SAPIENZA
UNIVERSITÀ DI ROMA

Generazione e visualizzazione grafica di traffico di reti

Facoltà di Ingegneria dell'informazione, informatica e statistica
Laurea Triennale in Informatica

Francesco Pannozzo
Matricola 699427

Relatore
Prof. Daniele De Sensi

Anno Accademico 2023/2024

Tesi non ancora discussa

Generazione e visualizzazione grafica di traffico di reti
Relazione di tirocinio. Sapienza Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: francesco.pannozzo@libero.it

Dedicato alla
mia famiglia

Sommario

Questa relazione descrive il lavoro di tirocinio interno svolto presso il dipartimento di Informatica dell'Università La Sapienza di Roma, concretizzato nella realizzazione di un progetto riguardante un software per poter visualizzare in forma grafica l'andamento del traffico di una rete. Il progetto ha come obiettivo di mostrare il traffico di rete al variare del tempo e ciò viene raggiunto tramite grafiche e animazioni generate programmaticamente. L'idea dell'ambito di tirocinio nasce dalla volontà di sperimentare una realizzazione front-end tramite la libreria Manim, un motore di animazioni per video matematici esplicativi.

Nel capitolo dedicato all'introduzione descriveremo l'ambito in cui si è tenuto il tirocinio, le motivazioni alla base in cui è stato idealizzato l'argomento, seguito da un'analisi dello stato dell'arte attuale, descrivendo diverse realtà affini al progetto e studiandone caratteristiche, i loro pregi e mancanze. A seguire ci sarà una descrizione dei contributi che vuole dare il progetto, scaturiti dall'analisi dello stato dell'arte e infine una breve panoramica sulla base di partenza del progetto, descrivendo da cosa si è partiti, come'è nata l'argomentazione che ha portato alla scelta del progetto e la modalità di svolgimento. Nel capitolo 2 parleremo del contesto di sviluppo, ovvero le tecnologie impiegate, seguirà quindi una descrizione approfondita della scelta di certe tecnologie ma soprattutto del perché. Il capitolo 3 descriverà il generatore di traffico e configurazione di rete, descrivendone funzionalità, condizioni inziali di setup e processo generativo, passando dalla distribuzione spaziale dei nodi e creazione dei links all'ottenimento dei files sui quali sono memorizzati i dati. Il capitolo 4 tratterà l'analizzatore di traffico, nello specifico dell'implementazione della logica che risiede dietro l'analisi del file contenente il traffico. Il capitolo 5 sarà sul visualizzatore grafico, in cui verrà fornita una panoramica sulle logiche dietro la creazione degli elementi grafici e delle animazioni, le modalità grafiche che è possibile ottenere, la descrizione delle scelte di design legate all'aspetto grafico e una sezione dedicata all'accessibilità, con cui si tratterà l'implementazione di soluzioni che trattano il tema del daltonismo. Il capitolo 6 darà un quadro generale sulla complessità computazionle delle sezioni di codice più sensibili, analizzando gli andamenti asintotici di tutte le principali parti. Infine avremo il capitolo 7, in cui tireremo le conclusioni del progetto, discutendo dei risulati finali e possibili impieghi futuri dell'applicazione.

Indice

Capitolo 1

Introduzione

Nel mondo le reti informatiche sono oramai un concetto ben istanziato nella colletività, la loro presenza è soverchiante e si dirama nei più disparati settori. Basti pensare già alle reti PAN (Personal Area Network) le quali connettono dispositivi personali entro pochi metri e che ognuno di noi usa abitualmente nella propria casa, alle reti LAN (Local Area Network), anch'esse presenti nelle nostre case così come in uffici o edifici scolastici, le reti dei data center fino a giungere alla rete globale internet, la quale è creatrice a sua volta di paradigmi come può essere l'internet of things. Le reti informatiche sono impiegate nei più vari settori come l'istruzione, in cui le reti sono cruciali nelle scuole e nelle università per avere accesso a risorse educative o sfruttare l'e-learning, i servizi pubblici governativi e sanitari, nel settore ludico e multimediale come il gioco online e l'attuale streaming di contenuti multimediali. Insomma, le reti informatiche sono di fatto una presenza piena e diffusissima ed è estremamente difficile riuscire a immaginare il mondo come lo vediamo oggi senza questa tecnologia.

Con l'aumentare delle funzionalità legate alle reti, così come i dispositivi collegati a esse, capire cosa succede al loro interno, come si muovono i dati, è quindi di cruciale importanza, tramite l'analisi dei dati che vi fruiscono è possibile fare diagnostica, per quanto riguarda un discorso di monitoraggio, ma è anche possibile applicare le analisi in un ambito didattico e accademico. Capire cosa sta succedendo in una rete in modo immediato e visivo è lo scopo di questo progetto, il quale punta a mostrare, in modo grafico, l'andamento del traffico di una rete.

1.1 Ambito del tirocinio

Il progetto fa parte del percorso di tirocinio interno intrapreso presso l'Università La Sapienza di Roma. L'argomento su cui verte il progetto è la realizzazione di un visualizzatore grafico dell'andamento del traffico di una rete di data center, basato su animazioni programmatiche. Il tool permette di visualizzare diversi switch/endpoints (nella pratica di fatto i nodi potrebbero rappresentare gli endpoint/server o gli switch in base a quello che vogliamo mostrare) e i link che li collegano i quali vengono colorati tramite animazioni nel tempo in base al traffico di rete precedentemente analizzato. Nel tool è presente anche una parte generativa di traffico di rete, una creazione di traffico fittizia di vitale importanza ai fini di testing. Il progetto è suddiviso quindi

in tre parti; una parte che si occupa della generazione di una configurazione network e relativo traffico di rete, una parte che analizza il traffico di rete calcolandone le medie percentuali di intervalli di tempo e aggiornate ripetutamente e infine una parte che visualizza il traffico analizzato producendo una rappresentazione video.

1.2 Motivazioni

L'idea di sviluppare un visualizzatore grafico di traffico di rete è nata, in sede di proposta, dal Professore Daniele De Sensi, relatore del tirocinio, e dalla mia volontà di sviluppare un'applicazione avente il front-end come focus dell'esperienza. Nel mio personale corso di studi presso il Dipartimento di Informatica non ho avuto modo di studiare e approfondire un discorso legato al front-end, per cui la volontà di intraprendere questo percorso nasce in primis da un forte interesse verso questo aspetto dell'informatica e in secondo luogo per un completamento di formazione professionale personale.

1.3 Stato dell'arte

L'esigenza di analisi di reti informatiche ha portato alla luce svariati tool che permettono appunto di analizzare cosa avviene in una rete, di studiarne i dati statistici e di visualizzare graficamente determinati scenari. L'universita americana Johns Hopkins[10] ha stilato una lista di software per la visualizzazione e analisi di reti[11]:

- **Gephi[6]:** Gephi è il software leader di visualizzazione ed esplorazione per tutti i tipi di grafici e reti ed è open source. Le sue caratteristiche includono l'analisi esplorativa dei dati mediante manipolazioni di reti in tempo reale e analisi dei collegamenti per rivelare le strutture sottostanti delle associazioni tra oggetti. Il software permette anche l'analisi dei social network per la creazione di connettori di dati sociali per mappare le organizzazioni della comunità e le reti di piccoli mondi, inoltre può essere impiegato per l'analisi della rete biologica per la rappresentazione di modelli di dati biologici con possiblità di creazione ed esportazione di mappe stampabili di alta qualità.

- **Cytoscape[4]:** è una piattaforma software open source per visualizzare reti complesse e integrarle con qualsiasi tipo di dati. Consiste in una piattaforma per visualizzare reti di interazioni molecolari e percorsi biologici, potendo integrare queste reti con annotazioni, profili di espressione genica e altri dati. Originariamente progettato per la ricerca biologica, ora è una piattaforma generale per l'analisi e la visualizzazione di reti complesse.

- **GraphVis[7]:** è un software di visualizzazione di grafici open source. Può accettare descrizioni di grafici in un semplice linguaggio di testo e creare diagrammi in formati utili, come immagini e SVG per pagine web; PDF o Postscript per l'inclusione in altri documenti o visualizzare in un browser grafico interattivo. Graphviz ha molte funzionalità utili per diagrammi, come

opzioni per colori, caratteri, layout di nodi tabulari, stili di linea, collegamenti ipertestuali e forme personalizzate.

- **igraph[8]:** è una collezione di librerie per creare, manipolare grafici e analizzare ponendo l'enfasi nell'efficienza, portabilità e facilità d'uso. Igraph è open source e gratuito e può essere programmato in R, Python, Mathematica e C/C++

- **UCINET6[24]:** è un pacchetto software per l'analisi dei dati dei social network. UCINET viene fornito con il tool di visualizzazione di rete NetDraw. Può leggere e scrivere una moltitudine di file di testo diversamente formattati, nonché file Excel. I metodi di analisi dei social network includono misure di centralità, identificazione di sottogruppi, analisi di ruolo, teoria dei grafi elementari e analisi statistica basata sulla permutazione. Inoltre, il pacchetto dispone di potenti routine di analisi delle matrici, come l'algebra delle matrici e la statistica multivariata.

- **SocNetV[20]:**è un'applicazione software gratuita multipiattaforma per l'analisi e la visualizzazione dei social network. Tra le caratteristiche principali troviamo il poter disegnare il grafo associato al social network, caricare i campi da un file supportato (GraphML, GraphViz, Adjacency, EdgeList, GML, Pajek, UCINET, ecc.), personalizzare attori e collegamenti tramite sistema punta e clicca, analizzare le proprietà dei grafici e dei social network, produrre report HTML e incorporare layout di visualizzazione di rete

- **Pajek[12]:** è un software per la visualizzazione e l'analisi delle reti. La sua forza risiede nel poter analizzare reti complesse potendo arrivare fino a un miliardo di vertici. L'analisi e la visualizzazione vengono eseguite utilizzando sei tipi di dati: rete (grafico), partizione, vettore, cluster (sottoinsieme di vertici), permutazione (riordinamento dei vertici, proprietà ordinali); e gerarchia (struttura generale ad albero sui vertici).

1.4 Contributi

Da un punto di vista grafico e quindi di visualizzazione, la maggior parte degli strumenti sopra elencati, permette una certa forma di personalizzazione nella disposizione dei nodi, sia automaticamente attraverso algoritmi di layout sia manualmente, permettendo agli utenti di spostare i nodi per ottimizzare la visualizzazione o per enfatizzare certi aspetti della rete. Tuttavia, la possibilità di avere animazioni dinamiche che mostrino l'andamento del traffico nel tempo, mostrando la variazione del colore in base alla quantità dello stesso, risulta essere una caratteristica meno comune nei software di analisi di rete, nello specifico:

- Gephi: Non supporta nativamente animazioni dinamiche basate su traffico in tempo reale. Tuttavia, la sua flessibilità e la capacità di aggiungere plugin potrebbero permettere implementazioni personalizzate.

- Cytoscape: Anche se fortemente orientato all'analisi statica, plugin o estensioni potrebbero aggiungere capacità simili.

- GraphVis (Graphviz): Principalmente orientato verso la visualizzazione statica; non supporta direttamente animazioni dinamiche dei link basate sul traffico.
- igraph: Come libreria di analisi, non è orientato verso la visualizzazione in tempo reale o animazioni dei link basate su traffico nel suo utilizzo standard.
- UCINET (con NetDraw): Focalizzato sull'analisi statica di reti sociali; non supporta animazioni dinamiche in base al traffico.
- SocNetV: Orientato all'analisi statica e alla visualizzazione; non è progettato per visualizzare animazioni dinamiche basate sul traffico.
- Pajek: Simile agli altri, è più un tool per l'analisi statica e la visualizzazione di grandi reti, senza un supporto diretto per animazioni dei link basate su traffico.

In questo contesto, l'inserimento di una caratteristica che permetta di fare quanto premesso come base del progetto di tirocinio, risulta particolarmente indicata nel contribuire a fornire una soluzione visiva come strumento aggiuntivo di analisi di una rete, di debugging e anche come strumento didattico. La possibilità di avere un riscontro visivo istantaneo di cosa avviene nel tempo in una rete, a livello di traffico, può dare immediato feedback nel caso ci fosse un problema di congestione in un punto nevralgico, oppure mostrare parti di rete libere dove poter studiare un reindirizzamento dello stesso, volto a ottimizzare le prestazioni. A livello didattico ciò si potrebbe mostrare per presentazioni così come per spiegazioni. Insomma i benefici derivanti da una rappresentazione del genere sono evidenti e ciò può essere di grosso aiuto nell'analisi così anche solo come semplice rappresentazione del traffico di rete, nonchè uno strumento complementare a quanto già presente in circolazione.

1.5 Base di partenza del progetto

Il progetto è partito da zero, si basa sullo sviluppo totalmente nuovo dell'applicazione ed è stato tutto idealizzato e pianificato in sede di proposta. Come approfondirò in seguito, nella sezione dedicata alla tecnologia impiegata, il progetto non è l'unica cosa a essere partita da zero, poiché il linguaggio scelto per sviluppare l'applicazione è Python[18], linguaggio non incluso nel mio personale percorso di studi e che ho dovuto necessariamente studiare da zero per poter affrontare il percorso di tirocinio. Il lavoro è stato svolto individualmente.

Capitolo 2

Contesto di sviluppo: le tecnologie impiegate

Il progetto, a livello di tecnologia impiegata, pone le fondamenta su tre aspetti che andremo a elencare di seguito, descrivendo le varie motivazioni che hanno spinto a sceglierli.

2.1 Python

Uno dei primi aspetti di cui si è tenuto conto è stata la scelta del linguaggio di programmazione che, come accennato precedentemente, è Python. Python risulta essere il linguaggio più usato al mondo, ad affermarlo è l' Institute of Electrical and Electronics Engineers (IEEE)[9][1] un'associazione internazionale di scienziati professionisti con l'obiettivo della promozione delle tecnologie, incluso il settore dell'information technology (IT). Il linguaggio ha molte caratteristiche ottime, come una sintassi semplice e leggibile che lo rende facile da imparare e semplice da usare per gli sviluppatori esperti, accorciando di gran lunga i tempi di sviluppo. Un altro pregio è l'avere una grande versatilità per poter essere usato in ambiti diversi come l'intelligenza artificiale, il web development, data analysys e molto altro, grazie a un ampio supporto delle librerie, due delle quali usate proprio nel progetto (di cui parlerò a breve). Python vanta di una grande comunità in cui trovare facilmente risorse, tutorial e supporto, un'interoperabilità che permette un'ottima integrazione con altre tecnologie e altri linguaggi, un linguaggio orientato agli oggetti volto a facilitare la gestione del codice e migliorare il riuso, scalabile e di facile integrazizone.

2.2 La libreria Manim

Il secondo aspetto risiede nella scelta della libreria Manim[22], che viene definita come *Animation engine for explanatory math videos*. Lo scopo di Manim è quindi quello di animare concetti tecnici legati alla matematica e si affida alla semplicità di Python per generare animazioni in modo programmatico. Manim può produrre anche immagini e gif, ma è nella produzione di video che da il meglio, in questo modo è possibile progettare animazioni e renderle visibili in movimento, creando figure algebriche, grafici cartesiani, grafi e molto altro[22]. La libreria offre molta libertà sui

risultati che si vogliono ottenere, può renderizzare singole immagini, gif e da molte opzioni per quanto riguarda l'output video, fornendo la possibilità di renderizzarli nei formati più comuni con la possibilità di personalizzazione del framerate:

- 480p (SD): 854 x 480
- 720p (HD): 1280 x 720
- 1080p (HD): 1920 x 1080
- 1440p (2K): 2560 x 1440
- 2160p (4K): 3840 x 2160

Manim viene impiegato principalmente per presentazioni che implichino aspetti matematici, la sfida del progetto è stata quella di cercare di sfruttare le potenzialità della libreria e renderle al servizio di uno strumento di analisi sul traffico di reti, una sfida affrontata con successo come potremo vedere in seguito. Il terzo aspetto tecnologico riguarda l'aspetto di gestione dei dati. Un visualizzatore grafico di traffico di reti ha bisogno principalmente di due insiemi di informazioni importanti; uno riguarda tutte le informazioni che riguardano il come è costruita la rate, parliamo quindi degli endpoints quali sono gli switch e conseguenti informazioni annesse, pensiamo ad esempio all'indirizzo di rete, un nome identificativo e così via, ma parliamo anche dei link che collegano i vari endpoints, con la necessità di tenere traccia delle loro capacità trasmissive, la tipologia di rete, se ha una struttura a grafo completo, mesh, torus o disposizione libera e molte altre informazioni che discuteremo in seguito.

2.3 I formati Json e Yaml

L'altro insieme di informazioni deriva dal traffico vero e proprio, rendendo quindi necessario un sistema di mantenimento dei dati legati ai pacchetti trasmessi. L'analisi di questi due insiemi di informazioni ha portato alla valutazione di tre sistemi per la strutturazione di dati; `json` [21], `yaml` [30] e `csv` [19]. Dopo attenta anlisi si è deciso di adottare il formato `json` per strutturare e memorizzare i dati legati al traffico, con la possibilità di scegliere anche il formato `yaml`, mentre per i dati relativi alla descrizione della rete il compito è stato affidato esclusivamente a `yaml`, `csv` è stato scartato. Perché `csv` è stato scartato? Sebbene `csv` rappresenti una valida alternativa tenendo conto di aspetti prestazionali, essendo un formato molto veloce da analizzare, da leggere e scrivere, tuttavia lo diventa meno quando c'è bisogno di strutturare maggiormente i dati con strutture più complesse della semplice forma tabellare, tipicamente usate nei database. L'idea di scartarla, sia per strutturare i dati della rete che per quelli del traffico, deriva principalmente dai seguenti motivi:

- **Leggibilità:** uno dei primi intenti del progetto era di rendere l'applicazione il più leggibile possibile, questo perché si è fortemente voluto attribuirne anche scopi di debugging e didattici, laddove avere una certa leggibilità è più che ragionevole.

