# SDN forming per mobile IoT

Le reti di sensori wireless, comunemente chiamate WSN Wireless Sensor Networks, vengono usate per molte applicazioni a lungo termine, come ad esempio:

- Applicazioni militari
- Città intelligenti e servizi alla popolazione
- Industria 4.0 e smart factories
- veicoli intelligenti

Tutte queste applicazioni innovative e fondamentali nei prossimi decenni possono trarre molto valore dalle reti di sensori. Più in generale possiamo pensare ad una rete come ad un modello che cerca di catturare la complessità delle relazioni e delle varie ricombinazioni ed interazioni fra loro e gli agenti fra i quali si esplicano queste relazioni. Con agenti intendiamo qualunque entità che ha interesse nell'interagire con gli altri, questa interazione può avvenire per ottenere un guadagno (non solo monetario), oppure per vincoli del sistema in cui acquisisce valore la rete.

Queste reti e relazioni giocano un ruolo molto importante negli scambi economici, ad esempio: scambio e commercio di beni in mercati decentralizzati, ricerca e sviluppo e relazioni anche collusive fra grandi aziende e infine trattati commerciali fra nazioni. Dobbiamo studiare le reti non solo per la loro espressività ed efficacia nel modellare relazioni economiche, ma soprattutto perché il modello di rete usato nelle telecomunicazioni, così come inteso nelle reti sociali o economiche, seguono dei precisi schemi di genesi. Questi processi di genesi della rete sono ancora oggetto di ricerca ma con le risposte che ad oggi abbiamo, attingendo dalle reti sociali ed economiche, possiamo trarre importanti lezioni e spunti di riflessione.

In prima battuta, dobbiamo dire che i processi di genesi e "formazione di reti" sono classificabili in due modi:

- Distribuiti, reti economiche e sociali prive di un controllo dall'alto
- Centralizzate, reti di telecomunicazione controllate dall'alto e progettate

Il motivo principale per cui ci preoccupiamo di scoprire quali sono i modelli di formazione, cercare di approfondirli e cercare di controllarli è perché la struttura a rete fa da discrimine fra una rete che non funziona e una rete che "funziona" e soprattutto riuscendo ad ottenere una vista in questi processi possiamo valutare i vari trade-off che caratterizzano queste reti. Queste sono alcune delle domande interessanti che nascono:

- Come fanno queste relazioni ad essere importanti nel determinare il risultato di una trasmissione dati?

**Random deployment**
È l'attività di distribuzione, installazione ed operazione dei dispositivi in posizioni geografiche totalmente causali

- Come possiamo predire reti che è probabile che si formino quando faccio un *random deployment* e lascio i dispositivi liberi di connettersi con i propri vicini?
- Quanto sono efficienti le reti che si formano e come fa questa efficienza a dipendere dal "valore" che ogni dispositivo ha?

Per una trattazione più indirizzata verso gli aspetti legati all'economia dei modelli a reti e i punti in comune con le reti di telecomunicazioni consultare gli articoli di , .

Un primo obiettivo che mi pongo in questo manoscritto è quello di considerare le differenze fra i vari modelli di network forming, le strategie e le differenze. Valuteremo anche le differenze fra un approccio totalmente distribuito ed uno centralizzato e tenterò di dare risposta alla domanda: quanto mi costa il *non* usare un controllo centralizzato della rete? Come misuro questo costo?

## Reti di sensori

Una rete di sensori solitamente consiste in device:

- sorgenti, i sensori che campionano l'ambiente e producono informazioni utili
- terminazioni, le base station o i clusterhead

- usati dagli utenti finali

I nodi sensori vengono installati in zone geografiche d'interesse, dove grazie alle loro caratteristiche di costo ridotto, auto-organizzazione e intercambiabilità permettono di avere un piano di raccolta dati fungibile e capillare. Per poter raggiungere l'obiettivo di avere una rete auto-organizzante per monitorare ambienti ho bisogno di campionare il fenomeno d'interesse, codificarlo e trasmettere questa rappresentazione verso un nodo consumatore dell'informazione attraverso un protocollo di routing.

Un'ipotesi che spesso viene fatta è che i nodi consumatori siano più potenti dei nodi campionatori. In figura vediamo uno scenario tipo di rete di sensori. È da osservare la presenza della nuvoletta che rappresenta la Wide Area Network che metterà in comunicazione il data center di elaborazione dati e l'utente finale.

**Clustertree**
Struttura a grafo aciclico in cui ogni nodo è un cluster di nodi rappresentati da un super nodo detto clusterhead

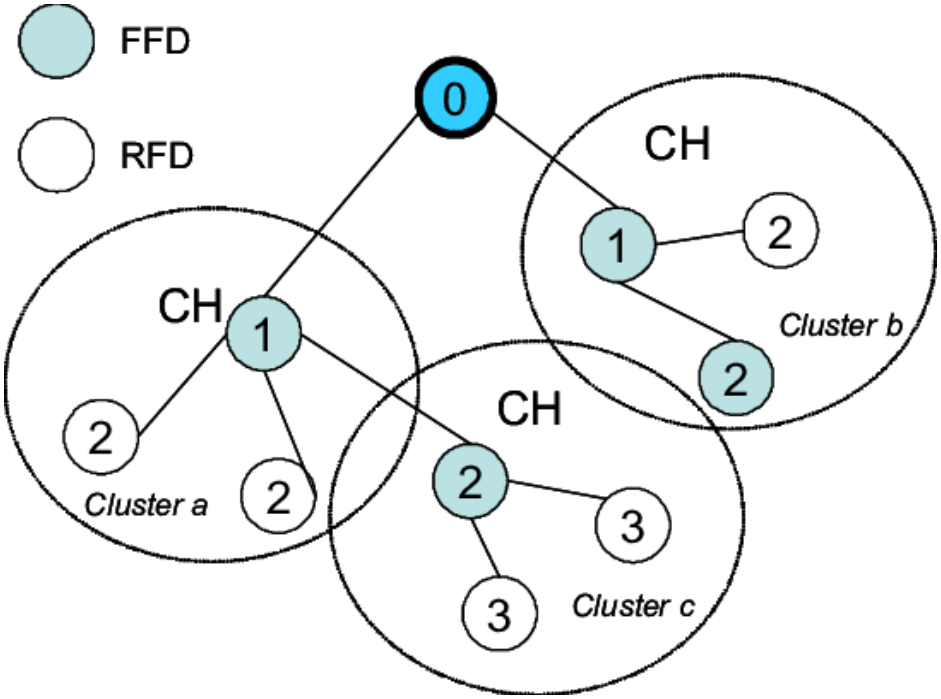



***Fig. 1*** Scenario tipo organizzato come **cluster-tree** con reduced function device e full function devices

Com'è composto il singolo device sensore? A grandi linee possiamo pensarlo come composto dai seguenti blocchi in figura:

**Precisazione sullo schema a blocchi**
I blocchi tratteggiati sono opzionali

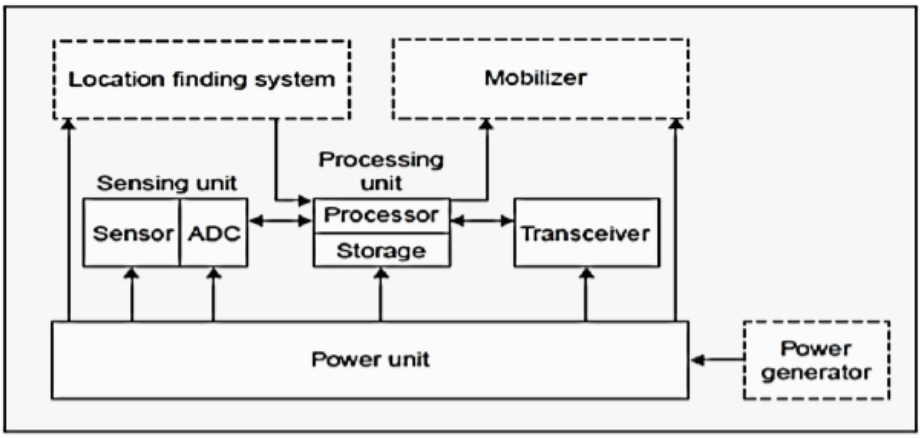



***Fig. 2*** Schema a blocchi di com'è composto un sensore intelligente

Le reti di sensori sono molto particolari per i seguenti motivi:

1. I nodi sensori sono limitati in termini di batteria (energia), capacità computazionale e memoria. Il parametro più sensibile è proprio la durata della batteria. Data la natura wireless delle comunicazioni bisogna fare un bilancio di tratta e ricordarsi che la *path loss* è il fattore predominante nel dispendio necessario per trasmettere da A a B.
   $FSPL = P_T G_T G_R (\frac{\lambda}{4\pi d})^2$
   dunque vediamo come la riduzione della potenza al ricevitore segue una legge quadratica $d^{-2}$. Questo è un problema che induce a progettare bene gli algoritmi di routing, in quanto una trasmissione completamente 1-hop è impensabile, servono algoritmi di routing multi-hop con le loro peculiarità.
2. Dato il costo ridotto dei dispositivi sensori è possibile installare un gran numero di dispositivi. Avere un gran numero di dispositivi significa coprire meglio il territorio ed essere più capillari (e di conseguenza efficaci) nel monitoraggio. D'altro canto non tutti i dispositivi verrebbero usati, avrei una certa percentuale inutilizzata e questo riduce l'efficienza della rete. Come in molti altri casi la questione sta nel trovare un equilibrio accettabile.

**Grado di connessione**
È la distribuzione di porbabilità del numero di collegamenti logici che un dispositivo ha con i suoi vicini

1. Dopo il rollout, i nodi si organizzano in una rete e cooperano per portare a termine un compito. In generale non c'è un nodo coordinatore centrale che orchestra tutto quanto. Inoltre data la natura della rete dopo verrà dimostato come il fallimento o guasto casuale nelle reti di sensori che soddisfano un criterio di normalità sulla distribuzione dei gradi di connessione, non vengono influenzati in modo rilevante dai guasti random.
2. La topologia della rete cambia di frequente poiché oiltre ai guasti dobbiamo tenere conto anche dei cicli di sleep/operazione del dispositivo, oltre che al cambio di ruolo che ogni nodo ricoprirà (vedi capitoli successivi).

# Ottimizzazione del traffico

In questo capitolo vado a mostrare gli aspetti di modellazione del sistema, enunciare il problema e gli algoritmi proposti.

## Introduzione

Il routing in una rete è un'operazione complessa che coinvolge molti attori e protocolli che permettono l'organizzazione fra tutti questi componenti indipendenti fra di loro. Le ragioni della complessità sono molteplici:

Il routing richiede un grado di coordinazione fra i vai nodi nella rete o nel segmento di rete considerato piuttosto che solo fra due nodi. Ad esempio pensiamo i protocolli di livello 2 e livello 4.

L'architettura di routing deve risolvere tutti quei casi di guasti e malfunzionamenti dei collegamenti o dei nodi, redirezionare il traffico ed aggiornare i database informativi di ogni device.

Per ottenere delle performance buone, l'algoritmo di routing può essere costretto a modificare le sue rotte precedentemente calcolate quando parti o segmenti di rete diventano congestionati.

## Difficoltà e obiettivi nel routing

Le due funzioni principali di un algoritmo di routing sono:

**Routing table**
È la struttura dati che un dispositivo ha al suo interno in cui troviamo una lista di destinatari, la distanza e come raggiungere quel preciso destinatario ed in più altre informazioni a seconda del protocollo finale

1. la selezione delle rotte fra le varie coppie origine destinazione
2. la consegna dei messaggi alla corretta destinazione una volta scelta la rotta

La seconda funzione può sembrare banale una volta usate le routing table. Ciò non è sempre vero, in quanto questo concetto è molto semplice nel caso che io possa fare *affidamento* sull'infrastruttura di rete. Nel caso in cui questo non sia vero e quindi io ho a che fare con una infrastruttura non affidabile, diventa fondamentale il modo in cui scelgo il next-hop ed il modo in cui progetto la rete per la sua robustezza.

**Next-hop**
È il dispositivo successivo rispetto al device sotto esame, all'interno del percorso che unisce topologicamente due punti di interesse, ad esempio possono essere una coppia origine-destinazione

È possibile che 2 scelte diverse di next-hop portino allo stesso destinatario nella topologia, ma supponiamo un next-hop venga disconnesso poiché il collegamento prescelto non è più disponibile. Nel caso che un collegamento non sia disponibile, non mi concentro più sulla seconda funzione e sul capire cosa succede, ma mi concentrerò sulla prima funzione e come influisce sulle performance della rete.

Per quanto riguarda la prima funzione, e cioè la selezione delle rotte corrette dall'origine alla destinazione, osserviamo che ci sono due metriche principali sulla performance di una rete che sono influenzati dall'algoritmo di routing:

1. Throughput, la quantità di servizio offerta dalla rete
2. Latenza media dei pacchetti, la qualità del servizio offerto dalla rete

Il routing interagisce con il controllo di flusso per determinare queste metriche di performance attraverso un controllo a catena chiusa.

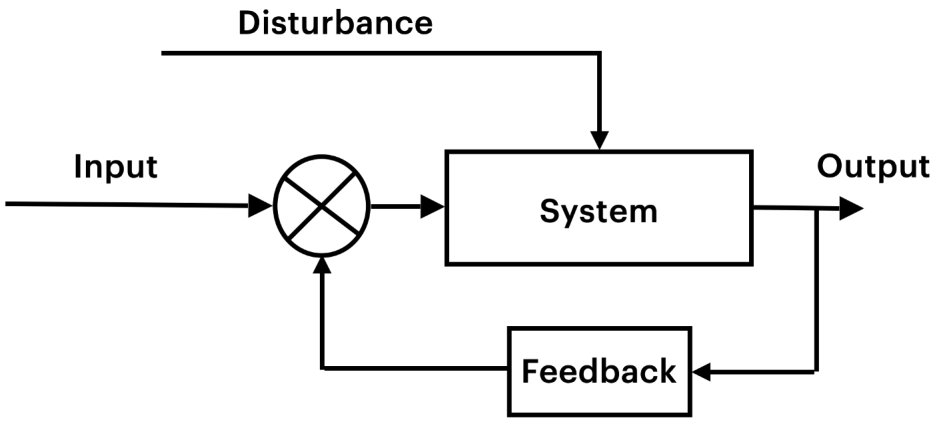



***Fig. 3*** Immagine **temporanea**

Una euristica molto efficace nel tentare di mantenere bassa la latenza di percorrenza del sistema all'aumentare del traffico è la seguente:

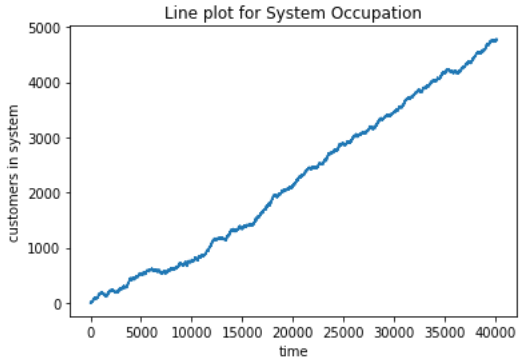
> All’aumentare dell’intensità media del carico offerto valuto la latenza media del sistema, quando qusta diventa eccessiva inizio a scartare dei pacchetti

Questo ha come effetto il comportamento secondo cui tanto più l’algoritmo riesce a tenere bassa la latenza e tanto più potrà ammettere traffico ed aumentare così il throughput che la rete offre.

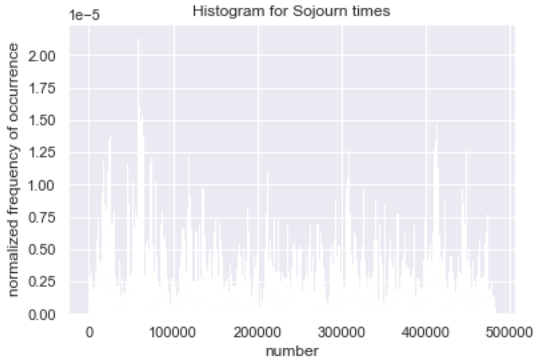
Propongo la seguente simulazione:

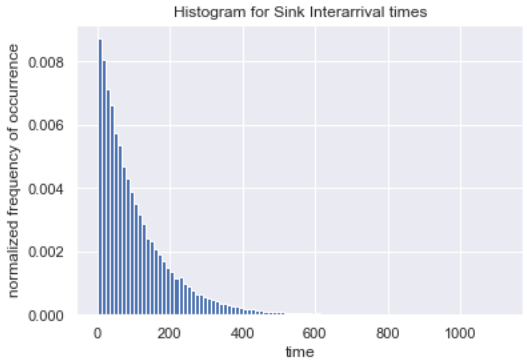
```
import sys
sys.path.append(r"C:\Users\DULLA\PycharmProjects\Thesis_Data_Analisys\models")
sys.path.append(r"C:\Users\DULLA\PycharmProjects\Thesis_Data_Analisys")
import random
import functools
import seaborn as sns
import simpy
import numpy
import matplotlib.pyplot as plt
import pandas
from models import SimComponents
from SimComponents import PacketGenerator, PacketSink, SwitchPort, PortMonitor
adist = functools.partial(random.expovariate, 0.01)
sdist = functools.partial(random.expovariate, 0.05)  # mean size 100 bytes
samp_dist = functools.partial(random.expovariate, 0.005)
port_rate = 1.5

env = simpy.Environment()  # Create the SimPy environment
# Create the packet generators and sink
ps = PacketSink(env, debug=False, rec_arrivals=True)
pg = PacketGenerator(env, "Greg", adist, sdist)
switch_port = SwitchPort(env, port_rate, qlimit=100000)
# Using a PortMonitor to track queue sizes over time
pm = PortMonitor(env, switch_port, samp_dist)
# Wire packet generators, switch ports, and sinks together
pg.out = switch_port
switch_port.out = ps
# Run it
env.run(until=8000000)
# print("Last 10 waits: " + ", ".join(["{:.3f}".format(x) for x in ps.waits[-10:]]))
# print("Last 10 queue sizes: {}".format(pm.sizes[-1000:]))

fig, axis = plt.subplots()
axis.plot(pm.sizes)
axis.set_title("Line plot for System Occupation")
axis.set_xlabel("time")
axis.set_ylabel("customers in system")
plt.show()
sns.set_theme(style="darkgrid")

# print("Last 10 sink arrival times: " + ", ".join(["{:.3f}".format(x) for x in
ps.arrivals[-10:]]))
print("average wait = {:.3f}".format(sum(ps.waits) / len(ps.waits)))
print(
    "received: {}, dropped {}, sent {}".format(switch_port.packets_rec,
switch_port.packets_drop, pg.packets_sent))
print("loss rate: {}".format(float(switch_port.packets_drop) / switch_port.packets_rec))
print("average system occupancy: {:.3f}".format(float(sum(pm.sizes)) / len(pm.sizes)))
fig, axis = plt.subplots()
axis.hist(ps.waits, bins=10000, density=True)
axis.set_title("Histogram for Sojourn times")
axis.set_xlabel("number")
axis.set_ylabel("normalized frequency of occurrence")
plt.show()
fig, axis = plt.subplots()
axis.hist(ps.arrivals, bins=100, density=True)
axis.set_title("Histogram for Sink Interarrival times")
axis.set_xlabel("time")
axis.set_ylabel("normalized frequency of occurrence")
plt.show()
i = 0
avgWait = []
del env, ps, pg, pm, switch_port
while i < 50:
    env = simpy.Environment()
    ps = PacketSink(env, debug=False, rec_arrivals=True)
    pg = PacketGenerator(env, "Greg", adist, sdist)
    switch_port = SwitchPort(env, port_rate, qlimit=1000)
    pm = PortMonitor(env, switch_port, samp_dist)
    pg.out = switch_port
    switch_port.out = ps
    env.run(until=800000)
    wait = sum(ps.waits) / len(ps.waits)
    # print("average wait = {:.3f}".format(wait))
    avgWait.append(str(wait))
    del env, ps, pg, pm, switch_port
    i += 1
sns.set()
fig, axes = plt.subplots()
plt.title("Latenza media con coda lunga 1000")
avgWait = numpy.asarray(avgWait, dtype=numpy.single)
df = pandas.DataFrame({"y": avgWait, "x": numpy.asarray(range(len(avgWait)),
dtype=numpy.single)})
sns.regplot(x="x", y="y", data=df, scatter=False)
p = sns.lineplot(x="x", y="y", data=df)
p.set(xlabel="Latency", ylabel="Simulation run number")
plt.show()
```

