UNIVERSITÀ DEGLI STUDI DI ROMA TOR VERGATA

MACROAREA DI SCIENZE MATEMATICHE, FISICHE E NATURALI

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

CORSO DI LAUREA IN INFORMATICA

TESI DI LAUREA IN ALGORITMI E STRUTTURE DATI

TITOLO
TESTARE LA CONNETTIVITÀ TEMPORALE DI UN ALBERO : UN ALGORITMO DI COMPLESSITÀ QUASI LINEARE

Relatore:
Prof.
Luciano Gualà

Candidato:
Franco Salvucci
Matricola: *0306609*

Anno Accademico 2023/2024

Indice

1 Introduzione

Negli ultimi anni, l'analisi dei **grafi temporali** (più informazioni le possiamo trovare qui [1],[2],[3],[4],[5]) ha attirato notevole interesse per via della loro capacità di modellare sistemi dinamici in cui i collegamenti tra entità variano nel tempo.
Diversamente dai grafi statici, in cui gli archi rappresentano connessioni permanenti, nei grafi temporali gli **archi attivi** sono associati a specifici istanti o intervalli temporali, introducendo una dimensione dinamica che complica la definizione stessa di connettività.
In questo contesto, un percorso non è più valutato solo in termini di presenza degli archi, ma deve rispettare la sequenza cronologica degli **archi attivi**.
Tale percorso, noto come **cammino temporale**, rappresenta una sequenza di passaggi che avvengono in un ordine coerente con il tempo, garantendo che ogni transizione da un nodo a un altro avvenga nel momento giusto.
Di conseguenza, due nodi si dicono **temporalmente connessi** se esiste almeno un cammino temporale che li collega, rispettando l'ordine temporale degli archi attivi.
Questa tesi si propone di indagare il problema della connettività in grafi temporali, proponendo un algoritmo quasi lineare per determinare se, dato un **grafo temporale con topologia ad albero**, esso risulti essere temporalmente connesso.
La struttura del lavoro è organizzata come segue:

- Capitolo 1: Introduzione – Fornisce una panoramica generale dei grafi temporali e del problema della temporal connectivity, insieme alla motivazione e agli obiettivi del lavoro.

- Capitolo 2: Preliminari e Definizione del problema– Introduce i concetti e le definizioni di base necessarie per lo studio dei grafi temporali.

- Capitolo 3: Algoritmo Efficiente – Presenta l'algoritmo sviluppato per verificare la connettività temporale, con un'analisi dettagliata della complessità computazionale.

- Capitolo 4 : Sviluppi Futuri e Conclusioni - Riassume il lavoro svolto nella tesi e apre le porte a possibili sviluppi del problema.

- Appendice dei codici : Un'appendice contenente i vari codici scritti in Python

Con questo lavoro, si intende offrire un contributo alla comprensione dei grafi temporali e alla progettazione di algoritmi veloci per testare la connettività in topologie di rete acicliche.

2 Preliminari e Definizione del Problema

In questo capitolo, introduciamo le definizioni formali e il problema della connettività temporale negli alberi.

2.1 Definizioni

Definizione 2.1.1 Grafo Temporale
Un **grafo temporale** è un grafo $G = (V, E, \tau)$, dove:

- V è l'insieme dei nodi
- E è l'insieme degli archi
- $\tau : E \to 2^{\mathbb{N}}$ è una funzione che assegna a ciascun arco $e = (u, v) \in E$ un insieme di timestamp $\tau(e) \subseteq \mathbb{N}$

Dato un arco $e = (u, v) \in E$ e un timestamp $t \in \tau(e)$, diciamo che $e = (u, v, t)$ è un **arco temporale** di G.

Definizione 2.1.2 Cammino Temporale
Un **cammino temporale** che parte dal nodo $v_O \in V$ e arriva al nodo $v_k \in V$ è definito da una sequenza di archi temporali $P = \langle e_0, e_1, \ldots, e_k \rangle$ e da una sequenza $t_1, t_2, \ldots, t_k$ tale che :

- $e_i = (v_i, v_{i+1}, t_i)$
- $t_1 \leq t_2 \leq \ldots t_k$

Questo ci dice che un percorso temporale tra due nodi, per essere valido, deve necessariamente rispettare l'ordinamento non decrescente dei timestamp sugli archi.

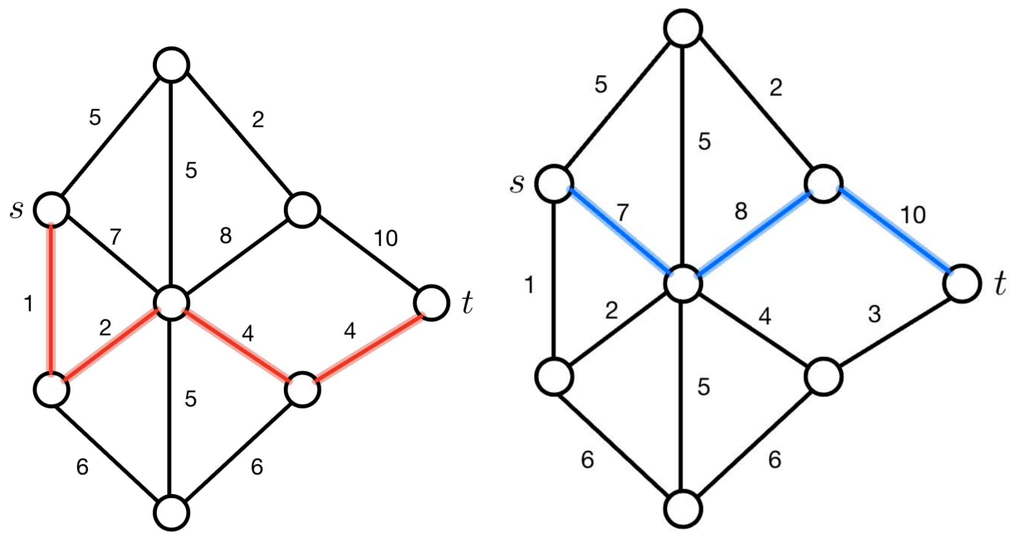
Definizione 2.1.3 Connessione temporale
Un nodo u si dice **temporalmente connesso** a un altro nodo v se esiste un cammino temporale da u a v.