- **Strutture complesse:** sicuramente il motivo più importante. Con `csv` non è possibile rappresentare strutture complesse, parliamo ad esempio di oggetti all'interno di altri oggetti. Sebbene per come sia ora strutturato il progetto una rappresentazione cvs è ancora possibile, ciò potrebbe non esserlo in futuro nell'ottica di espansione del progetto

Per esplicare meglio il concetto di struttura complessa non fattibile è possibile focalizzarsi sul seguente esempio. Segue la rappresentazione di un pacchetto di rete così come viene utilizzato nel progetto, in questo caso una lista contenente un solo elemento:

```
[
    {
        "A": 1,
        "B": 2,
        "t": "2024-03-22 12:30:00",
        "d": 1518
    }
]
```

Codice 2.1. *Pacchetto di rete rappresentato in* `json`

Dove A e B sono gli endpoints interessati, t è il timestamp di creazione pacchetto e d la dimesione del payload in bytes. In `csv` questa struttura è rappresentabile come segue:

```
    A,B,t,d
    2,1,2024-03-22 12:30:00,1518
    2,1,2024-03-22 12:30:00,1518
```

Codice 2.2. *Pacchetti di rete rappresentati in* `csv`

Tuttavia se si dovesse rendere questa struttura più complessa, avremmo problemi a realizzarla in `csv`, basterebbe l'aggiunta di un campo che a sua volta necessita di informazione strutturata, come ad esempio inserire le informazione dell'header [13]:

```
        {
          "A": 1,
          "B": 2,
          "t": "2024-03-22 12:30:00",
          "d": 4000,
          "ip_header": {
            "version": 4,
            "ihl": 5,
            "type_of_service": 0,
            "total_length": 300,
            "identification": 98765,
            "flags": {
              "reserved_bit": false,
              "dont_fragment": true,
              "more_fragments": false
            },
            "fragment_offset": 0,
            "time_to_live": 64,
            "protocol": 6,
            "header_checksum": "2B5A",
            "source_address": "192.168.1.1",
            "destination_address": "192.168.1.2"
          }
        }
```

Codice 2.3. *Pacchetto di rete maggiormente strutturato in* `json`

In questa struttura abbiamo ben due oggetti nidificati, `ip_header` e `flags`. Questo tipo di struttura è difficilmente replicabile in `csv` che si presta maggiormente per strutture piatte mentre `json` e `yaml` permettono molta più libertà di strutturazione. La scelta del formato `json` per rappresentare i dati del traffico di rete è quindi legata alla possibilità di espandere la rappresentazione con strutture più complesse in ottica di espansioni future del progetto, ma è indubbiamente legata alle prestazioni che questo formato riesce a dare. `json` è un formato ampiamente supportato e la sua semplicità permette di ottenere risultati ottimi per le operazioni di parsing, lettura e scrittura per strutture aventi grandi quantità di dati. Sebbene l'applicazione supporti sia `json` che `yaml` per quanto riguarda il memorizzare i dati legati al traffico, la scelta preferenziale ricade su `json`. Questo perché, nonostante `json` produca files di dimensioni maggiori rispetto a `yaml`, la schiacciante velocità di elaborazione di `json` rende la scelta ampiamente giustificabile, considerando soprattuto la natura del progetto in cui non vi è una criticità d'uso nella dimensione dei files. Come vedremo in Sec. 3.7, i test eseguiti sui tempi di lettura e scrittura, per la stessa struttura ricreata rispettivamente con `json` e `yaml`, sapranno ben dimostrare quanto appena affermato. Per impiegare questi formati sono state usate rispettivamente la libreria `json` [17], appartenente alla libreria standard di Python, e la libreria esterna PyYAML [2], un parser ed emitter di `yaml` per Python.

Infine giungiamo ai motivi per cui si è scelto invece esclusivamente `yaml` per rappresentare i dati relativi alle informazioni che descrivono la rete. Le motivazioni consistono sempre nella volontà di garantire un formato che si presti a espansioni future e quindi che possano richiedere strutturazioni complesse, ma soprattutto, in questo caso specifico, nell'alta leggiblità che `yaml` offre. Generalmente i dati che servono a descrivere una rete non sono mai paragonabili a quelli necessari per registrare tutto il traffico, si voleva dare quindi uno strumento molto chiaro da leggere per fare in modo che la configurazione di rete fosse sempre molto chiara, intuitiva e di facile accesso, sia in scrittura che in lettura. Per dare una stima di grandezza, per rappresentare una rete con 50 switch collegati come un grafo mesh, il file di configurazione network prodotto dal generatore di pacchetti pesa appena 9 KB.

Seguirà la descrizione approfondita del progetto, principalmente suddiviso in tre parti; la prima parte riguarda la generazione di traffico di rete fittizio, la seconda è l'analizzatore dei dati di configurazione di rete e relativo traffico mentre la terza parte riguarda il visualizzatore grafico vero e proprio.

Capitolo 3

Generatore di traffico e configurazione di rete

Il progetto è composto da tre anime, una di esse è un generatore di traffico di rete e relativa configurazione. Lo scopo del generatore è da ricercare principalmente in motivazioni di testing, ma può essere usato anche in ambito didattico per scopi esemplicativi. Con esso è stato possibile testare l'analizzatore e conseguentemente il visualizzatore grafico, nonchè si è creato uno strumento esemplicativo di determinati funzionamenti collegati al traffico di rete. La configurazione di rete consiste nella produzione di una struttura dati che descriva le caratteristiche della rete, la quale verrà memorizzata in un file dal nome `network` e avente estensione `yaml`, mentre per quanto riguarda il traffico, viene creata una struttura che consiste in una lista di pacchetti che verrà memorizzata in un file denominato `packets`.

3.1 Funzionalità e setup

Lo script permette di eseguire quanto descritto mettendo a disposizione due modalità, auto e user. Entrambe le modalità produrranno due files; `network.yaml` conterà le caratteristiche della rete e il file `packets.json` (o `packets.yaml` a seconda della scelta) conterrà il traffico vero e proprio con i pacchetti. La modalità auto chiede all'utente il numero di switch, la capacità dei link e la tipologia del grafo con il quale rappresentare la rete (completo, mesh, torus) e imposterà in modo del tutto automatico la disposizione degli switch in base alla scelta del grafo effettuata.

Per poter operare, in entrambe le modalità, lo script ha bisogno di leggere il file `sim_setup.yaml`, il quale contiene le informazioni necessarie per la configurazione. Il file di setup è impostato come segue:

```
averageDelta: 1000
updateDelta: 100
creationDelta: 100
startSimTime: 2024-03-22 12:30:00
simTime: 5
packetSize: 4000
colorblind: "no"
dotsSize: "fixed"
trafficVariation: "random"
packetsFile: "json"
```

Codice 3.1. *Esempio di setup file*

Andremo ora a spiegare il significato dietro ogni voce:

- **`averageDelta`:** rappresenta l'intervallo temporale in millisecondi delle medie percentuali di traffico da calcolare nell'analizzatore di traffico.
- **`updateDelta`:** rappresenta ogni quanti millisecondi debbano essere aggiornate le medie `averageDelta` nell'analizzatore
- **`creationDelta`:** rappresenta ogni quanti millisecondi debbano essere creati i pacchetti nel generatore
- **`startSimTime`:** è il datetime dell'inizio della generazione nel formato YY:MM:DD HH:MM:SS, inoltre è possibile specificare anche un eventuale tempo avente millisecondi da specificare come ad esempio 12:30:00.500
- **`simTime`:** è la durata della generazione in secondi
- **`packetSize`:** è la dimensione in bytes di un pacchetto, i pacchetti nella generazione avranno questa dimensione
- **`colorblind`:** è una stringa "yes" o "no" che abilita, se posta su "yes", una visualizzazione compatibile per persone daltoniche di cui parleremo successivamente sella sezione 5.3 dedicata.
- **`trafficVariation`:** è la variazione di traffico per secondo che si desidera impostare; può essere il valore "random" oppure uno dei seguenti [5, 10, 20, 25, 50] e di conseguenza determinerà la variazione percentuale di traffico che avviene ogni secondo di generazione. Il valore "random" sceglie casualmente una percentuale ogni secondo con un valore che va da 0 a 100
- **`packetsFile`:** specifica quale tecnologia si vuole utilizzare per il file packets, si può scegliere tra `json` e `yaml`

La generazione di pacchetti è calcolata sulla base della capacità dei link fornita e su un valore casuale di percentuale di traffico che varia ogni secondo, per esempio avendo 6 links su 3 secondi di generazione e il parametro `trafficVariation` settato a random potremmo avere delle assegnazioni di percentuali di traffico come le seguenti:

```
endpoints: [1, 2], sim second: 0, trafficPerc: 65
endpoints: [1, 3], sim second: 0, trafficPerc: 9
endpoints: [2, 4], sim second: 0, trafficPerc: 23
endpoints: [2, 5], sim second: 0, trafficPerc: 67
endpoints: [3, 6], sim second: 0, trafficPerc: 7
endpoints: [3, 7], sim second: 0, trafficPerc: 87
endpoints: [1, 2], sim second: 1, trafficPerc: 86
endpoints: [1, 3], sim second: 1, trafficPerc: 26
endpoints: [2, 4], sim second: 1, trafficPerc: 13
endpoints: [2, 5], sim second: 1, trafficPerc: 13
endpoints: [3, 6], sim second: 1, trafficPerc: 39
endpoints: [3, 7], sim second: 1, trafficPerc: 65
endpoints: [1, 2], sim second: 2, trafficPerc: 31
endpoints: [1, 3], sim second: 2, trafficPerc: 54
endpoints: [2, 4], sim second: 2, trafficPerc: 11
endpoints: [2, 5], sim second: 2, trafficPerc: 20
endpoints: [3, 6], sim second: 2, trafficPerc: 17
endpoints: [3, 7], sim second: 2, trafficPerc: 46
```

Codice 3.2. *Esempio di variazione traffico casuale*

Una volta lanciato il generatore avremo le seguenti possibili scelte:

- **auto:** la modalità automatica, provvederà a disporre i nodi (switch) di rete in modo del tutto automatico. Dopo aver scelto la modalità auto si dovranno inserire i seguenti parametri via prompt:
  - numero di switch
  - capacità dei link
  - tipologia della rappresentazione grafica della rete (grafo completo, mesh, torus)
- **user:** una modalità in cui l'utente può personalizzare la disposizione dei nodi della rete

I grafi automatici supportati sono:

- **grafo completo:** ogni nodo è collegato a tutti i nodi rimanenti [27]
- **mesh:** ogni nodo ha 4 nodi adiacenti fatta eccezione dei nodi posti alle estremità (Fig. 3.2).
- **torus:** ogni nodo ha 4 nodi adiacenti.

Il tipo di grafo torus [28] rappresentato nel progetto è quello in due dimensioni come mostrato in Fig. 3.1:

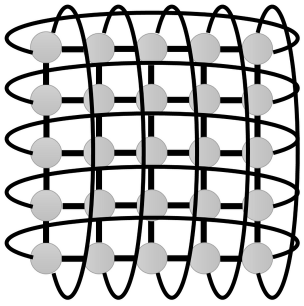


Figura 3.1. *Esempio di torus graph [29]*

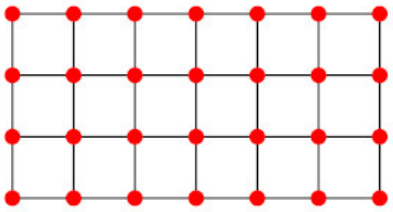


Figura 3.2. *Esempio di mesh grid graph [25]*

Nel progetto assumiamo che tutti i link che collegano i nodi nei grafi siano intesi come bidirezionali. La modalità user necessita di un file `custom_graph.yaml` con i parametri necessari a descrivere la rete del quale si vuole analizzare il traffico. Con questa modalità l'utente ha completa libertà nel personalizzare la rete ed è tenuto quindi a descriverne ogni suo aspetto. Il `custom_graph.yaml` prevede la struttura di un dizionario in cui a ogni chiave corrisponde un valore, mostriamo un esempio con possibili varianti che elencheremo:

```
---
data:
  graphType: mesh
  coordinates:
    - [1, 0, 2, 0]
    - [3, 0, 4, 5]
    - [6, 7, 8, 9]
```

```
    switches:
      1:
        ip: "123.123.123.0"
        switchName: Anthem
      2:
        ip: "123.123.123.1"
        switchName: Beta
      3:
        ip: "123.123.123.2"
        switchName: Cyber
      4:
        ip: "123.123.123.3"
        switchName: Dafne
      5:
        ip: "123.123.123.4"
        switchName: Eclipse
      6:
        ip: "123.123.123.5"
        switchName: Fox
      7:
        ip: "123.123.123.6"
        switchName: Gea
      8:
        ip: "123.123.123.7"
        switchName: H20
      9:
        ip: "123.123.123.8"
        switchName: Italy
    links:
      - { linkCap: 10,  endpoints: [1, 3] }
      - { linkCap: 100, endpoints: [1, 6] }
      - { linkCap: 10,  endpoints: [3, 6] }
      - { linkCap: 10,  endpoints: [4, 8] }
      - { linkCap: 10,  endpoints: [5, 9] }
      - { linkCap: 100, endpoints: [3, 5] }
      - { linkCap: 10,  endpoints: [6, 9] }
      - { linkCap: 100, endpoints: [4, 5] }
      - { linkCap: 10,  endpoints: [6, 7] }
      - { linkCap: 10,  endpoints: [7, 8] }
      - { linkCap: 100, endpoints: [8, 9] }
      - { linkCap: 10,  endpoints: [2, 4] }
      - { linkCap: 10,  endpoints: [2, 8] }
    phases:
      2024-01-01 00:00:01: "phase1"
      2024-01-01 00:00:02: "phase2"
```

Codice 3.3. *Esempio di custom file per la configurazione*

Descriviamo i vari parametri:

- **`graphType`:** identifica la tipologia del grafo da rappresentare, sono disponibili tre opzioni:
  - **mesh:** l'algoritmo individua in modo automatico gli archi (i link) che collegano i nodi (gli switch) adiacenti tra loro presenti nella matrice coordinates
  - **torus:** esegue lo stessa procedura usata per mesh e in addizione collega tra loro i nodi che si trovano alle estremità della matrice
  - **graph:** è la modalità più libera, collega gli switch tramite i link forniti dall'utente, indipendentemente da dove vengano collocati

- **coordinates:** rappresenta le coordinate dei vari switch, i quali vanno rappresentati con un id numerico che va da 1 a 1000. C'è una precisa motivazione di design per questa scelta ed è legata a una rappresentazione grafica ottimale del visualizzatore grafico. Gli id degli switch vanno posizionati nella matrice in base alla posizione desiderata, mentre tramite il posizionamento di uno zero si rappresenta uno spazio vuoto in cui non è presente uno switch.

- `links:` rappresenta i link della rete i quali possono essere specificati con i campi:

  - `linkCap:` esprime la capacità del link in Mbps
  - `endpoints:` esprime gli endpoints collegati al link

- `phases:` rappresenta le fasi temporali che accompagnano la durata dell'attività di rete, sono identificate tramite timestamp che ha come valore la descrizione della fase che parte dal timestamp stesso. Le fasi verranno visualizzate nel video prodotto.

Il lettore avrà notato che per il campo `graphType` manca la possibilità di scegliere un grafo completo. In realtà la possibilità c'è e risiede semplicemente nell'opzione "graph", essendo la modalità libera, l'utente può tranquillamente descrivere un grafo completo in questo modo, avendo chiaramente l'accortezza di posizionare i nodi in modo tale che i link non si sovrappongano. Tuttavia ciò e solo valido per quanto riguarda il generatore, se l'utente dovesse fornire un file `network.yaml` che descriva un grafo completo, da fornire direttamente come input per l'analizzatore di traffico, può farlo tranquillamente, in quel caso avrà ben due modi per farlo; scegliere il `graphType` come `complete` per avere una disposizione circolare dei nodi e lasciando il campo `coordinates` vuoto, oppure come "graph" ma specificando le coordinate. Torneremo sull'argomento nel capitolo dedicato al visualizzatore grafico. Come si può notare dalla figura 3.3, gli switch sono identificati tramite un valore numerico, questa è una precisa scelta di design e mira a mantenere il concetto di leggibilità sempre presente. In questo caso la scelta deriva dal semplificare e rendere immediatamente chiaro il campo `coordinates` (rappresentato da una matrice quadrata), così che l'occhio possa identificare immediatamente la rappresentazione della disposizione. Questa leggibilità è ottenuta grazie al formato `yaml`, il quale minimizza i caratteri da scrivere e rende più chiara l'associazione dei valori ai corrispettivi campi. Così facendo si rende intuitivo il posizionamento degli switch nello spazio, inoltre è una caratteristica importante per quanto riguarda il visualizzatore grafico, poiché rappresentando i nodi con un valore numerico che va da 1 a 1000, è possibile rappresentare su schermo una grande quantità di elementi per far si che possano essere visualizzati contemporaneamente e rimanere leggibili ma soprattuto, distinguibili. Uno dei problemi maggiormente riscontrati nelle applicazioni discusse nella sezione Stato dell'arte 1.3 è che, le varie soluzioni impiegate per la rappresentazione dei nodi, risultano essere confusionarie per via di una sovrapposizione massiva di link che collegano i nodi, mentre uno degli scopi del progetto è di rendere altamente leggibile l'interpretazione del traffico di rete ed è per questo che si è posto una particolare attenzione in termini di sovrapposizione degli elementi, spaziatura adeguata, scelta dei colori giusti e altri aspetti che approfondiremo nella sezione apposita riguardante

la parte di visualizzazione grafica. Qualora la capacità dei link sia uguale per tutti è possibile insererire il campo `linkCap` per poter così evitare di avere lo stesso valore nei vari link rappresentati nel campo links, potendo rappresentare solo gli endpoints:

```
data:
  graphType: torus
  coordinates:
    - [1, 0, 2, 0]
    - [3, 0, 4, 5]
    - [6, 7, 8, 9]
  linkCap: 10
  switches:
    1:
      ip: "123.123.123.0"
      switchName: A
    2:
      ip: "123.123.123.1"
      switchName: B
    3:
      ip: "123.123.123.2"
      switchName: C
    4:
      ip: "123.123.123.3"
      switchName: D
    5:
      ip: "123.123.123.4"
      switchName: E
    6:
      ip: "123.123.123.5"
      switchName: F
    7:
      ip: "123.123.123.6"
      switchName: G
    8:
      ip: "123.123.123.7"
      switchName: H
    9:
      ip: "123.123.123.8"
      switchName: I
  links:
    - [1, 3]
    - [1, 6]
    - [3, 6]
    - [4, 8]
    - [5, 9]
    - [3, 5]
    - [6, 9]
    - [4, 5]
    - [6, 7]
    - [7, 8]
    - [8, 9]
    - [2, 4]
    - [2, 8]
  phases:
    2024-01-01 00:00:01: "phase1"
    2024-01-01 00:00:02: "phase2"
```

Codice 3.4. *Esempio di custom file con capacità uguali a tutti i link*

Un'altra possibilità è quella di lasciare che il programma ricavi in automatico i link (come spiegato in Sec. 3.3), in questo caso basterà specificare solo linkCap che sarà uguale per tutti i link:

```
data:
  graphType: mesh
```

```
  coordinates:
    - [1, 2, 3]
    - [4, 5, 6]
    - [7, 8, 9]
  linkCap: 10
  switches:
    1:
      ip: "123.123.123.0"
      switchName: Anthem
    2:
      ip: "123.123.123.1"
      switchName: Beta
    3:
      ip: "123.123.123.2"
      switchName: Cyber
    4:
      ip: "123.123.123.3"
      switchName: Dafne
    5:
      ip: "123.123.123.4"
      switchName: Eclipse
    6:
      ip: "123.123.123.5"
      switchName: Fox
    7:
      ip: "123.123.123.6"
      switchName: Gea
    8:
      ip: "123.123.123.7"
      switchName: H20
    9:
      ip: "123.123.123.8"
      switchName: Italy
  phases:
    2024-01-01 00:00:01: "phase1"
    2024-01-01 00:00:02: "phase2"
```

Codice 3.5. *Esempio di custom graph file con link di eguale capacità*

Come si può notare, questo tipo di approccio concede una certa libertà di azione, dando la possibilità di prestarsi anche a soluzioni ibride di grafi canonici. Per esempio avendo una configurazione come quella riportata in figura 3.3, una volta analizzati i dati tramite l'analizzatore e forniti come imput al visualizzatore, potremmo ottenere un risultato in cui i nodi nella matrice che abbiano un valore pari a zero, non vengano collegati (Fig. 3.3).

