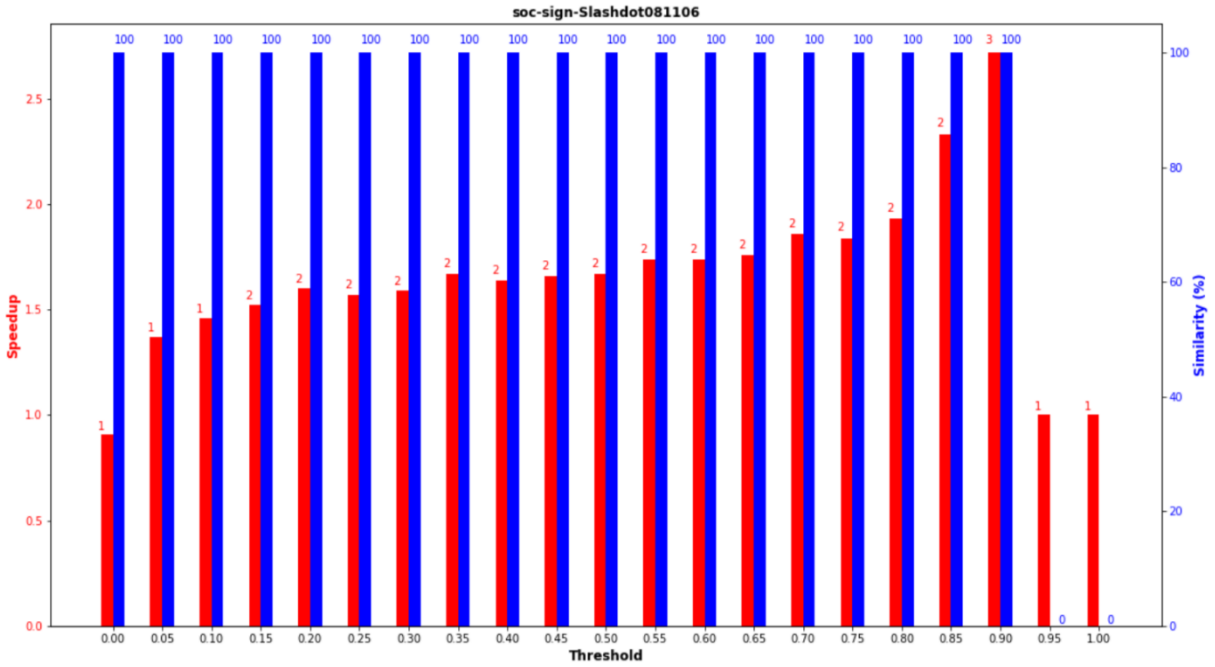


Figura 5.3.1.7: Similarity e Speedup al variare della soglia considerando il grafo soc-sign-Slashdot081106.

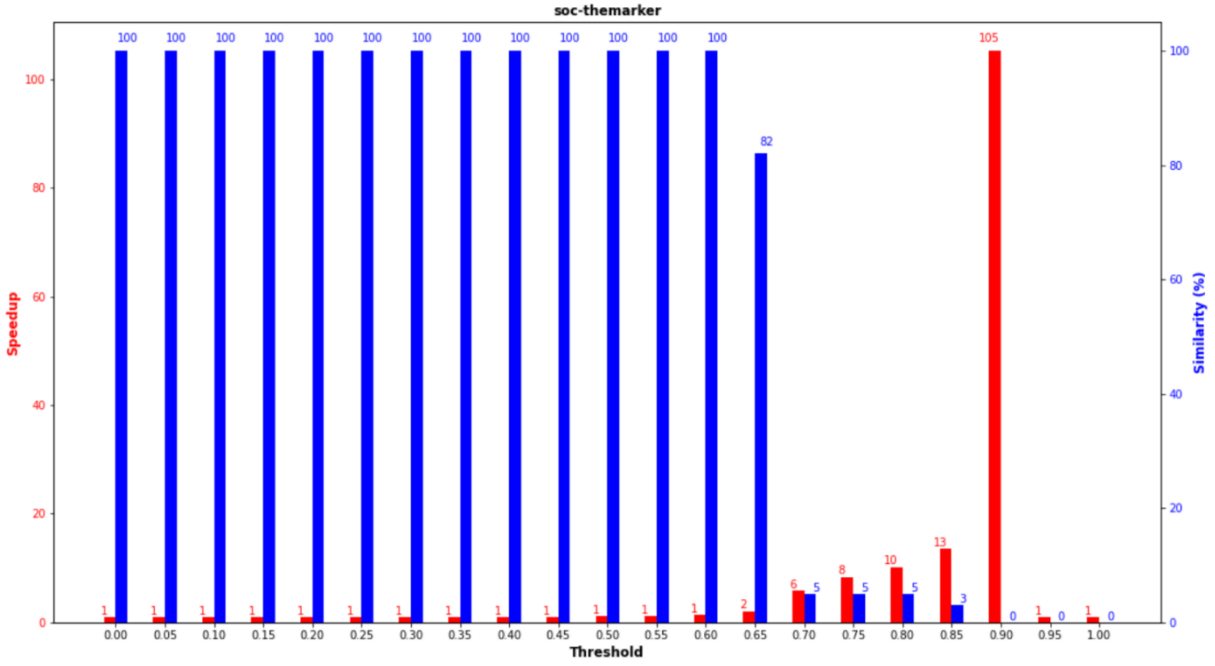


Figura 5.3.1.8: Similarity e Speedup al variare della soglia considerando il grafo soc-themarker.

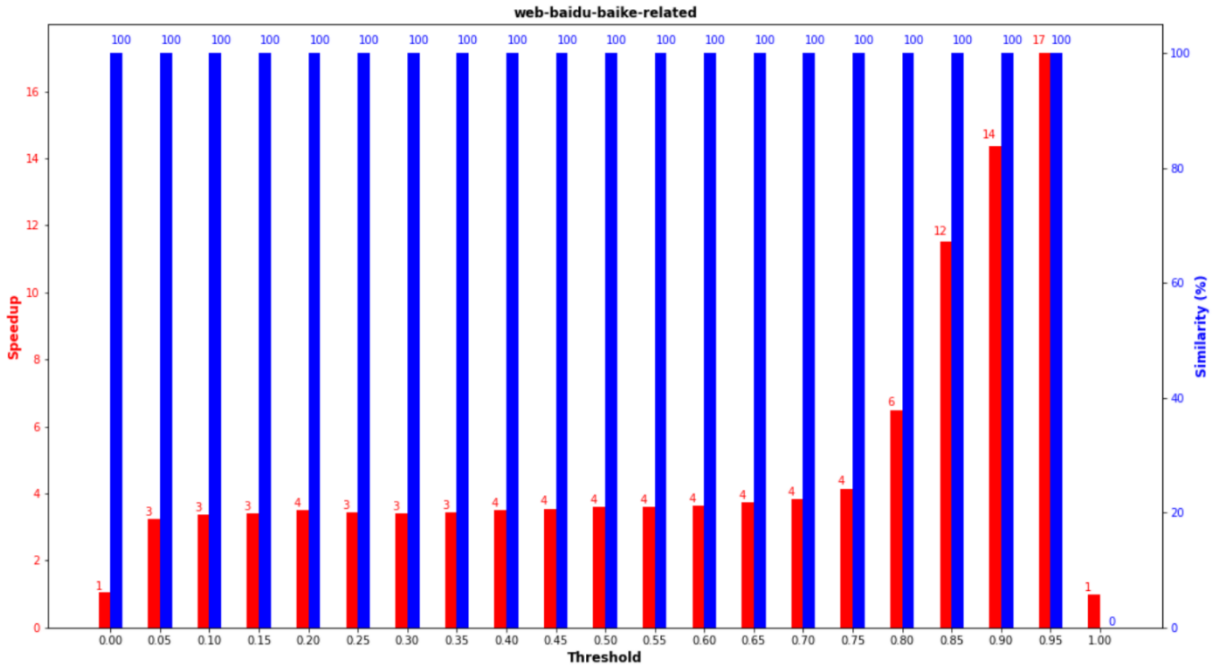


Figura 5.3.1.9: Similarity e Speedup al variare della soglia considerando il grafo web-baidu-baike-related.

