Algoritmo Funzionante

Algoritmo NAIVE

```python
def dfs_path_check(u, target, adj_list, current_time, visited):
    """DFS che verifica se esiste un percorso temporale valido da u a target."""
    if u == target:
        return True
    visited.add(u)

    for neighbor, timestamps in adj_list[u]:
        # Considera solo timestamp >= current_time per rispettare la condizione di crescita
        valid_timestamps = [t for t in timestamps if t >= current_time]
        if valid_timestamps:
            next_time = min(valid_timestamps)  # Scegli il minimo valido per continuare la DFS
            if neighbor not in visited:
                # Continua DFS per verificare se si può raggiungere il target
                if dfs_path_check(neighbor, target, adj_list, next_time, visited):
                    return True

    visited.remove(u)
    return False

def is_temporally_connected_v2(adj_list):
    # Step 1: Per ogni coppia di nodi (u, v), verifica se esiste un percorso temporale valido
    nodes = list(adj_list.keys())
    for u in nodes:
        for v in nodes:
            if u != v:
                visited = set()  # Traccia i nodi visitati per ogni coppia (u, v)
                if not dfs_path_check(u, v, adj_list, 1, visited):  # Partendo dal timestamp minimo 1
                    return False  # Se esiste una coppia non connessa temporalmente, ritorna False
    return True
```

Poco da dire, l'algoritmo ha complessità $O(N^2 \cdot M)$

Algoritmo Ottimizzato

L'algoritmo è il seguente

```
from collections import defaultdict, deque

def compress_timestamps(adj_list):
    # Estrai tutti i timestamp
    all_timestamps = set()
    for u in adj_list:
        for v, timestamps in adj_list[u]:
            all_timestamps.update(timestamps)

    # Ordina e assegna un indice a ogni timestamp per la compressione
    sorted_timestamps = sorted(all_timestamps)
    timestamp_index = {t: i for i, t in enumerate(sorted_timestamps)}
    return timestamp_index, sorted_timestamps

def bfs_temporal(u, target, adj_list, timestamp_index, sorted_timestamps):
    # BFS per cercare di raggiungere il target rispettando i tempi
    queue = deque([(u, 0)])  # Ogni elemento è una coppia (nodo, indice
timestamp)
    visited = {u: 0}  # Traccia il timestamp minimo visitato per ciascun
nodo

    while queue:
        current, time_idx = queue.popleft()

        if current == target:
            return True

        for neighbor, timestamps in adj_list[current]:
            # Considera solo timestamp >= sorted_timestamps[time_idx]
            for t in timestamps:
                if t >= sorted_timestamps[time_idx]:  # Usa
sorted_timestamps correttamente
                    next_time_idx = timestamp_index[t]
                    # Visita solo se non è già stato visitato con un tempo
minore
                    if neighbor not in visited or visited[neighbor] >
next_time_idx:
                        visited[neighbor] = next_time_idx
                        queue.append((neighbor, next_time_idx))
                    break  # Prendi solo il primo timestamp valido

    return False

def is_temporally_connected_v5(adj_list):
    # Step 1: Comprimi i timestamp
```

```
    timestamp_index, sorted_timestamps = compress_timestamps(adj_list)

    # Step 2: Verifica la connessione temporale per ogni coppia di nodi
    nodes = list(adj_list.keys())
    for u in nodes:
        for v in nodes:
            if u != v:
                if not bfs_temporal(u, v, adj_list, timestamp_index,
sorted_timestamps):
                    return False  # Se una coppia non è connessa, ritorna
False
    return True
```

Dimostrazione

La **dimostrazione di correttezza** dell'algoritmo si basa sul fatto che l'algoritmo verifica correttamente se esiste un percorso temporale valido tra ogni coppia di nodi nel grafo, seguendo la logica definita nella descrizione dell'algoritmo. La correttezza si fonda su due aspetti principali:

1. **Rilevare correttamente la connessione temporale tra due nodi**.
2. **Gestire correttamente la crescita temporale durante la BFS**.