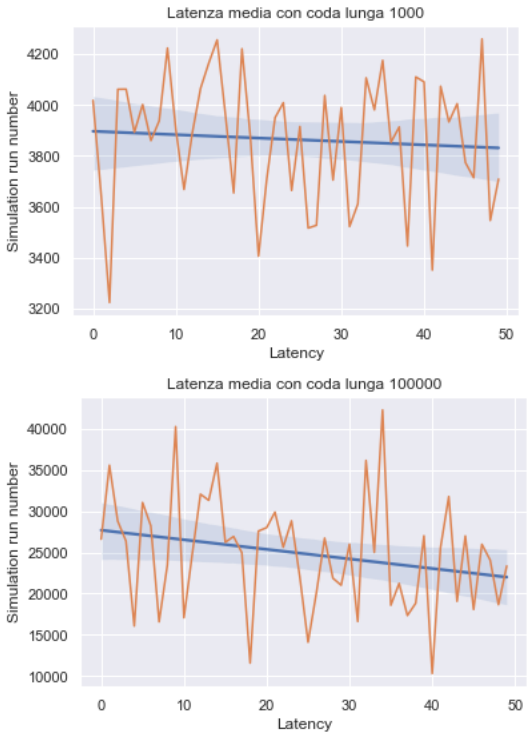
```
i = 0
del avgWait, df
# del env, ps, pg, pm, switch_port
avgWait = []
while i < 50:
    env = simpy.Environment()
    ps = PacketSink(env, debug=False, rec_arrivals=True)
    pg = PacketGenerator(env, "Greg", adist, sdist)
    switch_port = SwitchPort(env, port_rate, qlimit=100000)
    pm = PortMonitor(env, switch_port, samp_dist)
    pg.out = switch_port
    switch_port.out = ps
    env.run(until=800000)
    wait = sum(ps.waits) / len(ps.waits)
    # print("average wait = {:.3f}".format(wait))
    avgWait.append(str(wait))
    del env, ps, pg, pm, switch_port
    i += 1
sns.set()
fig, axes = plt.subplots()
avgWait = numpy.asarray(avgWait, dtype=numpy.single)
plt.title("Latenza media con coda lunga 100000")
df = pandas.DataFrame({"y": avgWait, "x": numpy.asarray(range(len(avgWait)),
dtype=numpy.single)})
sns.regplot(x="x", y="y", data=df, scatter=False)
p = sns.lineplot(x="x", y="y", data=df)
p.set(xlabel="Latency", ylabel="Simulation run number")
plt.show()
```




```
average wait = 221221.584
received: 79608, dropped 0, sent 79608
loss rate: 0.0
average system occupancy: 2208.712
```






Da questi grafici vediamo come il traffico sia da trattare in modo diverso rispetto alla classica elaborazione statistica dei segnali. Tutto questo nasce dal fatto che ho bisogno di implementare un controllo della rete. I benefici dell'uso di algoritmi di controllo SDN unito alle tecniche di controllo SDN per la disseminazione dei risultati e dei comandi, verranno esposti nell'ultimo capitolo che presenta le applicazioni e i risultati delle simulazioni delle varie applicazioni.

# Distribuzioni stabili

La discussione che segue non ha la pretesa di esaustività ma soltanto di portare alla luce gli aspetti principali e quali sono le conseguenze dell'uso e delle presenza di distribuzioni stabili nel modello del sistema.

## Introduzione

I processi stabili hanno attratto un'attenzione sempre maggiore, una descrizione sistematica della teoria dei processi "atipici" e cioè con code pesanti la possiamo trovare in [Bro55] oppure [Fel71] ed infine in

Il teorema limite centrale offre la giustificazione fondamentale per usare una trattazione più semplificata alle distribuzioni stabili, possiamo estenderlo ed enunciare che: le distribuzioni stabili sono le distribuzioni limite della somma di distribuzioni normalizzate, indipendenti ed identicamente distribuite. Il caso gaussiano come distribuzione limite è solo il caso più famoso ed ampiamente studiato. I processi gaussiani si prestano ad una trattabilità matematica molto agevole e quindi permettono di ottenere degli ottimi indizi anche nel caso in cui io abbia variabili aleatorie qualunque, da qui la "fama" dei processi gaussiani. Inoltre c'è da dire che i processi gaussiani sono stati i primi ad essere compresi completamente e ad esssere utilizzati nella pratica in quanto ottimi strumenti di modellazione analitica. Il problema sta in tutti quei casi di modellazione di processi stocastici soggetti a larghe fluttuazioni. Solitamente si modella le larghe fluttuazioni probabilistiche come processi *non* stazionari e si riduce l'orizzonte di osservazione ad un intervallo opportuno, sufficiente piccolo, sufficiente a conferire a quel processo delle caratteristiche di stazionarietà. Basti pensare all'uso di strumenti come la STFT (short time Fourier transform) per l'analisi di segnali vocali o multimediali i quali vengono in alcuni casi modellati come processi gaussiani non stazionari.

> **Note**
>
> Overdispersion È la caratterstica per cui un campione mostra una varianza maggiore di quella prevista. Questa varianza anomala avviene per colpa della varianza che diverge nell'espressione della mia pdf.

Il problema della distribuzione gaussiana usata per modellare è che non riesce a catturare le caratteristiche di grandi fluttuazioni dalla media come ad esempio fenomeni di overdispersion. Le distribuzioni stabili non hanno queste limitazioni. In generale, queste distribuzioni hanno delle code che decadono con una "power-law".

> **Note**
>
> Power-law decay Con decadimento con legge esponenziale intendiamo una relazione fra due quantità tale per cui una variazione nella prima risulta in un cambio proporzionale nella seconda indipendentemente dalle loro dimensioni iniziali (pensiamo all'area ed al legame con la lunghezza del lato)

Nelle distribuzioni stabili si ha poche "intuizioni applicate", la più famosa è al "Teoria del Cigno Nero" del dott. Nassim Taleb la quale spiega come un evento molto sorprendente venga erroneamente spiegato a posteriori nonstante la conoscenza a posteriori dell'evento. Più in particolare nella sua teoria del cigno nero, applicata principalmente nel mondo della finanza quantitativa, riesce a spiegare e dare degli indizi su:

1. il ruolo di eventi spoporzionatamente intensi (o di alto profilo), difficili da prevedere ed eventi rari che sono oltre le tecniche solitamente usate in discipline come tecnologia e finanza (ad esempio le medie storiche sono inaffidabili)
2. La non commensurabilità della probabilità di due eventi estremi consecutivi (poiché vigono leggi degli eventi rari)

Tornando alle distribuzioni a code pesanti, possiamo dire che queste dipendano in generale da un parametro $\alpha$ il quale ha come intervallo di definizione $0 \leq \alpha \leq 2$. Più piccolo è $\alpha$, tanto più lento sarà il decadimento e tanto più saranno pesanti, o grasse, le code. Una nota importante da fare è la seguente: le distribuzioni fat-tailed hanno sempre varianza infinita e se $\alpha \leq 1$ allora anche la media non converge.

Negli ultimi decenni, i dati con code pesanti sono stati raccolti nelle discipline più disparate:

- economia
- telecomunicazioni
- idrologia

In tutti questi campi nel corso degli anni, i dati raccolti, hanno suggerito distribuzioni non gaussiane come modello.

Le distribuzioni non gaussiane hanno l'ottimo pregio che permettono di modellare un range molto più ampio di comportamenti probabilistici se messi a confronto con distribuzioni gaussiane. Mentre da un lato ho dei modelli molto più espressivi grazie alle distribuzioni stabili, il costo che pago è una parametrizzazione molto più ricca e complicata da stimare. Questo poiché le distribuzioni gaussiane sono completamente identificate dalla loro funzione di autocovarianza e media, ciò non è più vero nelle distribuzioni stabili.

Seguo l'esposizione di [Bro55] nell'enunciare quattro definizioni equivalenti di distribuzioni stabili.

## Definizione $1$

Una variabile aleatoria $X$ è *stabile* se per dei numeri $A$ e $B$ esistono dei valori positivi $C$ e $D$ tali che

$$AX_1 + BX_2 \stackrel{d}{=} CX + D$$

dove $X_1$ e $X_2$ sono campioni di $X$ e $=^d$ denota l'uguaglianza in distribuzione. La distribuzione $X$ stabile viende detta *simmetrica* se la pdf di $X$ è uguale a quella di $-X$

## Teorema [Fel71]

Per ogni distribuzione stabile della variabile aleatoria $X$, esiste un numero $\alpha \in [0, 2]$ tale che il numero $C$ nella definizione precedente soddisfa

$$C^\alpha = A^\alpha + B^\alpha$$

il numero $\alpha$ viene chiamato *indice di stabilità* o *esponente caratteristico*. Una distribuzione stabile con variabile aleatoria $X$ con indice $\alpha$ viene definita $\alpha$-stabile.

## Definizione $2$

Una variabile aleatoria $X$ viene detta stabile se per ogni $n \geq 2$, è possibile trovare due numeri positivi $C_n$ e $D_n$ tali che

$$X_1 + X_2 + X_3 + \ldots + X_n \stackrel{d}{=} C_n X + D_n$$

dove $X_i{}_{i=1}^n$ sono campioni iid estratti da $X$. Nel libro di [Fel71] l'autore dimostra l'equivalenza fra le due definizioni in quanto se la definizione $1$ è vera allora per induzione sarà verificata anche la definizione $2$, per il viceversa consultare [Fel71].

## Osservazioni

Sempre nel libro ([Fel71] teorema VI.1.1) vediamo che se la definizione $2$ è vera allora vediamo che necessariamente che
$C_n = n^{1/2}$
per qualche $0 <= \alpha <= 2$. Questo $\alpha$ è ovviamente l'esponente caratteristico.

## Osservazione importante [GS94]

Consideriamo la sequenza $\mathbf{X} = (X_i)_{i=-\infty}^{\infty}$ di variabile aleatorie. Fissiamo un numero $\delta > 0$. Per ogni $n \geq 1$, definiamo la trasformazione $T_n$ come la mappa che dato il vettore aleatorio $\mathbf{X}$ produce $T_n\mathbf{X} = (T_nX)_{i-\infty}^{\infty}$ con
$(T_nX)_i = \frac{1}{n^\delta} \sum_{j=in}^{(i+1)n-1} X_j$
Le trasformazioni $T_n$ con $n \geq 1$ sono chiamate *trasformazioni rinormalizzanti di gruppo* con *esponentec critico* $\delta$.

### Lemma 1

$T_{mn} = T_mT_n$
Nella monografia [Bro55] viene dimostrato come $T_{mn} = T_mT_n$ cioè la trasformazione $T_m$ seguita dalla trasformazione $T_n$ sia euivalente alla trasformazione $T_{mn}$ e dunque la famiglia di trasformazioni $T_N, n \geq 1$ formano un gruppo.

### Definizione di punto fisso

Una sequenza $\mathbf{X} = (X_i)_{i=-\infty}^{\infty}$ viende detta *punto fisso* del gruppo di trasformazioni rinormalizzanti se $T_n\mathbf{X} \overset{d}{=} \mathbf{X}$ per ogni $n \geq 1$, cioè se la distribuzione di $\mathbf{x}$ è invariante sotto $T_n$ per ogni $n \geq 1$.

La sequenza $\mathbf{X} = (X_i)_{i=-\infty}^{\infty}$ di variabili aleatorie i.i.d $\alpha$-stabili è un punto fisso delle trasformazioni $T_n$ con $\delta = \frac{1}{\alpha}$ poiché:
$(T_n\mathbf{X})_i = \frac{1}{n^{\frac{1}{\alpha}}}(X_{in} + X_{in+1} + \cdot + X_{i(n+1)-1}) \overset{d}{=} X_i$
e poiché i campioni della sequenza $X_i$ sono indipendenti per ipotesi, ottengo per linearità anche l'indipendenza delle $(T_n\mathbf{X})$.

## Applicazioni

La teoria dei gruppi di trasfomazioni rinormalizzanti pone le basi per un metodo estrememante veloce per produrre e simulare variabili aleatorie con code pesanti in modo corretto. Segue una simulazione di come analizzare una distribuzione di somma di Pareto usando il package matlab [Vei22].

Consideriamo una variabile aleatoria $X$ la cui distribuzione cumulativa è data da:
$Prob[X \leq x] = \begin{cases} 1 - x^{-\frac{3}{4}} & \text{x} > 1 \\ 0 & \text{altrove} \end{cases}$

Posso campionare molto facilmente da questa, poiché inverto la CDF ottenendo:
$X = u^{-\frac{4}{3}}$ dove $u \approx \mathbf{U}(0, 1)$
i vari $X_i$ sono i.i.d.

Possiamo vedere come dalla coda di questa ditribuzione ed osservando che $\alpha = \frac{4}{3}$ qui avrò una media infinita. In questo caso, come minimo, non posso apllicare il teorema limite centrale. Tiriamo fuori il gruppo di trasformazioni rinormalizzanti ed in particolare consideriamo la somma riscalata dal fattore $n$ come nella definizione: $X^{+n} = n^{-\frac{4}{3}} \sum_{i=1}^{n} X_i$ in cui al solito ipotizziamo che i vari campio siano i.i.d.

Si può dimostrare [GS94] come la somma precedentemente descitta converga ad una distribuzione stabile $S(\alpha_0, \beta_0, \gamma_0, \delta_0)$. A questo punto possiamo verificare ciò che troviamo nel libro citato. Generiamo $X_i^{+100}$ una sequenza di 100 campioni ed applichiamo la trasformazione, come mostrato nel seguente listato:

```
N = 300;
sampsize = 100;
s = RandStream.create('mrg32k3a','NumStreams',1,'Seed',50); % For reproducibility
X = zeros(N,1);
for i = 1:N
    % Generate a normalized sum of Pareto-type random variables
    Samp = 1./rand(s,sampsize,1).^(4/3);
    X(i) = sum(Samp)/sampsize^(4/3); % Normalize sum
end
% estimate parameters
p = stblfit(X,'ecf',statset('Display','iter'));
% plot data with fit parameters
xmax = 15;
H = figure(1);
set(H,'Position', [517 626 939 410]);
clf;
title('Stable fit to sums of Pareto random variables');
subplot(1,2,1)
hold on
stem(X(X < xmax),stblpdf(X(X<xmax),p(1),p(2),p(3),p(4),'quick'));
x = 0:.1:xmax;
plot(x,stblpdf(x,p(1),p(2),p(3),p(4),'quick'),'r-')
hold off
xlabel(['\alpha_0 = ',num2str(p(1)),'  \beta_0 = ',num2str(p(2)),'  \gamma_0 = ',num2str(p(3)),'  \delta_0 = ',num2str(p(4))]);
legend('Data','Fit stable density')
subplot(1,2,2)
CDF = prctile(X,[1:75]);
cmin = CDF(1);
cmax = CDF(end);
x = cmin:.1:cmax;
estCDF = stblcdf(x,p(1),p(2),p(3),p(4));
plot(CDF,[.01:.01:.75],'b.',x,estCDF,'r-')
legend('Empirical CDF','Estimated CDF','Location','northwest')
```

```
iteration       alpha          beta         gamma         delta
   0          0.655308             1       1.10274       1.69911
   1          0.638528        1.0271       1.65565      0.636399
   2          0.735853      0.953656       1.59847      -0.107348
   3          0.746854      0.962963       1.59705      -0.311888
   4          0.747544      0.966093       1.59615      -0.338053
```

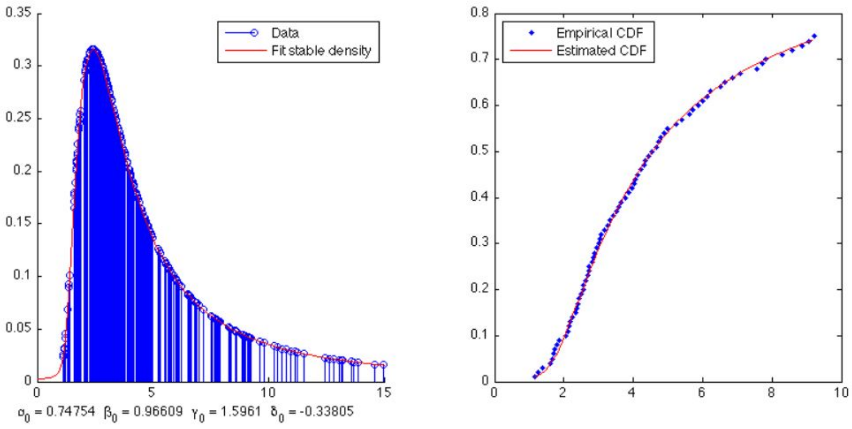



***Fig. 4*** Risultato dell'interprete

Questo risultato ci dice che l'uso dei metodi statistici classici è inefficace nel dimostrare la convergenza dei vari campioni $X_i$. Il package STBLFIT descritto nella repository [Vei22] mostra come ottenre un fit accurato e a quale distribuzione stabile convergeranno i miei dati. Ci tornerà molto utile nell'ultimo capitolo quado tenteremo un fit dei dati raccolti *live* nella rete.