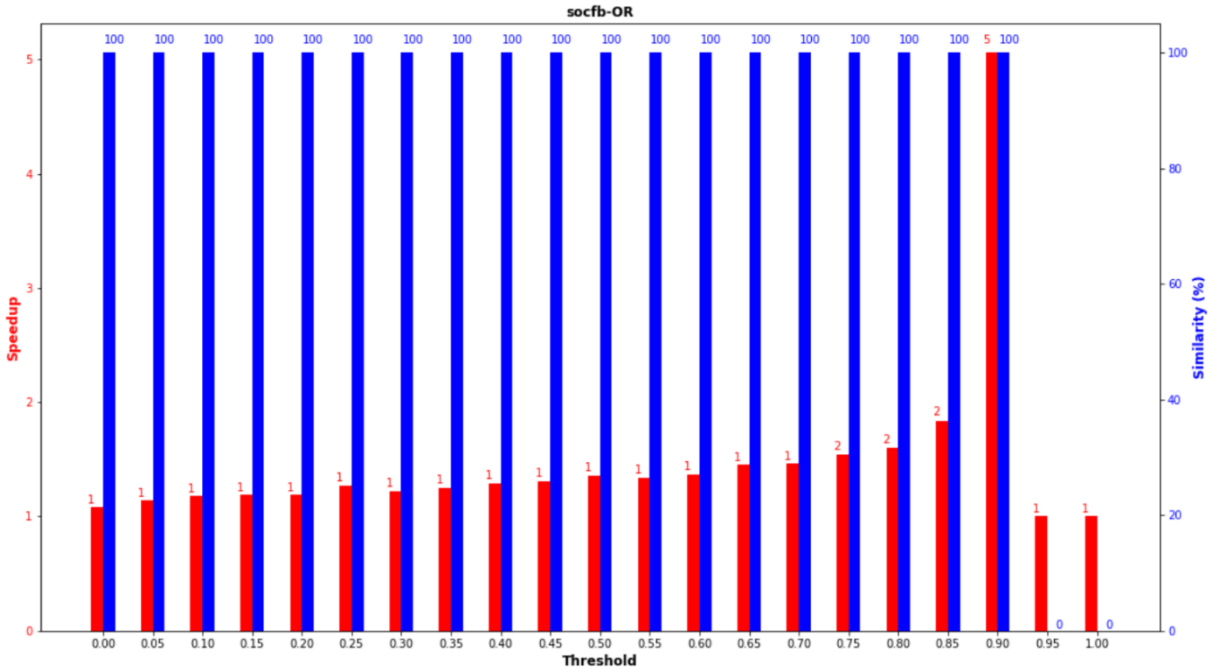


Figura 5.3.1.10: Similarity e Speedup al variare della soglia considerando il grafo socfb-OR.

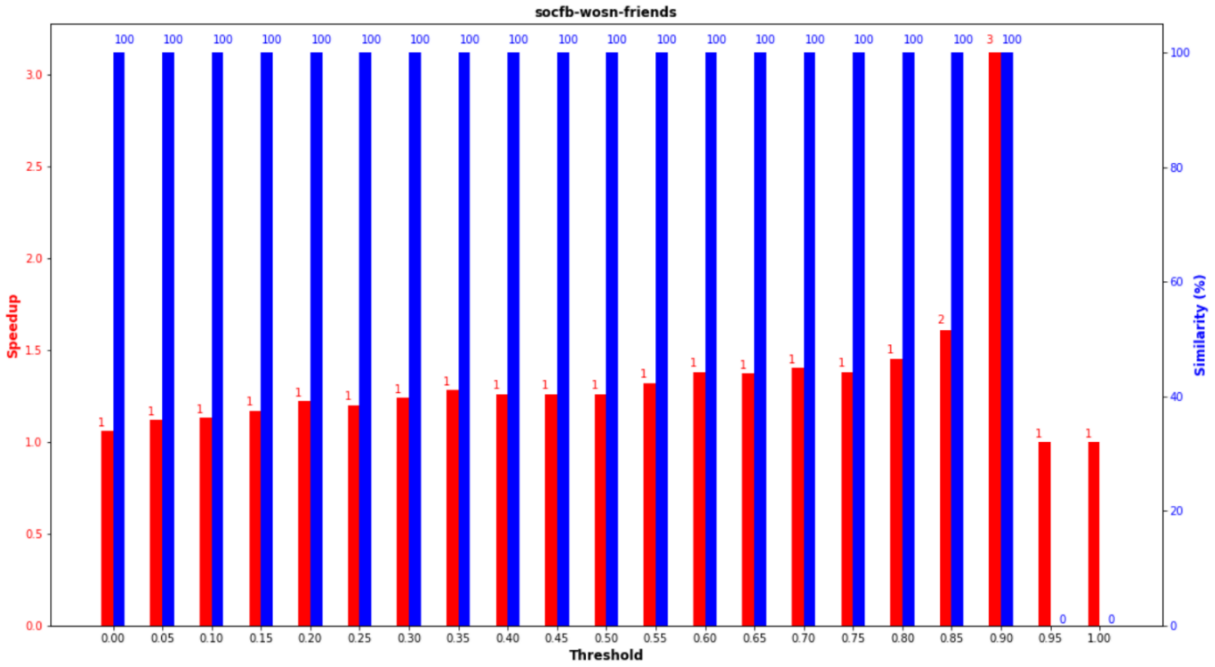


Figura 5.3.1.11: Similarity e Speedup al variare della soglia considerando il grafo socfb-wosn-friends.