Descrizione dell'algoritmo

- **Compressione dei timestamp**: L'algoritmo prima raccoglie tutti i timestamp associati agli archi, li ordina e li mappa su indici compatti. Questo step permette di evitare la gestione esplicita di timestamp duplicati e consente di lavorare su una rappresentazione più efficiente.
- **BFS temporale**: Per ogni coppia di nodi u e v , l'algoritmo esegue una BFS che esplora i nodi adiacenti di u e cerca di raggiungere v , rispettando la condizione di crescita temporale (un nodo può essere visitato solo se il suo timestamp è maggiore o uguale a quello del nodo precedente).

Dimostrazione di correttezza

1. Correttezza nella ricerca del percorso temporale valido (verifica della connessione temporale)

L'algoritmo utilizza una **BFS** per esplorare il grafo tenendo traccia dei timestamp. La BFS esplora i vicini di un nodo solo se il timestamp dell'arco che li collega è maggiore o uguale al timestamp dell'arco precedente.

- **Proprietà di crescita temporale**: La condizione fondamentale dell'algoritmo è che per ogni arco tra due nodi, il timestamp dell'arco deve essere maggiore o uguale a quello dell'arco precedente (quindi crescente nel tempo). La BFS implementa correttamente

questa condizione, dato che per ogni arco esamina solo quelli con timestamp >= a quello del nodo da cui si è arrivati.

La **condizione di crescita temporale** assicura che la BFS esplori i nodi in modo che il percorso rispettato sia valido temporalmente. Quindi, se troviamo un percorso che collega due nodi u e v , il percorso rispetterà sempre la condizione di crescita temporale, il che significa che **ogni arco nel percorso avrà un timestamp che cresce o resta costante**.

2. Correttezza nella gestione dei timestamp

Poiché l'algoritmo ordina i timestamp prima di eseguire la BFS, è sicuro che il processo di esplorazione avvenga in ordine crescente o uguale dei timestamp. Questo è fondamentale per garantire che non ci siano "salti" temporali durante l'esplorazione del grafo, rispettando la condizione che i nodi possano essere raggiunti solo tramite archi con timestamp crescente.

- L'ordinamento dei timestamp permette di determinare rapidamente quale sia il prossimo arco da esplorare e garantisce che la BFS visiti i nodi solo quando possibile secondo la regola di crescita temporale.

3. Correttezza per ogni coppia di nodi

L'algoritmo esegue una BFS per ogni coppia di nodi u e v , verificando se esiste un percorso temporale valido da u a v . Se per una qualsiasi coppia di nodi non esiste un percorso valido (ossia, la BFS non riesce a raggiungere v da u), l'algoritmo ritorna **False**, indicando che i nodi non sono temporaneamente connessi.

- La **BFS** garantisce che tutti i nodi che sono connessi a u nel rispetto della condizione temporale siano esplorati.
- Se il nodo v è raggiungibile partendo da u seguendo la regola dei timestamp crescenti, la BFS troverà il percorso e restituirà **True**.
- Se nessun percorso valido è trovato, la funzione restituirà **False** per quella coppia di nodi.

Poiché l'algoritmo esamina tutte le possibili coppie di nodi, la correttezza complessiva del risultato è garantita: l'algoritmo restituirà **True** se e solo se ogni nodo è connesso temporalmente a tutti gli altri nodi, rispettando la condizione di crescita temporale.

4. Completamento dell'esplorazione

Ogni volta che eseguiamo una BFS, esploriamo solo archi con timestamp maggiore o uguale rispetto al timestamp del nodo precedente, quindi non saltiamo mai nessun arco che possa violare la condizione di crescita temporale. Inoltre, l'ordinamento dei timestamp garantisce che esploreremo sempre il percorso più "vecchio" possibile per rispettare la

crescita temporale, e poiché esploriamo tutti gli archi, **l'algoritmo trova sempre un percorso valido** (se esiste).