## Appendice

Un risultato finale molto interessante riguarda un risultato di rappresentazione di una variabile aleatoria $\alpha$ stabile e simmetrica. In questo ultimo paragrafo mostrerò come si può decomporre una variabile aleatoria attraverso una somma di variabili di poisson indipendenti. Più in dettaglio si può dimostrare come una distribuzione $\alpha$ stabile $X$ la rappresento con una somma convergente di variabili aleatorie con un processo di arrivo di Poisson sottostante. Alla fine la discussione finale sull'utilità di questo approccio.

Sia $N(t)$ la funzione che denota il numero di clienti in arrivo al mio sistema nell'intervallo $[0, t]$. Possiamo dire che $N(t), t \geq 0$ è un *processo di Poisson* con rate $\lambda$ ed intertempi di arrivo $\tau_{i+1} - \tau_i, i \geq 1$. In tal caso gli intertempi di arrivo sono distribuiti secondo una distribuzione esponenziale con $\mathbf{E}\tau = \frac{1}{\lambda}$ e dunque $\mathbf{E}N(t) = \lambda t$ dunque il numero di clienti è proporzionale al tempo di osservazione ed il rate medio di arrivi.proporzionale

I processi di poisson hanno numerose proprietà interessanti. Questa caratteristica, insieme alla trattabilità, rende molto interessante l'enunciato ad inizio paragrafo. Vale la seguente

### proposizione

Sia $\tau_i$ la sequenza dei tempi di arrivo di un process di Poisson con rate 1 e sia $R_i$ la sequenza di variabili aleatorie i.i.d. e statisticamente indipendenti con i tempi di arrivo $\tau_i$. Se la serie
$\sum_{i=1}^{\infty} \tau_i^{-\frac{1}{\alpha}} R_i$
converge quasi ovunque, allora converge anche ad una variabile aleatoria $\alpha$-stabile.

La dimostrazione viene riportata in [GS94]

La proposizione precedente suggerisce che una variabile alteatoria stabile $X$ può essere rappresentata come una somma infinita. Devo fare alcune ipotesi aggiuntive su $\alpha$ e sulla variabile aleatoria $R_i$. Le ipotesi sono:

- $0 \leq \alpha \leq 2$ così da avere una distribuzione stabile non degenere
- la distribuzione di $R_i$ ha i momenti $\alpha$ assolutamente finiti e che sia simmetrica

Sempre nello stesso capitolo del libro menzionato viene riportato un risultato simile ma che usa le ulteriori ipotesi e dimostra come la stessa espressione precedente converga ad una distribuzione stabile e simmetrica.simmetrica

### Applicazioni

Posso usare questo "teorema di rappresentazione" per giustificare un metodo di stima molto più semplice rispetto al classico fit statistico della distribuzione. È prassi comune scomporre i segnali raccolti in somme di segnali più semplici, pensiamo alla decomposizione in basi trigonometriche con le serie di Fourier, oppure alla decomposizione in uno spazio ortonormale complesso proposto da [Zol86] con cui scompone distribuzioni stabili tramite FFT, ecc... Posso applicare lo stesso ragionamento anche qui, la variabile aleatoria ha una distribuzione non trattabile con facilità poiché molto complicata. Ammettendo un errore finito di rappresentazione, posso considerare i miei dati distribuiti secondo una distribuzione stabile ma, la rappresento come somma di Poisson indipendenti che sono molto più facili da trattare e stimare. Dunque al costo di un errore di approssimazione ottengo la trattabilità matematica, un ottimo compromesso...

# Analisi dei dati ottenuti dalla applicazione di controllo distribuito

Innanzitutto bisogna chiedersi perché è necessario parlare di controllo della rete e delle topologie. È necessario parlare di controllo ottimo della rete perché è la procedura che mi permette di garantire un'operazione corretta ed affidabile della rete. I problemi di stabilità della rete e la necessità di controllo nella rete, nasce dalla natura del mezzo fisico con cui interconnetto i vari device e dalla gestione dell'accesso al mezzo fisico. Il controllo della rete può essere fatto in più modi attraverso:

- gestione dei metodi di accesso del canale
- gestione dei traffici che generano un'evoluzione dello stato del sistema

La gestione del controllo della rete attraverso i metodi di accesso al canale è una macro categoria di cui non mi occupo. Per quanto riguarda il controllo della rete mi occupo del problema a livello E2E nell'ottima di controllo della congestione e controllo di flusso (primariamente attraverso TCP).

Espongo l'analisi dei dati e le conclusioni che ho tratto fino ad ora con le simulazioni.

## Ipotesi e statement del problema

Ipotizzo che la trasmissione dei dati soprattutto nel piano di controllo della mia rete segua una distribuzione heavy- tailed e che al limite converga ad una distribuzione stabile.

L'ipotesi è duplice:

- Il mio sistema genera dati nel piano di controllo con una distribuzione heavy-tailed
- Il processo che genera dati se osservato per un tempo sufficientemente lungo converge ad una distribuzione stabile

Il problema del primo punto è che fare un fit per una distribuzione heavy-tailed è estremamente complicato e i dettagli matematici sono particolarmente tecnici. Da un punto di vista numerico i test $\chi^2$ impone di conoscere la distribuzione sottostante così che possa verificare le "discrepanze" osservate siano sotto un certo valore. Questa distribuzione sottostante di probabilità non ce l'ho, non posso andare "a tentoni" provando tutte le varie distribuzioni fino a che no trovo quella con uno score minimo. Non posso neanche fare un fit con la log-verosimiglianza cercando con un MLE di trovare i vari parametri che compongono le distribuzioni, poiché i dati finiscono in regioni inammissibili dei metodi di ottimizzazione e quindi i valori calcolati sono spesso inaffidabili o con bound di incertezza così elevati che rendono inutili le stime prodotte.

Dunque per dimostrare il primo punto posso solo osservare che i dati di lunghezza dei pacchetti sulle reti dati (eg Internet) sono sempre stati classificati come distribuiti secondo pdf heavy-tailed. Il metodo usato solitamente è quello di fare una mean-excess analysis e vedere com'è fatta la distribuzione dei valori eccezionali. Un secondo metodo empirico che uso io in virtù del fatto che il processo sottostante è calssificabile come heavy-tailed, vado a cercare conferma della mia ipotesi guardando la media mobile a corto termine e poi allungando la lunghezza del filtro moving average. Se ottengo un con tempovariante e con un running average che mostra un trend questo è una condizione necessaria all'essere heavy tailed.

## Statement del problema

Ipotizzo che una rete di sensori in ambito mobile IoT abbia bisogno di una applicazione di rete per il controllo ottimo:

- del traffico
- della topologia
- delle congestioni
- degli sprechi in termini di banda/energia/slot-temporali

Il problema principale deriva dal fatto che il mezzo trasmissivo che sto usando è wireless ed usando un modello a livello fisico scopro che ci sono interferenze in ambienti in cui uso radio-propagazione. Solitamente questi problemi vengono chiamati come il problema del terminale nascosto e terminale scoperto. La soluzione proposta per questi due problemi "classici" è quella di usare un controllo centralizzato oppure l'uso di pacchetti RTS/CTS. Ognuno di questi metodi ha i suoi pregi e difetti. Mi concentro sul metodo del controllo centralizzato poiché è quello meno investigato di tutti in ambiente sensori mobile IoT.

### Esempio di non stazionierità del segnale traffico medio

Nel seguente dataset raccolto da un insieme di dispositivi con un'applicazione di controllo simile a quella usata nelle simulazioni del network forming. Dai risultati esposti con questo dataset ed il dataset delle simulazioni della tesi vediamo la conferma di ciò che viene riportato in [HCMSS02]. Nell'articolo viene dimostrato come traffici di tipo HTTP o simili siano distribuiti secondo pmf con heavy-tails. Nell' articolo menzionato viene data una modellazione dettagliata di quello che è la distribuzione del traffico. **Non** ripeto l'analisi fatta da loro, semplicemente è un'osservazione della ecdf e tramite osservazione affermano che il traffico segue le loro previsioni. Nella sezione dell'articolo citato viene data una dimostrazione delle proprietà delle pmf heavy-tailed e con una stima cercano di ottenere l'indice di coda che caratterizza il traffico. La mia analisi è molto più semplice in questo documento, mi limito a osservare che per il teorma limite centrale della probabilità io so che: $P[X_1 + X_2 + X_3 \dots] \sim N(\mu, \sigma)$ dove $\mu = \frac{1}{N}(\sum_{i=1}^{N} X_i)$ ciò che questo dice è che la somma di N variabili aleatorie convergerà ad una gaussiana con $\mu$ la media campionaria. Questo è vero in una moltitudine di casi, tra l'altro, la media campionaria è un'ottmo stimatore dela valore medio di una moltitudine di altre distribuzioni di probabilità. Perché concentrarsi tanto sulle distribuzioni heavy-tailed? Perché il teorma limite della probabilità **non** vale nel caso delle distribuzioni fat-tailed. Per convincersene basta osservare che per le funzioni regolarmente non variabili la "probabilità estrema" e cioè la probabilità che si avverino eventi "estremi" è molto più probabile che l'evento normale. Questo si può vedere osservando che una distribuzione fat-tailed se ne definiamo la *survival function* $F$ din una v.a. $X$ tale che $\overline{F}(X) = 1 - F(X) = Prob[X > x]$ allora indico con la seguente scrittura $\overline{F}(X) \sim x^{-\alpha}, x \to \infty$ per indicare che $\lim_{x\to\infty} \frac{\overline{F}(X)}{x^{-\alpha}} \to 1$. Qui $\alpha$ è il tail index cioè la velocità con cui la probabilità tende a un valore maggiore di $x$ con $x \to \infty$. Nelle distribuzioni per cui vale il teorma limite centrale questo tasso $\alpha$ è polinomiale, nelle distribuzioni fat-tailed è *subesponenziale*. Questo cosa comporta? Comporta il fatto che:
$P[X_1 + X_2 + X_3 \dots] \sim P[max(X_1, X_2, X_3, 0 \dots) > x], x \to \infty$
Tutto ciò che viene descritto qui è soltanto un *limiting behavior* cioè caratteristiche che si manifestano solo al limite. Il risultato di tutto è il seguente:

- Le medie storiche sono *inaffidabili* per la predizione
- Le differenze fra due predizioni successivamente sempre più "lontane" non decresce
- Il rapporto di due valori record successivi fra di loro non descesce
- La media di eventi in eccesso rispetto a una soglia aumenta all'aumentare della soglia
- L'incertezza dello stimatore statistico del comportamento medio di $n$ variabili aleatorie è simile a quelle originali
- I coefficienti delle regressioni che proviamo in molti casi danno risultati errati

Un errore molto comune e molto grave che viene commesso è quello di basare previsioni di dati di risk return nelle operazioni di trading finanziario usano la teoria "classica" invece che le distribuzioni fat-tailed e le loro cautele necessarie (si veda il libro del dott. Taleb al riguardo)

Ritornando ai dati, ciò che possiamo dire è che in un campione finito della popolazione, necessariamente i miei stimatori convergeranno a qualcosa di finito e le loro deviazioni standard, per quanto elevate, saranno finite. Ciò non dovrebbe trarre in inganno, poiché i miei stimatori mi possono dire che convergo a qualcosa quando in realtà sto inseguendo una quantità che diverge… Pensiamo ad esempio a stimare la media di una distribuzione di Cauchy. La distibuzione di Cauchy è una delle distribuzioni più "classiche" fra quelle fat-tailed in quanto ha una forma chiusa abbastanza usabile per le stime a massima verosimiglianza. La media della distribuzione di Cauchy *non* è definità poiché l'integrale non converge, lo stimatore MLE sì converge… A cosa converge? Ad un valore nullo, ma se non conosco la distribuzione sottostante io non saprei che convergo a un valore spazzatura, per questo motivo bisogna fare molta attenzione alle distibuzioni fat-tailed in quanto convergono a delle distribuzioni *stabili* piuttosto che a una gaussiana.

## Osservazione dei dati

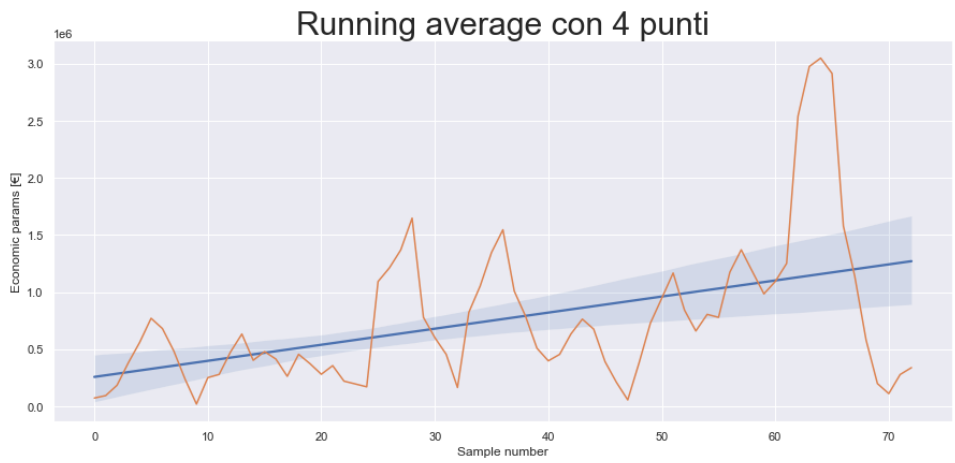
Il seguente dataset come anticipato mostra i "sintomi" di distribuzioni fat-tailed, **non** faccio un'analisi approfondita in questo documento ma mi limiterò a enunciare i risultati in modo qualitativo. In particolare:

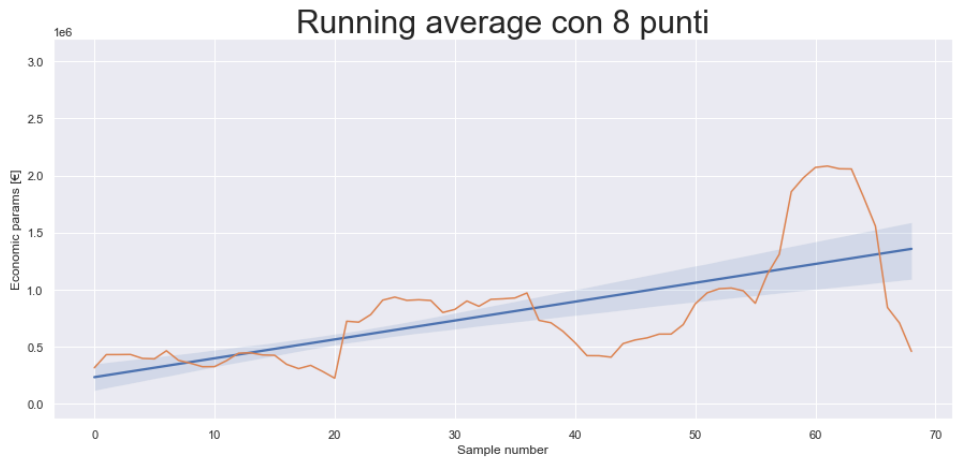
- Nel caso che la media mobile mostri chiaramente segnali di trend è plausibile ipotizzare una distribuzione fat-tailed
  Utilizzo un data set spesso usato per dimostrare queste proprietà, non l'ho raccolto io

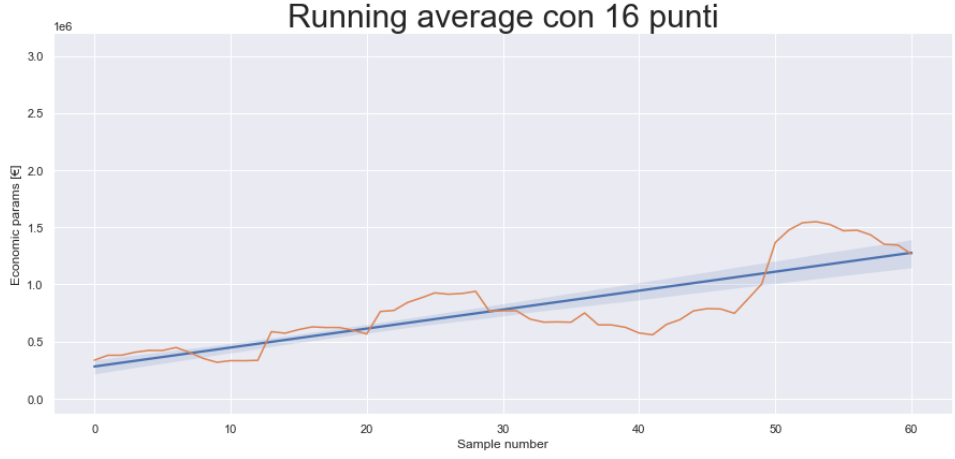
In realtà non è che nei miei dati ci sia un trend è solo che sto usando lo stimatore sbagliato. Nel dataset riportato si osserva proprio questo. Il dataset seguente è un dataset di parametri economici preso dal progetto GECON:

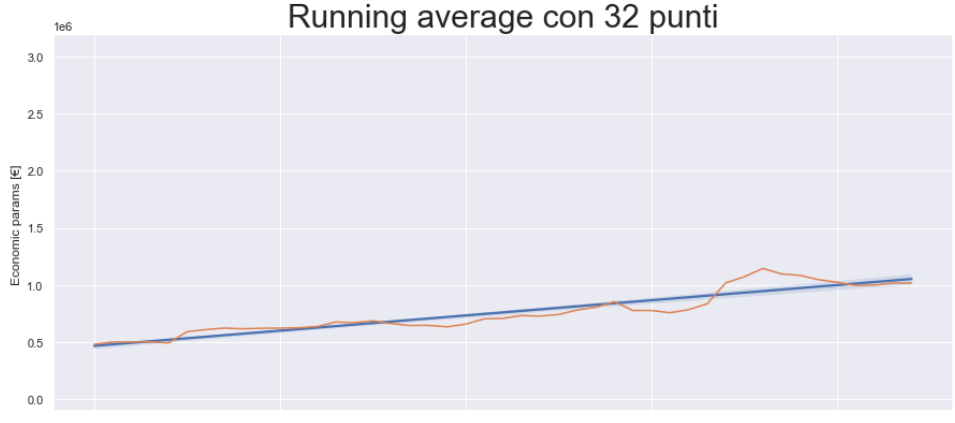
```
import copy
import numpy as np
import matplotlib as matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import pandas

ds = pandas.read_csv("../../data/GECON dataset.csv")
copyds = copy.deepcopy(ds)


def moving_average(x, w):
    return np.convolve(x, np.ones(w), 'valid') / w


windows = ["4", "8", "16", "32"]
sns.set()
fig, axes = plt.subplots(len(windows), 1, figsize=(15, 30), sharey=True)
x = 0
sns.set()
plt.suptitle("Dataset che a priori so essere fat-tailed",
             fontproperties=matplotlib.font_manager.FontProperties(size=50))
for win_length in windows:
    vsct = moving_average(copyds["Bps"], int(win_length))
    axes[x].set_title(f"Running average con {win_length} punti",
                      fontproperties=matplotlib.font_manager.FontProperties(size=30))
    df = pandas.DataFrame({"y": vsct, "x": list(range(len(vsct)))})
    sns.regplot(x="x", y="y", data=df, scatter=False, ax=axes[x])
    p = sns.lineplot(x="x", y="y", data=df, ax=axes[x])
    p.set(xlabel="Sample number", ylabel="Economic params [€]")
    x += 1
```

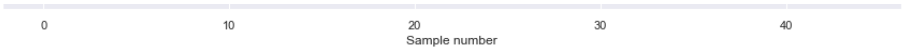
# Dataset che a priori so essere fat-tailed



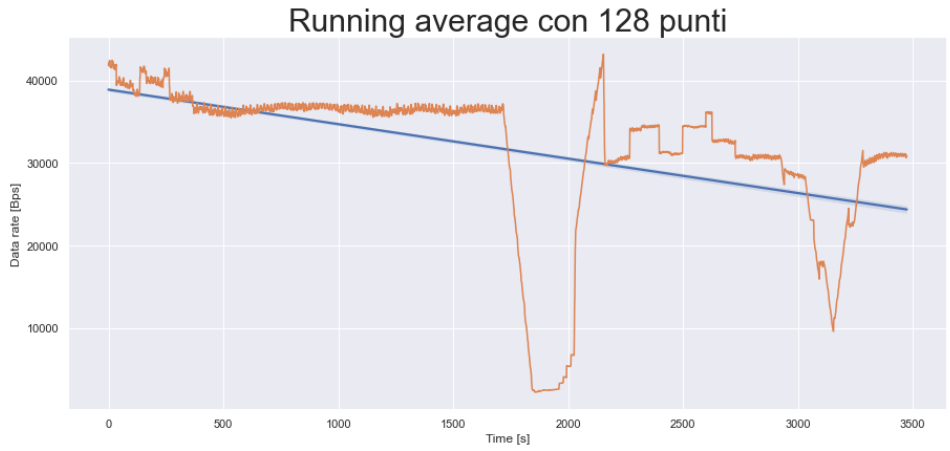



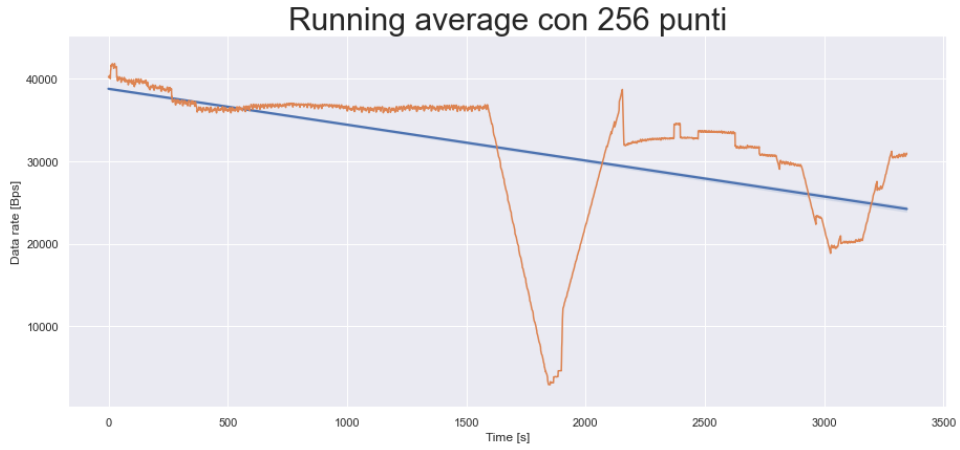


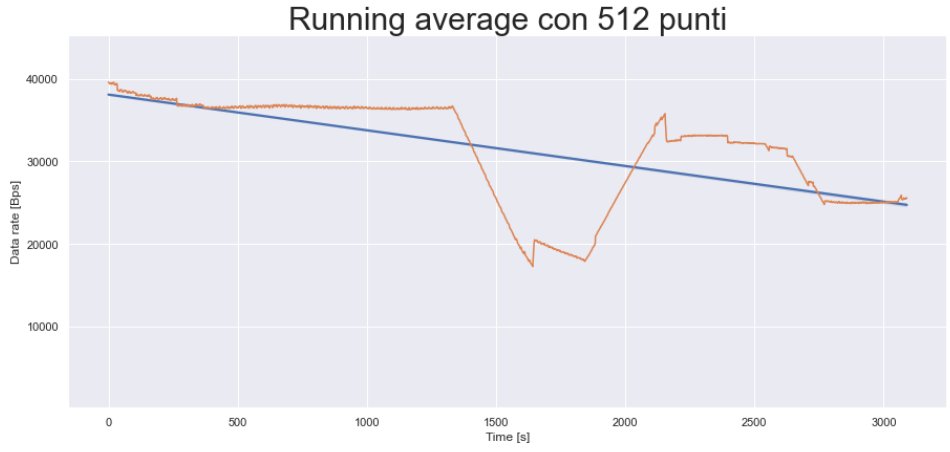

Dai plot sopra, nonostante abbia fatto una media mobile con una lunghezza della finestra notevole ripsetto al numero di campioni, questo non è risultato sufficiente a togliere i “trend” nei dati. Questa caratteristica che abbiamo osservato qualitativamente è un segnale di comportamento self-similar caratteristico nelle reti dati. Ripropongo la stessa visualizzazione con i dati della rete:

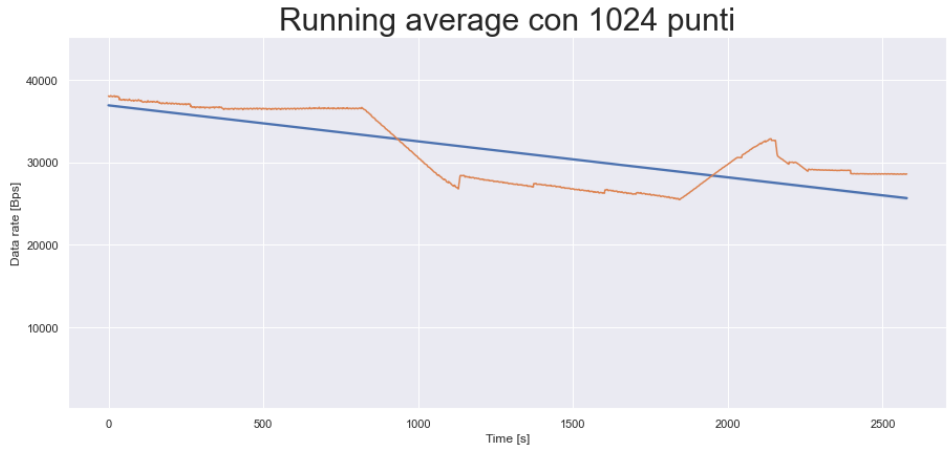
```
ds = pandas.read_csv("../../data/1 ora eserczio rete con piano lavoro REST.csv")
net_ds = copy.deepcopy(ds)
# indexNames = net_ds[net_ds['Bps'] < 1000].index
# net_ds.drop(indexNames, inplace=True)
windows = ["128", "256", "512", "1024"]
sns.set()
fig, axes = plt.subplots(len(windows), 1, figsize=(15, 30), sharey=True)
x = 0
sns.set()
plt.suptitle("Dataset raccolto dalle simulazioni",
             fontproperties=matplotlib.font_manager.FontProperties(size=50))
for win_length in windows:
    vsct = moving_average(net_ds["Bps"], int(win_length))
    axes[x].set_title(f"Running average con {win_length} punti",
                      fontproperties=matplotlib.font_manager.FontProperties(size=30))
    df = pandas.DataFrame({"y": vsct, "x": list(range(len(vsct)))})
    sns.regplot(x="x", y="y", data=df, scatter=False, ax=axes[x])
    p = sns.lineplot(x="x", y="y", data=df, ax=axes[x])
    p.set(xlabel="Time [s]", ylabel="Data rate [Bps]")
    x += 1
```

# Dataset raccolto dalle simulazioni



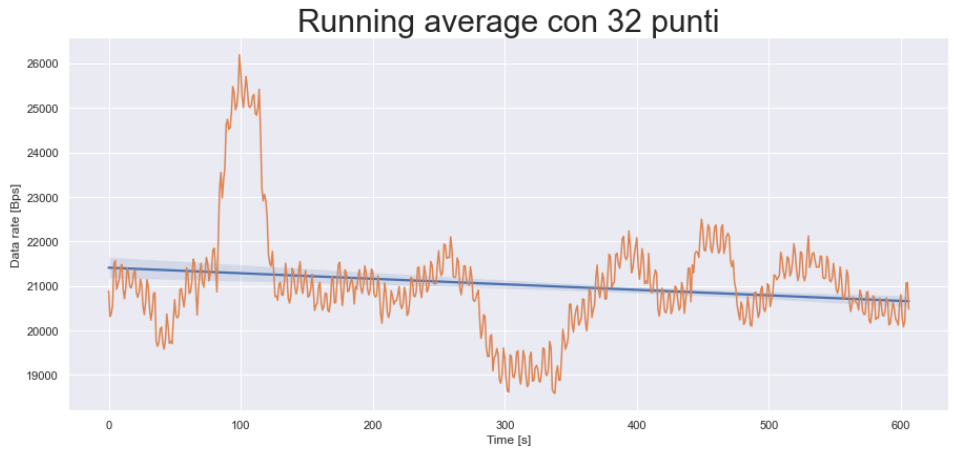



Il fatto che questi dati suggeriscano una presenza di distribuzioni di tipo fat-tailed induce a riflettere nel trarre le conclusioni. Una rete dati mostra un comportamento fat-tailed, dunque al limite tenderà ad una distribuzione *stable*. Questo che conseguenze ha nella pratica? Ragioniamo facendo uso di un ulteriore strumento: la teoria dei valori estremi. Secondo questa congettura nelle distribuzioni fat-tailed si vede un'applicazione del detto:"al peggio non c'è mai fine". Questo lo possiamo dire ragionando sui dati degli uragani negli USA i quali sono distribuiti in modo fat-tailed. Per quanto riguarda i dati distribuiti in modo fat-tailed e le loro proprietà applicati al dataset degli uragani, possiamo dire che usando la distribuzione degli eventi eventi in eccesso definita come $P[X > x|X > u]$ possiamo dire che in tutti i dataset si osserva che $E(P[X > x|X > u])$ per la linearità dell'operatore media possiamo dire che se fissata una soglia $u$ e volendo, fra tutti i valori oltre la soglia, cercare gli eventi estremi, possiamo dire che la loro media sarà sempre e comunque maggiore della soglia. Tutto ciò applicato agli uragani ci permette di dire che fissata una soglia di danni economici causati da questi uragani, presa una certa soglia, la media dei danni sarà maggiore di quella soglia. Da questo possiamo dire che guardando agli uragani non è possibile trovare ujna soglia $u$ tale che $E(P[X > x|X > u]) < u$ e questo è vero anche se spingo sempre più la soglia dei danni, al peggio non c'è mai fine...

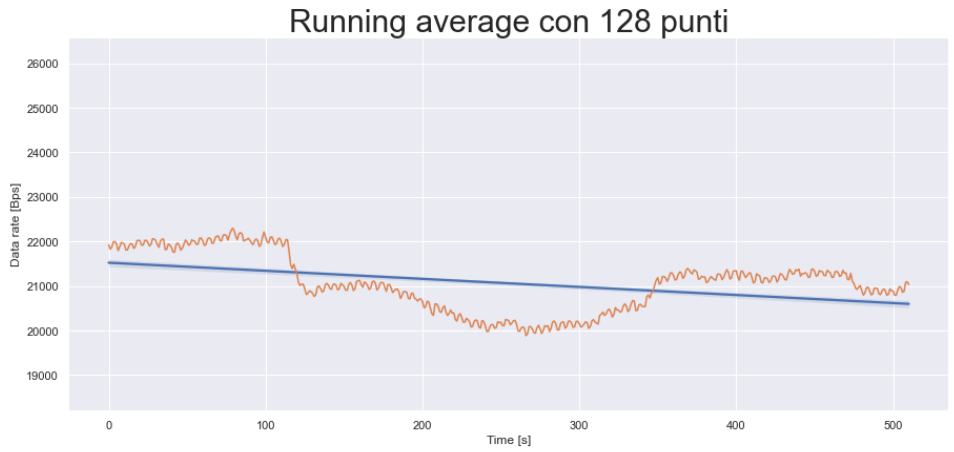
Questo principio vale per tutti i dati distribuiti secondo pmf fat-tailed, dunque anche per i dati che ho raccolto nella rete con controllo 1-hop. Anche le altre reti di sensori wifi generali si comportano in questo modo, dimostro la precedente con un precedente dataset raccolto in una rete wifi non infrastrutturata con un'applicazione di generazione dati UDP.

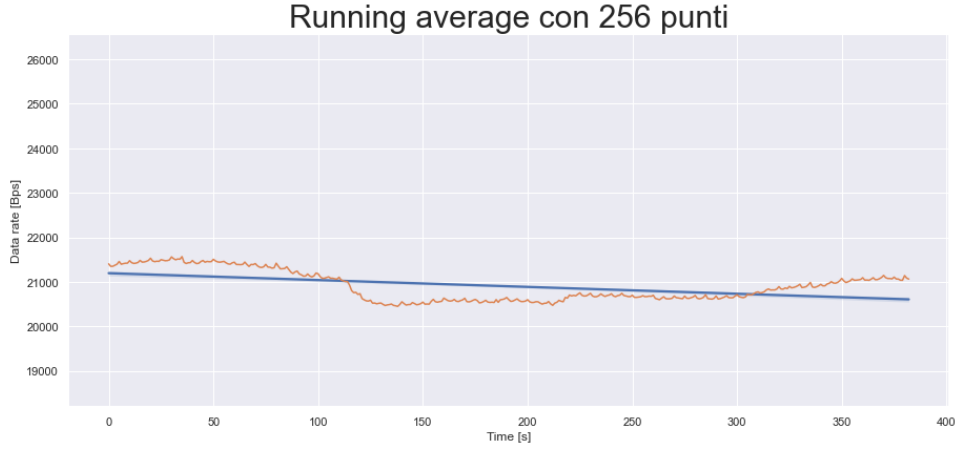
```
ds = pandas.read_csv("../../data/capture_wifi.csv")
net_ds = copy.deepcopy(ds)
# indexNames = net_ds[net_ds['Bps'] < 1000].index
# net_ds.drop(indexNames, inplace=True)
windows = ["32", "64", "128", "256"]
sns.set()
fig, axes = plt.subplots(len(windows), 1, figsize=(15, 30), sharey=True)
x = 0
sns.set()
plt.suptitle("Dataset wifi ad-hoc network",
             fontproperties=matplotlib.font_manager.FontProperties(size=50))
for win_length in windows:
    vsct = moving_average(net_ds["Bps"], int(win_length))
    axes[x].set_title(f"Running average con {win_length} punti",
                      fontproperties=matplotlib.font_manager.FontProperties(size=30))
    df = pandas.DataFrame({"y": vsct, "x": list(range(len(vsct)))})
    sns.regplot(x="x", y="y", data=df, scatter=False, ax=axes[x])
    p = sns.lineplot(x="x", y="y", data=df, ax=axes[x])
    p.set(xlabel="Time [s]", ylabel="Data rate [Bps]")
    x += 1
```

# Dataset wifi ad-hoc network





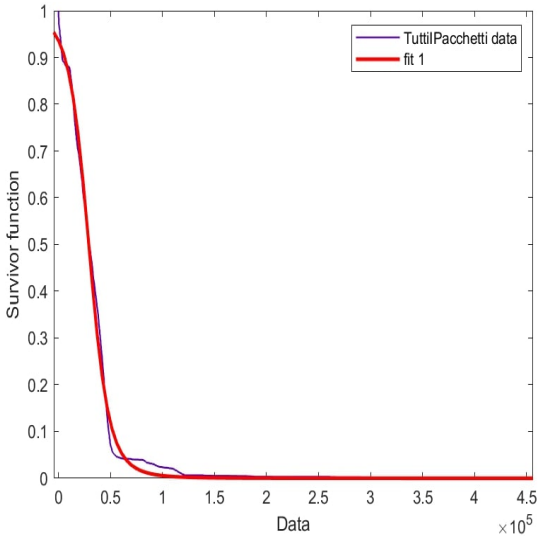


### Conlusioni analisi empirica dei dati

La cosa principale che capiamo da questi grafici e le formule esposte è il fatto che: il fatto che al peggio non ci sia mai fine vale anche nella rete. Questo lo capiamo in quanto i dataset appartengono a categorie di applicazioni generiche che sono distribuite, tutte quante, secondo distribuzioni fat-tailed. Dunque, anche per le reti vale la proprietà riguardante la excess distribution per la quale fissata una soglia di throughput lungo il control plane, la media dei throughput osservati nella excess distribution cioè tali per cui $X > u$ la media di questi throughput osservati sarà sempre maggiore della soglia. Peggio ancora, più spingo la soglia dei throughput e anche lì la media dei dati osservati sarà in ogni caso maggiore della soglia, in teoria anche se la soglia viene trascinata al limite... Quest'ultima cosa avviene ad esempio nei casi di broadcast storm.