L'ultima possibilità di personalizzazione consiste nel poter personalizzare la matrice `coordinates` come si preferisce, seguendo sempre la regola che laddove c'è un valore numerico, esso sarà un nodo, mentre laddove ci sarà uno zero verrà inteso come spazio vuoto. Quindi se si vuole rappresentare un grafico che non sia di tipo mesh o torus, bisogna specificare nel campo `graphType` il valore "graph", mostriamo un esempio:

```
  ---
  data:
    graphType: graph
    coordinates:
      - [0, 0, 0, 0, 1, 0, 0, 0, 0]
      - [0, 0, 2, 0, 0, 0, 3, 0, 0]
      - [0, 4, 0, 5, 0, 6, 0, 7, 0]
    linkCap: 10
    switches:
      1:
```

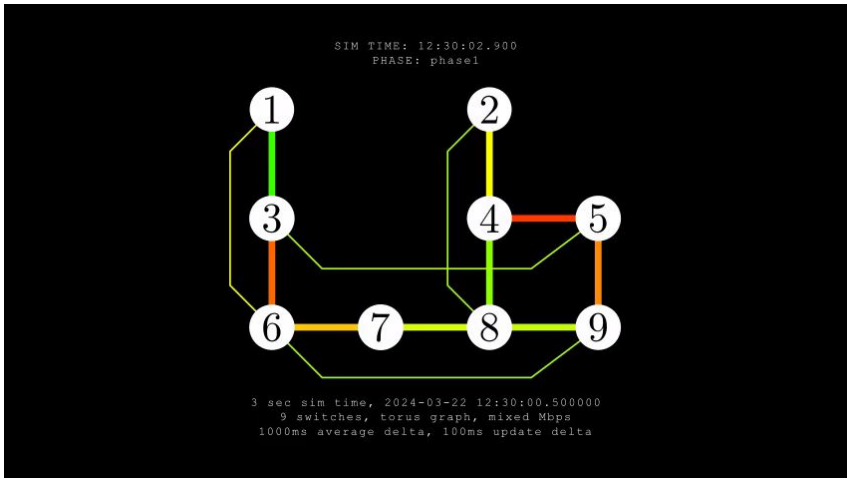


Figura 3.3. *Risultato grafico del visualizzatore per un custom graph file*

```
        ip: "123.123.123.0"
        switchName: A
      2:
        ip: "123.123.123.1"
        switchName: B
      3:
        ip: "123.123.123.2"
        switchName: C
      4:
        ip: "123.123.123.3"
        switchName: D
      5:
        ip: "123.123.123.4"
        switchName: E
      6:
        ip: "123.123.123.5"
        switchName: F
      7:
        ip: "123.123.123.6"
        switchName: G
    links:
      - [1, 2]
      - [1, 3]
      - [2, 4]
      - [2, 5]
      - [3, 6]
      - [3, 7]
    phases:
      2024-01-01 00:00:01: "phase1"
      2024-01-01 00:00:02: "phase2"
```

Codice 3.6. *Esempio di custom graph file,* `graphType` *posto al valore graph*

Come possiamo intuire osservando i valori nel campo `coordinates`, l'esempio ricrea la struttura di un albero, i links sono appunto specificati nel campo `link` e una volta generato il traffico con il generatore, analizzato e dato come input al visualizzatore otteniamo un risultato come in figura 3.4.

Vien da sè che si ottiene così una certa libertà di espressione bidimensionale, con la quale l'utente può scegliere la rappresentazione che più si addice alla rete che si vuole realizzare.

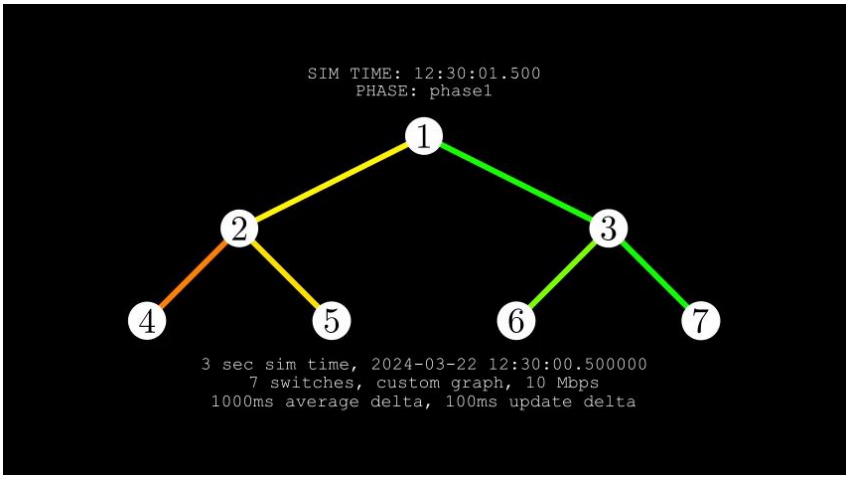


Figura 3.4. *Risultato grafico del visualizzatore per un custom graph file in modalità "graph"*

3.2 Esecuzione del generatore

Il generatore può essere pensato come suddiviso in sezioni logiche per quanto riguarda la stesura del codice nello script; la prima riguarda la fase di interazione con l'utente, il codice strutturato in una serie di richieste per l'utente e controlli sul corretto inserimento dei dati, successivamente vi è la parte di caricamento del file contentente i parametri di rete 3.1, segue la creazione dei links, la generazione del traffico creando tutti i pacchetti, si creano le informazioni inerenti agli switch e il tutto viene memorizzato sui files. Vedremo più in dettaglio questi aspetti, a livello di codice, nel capitolo dedicato allo studio della complessità computazionale e prestazioni, mentre in questo capitolo ci focalizzeremo sulla descrizione su come viene generato il tutto.

Descriveremo ora il funzionamento dell'esecuzione vera e propria dello script relativo alla generazione del traffico e dei files che produrrà. All'avvio l'utente sarà chiamato a scegliere tra due modalità, come accennato precedentemente, auto e user; scegliendo auto verrà chiesto all'utente di scegliere il numero di switch che si vuole attribuire alla rete, da un minimo di due a un massimo di mille. Questo intervallo numerico nasce per poter rappresentare una rete in un generico data center, il quale generalmente ospita fino a un massimo di mille switch. Ciò non toglie che, in un ottica futura di espansione del software, questo limite non possa essere espanso con le dovute modifiche. Successivamente alla scelta nel numero di switch l'utente sarà chiamato a scegliere la tipologia di rete potendo scegliere tra i valori "c" (completo), "m" (mesh) e "t" (torus). Per grafo completo si intende quando un nodo è collegato a tutti i nodi rimanenti, è il grafo computazionalmente parlando più oneroso da ricreare per via della grande crescita del numero di archi da rappresentare al crescere dei nodi. Avendo infatti n nodi avremo un numero di link l pari a:

$$l = \frac{n(n-1)}{2} \tag{3.1}$$

Ogni nodo è collegato a tutti gli altri meno se stesso, $n(n-1)$, e si divide per due perché gli archi non vanno contati due volte. Si parla quindi di una crescita esponenziale al variare del numero di nodi.

Nel caso della scelta di un grafico torus, il visualizzatore provvederà a disegnare la struttura come mostrata in figura 3.1. La scelta successiva riguarda la capacità che si vuole associare ai link appartenenti alla rete, potendo scegliere tra i valori 10, 100 e 1000 e vengono intese come grandezze espresse in megabits per secondo (Mb/s). Una volta inseriti questi parametri il generatore provvederà a creare la configurazione di rete e il rispettivo traffico, memorizzando i dati rispettivamente nei files `network.yaml` e `packets.json/yaml`. Scegliendo la modalità user non viene richiesto alcun dato via prompt, questo perché, come descritto in precedenza, tutti i dati di cui ha bisogno il generatore devono essere presenti nel file `custom_graph.yaml` 3.3. Un diagramma che illustra le fasi e le scelte cui è possibile fare è rappresentato in figura 3.5.

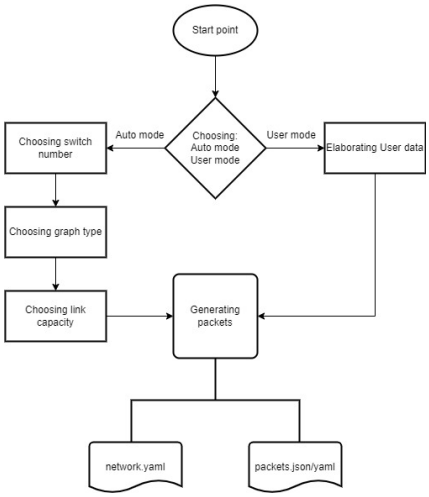


Figura 3.5. *Diagramma di flusso di esecuzione del generatore*

Il generatore si serve di una serie di funzioni di utilità per creare le strutture in base alle scelte dell'utente. Tutte le tipologie di rappresentazioni avvengono seguendo una logica di distribuzione dei nodi in forma matriciale, fatta eccezione della tipologia di grafo completo (spiegheremo nella parte dedicata al visualizzatore il perché), come descritto in dettaglio nel prossimo paragrafo.

3.3 Creazione dei links e nodi

Dopo aver effettuato le scelte via prompt e dopo che il programma ha caricato i parametri dal file `sim_setup.yaml` avviene la creazione dei link e dei nodi. Per i link viene creata una struttura che consiste in un dizionario in cui ogni chiave sarà associata ai parametri del link con un identificativo come chiave:

```
{
    "endpoints": sorted((switch_a, switch_b)),
    "capacity": link_cap,
    "trafficPerc": 0
}
```

Codice 3.7. *Esempio di rappresentazione di un link*

Il campo `endpoints` rappresenta i nodi che il link collega, `capacity` è la capcità del link e `trafficPerc` è la percentuale di traffico del link. Le percentuali di traffico dei link, presenti nel campo `trafficPerc`, vengono aggiornate ogni secondo di generazione, definendo nuove percentuali di traffico da creare. Questo specifico campo non sarà riportato nel `network.yaml` poichè è un dato che non servirà più, difatti la struttura che rappresenta i link verrà memorizzata come una lista di dizionari come nell'esempio 3.8.

```
- capacity: 10
  endpoints:
  - 1
  - 2
- capacity: 10
  endpoints:
  - 2
  - 3
- capacity: 10
  endpoints:
  - 1
  - 4
- capacity: 10
  endpoints:
  - 4
  - 5
```

Codice 3.8. *Esempio di struttura finale dei links*

Il campo `capacity` identifica la capacità del link espressa in Mbps, mentre `endpoints` è una lista di due identificatori numerici compresi tra 2 e 1000 che rappresentano i nodi collegati tramite il link. Ci sono diverse configuazioni che lo script sarà tenuto a considerare in base alle scelte dell'utente, possiamo individuarle in:

- **Modalità auto:** ci sono 3 configurazioni possibili, complete graph, mesh e torus.
  - **complete graph:** i links vengono creati in modo tale che ogni nodo sia collegato a tutti i nodi rimanenti. Non c'è un riferimento esplicito al posizionamento poiché, come vedremo nella sezione 5.1, sarà compito del visualizzatore grafico disporre i nodi nello spazio.
  - **mesh graph:** i links vengono creati in modo tale che ogni nodo abbia 4 nodi adiacenti rispetto a una distribuzione spaziale su matrice rettangolare, a eccezione dei nodi che risiedono alle estremità.
  - **torus graph:** i links vengono creati in modo da replicare il più possibile la struttura di un torus graph completo, come mostrato in Fig. 3.1 utilizzando lo stesso sistema di disposizione usato per il mesh graph basato su matrice rettandolare. Qualora il numero di nodi scelto non riesca a riempire la matrice, l'algoritmo non creerà i link di nodi che non risultino essere alle estremità.
- **Modalità user:** i links vengono semplicemente caricati dal file e copiati nella struttura. Nel caso in cui i link non siano specificati nel custom file allora vengono generati.

In questa fase i nodi (ovvero gli endpoints) sono definiti tramite il campo `endpoints` dei vari links, solo successivamente verrà aggiunto l'indirizzo di ciascun endpoint, un suo nome e l'id. Nel caso di modalità auto vengono creati indirizzi automaticamente a partire da 10.0.0.1, l'identificativo è un valore numerico crescente a partire da 1 e il nome è semplicemente la composizione della parola *switch* seguita dall'id, come ad esempio *switch*1. Nel caso di modalità user tutte le informazioni inerenti agli switch saranno copiate nella struttura così come sono memorizzate nel file `custom_graph.yaml`

3.4 Distribuzione dei nodi: aspect ratio

La distribuzione dei nodi per i tipi di grafi mesh, torus e modalità libera è pensata per essere disegnata in forma matriciale. Questo segue una precisa scelta di design e riguarda l'aspect ratio adotatto per disegnare al meglio gli elementi su schermo tramite la libreria Manim, la quale produce video in 16:9, quindi è sorta l'esigenza di sfruttare al massimo questo formato visivo per poter inserire nel modo più coerente e ottimizzato possibile i vari elementi della rete a schermo. La soluzione adottata è stata quella di pensare al numero di nodi come un'area di un quadrato (matrice quadrata) e successivamente trovare l'area equivalente di un rettangolo (matrice rettangolare) considerando un aspect ratio di 16:9. Consideriamo il numero di nodi A come l'area del quadrato e con n un suo lato $n = \sqrt{A}$. Denotiamo con l la lunghezza e con h l'altezza del rettangolo. La condizione di proporzione si può esprimere come

$$\frac{l}{h} = \frac{16}{9}.$$

Dato che l'area del rettangolo deve essere uguale a quella del quadrato, abbiamo che

$$l \cdot h = n^2.$$

Utilizzando la proporzione, possiamo esprimere l in termini di h come

$$l = \frac{16}{9}h.$$

Sostituendo questa espressione nell'equazione dell'area, otteniamo

$$\frac{16}{9}h \cdot h = n^2,$$

che si semplifica in

$$\frac{16}{9}h^2 = n^2.$$

Da qui, isoliamo h ottenendo

$$h^2 = \frac{9}{16}n^2 \quad \Longrightarrow \quad h = n \cdot \frac{3}{4}.$$

Risostituendo il valore di h nell'espressione di l, abbiamo

$$l = \frac{16}{9} \cdot n \cdot \frac{3}{4} = n \cdot \frac{4}{3}.$$

Abbiamo ottenuto la base l, corrispondente al numero di colonne della matrice, e l'altezza h corrispondente al numero di righe del rettangolo in 16:9. Tuttavia c'è l'esigenza di rappresentare numeri interi poiché la radice dell'area A può non essere un numero intero e di conseguenza anche l e h. Per ovviare al problema si è deciso di arrotondare l e h all'intero superiore, così facendo, qualora si avesse il lato s avente parte decimale, sarà sempre verificato:

$$(l+1)(h+1) > lh \tag{3.2}$$

riuscendo a contenere l'area e rimanendo in proporzione.

3.5 Generazione del traffico

Successivamente alla creazione dei link e dei nodi vi è la generazione del traffico, ovvero la creazione di tutti i pacchetti che vengono creati. Descriveremo ora il criterio di creazione del traffico; l'idea è quella di iterare su frazioni di tempo la cui unità temporale è definita dal campo `creationDelta` (tutti i campi che citeremo provengono dal file setup `sim_setup.yaml`). L'algoritmo cicla per ogni frazione di tempo per la durata della generazione, definita nel campo `simTime` e per ciascuna calcola un quantitativo di pacchetti, proporzionale alla percentuale di traffico, per ogni link presente nella rete. Ogni secondo trascorso viene cambiata la percentuale di traffico di ogni link secondo la scelta fatta nel campo `trafficVariation`. Nei grafici nelle figure 3.6, 3.7 possiamo vedere due esempi di andamento di variazione di traffico percentuale. Il tempo, data e ora, che definisce quando è cominciata la trasmissione lo ritroviamo nel timestamp `startSimTime`. Poichè per ogni frazione il calcolo dei pacchetti può non essere preciso, ovvero può capitare che ci sia un numero con una parte decimale, l'algoritmo provvede a recupere questa rimanenza, tramutarla in pacchetto e aggiungerla. Questo fa si di avere un'elevata approssimazione, tendente al 100% per quanto riguarda l'essere più fedeli possibili alle percentuali di traffico da generare.

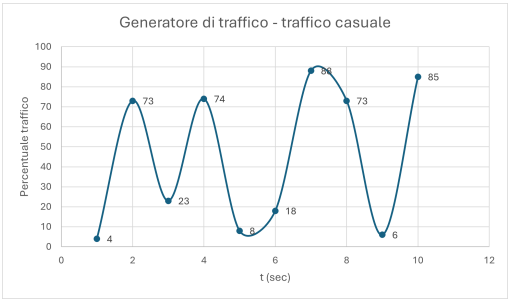


Figura 3.6. *Esempio di variazione di traffico percentuale casuale per 10 secondi*

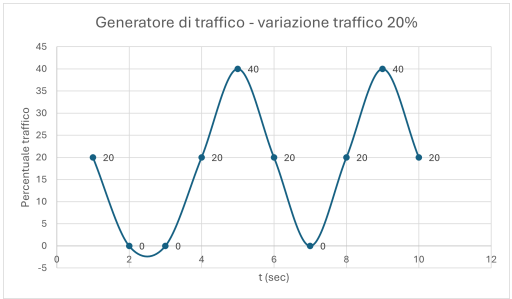


Figura 3.7. *Esempio di variazione di traffico percentuale fissato per 10 secondi*

3.6 Files prodotti

Una volta creati links e nodi, generato il traffico, costruito le strutture del traffico e della rete, lo script provvederà a salvare il risultato nei files descritti precedentemente. Un esempio di file `network` lo ritroviamo in figura 3.9.

```
- - capacity: 10
    endpoints:
    - 1
    - 2
  - capacity: 10
    endpoints:
    - 1
    - 3
  - capacity: 10
    endpoints:
    - 3
    - 4
  - capacity: 10
    endpoints:
    - 2
    - 4
- - address: 10.0.0.1
    switchID: 1
    switchName: switch1
  - address: 10.0.0.2
    switchID: 2
    switchName: switch2
  - address: 10.0.0.3
    switchID: 3
    switchName: switch3
  - address: 10.0.0.4
    switchID: 4
    switchName: switch4
- averageDelta: 1000
  colorblind: 'no'
  dotsSize: fixed
  graphType: mesh
  linkCap: 10
  packetsFile: json
  simTime: 5
  startSimTime: 2024-03-22 12:30:00
  updateDelta: 100
- coordinates:
  - - 1
    - 2
  - - 3
    - 4
- 2024-03-22 12:30:01: phase1
```

Codice 3.9. *Esempio di network file*

Il file è una lista contenente tutti i valori relativi alla configurazione network, avendo i seguenti campi nell'ordine:

- Una lista di links ognuno dei quali descrive la capacità e gli endpoints al quale è collegato
- Una lista di switch ognuno dei quali descrive l'indirizzo, il proprio identificativo e un eventuale nome associato
- Un oggetto rappresentante un dizionario con i parametri di rete:
  - `averageDelta`: l'intervallo di tempo delle medie percentuali di traffico da calcolare con l'analizzatore espresso in millisecondi
  - `colorblind`: yes/no, abilita nel caso di scelta "yes" una palette cromatica nel visualizzatore grafico adatta a persone affette da daltonismo

  - `dotSize`: fixed/adaptive, un'opzione che rende uguale la dimensione a tutti gli elementi raffiguranti gli switch nel visualizzatore grafico (fixed), oppure che adatta la dimensione alla lunghezza dell'identificativo degli switch (adaptive)
  - `graphType`: la tipologia del grafo da visualizzare (complete, mesh, torus, graph)
  - `linkCap`: valore compreso tra [10, 100, 1000]/mixed, un valore che identifica la capacità dei links qualora fossero tutti della stessa grandezza, mixed altrimenti
  - `packetsFile`: `json/yaml`, la scelta del formato con il quale si vuole memorizzare il traffico di rete
  - `simTime`: la durata complessiva delle operazioni di rete
  - `startSimTime`: timestamp, data e ora nel formato YY:MM:DD HH:MM:SS dell'inizio della trasmissione del traffico
  - `updateDelta`: un intervallo temporale che identifica ogni quanto la media `averageDelta` debba essere aggiornata

- Una lista di liste che descrive la matrice del posizionamento degli switch.Nel caso di scelta di grafo completo questo campo è una lista vuota ([])
- Una lista di fasi temporali che descrivono specifici intervalli temporali nel formato "YY:MM:DD HH:MM:SS = descrizione fase"