Definizione 2.1.4 Grafo Temporalmente Connesso
Un grafo temporale si dice **temporalmente connesso** se per ogni coppia di nodi $u, v, \in V$ vale che u è temporalmente connesso a v.

Diamo ora le definizioni di *Earliest-Arrival time* e *Latest-Departure time*, che ci serviranno più avanti:

Definizione 2.1.5 Earliest-Arrival time
Data una coppia di nodi $u, v \in V$ definiamo l'**Earliest-Arrival time** tra u e v come il **tempo minimo** in cui è possibile raggiungere v partendo da u, dove, dato un cammino temporale da u a v, il tempo a cui si raggiunge v è il **timestamp dell'ultimo arco temporale** del cammino.

Definizione 2.1.6 Latest-Departure time
Data una coppia di nodi $u, v \in V$ definiamo il **Latest-Departure time** tra u e v come il **tempo massimo** in cui è possibile partire da u per raggiungere v, dove, dato un cammino temporale da u a v, il tempo in cui si parte da u è il **timestamp del primo arco temporale** del cammino.



(a) Earliest-Arrival time fra s e t **(b)** Latest-Departure time fra s e t

2.2 Definizione del Problema

Dopo aver dato la definizione di grafo temporale, cammino temporale e grafo temporalmente connesso, diamo la definizione del problema preso in esame.

Definizione 2.2.1 Problema della Temporal Connectivity su alberi
Dato un grafo temporale G, che è un albero, il **problema della Temporal Connectivity su alberi** consiste nel determinare se tale grafo risulti essere temporalmente connesso oppure no.
In altre parole, dato il grafo G dobbiamo verificare che per ogni coppia di nodi u, v esista un cammino temporale che parte da u e arriva a v.

3 Algoritmo Efficiente

L'algoritmo per verificare se un dato albero T è temporalmente connesso oppure no è diviso in due fasi:

- **Fase (1)** : Preprocessing dell'albero T in cui vengono calcolate informazioni utili alla fase successiva. In particolare in questa fase vengono calcolati due vettori, chiamati $EA_{\max}, LD_{\max}$, indicizzati dai nodi dell'albero.
- **Fase (2)** : Check per determinare se l'albero è temporalmente connesso oppure no. Questa fase effettua il check vero e proprio della connettività temporale dell'albero. Per fare questo usa in modo accorto i due vettori calcolati nela fase precedente.

Vedremo nel dettaglio le due fasi, andando a spiegarne il funzionamento e ad analizzarne i costi computazionali.

3.1 Algoritmo - Fase di preprocessing

3.1.1 Spiegazione

La **fase di preprocessing** è la fase che calcola con approccio bottom-up i valori $EA_{\max}[v], LD_{\max}[v]$ per ogni nodo v.
In seguito, dato un nodo v denoteremo con:

- T_v il sottoalbero di T radicato nel nodo v
- $p(v)$ il padre di v nell'albero (se v è la radice, $p(v)$ è indefinito)

Inoltre, con L_v denoteremo l'insieme di timestamp associati all'arco dell'albero $(v, p(v))$.
In questa sezione, per semplicità di trattazione, assumeremo che i timestamp di ogni L_v sono mantenuti ordinati.
Discuteremo più in dettaglio questa assunzione più avanti nel capitolo, in particolare nella Sezione (3.4).
Questi valori sono definiti nel seguente modo :

- Per ogni nodo $v \in V, v \neq$ radice, definiamo l'$EA_{max}(v)$ in questo modo

$$EA_{\max}(v) = \max_{\substack{f : f \text{ è foglia} \\ f \in T_v}} EA(f, p(v))$$

- Per ogni nodo $v \in V, v \neq$ radice, definiamo il $LD_{max}(v)$ in questo modo

  $LD_{\max}(v)$ = istante di tempo t tale che, se si arriva al padre di v a tempo $\leq t$, allora è possibile visitare tutto T_v.

Questi due valori vengono calcolati seguendo logiche diverse, in base alle due casistiche qui sotto:

- **Casistica (1)** : Nodo v è foglia
- **Casistica (2)** : Nodo v è nodo interno

Casistica (1) :
Se il nodo v è una foglia, i valori verranno calcolati così :

- $EA_{\max}[v] = \min(L_v)$
- $LD_{\max}[v] = \max(L_v)$
- Il calcolo di questi valori risulterà essere costante se i timestamp di L_v sono ordinati

Casistica (2) :
Se il nodo v non è una foglia, i valori verrano calcolati così :

- **Elaborazione dei figli** : Per ogni $u \in \text{child}(v)$, viene eseguita ricorsivamente la procedura **Preprocessing** per calcolare $EA_{max}[u], LD_{max}[u]$
- **Calcolo valori intermedi** : Calcola il valore:
  - EA come il **massimo** tra tutti i valori $EA_{max}[u], \forall u \in \text{child}(v)$
  - LD come il **minimo** tra tutti i valori $LD_{max}[u], \forall u \in \text{child}(v)$
- **Query successore/predecessore** : Dopo aver calcolato i due valori, tramite query di successore/predecessore, implementate con la ricerca binaria:
  - impostiamo il valore $EA_{\max}[v]$ come il successore di EA in L_v
  - impostiamo il valore $LD_{\max}[v]$ come il predecessore di LD in L_v

Finita l'eseguzione di questa fase verranno ritornati due vettori, contenenti tutti i valori $EA_{max}[v], LD_{max}[v], \forall v \in V$, che poi useremo nella fase di check per determinare se l'albero è temporalmente connesso oppure no.