Come dimostrazione finale mostro un fit fatto con la survivor function della distribuzione fat-tailed dei miei dati ed una distribuzione fat-tailed del tipo Pareto generalizzata




## Modello della rete

Adesso analizziamo una applicazione di controllo della rete, questo per mitigare le problematiche esposte prima attraverso la distribuzione dei dati.

Una prima soluzione è quella di identificare gli eventi estremi e risolvere quei casi. Ragionanando in questo modo possiamo pensare che l'evento catastrofico da risolvere siano i broadcast storm dovuti ai cicli nella rete.

```
soluzione: eliminiamo i cicli ed usiamo STP
```

Questa soluzione non tiene conto della stabilità della rete... È possibile dimostrare che una rete con UDP è modellabile dal punto di vista della propagazione dati come un sistema lineare e quindi posso fare un'analisi della stabilità. Ciò che si può dimostrare è che una rete con UDP è un sistema stabile ma che in generale ha caratteristiche di margine di fase molto fragili, basta poco per far "scoppiare" la rete. Per convicnersene basti pensare cagli attacchi "smurf" cioè di amplificazione, principalmente utilizzano protocolli trasporto che non prevedono controlli distribuiti sul carico immesso nella rete (ad esempio broadcast ping). Col protocollo TCP risolviamo il problema, grazie al protocollo e ai suoi algoritmi di gestione della congestione (livello IP) e del flusso (sliding window), non sono soluzioni definitive in quanto non effettuo un'ottimizzazione congiunta ma opero su due livelli diversi, ma è comunque un buon risultato.

Quest'ultima frase significa che nel piano di controllo di una rete di sensori SDN è necessario usare protocolli di trasmissione con qualche meccanismo di controllo della congestione e di controllo del flusso, anche basilari. Se queste precazioni non vengono intraprese, ci esponiamo al rischio di overflowing dei dispositivi. Questo perché nel piano di controllo ipotizzato nella simulazione, ho 1 hop di propagazione. Con unm modello del genere, nel caso peggiore trasmetterò un numero $o(n^2)$ pacchetti con $n$ numero di nodi. Il bound quadratico nel numero dei nodi si giustifica pensando che qualunque pacchetto da qualunque nodo verrà ritrasmesso in ogni collegamento. Il caso peggiore è che ci sia un messaggio per ogni collegamento e per ogni nodo, cioè $\frac{n(n-1)}{2} \sim n^2$ in pratica la situazione di una mesh.

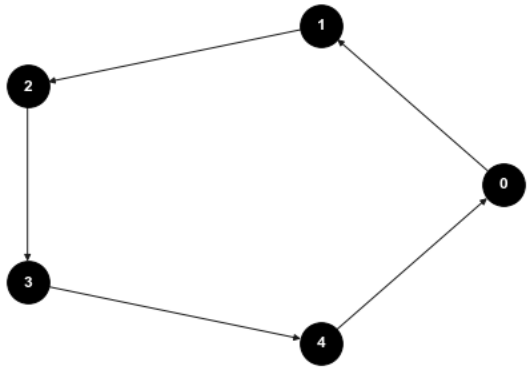
Nello studio dell'applicazione di controllo della rete, intanto inizio studiando gli algoritmi di controllo distribuiti. Principalmente gli algoritmi di controllo del carico su una rete sono di due tipi:

- Distribuito
- Centralizzato

Nel seguente paragrafo vediamo gli algoritmi distribuiti di controllo della rete e poi nell'ultimo paragrafo vediamo gli algoritmi centralizzati e come li ho implementato nel controller Ryu.

```
print("Topologia: ")
import matplotlib.pyplot as plt
import networkx as nx

G = nx.DiGraph()

for i in range(0, 5):
    G.add_node(i)

pos = nx.circular_layout(G)

G.add_edge(0, 1)
G.add_edge(1, 2)
G.add_edge(2, 3)
G.add_edge(3, 4)
G.add_edge(4, 0)

nx.draw(G, pos=pos, with_labels=True, font_weight='bold', node_size=1000,
node_color='black', font_color='white')
plt.show()
```

```
Topologia:
```




Questo è il modello della rete ad anello che uso per far vedere quali sono alcuni problemi che possono nascere. Supponiamo il traffico entri nel nodo $i$ e lasci la rete attraverso il nodo $[(i+1)modn]+1$. Indicando con $x_i$ il rate del traffico che il nodo $i$ offre alla rete ed indicando $x_i'$ il rate del traffico nel collegamento col nodo successivo allora abbiamo:

\begin{equation} \begin{cases} x_i = min(x_i, \frac{c_i}{x_i+x_{i-1}'})\ x_i'' = min(x_i', \frac{c_{i+1}}{x_i'+x_{i+1}}x_i') \end{cases}
\end{equation}

Per semplicità ipotizzimo $c_i = c$ ed $x_i = x \quad \forall i \quad in \quad N$. A questo punto possiamo dire che se $x \leq c/2$ allora non ci saranno perdite $x = x' = x''$ ed il throughput della rete sarà $nx$.
Altrimenti si può dire che:
$x' = \frac{cx}{x+x'}$ ed ottengo $x' = \frac{x}{2}(\sqrt{1+4\frac{c}{x}} - 1)$
ed inoltre sempre dalle equazioni precedenti so che
$x'' = \frac{cx'}{x+x'}$ ed ottengo $x'' = c - \frac{x}{2}(\sqrt{1+4\frac{c}{x}} - 1)$
uso l'approssimazione di Taylor al primo ordine ed ottengo $x'' = \frac{c^2}{x} + o(\frac{1}{x})$ dove $o(\frac{1}{x})$ denota tutti i termini di ordine superiore al secondo. Dunque si vede che $\lim_{x\to\infty} x'' = 0$ il rate dunque va a zero ed anche il throughput andrà a zero, questo è il *congestion collapse*. Ogni device dovrà limitare la quantità di traffico offerto alla rete, *ma come*?

## Efficienza ed equità

Suppondendo che ci sia qualche meccanismo per il controllo del traffico e che quindi le perdite siano trascurabili si ha che:
$x_n + x_i = c \quad \forall i \in [1, n-1]$ dunque il throughput $\theta = (n-1)(c-x_n) + x_n = c(n-1) - x_n(n-2)$
Dunque una streategia per aumentare il thorughput è quella di spingere il traffico offerto da $x_n \to 0$, ma questo non è molto equo nei suoi confronti...
C'è un altro modo per garantire un uso completo delle risorse ed equo.

## Allocazione ottima delle risorse

Per ottimizzare l'allocazione delle risorse di rete per tutti gli utenti, possiamo ragionare:

- in termini di ottimizzazione distribuita
- dell'equilibrio di Nash in cui la rete si troverà a regime
- secondo i principi di ottimalità di Wardrop

Per l'analisi del problema delle ipotesi e della soluzione algoritmica si veda il seguente notebook.

A questo punto ragionando dell'ottimizzazione distribuita introduco la funzione ulitità $U_i(x_i)$ che ogni device produce per ogni info che trasmette. Ciò che voglio è massimizzare l'utilità dell'intero sistema.
$\max_{x_i \geq 0} \sum_{i \in \overline{S}} U_i(x_i)$. Questo è molto diverso rispetto al concetto di system optimum descritto da Wardrop.

## Ipotesi 1

Per ogni $i \in \overline{S}$, $U_i(x_i)$ è una funzione monotona crescente, liscia e convessa.

Sia $x_i^*, i \in \overline{S}$ soluzione ottima del problema. Poiché per ipotesi la funzione è convessa e per le proprietà del punto ottimo si ha che $\nabla U = 0$ e quindi
$\sum_{i \in \overline{S}} \nabla U_i(x_i^*)(x_i - x_i^*) \leq 0$
da questo punto in poi ragiono con un'analisi del punto fisso $x_i^*$. Riorganizzo i termini e posso dire:
$\sum_{i \in \overline{S}} \nabla U_i(x_i^*) x_i^* \frac{x_i - x_i^*}{x_i^*} \leq 0$
qual'è l'interpretazione dei termini:

- $\frac{x_i - x_i^*}{x_i^*}$ è il cambio marginale di rate di trasmissione della sorgente $i$
- $\nabla U_i(x_i^*) x_i^*$ può essere interpetato come la *fairness* della sorgente

L'ultima equazione ci dice la somma pesata dei cambi di rate di trasmissione in ogni utente è meno di zero.

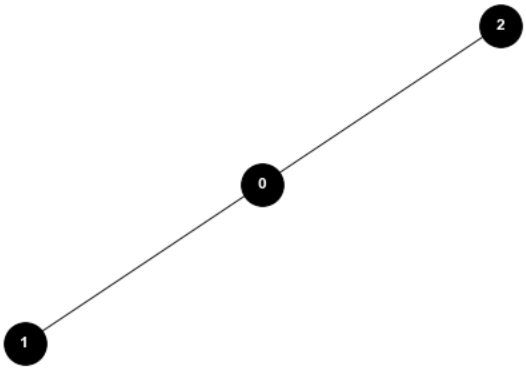
## Proportional fairness

Scegliamo per esempio $U_i(x_i) = w_i log x_i$ con $w_i \geq 0$. La disequazione variazionale diventa:
$\sum_{i \in \overline{S}} w_i \frac{x_i - x_i^*}{x_i^*} \leq 0$, dunque le variazioni di rate per ogni utente pesati per la loro "importanza" sono non positivi.

### Esempio

Sia data la rete in figura, chiamato anche **modello KP**

```
G = nx.Graph()
G = nx.Graph()

for i in range(0, 2):
    G.add_node(i)

G.add_edge(0, 1)
G.add_edge(0, 2)
nx.draw(G, with_labels=True, font_weight='bold', node_size=1000, node_color='black',
font_color='white')
plt.show()
```




Supponiamo di usare una allocazione *proporional fair* e cioè
$U_i = w_i log x_i$ in cui $w_i = 1 \; \forall i \in \overline{S}$ il problema diventa
$\max_{\mathbf{x} \geq 0} \quad log x_0 + log x_1 + log x_2$

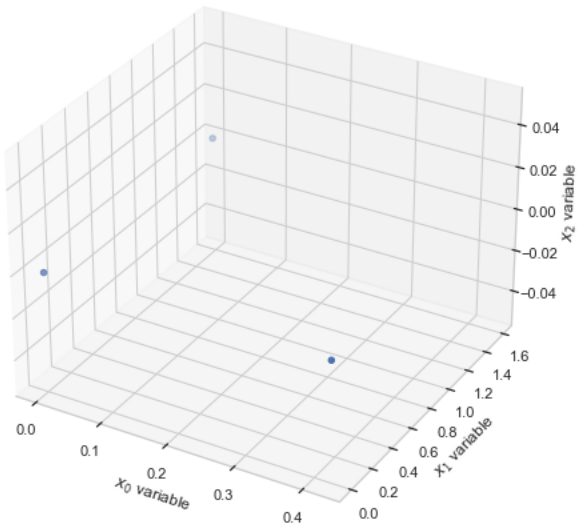
$st$

\begin{equation} \begin{cases} x_0 + x_1 \leq 2\ x_0 + x_2 \leq 1 \end{cases}
\end{equation}

Poiché il $\lim_{x \to 0} log x = -\infty$ è chiaro che la politica di allocazione del traffico non potrà asssegnare zero traffico ad un collegamento. Questo mi permette di considerare i vincoli $\mathbf{x} \geq 0$ ridondanti. Dunque volendo risolvere il problema tramite rilassamento lagrangiano posso rilassare i vincoli di capacità ottenendo:
$\mathbf{L}(\mathbf{x}, \lambda) = \quad log x_0 + log x_1 + log x_2 - \lambda_A (x_0 + x_1) - \lambda_B (x_0 + x_2)$
Ora cerco il punto stazionario ponendo $\frac{\partial L}{\partial x_i} = 0 \; \forall i \in \overline{S}$ ottengo:
$x_0 = \frac{1}{\lambda_A + \lambda_B}$ , $x_1 = \frac{1}{\lambda_A}$ , $x_2 = \frac{1}{\lambda_B}$
Questo unito al fatto che $x_0 + x_1 = 2$ e $x_0 + x_2 = 1$ ottengo
$\lambda_A = \frac{\sqrt{3}}{\sqrt{3}+1}$ e $\lambda_B = \sqrt{3}$
dunque \begin{equation} \begin{cases} x_0^* = \frac{\sqrt{3}+1}{2\sqrt{3}+3} \ x_1^* = \frac{\sqrt{3}+1}{\sqrt{3}} \ x_2^* = \frac{1}{\sqrt{3}} \end{cases}
\end{equation}

```
import seaborn as sns
from pylab import *
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

sns.set_style("whitegrid", {'axes.grid': False})

fig = plt.figure(figsize=(6, 6))

ax = Axes3D(fig)  # Method 1

x = [0, 0, (np.sqrt(3) + 1) / (3 + 2 * np.sqrt(3))]
y = [0, (np.sqrt(3) + 1) / (np.sqrt(3)), 0]
z = [1 / (np.sqrt(3)), 0, 0]

ax.scatter(x, y, marker='o')
ax.set_xlabel(r'$x_0$ variable')
ax.set_ylabel(r'$x_1$ variable')
ax.set_zlabel(r'$x_2$ variable')

plt.show()
```

```
C:\Users\DULLA\AppData\Local\Temp\ipykernel_26176\2092273623.py:10:
MatplotlibDeprecationWarning: Axes3D(fig) adding itself to the figure is deprecated since
3.4. Pass the keyword argument auto_add_to_figure=False and use fig.add_axes(ax) to
suppress this warning. The default value of auto_add_to_figure will change to False in
mpl3.5 and True values will no longer work in 3.6.  This is consistent with other Axes
classes.
  ax = Axes3D(fig)  # Method 1
```




Qui verifico con Wolfram Mathematica che il procedimento sia corretto:

```
max{log(x) + log(y) + log(z)|x + y<=1 ∧ x + z<=2}≈-0.954769 at (x, y, z)≈(0.42265, 0.577351, 1.57735)
```

## Algoritmi per il controllo della congestione

Posso osservare che il problema definito precedentemente è un problema convesso e i vincoli di capacità sono dei vincoli di simplesso. I vincoli di simplesso li ritroviamo anche nei problemi di ottimizzazione sparsa. Nei problemi di ottimizzazione sparsa uno dei metodi principali per risolvere questi programmi è l'utilizzo del metodo a *penalty function*. In pratica invece di considerare i vincoli di capacità come se fossero dei vincoli esterni, io li metto nella funzione obiettivo attraverso l'uso di una funzione penalità. Se procedo in questo modo ottengo:

$$max_{x\geq 0} \sum_{i\in\overline{S}} U_i(x_i) - \sum_{l\in\overline{L}} \int_0^{\sum_{i\in S_l} x_i} p_l(x)\,dx$$

La funzione $p_l(x)$ denota il *price* del collegamento, cioè il costo a cui vengo sottoposto per attraversare quel collegamento e $c_l$ è la sua capacità. Attenzione per i vincoli di conservazione in una rete di flusso a divergenza nulla bisogna fare attenzione al fatto che il *price* del link $l$ è funzione del tasso di arrivo *aggregato* cioè faccio un taglio attorno al nodo e sommo su tutti i flussi che attraversano quel taglio.

Nel libro di Srikant l'autore dimostra come le condizioni necessarie di ottimalità siano le seguenti: $U_i'(x_i) - \sum_{l\in L_i} p_l(\sum_{r\in S_l} x_r) = 0,\ i \in \overline{S}$ Dunque posso creare un algoritmo di controllo con un metodo del gradiente, ottenendo:
$\frac{dx_r}{dt} = k_i(x_i)(U_i'(x_i) - \sum_{l\in L_i} p_l(\sum_{r\in S_l} x_r))^+_{x_i}$, $i \in \overline{S}$ il simbolo $[\circ]^+$ indica la parte positiva, dunque useremo un gradiente proiettato. Questo algoritmo sempre nel libro di Srikant viene dimostrato come l'obiettivo sia una funzione di Lyapunov per il sistema, dunque qualunque traiettoria venga indicata dalla soluzione del mio controllore sarà asintoticamente stabile. Una verisione semplificata ed implementata di questo controllore è l'algoritmo di Kelly.

Si può ragionare in modo analogo anche con il problema duale. In questa versione ogni collegamento usa il proprio tasso di arrivo totale e con quello si calcola il prezzo di sé stesso. Questo algoritmo si può intereptare nel seguente modo: se il tasso totale di arrivi è superiore alla propria capacità, allora incrementa il proprio prezzo secondo una *barrier* function opportuna, altrimenti decrementa il prezzo. Perché parlare di algoritmi di ottimizzazione duali e primali nel contesto delle reti? Perché è giusto capire con che logica funzionano gli algoritmi ad oggi usati per la gestione delle congestioni e del flusso. In modo approssimato si possono classificare in 2 categorie:

- A livello di trasporto come nel TCP NewReno oppure Vegas i quali basano il traffico offerto su dei feedback disponibili
- A livello di rete attraverso meccanismi impliciti come AQM DropTail o RED

In prima approssimazione possiamo dire come l'algoritmo primale ha una legge dinamica per gestire il tasso di invio dalla sorgente ed una legge statica per generare il prezzo del collegamento, un po' come in TCP. Dall'altra parte possiamo dire che il problema duale usa una legge statica per gestire il tasso di invio della sorgente ed una legge dinamica (tempovariante) per assegnare il *price* al collegamento, similmente a come lavora AQM. Giusto per fare un esempio: nei meccanismi di AQM il prezzo ombra (il *price*) è il $p_l$ e nel metodo di Floyd e Jacobson la lunghezza della coda viene usata come stimatore del tasso di congestione del collegamento. In quel caso uso la versione duale dell'algoritmo per aggiustae il prezzo del collegamento.

Il problema di questi metodi è che sono computazionalmente molto dispensiosi, questo a causa del calcolo all'interno dell'algoritmo (vedi *steepset descent* nel metodo primale) oppure per il calcolo di uno stimatore per la congestione in un collegamento (vedi lo stimatore empirico lunghezza del buffer nel AQM).