Il generatore crea delle fasi descrittive ogni 10 secondi di generazione e le memorizza come ultimo valore della lista della struttura network. Per il file packets abbiamo invece due possibili risultati in base alla scelta fatta nel file di setup 3.1, ovvero `json` e `yaml`. Il generatore crea i pacchetti secondo l'assunzione in cui essi vengano registrati in ordine di invio, avremo quindi una struttura corrispondente a una lista di pacchetti ordinati per timestamp di invio. Le strutture si presentano come in figura 3.10 per il formato `json` e in figura 3.11 per il formato `yaml`. Come vedremo nel prossimo capitolo, questi due files rappresentano l'esatto formato che riesce a leggere l'analizzatore, quindi un utente che voglia usare l'analizzatore e il visualizzatore grafico dovrà formattare accuratamente i suoi dati come descritto.


```
[
    {
        "A": 1,
        "B": 2,
        "t": "2024-03-22 12:30:00",
        "d": 4000
    },
    {
        "A": 2,
        "B": 1,
        "t": "2024-03-22 12:30:00",
        "d": 4000
    },
    {
        "A": 2,
        "B": 1,
        "t": "2024-03-22 12:30:00",
        "d": 4000
```

```
    }
]
```

Codice 3.10. *Esempio di packets file in formato* `json`

```
  - A: 1
    B: 2
    d: 4000
    t: &id001 2024-03-22 12:30:00
  - A: 1
    B: 2
    d: 4000
    t: *id001
  - A: 1
    B: 2
    d: 4000
    t: *id001
```

Codice 3.11. *Esempio di packets file in formato* `yaml`

In entrambi i casi abbiamo i seguenti campi:

- A: l'endpoint di partenza del pacchetto
- B: l'endpoint di arrivo del pacchetto
- d: la dimensione del payload del pacchetto espresso in bytes
- t: il timestamp rappresentante la data e l'ora di invio del pacchetto

La scelta di usare un solo carattere per descrivere i vari campi può sembrare in contrasto con la volontà di rendere leggibile il tutto, ma c'è una valida motivazione dietro. Il risparmio di caratteri, in files che sono potenzialmente molto grandi, è un aspetto cruciale sia per quanto riguarda l'occupazione di memoria, sia per motivi di efficienza. Consideriamo una rappresentazione di un pacchetto più descrittivo come in figura 3.13:

```
{
    "endpointA": 1,
    "endpointB": 2,
    "timestamp": "2024-03-22 12:30:00",
    "dimension": 1518
}
```

Codice 3.12. *Rappresentazione di un pacchetto con descrizione completa*

Avremmo così 37 bytes necessari per i nomi dei campi contro i 4 per la rappresentazione di un pacchetto descritta precedentemente, avendo uno scarto di 33 bytes per pacchetto. Questa quantità, moltiplicata la grande quantità di pacchetti da rappresentare, può pesare non poco. Indipendentemente dal numero di bytes necessari a rappresentare un pacchetto in forma minimale, possiamo fare una stima di quanto sarebbe più grande un file packets avendo descrizioni di nomi più lunghi; supponiamo di avere un grafo completo costituito da 5 switch collegati tra loro con links aventi capienza 1 Gbps (gigabit/s), quindi il numero dei link, per l'equazione in Fig. 3.1, risulta essere pari a 10. Supponiamo che tutta la rete stia trasmettendo al 50% della sua capacità trasmissiva per tutta la durata dell'attività di rete. Considerando una dimensione d di payload pari a 1518 bytes uguale per tutti i pacchetti

trasmessi, definiamo come *pps* i pacchetti al secondo, c il 50% di capacità di un link, in un dato secondo:

$$
\begin{aligned}
1 \text{ Gbps} &= 10^9 \text{ bit} \\
1 \text{ GBps} &= \frac{10^9}{8} = 125,000,000 \text{ bytes} \qquad (1 \text{ byte} = 8 \text{ bit}) \\
\text{c} &= \frac{125,000,000}{2} = 65,500,000 \text{ bytes} \\
\text{d} &= 1518 \text{ bytes} \\
\text{pps} &= \frac{65,500,000 \text{ bytes}}{d} \approx 41172.59 \\
\text{bytes di differenza} &= 41172.59 \times 33 \approx 1.35 \text{ MBytes}
\end{aligned}
$$

Ciò significa che per ogni secondo di traffico avremmo 1.35 MB $\times$ 10 links = 13.5MB in più. In 100 secondi di traffico avremmo 1350 Mbytes, cioè 1.35 GBytes in più per una rete avente soltanto 10 links, quindi con la scelta di rendere mono carattere le descrizioni si ha un notevole risparmio in termini di data storage. Chiaramente in ottica di espansione futura del progetto, avendo strutture più complesse per i pacchetti, potrebbe essere necessario avere elementi maggiormente descrittivi, tuttavia ciò non toglie che una certa cura nella scelta di una descrizione minimale non possa essere comunque adottata in via preferenziale.

3.7 Aspetti realizzativi scartati: yaml block e flow style

Un approccio inizialmente realizzato e poi scartato, è stato quello di poter dare la possibilità di scegliere la modalità di salvataggio del file `packets.yaml`. La libreria PyYAML permette di salvare un file `yaml` secondo due modalità; la prima è definita block style e l'altra flow style. Di default, PyYAML sceglie lo stile di una collection a seconda che abbia a sua volta collection nidificate. Qualora ci fossero collection nidificate verrà assegnato lo stile block, il quale è il formato come lo abbiamo già visto in figura 3.11, altrimenti verrà assegnato lo stile flow che corrisponde al codice in figura.

```
[{A: 2, B: 1, d: 4000, t: &id001 !!timestamp '2024-03-22 12:30:00'}, {A: 1, B
    : 2,
    d: 4000, t: *id001}, {A: 1, B: 2, d: 4000, t: *id001}, {A: 2, B: 1, d:
        4000, t: *id001},
  {A: 2, B: 1, d: 4000, t: *id001}, {A: 2, B: 1, d: 3752, t: *id001}, {A: 1,
      B: 2,
    d: 4000, t: &id002 !!timestamp '2024-03-22 12:30:00.100000'}, {A: 2, B:
        1, d: 4000,
    t: *id002}, {A: 1, B: 2, d: 4000, t: *id002}, {A: 1, B: 2, d: 4000, t: *
        id002},
  {A: 2, B: 1, d: 4000, t: *id002}, {A: 2, B: 1, d: 3752, t: *id002}, {A: 1,
      B: 2,
    d: 4000, t: &id003 !!timestamp '2024-03-22 12:30:00.200000'}, {A: 2, B:
        1, d: 4000,
    t: *id003}, {A: 2, B: 1, d: 4000, t: *id003}, {A: 1, B: 2, d: 4000, t: *
        id003},
```

Codice 3.13. *Rappresentazione dei pacchetti con stile yaml flow style*

Inizialmente si era provato a dare la possibilità di scegliere, tramite file `sim_setup`, tra il formato block e flow poiché in fase di test si è visto che l'opzione flow riduce ulteriormente la dimensione del file `packets.yaml`. Tuttavia i test hanno dimostrato che le differenze delle grandezze dei files e i tempi necessari a realizzarli non erano così diversi tra loro, quindi si è deciso di scartare del tutto l'opzione. Possiamo vedere il risultato del test sulle dimensioni dei files in figura 3.8, dove troviamo sull'asse delle ascisse il numero di switch per test, mentre sull'asse delle ordinate abbiamo la variabile che esprime i megabytes, questo perché essa è la variabile dipendente, ovvero i megabytes dipendono dal numero di switch che viene scelto. Il grafico relativo ai tempi di scrittura/lettura lo troviamo in figura 3.9, in questo caso abbiamo sull'asse delle ascisse sempre il numero di switch per test, mentre sull'asse delle ordinate il tempo (in secondi) necessario per le scritture e letture, poiché il tempo di computazione dipende dal numero di switch scelto. Possiamo notare dai grafici che il risparmio di bytes è di circa 4 MB per il test effettuato su 10 switch, mentre i tempi di esecuzione, sia di scrittura che di lettura, sono praticamente sovrapponibili, infatti l'operazione di scrittura per block e flow style è identico, mentre lievemente più performante è il block style in lettura.

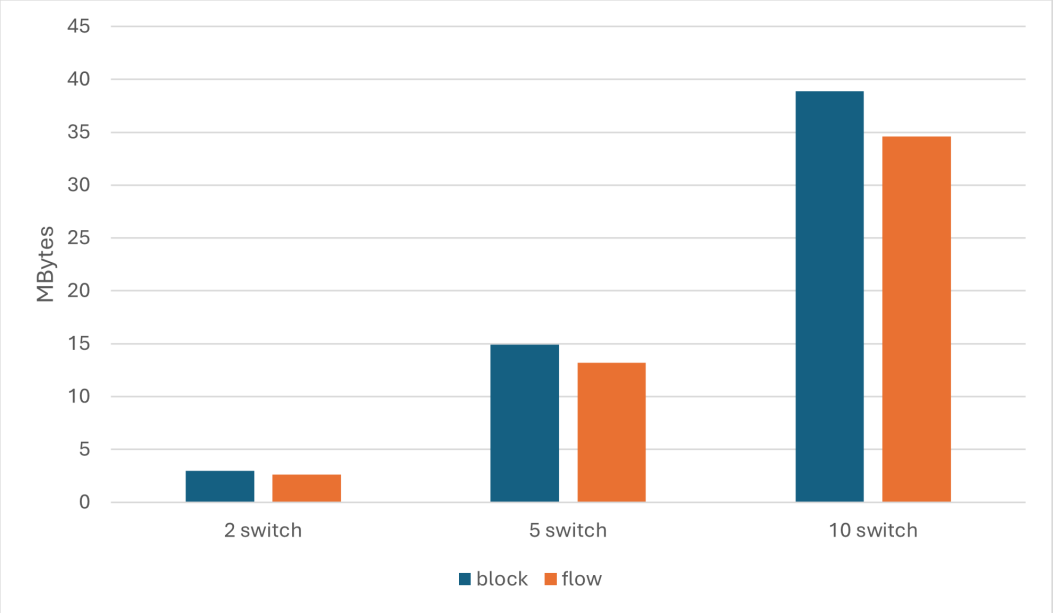
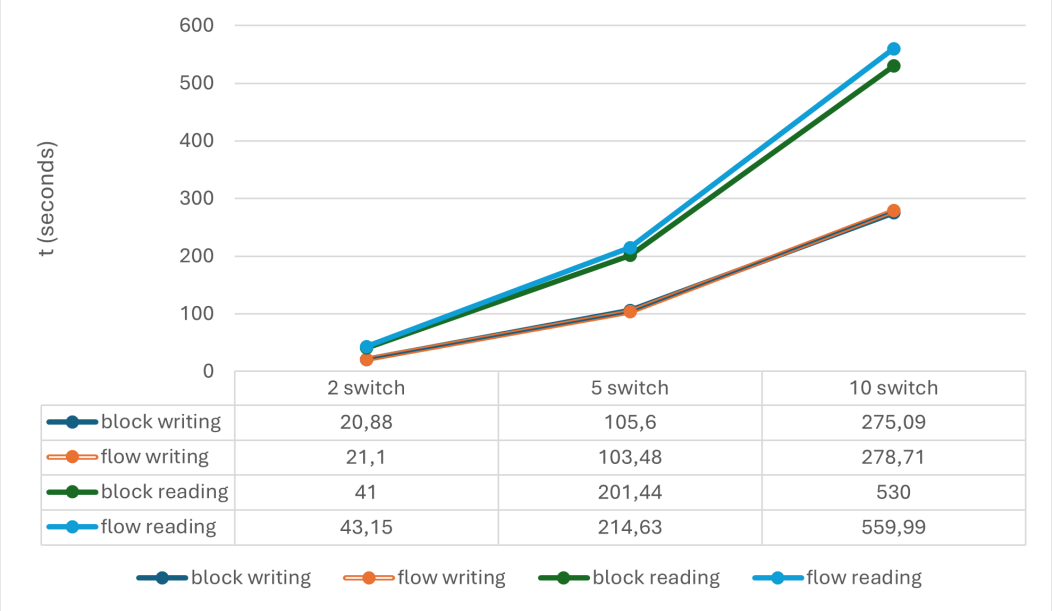


Figura 3.8. *Test* `yaml` *block/flow style, dimensione files*



	2 switch	5 switch	10 switch
block writing	20,88	105,6	275,09
flow writing	21,1	103,48	278,71
block reading	41	201,44	530
flow reading	43,15	214,63	559,99

Figura 3.9. *Test* `yaml` *block/flow style, tempi di scrittura e lettura*

Capitolo 4

Analizzatore di traffico

L'analizzatore di traffico è la seconda anima del progetto essendo la parte che analizza il traffico calcolando le medie percentuali e che produce un file, `analyzed_data.yaml`, che conterrà sia le informazioni di rete come links, switches e parametri di rete, sia le registrazioni del traffico percentuale di ogni link. Per poter operare, l'analizzatore ha bisogno dei due files, `network.yaml` e `packets.yaml`, creati dal generatore o che siano creati e formattati dall'utente. Abbiamo descritto in dettaglio la composizione dei files nella sezione 3.6. L'esecuzione dello script parte in primo luogo caricando i files richiesti e creando le strutture necessarie al calcolo delle medie, dopodichè avviene il calcolo vero e proprio, viene creata la struttura che servirà come input del visualizzatore grafico e infine viene creato il file `analyzed_data.yaml` con tale struttura. Poichè per motivi di efficienza si è scelto di strutturare i link come una lista nel file `network.yaml` e non come dizionario (in cui ogni link avrebbe avuto un identificativo come chiave), si è presentato il problema dell'identificazione dei link. I link in fin dei conti possono essere identificati dai propri endpoints (il grafo viene inteso in modo tale che ci sia un unico link tra due endpoints), ovvero una coppia di nodi, nel nostro caso, una coppia di switch. Tuttavia la sfida che si è presentata è rappresentata dal fatto che se è vero quindi che un link è identificato da due endpoints, a e b, è vero anche che in questo caso l'ordinamento degli endpoints non deve essere rilevante. Abbiamo infatti pacchetti che vengono inviati sia da a per poi arrivare a b, sia da b verso a. Questo è un problema poiché stiamo cercando di identificare un link ma abbiamo due coppie, (a, b) e (b, a) che sì sono ugualmente valide ma rappresentano anche due coppie distinte, il che contrasta con il concetto di identificatore. L'ideale sarebbe usare il tipo `set`, ovvero il concetto di insieme come collezione non ordinata in cui due insiemi $\{a, b\}, \{b, a\}$ sono considerati uguali in termini di contenuto. Sembrerebbe un problema risolto, ma non è così, il perché risiede nel fatto che i set in Python sono mutabili, cioè il contenuto può essere modificato utilizzando metodi come `add()` e `remove()`. Poiché set è mutabile, non ha valore hash [16] e non può essere utilizzato come chiave del dizionario [14]. Il tipo di dato che fa al caso nostro è quindi il frozenset, un set che è immutabile e hashable, quindi utilizzabile come chiave di un dizionario.

Abbiamo assunto in fase di progettazione di avere link bidirezionali, il che è ragionevole nelle reti di calcolatori, tuttavia le soluzioni trovate per la creazione dei links non precludono l'utilizzo del software per casi di reti con link unidirezionali,

il modello infatti è facilmente generalizzabile nel caso in cui si volessero modellare altri tipi di reti che non sono reti di calcolatori e per cui potrebbe aver senso utilizzare dei link unidirezionali. Nell'analizzatore l'identificazione dei link è molto importante, perché nel momento in cui si va a scorrere la lista di tutti i pacchetti registrati vi è il bisogno di identificare il link per potergli attribuire le varie medie da calcolare. Viene quindi creata una struttura ausiliaria in cui la lista di link presente nel file `network.yaml` viene tramutata in un dizionario avente come chiavi dei frozenset con gli endpoints e, come valori associati, una serie di variabili atte a mantenere le somme dei pacchetti che man mano verrano scanditi. Segue la scansione dei pacchetti in cui, per ogni unità frazionaria di tempo dettata dal campo `updateDelta` presente in `network.yaml`, vengono sommati tutti i bytes di tutti i pacchetti e le somme attribuite ai link corrispettivi. Una volta finita la scansione, la struttura ausiliaria viene scandita per ricavare le medie percentuali delle varie somme calcolate dal passaggio precedente e viene creata la struttura che verrà memorizzata in `analyzed_data.yaml`. La struttura memorizzata è raffigurata in figura 4.2.

```
{
  - averageDelta: 1000
    simStartTime: 2024-03-22 12:30:00
    simTime: 2
    updateDelta: 100
  - - endpoints:
      - 1
      - 2
      traffic:
      - 59.0
      - 53.5
      - 48.0
      - 42.5
      - 37.0
      - 31.5
      - 26.0
      - 20.5
      - 15.0
      - 9.5
      - 4.0
    - endpoints:
      - 1
      - 3
      traffic:
      - 33.0
      - 38.8
      - 44.6
      - 50.4
      - 56.2
      - 62.0
      - 67.8
      - 73.6
      - 79.4
      - 85.2
      - 91.0
}
```

Codice 4.1. *Rappresentazione della struttura analyzed_data*

La struttura consiste in una lista di due elementi, il primo è un dizionario rappresentate i parametri di rete necessari al visualizzatore grafico, il secondo è una lista di dizionari rappresentante i link, definiti dal campo `endpoints`, e le loro medie percentuali memorizzate nel campo `traffic`. L'unico campo che non abbiamo anco-

ra incontrato finora è `traffic`, il quale rappresenta le medie percentuali di tempo `averageDelta`, calcolate ogni `updateDelta` millisecondi. Nell esempio in figura 4.2 abbiamo che le medie vengono calcolate quindi in intervalli di 1000 ms (millisecondi), aggiornati ogni 100 ms. Prendiamo ad esempio il primo link identificato dalla coppia di endpoints $(1, 2)$, i suoi valori del campo traffic sono associati ai seguenti tempi:

```
2024-03-22 12:30:01 - 59
2024-03-22 12:30:01.100000 - 53,5
2024-03-22 12:30:01.200000 - 48
2024-03-22 12:30:01.300000 - 42,5
2024-03-22 12:30:01.400000 - 37
2024-03-22 12:30:01.500000 - 31,5
2024-03-22 12:30:01.600000 - 26
2024-03-22 12:30:01.700000 - 20,5
2024-03-22 12:30:01.800000 - 15
2024-03-22 12:30:01.900000 - 9,5
2024-03-22 12:30:02 - 4
```

Codice 4.2. *Timestamps e medie percentuali calcolate dall'analizzatore*

I timestamps sono espressi nel formato YY:MM:DD HH:MM:SS.Microseconds, questo per via dell'impiego dell'oggetto `timeDelta` [15] che converte un millisecondo in 1000 microsecondi e con cui è possibile effettuare operazioni aritmetiche di somma e differenza tra due oggetti di tipo `datetime` [15]. Il primo valore che abbiamo, 59, è quindi la media percentuale dei pacchetti inviati tra il tempo 12:30:00 e 12:30:01, il secondo valore, 53.3 è la media dell'intervallo 12:30:00.100000 e 12:30:01.100000, e così via. Il file prodotto `analyzed_data.yaml` sarà l'input necessario per il visualizzatore grafico, che verrà discusso nel capitolo successivo.

Capitolo 5

Visualizzatore grafico

La terza anima del progetto è il visualizzatore grafico sviluppato tramite l'ausilio della libreria Manim. Andremo a descrivere il funzionamento dello script, come è strutturato e organizzato, in seguito descriveremo le modalità grafiche dei grafi che è possibile ottenere, inclusa la rappresentazione video delle informazioni relative ai switch e link. Seguirà una sezione sulle scelte di design servite a curare l'aspetto visivo e infine ci sarà spazio per l'accessibilità inclusa nel progetto che prevede un'opzione per daltonici e che consente quindi di cambiare i colori che rappresentano il traffico.