Di seguito è mostrato lo pseudocodice che descrive in dettaglio questa fase.

Algorithm 1 Procedura Preprocessing

```
Require: L_V: Lista di timestamp sull'arco entrante in v, per ogni v nodo.
Require: Vettore EA_max, Vettore LD_max.
procedure PREPROCESSING(Nodo v)
    if v è foglia then
        EA_max[v] ← min(L_V)
        LD_max[v] ← max(L_V)
    end if
    for all u ∈ child(v) do
        Preprocessing(u_i)
    end for
    EA ← max_{u∈child(v)} EA_max[u]
    LD ← min_{u∈child(v)} LD_max[u]
    NextEA ← Successor(L_V, EA)
    NextLD ← Predecessor(L_V, LD)
    if NextEA = −1∨ NextLD = −1 then
        EA_max[v] ← ∞
        LD_max[v] ← ∞
    else
        EA_max[v] ← NextEA
        LD_max[v] ← NextLD
    end if
end procedure
```

3.1.2 Analisi

Andiamo ad analizzare quindi la fase di preprocessing, dal punto di vista computazionale.
Sia $L = \max_v |L_v|$

Il costo per il calcolo dei valori EA_{max}, LD_{max} si divide così :

- **Nodo foglia** : Il calcolo avrà costo $O(1)$, dato che i timestamp su ogni L_v risultano essere ordinati
- **Nodo interno** : Il calcolo avrà costo :
  - $O(\delta_v)$ per le chiamate ricorsive della procedura
  - $O(\log(L))$ per le query di `successore/predecessore`, avendole implementate con l'idea della ricerca binaria

Il costo totale di questa fase per un singolo nodo v è quindi:

$$O(\delta_v) + O(\log(L))$$

Per ogni nodo $v \in T$, il costo complessivo sarà quindi:

$$\sum_{v \in T} [O(\delta_v) + O(\log(L))] = O(n + n\log(L)) = O(n\log(L))$$

3.2 Algoritmo - Check Temporal Connectivity

3.2.1 Spiegazione

Passiamo ora alla seconda fase dell'algoritmo, ovvero la **fase di check della temporal connectivity**.
Questa fase si occupa di verificare due condizioni temporali.
Se le due condizioni saranno sempre verificate, allora l'algoritmo ritornerà *True*, in caso contrario ritornerà *False*.

Le condizioni temporali sono le seguenti :

- **Condizione Temporale (1)** Per ogni nodo $u \in V$, T_u deve essere essere temporalmente connesso.
- **Condizione Temporale (2)** Ogni nodo di T_{u_i} deve essere temporalmente connesso ad ogni altro nodo di $T_{u_j}, \forall u_i \neq u_j$. Quindi ogni nodo di T_{u_i} deve poter *raggiungere* ogni altro nodo di T_{u_j}

Vediamo un esempio di albero temporale tramite la seguente figura, e spieghiamo le due condizioni appena espresse:

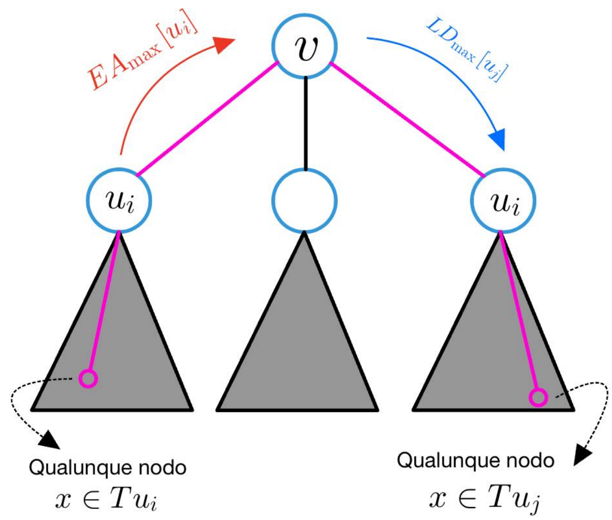


Figura 2: Albero Temporale

- $EA_{\max}[u_i]$: Tempo di arrivo massimo al nodo v partendo da u_i.
- $LD_{\max}[u_j]$: Istante di tempo massimo per visitare tutto T_{u_j}.