## Algoritmi centralizzati: metodi SDN

Adesso ci concentriamo su gli algoritmi centralizzati. Questi metodi possono essere di due tipi:

- Centralizzati eseguiti su ogni device (ad esempio Djikstra con OSPF) ogni device manda in flooding *tutto*
- Centralizzati su un unico device, oggetto di questo paragrafo

Un'ultima postilla che faccio sul primo punto è la seguente osservazione: l'algoritmo OSPF con Djikstra è un ottimo algoritmo di label correcting. Ha una debolezza: ogni device ha bisogno dell'*intera* topologia e quindi ogni device deve trasmettere un LSA (link state advertisement) in ogni singolo collegamento non bloccato con STP. Ho quindi un numero di pacchetti che nel peggiore dei casi sarà $o(n^2)$. Dunque OSPF è il miglior algoritmo che abbiamo a disposizione ma è molto pesante...

Nel caso degli algoritmi centralizzati in questo documento essi vengono eseguiti tutti

# Albero ricoprente di costo minimo capacitato

## Abstract

Presento un algoritmo genetico con una codifica specializzata, inizializzazione aleatoria per ottimizzare topologie delle reti di telecomunicazione.Questo problema NP-hard è spesso così vincolato che un'inizializzazione aleatoria e algoritmi genetici standard generano spesso soluzioni inammissibili. Questa strategia di ottimizzazione può essere usata anche per altri problemi simili.

## Introduzione

Il problema CMST è un'estensione del MST (minimum spanning tree). Un problema che mi chiede di trovare quale struttura garantisca la connessione fra tutti i nodi con il costo minore. Qui considero un solo flusso d'informazioni (o merci), cioè single commodity. In generale queste informazioni scorrono su link di capacità finita da un nodo radice a dei nodi foglia che agiscono da "sink" delle informazioni, i consumatori. Il problema più spesso risolto nella pratica è quello del MST; dal quale deriviamo il protocollo di rete STP (spanning tree protocol) di livello link-local (ISO/OSI layer 2). Questa invece è una versione constrained (capacitata) MST. La formulazione proposta in questo progetto va in una direzione diversa, sappiamo che nel caso in cui ci sia un server centrale che spinge informazioni lungo la rete il limite di capacità dei link è ricavabile con un probelma di flusso massimo (risolvibile con ad esempio l'algoritmo di Ford&Fulkerson).

- Che dire del caso in cui io abbia decine di nodi collegati al server centrale tramite una dorsale, come valuto la resilienza di una tale rete?
- Come posso garantire che il minor numero di utenti si scolleghi dalla rete nel caso in cui la dorsale si interrompa?
- Per quale tipo di metrica è meglio ottimizzare, la posa in opera dell'infrastruttura o la latenza sulla linea?
- Che dire della ridondanza della rete?

Queste sono solo alcune delle domande che posso includere nel modello come "criteri" guida decisionali. Nel caso in esame prendo solo in considerazione le prime due domande della lista.

```
Vorrei limitare il numero di nodi che sono collegati alla radice attraverso un singolo collegamento (dorsale).
```

Un primo modo per risolvere è avere un albero "bilanciato", cioè con un numero di nodi circa uguale in ogni ramo della radice. Questo garantisce che, per esempio, se su $N$ collegamenti uscenti dalla radice se ne interrompesse uno, avrei $\frac{1}{N}$ utenti scollegati rispetto al totale. Lo stesso ragionamento lo posso ripetere ricorsivamente per ogni sottoalbero che si diparte dalla sorgente. Ciò che vorrei è limitare il numero massimo di utenti serviti da un certo collegamento, in modo da limitare il numero di utenti scollegati in caso di guasti su quello specifico collegamento.

# Formulazione del problema

Una rete di telecomunicazioni può essere modellata da un grafo $G = (N, A)$ con $N$ insieme dei nodi che qui rappresenta un cliente che si connette alla rete (telecomunicazioni, gas, in generale è una rete di flusso). L' insieme $A$ è l'insieme degli archi che sono attivi e che collegano i nodi nel grafo, in questo caso rappresentano linee attive ed utilizzabili. La rete si ipotizza usi collegamenti bidirezionali e quindi uso un grafo non orientato. Inoltre ipotizzo che non ci siano archi paralleli (ridondanti), questi in realtà si aggiungono per aumentare il data rate o l'affidabilità. Qui non vengono esaminati, anche se per questo algoritmo non cambia molto.

Impostiamo il seguente programma:

$$Min \sum_{(i,j)\in A} c_{ij}x_{ij}$$

$$\sum_{(j,i)\in A} x_{ij} - \sum_{(i,j)\in A} x_{ij} = b_j \quad \forall i \in N \qquad (1)$$

$$\sum_{(i,j)\in A} y_{ij} \leq 1 \quad \forall j \in N \setminus i \qquad (2)$$

$$\sum_{(i,j)\in S} x_{ij} \leq |S| - 1 \quad \forall S \in N \qquad (3)$$

$$x_{ij} \leq Q y_{ij} \quad \forall (i,j) \in A \qquad (4)$$

$$y_{ij} \in \{0,1\} \quad \forall (i,j) \in A \qquad (5)$$

$$x_{ij} \geq 0 \quad \forall (i,j) \in A \qquad (6)$$

$y_{ij}$ sono le variabili decisionali (creo un collegamento fra $i$ e $j$)

$x_{ij}$ sono le variabili di flusso

Questa variante single commodity del problema MST capacitato può essere modificata per accomodare altri tipi di flusso i quali non devono mescolarsi o che devono essere in generale sempre distinguibili (eg. traffico di controllo e traffico utenti) ottenendo la variante multicommodity.

## Descrizione dei vincoli

- Il vincolo $(1)$ è il vincolo di conservazione del flusso, che garantisce che ogni utente venga soddisfatto con la sua richiesta di pacchetti e che i pacchetti non vengano creati o distrutti dal nulla.
- Il vincolo $(2)$ è il vincolo che mi permette di selezionare un solo arco entrante nel generico nodo $i - esimo$
- Il vincolo $(4)$ è il vincolo di linking. Questo è il vincolo complicante, infatti se $y_{ij} = 0$ allora non scorre niente; invece se $y_{ij} = 1$ il flusso potrebbe non avere più limiti perché posso porre $Q \to \infty$ (o comunque un big-M). Nel mio caso invece pongo $Q = n$ con $n$ un numero opportuno, un upper bound sul numero di nodi serviti da un certo arco. Questo upper bound sarà tanto più stringente quanto più il ramo è vicino alla radice (più sono in alto e più traffico trasporto).
- Il vincolo $(3)$ è il vincolo di subtour elimination, ogni sottoinsieme di $k$ nodi deve avere al massimo $k - 1$ archi. Questo potrebbe essere sostituito da i vincoli di Miller-Tucker-Zemlin che sono in numero polinomiale.

## Complessità

Questo problema a causa dei vincoli di linking è in complessità esponenziale. Questi vincoli legano le variabili decisionali con quelle di flusso. Come primo tentativo potrei fare un rilassamento continuo della variabile binaria $y_{ij}$. Questo però, porterebbe a snaturare il vincolo logico ottenendo $y_{ij} \geq \frac{x_{ij}}{Q}$ che per $Q \to \infty$ che diverrebbe $y_{ij} \geq 0$ e cioè un vincolo inutile. Oppure posso tentare di rafforzare la formulazione prendendo un $Q$p leggermente sopra la capacità massima (dunque non un big-M) e sottraendo la richiesta $b_j$. Attenzione, è possibile costruire un piccolo esempio che dimostra come questo porta a sottostimare i costi della rete. Costruirsi un rilassamento adeguato non è banale, dunque la prima soluzione euristica che ha dato dei risultati "accettabili" è quella dell'algoritmo genetico di seguito descritto.

## Ipotesi operative

Ora proseguo con le ipotesi operative:

1. La locazione di ogni nodo è già fornita
2. I nodi sono perfettamente affidabili

3. Ogni collegamento è bidirezionale
4. Non ci sono collegamenti ridondanti
5. I collegamenti sono attivi o rotti
6. I fallimenti sono indipendenti fra loro
7. Non vengono esaminate riparazioni
8. La capacità $Q$ è fissata ad un valore intero

## Algoritmo genetico

Inizializzo con tutti gli import e i parametri interattivi.

```
import sys
sys.path.append(r"C:\Users\DULLA\PycharmProjects\Thesis_Data_Analisys")
from random import seed, randint
from time import process_time

import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np

from CMST_package.population_generator import draw_graph, generate_population,
TREE_LIKE_LAYOUT, breeding_mutation, \
self_mutation, prim, measure_branch

# MODIFICAMI!
NUM_NODES = 10
NUMB_GENER = 300
MESH_LIKE_LAYOUT = 0
HOF_SIZE = 3
MAXIMUM_FITNESS = 1000
POP_SIZE = 100
SELF_MUTATE_PROB = 0.3
CROSS_MUTATE_PROB = 0.7
ROOT = 5
CAPACITY = 4

#img = mpimg.imread('../artifacts/flowchart.PNG')
#plt.figure(figsize=(12, 12))
#plt.axis("off")

#imgplot = plt.imshow(img)
```

## Matrice di adiacenza aleatoria

Per testare l'algoritmo ho bisogno di avere dei grafi di prova. Questi li posso produrre a partire da dati reali (ad esempio posizioni GPS) oppure posso generarli sinteticamente. Ho scelto la seconda strada perché più veloce, non devo stare a trovare dati significati e prepararli.

```
ADJACENCY_MATRIX = [[0 for x in range(NUM_NODES)] for x in range(NUM_NODES)]


def random_adjacency_matrix(matrix):
    """
    Funzione che crea una matrice di adiacenza aleatoria per la simulazione
dell'algoritmo. Questa sezione è necessaria
    per testare come si comporta l'algoritmo al variare del grafo, dei pesi o degli archi
cioè di chi collego o no.

    QUI IL SEED È PREFISSATO, NON È REALMENTE ALEATORIO, SE LO SI VUOLE ALEATORIO,
TOGLIERE IL SEED
    #TODO: La firma è simil-C e fa schifo, restituisci la matrice

    :param matrix: matrice in cui appoggiare i risultati
    :return: matrice del grafo su cui si lavora (da implementare il return)
    """
    global MAXIMUM_FITNESS
    for col in range(len(ADJACENCY_MATRIX)):
        for row in range(col, len(ADJACENCY_MATRIX[col])):
            if col != row:
                value = randint(0, 100)
            else:
                value = sys.maxsize
            ADJACENCY_MATRIX[col][row] = value
            ADJACENCY_MATRIX[row][col] = value

            if row != col:
                if value > MAXIMUM_FITNESS:
                    MAXIMUM_FITNESS = value
                print('{:3}'.format(ADJACENCY_MATRIX[col][row]), end=" ")
                matrix[col][row] = ADJACENCY_MATRIX[col][row]
        print()
```

# Hall of fame

Questo notebook descrive il modulo deputato a salvare le migliori soluzioni mai viste e a raccoglierle fino alla presentazione finale.

È composto dai seguenti attributi:

1. individuals: coda degli individui migliori (vedi l'implementazione dell'individuo)
2. size: dimensione della HOF (quante soluzioni considero le migliori)
3. paths: lista in cui aggrego tutti gli MST migliori (utile per la visualizzazione)

Essendo l'oggetto Hall of Fame un semplice contenitore delle soluzioni migliori, la sua implementazione è piuttosto semplice. Tutto ruota attorno alla coda `individuals` e definisco solo le seguenti operazioni:

## Add

```
import inspect

from CMST_package.hall_of_fame import Hof
lines = inspect.getsource(Hof.add)
print(lines)
```

```
    def add(self, individual):
        """
        Aggiungi un nuovo membro alla hall of fame. Ho osservato una nuova soluzione e
devo valutare se è il caso di
        immetterlo nella hall of fame. Creo una coda ordinata (in ordine di fitness),
inserisco gli individui mai visti
        e quelli vecchi e poiché le dimensioni fra HOF e popolazione potrebbero essere
diverse, fintanto che una delle
        due non è vuota continua a immettere gli elementi dalla coda di appoggio alla HOF.

        :type individual: Individual
        :param individual: Individuo da aggiungere nella HOF
        :return:
        """
        temp = q.PriorityQueue()
        while not self.individuals.empty():
            old_ind = self.individuals.get()
            if individual == old_ind:
                continue
            temp.put(old_ind)
        # Se nella coda temporanea non c'è questo individuo allora aggiungi
        temp.put(individual)
        # Finché hai individui a disposizione e posto nella coda hall of fame, aggiungi in
ordine di fitness
        while not self.individuals.full() and not temp.empty():
            self.individuals.put(temp.get())
```

Possiamo dire che qui c'è semplicemente un algoritmo che popola una coda con priorità i cui l'ordine è basato sulla fitness. Si distringuono i casi in cui sia la coda sia vuota (prima inizializzazione) e il caso in cui sia piena (nel ciclo evolutivo). Per l'implementazione ed una spiegazione dettagliata del codice fare riferimento alla documentazione.

## Update

Questa funzioe è un po' più interessante in quanto qui si vede l'operazione di selezione dei migliori. Questa operazione viene fatta prendendo la polazione, ordinandola in base alla fitness e poi selezionando (con uno slicing) i primi `size` elementi e infine questi li rimetto nella coda delle migliori soluzioni. Con una strutta dati del genere sono sicuro che i primi elementi saranno i migliori, questo torna utile nella visualizzazione finale.

```
lines = inspect.getsource(Hof.update)
print(lines)
```

```
    def update(self, population):
        """
        Aggiorna la hall of fame poiché è cambiata la popolazione a seguito di un ciclo
evolutivo. Il procedimento è il
        seguente:
        1. Ordina la popolazione in base alla fitness
        2. Seleziona i migliori individui (i quali stanno in cima alla coda)
        3. Per ognuno di questi individui migliori rimettili nella hall of fame

        :type population: list
        :param population: popolazione su cui valutare la hall of fame
        :return:
        """
        population.sort()
        best_candidates = population[0:self.size]  # Slicing della lista e selezione dei
migliori (da 0 a N)
        for individual in best_candidates:
            self.add(individual)
```

## Codifica

Definisco come soluzione del mio problema una qualunque struttura connessa aciclica che garantisca la copertura di tutti i nodi della rete. Una tale soluzione è un albero ricoprente, quello a cui sono interessato è l'albero ricoprente di costo minimo soggetto al vincolo di capacità che dovrà essere rispettato per ogni singolo arco. Poiché con questo algoritmo genetico in uscita avrò un cromosoma (o una rappresentazione equivalente) la soluzione dunque è il cromosoma vincente di fra tutta la popolazione. A questo proposito devo rappresentare:

1. Cosa è un cromosoma
2. Come è fatto un cromosoma

Savic e Walter hanno usato stringhe di interi a lunghezza variabile per rappresentare una rete di distribuzione idrica.

Consultare il seguente notebook riguardante l'individuo soluzione e la sua codifica.

# Codifica dell'individuo

Nel rappresentare una soluzione bisogna usare una struttura a grafo, più precisamente uno spanning tree con il costo globalmente minore e che rispetti il vincolo di capacità.

## Descrizione strutture dati

Per fare ciò uso l'oggetto `individual` che rappresenta un individuo della popolazione. Ogni oggetto contiene al suo interno:

- **tree** che è la lista di tuple che implementa l'albero
- **fitness** che rappresenta la fitness della soluzione
- **optim_type** flag che indica se sto minimizzando o massimizzando
- **genotype** lista di interi a lunghezza variabile che rappresenta la rete

## Aspetti salienti

La scopo principale dell'oggetto `individual` è quello di rappresentarmi la rete in modo maneggiabile. Per farlo creo la rete e la memorizzo nell'attributo *genotype*, così che ogni individuo sia (all'inizio) una rete completa e poi a fine elaborazione un grafo (sperabilmente) soluzione. Ogni link possibile gli viene assegnato un intero che segnala la presenza dell'arco. Non solo questo intero segnala la presenza dell'arco ma rappresenta anche il peso dell'arco stesso nella lista *ordinata*. Ogni lista può definire in modo univoco quali collegamenti esistono e quali no.

Mi appresto a dimostrare la seguente proposizione: la rappresentazione genoma <-> grafo è biunivoca e corretta. Questa proposizione implica che è possibile partire da una rappresentazione (compatibile con le librerie usate) sotto forma di grafo, ed arrivare ad una rappresentazione sotto forma di genoma (cioè lista di interi a lunghezza variabile) *e viceversa*.

### dimostrazione

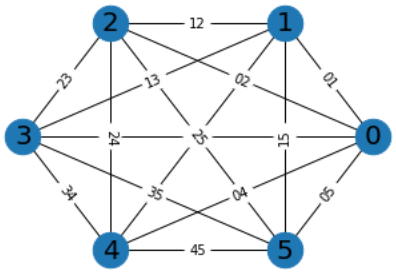
```
import numpy as np
from CMST_package.individual import Individual
from CMST_package.population_generator import draw_graph
import networkx as nx
```

Costruisco il grafo assegnando come peso (ed etichetta identificativa) la coppia nodo-sorgente/nodo-destinatario dunque $C_{ij} = sorgente|destinazione$

```
G = nx.complete_graph(6)
G[0][1]['weight'] = "01"
G[0][2]['weight'] = "02"
G[0][3]['weight'] = "03"
G[0][4]['weight'] = "04"
G[0][5]['weight'] = "05"
G[1][2]['weight'] = "12"
G[1][3]['weight'] = "13"
G[1][4]['weight'] = "14"
G[1][5]['weight'] = "15"
G[2][3]['weight'] = "23"
G[2][4]['weight'] = "24"
G[2][5]['weight'] = "25"
G[3][4]['weight'] = "34"
G[3][5]['weight'] = "35"
G[4][5]['weight'] = "45"
```

Creo e visualizzo il grafo full mesh con $n * (n-1)$ archi, dove $n$ è il numero di nodi nella rete, in questo caso 6. per cambiare numero nodi, modificare la sezione precedente.

```
draw_graph(G, 0)
```




```
<Figure size 864x864 with 0 Axes>
```

A questo punto genero la lista di adiacenza in modo da facilitare la conversione in lista normale. Segue poi la visualizzazione della lista di adiacenza.

```
print("Matrice di adiacenza: ")
adj_list = nx.generate_adjlist(G)

string = ""
i = 0
for arc in adj_list:
    print("Il nodo {0} è connesso con i seguenti nodi: \t{1}".format(i, arc))
    string += arc
    i += 1
```

```
Matrice di adiacenza:
Il nodo 0 è connesso con i seguenti nodi:       0 1 2 3 4 5
Il nodo 1 è connesso con i seguenti nodi:       1 2 3 4 5
Il nodo 2 è connesso con i seguenti nodi:       2 3 4 5
Il nodo 3 è connesso con i seguenti nodi:       3 4 5
Il nodo 4 è connesso con i seguenti nodi:       4 5
Il nodo 5 è connesso con i seguenti nodi:       5
```

Da notare come la matrice sia triangolare, questo poiché il grafo è *non* orientato e quindi la lista di adiacenza stampata in forma matriciale non riporta gli archi in verso opposto. Attenzione alla prima colonna, quelli sono tutti collegamenti loopback, nella matrice di adiacenza sarebbero sulla diagonale. In questo caso non vengono considerati e in seguito verranno scartati.