5.1 Strutturazione e organizzazione del visualizzatore

Andremo ora a descrivere come è strutturato il codice a livello logico. La prima cosa che salta all'occhio aprendo il progetto potrebbe essere la riga di codice:

```
from manim import *
```

Potrebbe sembrare un errore, o una scelta pigra per velocizzare le importazioni delle sezioni di libreria necessaria, ma in realtà questo è proprio l'uso raccomandato dalla documentazione ufficiale di utilizzare Manim [23]. Una volta avviato lo script verranno caricati i files `analyzed_data.yaml` e `network.yaml`, successivamente verranno impostati i colori del traffico in base alla scelta fatta nel campo `colorblind` all'interno del file `network`, dopodichè possiamo pensare al resto dello script come un bivio in cui ci sono le due funzioni principali; la funzione che renderizzerà il grafo in caso di grafo completo e la funzione che renderizzerà un grafo pensato per essere disposto tramite matrice. La funzione relativa al grafo completo sfrutta la potenza di Manim per la disposizione dei nodi e degli archi (links), provvedendo una disposizione automatica ed organica degli elementi e rappresenta la scelta più indicata per questo tipo di rappresentazione; per farlo bisogna settare il campo `customGraph` nel file `network` al valore "complete" e lasciare il campo `coordinates` vuoto. Per ottenere un grafo completo con coordinate personalizzate bisognerà assegnare a `graphType` il valore "graph" e specificare le coordinate nel campo `coordinates`. Entrambe le funzioni hanno lo stesso pattern di creazione del grafico e di creazione delle animazioni; prima si costruiscono le strutture da visualizzare, inclusi i parametri di rete da visualizzare, successivamente viene eseguita una iterazione sui timestamps

ricavati dal file `analyzed_data` in cui per ogni intervallo di tempo `updateTime` viene presa la parcentuale corrispondente di ciascun link, si prendono i colori associati alle varie percentuali e si crea un'animazione di durata 1 secondo di transizione verso il colore scelto e si ripete fino a quando non si arriva a fine `simTime`. Per dare una maggiore idea sul come vengono create le forme e le animazioni in Manim, faremo una breve introduzione alla libreria. In Manim ogni oggetto viene definito Mobject, ovvero un *Mathematical Object*, e deriva dalla classe di base Mobject, cioè un oggetto che può essere visualizzato a schermo, per esempio una forma rettangolo, una linea o un cerchio, vengono definiti tutti mobject. La classe `Scene` è la tela sul quale si disegnano i mobjects, una volta istanziati essi vanno aggiunti alla `Scene` per poter essere visualizzati, ciò e possibile tramite il metodo `add()`. Prendiamo il primo esempio fornito dalla documentazione Manim:

```
{
    from manim import *

    class CreatingMobjects(Scene):
        def construct(self):
            circle = Circle()
            self.add(circle)
            self.wait(1)
            self.remove(circle)
}
```

Codice 5.1. *Un primo esempio di utilizzo di Manim [23]*

Come descrive la documentazione ufficiale, generalmente tutto il codice che descrive un video va all'interno del metodo `construct()` della classe `Scene`, per poter visualizzare un mobject si chiama il metodo `add()` della `Scene` contenuta, in questo caso passiamo un mobject `circle` come parametro, aggiungendolo alla `Scene`. Il metodo `wait(1)` fa passare un secondo di tempo e `remove(circle)` lo rimuove. Si intuisce da subito le potenzialità che offre la libreria ma sono le animazioni il cuore di Manim. Ogni proprietà di un mobject che può essere modificata, può essere animata. Infatti ogni metodo che cambia una proprietà di un mobject può essere usato come un'animazione tramite l'uso di `animate()`.

```
{
    from manim import *

    class AnimateExample(Scene):
        def construct(self):
            square = Square().set_fill(RED, opacity=1.0)
            self.add(square)

            # animate the change of color
            self.play(square.animate.set_fill(WHITE))
            self.wait(1)

            # animate the change of position and the rotation
            self.play(square.animate.shift(UP).rotate(PI / 3))
            self.wait(1)
}
```

Codice 5.2. *Un primo esempio di animazioni in Manim [23]*

Nel codice in Fig. 5.2, vediamo come impostare il colore rosso e il livello di opacità al mobject `Square()`, applicando poi il metodo `square.animate.set_fill(WHITE)` viene creata l'animazione di transizione di colore da rosso a bianco. L'animazione

creata deve essere aggiunta alla `Scene` per poterla visualizzare e lo si fa tramite `play()`. Questa breve introduzione a Manim ci da già l'idea della potenza che offre ponendo come unico limite la creatività dello sviluppatore. Ritornando alla descrizione delle funzioni principali, il grafo completo della funzione dedicata, viene creato tramite la classe `Graph()`, in cui due dei principali parametri richiesti sono una lista di valori che identificano i nomi dei nodi e una lista di coppie (tuple) che descrivono gli archi (links). Chiaramente la liste degli switch e delle tuple di endpoints vengono create a monte. Per quanto riguarda la creazione delle animazioni, il cuore risiede nello pseudocodice in Fig. 5.3.

```
animations.append(grafo.edges[(A, B)].animate.set_color(traffic_color))
```

Codice 5.3. *Creazione animazione traffico*

In questo pseudocodice abbiamo una lista di animazioni `animations`, in cui aggiungiamo un'animazione che interessa l'oggetto arco restituito da `grafo.edges` e individuato dalla coppia di endpoints (A, B). Dopodichè definiamo quale proprietà deve essere animata, in questo caso l'impostazione di un nuovo colore tramite `set_color()`, il quale è il colore corrispondente al timestamp correntemente analizzato. Questa operazione viene effettuata per tutti i link per l'intervallo di tempo `updateDelta`. Quando tutte le animazioni sono settate si usa il metodo `play(animation*)` per animarle tutte conteporaneamente e ciò si ripete per la durata del tempo registrato.

Per quanto riguarda la seconda funzione principale, il pattern di creazione è analogo, tuttavia è interessante descrivere il posizionamento dei nodi.

```
for row in range(rows):
    for col in range(cols):
        opacity = 1
        # Switches with ID=0 are considered as empty spaces
        if str(mesh[row][col]) == EMPTY_SPACE:
            opacity = 0
        dot = None
        if network_data[CONST.NETWORK["SIM_PARAMS"]]["dotsSize"]=="adaptive":
            dot = LabeledDot(str(mesh[row][col]),
                             point=np.array([col * spacing, row*-spacing, 0]))
        elif network_data[CONST.NETWORK["SIM_PARAMS"]]["dotsSize"]=="fixed":
            dot = LabeledDot(str(mesh[row][col]),
                             radius=0.6,
                             point=np.array([col*spacing, row*-spacing, 0]))
        dot.set_opacity(opacity)
        graph_mesh[mesh[row][col]] = dot
        mesh_grid.add(dot)
```

Codice 5.4. *Creazione posizionamento nodi nella griglia*

Il codice in figura 5.4 mostra come vengono creati e disposti i nodi in una griglia. Vengono considerati tutti i valori rappresentanti i nodi (switch) presenti nel campo `coordinates` del file `network.yaml`, anche quelli aventi valori zero, i quali vengono comunque aggiunti ma settati con opacità pari a zero non venendo renderizzati nel risultato finale, quindi non peseranno in termini computazionali e fungono da riferimento per i valori adiacenti al pari dei valori diversi da zero. I nodi sono rappresentati da mobject `LabeledDot()`, ovvero dei cerchi pieni con raffigurato al loro interno il valore dell'id dello switch. Al momento di creazione, nel `LabeledDot()`, viene passato anche un array rappresentante le coordinate [x, y, z] che il nodo dovrà

avere. Tutti i nodi vengono aggiunti singolarmente in un mobject `VGroup()` (variabile `mesh_grid`), ovvero un gruppo che può contenere oggetti vettorializzati. Per quanto riguarda invece la creazione dei links, la parte che rappresenta il mesh graph è creata tramite mobjects `Line()`, i quali rappresentano rette che congiungono due punti, avremo così i link definiti come in figura 5.5.

```
line = Line(dot_a.get_center(),
    dot_b.get_center(),
    color=START_COLOR,
    stroke_width=8)
```

Codice 5.5. *Creazione link*

Il mobject prende le coordinate dei nodi da collegare come parametri, oltre al colore di partenza e alla dimesione. Nel caso di grafo torus si aggiunge il controllo dei nodi posti all'estremità della griglia, difatti qui vi è un controllo preciso circa la tipologia di posizionamento poiché se nell'analizzatore di traffico avevamo il problema del rappresentare in un unico modo due tuple con ordinamento diverso, in questo caso l'ordinamento è molto importante per poter disegnare gli archi dei nodi alle estremità, questo perché i link che le collegano devono essere disegnati manualmente creando più segementi concatenati. Il lettore si starà chiedendo se non esistesse un mobject come `Line()` per rappresentare dei link arcati, e in effetti uno c'è ed è `ArcbetweenPoints()`, il quale funzionamento è esattamente come `Line()`. Il problema che è sorto in fase di sviluppo è che, per ridimensionare i grafi quando diventano più grandi dell'area di visualizzazione, si è reso neccessario l'utilizzo del metodo `auto_zoom()` della classe `MovingCameraScene`, ovvero la scena utilizzata nel progetto che permette cambiamenti di visuale della camera. Il ridimensionamento generato automaticamente dal metodo va in conflitto con `ArcsBetweenPoints()`, creando artefatti grafici non desiderati. Si è quindi passati a una realizzazione manuale degli archi che segue la creazione di 3 segmenti con mobjects `Line()` concatenati per rappresentare i link del torus dei nodi posti alle estremità. Perticolare attenzione va data al metodo `get_center()` del mobject `LabeledDot()`, poiché il metodo ritorna le coordinate del nodo, ci servirà per descrivere come vengono creati i link manuali. L'idea di creazione deriva dal considere il centro dei nodi delle estremità come punti di origine di piani cartesiani x, y dai quali ricavare le coordinate dei 3 segmenti, i vari sistemi di coordinate sono;

- **Links verticali:**
  - prima linea: si parte da un nodo posto all'estremità superiore della griglia ottenendo le coordinate con `get_center()` e ne definiamo il punto $pointA$, si prende il punto $p1 = (-x, -y)$, rispetto alle coordinate di $pointA$, e si crea la linea che parte da p1 a $pointA$
  - seconda linea: si prende il nodo posto all'estremità inferiore della griglia nella stessa colonna in cui è presente $pointA$ e, ricavandone le coordinate con il metodo, abbiamo un punto che definiamo come $pointB$, si prende il punto $p2 = (-x, y)$, rispetto alle coordinate di $pointB$, e si crea la linea che parte da p2 a pointB
  - terza linea: si crea una linea che va dal punto p1 al punto p2.

- **Links orizzontali:**
  - prima linea: si parte da un nodo posto all'estremità sinistra della griglia ottenendo le coordinate con `get_center()` e ne definiamo il punto $pointA$, si prende il punto $p1 = (x, -y)$, rispetto alle coordinate di $pointA$, e si crea la linea che parte da p1 a pointA
  - seconda linea: si prende il nodo posto all'estremità destra della griglia nella stessa riga in cui è presente $pointA$ e, ricavandone le coordinate con il metodo, abbiamo un punto che definiamo come $pointB$, si prende il punto $p2 = (-x, -y)$, rispetto alle coordinate di $pointB$, e si crea la linea che parte da p2 a pointB
  - terza linea: si crea una linea che va dal punto p1 al punto p2.

Vien da sè che in questo caso è indispensabile l'ordinameto, poiché per ogni specifico nodo ci sono delle specidiche coordinate per la creazione dei link manuali. Se non si facesse questo tipo di controllo, potremmo avere una situazione come in figura 5.1, ottenendo un grafico molto disorganico e asimmetrico.

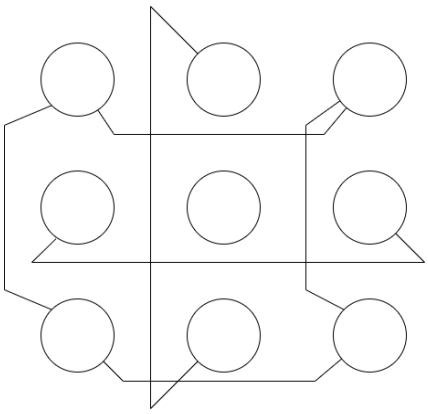


Figura 5.1. *Visualizzazione di archi non ordinati alle estremità di un torus graph*

Per questo motivo viene eseguito un controllo nel caso di analisi di un grafo torus, con l'algoritmo che scansiona la matrice e seleziona gli archi che sono verticali, orizzontali e ne assegna le giuste coordinate. Infine la parte di animazione è pressoché uguale come logica. Per quanto riguarda le informazioni della rete che vengono mostrate a schermo si è implementato un effetto contatore sul tempo che scorre ed è presente un ridimensionamento dinamico dei font che si adatta al crescere del grafico, questo per poter rendere sempre leggibile il testo.

Per visualizzare le informazioni relative agli switch e ai link è stata realizzata una classe apposita, la quale può essere renderizzata separatamente dai grafici, producendo una presentazione animata di switch e links con le relative informazioni. Gli switch, correlati con le loro info, vengono mostrati cinque alla volta con un effetto di comparsa e scomparsa, con effetto slide, mostrando poi i successivi cinque switch e relative info. Una volta terminati gli switch si susseguiranno i link, animati nello stesso modo e comparendo otto alla volta, mostrando gli endpoints al quale sono collegati e le capacità. Mostreremo graficamente ciò nel capitolo dedicato alle modalità grafiche.

5.2 Modalità grafiche e design

In questo capitolo si vogliono mostrare tutte le configurazioni grafiche, i modi in cui l'algoritmo adatta gli elementi a schermo e le soluzioni trovate. Essendo un progetto che si basa fortemente su un aspetto front-end, questo capitolo sarà focalizzato sul mostrare visivamente cosa può produrre il visualizzatore nelle varie configurazioni possibili. Partendo con la scelta di design dei colori che rappresentano le intensità di traffico, la scelta è ricaduta su colori ai quali generalmente associamo significati abitudinari e comuni, come il verde per indicare traffico zero o basso, per poi arrivare al giallo per indicare un traffico di intensità media, fino ad arrivare al rosso per indicare il traffico molto intenso o massimo. Le varie gradazioni sono calcolate a partire da valori rgb e distribuiti su una scala che va da 1 (verde) a 100 (rosso), modificando opportunamente le graduazioni intermedie a cavallo tra le diverse colorazioni. Parlando invece del grafo completo, come descritto precedentemente, può essere disposto in modo automatico oppure impostato tramite coordinate dei nodi dall'utente. Considerandone un esempio, sulla base dello stesso packets file, i due modi di visualizzare il grafo completo possono apparire come nella Fig. 5.2 per l'auto-posizionamento e come in fig 5.3 per il posizionamento manuale.

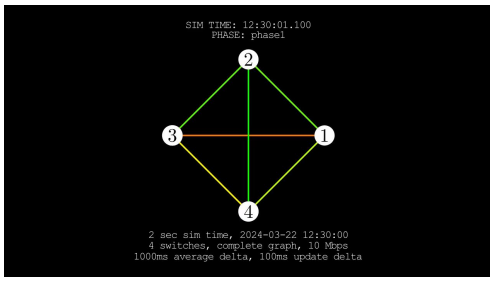


Figura 5.2. *Visualizzazione per opzione grafo completo con disposizione automatica*

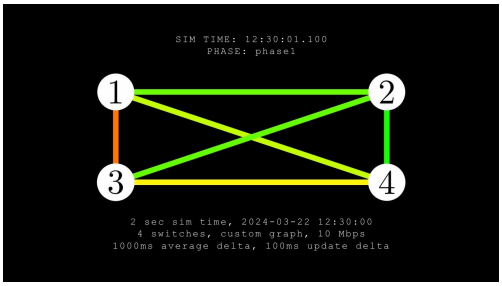


Figura 5.3. *Visualizzazione per opzione grafo completo con coordinate personalizzate*

La scelta di design dietro la visualizzazione del grafo completo in modo circolare è stata fatta per motivi di leggibilità. Manim propone diversi layout di configurazione, come possiamo vedere dall'immagine presa dalla documentazione ufficiale [3] in Fig. 5.4. Dagli esempi si nota subito che l'opzione circolare è quella meglio distribuita, il che è ragionevolmente utile quando si tratta di disegnare molti elementi a schermo. I layouts possibili come scelta sono, in ordine di apparizione; spring, circular, kamada kawai, planar, random, shell, spectral e spiral.

Un esempio di grafo completo con modalità auto, settato con 10 switch in modalità circular, lo troviamo in figura 5.5.

Come abbiamo detto nella sezione 3.4, la modalità auto del generatore, per grafi mesh, torus e personalizzati, gestisce l'aspect ratio sulla base del numero di switch scelti, vi sono quindi due risultati possibili di visualizzazione; il primo è che il numero di switch combacia con il calcolo dell'aspect ratio e fornisce una riempimento completo della matrice da visualizzare (Fig. 5.7), il secondo coincide con un riempimento parziale e in questo caso l'algoritmo lascerà semplicemente dello spazio vuoto (Fig. 5.9). La figura 5.6 mostra inoltre il comportamento del

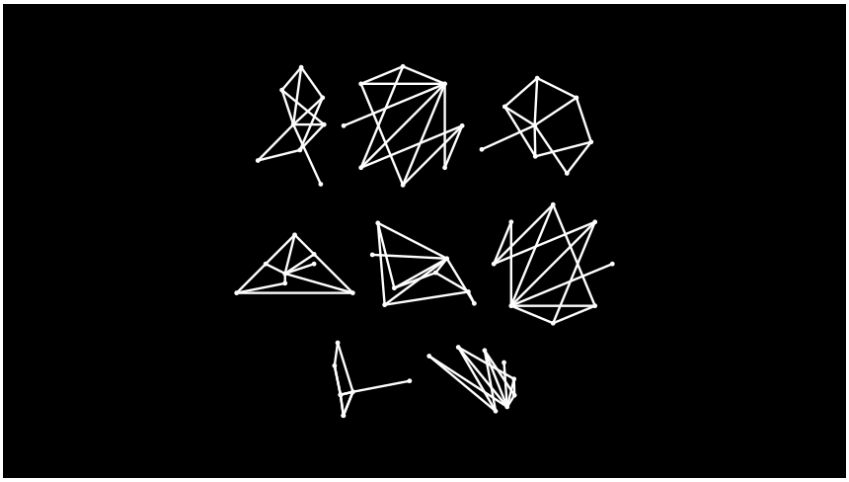


Figura 5.4. *Layouts possibili per Graph()*

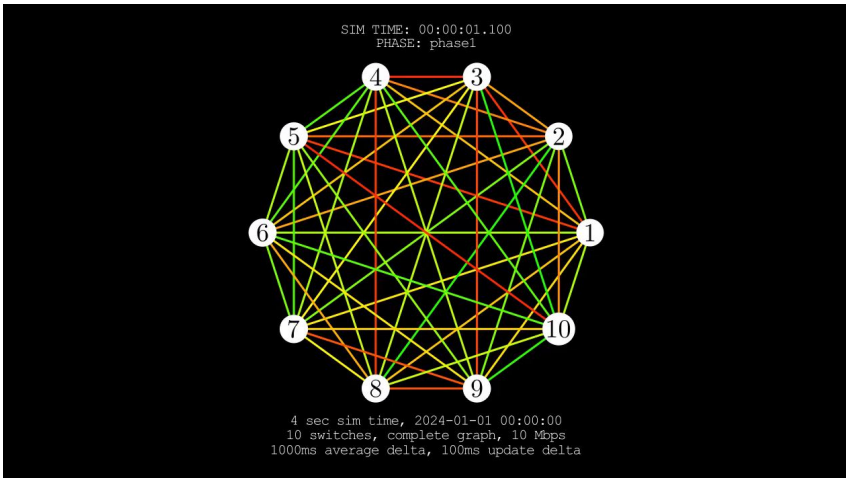


Figura 5.5. *Visualizzazione grafo completo con disposizione automatica di 10 switch*

grafo qualora si decidesse di settare il parametro `fixedDots` al valore `fixed`, il quale rende la dimensione dei nodi uguale per tutti, la stessa tipologia di grafo con il parametro settato ad `adaptive` adatterà la dimensione dei nodi in base all'id con il quale vengono rappresentati gli switch, possiamo vedere un esempio in figura 5.7.