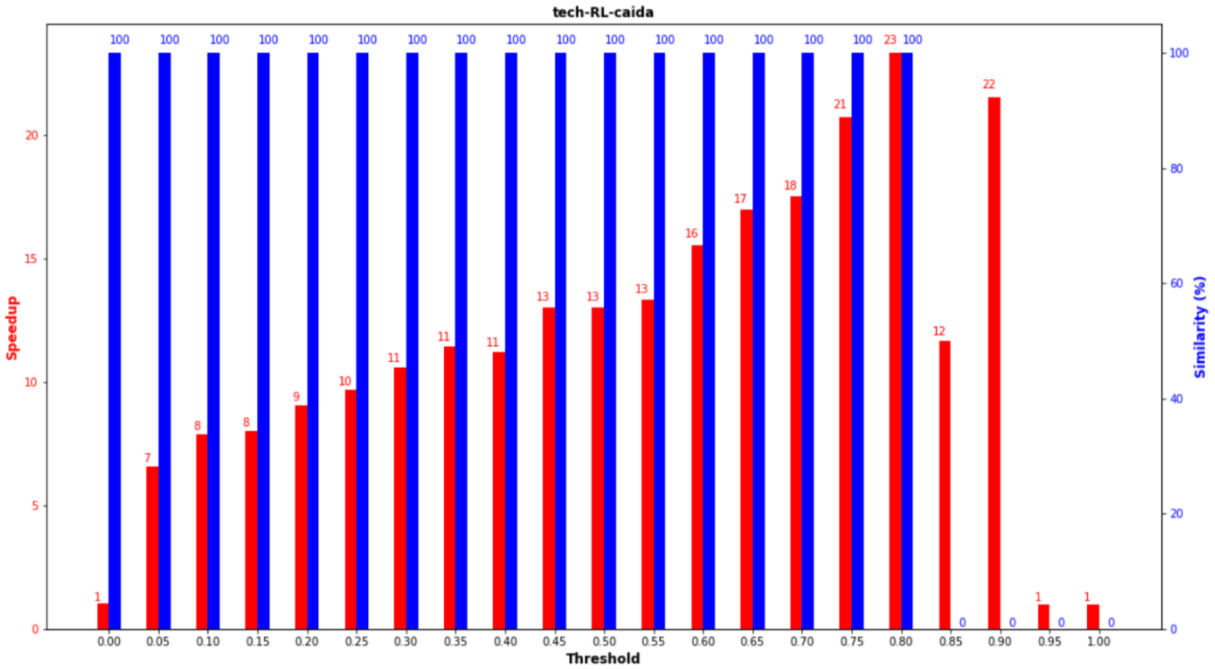


Figura 5.3.1.12: Similarity e Speedup al variare della soglia considerando il grafo tech-RL-caida.

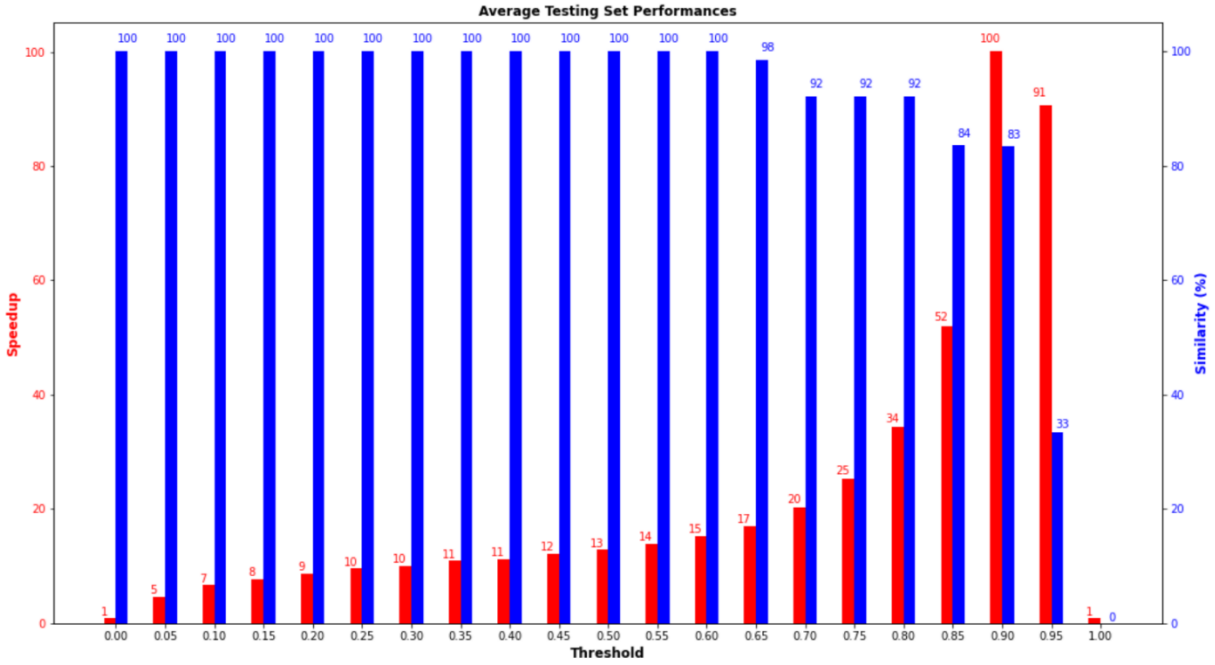


Figura 5.3.1.13: Similarity e Speedup medi al variare della soglia considerando l'intero testing set.

La figura sovrastante è estremamente importante in quanto consente di valutare le performance (Speedup e Similarity) medie del modello al variare della soglia, considerando l'intero testing set. Osservando la figura è evidente che all'aumentare della soglia aumenta lo Speedup medio e, al tempo stesso, si riduce la Similarity media. Il fatto che ciò accada è normale in quanto all'aumentare della soglia l'operazione di **node pruning** attuata sui vari grafi del testing set diventa sempre più aggressiva con la conseguenza che lo Speedup medio aumenta a causa del fatto che, via via che la soglia aumenta, il Solver viene applicato a grafi sempre più piccoli e che la Similarity media si riduce a causa del fatto che, man mano che la soglia aumenta, essendo l'operazione di **node pruning** sempre più aggressiva, c'è un rischio sempre maggiore che dai vari grafi appartenenti al testing set vengano eliminati nodi appartenenti ad una clique di dimensione massima.

Osservando il grafico sovrastante, in funzione del compromesso **Similarity-Speedup** che si vuole ottenere, è possibile fissare una **soglia specifica** in corrispondenza della quale è possibile calcolare le 4 metriche evidenziate nel paragrafo precedente.

Dal momento che la **soglia specifica** in corrispondenza della quale si ottiene una Similarity media pari al 100% e, contemporaneamente, uno Speedup medio massimo possibile è 0.6, il **testing fissata una soglia specifica** è stato effettuando utilizzando proprio 0.6 come soglia specifica. Ovviamente, se si è disposti ad accettare una riduzione della Similarity media, è possibile aumentare ulteriormente la soglia in maniera tale da sperimentare uno Speedup medio ancora superiore rispetto a quello pari a 15 ottenuto utilizzando 0.6 come soglia.

5.3.2 Soglia specifica

Come detto nel paragrafo precedente, dal momento che la **soglia specifica** in corrispondenza della quale si ottiene una Similarity media pari al 100% e, contemporaneamente, uno Speedup medio massimo possibile è pari a 0.6, il **testing fissata una soglia specifica** è stato effettuato utilizzando proprio tale soglia.