Conclusione

L'algoritmo è **corretto** perché:

- La BFS garantisce che ogni percorso trovato rispetti la condizione di crescita temporale.
- L'ordinamento dei timestamp assicura che non vengano saltati archi validi.
- Per ogni coppia di nodi u, v , la BFS verificherà correttamente se esiste un percorso temporale valido. Se non esiste, l'algoritmo restituirà correttamente **False**.

In questo modo, l'algoritmo verifica correttamente la connessione temporale tra tutte le coppie di nodi nel grafo.

Complessità

Se K rappresenta il numero di **etichette totali** (ovvero il numero di timestamp distinti che compaiono tra tutti gli archi), la complessità cambia di nuovo in modo significativo. Ecco come possiamo analizzare la situazione in base a questa definizione di K .

1. Compressione dei Timestamp con K come numero totale di etichette

Se K è il numero totale di etichette distinte (o timestamp) tra tutti gli archi, questo significa che dovremo ordinare e associare un indice a ciascun timestamp. In questo caso, la complessità della **compressione dei timestamp** diventa:

$$O(K \log K)$$

Dove K è il numero totale di etichette. Questo passaggio è relativamente costoso, ma viene eseguito solo una volta.

2. BFS Temporale per ciascuna coppia di nodi

La BFS temporale esplora gli archi, e per ciascun arco dovremo considerare i timestamp associati. In questo caso, la BFS esplora gli archi e i loro timestamp, e nel peggiore dei casi può essere necessario verificare tutti i timestamp associati a ciascun arco. Se ogni arco ha al massimo K timestamp, allora:

- La **BFS** esplorerà ogni arco e considererà al massimo K timestamp, il che comporta un costo di O(M \cdot K) per ogni chiamata alla BFS, dove M è il numero di archi.

Complessità Totale

1. **Compressione dei timestamp**:

- Ordinamento dei timestamp distinti: $O(K \log K)$.

2. **BFS per tutte le coppie di nodi**:
   - Per ogni coppia di nodi, eseguiamo una BFS che ha un costo di $O(M \cdot K)$ (poiché esploriamo gli archi e, per ciascun arco, possiamo esaminare fino a K timestamp).

Poiché dobbiamo eseguire la BFS per ogni coppia di nodi, il costo totale dell'algoritmo diventa:

$$O(K \log K + N^2 \cdot M \cdot K)$$

Dove:

- N è il numero di nodi,
- M è il numero di archi,
- K è il numero totale di etichette (timestamp distinti).

3. Caso particolare: K = N-1 (una sola etichetta per arco)

Se hai solo una singola etichetta per arco, ovvero ogni arco ha un solo timestamp, allora:

- K = N - 1 , e la **compressione dei timestamp** diventa $O((N-1)\log(N-1)) = O(N \log N)$.
- La **BFS** per ogni coppia di nodi ha un costo di O(M) , perché ogni arco ha solo un timestamp da esaminare.

In questo caso, la complessità totale dell'algoritmo diventa:

$$O(N \log N + N^2 \cdot M)$$

4. Caso particolare: $K \gg N$ (molti timestamp per arco)

Se K è molto grande, ad esempio $K \gg N$, allora la complessità diventa:

$$O(K \log K + N^2 \cdot M \cdot K)$$

Sintesi della Complessità

- Se K è piccolo (ad esempio K = 1 o costante), la complessità è sostanzialmente O(N^2 \cdot M) , che è molto più gestibile.
- Se K è grande (ad esempio $K \gg N$, la complessità può crescere rapidamente, in particolare a causa del termine $O(N^2 \cdot M \cdot K)$.

Considerazioni finali

- **Se K è piccolo o moderato**, la complessità totale può essere accettabile, ma se K cresce significativamente, l'algoritmo può diventare costoso, soprattutto se hai molti archi e un numero elevato di timestamp.