La logica di riempimento vale anche per il torus graph (Fig. 3.1), con l'unica differenza che nel caso di mancato riempimento vengono collegati solo i nodi che risultano essere posti alle estremità. Questa soluzione ibrida segue la convenzione per cui generalmente il torus graph è inteso come completo, ovvero che riempie tutte le posizioni della matrice immaginaria sul quale ci sono i nodi.

Osservando gli esempi del torus graph, si può notare come i link dei nodi delle estremità siano meno spessi degli altri che formano la griglia. Questa è una scelta di design volta a rendere maggiormente leggibile il grafo, in questo modo si percepisce subito la tipologia di link a un primo sguardo e il posizionamento dei punti che definiscono i link sono stati studiati appositamente per fare in modo che non si sovrapponessero, mantenendo l'immagine pulita e organizzata. Per quanto

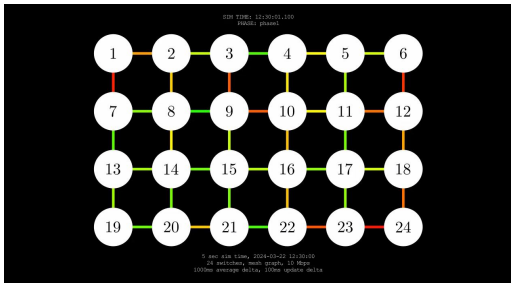


Figura 5.6. *Visualizzazione grafo mesh completo, opzione dotsSize posta a fixed*

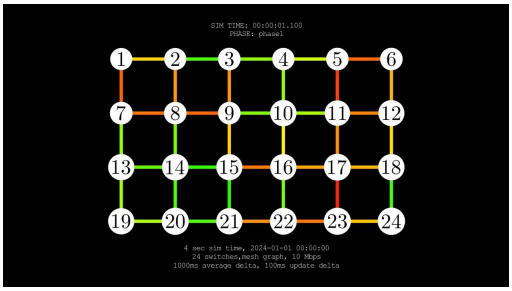


Figura 5.7. *Visualizzazione grafo mesh completo, opzione dotsSize posta ad adaptive*

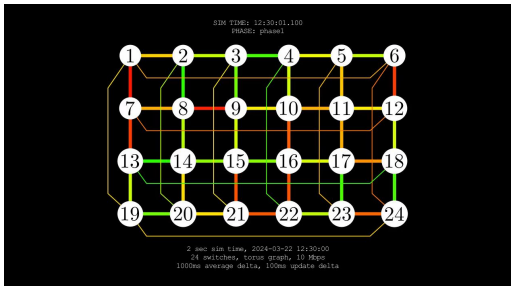


Figura 5.8. *Grafo torus completo*

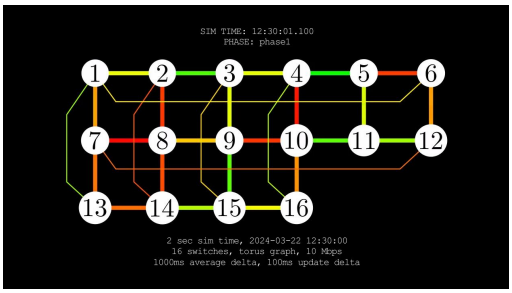


Figura 5.9. *Grafo torus non completo*

riguarda la scelta di design dietro il testo visualizzato a schermo, è stato scelto di suddividere le informazioni e distribuirle in due gruppi, uno posto nel margine superiore e l'altro in quello inferiore. Il motivo è per dare enfasi al grafico, che rimane centrale nell'economia dell'esperienza, in questo modo il focus principale rimane sul protagonista della scena, il grafo, mentre i parametri descrittivi della rete sono presenti in modo tale da non essere invasivi. Infine abbiamo la modalità "graph", la modalità in cui l'utente può scegliere dove posizionare i nodi come già visto in fig 3.4. Il visualizzatore possiede anche una modalità per generare un video in cui si mostrano le informazioni degli switch e nei nodi come spiegato nella sezione precedente 5.1. Per dare un'idea di come viene mostrato, riportiamo due frammenti di video in cui vengono mostrate le varie tipologie di dati (Fig. 5.10 e Fig. 5.11).

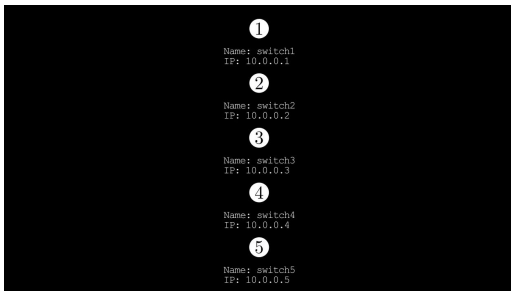


Figura 5.10. *Switch info*

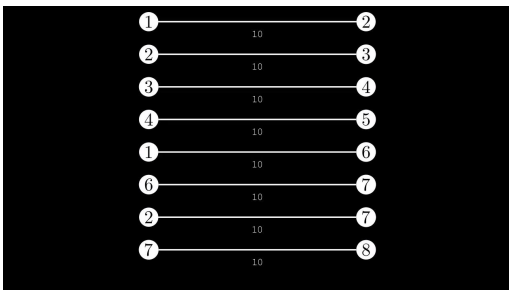


Figura 5.11. *Link info*

Per quanto riguarda la visualizzazione dello scorrimento del tempo, l'applicazione è pensata per far sì che, ogni secondo di video, l'applicazione mostri la percentuale di traffico relativa di ciascun intervallo di tempo `updateDelta`. Per esempio, se avessimo un valore di `updateDelta` pari a 100 ms, avremmo 10 secondi di video per mostrare l'andamento del traffico avvenuto in un secondo di trasmissione. Questo intervallo temporale di un secondo è un valore di default dell'applicazione, tuttavia può essere facilmente modificato per ottenere video più lunghi, ponendo per esempio a 2 secondi il tempo di rendering di ciascun aggiornamento delle medie percentuali.

5.3 Accessibilità: modalità daltonismo

In un'applicazione in cui i colori sono parte integrante dell'esperienza, è più che opportuno approcciarsi alla progettazione tenendo in considerazione delle possibili implementazioni che possano affrontare la questione del daltonismo. Il daltonismo è una condizione in cui l'occhio umano presenta cecità ereditaria verso uno o più colori. Le forme più comuni sono:

- **protanopia:** sensibilità deficitaria per il colore rosso.
- **deuteranopia:** sendibilità deficitaria per il colore verde.
- **tritanopia:** sensibilità deficitaria per i colori giallo e blu.

Come detto nel capitolo precedente, utilizziamo i colori verde, giallo e rosso per esprimere rispettivamente basso, medio e alto traffico. Andando a vedere le varie tipologie più comuni di daltonismo, notiamo da subito che i colori scelti non sono adatti a tutti i tipi di daltonismo. Si è pensato quindi di introdurre l'opzione `colorblind` che se posta su "yes" cambia i colori in; verde per un basso traffico, bianco per traffico medio e fucsia per elevato traffico. Il colore bianco e nero sono visti in egual modo da tutti, questo è anche il motivo per cui lo sfondo del visualizzatore è proprio nero, quindi si è deciso di utilizzare il bianco come uno dei colori adatti, mentre per gli altri colori si è fatto ricorso al tool *Coloring for Colorblindness* [5] del Professor Nichols, docente dell' Università del Connecticut, il quale permette di visualizzare una palette cromatica nel modo in cui i vari tipi di daltonismo si manifestano. Mostriamo le due palette cromatiche, quella di default 5.12 e quella scelta per l'opzione daltonismo 5.13.

Come possiamo notare dalle palette cromatiche, quella di default costituirebbe un grosso problema visivo per chi è affetto da protanopia e da deuteranopia, avendo una selezione di colori davvero molto simili tra loro non si potrebbe distinguere efficacemente il livello di traffico. La soluzione proposta risiede nella palette cromatica per daltonismo, in cui i colori scelti si riescono a differenziare per ogni tipologia di condizione visiva, potendo così attribuire al colore verde un livello di traffico nullo o basso, bianco per traffico medio e fucsia per traffico alto o massimo. Un esempio del risultato finale lo possiamo vedere in figura 5.14.

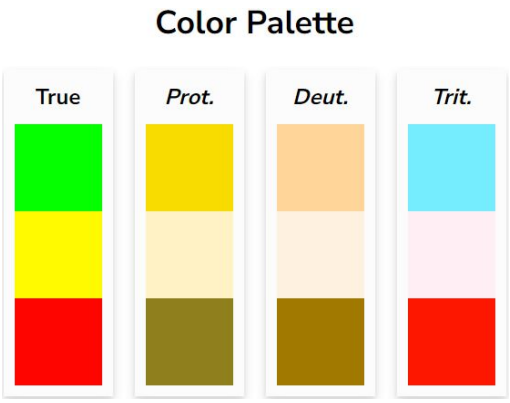


Figura 5.12. *Palette cromatica di default*

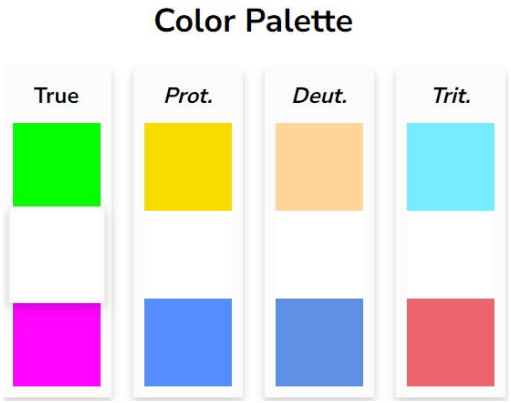


Figura 5.13. *Palette cromatica daltonismo*

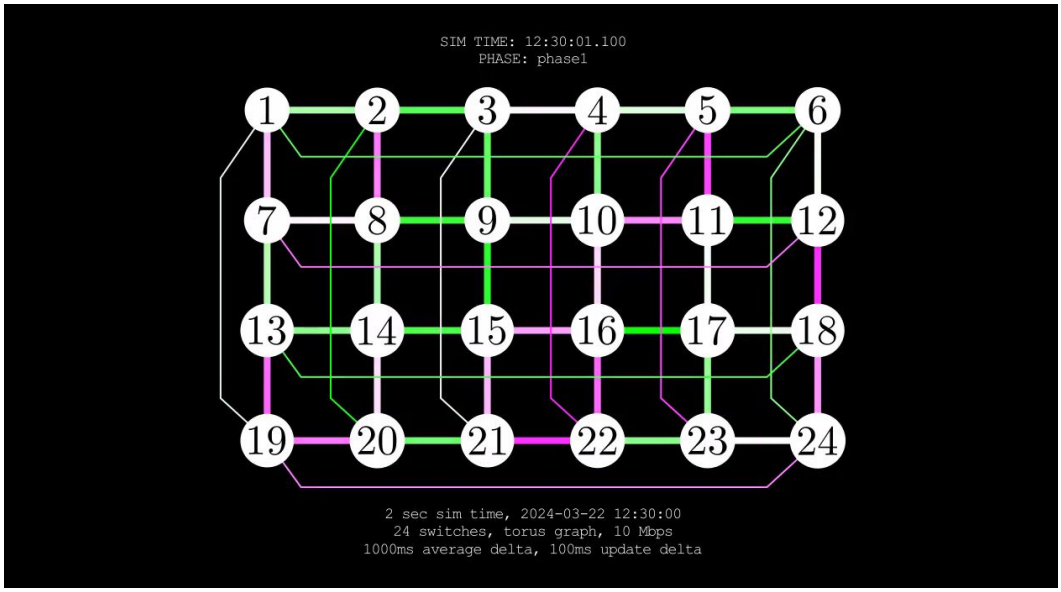


Figura 5.14. *Opzione* `colorblind` *per torus graph*

Capitolo 6

Prestazioni e complessità computazionale

6.1 Complessità computazionale

Andremo ora ad analizzare le complessità computazionali dei vari componenti del progetto, mostrando anche le prestazioni ottenute in fase di test.

6.1.1 Generatore di traffico

Analizzando il codice inerente al generatore, un approccio allo studio della complessità potrebbe essere quello di individuare quelli che si ritengono essere i punti computazionalmente critici, al fine di individuare l'andamento degli algoritmi nelle varie situazioni. Essendo lo script dedicato alla creazione della struttura di rete e del traffico, è stato ritenuto opportuno cominciare l'analisi dalla creazione della struttura inerente ai links. Per quanto riguarda la creazione automatica dei links e dei nodi per la tipologia di grafo completo, la funzione dedicata al compito è riportata in figura 6.1

```
    def create_auto_complete_links(link_cap, switch_number):
        # The arcs representing the links connecting the switches (nodes)
        links = {}
        # The link ID counter
        link_id = 1

        for i in range(1, switch_number + 1):
            for p in range(i, switch_number + 1):
                if i != p:
                    if link_id not in links:
                        links[link_id] = link_format(i, p, link_cap)
                        link_id += 1
        return links
```

Codice 6.1. *Funzione per la creazione automatica di un grafo di rete completo*

Consideriamo il numero di switch come n; notiamo che il ciclo interno parte dall'indice del ciclo `for` esterno avendo così una complessità con equazione 6.1.

$$O\left(\sum_{i=1}^{n}(n-i)\right) \tag{6.1}$$

Che andando a sviluppare risulterà essere:

$$O\left(\sum_{i=1}^{n}(n-i)\right) = O\left(\sum_{i=1}^{n}n - \sum_{i=1}^{n}i\right) \tag{6.2}$$

$$= O\left(n\sum_{i=1}^{n}1 - \sum_{i=1}^{n}i\right) \tag{6.3}$$

$$= O\left(n \cdot n - \frac{n(n+1)}{2}\right) \tag{6.4}$$

$$= O\left(n^2 - \frac{n^2+n}{2}\right) \tag{6.5}$$

$$= O\left(\frac{2n^2 - n^2 - n}{2}\right) \tag{6.6}$$

$$= O\left(\frac{n^2-n}{2}\right) \tag{6.7}$$

$$= O(n^2) \tag{6.8}$$

Poiché il termine lineare è dominato dal termine quadratico per grandi n. Ciò è in linea con la grandezza intrinseca del numero di links per un grafo completo, come visto nell'equazione 3.1, per cui siamo obbligati a una complessità quadratica. Possiamo inoltre osservare dal passaggio 6.7:

$$\sum_{i=1}^{n}(n-i) = \frac{n^2-n}{2} = \frac{n(n-1)}{2}$$

Il che è esattamente l'equazione $\frac{n(n-1)}{2}$ che determina il numero di links in un grafo completo. Con questo vogliamo dire che l'algoritmo esegue esattamente il numero di iterazioni necessarie a creare i link, ottimizzando al massimo le risorse computazionali. Va notato che il costo della riga di codice `if link_id not in links` è costante per il caso medio, dato che `links` è un dizionario, ma può essere lineare nel caso pessimo [26], quindi nel caso avremmo complessità pari a $O(n^3)$.

Vi sono poi le varie funzioni dedicate alla creazione delle strutture mesh, torus e graph (la modalità libera), le quali sono tutte basate sul concetto di disporre i nodi su una matrice (griglia). Analizziamo la funzione dedicata alla creazione automatica di un grafo mesh.

```
    def create_auto_mesh_links(link_cap, switch_number):
        """ Auto detect links """
        data_links = {}
        link_id = 1

        side = math.sqrt(switch_number)
        # 16/9 aspect ratio
        rows = math.ceil(side * (3/4))
        cols = int(side * (4/3))

        if (rows * cols) < switch_number:
            cols += 1

        switches = [[0 for _ in range(cols)] for _ in range(rows)]
        switch_cont = 1
```

```
        for r in range(0, rows):
            for c in range(0, cols):
                if(switch_cont > switch_number):
                    break
                switches[r][c] = switch_cont
                switch_cont += 1

                if c > 0:
                    data_links[link_id] = link_format(switches[r][c-1],
                                                      switches[r][c],
                                                      link_cap)
                    link_id += 1
                if r > 0:
                    data_links[link_id] = link_format(switches[r-1][c],
                                                      switches[r][c],
                                                      link_cap)
                    link_id += 1
            if(switch_cont > switch_number):
                break
        return data_links, switches
```

Codice 6.2. *Funzione per la creazione automatica di un grafo mesh*

Come possiamo vedere nel codice 6.2, la condizione che domina la complessità è rappresentata dai due cicli `for` annidati mentre il resto delle operazioni viene effettuato in tempo costante. I cicli `for` vengono effettuati sui valori rows e cols, i quali sono derivati dalla radice quadrata del numero di switch. Ipotizziamo se il numero di switch fosse n, con n molto grande. Consideriamo i due cicli `for` annidati, ognuno dei quali itera fino alla radice quadrata di n:

```
for i in range(sqrt(n)):
    for j in range(sqrt(n)):
        # operazioni
```

Poiché ogni ciclo itera $\sqrt{n}$ volte, il numero totale di iterazioni è $\sqrt{n} \times \sqrt{n} = n$. Pertanto, la complessità computazionale è $O(n)$, quindi lineare. Analizziamo la funzione dedicata alla creazione automatica di un grafo torus.

```
    def create_auto_toro_links(link_cap, switch_number):
        links, switches = create_auto_mesh_links(link_cap, switch_number)
        rows = len(switches)
        cols = len(switches[0])
        links_id = len(links) + 1
        linkList = []
        for _, content in links.items():
            linkList.append(content["endpoints"])
        for i in range(cols):
            if switches[0][i] != 0 and switches[rows-1][i] != 0 and [switches
                [0][i],switches[rows-1][i]] not in linkList:
                links[links_id] = link_format(switches[0][i], switches[rows
                    -1][i], link_cap)
                links_id += 1
        for i in range(rows):
            if switches[i][0] != 0 and switches[i][cols-1] != 0 and [switches
                [i][0], switches[i][cols-1]] not in linkList:
                links[links_id] = link_format(switches[i][0], switches[i][
                    cols-1], link_cap)
                links_id += 1
        return links, switches
```

Codice 6.3. *Funzione per la creazione automatica di un grafo torus*

La funzione 6.3, si serve del codice di creazione mesh graph automatica 6.2 per creare i link interni alla struttura, successivamente itera prima sulle colonne per individuare i link che collegano i nodi dell' estremità superiore della matrice con quelli dell'estremità inferiore e, successivamente, itera sulle righe per fare la stessa cosa con i nodi posti alle estremità laterali. Calcolando la complessità abbiamo, considerando sempre n il numero di switch, il costo computazionale della funzione di creazione automatica dei link di un mesh graph $O(n)$, il quale viene sommato al costo del primo ciclo $O(\sqrt{n})$ e al costo del secondo ciclo $O(\sqrt{n})$ mentre tutte le altre operazioni hanno tempo costante.

$$O(n) + O(\sqrt{n}) + O(\sqrt{n}) = O(n + \sqrt{n} + \sqrt{n}) = O(n + 2\sqrt{n}) = O(n)$$

La complessità risulta quindi lineare. Passando alle funzioni user per la creazione dei link troviamo che le quelle dedicate per il mesh e torus graph sono logicamente molto simili a quelle auto, con la differenza che in quelle user ci sono dei controlli legati all'inserimento dei valori numerici dei nodi e degli zeri per gli spazi vuoti. Sono quindi tutte operazioni che hanno la stessa complessità delle controparti auto analizzate precedentemente. Diverso il discorso per la funzione per tipo di grafo personalizzato, qui la logica è molto più semplice poiché essendo già tutto descritto nel custom file, si tratta soltanto di ricopiare la struttura dei links così come rappresentata nel file, quindi in un caso peggiore in cui ci sia un numero elevato n di links la complessità sarà lineare.