Condizione Temporale (1)
La prima condizione la possiamo verificare in modo ricorsivo, con una visita dell'albero.
Condizione Temporale (2)
La seconda condizione la verifichiamo sfruttando i valori EA_{max}, LD_{max} calcolati in precedenza.
Nel dettaglio, per far si che ogni nodo di T_{u_i} possa raggiungere ogni nodo di T_{u_j} bisogna controllare la seguente condizione :

$$EA_{max}[u_i] \leq LD_{max}[u_j]$$

Questo perchè, per come abbiamo definito i valori EA_{max} e LD_{max}, se la condizione viene verificata allora avremo la certezza che ogni nodo di T_{u_i} potrà raggiungere ogni altro nodo di T_{u_j}.
Questo vale per una singola coppia di nodi, ora per espandere il check ad ogni coppia di nodi u_i, u_j dobbiamo modificare la condizione espressa prima nel modo seguente:

$$\forall v \in V, \forall u_i \in child(v)$$
$$EA_{\max}[u_i] \leq \min_{\substack{u_j \in \text{child}(v) \\ u_j \neq u_i}} LD_{\max}[u_j]$$

È facile vedere che se l'$EA_{max}[u_i]$ è minore o uguale al minimo tra tutti i $LD_{max}[u_j]$, con $u_i \neq u_j$, allora ogni nodo di T_{u_i} si potrà connettere con ogni nodo di ogni T_{u_j} con $j \neq i$.
Per controllare queste condizioni l'algoritmo crea un vettore, chiamato D_v, definito in nel seguente modo :

Definizione 3.2.1 Vettore D_v
Il vettore D_v è un vettore di size δ_v = numero figli di v che contiene le coppie $(EA_{\max}[u], LD_{\max}[u])$ per ogni $u \in \text{child}(v)$.

Con questo vettore poi faremo tutti i check necessari per determinare se ogni sottoalbero si può connettere temporalmente con gli altri.
Per fare ciò, il **check tra i sottoalberi** opererà in questo modo :

1. Trova i due $LD_{\max}$ minimi nel vettore D_v. Una volta trovati, li mette in due tuple definite così :
$$(i_1, LD_1), (i_2, LD_2)$$

2. Per ogni valore $EA_{\max} \in D_v$, controlla le seguenti condizioni
   (a) Se l'indice del EA che stiamo controllando è uguale a i_1, allora eseguiamo il check tra l'EA attuale e il secondo LD minimo, ovvero LD_2
   (b) Altrimenti, eseguiamo il check tra l'EA attuale e il primo LD minimo, ovvero LD_1

Queste condizioni sono espresse all'interno del codice nelle righe da 17 a 27.

In modo ricorsivo, l'algoritmo risale l'albero verso la radice propagando il valore del check verso l'alto.
Se, alla fine, il controllo sarà `True`, allora l'algoritmo determinerà che l'albero è temporalmente connesso; altrimenti, stabilirà che l'albero non è temporalmente connesso.

Il codice della fase di check con calcolo del vettore D_v è il seguente :

Algorithm 2 Procedura Check Temporal Connectivity

Require: Vettori $EA_{\max}, LD_{\max}$

procedure CheckTemporalConnectivity($v, EA_{\max}, LD_{\max}$)
    $D_v \leftarrow$ vettore vuoto di dimensione δ_v
    if v è foglia **then**
        return True
    end if
    for all $u \in \text{child}(v)$ **do**
        if Not CheckTemporalConnectivity($u, EA_{\max}, LD_{\max}$) **then**
            return False
        end if
        if $EA_{\max}[u] = \infty \vee LD_{\max}[u] = \infty$ **then**
            return False
        else
            Appendo al vettore D_v la coppia $\langle EA_{\max}[u], LD_{\max}[u] \rangle$
        end if
    end for
    Trovo i due minimi del vettore D_v, e li metto nelle tuple $(i_1, LD_1), (i_2, LD_2)$
    for all $u \in \text{child}(v)$ **do**
        if $index(EA_{max}[u]) = i_1$ **then**
            if $EA_{max}[u] > LD_2$ **then** ▷ Controllo $EA_{max}[u]$ con LD_2
                return False
            end if
        else
            if $EA_{max}[u] > LD_1$ **then** ▷ Controllo $EA_{max}[u]$ con LD_1
                return False
            end if
        end if
    end for
    return True
end procedure

Dopo aver definito le due procedure separate, definiamo ora lo pseudocodice dell'algoritmo completo, ovvero l'algoritmo che richiama le due procedure e ritorna *True* o *False* in modo corretto:

Algorithm 3 Algoritmo Completo

```
procedure ALGORITMO(Albero T)
    EA_max ← vettore vuoto di dimensione n
    LD_max ← vettore vuoto di dimensione n
    EA_max, LD_max ← Preprocessing(Radice di T)
    if CheckTemporalConnectivity(Radice di T, EA_max, LD_max) then
        return True
    else
        return False
    end if
end procedure
```

3.2.2 Analisi

Come abbiamo detto, la fase di check opera in 2 fasi.
Vediamo il costo di queste fasi:

1. **Fase (1)**: Dato che il vettore D_V ha size δ_V, la ricerca dei 2 minimi di tale vettore costerà $O(\delta_V)$

2. **Fase (2)**: Per ogni EA, il check tra lui e il LD minimo ha costo lineare nel numero di figli di v, ovvero $O(\delta_V)$

Quindi, in totale, il costo per un nodo interno v sarà $O(\delta_V)$.
Di conseguenza, per ogni nodo interno $v \in T$, il costo complessivo dell'algoritmo di check sarà

$$\sum_{v \in T} O(\delta_V) = O(n)$$

Unendo i costi per le due procedure , otteniamo il costo dell'algoritmo completo, che sarà :

$$O(n \log(L)) + O(n) = O(n \log(L))$$

3.3 Unificazione delle procedure

Possiamo notare che le due procedure possono essere unite in un'unica procedura, ovvero una procedura che, mentre calcola i valori EA e LD, esegue anche il check temporale tra i sottoalberi.
Questa unificazione non risulta in un algoritmo asintoticamente migliore, ma permette di risolvere il problema con un'unica visita di T, invece che due visite come descritto in precedenza.

