UNIVERSITÀ DEGLI STUDI DI ROMA TOR VERGATA

MACROAREA DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA IN INFORMATICA

SPANNER E ORACOLI PER IL PROBLEMA DEL K-WAYPOINT ROUTING IN GAFI TEMPORALI

Relatore:
Prof.
LUCIANO GUALÀ

Candidato:
FRANCESCO COSCIOTTI
Matricola: 0323545

Anno Accademico 2024/2025

A chi ha saputo capire
quando era il momento di ricominciare

Indice

Introduzione

I grafi rappresentano una delle strutture dati più versatili e diffuse nell'informatica moderna, poiché consentono di modellare in modo efficace una vasta gamma di fenomeni complessi della vita reale, come reti di trasporto, sistemi di comunicazione o interazioni sociali.

Tuttavia, in molte applicazioni pratiche, le relazioni tra gli elementi non sono statiche, ma variano nel tempo. Di conseguenza, l'esistenza di un arco tra due nodi può dipendere dall'istante temporale considerato.
Per gestire tali contesti dinamici, negli ultimi anni si è affermato un crescente interesse verso i **grafi temporali**: a differenza dei grafi statici, essi associano a ciascun arco uno o più istanti di disponibilità, permettendo così di rappresentare in modo accurato scenari in cui le connessioni evolvono nel tempo.

L'introduzione della dimensione temporale modifica in maniera sostanziale anche la definizione di *cammino*. Mentre in un grafo statico l'esistenza di un cammino tra due nodi è garantita dalla sola presenza degli archi corrispondenti, in un grafo temporale essa dipende dal rispetto dell'ordine cronologico: gli archi devono essere attraversati in sequenza temporale non decrescente, condizione necessaria per la validità del cammino.

In questa tesi ci si concentrerà sui cammini di tipo waypoint routing.
Dati due nodi s e t, un cammino k-waypoint routing con waypoint $x_1, ..., x_k$ è un cammino temporale da s a t che passa per tutti i nodi $x_1, ..., x_k$ (in qualsiasi ordine). Si noti che k è un parametro indicante il numero di waypoint attraverso i quali il cammino deve passare.

Nel presente lavoro si definiscono due problemi:

Il primo problema è quello di progettare un oracolo, ovvero una struttura dati definita su un grafo temporale G e due nodi specifici s e t che sia in grado di rispondere a domande del tipo:

dati k specifici waypoint, esiste un cammino k-waypoint routing con tali waypoint?

La qualità di un oracolo si misura generalmente secondo diversi parametri:

- **dimensione** (lo spazio occupato dall'oracolo)
- **query** (il tempo necessario all'oracolo per processare una query)
- **building time** (il tempo necessario per la costruzione dell'oracolo)

Il secondo problema, correlato al primo, si basa sul trovare un *k-waypoint rounting spanner* per G rispetto a due nodi s e t, ovvero un sottografo H di G che preserva l'esistenza di cammini k-waypoint routing di G per ogni possibile k-tupla di waypoint. La qualità di uno spanner si misura generalmente secondo le seguenti misure:

- **dimensione dello spanner** (il numero di archi di H)
- **building time** (il tempo necessario per la costruzione dello spanner)

Nella seguente tabella vengono presentati i risultati conseguiti alll'interno della tesi:

	k-waypoint routing		**k-waypoint routing with Window**	
	Oracolo	**Spanner**	**Oracolo**	**Spanner**
$k = 1$				
Building time	$O(n + M)$	$O(n + M)$	$O(n + M)$	—
Dimensione	$O(n)$	$O(n)$	$O(n + M)$	—
Query time	$O(1)$	—	$O(\log M)$	—
$k \geq 2$				
Dimensione	—	$\Omega(n^2)$	—	$\Omega(n^2)$

Tabella 1: Risultati conseguiti riguardanti i problemi considerati.