Successivamente alla creazione dei link, la struttura critica è la creazione dei pacchetti in quanto il loro numero può essere davvero elevato.

```
for fractional_unit in range(0, SIM_TIME * creationRate):
    # Changing trafficPercentage every second
    if fractional_unit % creationRate == 0:
        for link, content in links.items():
            content["trafficPerc"] = config_gen_utils.change_traffic_perc(
                content["trafficPerc"], setup["trafficVariation"])
    ENDP_A = 0
    ENDP_B = 1
    for link, content in links.items():
        trafficPerc = content["trafficPerc"]
        # Packets Per Second
        PPS = ((content["capacity"] * 1e6) / 8) / PACKET_SIZE
        timeWalker_toStore = None
        if setup["packetsFile"] == "json":
            timeWalker_toStore = str(timeWalker)
        else:
            timeWalker_toStore = timeWalker
        for i in range(0, int((PPS*(trafficPerc/100))/creationRate) ):
            packet = config_gen_utils.create_packet(content["endpoints"][
                ENDP_A],
                                        content["endpoints"][ENDP_B],
                                        timeWalker_toStore,
                                        PACKET_SIZE)
            packets.append(packet)
        remaining_packets, _ = math.modf((PPS*(trafficPerc/100))/creationRate
            )
        if remaining_packets != 0:
            remaining_packet_size = int(round(remaining_packets, 3) *
                PACKET_SIZE)
            packet = config_gen_utils.create_packet(content["endpoints"][
                ENDP_A],
```

```
                                        content["endpoints"][ENDP_B],
                                        timeWalker_toStore,
                                        remaining_packet_size)
            packets.append(packet)
    timeWalker += timedelta(milliseconds=PC_DELTA)
```

Codice 6.4. *Logica creazione pacchetti*

Nel codice 6.4, il ciclo esterno che dipende da `SIM_TIME * creationRate`, sim time è il tempo totale della generazione di pacchetti, il quale dipende da quanto si vuol fare durare la generazione. La variabile `creationRate` rappresenta in quante parti è suddiviso un secondo (cioè la risoluzione temporale), questo è dato dalla scelta del parametro `creationDelta` del file di setup e che indica la frequenza temporale di creazione dei pacchetti, quindi il ciclo dipende dalla quantità di frazioni temporali in cui si generano i pacchetti.

Consideriamo m la grandezza che determina la complessità del primo ciclo `for` esterno, definita da `SIM_TIME * creationRate`, definiamo con l il numero di links e con p il numero dei pacchetti al secondo da generare e, analizzando il codice 6.4, possiamo dedurre lo pseudocodice in Fig. 6.5.

```
for i in range(0, m):
    if fractional_unit % creationRate == 0:
        for i in range(0, l):
            # operazioni in tempo costante
    for i in range(0, l):
        # operazioni in tempo costante
        for i in range(0, p):
            # operazioni in tempo costante
    # operazioni in tempo costante
```

Codice 6.5. *Logica creazione pacchetti, studio della complessità*

Ne ricaviamo che la complessità risulta essere:

$$O(m(l + lp)) = O(m(l(1 + p))) = O(m(lp))$$

Possiamo dedurre che la complessità dipende fortemente da quanto grandi siano i valori m, n e p. Tuttavia i paramentri m e n non sono grandi per le finalità che si pone il progetto e potremmo ipotizzare un caso medio di utilizzo per un grafo torus, per poi ricavarne la complessità. Supponiamo di avere un `SIM_TIME` di 3600 secondi (un'ora), `creationDelta` posto a 100 millisecondi e un creationRate di 10 dato da:

$$\text{creationRate} = \frac{1 \text{ sec}}{\text{creationDelta}} = 10 \tag{6.9}$$

Ne ricaviamo un valore pari a 36000, il quale può essere considerato una costante. Il secondo ciclo che troviamo è all'interno del primo costrutto `if`, il quale scandisce tutti i link per cambiarne le percentuali di traffico da creare; il caso peggiore qui è il caso di grafo completo avente un numero di link riportato nell'equazione 3.1, quindi con complessita quadratica, tuttavia è bene precisare che il progetto vuole rappresentare maggiormente la rete in un data center, la quale nella realtà non è mai costruita come un grafo completo poiché i costi sarebbero proibitivi e la complessità aumenta notevolmente, piuttosto è comune avere delle rappresentazioni come il grafo mesh o torus, per questo motivo d'ora in poi consideremo anche numerazioni di links e nodi legati a questi due tipi di grafi. Analizzando i grafi mesh e torus troviamo che

la quantità di link che possiamo considerare è costante, infatti se consideriamo n il numero di switch, troviamo che il numero l di link di un grafo torus è dall'equazione:

$$l = \frac{4n}{2} = 2n \tag{6.10}$$

Poichè ogni nodo ha 4 nodi adiacenti che vengono contati due volte. Essendo il grafo torus più ricco di links rispetto al grafo mesh, definisce il caso peggiore tra i due risultando così $O(\frac{4c}{2}) = O(c)$, essendo il numero di switch massimo pari a 1000 quindi costante. Una precisazione sulla funzione `change_traffic_perc()`, la quale opera in tempo costante poiché ha solo il compito di cambiare il valore di percentuale di traffico. Il secondo ciclo che troviamo ha all'interno un altro ciclo; il primo itera sempre sui links, il secondo itera per il numero di pacchetti da creare, dato un certa percentuale di traffico, in una data frazione temporale. Riprendendo la definizione p come il numero di pacchetti da generare, possiamo affermare che il parametro dominante per la complessità sia:

$$O(\frac{\text{PPS}(\frac{\text{trafficPerc}}{100})}{\text{creationRate}}) = O(\frac{cp}{c}) = O(p) \tag{6.11}$$

La funzione `create_packet()` opera in tempo costante poiché semplicemente formatta in modo corretto il pacchetto da salvare, così come operano in tempo costante tutte le restanti operazioni. Facendo quindi una somma di tutte le complessità riguardanti i punti critici 6.7 troviamo che la complessità di questo pezzo logico di codice è $O(p)$ **??**, fatta eccezione per l'opzione di grafo completo, poiché avremmo un numero di links molto grande pari a l così come il numero di pacchetti e la parte di codice 6.6 avrebbe complessità $O(lp)$.

```
O(c)        for link, content in links.items():
O(1)            # operazioni
O(p)            for i in range(0, int((PPS*(trafficPerc/100))/creationRate)):
O(1)                # operazioni
```

Codice 6.6. *Complessità per numero di links di un grafo completo*

```
O(c)    for fractional_unit in range(0, SIM_TIME * creationRate):
O(1)        if fractional_unit % creationRate == 0:
O(c)            for link, content in links.items():
O(1)                content["trafficPerc"] = config_gen_utils.
    change_traffic_perc(content["trafficPerc"], setup["trafficVariation"])
O(1)        # operazioni
O(c)        for link, content in links.items():
O(1)            # operazioni
O(p)            for i in range(0, int((PPS*(trafficPerc/100))/creationRate)):
O(1)                # operazioni
O(1)         # operazioni
O(1)    # operazioni
```

Codice 6.7. *Complessità della logica di creazione pacchetti*

Assumendo che i grafi completi siano estremamente rari nella realtà e che nei data center troviamo per lo più configurazioni mesh e torus, possiamo affermare che la complessità globale per il generatore, nell'ambito del progetto, può dirsi essere lineare, come lo mostrano anche i test effettuati in figura 6.1. I test hanno come

parametri un tempo di generazione pari a due secondi con links aventi capacità di 1 Gbps in un grafo di tipo torus. Si può vedere che all'aumentare del numero di nodi (switch) il tempo necessario a generare i pacchetti cresce in modo lineare.

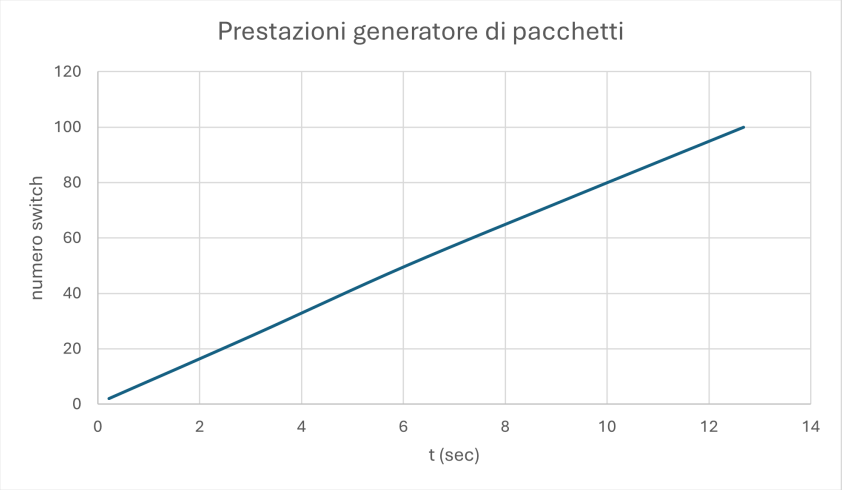


Figura 6.1. *Prestazioni per il generatore di traffico*

6.1.2 Analizzatore di traffico

Il cuore dell'analizzatore è la parte logica di codice incaricata di scandire tutti i pacchetti, estrarne le somme dei valori di payload e relative medie percentuali. L'analizzatore in primis carica i due files `network.yaml` e `packets.json/yaml`, in questo caso creiamo sempre una struttura ausiliaria necessaria a memorizzare man mano le somme pei payload; ciò consiste semplicemente in un ciclo for che scandisce la lista di link presente nel network file, questo chiaramente dipende dal numero di link memorizzato quindi ne deduciamo che il caricamento abbia complessità $O(l)$, con l rappresentante il numero di link.

Per quanto riguarda invece il caricamento del file dei pacchetti, la complessità dipende dal numero di pacchetti p memorizzato, abbiamo quindi una complessità di $O(p)$. Analizziamo ora il codice 6.8 in cui avviene l'analisi vera e propria.

```
    # Every loop is an analyzed fractional unit
    while timeWalker <= startTime + (simTime - updateDelta):
        # For each updateDelta reset values
        for link, content in links.items():
            # Reset traffic
            content["trafficUDT"] = 0
        # Identify the link belonging to the analyzed packet
        if packet is not None:
            link = links[frozenset({packet["A"], packet["B"]})]
            # Analyzing every packet in the analyzed range
            print("analyzing time: ", packet["t"])
            packetTimestamp = None
            if networkData[SIM_PARAMETERS]["packetsFile"] == "json":
                packetTimestamp = utils.str_to_datetime(packet["t"])
            else:
                packetTimestamp = packet["t"]
            while packetTimestamp >= timeWalker and packetTimestamp <
                timeWalker + updateDelta:
```

```
                link["trafficUDT"] += packet["d"]
                link["trafficDT"] += packet["d"]
                packet = next(packetsDataIterator, None)
                if packet is not None:
                    if networkData[SIM_PARAMETERS]["packetsFile"] == "json":
                        packetTimestamp = utils.str_to_datetime(packet["t"])
                    else:
                        packetTimestamp = packet["t"]
                    link = links[frozenset({packet["A"], packet["B"]})]
                else:
                    break
        # Pushing forward the analyzing time
        timeWalker += updateDelta
        # Storing the fractional time units values
        for link, content in links.items():
            content["updateDeltaTraffic"].append(
                {"updateTime": timeWalker, "traffic": content["trafficUDT"]}
            )

        # Storing the averageDelta units value
        # If the averageDelta average is not the first one
        if timeWalker > startTime + averageDelta:
            for link, content in links.items():
                # Calcolate the element index to subtract from `
                    updateDeltaTraffic`
                index = (len(content["updateDeltaTraffic"]) - 1) -
                    lastFirstUDindex
                # Traffic value to subtract from the averageTraffic
                traffic_to_subtract = content["updateDeltaTraffic"][index]["
                    traffic"]
                content["trafficDT"] -= traffic_to_subtract
                content["traffic"].append({"updateTime": timeWalker, "traffic
                    ": content["trafficDT"]})
        # Else if the averageDelta average is the first one:
        # in this case we don't need to subtract an updateDeltaTraffic
        elif timeWalker == startTime + averageDelta:
            for link, content in links.items():
                content["traffic"].append({"updateTime": timeWalker, "traffic
                    ": content["trafficDT"]})
```

Codice 6.8. *Logica analisi pacchetti*

Il primo ciclo while scandisce semplicemente le unità frazionarie temporali m scandite ogni intervallo di tempo `updateTime`, Segue un ciclo for che itera sui link, anche qui abbiamo una complessità di $O(l)$. Dopo diverse operazioni con tempo O(c), la parte che merita attenzione è il ciclo while nidificato. Questo ciclo itera su tutti i pacchetti che hanno un timestamp che ricade nell'unità frazionaria di tempo da analizzare, sono quindi tutti i pacchetti p in un dato intervallo di tempo `updateDelta` per un dato traffico percentuale; la complessità dipende quindi dai pacchetti ottenendo $O(p)$. Dopo diverse operazioni con complessità O(c), i cicli che troviamo sono tre iterazioni sui link, ciascuno con complessità $O(l)$. Possiamo quindi definire la complessità compelssiva sommando le varie parti 6.12.

$$O(m(l + p + l + l + l)) = O(m(4l + p)) = O(m(l + p)) \tag{6.12}$$

Anche in questo caso la complessità dipende fortemente dal numero di untià frazionarie m, dal numero di links l e dalla quantità di pacchetti p da leggere. Considerando lo stesso caso d'uso medio presente nella sezione 6.1.1, troviamo che m e l possono considerarsi costanti, quindi la complessità dipende linearmente da

p. Un'evenutale conferma arriva dai test dove possiamo vedere l'andamento lineare nel grafico nel caso medio ipotizzato precedentemente 6.2. Possiamo notare che all'aumentare del numero di switch nella rete, il tempo cresce in modo lineare.

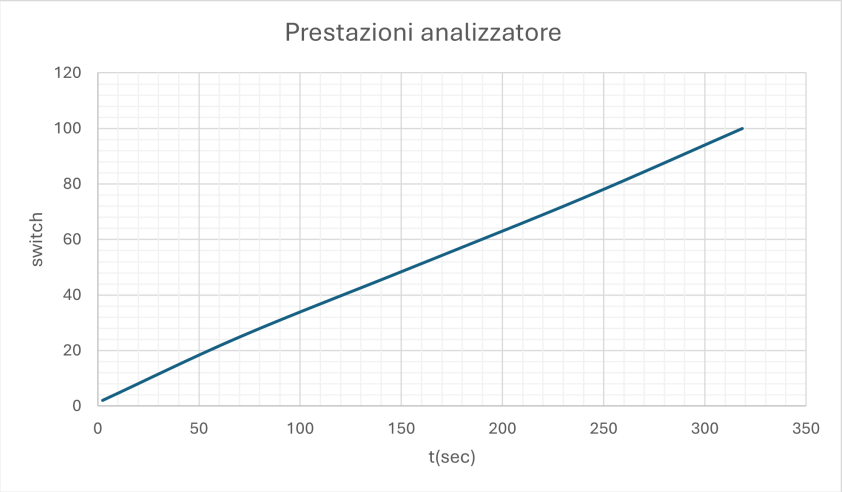


Figura 6.2. *Prestazioni dell'analizzatore*

6.1.3 Visualizzatore grafico

Analizzando il visualizzatore troviamo che il cuore da analizzare ricade sulla parte di codice che crea le animazioni. Nel creare le strutture necessarie, caricandole dai files, troviamo le stesse complessità computazionali ricavate per i precedenti moduli sofware, mentre per la creazione della base di partenza grafica abbiamo la funzione `Graph()` per disegnare i grafi completi mentre abbiamo il codice visto in Fig. 5.4 per i grafi mesh, torus e personalizzato. Per quanto riguarda `Graph()` rimane valida l'analisi di complessità quadratica per via nel calcolo del numero di link in un grafo completo, mentre per la creazione delle altre tipologie di grafo osserviamo dalla Fig. 5.4 che la complessità è dominata dal numero n di switch, essendo riproposti due cicli annidati che iterano sulla radice quadrata del numero di nodi. Analizziamo quindi la logica che risulta maggiormente critica, la creazione delle animazioni che, per motivi di lunghezza del codice sarà descritto come pseudocodice:

```
while time_walker <= end_time:
    animations = []
    # definig all the traffic color links at timeWalker time
    for content in traffic_data[CONST.ANALYZED_DATA["TRAFFICS"]]:
        color_perc = int(content["traffic"][traffic_count])
        animations.append(animation for the link in content)
    # playing animations
    if(time_walker == phase_time):
        self.play(every traffic animation and the changing phases text
            animation)
    else:
        self.play(every traffic animation)
    traffic_count += 1
    # pushing forward sim time to check
    time_walker += timedelta(milliseconds=show_delta)
```

Codice 6.9. *Logica creazione animazioni*

Il primo ciclo while itera su ciscuna unità frazionaria (ricordiamo definiamo con m la totalità delle unità frazionarie), vengono poi scanditi l links nel `for` interno andando a prendere il valore del traffico percentuale per l'unità frazionaria analizzata, abbiamo quindi $O(l)$. Si disegnano le animazioni e il resto sono operazioni con complessità $O(c)$. La libreria non dichiara la complessità che può interessare il metodo `play()`, tuttavia i parametri di ingresso rappresentano principalmente una lista di animazioni e diversi mobject di tipo `Text()`. La complessità in questo caso possiamo ricavarla dai test, come mostrato in figura 6.3.

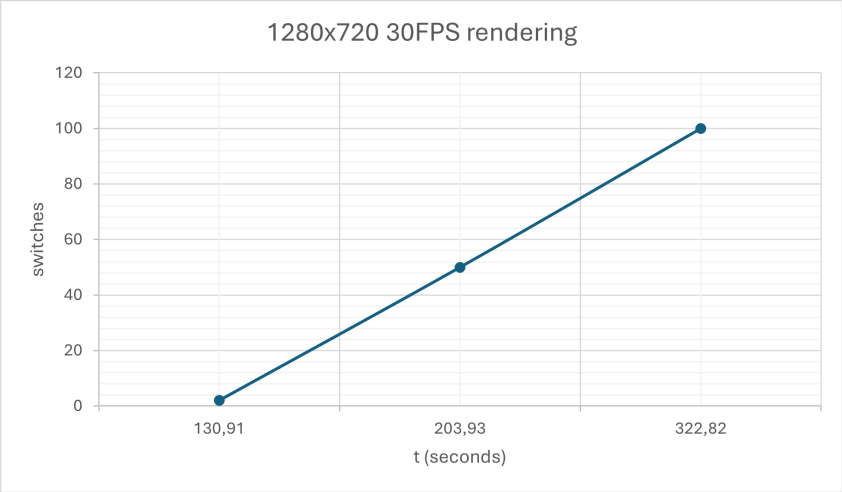


Figura 6.3. *Prestazioni per video con risoluzione di 720p a 30 fps*

Il grafico in Fig. 6.3 si basa su una generazione di 5 secondi con tipologia di grafo mesh; possiamo notare che al crescere del numero degli switch il tempo per il rendering del video cresce in modo lineare.