Il costo computazionale dell'algoritmo chiaramente non cambia e risulta essere quindi ancora $O(n \log(L))$.

Lo pseudocodice della procedura unificata sarà il seguente:

Algorithm 4 Algoritmo Unificato

Require: L_v : Lista di timestamp sull'arco entrante in v, per ogni v nodo
Require: Vettore $EA_{\max}$, Vettore $LD_{\max}$

procedure VISITA-DFS(Nodo v, $EA_{\max}$, $LD_{\max}$)
    $D_v \leftarrow$ vettore vuoto di dimensione δ_v
    if v è foglia **then**
        $EA_{\max}[v] \leftarrow \min(L_v)$
        $LD_{\max}[v] \leftarrow \max(L_v)$
        Check $\leftarrow$ True
        return Check
    end if
    for all $u \in \text{child}(v)$ **do**
        if Not Visita-DFS(u,$EA_{\max}$, $LD_{\max}$) **then**
            return False
        end if
        Appendo al vettore D_v la coppia $\langle EA_{\max}[u], LD_{\max}[u]\rangle$
    end for
    Trovo i due minimi del vettore D_v, e li metto nelle tuple $(i_1, LD_1), (i_2, LD_2)$
    for all $u \in \text{child}(v)$ **do**
        if $index(EA_{max}[u]) = i_1$ **then**
            if $EA_{max}[u] > LD_2$ **then** ▷ Controllo $EA_{max}[u]$ con LD_2
                return False
            end if
        else
            if $EA_{max}[u] > LD_1$ **then** ▷ Controllo $EA_{max}[u]$ con LD_1
                return False
            end if
        end if
    end for
    $EA \leftarrow \max_{u\in\text{child}(v)} EA_{\max}[u]$
    $LD \leftarrow \min_{u\in\text{child}(v)} LD_{\max}[u]$
    NextEA $\leftarrow$ Successor(L_v, EA)
    NextLD $\leftarrow$ Predecessor(L_v, LD)
    if NextEA = -1 $\vee$ NextTime = -1 **then**
        $EA_{\max}[v] \leftarrow \infty$
        $LD_{\max}[v] \leftarrow \infty$
        Check=False
    else
        $EA_{\max}[v] \leftarrow$NextEA
        $LD_{\max}[v] \leftarrow$NextLD
        Check=True
    end if
    return Check
end procedure

L'algoritmo che richiamerà questa funzione sarà il seguente:

Algorithm 5 Algoritmo Completo

```
procedure ALGORITMO(Albero T)
    EA_max ← vettore vuoto di dimensione n
    LD_max ← vettore vuoto di dimensione n
    Check ← False
    Check ← Visita-DFS(Radice di T, EA_max, LD_max)
    if Check = False then
        return False
    else
        return True
    end if
end procedure
```

3.4 Ordinamento dei timestamp

Finora abbiamo supposto che i timestamp associati agli archi siano ordinati in ordine crescente, ma questa condizione non è sempre garantita. Se i timestamp non fossero ordinati, le query di successore e predecessore non potrebbero essere eseguite in tempo logaritmico $O(\log(L))$, bensì richiederebbero un tempo lineare $O(L)$.
Questo comporterebbe un significativo rallentamento delle operazioni, facendo lievitare il costo computazionale da $O(n\log(L))$ a $O(nL)$.

Per risolvere questo problema, possiamo semplicemente ordinare ogni L_V in tempo

$$O(|L_V|\log(|L_V|))$$

.
Sommando su tutti i nodi v, otteniamo

$$\sum_v O(|L_V|\log(L)) = O(M\log(L))$$

Dove M è il numero totale di etichette associate agli archi dell'albero.

In questo caso il costo del nostro algoritmo che testa la connettività è quindi:

$$O(M\log(L)) + O(n\log(L)) = O(M\log(L))$$

3.5 Risultati Sperimentali

In questa sezione vedremo vari test effettuati su un dataset preso in input, in modo da vedere l'andamento dell'algoritmo proposto rispetto all'algoritmo Naive.

L'algoritmo Naive è quello che controlla, per ogni coppia di nodi $u, v \in V$, se esiste un cammino temporale $u \to v$, usando una vista DFS temporale. (L'implementazione dell'algoritmo Naive la possiamo trovare nella sezione **A.1** dell'Appendice dei Codici).

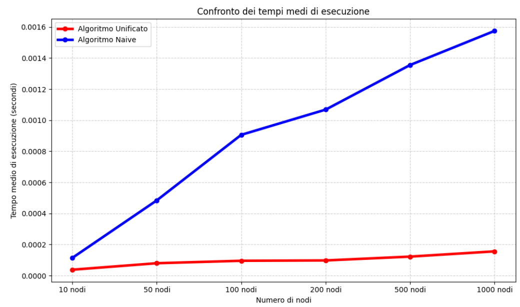
Tutti i test sono stati effettuati usando le implementazioni dei codici in linguaggio Python [6]
Gli alberi sono stati implementati usando il modulo NetworkX [7] di Python.
I grafici sono stati creati con il modulo Matplotlib [8] sempre di Python.