- In questo caso, ottimizzare la parte di esplorazione dei timestamp, per esempio limitando il numero di timestamp considerati durante la BFS, o implementando tecniche più efficienti di ricerca dei timestamp validi, potrebbe essere utile per ridurre la complessità.

Algoritmo Ottimizzato con approccio Dijkstra-like

Algoritmo

```
# Dijkstra-like

import heapq

def temporal_bfs(u, adj_list, n):
    """Esegui una BFS che esplora i nodi partendo da u, rispettando
l'ordine temporale dei timestamp"""
    # Coda di priorità (heap), contiene tuple del tipo (timestamp, nodo)
    heap = []
    # Inizializza la coda con i nodi vicini di u, con il minimo timestamp
    for neighbor, timestamps in adj_list[u]:
        for timestamp in timestamps:
            heapq.heappush(heap, (timestamp, neighbor))

    visited = set()
    visited.add(u)

    while heap:
        current_time, current_node = heapq.heappop(heap)

        if current_node not in visited:
            visited.add(current_node)

            # Aggiungi i vicini di current_node alla coda se non sono già
stati visitati
            for neighbor, timestamps in adj_list[current_node]:
                if neighbor not in visited:
                    for timestamp in timestamps:
                        if timestamp >= current_time:  # Solo timestamp
validi
                            heapq.heappush(heap, (timestamp, neighbor))

    return visited

def is_temporally_connected_v3(adj_list):
    """Verifica se il grafo è temporaneamente connesso, per ogni coppia di
nodi"""
    nodes = list(adj_list.keys())
    n = len(nodes)
```

```
    # Per ogni nodo u, esegui BFS temporale per trovare tutti i nodi
raggiungibili da u
    for u in nodes:
        reachable = temporal_bfs(u, adj_list, n)

        # Se un nodo non è raggiungibile da u, il grafo non è connesso
temporalmente
        for v in nodes:
            if v != u and v not in reachable:
                return False

    return True
```

Complessità

Strategia: Uso di Dijkstra-like con timestamp

L'idea di base è di non eseguire una BFS per ogni coppia di nodi, ma piuttosto una ricerca "temporalmente ottimizzata" su tutta la rete, usando le informazioni sui timestamp per limitare l'esplorazione.

Idea di base dell'approccio

Invece di eseguire una BFS per ogni coppia di nodi, possiamo fare una **Dijkstra-like** per esplorare il grafo partendo da ogni nodo u . All'interno di questa ricerca, gestiamo il **minimo timestamp** necessario per proseguire, usando una coda di priorità per scegliere gli archi con il timestamp più basso.

1. **Ogni nodo ha una coda di archi** ordinata per timestamp crescente.
2. **Per ogni nodo**, esploriamo i suoi archi **in ordine crescente di timestamp**.
3. **La coda di priorità** garantisce che gli archi con timestamp più basso vengano esplorati per primi.

Passaggi dell'approccio:

1. **Preprocessing (ordinamento degli archi):**
   - Per ogni nodo u , ordina gli archi uscenti in base ai timestamp. Questo impone un costo di $O(M \log M)$ nel caso peggiore, dove M è il numero di archi. Se ogni arco ha un solo timestamp, il costo per ordinare gli archi per ogni nodo è $O(M \log M)$.
2. **Ricerca dei percorsi temporali:**
   - Partendo da ogni nodo u , eseguiamo una **ricerca con priorità temporale**. Per ogni nodo u , esploriamo i suoi vicini v seguendo i timestamp in ordine crescente.
   - Manteniamo una coda di priorità (heap) che ci permette di esplorare i vicini con il **minimo timestamp possibile**.

3. **Operazioni della coda di priorità (heap):**
    - Ogni operazione di inserimento e estrazione dalla coda di priorità ha un costo di O(\log M) (dato che ci sono al massimo M archi).
    - In totale, la complessità di una ricerca di tipo Dijkstra per un singolo nodo diventa O(M \log M).