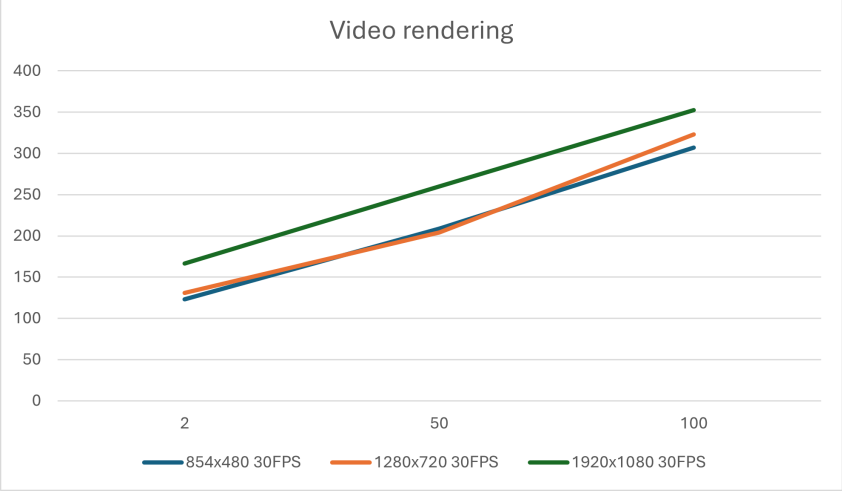


Figura 6.4. *Comparazione prestazioni per risoluzioni di 480, 720p e 1080 a 30 fps*

Volendo osservare i test da un'altra prospettiva, possiamo intendere il tempo di rendering come la variabile dipendente rispetto al numero di switch; in questo esempio

mostriamo una comparazione con le risoluzioni 480p, 720p e 1080p renderizzati a 30 fps(frames per second), sempre per 5 secondi di generazione con grafo mesh, come mostrato in Fig. 6.4. Possiamo osservare dal grafico che l'andamento è lineare per tutti i tipi di rendering analizzati.

6.2 Prestazioni dei formati json e yaml

Dal punto di vista delle prestazioni implicate dai formati scelti `json` e `yaml`, si è dimostrato che `json` è incredibilmente più performante rispetto a `yaml` per quanto riguarda files di grandi dimensioni, possiamo vederne un risultato nel test raffigurato in Fig. 6.5.

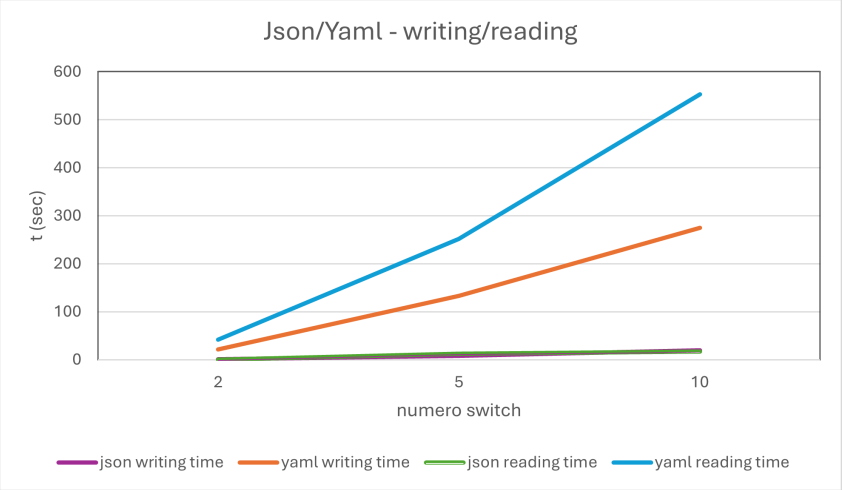


Figura 6.5. *Comparazione prestazioni* `json` *e* `yaml`*, scrittura e lettura*

Il test è basato sulla configurazione avente tempo di generazione 5 secondi e tipologia di grafo mesh con links aventi capacità 1 Gbps. In questo caso, così come anche i futuri test che verrano riportati, si considera il tempo come la variabile dipendente dal numero di switch impiegati. Possiamo osservare dal test che `json` è decisamente più veloce di `yaml` sia in scrittura che in lettura, tanto da avere tempi praticamente sovrapposti, mentre per `yaml` si ha un coefficiente angolare che fa pesare molto di più l'andamento della curva. Quindi possiamo concludere che `json` è decisamente la scelta più opportuna per i fini del progetto. Risulta comunque interessante notare l'andamento di `json` nella lettura e scrittura per un numero di switch superiore a 7; avendo la stessa configurazione del test precedente otteniamo i grafici in Fig. 6.6 e 6.7.

Osservando i grafici notiamo come il carattere degli andamenti di scrittura e lettura di `json` prenda forma a partire da una configurazione con 7 switch, delineando un andamento lineare in entrambi i casi, con un'ottima performance per la lettura. Per quanto riguarda la dimensione dei files prodotti, abbiamo visto `yaml` produrre files decisamente più piccoli di `json`, possiamo così farci un'idea dai dati registrati in fase di test in Fig. 6.8.

Possiamo concludere che la dimensione dei file `json` è in media 2.7 volte maggiore rispetto ai corrispondenti file `yaml`, indicando un rapporto di dimensione di circa 2.7

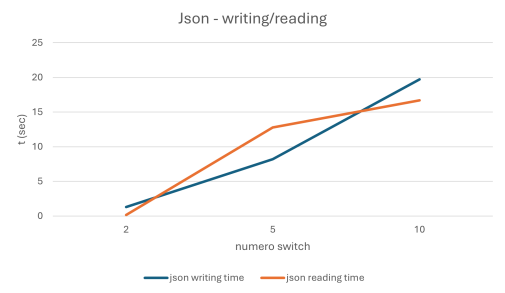


Figura 6.6. *Test* `json`*, lettura e scrittura da 2 a 10 switch*

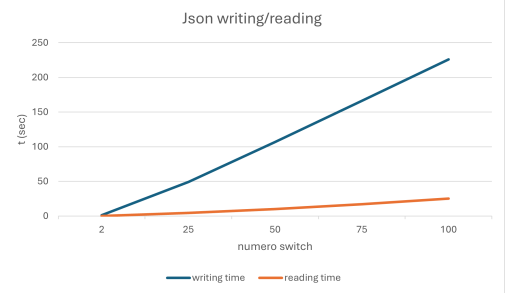


Figura 6.7. *Test* `json`*, lettura e scrittura da 2 a 10 switch*

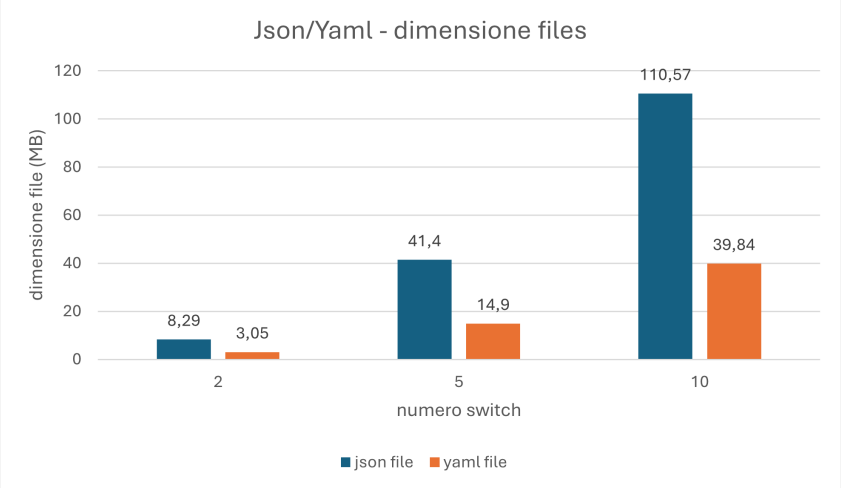


Figura 6.8. *Comparazione dimensioni files* `json` *e* `yaml`

tra i due formati. Ciò è chiaramente un vantaggio nei confronti di `yaml`, tuttavia la velocità che riesce a garantire `json` è talmente superiore da far propendere per un compromesso in termini di dimensione dei files, rendendo quindi `json` una scelta preferenziale.

Capitolo 7

Conclusioni

Lo scopo di questo progetto era di creare un'applicazione che mostrasse l'intensità di traffico di una rete in modo grafico e animato. Come si è avuto modo di vedere, l'intento è pienamente riuscito Abbiamo realizzato.. Per quanto riguarda generatore, analizzatore e visualizzatore grafico, possiamo concludere che tutti i moduli del progetto operano in tempo lineare, fatta eccezione della tipologia di grafo completo che intrinsicamente porta dietro di se una natura di complessità quadratica, rappresentando così un'ottimo risultato in termini prestazionali. Un'altra conclusione, non banale, riguarda l'essere riusciti a realizzare un prodotto di analisi di rete da una libreria che nasce come scopi relativi a video esplicativi matematici, come possono essere grafi cartesiani animati, spazi vettoriali animati e così via; si è riusciti quindi a sfruttare la libreria per fini non strettamente inerenti ai suoi scopi di esistenza, questo processo di adattabilità ha portato a un percorso formativo decisamente creativo, sperimentando nuove soluzioni con in mezzi forniti.

7.1 Sviluppi futuri

La base del progetto e l'adattabilità di Manim fanno sì che ci siano ampi margini di manovra per applicazioni future, sia legati al concetto di cosa si propone di fare questo progetto sia in termini di integrazioni future di caratteristiche. Il concetto che sta alla base del progetto può essere allargato al di fuori di un data center e trovare altri terreni in cui potersi adattare, sfruttandone le caratteristiche di base. Un'altra caratteristica di Manim che può essere sfruttata in futuro è la possibilità di renderizzare video in 3D, infatti la libreria supporta anche mobject e spazio vettoriale 3D.

Possiamo proporre qualche esempio di impiego che sfrutti il progetto come base: si potrebbe adattare l'applicazione ad esempio per il traffico di reti stradali in cui i nodi sono rappresentati da punti d'interesse, raggiungibili tramite strade (i link) e con i pacchetti che rappresentano i veicoli, analizzandone il traffico, oppure per reti di gioco multiplayer massivo (MMO), in cui la topologia mesh può essere utilizzata per gestire la comunicazione tra server distribuiti geograficamente, reti di sensori in cui la topologia mesh permette ai dispositivi di comunicare anche in condizioni ambientali difficili o in scenari dove i nodi sono distribuiti in modo irregolare, gli

ambiti sono potenzialmente tanti. e potrebbe essere interessante l'impiego per diversi temi:

- **reti stradali:** un'associazione quasi istantanea che può sorgere è l'adattamento dell'applicazione per la visualizzazione dell'intensità del traffico stradale. In questo caso i nodi potrebbero essere punti d'interesse come le città, raggiungibili tramite strade (i link) e con i pacchetti che rappresentano i veicoli.
- **reti di gioco multiplayer massivo (MMO):** in questo caso la topologia mesh può essere utilizzata per gestire la comunicazione tra server distribuiti geograficamente, mostrando l'intensità di popolazione giocante per determinate aree di interesse.
- **reti di sensori:** questo tipo di reti interessano un insieme di dispositivi elettronici, ognuno con uno o più sensori, i quali prelevano o elaborano dati in modo autonomo per poi comunicare tra loro. La topologia mesh permette ai dispositivi di comunicare anche in condizioni ambientali difficili o in scenari dove i nodi sono distribuiti in modo irregolare. In questo scenario si potrebbe analizzare l'intensità di scambio di informazioni tra i vari dispositivi.
- **reti satellitari:** sfruttando la capacità di rendering 3D di Manim, sarebbe interessante un impiego di rete satellitare che faccia uso di topologie torus e mesh, potendo rappresentare la tridimensionalità posizionale relativa a un'orbita terrestre. In questo caso si potrebbe rappresentare l'intensità delle comunicazioni scambiate tra i vari satelliti.
- **reti di droni:** la topologia mesh è particolarmente indicata per rappresentare una rete del genere, infatti permette ai nodi (i droni) di comunicare direttamente l'uno con l'altro senza necessità di un'infrastruttura centrale e si potrebbe sfruttare lo spazio 3D renderizzato da Manim per rappresentare i droni in un ambiente tridimensionale.
- **reti sociali:** parlando di social network, l'applicazione potrebbe essere adattata ed espansa per rappresentare l'intensità di comunicazione tra utenti
- **applicazioni distribuite:** in questo caso si potrebbe pensare a una rete composta dalle parti più piccole che compongono l'applicazione e collegate tra loro. La rappresentazione potrebbe interessare l'intensità di comunicazioni tra i vari processi che interessano le diverse parti.

Ringraziamenti

Questo percorso è stato per il sottoscritto una grande opportunità sotto diversi punti di vista; ha rappresentato grande fonte di arricchimento professionale e personale, mi ha fornito competenze complementari agli insegnamenti ricevuti durante il mio percorso di studi soprattutto in ambito front-end, ho approfondito la mia comprensione i tutti i campi incontrati, acquisito nuove competenze tecniche e consolidate altre con cui sono entrato in contatto precedentemente. Personalmente trovo di non poter concludere meglio di così il mio percorso accademico, avendo avuto l'opportunità di dedicare gli sforzi, in questa prova finale, verso qualcosa su cui nutrivo forte interesse, ovvero la programmazione di elementi grafici e loro animazioni in modo programmatico. Ed è in termini di opportunità ricevuta che desidero esprimere la mia sincera gratitudine al mio Relatore, Daniele De Sensi, per avermi accolto, per essersi prodigato tanto per trovare l'ambito di interesse che mi sarebbe piaciuto affrontare, per la sua guida e per il supporto inestimabile e prezioso, fondamentale per il mio sviluppo professionale durante questo periodo.

Desidero ringraziare la mia compagna di vita, Chiara, il mio faro nella notte, senza la quale questo traguardo probabilmente non sarebbe stato possibile, per il supporto costante, la pazienza e per aver creduto fortemente in me. Ringrazio la mia piccola Giulia, Atena, mia figlia, che ogni giorno mi insegna un pezzo di vita, mi dona gioia e speranza.

Ringrazio la mia famiglia, per l'amore, il supporto costante e la pazienza che hanno avuto nel mio lungo e tortuoso percorso.

Un doveroso ringraziamento va a Lucio Molo e Mauro Semproni, miei colleghi universitari e cari amici, che hanno condiviso con me parte della mia vita accademica e mi hanno aiutato a viverla nel modo più sereno e pieno possibile.

Desidero infine ringraziare gli amici della *League of dogs*, che mi hanno accompagnato durante il percorso, per l'immensa compagnia e per avermi fatto sentire bene anche nei momenti più duri.

Bibliografia

[1] Stephen Cass Author. *The Top Programming Languages 2023*. https://spectrum.ieee.org/the-top-programming-languages-2023. Accessed: 2024-04-12. 2023.

[2] Canonical. *PyYAML Documentation*. Accessed: 2024-04-12. 2024. URL: https://pyyaml.org/wiki/PyYAMLDocumentation.

[3] Manim Community. *Graph Class Documentation*. Accessed: 2024-04-12. 2024. URL: https://docs.manim.community/en/stable/reference/manim.mobject.graph.Graph.html.

[4] Cytoscape Consortium. *Cytoscape: An Open Source Platform for Complex Network Analysis and Visualization*. https://cytoscape.org. Accessed: 2024-04-12. 2024.

[5] David Nichols. *Color Blindness Palette Generator*. https://davidmathlogic.com/colorblind. Accessed: 2024-04-12. 2024.

[6] Gephi. *Gephi: The Open Graph Viz Platform*. https://gephi.org. Accessed: 2024-04-12. 2024.

[7] Graphviz. *Graphviz - Graph Visualization Software*. https://graphviz.org. Accessed: 2024-04-12. 2024.

[8] igraph. *igraph - The Network Analysis Package*. https://igraph.org. Accessed: 2024-04-12. 2024.

[9] Institute of Electrical and Electronics Engineers. *IEEE - Advancing Technology for Humanity*. https://www.ieee.org. Accessed: 2024-04-12. 2024.

[10] Johns Hopkins University. *Homepage of Johns Hopkins University*. https://www.jhu.edu/. Accessed: 2024-04-12. 2024.

[11] Johns Hopkins University Libraries. *Network Data Visualization Guide*. https://guides.library.jhu.edu/datavisualization/network. Accessed: 2024-04-12. 2024.

[12] Pajek. *Pajek - Program for Large Network Analysis*. http://mrvar.fdv.uni-lj.si/pajek/. Accessed: 2024-04-12. 2024.

[13] Jon Postel. *Internet Protocol*. RFC 791. Accessed: 2024-04-12. RFC Editor, 1981. URL: https://www.rfc-editor.org/rfc/rfc791#page-11.

[14] Python Software Foundation. *8.3. Mapping Types — dict*. Accessed: 2024-04-12. 2023. URL: https://docs.python.org/3/library/stdtypes.html#typesmapping (visitato il 12/04/2024).

[15] Python Software Foundation. *datetime — Basic date and time types. timedelta Objects*. Accessed: 2024-04-12. 2023. URL: https://docs.python.org/3/library/datetime.html (visitato il 12/04/2024).

[16] Python Software Foundation. *Glossary — Python 3.10.8 documentation. Term: hashable*. Accessed: 2024-04-12. 2023. URL: https://docs.python.org/3/glossary.html#term-hashable (visitato il 12/04/2024).

[17] Python Software Foundation. *JSON encoder and decoder – Python 3.10.8 documentation*. Accessed: 2024-04-12. 2024. URL: https://docs.python.org/3/library/json.html.

[18] Python Software Foundation. *Python Programming Language – Official Website*. https://www.python.org. Accessed: 2024-04-12. 2024.

[19] Y. Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180. Accessed: 2024-04-12. RFC Editor, 2005. URL: https://www.rfc-editor.org/rfc/rfc4180.html#page-2.

[20] SocNetV. *Social Network Visualizer (SocNetV)*. https://socnetv.org. Accessed: 2024-04-12. 2024.

[21] *The JavaScript Object Notation (JSON) Data Interchange Format*. Internet Requests for Comments. RFC. Accessed: 2024-04-22. IETF, dic. 2017. URL: https://www.rfc-editor.org/rfc/rfc8259.txt.

[22] The Manim Community Developers. *Manim Documentation: Quickstart Guide*. Accessed: 2024-03-25. 2024. URL: https://docs.manim.community/en/stable/index.html.

[23] The Manim Community Developers. *Manim Documentation: Quickstart Guide*. Accessed: 2024-03-25. 2024. URL: https://docs.manim.community/en/stable/tutorials/quickstart.html.

[24] UCINET. *UCINET Software*. https://sites.google.com/site/ucinetsoftware/. Accessed: 2024-04-12. 2024.

[25] Eric W. Weisstein. *Grid Graph*. https://mathworld.wolfram.com/GridGraph.html. Accessed: 2024-04-22. 2024.

[26] Python Wiki. *Time Complexity – Python Wiki*. [Online; accessed 22-April-2024]. 2024. URL: https://wiki.python.org/moin/TimeComplexity.

[27] Wikipedia. *Grafo completo*. https://it.wikipedia.org/wiki/Grafo_completo. Accessed: 2024-04-22. 2024.

[28] Wikipedia. *Torus – Wikipedia, The Free Encyclopedia*. [Online; accessed 22-April-2024]. 2024. URL: https://en.wikipedia.org/wiki/Torus.

[29] Wikipedia contributors. *File:Torus graph.png*. https://en.m.wikipedia.org/wiki/File:Torus_graph.png. Accessed: 2024-04-12. 2024.

[30] YAML Core Team. *YAML Ain't Markup Language (YAML™) Version 1.2*. https://www.rfc-editor.org/info/rfc9512. Accessed: 2024-04-12. 2009.