3.5.1 Variazione dei tempi medi dei due algoritmi

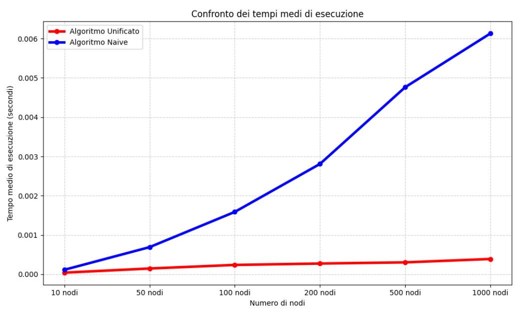
In questo secondo test, vedremo alcuni grafici che mostrano come variano i tempi medi di esecuzione dei due algoritmi, al crescere della dimensione delle istanze.
Per dimensione delle istanze intendiamo il numero di nodi n e numero totale di timestamp su arco L.
Questi due valori crescono secondo le seguenti metriche:

- Per ogni test sono state effettuate 100 prove diverse
- Per tutti i test il numero di nodi è $n = [10, 50, 100, 200, 500, 1000]$
- I test usano i seguenti valori $L = [25, 70, 150, 300, 2n]$ rispettivamente.

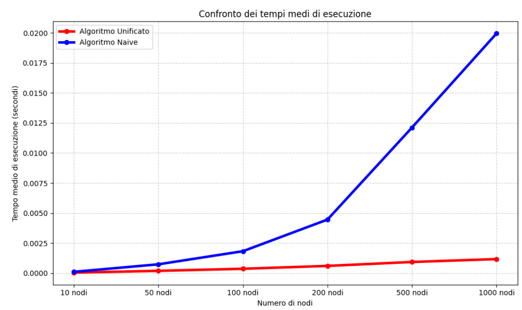
I risultati sono i seguenti :



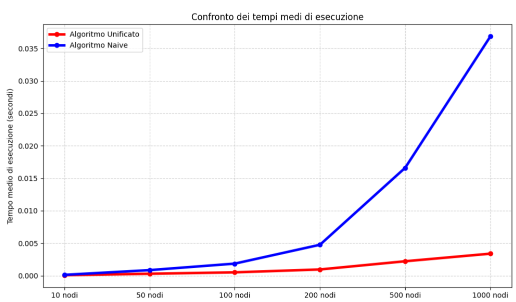
(a) Test con al più $n = 1000$ e $L = 25$



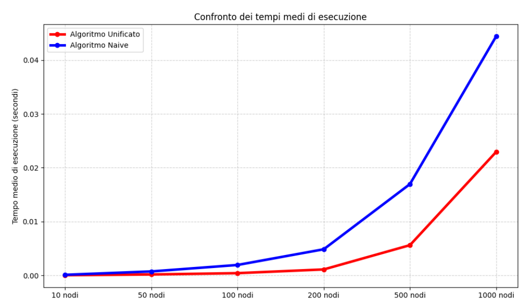
(b) Test con al più $n = 1000$ e $L = 70$



(c) Test con al più $n = 1000$ e $L = 150$



(d) Test con al più $n = 1000$ e $L = 300$



(e) Test con al più $n = 1000$ e $L = n \cdot 2$

Figura 3: Serie di esperimenti condotti su alberi generati casualmente

4 Sviluppi Futuri e Conlusioni

4.1 Conclusioni

In questa tesi abbiamo affrontato il problema della connettività temporale negli alberi.
Dopo aver fornito le definizioni formali e aver analizzato le difficoltà computazionali legate a questa formulazione, abbiamo sviluppato un algoritmo quasi lineare in grado di verificare la connettività temporale di un albero.
L'algoritmo proposto sfrutta una visita strutturata dell'albero per analizzare la compatibilità temporale delle connessioni tra i nodi, garantendo un'efficienza significativamente migliore rispetto a un approccio ingenuo.
Questa efficienza è ottenuta grazie a una gestione ottimizzata delle etichette temporali e al calcolo di valori specifici, che ci permettono di stabilire se i nodi possono connettersi fra loro oppure no.
I test sperimentali condotti su alberi di grandi dimensioni hanno confermato l'efficacia del nostro approccio.
In particolare, il nostro algoritmo ha dimostrato di essere molto più veloce rispetto alla soluzione naïve, con un miglioramento sostanziale su ogni tipo di istanza, sia istanze piccole che molto grandi.
Questo risultato evidenzia il contributo pratico del metodo sviluppato, rendendolo una soluzione altamente scalabile per la verifica della connettività temporale negli alberi.
I risultati ottenuti forniscono quindi un contributo significativo alla comprensione del problema della connettività temporale negli alberi, aprendo al tempo stesso nuove direzioni di ricerca che potrebbero estendere e migliorare l'approccio proposto.

4.2 Sviluppi Futuri

Un primo possibile sviluppo riguarda l'estensione del metodo ai grafi generali.
Il problema della connettività temporale è ancora più complesso nei grafi con cicli, poiché la presenza di percorsi alternativi introduce ulteriori sfide nella gestione della propagazione temporale.
Un adattamento dell'algoritmo attuale a grafi generali potrebbe quindi essere un passo naturale per future ricerche.
Un altro aspetto importante è lo studio della complessità computazionale ottimale del problema.
Sebbene il nostro algoritmo sia quasi lineare, potrebbe essere utile indagare se esiste un limite inferiore teorico che dimostri l'impossibilità di ottenere soluzioni più efficienti, oppure se nuove strategie algoritmiche possano ridurre ulteriormente il costo computazionale.
Infine, un'altra possibile direzione di ricerca riguarda varianti del problema, come la connettività temporale con vincoli addizionali (ad esempio limiti sulla latenza tra nodi, restrizioni di capacità sugli archi o vincoli su percorsi obbligati).
Queste estensioni potrebbero trovare applicazioni concrete in ambiti come le reti di comunicazione dinamiche, la simulazione della diffusione di informazioni e la biologia computazionale.