Combinazione dei passi

- Per ogni nodo u , dobbiamo eseguire una ricerca, quindi il costo per ogni nodo diventa O(M \log M).
- L'algoritmo complessivo sarà quindi:

[
O(N \cdot M \log M)
]

Questo è significativamente più efficiente di O(N^2 \cdot M) , soprattutto se M è molto più grande di N .

Considerazioni finali

Questo approccio sfrutta la struttura del problema usando **una coda di priorità** per esplorare i nodi in modo più efficiente, riducendo la necessità di una ricerca completa per ogni coppia di nodi. Inoltre, **l'ordinamento degli archi** è fondamentale per assicurarsi che ogni esplorazione avvenga nel corretto ordine temporale, rispettando i vincoli del problema.

Quindi, il costo finale O(N \cdot M \log M) è una **significativa ottimizzazione rispetto a O(N^2 \cdot M)** e, se M è davvero molto maggiore di N , questa complessità può risultare molto più veloce.

Correttezza

1. Proprietà fondamentale dell'algoritmo:

L'algoritmo cerca di determinare se per ogni coppia di nodi u e v esiste un percorso valido temporale, in cui i timestamp sugli archi che compongono il percorso siano crescenti. In altre parole, se siamo in un nodo u e vogliamo raggiungere un nodo v , dobbiamo esplorare solo gli archi che rispettano il vincolo temporale $t_{u \to v} \geq t_{u \to u}$, dove $t_{u \to v}$ è il timestamp dell'arco da u a v .

2. Usare una coda di priorità (Dijkstra-like):

Dijkstra è un algoritmo che esplora i nodi di un grafo partendo da un nodo di origine, scegliendo sempre il percorso minimo (o, in questo caso, il percorso temporale con il

timestamp minimo) attraverso una coda di priorità. Nel nostro caso, la **priorità** non è il "peso" degli archi, ma il **timestamp** associato a ciascun arco.

L'idea è simile a Dijkstra, ma invece di minimizzare la distanza, minimizziamo il tempo, cioè scegliamo gli archi con il timestamp più basso.

3. Correttezza dell'approccio:

La correttezza di questo approccio si basa sul fatto che:

- Quando esploriamo un nodo u , lo facciamo considerando solo gli archi che rispettano il vincolo temporale: selezioniamo solo archi (u, v) in cui il timestamp è maggiore o uguale al timestamp del nodo corrente.
- La coda di priorità garantisce che esploriamo prima i nodi attraverso gli archi con il timestamp più basso possibile, il che implica che esploreremo i percorsi temporali in ordine crescente di timestamp.

Dimostrazione di correttezza:

A. Proprietà di scelta ottimale:

Ogni volta che esploriamo un nodo u , lo facciamo scegliendo l'arco con il **minimo timestamp** (questo è simile a Dijkstra, che sceglie il percorso di peso minimo). Questo assicura che esploriamo sempre il percorso più "veloce" (temporalmente più vicino), in modo da rispettare la condizione di crescita dei timestamp lungo il percorso.

B. Proprietà di esplorazione corretta:

Quando esploriamo un nodo u e passiamo a un nodo v , esaminiamo solo gli archi che partono da u e che hanno un timestamp maggiore o uguale a quello corrente. In altre parole, esploriamo i percorsi che rispettano la condizione di crescita dei timestamp. Se esiste un percorso valido temporale da u a v , lo troveremo esplorando questi archi in ordine di timestamp crescente.

C. Correttezza del percorso:

Se esiste un percorso temporale valido da u a v , il nostro algoritmo lo troverà:

- Ogni volta che esploriamo un nodo v a partire da un nodo u , lo facciamo con un timestamp che rispetta la condizione di crescita.
- Poiché esploriamo prima gli archi con il timestamp più basso, garantiamo che ogni percorso temporale che esploriamo sia valido.