Nella figura sottostante è possibile osservare il valore delle 4 metriche di valutazione delle performance descritte nei paragrafi precedenti per ogni singolo grafo appartenente al **testing set**:

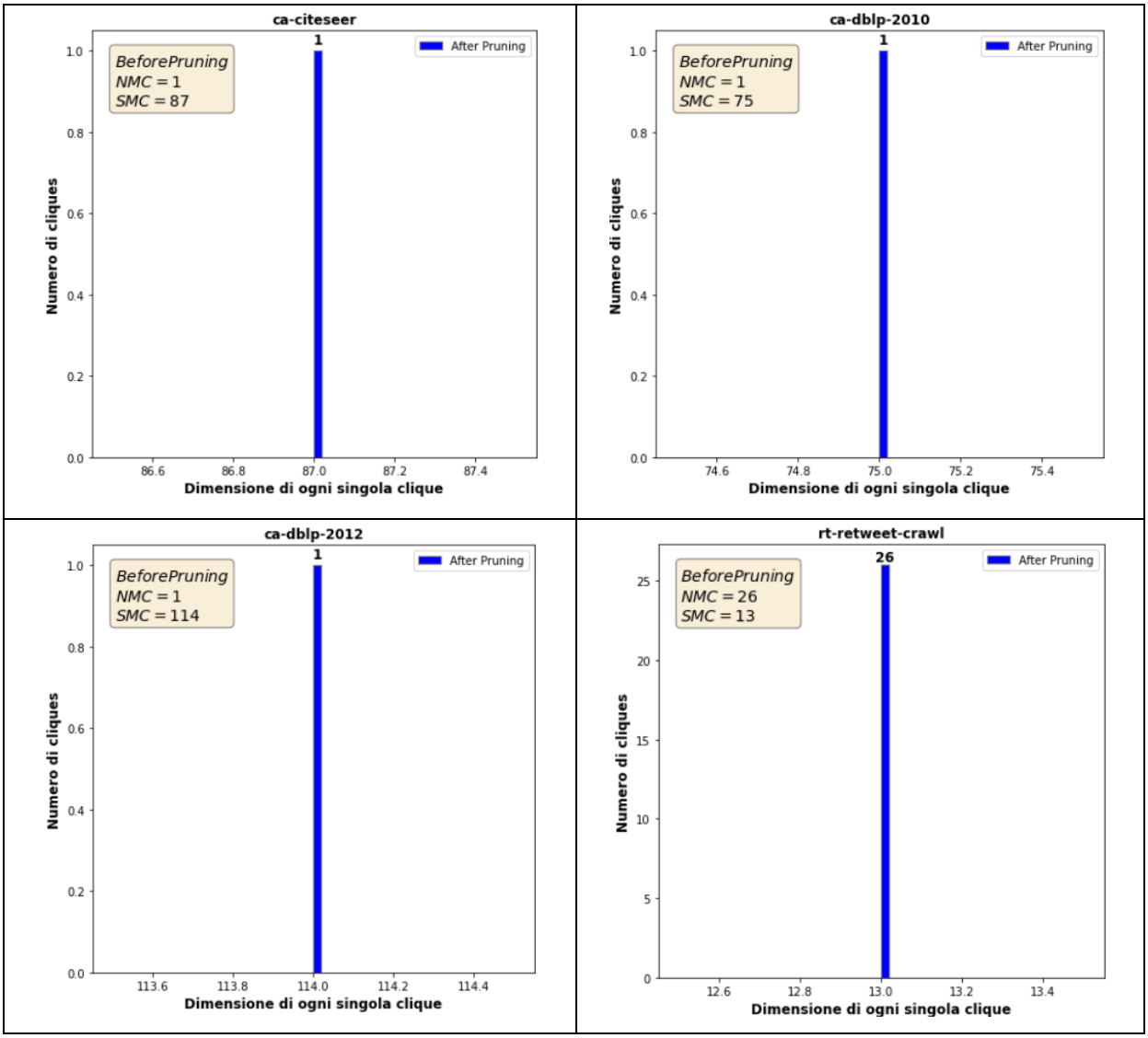
	Graph	Node Pruning Rate	Edge Pruning Rate	Solver Execution Time Before Pruning	Solver Execution Time After Pruning	Speedup	Similarity
0	ca-citeseer	88.85%	74.54%	0m 6.05s	0m 0.19s	31.84	100.0%
1	ca-dblp-2010	88.37%	77.38%	0m 5.96s	0m 0.21s	28.38	100.0%
2	ca-dblp-2012	84.69%	74.72%	0m 11.76s	0m 0.49s	24.0	100.0%
3	rt-retweet-crawl	87.03%	76.44%	2m 23.03s	0m 3.36s	42.57	100.0%
4	rt-higgs	84.04%	70.4%	0m 18.58s	0m 0.61s	30.46	100.0%
5	soc-delicious	81.73%	62.78%	0m 33.57s	0m 2.05s	16.38	100.0%
6	soc-sign-Slashdot081106	84.61%	68.75%	0m 4.81s	0m 2.43s	1.98	100.0%
7	soc-themarker	73.48%	55.29%	13m 31.55s	10m 0.89s	1.35	100.0%
8	web-baidu-baike-related	88.53%	71.39%	0m 48.69s	0m 13.69s	3.56	100.0%
9	socfb-OR	78.0%	53.63%	0m 9.91s	0m 7.12s	1.39	100.0%
10	socfb-wosn-friends	71.39%	45.35%	0m 10.08s	0m 7.71s	1.31	100.0%
11	tech-RL-caida	87.74%	80.75%	0m 5.60s	0m 0.49s	11.43	100.0%