A Appendice dei codici

A.1 Algoritmo Naïve

Implementazione dell'algoritmo Naïve, ovvero l'algoritmo che impiega tempo

$$O(n^2M)$$

```python
def exists_temporal_path_for_nodes(path, tree):
    """
    Data una lista di nodi (path) e il grafo (tree),
    verifica se esiste un cammino temporale valido, cioe\' se e\' possibile
    scegliere, per ogni nodo del percorso, un timestamp tale che la sequenza
    dei timestamp sia strettamente crescente.

    Se un nodo non ha timestamp (weight e\' None), lo consideriamo come [0].
    """
    last_time = -float('inf')
    for node in path:
        # Ottieni la lista dei timestamp associata al nodo;
        # se il nodo non ha timestamp, usiamo [0] come default.
        timestamps = tree.nodes[node].get("weight")
        if timestamps is None:
            timestamps = [0]
        # Cerca il piu\' piccolo timestamp che sia maggiore di last_time
        found = False
        for t in timestamps:
            if t >= last_time:
                last_time = t
                found = True
                break
        if not found:
            return False
    return True
```

Algoritmo Principale

```python
def naive_temporal_connectivity(tree):
    """
    Algoritmo naive per verificare la temporal connectivity.

    Per ogni coppia ordinata di nodi (u, v) nel grafo:
      - Se esiste un cammino da u a v (usando nx.has_path), si recupera il percorso
        (dato che in un albero esiste un solo semplice cammino).
      - Si verifica se lungo questo percorso esiste una sequenza di timestamp
        strettamente crescente.

    Restituisce True se per ogni coppia (u, v) esiste un cammino temporale valido,
    False altrimenti.
    """
    nodes = list(tree.nodes())
    for u in nodes:
        for v in nodes:
            if u == v:
                continue
            # Se non esiste un cammino diretto da u a v, la connettivita\' temporale non e\' soddisfatta
            if not nx.has_path(tree, u, v):
                return False
            # Recupera il percorso (in un albero il percorso e\' unico)
            path = nx.shortest_path(tree, u, v)
            if not exists_temporal_path_for_nodes(path, tree):
                return False
    return True
```

A.2 Algoritmo Separato

Implementazione in codice python dell'algoritmo per la fase di preprocessing :

```python
def preprocessing(tree, v, EA_max, LD_max):
    """
    Procedura di Preprocessing per il nodo v.

    Parametri:
      - tree: grafo (networkx.DiGraph) rappresentante l'albero.
      - v: nodo corrente.
      - EA_max, LD_max: dizionari (globali o passati come argomento)
   in cui salvare i valori.
    """
    # Recupera L_v (lista dei timestamp sull'arco entrante in v)
    L_v = tree.nodes[v].get("weight", [])
    children = list(tree.successors(v))

    # Caso base: se v e\' foglia
    if len(children) == 0:
        if L_v_sorted:
            EA_max[v] = min(L_v)
            LD_max[v] = max(L_v)

    # Richiama ricorsivamente il preprocessing per ciascun figlio
    for u in children:
        preprocessing(tree, u, EA_max, LD_max)

    # Calcola EA e LD aggregati dai figli
    EA = max(EA_max[u] for u in children)
    LD = min(LD_max[u] for u in children)

    # Calcola NextEA e NextLD usando le funzioni di ricerca sulla
   lista L_v
    NextEA = successor(L_v, EA)
    NextLD = predecessor(L_v, LD)

    if NextEA == -1 or NextLD == -1:
        EA_max[v] = float('inf')
        LD_max[v] = float('inf')
    else:
        EA_max[v] = NextEA
        LD_max[v] = NextLD
```

Implementazione in codice python dell'algoritmo per la fase di check della temporal connectivity :

```python
def check_temporal_connectivity(tree, v, EA_max, LD_max):
    """
    Procedura che controlla la connettivita\' temporale per il sottoalbero radicato in v.

    Parametri:
      - tree: grafo (networkx.DiGraph) rappresentante l'albero.
      - v: nodo corrente.
      - EA_max, LD_max: dizionari contenenti i valori calcolati in preprocessing.
    """
    D_v = []  # Vettore vuoto di dimensione pari al numero di figli di v
    children = list(tree.successors(v))

    # Se v e\' foglia, la connettivita\' temporale e\' verificata
    if len(children) == 0:
        return True

    # Per ogni figlio u di v
    for u in children:
        if not check_temporal_connectivity(tree, u, EA_max, LD_max):
            return False
        if EA_max[u] == float('inf') or LD_max[u] == float('inf'):
            return False
        else:
            D_v.append((EA_max[u], LD_max[u]))

    # Trova i due minimi (basandosi sul valore EA_max) nel vettore D_v
    if len(D_v) == 1:
        minEA, LD1 = D_v[0]
        # Se c'e\' un solo elemento, usiamo lo stesso valore per entrambi i minimi
        secondEA, LD2 = D_v[0]
    else:
        sorted_D = find_2_min(D_v)
        minEA, LD1 = sorted_D[0]
        secondEA, LD2 = sorted_D[1]

    # Determina quale figlio ha il minimo EA_max (i1)
    children_list = list(tree.successors(v))
    min_child = None
    for u in children_list:
        if EA_max[u] == minEA:
            min_child = u
            break

    # Per ciascun figlio u di v, esegui i controlli:
```

```
    for u in children_list:
        if u == min_child:
            # Se u e\' il figlio con il minimo EA_max, controlla EA_max[u] con LD2
            if EA_max[u] > LD2:
                return False
        else:
            # Altrimenti, controlla EA_max[u] con LD1
            if EA_max[u] > LD1:
                return False
    return True
```