Se riusciamo a raggiungere un nodo v partendo da u , vuol dire che esiste un percorso temporale valido tra u e v , rispettando la condizione sui timestamp.

4. Rispetto del vincolo di crescita:

Il nostro algoritmo rispetta sempre il vincolo di crescita sui timestamp, poiché:

- Gli archi sono esplorati solo se il loro timestamp è maggiore o uguale al timestamp corrente.
- La coda di priorità gestisce l'esplorazione in modo che non vengano mai esplorati archi con timestamp "retroattivi", mantenendo così la correttezza temporale del percorso.

5. Verifica globale della connettività temporale:

Nel nostro algoritmo, per ogni nodo u , esploriamo tutti i nodi raggiungibili da u rispettando i vincoli temporali. Dopo aver eseguito la ricerca per un nodo, verifichiamo che tutti gli altri nodi siano raggiungibili. Se, per qualsiasi nodo u , esiste un nodo v che non è raggiungibile partendo da u , il grafo non è temporaneamente connesso, e quindi l'algoritmo restituirà **False**.

Conclusioni sulla correttezza:

- L'algoritmo è corretto perché esplora i nodi in ordine di timestamp crescente, garantendo che ogni percorso temporale che esplora sia valido.
- Ogni nodo viene visitato solo quando il timestamp degli archi che partono da esso soddisfa il vincolo temporale.
- La coda di priorità assicura che esploriamo sempre il percorso temporale più promettente, ossia quello con il timestamp più basso.
- Alla fine dell'esecuzione, l'algoritmo verifica la connettività temporale globale, restituendo **True** se ogni coppia di nodi è connessa temporalmente e **False** altrimenti.

In sintesi, il comportamento del nostro algoritmo garantisce che esploreremo correttamente tutti i percorsi temporali validi, rispettando i vincoli sui timestamp, e quindi è **corretto**.

Algoritmo Dijkstra-Like ottimizzato con Memoization

```
def temporal_bfs_memo(u, adj_list, memo):
    """Esegue una BFS temporale con memorizzazione (memoization)"""
    # Coda di priorità (heap), contiene tuple (timestamp, nodo)
    heap = []
    heapq.heappush(heap, (0, u))  # Partiamo da u con il timestamp minimo

    # Inizializza il dizionario memo per u
    if u not in memo:
        memo[u] = {u: 0}
    visited = memo[u]
```

```
    while heap:
        current_time, current_node = heapq.heappop(heap)

        # Esplora i vicini di current_node
        for neighbor, timestamps in adj_list[current_node]:
            # Trova il primo timestamp >= current_time
            idx = bisect_left(timestamps, current_time)
            if idx < len(timestamps):
                next_time = timestamps[idx]
                # Se il vicino non è stato visitato o se troviamo un
percorso temporale migliore
                if neighbor not in visited or next_time <
visited.get(neighbor, float('inf')):
                    visited[neighbor] = next_time
                    heapq.heappush(heap, (next_time, neighbor))

    # Restituisce i nodi raggiungibili
    return set(visited.keys())

def is_temporally_connected_v5(adj_list):
    """Verifica se il grafo è temporaneamente connesso usando la
memorizzazione dei percorsi."""
    nodes = list(adj_list.keys())
    memo = defaultdict(dict)  # Dato che vogliamo lanciare BFS per ogni
nodo

    for u in nodes:
        reachable = temporal_bfs_memo(u, adj_list, memo)
        # Se un nodo non è raggiungibile da u, il grafo non è connesso
temporalmente
        if len(reachable) != len(nodes):
            return False
    return True
```

Complessità

Facciamo l'analisi della complessità dell'algoritmo, partendo dall'interno verso l'esterno:

1. Complessità della `temporal_bfs_memo`

Questa funzione esegue una BFS temporale per ogni nodo `u` con l'ottimizzazione della memorizzazione. Analizziamo ogni passo:

a. Heap e Aggiunta degli Elementi

- All'inizio, `heapq.heappush(heap, (0, u))` impiega $O(\log 1) = O(1)$ per inserire il nodo di partenza `u`.