1 Definizioni

1.1 Grafo Temporale

Sia $G = (V, E, \lambda)$ un grafo temporale non diretto tale che:

- V è l'insieme dei nodi di G, con $|V| = n$;
- E è l'insieme degli archi di G, con $|E| = m$;
- λ è una funzione che associa a ciascun arco $e = (u, v) \in E$, $\forall u, v \in V$, un insieme finito di timestamp, ovvero
$$\lambda : E \to 2^{\mathbb{N}}.$$

1.2 Arco Temporale

Dato un arco $e = (u, v) \in E$ e l'insieme dei timestamp ad esso associati $\lambda(e)$, definiamo un *arco temporale* come la tripla:

$$\tau = (u, v, t) \quad \forall t \in \lambda(e).$$

1.3 Stream Temporale

Dato un grafo temporale $G = (V, E, \lambda)$, esso può essere rappresentato attraverso il proprio *stream temporale*, ossia una sequenza ordinata di archi temporali:

$$Stream(G) = (\tau_1, \tau_2, \ldots, \tau_M)$$

dove ciascun arco temporale è della forma

$$\tau_i = (u, v, t_i)$$

con $u, v \in V$, $t_i \in \lambda(u, v)$ e $M = |Stream(G)|$ il numero totale di archi nello stream.

Gli archi temporali sono ordinati in base al tempo, ovvero vale la relazione:

$$t_1 \leq t_2 \leq \cdots \leq t_M,$$

che garantisce che lo stream sia disposto in ordine cronologico crescente rispetto ai timestamp degli archi.

1.4 Finestra Temporale

Sia $G = (V, E, \lambda)$ definito come sopra. Una *finestra temporale* è un intervallo $[\alpha, \beta]$, con $\alpha, \beta \in \mathbb{N}$, sul quale si costruisce un sottografo di G, indicato con:

$$G_{\alpha,\beta} = (V, E', \lambda)$$

dove

$$E' = \{\, e \in E : \exists\, t \in \lambda(e) \text{ tale che } t \in [\alpha, \beta] \,\}.$$

In altre parole, $G_{\alpha,\beta}$ è il sottografo di G composto esclusivamente dagli archi i cui timestamp appartengono all'intervallo temporale $[\alpha, \beta]$.

1.5 Cammino Temporale

Sia π un cammino temporale da v_0 a v_k in G , con $v_i \in V$ per ogni $0 \leq i \leq k$, al quale ci si riferirà tramite la notazione $\pi(v_0, v_k)$.
Siano $\tau_1, \tau_2, \ldots, \tau_k$ gli archi temporali attraversati da π, tali che

$$\pi(v_0, v_k) = \langle \tau_0, \tau_1, \ldots, \tau_{k-1} \rangle,$$

dove ciascun arco temporale è della forma

$$\tau_j = (v_j, v_{j+1}, t_j) \quad \forall 0 \leq j < k.$$

Affinché π sia un cammino temporale valido, deve valere la condizione di non decrescenza dei timestamp:

$$t_j \leq t_{j+1} \quad \forall 0 \leq j < k - 1.$$

Di seguito saranno introdotte due definizioni utili che verranno utilizzate in seguito:

- **Tempo di partenza** del cammino π:

  $$start(\pi) = t_0$$

  ovvero il primo timestamp incontrato lungo il cammino.

- **Tempo di arrivo** del cammino π:

  $$end(\pi) = t_{k-1}$$

  ovvero l'ultimo timestamp incontrato lungo il cammino.

1.5.1 Earliest arrival path

Sia $G = (V, E, \lambda)$ un grafo temporale.
Siano $v_0, v_k \in V$ due nodi di G tali che v_0 è il nodo sorgente e v_k è il nodo destinazione.
Sia:

$$P(v_0, v_k) = \{\pi(v_0, v_k) \mid \forall \pi(v_0, v_k) \text{ è un cammino temporale valido in } G \text{ da } v_0 \text{ a } v_k\}$$

l'insieme di tutti i cammini temporali validi in G che collegano v_0 a v_k.
Definiamo come *earliest arrival path* il cammino $p \in P(v_0, v_k)$ tale che il tempo di arrivo al nodo destinazione sia minimo, ovvero:

$$end(p) = \min\{\, end(\pi) : \pi \in P(v_0, v_k) \,\}.$$

Il valore $end(p)$ si definisce come l'**earliest arrival time** dal nodo v_0 al nodo v_k e si indica con:

$$ea[v_0, v_k] = end(p)$$

1.5.2 Earliest arrival Tree

Sia $G = (V, E, \lambda)$ un grafo temporale come definito in precedenza, e sia $v_0 \in V$ un nodo sorgente. Per ogni vertice $v \in V$, indichiamo con $P(v_0, v)$ l'insieme degli *earliest arrival path* da v_0 a v.

A partire da tali cammini, si può definire l'*earliest arrival tree* come la struttura che rappresenta, per ciascun vertice, il cammino di arrivo più precoce che lo collega alla sorgente.
Formalmente, l'*earliest arrival tree* è definito come la tripla:

$$T_{ea}[v_0] = (V, E', \lambda),$$

dove:

- $T_{ea}[v_0]$ è un albero radicato in v_0;
- $E' \subseteq E$ è l'insieme degli archi appartenenti all'insieme degli earliest arrival path, ossia $E' = \{e \in E \mid e \in P(v_0, v),\ \forall v \in V\}$;
- λ conserva le informazioni temporali originali associate agli archi.

1.5.3 Latest departure path

Sia $G = (V, E, \lambda)$ un grafo temporale.
Siano $v_0, v_k \in V$ due nodi di G tali che v_0 è il nodo sorgente e v_k è il nodo destinazione.
Sia:

$$P(v_0, v_k) = \{\pi(v_0, v_k) \mid \forall \pi(v_0, v_k) \text{ è un cammino temporale valido in } G \text{ da } v_0 \text{ a } v_k\}$$

l'insieme di tutti i cammini temporali validi in G che collegano v_0 a v_k.
Definiamo come *latest departure path* il cammino $p \in P(v_0, v_k)$ tale che il tempo di partenza dal nodo sorgente sia massimo, ovvero:

$$start(p) = \max\{\, start(\pi) : \pi \in P(v_0, v_k) \,\}.$$

Il valore $start(p)$ prende il nome di **latest departure time** da v_0 a v_k e si indica con:

$$ld[v_0, v_k] = start(p)$$

1.5.4 Latest departure Tree

Sia $G = (V, E, \lambda)$ un grafo temporale come definito in precedenza, e sia $v_0 \in V$ un nodo destinazione. Per ogni vertice $v \in V$, indichiamo con $P(v_0, v)$ l'insieme dei *latest departure path* verso v_0 da v.

A partire da tali cammini, si può definire il *latest departure tree* come la struttura che rappresenta, per ciascun vertice, il *latest departure path* che lo collega alla destinazione.
Formalmente, il *latest departure tree* è definito come la tripla:

$$T_{ld}[v_0] = (V, E', \lambda),$$

dove:

- $T_{ld}[v_0]$ è un albero radicato in v_0;
- $E' \subseteq E$ è l'insieme degli archi appartenenti all'insieme dei latest departure path, ossia $E' = \{e \in E \mid e \in P(v_0, v),\ \forall v \in V\}$;
- λ conserva le informazioni temporali originali associate agli archi.

Osservazione. Si può osservare come, essendo l'albero composto interamente da latest departure path da ogni nodo v verso la destinazione v_0, per tutti i cammini temporali $\pi(v_0, v) = \langle \tau_0, \tau_1, \ldots, \tau_k \rangle$ vale la relazione:

$$t_j \geq t_{j+1} \quad \forall 0 \leq j \leq k.$$

E quindi per esser reso valido basterà semplicemente invertire l'ordine di visita degli archi nel cammino.

2 Algoritmi per il calcolo dei tempi di percorrenza nei grafi temporali

In questo capitolo vengono presentati gli algoritmi impiegati per la risoluzione del problema, basati su due procedure note in letteratura per il calcolo degli *earliest arrival time* e dei *latest departure time* in grafi temporali. Dopo averne descritto il funzionamento originale, vengono illustrate le modifiche apportate per adattarli al modello proposto in questa tesi.