Ora proseguo con il mostrare come convertire questa rappresentazione lista-adiacenza in genoma in step successivi: 1. Reshaping della lista di adiacenza in un vettore 1-d

```
lista = [item for item in string if item != ' ']
print("La lista srotolata in formato simil-genoma è: \n\t{0}".format(lista))
```

```
La lista srotolata in formato simil-genoma è:
	['0', '1', '2', '3', '4', '5', '1', '2', '3', '4', '5', '2', '3', '4', '5', '3',
'4', '5', '4', '5', '5']
```

```
2. A questo punto creo un nuovo individuo di prova e gli assegno la lista di adiacenza srotolata come genoma.
```

```
ind = Individual(size=10, min=1, iter_range=256)
ind.genotype = lista
print(ind.genotype)
```

```
['0', '1', '2', '3', '4', '5', '1', '2', '3', '4', '5', '2', '3', '4', '5', '3', '4', '5',
'4', '5', '5']
```

Qui vediamo la correttezza dell'operazione in un verso, da grafo l'ho trasformato in genoma.

Adesso il viceversa. Partendo dallo steso individuo voglio, partendo dal suo genoma, ottenere il grafo originario:

```
1. Faccio il reshaping del genoma in lista di adiacenza (in forma matriciale), attenzione l'etichetta dell'arco è composta da
_sorgente|destinazione|index_
```

```
matrix = np.array(ind.reshape2matrix(NUM_NODES=6))
n = matrix.shape[0]
matrix[range(n), range(n)] = "0"
print(matrix)
```

```
[['0' '104' '204' '305' '402' '505']
 ['014' '0' '215' '311' '413' '512']
 ['024' '125' '0' '322' '424' '523']
 ['035' '131' '232' '0' '435' '534']
 ['042' '143' '244' '345' '0' '545']
 ['055' '152' '253' '354' '455' '0']]
```

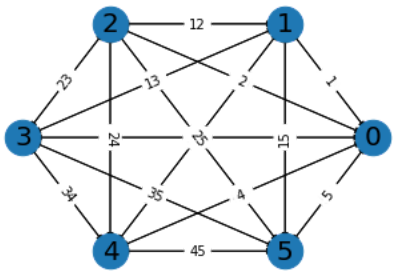
2. Creo una nuova matrice dei risultati eliminando la diagonale (in realtà la pongo a zero per problemi di incompatibilità fra librerie)

```
shape = matrix.shape
result = np.zeros(shape, dtype=int)
for x in range(0, shape[0]):
    for y in range(0, shape[1]):
        result[x, y] = str(matrix[x, y])[0:2]
print(result)
```

```
[[ 0 10 20 30 40 50]
 [ 1  0 21 31 41 51]
 [ 2 12  0 32 42 52]
 [ 3 13 23  0 43 53]
 [ 4 14 24 34  0 54]
 [ 5 15 25 35 45  0]]
```

3. Visualizzazione del grafo a partire dalla lista di adiacenza, mi aspetto sia lo stesso grafo visualizzato all'inizio.

```
g = nx.from_numpy_matrix(result, create_using=nx.DiGraph)
draw_graph(g, layout=0)
```




```
<Figure size 864x864 with 0 Axes>
```

A meno di alcuni difetti dovuti alla gestione Python delle stringhe, i grafi sono gli stessi. In pratica `01` non è possibile visualizzarlo poiché diventa `1` dato che viene trattato come un intero e non un'etichetta.

## Reshaping function

Per riarrotolare il genoma dell'individuo devo tenere conto del fatto che il genoma è una versione undimensionale della matrice di adiacenza e che la matrice dei costi è una versione bidimensionale della lista di adiacenza. Sono entrambe rappresentazioni dei costi ma con forme (dimensioni) diverse. La sezione di codice che esegue l'operazione di riarrotolamento è la seguente:

```
python
    L = max(row, col)                       # ['0', '1', '2', '3', '4', '5', '1', '2', '3', '4'
    S = min(row, col)                       # 1° step seleziono 0 poi con l'indice L scorro verso
    index = L * (L - 1) / 2                 # destra, ottenendo (0, 1) -> (0,2) e così via
    index += S - 1                          # uso l'indice _index_ come terza cifra per far capire
    index -= L                              # a che passaggio ero quando ho ricostruito quell'arco
    matrix[col][row] = genome[row]          # sommo (o concateno) i singoli geni del genotipo.
    matrix[col][row] += genome[col]         # nella diagonale avrò sys.maxsize, cioè un peso infinito
    matrix[col][row] += genome[int(index)]  # poiché non voglio archi loopback
```

## Descrizione dell'algoritmo

Per rendere la ricerca efficiente la popolazione iniziale consiste di reti a mesh rappresentate dall'individuo con genoma composto da $\frac{n*(n-1)}{2}$ interi. In sostanza tratto una popolazione come una lista di individui. Per una descrizione delle funzioni usate per l'evoluzione della popolazione e la fitness dell'individuo consultare il file riguadante la popolazione.

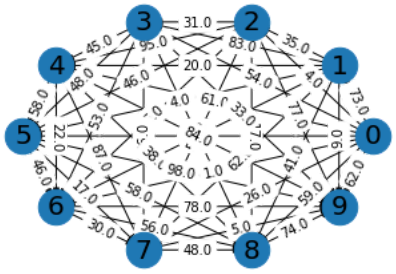
1. Setup dell'algoritmo

```
* Settaggio dei parametri
* Matrice `ADJACENCY_MATRIX` come matrice iniziale in cui effetturae gli step intermedi (per non sporcare quella buona finale)
```

1. Inizializzazione dell'algoritmo, creo la matrice di adiacenza aleatoria e la visualizzo

```
from CMST_package.hall_of_fame import Hof

seed(10)
Mat = np.zeros((NUM_NODES, NUM_NODES))
random_adjacency_matrix(Mat)
g = nx.from_numpy_matrix(Mat, create_using=nx.DiGraph)
draw_graph(g, MESH_LIKE_LAYOUT)
```

```
 73   4  54  61  73   1  26  59  62
 35  83  20   4  66  62  41   9
 31  95  46   5  53  17  77
 45  48  53  36  86  33
 58  22  87  38  84
 46  17  58  98
 30  56  78
 48   5
 74
```




```
<Figure size 864x864 with 0 Axes>
```

1. Inizializzo la Hall of Fame per memorizzare le migliori soluzioni

```
hall_of_fame = Hof(HOF_SIZE)
```

1. Genero una popolazione con la quale lanciare l'algoritmo

```
genome = generate_population(POP_SIZE + (POP_SIZE % 2), int(NUM_NODES * (NUM_NODES + 1) /
2))
```

1. Valutazione della fitness di ogni individuo della popolazione e modifica dei parametri degli individui (riporto il codice per comodità)

```
def fitness_evaluation(individuals):
    for ind in individuals:
        matrix = ind.reshape2matrix(NUM_NODES)

        # Apply Prim's Algorithm to get Tree
        tree = prim(matrix)

        # Depth first search starting from ROOT
        over_capacity = False
        for edge in tree:
            temp_weight = 0
            if edge[0] == ROOT:
                temp_weight = measure_branch(tree, edge[1], ROOT)
            elif edge[1] == ROOT:
                temp_weight = measure_branch(tree, edge[0], ROOT)
            if temp_weight > CAPACITY:
                ind.tree = tree
                ind.fitness = MAXIMUM_FITNESS
                over_capacity = True

        # Get weight of converted representation if under capacity
        if not over_capacity:
            weight = 0
            for edge in tree:
                (col, row) = edge
                weight += ADJACENCY_MATRIX[col - 1][row - 1]
            ind.tree = tree
            ind.fitness = weight


fitness_evaluation(genome)
```

1. Inizio a popolare la Hall of Fame e visualizzare le migliori soluzioni intermedie

```
hall_of_fame.update(genome)
print(hall_of_fame)
```

```
Fitness: 328
Tree: [(1, 5), (1, 10), (5, 7), (5, 2), (2, 9), (7, 6), (1, 8), (7, 3), (5, 4)]

Fitness: 374
Tree: [(1, 7), (7, 5), (5, 2), (5, 3), (7, 8), (7, 9), (2, 10), (2, 4), (5, 6)]

Fitness: 417
Tree: [(1, 5), (5, 6), (5, 3), (5, 2), (5, 8), (2, 10), (1, 7), (2, 9), (5, 4)]
```

1. Loop evolutivo * Mutazione crossover * Mutazione singola dalla popolazione in uscita dal crossover * Valutazione della fitness * Update della Hall of Fame

```
start = process_time()

for cur_gen in range(NUMB_GENER):
    new_chromosome = breeding_mutation(genome, CROSS_MUTATE_PROB)
    genome = self_mutation(new_chromosome, SELF_MUTATE_PROB)
    fitness_evaluation(genome)
    hall_of_fame.update(genome)

stop = process_time()
```

1. Visualizzazione dei risultati

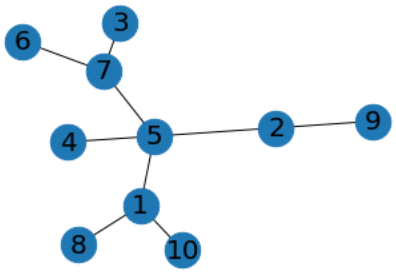
```
print(hall_of_fame)
print("Time: ", stop - start)
print("ROOT: ", ROOT)
print("CAPACITY: ", CAPACITY)
```

```
Fitness: 335
Tree: [(1, 5), (1, 8), (8, 6), (8, 2), (5, 7), (5, 4), (5, 9), (4, 3), (4, 10)]

Fitness: 391
Tree: [(1, 5), (5, 6), (6, 7), (1, 8), (5, 9), (1, 10), (5, 4), (6, 3), (10, 2)]

Fitness: 419
Tree: [(1, 9), (9, 3), (3, 5), (5, 4), (5, 2), (5, 7), (4, 8), (3, 10), (4, 6)]


Time:  6.5
ROOT:  5
CAPACITY:  4
```

1. Plot di alcune soluzioni fra le migliori

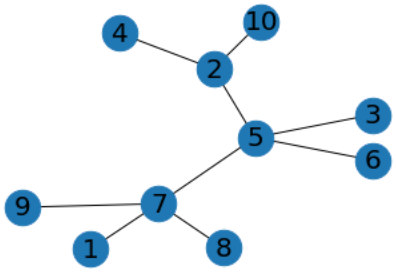
```python
for candidate in hall_of_fame.paths:
    candidate = [item for item in candidate if item != (0, 0)]
    print(candidate)
    G = nx.Graph()
    G.add_edges_from(candidate)
    draw_graph(G, TREE_LIKE_LAYOUT)
```

```
[(1, 5), (1, 10), (5, 7), (5, 2), (2, 9), (7, 6), (1, 8), (7, 3), (5, 4)]
```



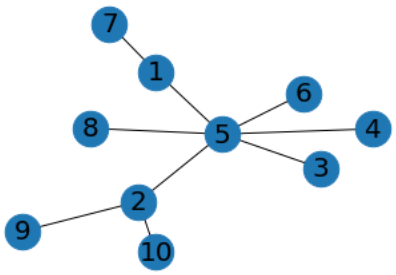

```
<Figure size 864x864 with 0 Axes>
```

```
[(1, 7), (7, 5), (5, 2), (5, 3), (7, 8), (7, 9), (2, 10), (2, 4), (5, 6)]
```



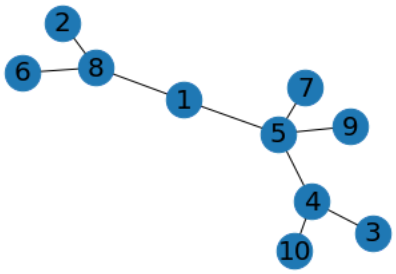

```
<Figure size 864x864 with 0 Axes>
```

```
[(1, 5), (5, 6), (5, 3), (5, 2), (5, 8), (2, 10), (1, 7), (2, 9), (5, 4)]
```



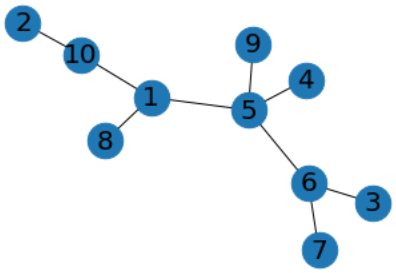

```
<Figure size 864x864 with 0 Axes>
```

```
[(1, 5), (1, 8), (8, 6), (8, 2), (5, 7), (5, 4), (5, 9), (4, 3), (4, 10)]
```



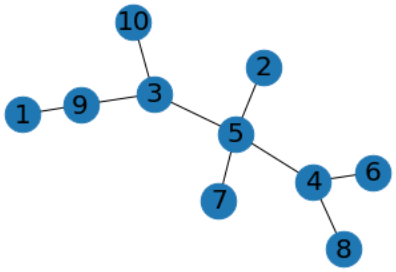

```
<Figure size 864x864 with 0 Axes>
```

```
[(1, 5), (5, 6), (6, 7), (1, 8), (5, 9), (1, 10), (5, 4), (6, 3), (10, 2)]
```




```
<Figure size 864x864 with 0 Axes>
```

```
[(1, 9), (9, 3), (3, 5), (5, 4), (5, 2), (5, 7), (4, 8), (3, 10), (4, 6)]
```




```
<Figure size 864x864 with 0 Axes>
```

## Conclusioni

Da questi esperimenti eseguiti si vede come la fitness delle varie soluzioni sia piuttosto vicina (in termini percentuali) a quella della miglior soluzione. Ciò vuol dire che se la prima soluzione è molto buona allora anche le altre saranno un pochino peggio ma comunque usabili. Il problema è quando la miglior soluzione ha una fitness vicino al valore `MAXIMUM_FITNESS`, in tal caso è molto probabile trovare qualche soluzione che violi i vincoli o che abbia caratteristiche pessime. In molti casi (verificabile lanciando più volte l'algoritmo) si scopre che le soluzioni con fitness più bassa (e quindi più desiderabili) di solito sono costituite da alberi all'incirca bilanciati. Questo conferma l'intuizione iniziale descritta nella discussione delle possibili modalità di soluzione tramite rilassamento. Un'ultima nota, questo algoritmo essendo un'euristica fornisce soluzioni che non posso sapere se sono ottime, quindi non dovrebbero guidare decisioni sensibili/strategiche ma solo fornire un'indicazione di massima.

## Appendice

# Generatore della popolazione

Questo modulo contiene tutte le funzioni necessarie all'algoritmo genetico. Non prevedo un oggetto "problema" con cui agganciare tutti i metodi o gli attributi ma sono distribuiti nel file.

## Funzione fitness

Definisco la fitness come il costo associato alla soluzione albero nel seguente modo:

$\sum_{(i,j)\in A} c_{ij}x_{ij} + \delta(c_{max}(R(\mathbf{x}) - R_0)^2)$

$\delta = \begin{cases} 0 & R(\mathbf{x}) < R_0 \\ 1 & R(\mathbf{x}) \geq R_0 \end{cases}$

$c_{max} = max(c_{ij})$ $R_0$ è il target di capacità massima che voglio per ogni arco

Sia dato un grafo $G = (N, A)$ e un albero di copertura $T \subset A$. Definiamo la funzione $R(\mathbf{x}) : A \to \Re$ con $\mathbf{x} \in T$ vettore di archi, come la funzione di conteggio dei nodi che vengono serviti da un certo arco $x_{ij}$, gli archi appartengono tutti all'albero $T$ sotto esame. Questa funzione mi dice qual è il numero totale di nodi che un certo arco sta servendo.

Questa formulazione può essere vista come una formulazione rilassata di quella originale. In particolare ho rilassato con la variabile indicatore il vincolo del rispetto della capacità. Come descritto sopra ho usato una funzione di accumulazione $R(\mathbf{x})$ e non più la formulazione classica basata su archi.

Poiché nella formulazione originale i vincoli accoppianti erano proprio quelli di capacità e sono sempre questi che rendono il problema NP-hard, è naturale lavorare principalmente su questo vincoli e poi pensare all'interezza delle variabili (si veda la formulazione nel file principale).

Se considerassi $\delta$ come una sorta di moltiplicatore di Lagrange frutto del rilassamento (col quale penalizzo il non rispetto del vincolo $R(\mathbf{x}) < R_0$) allora ha senso inserire il termine $\delta(c_{max}(R(\mathbf{x}) - R_0)^2)$ come penalità nella funzione fitness.

## Funzione di conteggio $R(\mathbf{x})$

Per il conteggio si usa la seguente funzione `measure_branch`, si riporta di seguitio il codice per comodità:

```
import inspect
from random import random, randint

import networkx as nx
import numpy as np

from CMST_package.individual import Individual
from CMST_package.population_generator import measure_branch, draw_graph, self_mutation

NUM_NODES = 7
CAPACITY = 4
MAXIMUM_FITNESS = 1000
POP_SIZE = 10
TOURNAMENT_SIZE = 2
breeding_mutation_pb = 0.3

lines = inspect.getsource(measure_branch)
print(lines)
```

```
def measure_branch(tree, parent, prev):
    """
    Questa funzione conta in modo ricorsivo quanti nodi ci sono in un certo
sottoalbero/brach e mi accumula il tutto.

    :param tree:
    :param parent:
    :param prev:
    :return:
    """
    weight = 0
    for edge in tree:
        if edge[0] == parent and edge[1] != prev:
            weight += measure_branch(tree, edge[1], parent)
        elif edge[0] != prev and edge[1] == parent:
            weight += measure_branch(tree, edge[0], parent)
    return weight + 1
```

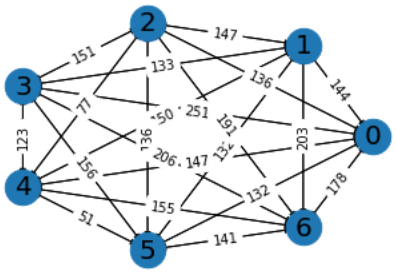
La funzione sopra riportata esegue in modo ricorsivo una ricerca simile allo stile breadth-first in modo ricorsivo. Per ulteriori dettagli si veda la documentazione.