Implementazione dell'algoritmo completo, quello che richiamerà le due funzioni viste sopra

```
def AlgoritmoSeparato(T):
    """
    Algoritmo per la verifica della connettivit\`a temporale di un albero.

    T: grafo orientato rappresentante l'albero
    """
    n=T.number_of_nodes()
    EA_max = [0]*(n+1)
    LD_max = [0]*(n+1)
    preprocess(T, 1, EA_max, LD_max)
    check = check_temporal_connectivity(T, 1, EA_max, LD_max)
    if check:
        return True
    else:
        return False
```

A.3 Algoritmo Unificato

Implementazione della visita DFS che unisce le due procedure viste in precedenza

```python
def visit_dfs(tree, node, EA_max, LD_max):
    """
    Visita DFS unificata che, per ogni nodo v:
      - Se v e\' foglia, imposta EA_max[v] = min(L_v) e LD_max[v] = max(L_v)
      - Altrimenti, visita ricorsivamente i figli, raccoglie le coppie (EA_max, LD_max)
        e controlla la consistenza degli intervalli secondo lo pseudocodice.
      - Infine, calcola EA = max_{u in child(v)} EA_max[u] e LD = min_{u in child(v)} LD_max[u],
        e determina NextEA e NextLD in L_v (lista dei pesi associati a v) tramite funzioni di ricerca (binary_search e binary_search_leq).

    Se per qualche ragione il calcolo fallisce (NextEA o NextLD == -1), imposta EA_max[v] e LD_max[v] a infinito e ritorna False, altrimenti ritorna True.

    Parametri:
      - tree: grafo orientato rappresentante l'albero.
      - node: nodo corrente.
      - EA_max: array per memorizzare i valori EA_max per ciascun nodo.
      - LD_max: array per memorizzare i valori LD_max per ciascun nodo.

    Ritorna:
      - True se la connettivita\' temporale risulta consistente lungo il ramo, False altrimenti.
    """
    # L_v: vettore dei pesi associato al nodo corrente
    weights = tree.nodes[node].get("weight", [])
    children = list(tree.successors(node))

    # Caso base: foglia
    if not children:
        EA_max[node] = min(weights)
        LD_max[node] = max(weights)
        return True

    # Inizializza il vettore D_v per raccogliere le coppie dai figli
    D_v = []
    for child in children:
        if not visit_dfs(tree, child, EA_max, LD_max):
            return False
        if EA_max[child] == float('inf') or LD_max[child] == float('inf'):
```

```
            return False
        D_v.append((EA_max[child], LD_max[child]))

    # Se sono presenti almeno due figli, eseguo il controllo di consistenza
    if len(D_v) > 1:
        # Trovo i due minimi rispetto a EA_max:
        # (min1, ld1) e\' la coppia con EA_min piu\' piccolo e (min2, ld2) la seconda
        sorted_intervals = find_2_min(D_v)
        minEA, ld1 = sorted_intervals[0]
        secondEA, ld2 = sorted_intervals[1]
        # Per ogni figlio, effettuo il controllo:
        for child in children:
            if EA_max[child] == minEA:
                # Se il figlio con EA_min ha un EA_max maggiore di ld2, la condizione non e\' soddisfatta
                if EA_max[child] > ld2:
                    return False
            else:
                # Gli altri devono rispettare EA_max[u] <= ld1
                if EA_max[child] > ld1:
                    return False

    if node==1:
        return True

    # Calcolo EA e LD aggregati dai figli
    EA = max(EA_max[child] for child in children)
    LD = min(LD_max[child] for child in children)

    # 'binary_search' trova il successore di EA in weights,
    # 'binary_search_leq' trova il predecessore di LD.
    NextEA = binary_search(weights, EA) if weights else -1
    NextLD = binary_search_leq(weights, LD) if weights else -1

    if (NextEA == -1 or NextLD == -1) and node!=1:
        EA_max[node] = float('inf')
        LD_max[node] = float('inf')
        return False
    elif NextEA != -1 and NextLD != -1:
        EA_max[node] = NextEA
        LD_max[node] = NextLD
        return True
```

Implementazione dell'algoritmo completo

```
def algoritmo_unificato(T):
    """
    Algoritmo unificato per la verifica della connettivita\'
   temporale di un albero.

    T: grafo orientato rappresentante l'albero.
    """
    n=T.number_of_nodes()
    EA_max = [0]*(n+1)
    LD_max = [0]*(n+1)
    if visit_dfs(T,1, EA_max, LD_max): #1 e\' la radice
        return "L'albero e\' temporalmente connesso"
    else:
        return "L'albero non e\' temporalmente connesso"
```

Riferimenti bibliografici

[1] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "*Path Problems in Temporal Graphs*," 2014.

[2] P. Holme, "*Temporal networks*," *In Encyclopedia of Social Network Analysis and Mining, pages 2119–2129*, Springer, 2014.

[3] D. Kempe, J. Kleinberg, and A. Kumar, "*Connectivity and inference problems for temporal networks*," *Journal of Computer and System Sciences, 64(4):820–842*, 2002.

[4] M. Barjon, A. Casteigts, S. Chaumette, C. Johnen, and Y. M. Neggaz, "*Testing temporal connectivity in sparse dynamic graphs*," *CoRR, abs/1404.7634*, 2014.

[5] D. Bilò, G. D'Angelo, L. Gualà, S. Leucci, and M. Rossi, "*Blackout-tolerant temporal spanners*," *J. Comput. Syst. Sci., 141:103495*, 2024.

[6] Python. https://www.python.org/.

[7] NetworkX. https://networkx.org/.

[8] Matplotlib. https://matplotlib.org/.