2.1 Definizioni preliminari

È opportuno osservare che la definizione di grafo temporale adottata in [1] differisce leggermente da quella utilizzata in questa tesi.
Nel lavoro originale, infatti, un grafo temporale è definito come una coppia $G = (V, E')$ con E' l'insieme degli archi temporali, anziché come una tripla $G = (V, E, \lambda)$.
Infine ciascun arco temporale è rappresentato dalla quadrupla:

$$e = (u, v, t, \phi) \in E'$$

dove $t \in \mathbb{N}$ indica l'istante in cui l'arco e può essere attraversato e $\phi \in \mathbb{N}$ rappresenta il tempo di percorrenza dell'arco.

Le due rappresentazioni risultano tuttavia *concettualmente equivalenti*. Nel caso particolare in cui il tempo di percorrenza sia nullo ($\phi = 0$), un arco temporale in [1] del tipo $e = (u, v, t, \phi)$ può essere interpretato come un'istanza del modello adottato in questa tesi, corrispondente all'arco temporale

$$\tau = (u, v, t) \in Stream(G),$$

Pertanto, la differenza tra le due definizioni è puramente formale e non sostanziale ai fini del problema trattato.

2.2 Algoritmo per il calcolo degli *earliest arrival time*

2.2.1 Spiegazione dell'algoritmo per il calcolo degli earliest arrival time in [1]

L'algoritmo proposto in [1] calcola, per ogni vertice $v \in V$, il tempo minimo necessario per raggiungerlo a partire da una sorgente x.

In input riceve lo *stream* temporale del grafo $G = (V, E)$, un nodo sorgente $x \in V$ e un intervallo temporale $[t_\alpha, t_\omega]$, dove:

- t_α rappresenta il tempo di partenza dal nodo sorgente;
- t_ω rappresenta il limite massimo di tempo entro cui considerare i cammini validi.

L'algoritmo inizializza $ea[x] = t_\alpha$ e $ea[v] = \infty$ per ogni altro nodo $v \in V \setminus \{x\}$.

Successivamente scorre, in ordine crescente di tempo, tutti gli archi temporali $e = (u, v, t, \phi)$ presenti nello *stream*.
Per ciascun arco verifica se:

$$t \geq ea[u] \quad \wedge \quad t + \phi < ea[v] \quad \wedge \quad t + \phi \leq t_\omega,$$

cioè se l'arco può essere effettivamente percorso (l'istante t non precede l'arrivo a u e il tempo di arrivo $t + \phi$ rientra nell'intervallo di interesse).
Se queste condizioni sono soddisfatte, allora si aggiorna il valore $ea[v]$:

$$ea[v] \leftarrow t + \phi.$$

Questo meccanismo garantisce che, al termine dell'esecuzione, per ogni nodo $v \in V$ la variabile $ea[v]$ contenga l'*earliest arrival time* dal nodo sorgente x entro l'intervallo $[t_\alpha, t_\omega]$.

2.2.2 Adattamento dell'algoritmo per il calcolo degli earliest arrival time

L'algoritmo è stato adattato al modello di grafo adottato in questa tesi con le seguenti modifiche:

- si trascurano i vincoli di intervallo temporale ponendo $t_\alpha = -\infty$ e $t_\omega = \infty$;
- si considera $\phi = 0$, ignorando i tempi di percorrenza.

Algorithm 1: *earliest arrival time algorithm*

Input: Un grafo temporale $G = (V, E, \lambda)$ nella sua rappresentazione a *stream* di archi, un nodo sorgente $x \in V$
Output: Il vettore ea contenente, per ogni vertice $v \in V$, l'*earliest arrival time* da x

```
1 Inizializza il vettore ea come segue;
2     ea[x] ← 0;
3     ea[v] ← ∞ per ogni v ∈ V \ {x};
4 foreach arco temporale τ = (u, v, t) nello stream do
5     if t ≥ ea[u] e t < ea[v] then
6         ea[v] ← t;
7 return ea;
```

2.2.3 Variante dell'algoritmo per il calcolo dell'earliest arrival Tree

L'algoritmo precendente, con poche modifiche, può essere adattato in modo da generare un earliest arrival Tree radicato in x.