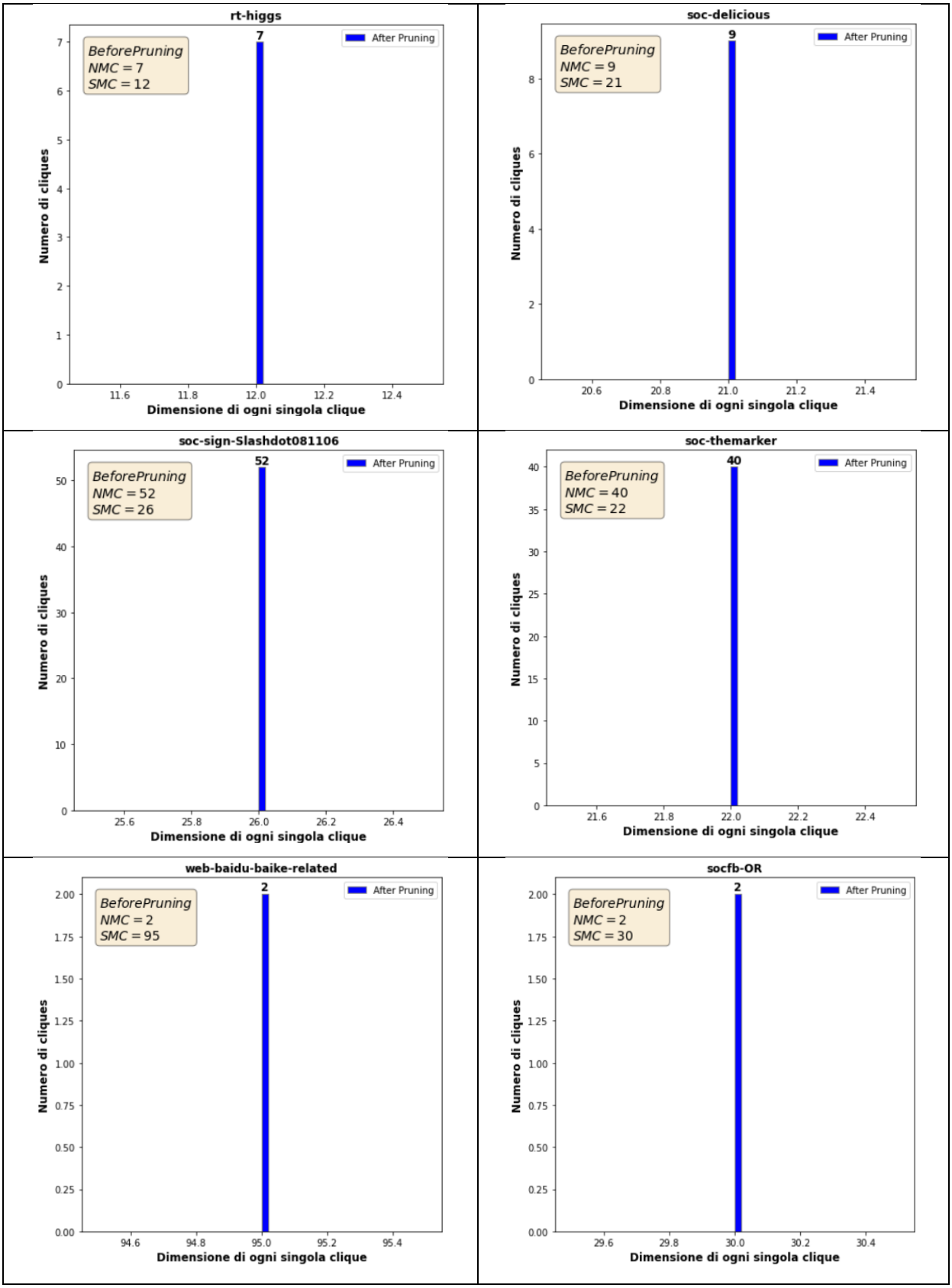
Figura 5.3.2.1: Valutazione delle performance del modello fissata una soglia specifica.

Il punto di forza del modello utilizzato all'interno del presente lavoro di tesi è che esso al termine della fase di **training** è riuscito ad apprendere talmente bene che, a seconda del dominio a cui appartiene ogni singolo grafo che gli viene fornito in ingresso, automaticamente, riesce a capire se è possibile spingere al massimo l'operazione di **node pruning** oppure no.

Ad esempio, se consideriamo tutti i grafi del testing set, eccetto quelli appartenenti ai domini **social network** e **facebook network**, notiamo che il modello, rendendosi conto del fatto che la struttura di questi grafi lo consente, spinge al massimo l'operazione di **node pruning**; mentre se consideriamo i grafi appartenenti ai domini **social network** e **facebook network** notiamo che il modello, rendendosi conto del fatto che la struttura di questi grafi non lo consente, limita il più possibile l'operazione di **node pruning** in maniera tale da preservare la Similarity. Purtroppo, per i grafi appartenenti ai due domini sopra citati non è possibile attuare un'operazione di **node pruning** troppo aggressiva in quanto la struttura di questi grafi è tale per cui basta rimuovere anche un solo nodo o arco strategico che la Similarity vale immediatamente 0. Ecco perché in questi casi è bene che il modello limiti il più possibile l'operazione di **node pruning**.

Il fatto che per ogni singolo grafo del **testing set** la Similarity è pari al 100% è confermato dalle figure sottostanti dove, per ogni singolo grafo appartenente al testing set, abbiamo una coincidenza tra il numero di clique di dimensione massima e la dimensione di tali clique ricavate a partire dal grafo originario e il numero di clique di dimensione massima e la dimensione di tali clique ricavate a partire dal grafo risultante dall'operazione di **node pruning** applicata al grafo originario:




rt-higgs
BeforePruning
NMC = 7
SMC = 12
After Pruning
7
Numero di cliques
Dimensione di ogni singola clique
soc-delicious
BeforePruning
NMC = 9
SMC = 21
After Pruning
9
Numero di cliques
Dimensione di ogni singola clique
soc-sign-Slashdot081106
BeforePruning
NMC = 52
SMC = 26
After Pruning
52
Numero di cliques
Dimensione di ogni singola clique
soc-themarker
BeforePruning
NMC = 40
SMC = 22
After Pruning
40
Numero di cliques
Dimensione di ogni singola clique
web-baidu-baike-related
BeforePruning
NMC = 2
SMC = 95
After Pruning
2
Numero di cliques
Dimensione di ogni singola clique
socfb-OR
BeforePruning
NMC = 2
SMC = 30
After Pruning
2
Numero di cliques
Dimensione di ogni singola clique

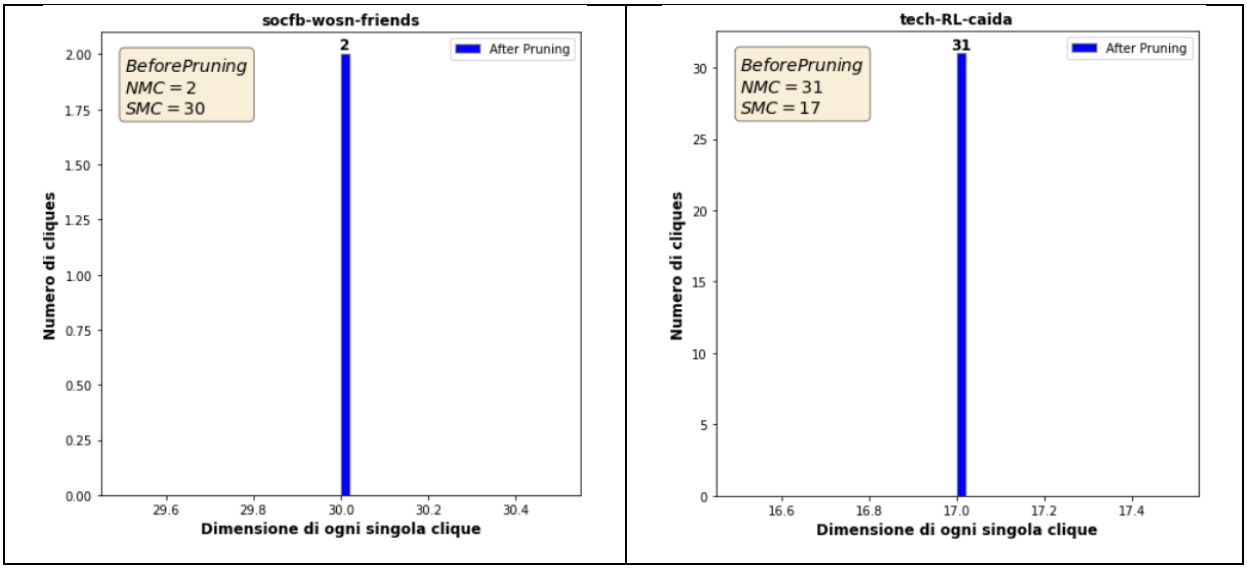


Tabella 5.3.2.1: Numero e dimensione delle clique di dimensione massima di ogni singolo grafo del testing set, prima del pruning e dopo il pruning.