- Durante l’esplorazione, ogni volta che aggiungiamo un nuovo elemento all'heap, il costo è $O(\log k)$, con `k` che rappresenta il numero di elementi nella coda in quel momento.

b. Iterazione sui Vicini e Uso di `bisect_left`

Per ogni nodo corrente `current_node`, visitiamo i suoi vicini. Il costo per ogni vicino è composto da:

- **Ricerca dei Timestamp**: `bisect_left(timestamps, current_time)` ha complessità $O(\log M)$ per ogni ricerca, dove M è il numero massimo di timestamp associati agli archi nel grafo.

c. Visita e Aggiornamento dei Nodi

Quando un vicino `neighbor` è trovato a un `next_time` valido:

- **Aggiornamento di `visited`**: Aggiungere un nuovo elemento o aggiornare un timestamp minimo richiede O(1) perché `visited` è un dizionario.
- **Aggiunta all'Heap**: Ogni aggiunta all’heap costa $O(\log k)$, ma in media si mantiene su $O(\log M)$, poiché il numero di timestamp totali è limitato da M.

Dunque, per ogni arco, il costo è $O(\log M)$, e in un grafo con N nodi e M timestamp totali (sommando tutti i timestamp sui vari archi), il costo della BFS è in media:

$$O(M \log M)$$

2. Complessità Complessiva di `is_temporally_connected_v4`

Ora, guardiamo la complessità totale dell’algoritmo:

- **Chiamate a `temporal_bfs_memo`**: Eseguiamo `temporal_bfs_memo` per ciascuno dei N nodi.
  - Se ciascuna BFS costa $O(M \log M)$, il costo complessivo diventa $O(N \cdot M \log M)$.
- **Controllo di Connessione**: Dopo ogni BFS, verifichiamo se tutti i nodi sono stati raggiunti.
  - Questo richiede O(N) per ogni chiamata a `temporal_bfs_memo`, quindi il costo è trascurabile rispetto al costo di $O(N \cdot M \log M)$.

Complessità Complessiva

L'algoritmo ha complessità temporale totale di:

$$O(N \cdot M \log M)$$

Spazio

- **Spazio per `memo`**: Memorizziamo i timestamp minimi per ogni nodo esplorato in ciascuna BFS. Nella peggiore delle ipotesi, la complessità spaziale per `memo` è $O(N \cdot M)$.

Riassunto della Complessità

- **Tempo**: $O(N \cdot M \log M)$
- **Spazio**: $O(N \cdot M)$

Questo algoritmo mantiene il costo a $O(N \cdot M \log M)$ senza ripetere inutilmente l'esplorazione di sottografi, rendendolo più efficiente.

Dimostrazione

Per dimostrare la correttezza dell'algoritmo, analizziamo i suoi obiettivi principali e come ciascun componente contribuisce a raggiungerli:

Obiettivo

L'obiettivo dell'algoritmo `is_temporally_connected_v4` è verificare se il grafo è **connesso temporalmente**. Cioè, deve esistere un percorso da ogni nodo `u` a ogni altro nodo `v`, in cui le etichette temporali sugli archi lungo il percorso rispettano un ordine non decrescente.

Struttura della Dimostrazione

1. **Correctness della `temporal_bfs_memo`**: Dimostriamo che `temporal_bfs_memo` trova tutti i nodi raggiungibili da `u` con un percorso temporale valido.
2. **Correctness di `is_temporally_connected_v4`**: Dimostriamo che l'algoritmo globale verifica correttamente la connessione temporale.