Algorithm 2: *earliest arrival Tree Algorithm*

Input: Un grafo temporale $G = (V, E, \lambda)$ nella sua rappresentazione a *stream* di archi, un nodo sorgente $x \in V$

Output: L'albero temporale $T_{ea}[x]$ contenente, per ogni vertice $v \in V$, l' *earliest arrival path* da x verso v

```
1  Inizializza l'albero T_ea[x] radicato in x;
2  Inizializza il vettore ea come segue;
3      ea[x] ← 0;
4      ea[v] ← ∞ per ogni v ∈ V \ {x};
5  foreach arco temporale τ = (u, v, t) nello stream do
6      if t ≥ ea[u] e t < ea[v] then
7          ea[v] ← t;
8          stacca l'eventuale nodo v in T_ea[x];
9          crea l'arco temporale (u, v, t) in T_ea[x];
10 return T_ea[x];
```

2.3 Algoritmo per il calcolo dei *latest departure time*

2.3.1 Spiegazione dell'algoritmo per il calcolo dei latest departure time in [1]

L'algoritmo proposto in [1] calcola, per ogni vertice $v \in V$, il tempo massimo con cui lasciarlo per raggiungere una sorgente x.

In input riceve lo *stream* temporale del grafo $G = (V, E)$, un nodo sorgente $x \in V$ e un intervallo temporale $[t_\alpha, t_\omega]$ come sopra.
L'algoritmo inizializza $ld[x] = t_\omega$ e assegna $ld[v] = -\infty$, $\forall v \in V \setminus \{x\}$.

Successivamente scorre, in ordine decrescente di tempo, tutti gli archi temporali $e = (u, v, t, \phi)$ presenti nello *stream*.
Per ciascun arco verifica se:

$$t + \phi \geq t_\alpha \quad \wedge \quad t + \phi \leq ld[v] \quad \wedge \quad t > ld[u],$$

cioè se l'arco possa essere effettivamente percorso.
Se queste condizioni sono soddisfatte allora $ld[u]$ si aggiorna:

$$ld[u] \leftarrow t.$$

Questo meccanismo garantisce che, al termine dell'esecuzione, per ogni nodo $v \in V$ la variabile $ld[v]$ contenga il *latest departure time* verso la destinazione x entro l'intervallo $[t_\alpha, t_\omega]$.

2.3.2 Adattamento dell'algoritmo per il calcolo del latest departure time

Le modifiche adottate sono analoghe a quelle per l'algoritmo precedente:

- si trascurano gli intervalli di tempo ($t_\alpha = -\infty$, $t_\omega = \infty$);
- si assume $\phi = 0$;

Algorithm 3: *Latest departure time algorithm*

Input: Un grafo temporale $G = (V, E, \lambda)$ nella sua rappresentazione a *stream* di archi, una destinazione $x \in V$

Output: Il vettore ld contenente, per ogni vertice $v \in V$, il *latest departure time* verso x

```
1  Inizializza il vettore ld come segue;
2      ld[x] ← ∞;
3      ld[v] ← −∞, ∀v ∈ V \ {x};
4  foreach arco temporale τ = (u, v, t) nello stream in ordine inverso do
5      if t ≤ ld[v] e t > ld[u] then
6          ld[u] ← t;
7  return ld;
```

2.3.3 Variante dell'algoritmo per il calcolo del latest departure Tree

L'algoritmo precendente, con poche modifiche, può essere adattato in modo da generare, invece che un vettore di latest departure verso x a partire da tutti gli altri nodi $v \in V \setminus \{x\}$, un latest departure Tree radicato in x.