5.3.3 Curve ROC

Le **curve ROC (Receiver Operating Characteristic)** rappresentano uno strumento molto potente tramite il quale è possibile visualizzare le performance di un modello. Nello specifico, una **curva ROC** è un grafico che mostra l'andamento delle performance di un classificatore binario a soglia al variare della soglia. Essa grafica il **True Positive Rate (TPR)** rispetto al **False Positive Rate (FPR)** al variare della soglia.

Una curva ROC viene costruita unendo un certo numero di punti, dove ogni singolo punto viene calcolato così:

1. Viene fissata una soglia. Inizialmente, si parte da una soglia di $-\infty$ e man mano la si incrementa.

   A partire da ogni singola soglia che man mano viene fissata, utilizzando le uscite prodotte dal modello e i targets, è possibile ricavare una **matrice di confusione** differente.

2. Una volta ottenuta la matrice di confusione a partire dalla soglia fissata, considerando i **true positive**, **false negative**, **false positive** e **true negative**, si calcola il **True Positive Rate (TPR)** applicando la seguente formula:

   $$TPR = \frac{TP}{TP + FN}$$

   dove *TP* è il numero di true positive presenti all'interno della matrice di confusione e *FN* è il numero di false negative presenti all'interno della matrice di confusione.

3. Una volta ottenuta la matrice di confusione a partire dalla soglia fissata, considerando i **true positive**, **false negative**, **false positive** e **true negative**, si calcola il **False Positive Rate (TPR)** applicando la seguente formula:

$$FPR = \frac{FP}{FP + TN}$$

dove *FP* è il numero di false positive presenti all'interno della matrice di confusione e *TN* è il numero di true negative presenti all'interno della matrice di confusione.

4. Una volta calcolato il **True Positive Rate (TPR)** (ordinata del punto) e il **False Positive Rate (FPR)** (ascissa del punto) abbiamo ottenuto le due coordinate del punto e, dunque, siamo riusciti a calcolare il punto con successo.

Unendo tutti questi punti otteniamo la curva ROC che può essere rappresentata in tal modo:

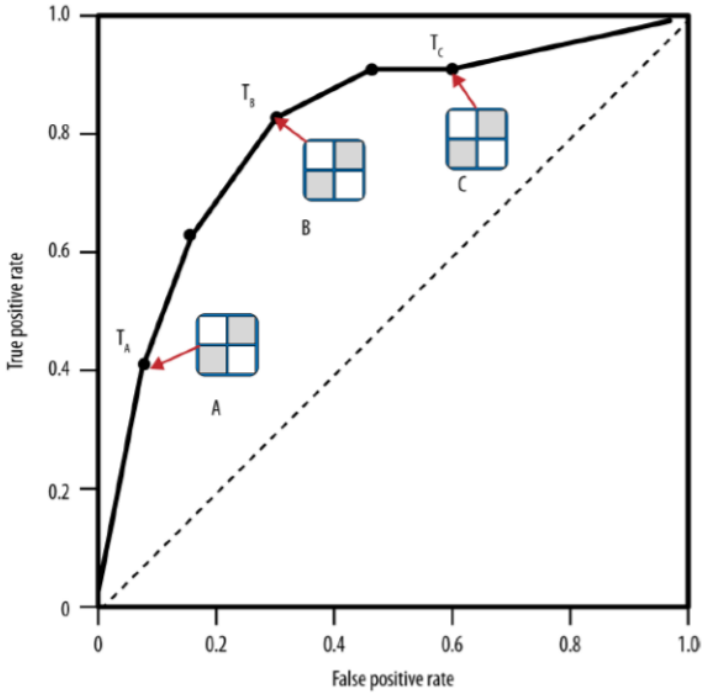


Figura 5.3.3.1: Esempio di curva ROC.

Una curva ROC ha sempre l'origine (0,0) e il punto (1,1) come estremi. Infatti, quando la soglia assume, rispettivamente, i suoi valori minimo e massimo il classificatore ha un output costante: nel primo caso tutte le osservazioni saranno predette come appartenenti alla classe positiva, nel secondo saranno predette come appartenenti alla classe negativa.

Una volta realizzata la curva ROC è possibile scegliere un valore adeguato di soglia tale per cui si ottiene un buon compromesso tra **True Positive Rate (TPR)** e **False Positive Rate (FPR)**. Idealmente, vorremmo che il **True Positive Rate (TPR)** sia il massimo possibile e che il **False Positive Rate (FPR)** sia il minimo possibile.

Le **curve ROC** vengono utilizzate tantissimo anche perché consentono di valutare la bontà di un classificatore binario indipendentemente da uno specifico valore di soglia.

Ciò viene fatto misurando l'area compresa tra l'asse delle ascisse e la curva stessa.

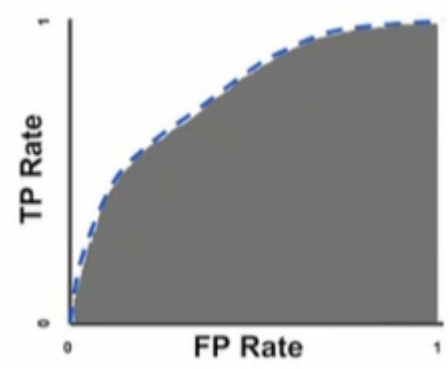
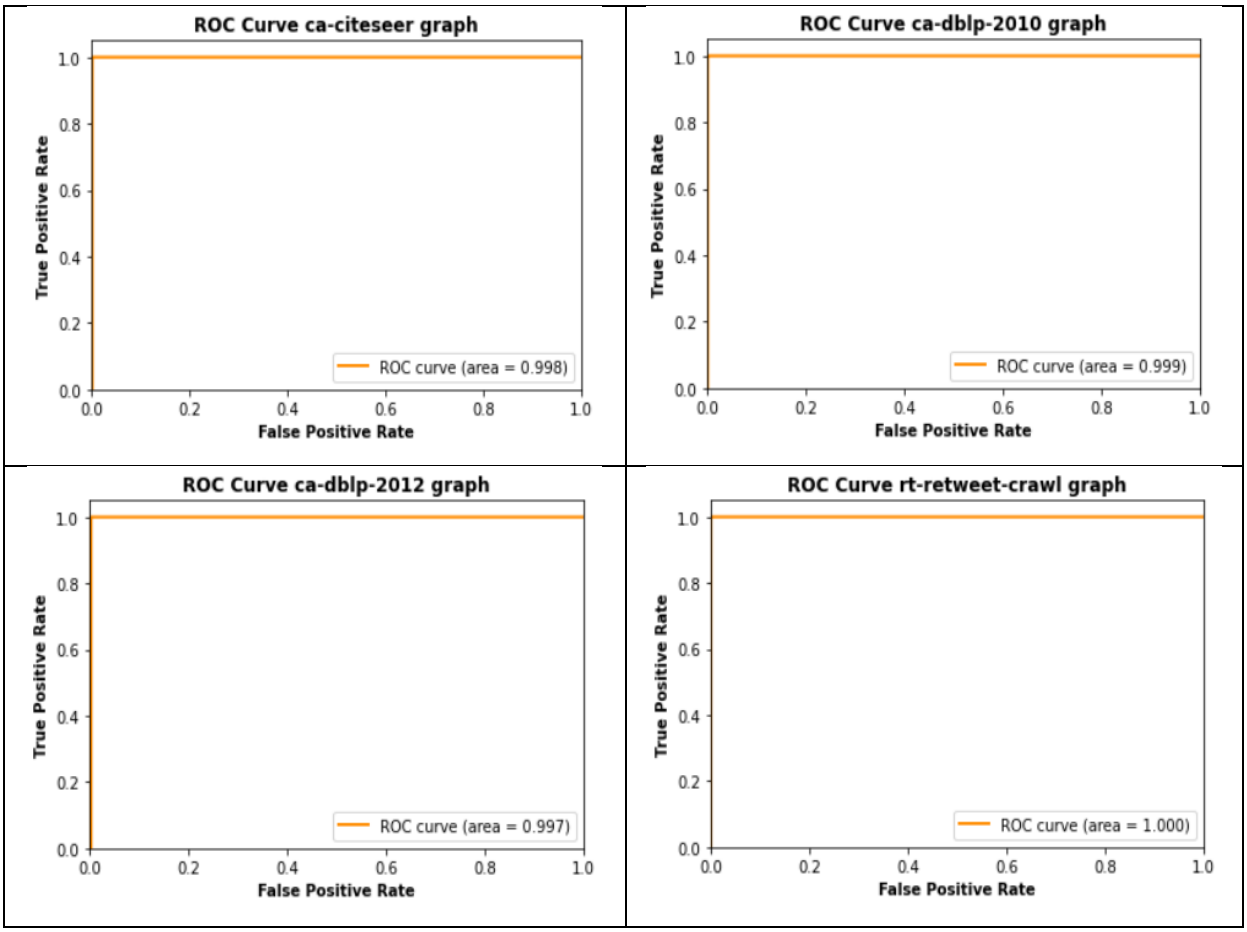