Parte 1: Correctness di `temporal_bfs_memo`

La funzione `temporal_bfs_memo` effettua una BFS temporale a partire da un nodo `u` e trova tutti i nodi raggiungibili rispettando la condizione temporale.

Passaggi Chiave della `temporal_bfs_memo`

1. **Inizializzazione della Coda**: `temporal_bfs_memo` inserisce `u` nella coda con un timestamp iniziale di 0, permettendo di considerare ogni arco disponibile a partire da `u`.
2. **Esplorazione con Heap**:
    - L'heap garantisce che esploriamo sempre il prossimo arco disponibile con il timestamp più basso, rispettando la condizione di ordine non decrescente dei

timestamp.
    - Quando estraiamo un nodo `current_node` con `current_time`, cerchiamo tra i suoi vicini `neighbor` solo quei timestamp che sono validi (i timestamp $\geq$ `current_time`).
3. **Memorizzazione dei Minimi Timestamp (`memo`)**:
    - `memo` tiene traccia del minimo timestamp con cui abbiamo visitato ciascun nodo, per evitare cicli o percorsi ridondanti.
    - Se troviamo `neighbor` con un timestamp inferiore al minimo salvato in `memo`, aggiungiamo `neighbor` all'heap e aggiorniamo `memo[neighbor]`.
4. **Condizione di Arresto**:
    - La BFS termina quando l'heap è vuoto, garantendo che tutti i nodi raggiungibili in modo temporale da `u` siano stati esplorati.
    - Alla fine, `visited` contiene tutti i nodi accessibili in ordine temporale a partire da `u`.

Conclusione

La `temporal_bfs_memo` restituisce quindi l'insieme di tutti i nodi `v` tali che esiste un percorso da `u` a `v` con timestamp in ordine non decrescente, rispettando la connessione temporale a partire da `u`.

Parte 2: Correctness di `is_temporally_connected_v4`

La funzione `is_temporally_connected_v4` verifica se il grafo è temporalmente connesso, ovvero se ogni nodo può raggiungere ogni altro nodo con un percorso che rispetti l'ordine temporale.

Passaggi Chiave di `is_temporally_connected_v4`

1. **Esecuzione di `temporal_bfs_memo` da ogni nodo `u`**:
    - L'algoritmo invoca `temporal_bfs_memo` per ogni nodo `u` nel grafo, trovando l'insieme `reachable` di nodi che possono essere raggiunti da `u` rispettando i vincoli temporali.
2. **Controllo di Connessione Completa**:
    - Dopo l'esecuzione di `temporal_bfs_memo(u)`, l'algoritmo verifica che ogni altro nodo `v` nel grafo appartenga a `reachable`.
    - Se esiste un nodo `v` non raggiungibile da `u`, allora il grafo non è connesso temporalmente, e l'algoritmo restituisce `False` immediatamente.
3. **Copertura Completa di Tutte le Coppie**:
    - L'algoritmo esegue `temporal_bfs_memo` per ogni nodo, quindi verifica l'accessibilità reciproca per tutte le coppie di nodi.

- Se nessuna chiamata a `temporal_bfs_memo` trova una coppia non connessa temporalmente, l’algoritmo conferma che il grafo è temporalmente connesso e restituisce `True`.

Conclusione

`is_temporally_connected_v4` garantisce la connettività temporale globale del grafo controllando che ogni nodo possa raggiungere tutti gli altri nodi tramite percorsi che rispettano i vincoli temporali. L'algoritmo è quindi corretto per determinare la connessione temporale.

Conclusione Finale

Il nostro algoritmo è corretto perché:

1. **`temporal_bfs_memo`** esplora correttamente tutti i percorsi temporali validi da un nodo `u`.
2. **`is_temporally_connected_v4`** verifica che ogni nodo sia temporalmente connesso a ogni altro nodo, coprendo tutte le coppie possibili.

In questo modo, l'algoritmo rispetta i vincoli temporali e garantisce la connettività temporale del grafo.