1. Creo un individuo di test:

```
matrix = np.zeros((NUM_NODES, NUM_NODES))
ind = Individual(size=int(NUM_NODES * (NUM_NODES + 1) / 2), min=1, iter_range=100)
```

2. Rieseguo la procedura per ritornare dal genoma (aleatorio) del mio [individuo](individual.ipynb) test alla matrice di adiacenza (con cui posso lavorare)

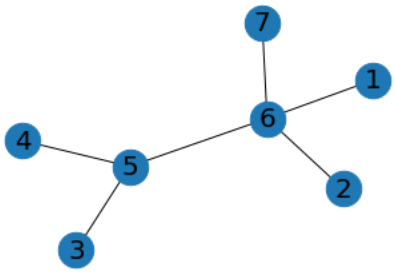
```
matrix = np.array(ind.reshape2matrix(NUM_NODES))
n = matrix.shape[0]
matrix[range(n), range(n)] = "0"
shape = matrix.shape
result = np.zeros(shape, dtype=int)
for x in range(0, shape[0]):
    for y in range(0, shape[1]):
        result[x, y] = str(matrix[x, y])[0:2]
g = nx.from_numpy_matrix(matrix, create_using=nx.DiGraph)
draw_graph(g, 0)
```




```
<Figure size 864x864 with 0 Axes>
```

3. Ottengo un MST classico con l'algoritmo di Prim

```
from CMST_package.population_generator import prim

tree = prim(matrix)
G = nx.Graph()
tree = [arc for arc in tree if arc != (0, 0)]
G.add_edges_from(tree)
draw_graph(G, 1)
```




```
<Figure size 864x864 with 0 Axes>
```

4. Conto i nodi che ci sono in ogni ramo con una procedura breadth-first search, realizzando un primo prototipo della funzione $R(\mathbf{x})$

```
# Depth first search starting from ROOT
over_capacity = False
ROOT = 1
tree_final = []
for edge in tree:
    temp_weight = 0
    if edge[0] == ROOT:
        temp_weight = measure_branch(tree, edge[1], ROOT)
        print("Dal nodo radice {0} ho {1} nodi".format(ROOT, temp_weight))
    elif edge[1] == ROOT:
        temp_weight = measure_branch(tree, edge[0], ROOT)
        print("Nodi incidenti nella radice----> {0}".format(temp_weight))
    if temp_weight > CAPACITY:
        tree_final = tree
        ind.fitness = MAXIMUM_FITNESS
        over_capacity = True
        print(
            "Violato il vincolo di capacità!! Ho {0} nodi nell'arco {1} arco invece che
{2} ".format(temp_weight, edge,

CAPACITY))
        break
```

```
Dal nodo radice 1 ho 6 nodi
Violato il vincolo di capacità!! Ho 6 nodi nell'arco (1, 6) arco invece che 4
```

## Osservazioni

Poiché questi genomi sono (per facilitare di test) aleatori, è probabile che si possano incontrare casi in cui un arco sia sovraccarico. Quando questo accade scatta la guardia situata nell'ultimo `if` e cioè:

```
python
if temp_weight > CAPACITY:
    tree_final = tree
    ind.fitness = MAXIMUM_FITNESS
    over_capacity = True
    print("Violato il vincolo di capacità!! Ho {0} nodi nell'arco {1} arco invece che {2} ".format(temp_weight, edge, CAPACITY))
    break
```

## Calcolo della fitness

**Se** il numero di nodi rispetta il vincolo di capacità si passa al calcolo del costo effettivo sulla *matrice originale* $\sum_{(i,j)\in A} c_{ij}x_{ij}$

```
if not over_capacity:
    weight = 0
    for edge in tree:
        (col, row) = edge
        weight += matrix[col - 1][row - 1]
    ind.tree = tree
    ind.fitness = weight
    print("Costo dell'albero soluzione: {0}".format(weight))
else:
    print("Non ho trovato soluzioni, riprova a lanciare l'algoritmo...")
```

```
Non ho trovato soluzioni, riprova a lanciare l'algoritmo...
```

### Osservazioni

Anche qui bisogna precisare che mi sono allontanato leggermente dalla formulazione originale. Il costo $\sum_{(i,j)\in A} c_{ij}x_{ij}$ è calcolato sulla matrice originale, cioè quella data in input da ottimizzare, ma se l'individuo sotto esame ha un albero che non rispetta i vincoli di capacità allora gli viene assegnata una fitness massima (nel codice è uguale a `MAXIMUM_FITNESS`) e viene calcolato il costo comunque. Questo costo, che viene tradotto in fitness dalle funzioni `fitness_evaluation` e `measure_branch` sarà sopra al valore `MAXIMUM_FITNESS`, rendendo questo individuo inutile.

- Per quale motivo tenere delle soluzioni che violano il vincolo di capacità? Poiché può succedere che nonostante tutte le iterazioni non riesco a generare un numero sufficiente di soluzioni ammissibili (per vederlo basta allargare la dimensione Hall of Fame e diminuire le iterazioni e si osservano soluzioni assurde addirittura con cicli...)
- Come mi accorgo di avere una soluzione finale che viola questi vincoli? Basta osservare la sua fitness, se è pari a `MAXIMUM_FITNESS` allora la soluzione è inammissibile, oppure anche se non del tutto inammissibile (cioé mi accontento di qualcosa di papabile) questa soluzione sarà pessima. (Invito a provare con varie dimensioni di Hall of Fame e iterazioni nel loop evolutivo).

## Operatore di crossover

La mutazione a coppia implementata qui è un crossover uniforme. Per mostrare come opera questa funzione:

1. Creazione di una popolazione iniziale (per l'implementazione del generatore di popolazioni si veda la documentazione)

```
from CMST_package.population_generator import generate_population

individuals = generate_population(POP_SIZE + (POP_SIZE % 2), int(NUM_NODES * (NUM_NODES +
1) / 2))
```

1. Selezione del gruppo che si accoppierà

```
group_A = list()
group_B = list()

# Selezione di chi si accoppia Tournament style
mating_group = list()
while len(mating_group) < len(individuals):
    best = None
    for i in range(TOURNAMENT_SIZE):
        # Scegli un individuo a caso
        ind = individuals[randint(0, len(individuals) - 1)]
        # Se l'individuo scelto a caso ha una fitness maggiore -> prendilo
        if best is None or ind.fitness > best.fitness:
            best = ind
    mating_group.append(best)
print("Visualizzazione di un individuo a caso: {0}".format(mating_group[1]))
```

```
Visualizzazione di un individuo a caso: 56 51 26 52 185 26 241 135 249 86 121 180 75 221
250 84 112 145 130 210 68 82 42 109 243 137 5 207
```

1. Estrazione a coppie dei genitori e ripopolazione della “lista padri” e della “lista madri”

```
while mating_group:
    group_A.append(mating_group.pop())
    group_B.append(mating_group.pop())
```

1. Crossover

```
ret = list()
for i in range(len(group_A)):
    ind1 = group_A.pop()
    ind2 = group_B.pop()
    if random() < breeding_mutation_pb:
        for i in range(randint(1, len(ind1))):
            # Qui avviene lo scambio
            temp = ind1[i]
            ind1[i] = ind2[i]
            ind2[i] = temp
    ret.append(ind1)
    ret.append(ind2)
```

## Single Mutation

In modo simile a come fatto per la mutazione a coppia si procede qui nella mutazione del singolo individuo. In pratica seleziono un gene (che qui sono i costi della lista di adiacenza) e li modifico in modo casuale in modo da cambiare il costo del singolo arco.

```
lines = inspect.getsource(self_mutation)
print(lines)
```

```
def self_mutation(individuals, mutation_pb):
    """
    Self-mutation qui con questa funzione modifico il genotipo del singolo individuo, in
pratica è come se gli
    modificassi la matrice equivalente dei costi in modo casuale.
    Potrei modificare un ramo, ma se così facessi otterrei una euristica di scambio
(branch exchange heuristic).
    Non seguo questa seconda strada (altrimenti otterrei qualcosa come Esau-Williams)
modifico la matrice di adiacenza.

    :param individuals: popolazione da mutare (in pratica un insieme di individui)
    :param mutation_pb: probabilità di mutazione di un gene
    :return:
    """
    for individual in individuals:
        for value in individual:
            if random.random() < mutation_pb:
                value = random.randrange(0, INDEX_RANGE)
    return individuals
```

Per dettagli e scelte implementative si veda la documentazione delle funzion, oppure il `help()` delle funzioni.

# Metodo Frank & Wolfe

Questo file risolve il problema dell'equilibrio su reti. Per la risoluzione di questo problema si usa l'algoritmo di Frank & Wolfe per la risoluzione del problema di ottimizzazione che ne deriva dal calcolo di equilibrio su reti. Non approfondisco le ipotesi che portano a questa formulazione, sono molto semplici.

```
import numpy as np
import sympy
import sympy as sym
from sympy import init_printing
import warnings

warnings.filterwarnings("ignore", category=UserWarning)
from IPython.display import display, Math

x1, x2, x3 = sym.symbols('x1, x2, x3')
x = sym.Matrix([[x1], [x2], [x3]])
Q = sym.Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 0.1]])
c = sym.Matrix([[0], [0], [0.55]])
```

Adesso posso visualizzare la funzione obiettivo come espressione bellina

```
init_printing()

obj = 0.5 * (x.T * Q * x) + c.T * x
obj
```

$$\left[0.5x_1^2 + 0.5x_2^2 + 0.05x_3^2 + 0.55x_3\right]$$

Adesso visualizzo anche l'obiettivo al punto iniziale:

```
cost = obj.subs({x1: 0.4, x2: 0.3, x3: 0.3})
display(Math(r'D(x_0): {:.4f}'.format(cost[0])))
```

$$D(x_0) : 0.2945$$

Adesso posso visualizzare la funzione con lo step size e l'aggiornamento

```
x1_bar, x2_bar, x3_bar = sym.symbols('\overline{x_1}, \overline{x_2}, \overline{x_3}')
x_bar = sym.Matrix([[x1_bar], [x2_bar], [x3_bar]])
d_k = x_bar - x
display(Math(r'd_k: '))
d_k
```

$$d_k :$$

$$\begin{bmatrix} \overline{x_1} - x_1 \\ \overline{x_2} - x_2 \\ \overline{x_3} - x_3 \end{bmatrix}$$

Adesso visualizzo il punto iniziale e mi calcolo il nuovo punto

```
a = sym.symbols('\\alpha')
x_k = x + a * d_k

display(Math(r'x_{k+1}: '))
x_k
```

$$x_{k+1} :$$

$$\begin{bmatrix} \alpha\left(\overline{x_1} - x_1\right) + x_1 \\ \alpha\left(\overline{x_2} - x_2\right) + x_2 \\ \alpha\left(\overline{x_3} - x_3\right) + x_3 \end{bmatrix}$$

Adesso mi calcolo il gradiente

```
grad = x.T * Q + c.T
display(Math(r'\nabla: '))
grad.T
```

$\nabla$ :

$$\begin{bmatrix} x_1 \\ x_2 \\ 0.1x_3 + 0.55 \end{bmatrix}$$

Adesso mi calcolo $\alpha$ come forma chiusa e nella cella successiva sostituisco i valori:

```
alpha = -(grad * d_k) / (d_k.T * Q * d_k)[0]
alpha
```

$$\left[\frac{-x_1(\overline{x_1}-x_1)-x_2(\overline{x_2}-x_2)-(\overline{x_3}-x_3)(0.1x_3+0.55)}{(\overline{x_1}-x_1)^2+(\overline{x_2}-x_2)^2+(0.1\overline{x_3}-0.1x_3)(\overline{x_3}-x_3)}\right]$$

Adesso sostituisco i valori e verifico:

```
1. I valori tornano con Gallagher
2. La direzione scelta con Q
3. Calcolo del nuovo punto con il quale calcolare la direzione di discesa
```

\begin{align} \overline{x}_{k+1} = argmin {<\nabla f(x_k), x>} \quad \forall x \in C \end{align}

Punto iniziale scelto per far partire l'algoritmo, uso il seguente:

```
result_x = x.subs({x1: 0.4, x2: 0.3, x3: 0.3})
result_x
```

$$\begin{bmatrix} 0.4 \\ 0.3 \\ 0.3 \end{bmatrix}$$

Uso questa soluzione iniziale:

```
result_x_bar = x_bar.subs({x1_bar: 1, x2_bar: 0, x3_bar: 0})
result_x_bar
```

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Mi calcolo la nuova direzione

```
result_d_k = d_k.subs({x1: 0.4, x2: 0.3, x3: 0.3, x1_bar: 1, x2_bar: 0, x3_bar: 0})
result_d_k
```

$$\begin{bmatrix} 0.6 \\ -0.3 \\ -0.3 \end{bmatrix}$$

Adesso mi calcolo lo step size con la line minimization

```
step_result = alpha.subs({x1: 0.4, x2: 0.3, x3: 0.3, x1_bar: 1, x2_bar: 0, x3_bar: 0})
step_result
```

$$\left[0.0522875816993464\right]$$

Il nuovo punto è

```
calc_x = np.array(x.subs({x1: 0.4, x2: 0.3, x3: 0.3}))
calc_a = np.array(a.subs({a: step_result}))
calc_d_k = np.array(d_k.subs({x1: 0.4, x2: 0.3, x3: 0.3, x1_bar: 1, x2_bar: 0, x3_bar:
0}))
new_point = calc_x + calc_a * calc_d_k
new_point
```

```
array([[0.431372549019608],
       [0.284313725490196],
       [0.284313725490196]], dtype=object)
```

## Seconda iterazione

Adesso inizio con la nuova iterazione. Inizio col punto che mi sono appena calcolato e pongo:

\begin{align} \overline{x}_{1} = argmin {<\nabla f(x_0), x>} \quad \forall x \in C \end{align}

Questo me lo calcolo con le condizioni KKT per i problemi con vincoli di simplesso. Dunque il nuovo punto che userò è soluzione del seguente problema

\begin{align} minimize \sum_{i=1}^{n} \frac{\partial f(x_{k})}{\partial x_{i}}(x_{i}-x_{i}^k) \ st \sum_{i=1}^{n} x_{i} = 1 \end{align}

La soluzione a questo problema è un punto $\bar{x}_k$ il quale ha tutte le coordinate uguali a zero eccezione fatta per una sola coordinata la quale è uguale ad 1. La j-esima coordinata corrisponde a quella coordinata con valore minimo di derivata (vedi le condizioni di KKT per il problema)

\begin{align} j = argmin \frac{\partial f(x_{k})}{\partial x_{i}}\quad \forall i =1 ... n \end{align}

```
gradient = grad.subs({x1: new_point[0][0], x2: new_point[1][0], x3: new_point[2][0]})
display(Math(r'\nabla f(x_1): '))
gradient
```

$$\nabla f(x_1) :$$

$$\begin{bmatrix}0.431372549019608 & 0.284313725490196 & 0.57843137254902\end{bmatrix}$$

Adesso applico le KKT e calcolo il nuovo punto che userò per la direzione di discesa

idx_oracle = np.argmax(np.abs(grad)) mag_oracle = alpha * np.sign(-grad[idx_oracle])

Definisco la per il singolo step dell'iterazione poi la lancio dalla funzione FW (principale)

```
def step_iteration(x_k: np.ndarray):
    grad = (x_k.T.dot(Q) + c.T).ravel()
    idx_oracle = np.argmin(np.abs(grad))
    mag_oracle = np.sign(grad[idx_oracle])
    d_t = -x_k.copy()
    d_t[idx_oracle] += mag_oracle
    g_t = - d_t.T.dot(grad).ravel()
    step_size = -grad.dot(d_t) / (d_t.T.dot(Q)).dot(d_t)
    step_size = np.minimum(step_size, 1.)
    x_k = x_k + step_size * d_t
    return x_k, g_t, d_t
```

Questa è la funzione principale in cui metto tutto insieme

```
Q = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 0.1]])
c = np.array([[0], [0], [0.55]])
```

Esecuzione e test obiettivo, direzione se è di discesa, certificato duale

```
x_k = np.array([[0.4], [0.3], [0.3]])
s_k = np.array([[1], [0], [0]])
grad = x_k.T.dot(Q) + c.T
point, dual_value, direction = step_iteration(x_k)
display(Math(r'<\nabla f(x_2),d_2> : {:.4f}'.format(direction.T.dot(grad.T).ravel()[0])))
display(Math(r'Dual(x_2): {:.4f}'.format(dual_value.ravel()[0])))
print("Direzione di discesa: ")
print(direction)
print("Punto calcolato")
print(point)
```

$$< \nabla f(x_2), d_2 >: -0.1240$$

$$Dual(x_2) : 0.1240$$

```
Direzione di discesa:
[[-0.4]
 [ 0.7]
 [-0.3]]
Punto calcolato
[[0.32473445]
 [0.43171472]
 [0.24355083]]
```

```
for i in range(1,321):
    point, dual_value, direction = step_iteration(point)
display(Math(r'<\nabla f(x_3),d_3> : {:.4f}'.format(direction.T.dot(grad.T).ravel()[0])))
display(Math(r'Dual(x_3): {:.4f}'.format(dual_value.ravel()[0])))
print("Direzione di discesa: ")
print(direction)
print("Punto calcolato")
print(point)
obj = 0.5*(point.T.dot(Q)).dot(point) + c.T.dot(point)
display(Math(r'Objective(x_3): {:.4f}'.format(obj.ravel()[0])))
```

$$< \nabla f(x_3), d_3 >: -0.0526$$

$$Dual(x_3) : 0.0013$$

```
Direzione di discesa:
[[-0.4950977 ]
 [ 0.50615651]
 [-0.01105881]]
Punto calcolato
[[0.49385908]
 [0.49510978]
 [0.01103114]]
```

$$Objective(x_3) : 0.2506$$

---

By Andi Dulla