Figura 5.3.3.2: Esempio di AUC.

Il valore di tale area (zona grigia della figura sovrastante) viene indicato con la sigla **AUC** (che corrisponde ad "Area Under the ROC Curve"): più si avvicina a 1, più il classificatore ha un comportamento che approssima quello del caso ideale e più il classificatore si comporta bene. Viceversa, più si avvicina a 0, più il classificatore ha un comportamento che si allontana da quello ideale e più il classificatore si comporta male.

Fatta questa premessa prettamente teorica, presentiamo le **curve ROC** tracciate per ogni singolo grafo appartenente al **testing set**:



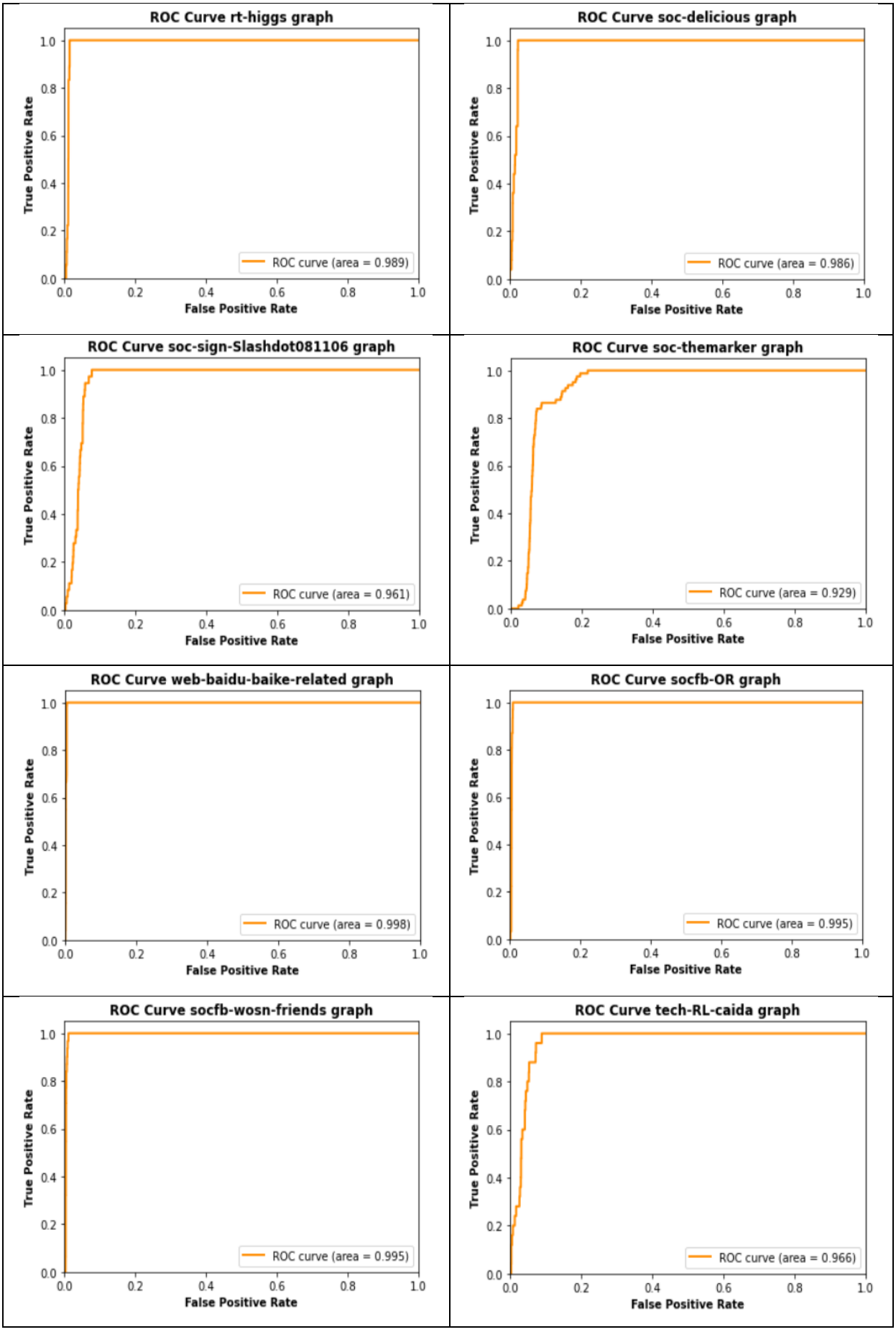


Tabella 5.3.3.1: Curve ROC e AUC per ogni singolo grafo del testing set.

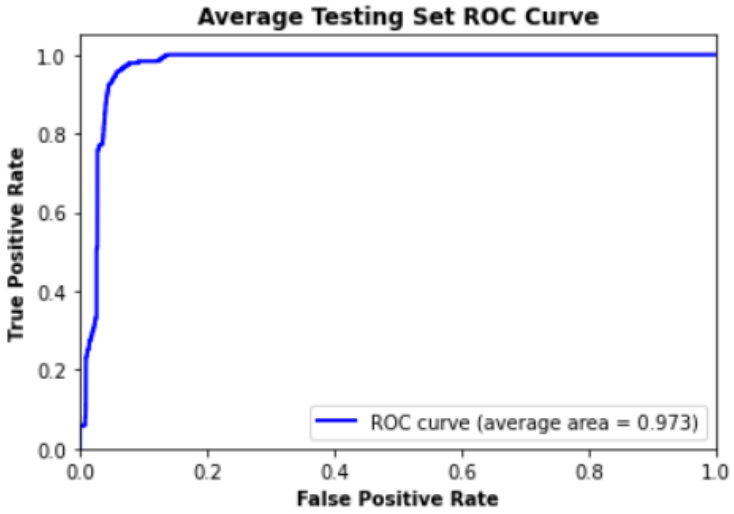


Figura 5.3.3.3: Curva ROC e AUC considerando l'intero testing set.

La curva ROC mostrata nella figura sovrastante è estremamente importante in quanto consente di valutare le performance del modello utilizzato all'interno di questa tesi considerando l'intero **testing set**. In sintesi, tramite questa curva, mediamente, è possibile comprendere quanto bene ha appreso il modello considerando l'intero testing set.

Come si vede dalla figura sovrastante, dopo aver fornito l'intero testing set al modello e, dopo aver calcolato l'**AUC**, è venuto fuori un valore pari a **0.973**, il quale è elevatissimo e, soprattutto, molto vicino ad 1.

Il fatto che l'**AUC** sia vicinissimo ad 1 è un'ulteriore conferma, se mai ce ne fosse bisogno, del fatto che il modello utilizzato all'interno del presente lavoro di tesi consente, effettivamente, di abbassare i tempi di computazione necessari affinché il problema della **determinazione delle clique di dimensione massima di un grafo** possa essere risolto.

5.4 Conclusioni

I risultati raggiunti all'interno del presente lavoro di tesi confermano il fatto che l'utilizzo del **Geometric Deep Learning** rappresenta una strategia adeguata tramite la quale è possibile accelerare la risoluzione del problema della **determinazione delle clique di dimensione massima di un grafo**, il quale, come detto, è un problema di tipo **NP-Hard**.

I risultati ottenuti, non solo dimostrano che è possibile ridurre i tempi di computazione del problema appena citato, ma sono anche la prova inconfutabile del fatto che è possibile raggiungere tale obiettivo preservando la soluzione finale e, dunque, evitando la riduzione della dimensione delle clique di dimensione massima e la perdita di alcune di esse.

Ciò è stato possibile grazie alla scelta di una **soglia** che, per tutti i grafi appartenenti al **testing set**, ha consentito di massimizzare il **Node Pruning Rate**, l'**Edge Pruning Rate** e lo **Speedup** (in maniera tale da ridurre il più possibile il tempo necessario affinché il Solver possa determinare le clique di dimensione massima di ogni singolo grafo) garantendo, al tempo stesso, una **Similarity** massima pari al 100%.

In conclusione, la forza del modello di Geometric Deep Learning utilizzato all'interno di questa tesi è da ricercare nella sua capacità di comprendere, compatibilmente al dominio di appartenenza del grafo che gli viene fornito in ingresso, quando è possibile generare delle probabilità in uscita che sono tali per cui l'operazione di **node pruning** viene enfatizzata moltissimo e quando, invece, è necessario generare delle probabilità in uscita che sono tali per cui l'operazione di **node pruning** viene enfatizzata poco.

Ciò che realmente avvalora il lavoro svolto è il fatto che i risultati presentati nei paragrafi precedenti sono stati raggiunti, dopo aver effettuato il **training**, la **validation** e il **testing** di un modello di Geometric Deep Learning, utilizzando delle risorse di calcolo **limitate**.

In **prospettiva futura**, avendo a disposizione delle risorse di calcolo adeguate per il problema che si vuole risolvere, è possibile incrementare il numero di **features** utilizzate e il numero di grafi appartenenti al **training set** in maniera tale da spingere le prestazioni ad un livello ancora superiore e, magari, utilizzare per il **testing set** dei grafi aventi 30-40 milioni di nodi/archi. Ovviamente, sempre in prospettiva futura, è possibile anche adottare lo stesso identico approccio utilizzato all'interno di questa tesi al fine di ridurre i tempi di computazione di altri problemi di tipo NP-Hard.

Bibliografia

[1] Albert-László Barabási (2015). Network Science.
Available from: http://networksciencebook.com/chapter/1#networks

[2] Juho Lauri and Sourav Dutta (2019). Fine-grained Search Space Classification for Hard Enumeration Variants of Subset Problems.
Available from: https://arxiv.org/pdf/1902.08455.pdf

[3] Marco Grassia, Juho Lauri, Sourav Dutta and Deepak Ajwani (2019). Learning Multi-Stage Sparsification for Maximum Clique Enumeration.
Available from: https://arxiv.org/pdf/1910.00517.pdf

[4] National Research Council. Network Science. The National Academies Press, 2005.

[5] Giorgio Poletti (2015). Grafi e strutture.
Available from:
http://www.unife.it/letterefilosofia/comunicazione/insegnamenti/tecnologie_informatiche_multimediali/archivio/aa-2012-2013-1/materiale-didattico/dispense-e-link/note-di-teoria-dei-grafi

[6] S. Bonaccorsi (2016). Teoria dei grafi e applicazioni.
Available from: http://www.science.unitn.it/probab/Mathmodels/article-grafi-noslide.pdf

[7] Noël Malod-Dognin, Rumen Andonov and Nicola Yanev. Maximum cliques in protein structure comparison. 6049:106–117, 05 2010.

[8] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. Commun. ACM, 16(9):575–577, September 1973.

[9] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. InterJournal, Complex Systems:1695, 2006.

[10] David Eppstein, Maarten Löffler and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. CoRR, abs/1006.5440, 2010.

[11] Bharath Pattabiraman, Md. Mostofa Ali Patwary, Assefaw Hadish Gebremedhin, Weikeng Liao and Alok N. Choudhary. Fast algorithms for the maximum clique problem on massive graphs with applications to overlapping community detection. CoRR, abs/1411.7460, 2014.

[12] Sampo Niskanen and Patric RJ Östergård. Cliquer User's Guide: Version 1.0. Helsinki University of Technology Helsinki, Finland, 2003.

[13] Patric R. J. Östergård. A fast algorithm for the maximum clique problem. Discrete Appl. Math., 120(1-3):197–207, August 2002.

[14] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu and Linhong Zhu. Finding maximal cliques in massive networks. ACM Trans. Database Syst., 36(4):21:1–21:34, December 2011.

[15] Kevin P. Murphy. Machine Learning: A Probabilistic Perspective. The MIT Press, 2012.

[16] Thomas M. Mitchell. Machine Learning. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[17] Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[18] F. Provost and T. Fawcett. Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking. O'Reilly Media, 2013.

[19] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. CoRR, abs/1611.08097, 2016.

[20] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. CoRR, abs/1607.00653, 2016.

[21] William L. Hamilton, Rex Ying and Jure Leskovec. Representation learning on graphs: Methods and applications. CoRR, abs/1709.05584, 2017.

[22] William L. Hamilton, Rex Ying, Jure Leskovec and Rok Sosic. Representation learning on networks. Lyon, France, April 2018. The Web Conference, WWW-18.

[23] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907, 2016.

[24] William L. Hamilton, Rex Ying and Jure Leskovec. Inductive representation learning on large graphs. CoRR, abs/1706.02216, 2017.

[25] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, 2015.

Indice delle figure

Indice delle tabelle