Algorithm 4: *Latest departure Tree algorithm*

Input: Un grafo temporale $G = (V, E, \lambda)$ nella sua rappresentazione a *stream* di archi, un nodo sorgente $x \in V$

Output: L'albero temporale $T_{ld}[x]$ contenente, per ogni vertice $v \in V$, il *latest departure path* da v verso x

```
1  Inizializza l'albero T_ld[x] radicato in x;
2  Inizializza il vettore ld come segue;
3      ld[x] ← ∞;
4      ld[v] ← −∞, ∀v ∈ V \ {x};
5  foreach arco temporale τ = (u, v, t) nello stream in ordine inverso do
6      if t ≤ ld[v] e t > ld[u] then
7          ld[u] ← t;
8          stacca l'eventuale nodo u in T_ld[x];
9          crea l'arco temporale (v, u, t) in T_ld[x];
10 return T_ea[x];
```

2.4 Correttezza degli algoritmi

In quanto le varianti degli algoritmi per il calcolo degli earliest arrival time e dei latest departure time provenienti da [1] non vanno a modificare in maniera sostanziale il funzionamento degli stessi, essendo gli algoritmi sopracitati già dimostrati, possiamo dire che entrambi gli algoritmi siano corretti.

2.5 Analisi delle complessità

Poiché gli algoritmi proposti per il calcolo degli *earliest arrival time*, dei *latest departure time* e delle rispettive varianti per la costruzione degli alberi condividono la stessa struttura generale, è sufficiente analizzare nel dettaglio la complessità del primo, in quanto le differenze tra gli altri sono limitate ai controlli condizionali, all'ordine di scansione dello *stream* e all'eventuale aggiornamento della struttura ad albero.

Analizzando l'algoritmo per il calcolo degli *earliest arrival time*, si osserva che:

- l'inizializzazione del vettore *ea* (riga 1) ha costo $O(n)$, dove $n = |V|$;
- la scansione dello *stream* temporale (riga 4) ha costo $O(M)$, dove $M = |Stream|$;
- tutte le altre operazioni (righe 5–7) hanno costo costante $O(1)$.

Ne consegue che la complessità temporale complessiva è $O(n + M)$ mentre la complessità spaziale è $O(n)$, dovuta alla memorizzazione del vettore dei tempi di arrivo.

Per quanto riguarda gli algoritmi che generano gli alberi (*earliest arrival tree* e *latest departure tree*), la complessità temporale rimane invariata ($O(n+M)$), mentre la complessità spaziale raddoppia asintoticamente $O(2n) = O(n)$ poiché oltre al vettore di dimensione n viene mantenuta in memoria anche la struttura dell'albero, anch'essa proporzionale al numero di vertici.

Algoritmo	**Complessità temporale**	**Complessità spaziale**
Earliest Arrival Time	$O(n + M)$	$O(n)$
Latest Departure Time	$O(n + M)$	$O(n)$
Earliest Arrival Tree	$O(n + M)$	$O(n)$
Latest Departure Tree	$O(n + M)$	$O(n)$

Tabella 2: Riepilogo delle complessità degli algoritmi proposti

3 Definizone del problema

3.1 k-waypoint routing

Utilizzando le definizioni introdotte nelle sezioni precedenti, possiamo ora formulare in modo preciso il problema oggetto di questa tesi.
Siano:

- $G = (V, E, \lambda)$ un grafo temporale non orientato;
- $s, t \in V$ due nodi di G, rispettivamente sorgente e destinazione.

L'obiettivo della tesi è di progettare e realizzare una struttura dati $\mathcal{O}$ in grado di rispondere a query del seguente tipo:
dato in input un insieme $K = \{x_1, \ldots, x_k\} \subseteq V$ di nodi, esiste un cammino temporale da s a t che attraversi, in qualsiasi ordine, tutti i nodi $x \in K$?

In altri termini, dato un insieme $K \subseteq V$, vogliamo determinare se esiste un cammino temporale valido π in G che soddisfi simultaneamente le seguenti condizioni:

- ha origine nel nodo sorgente s;
- termina nel nodo destinazione t;
- attraversa ciascun nodo $x_i \in K$ con $1 \leq i \leq k$, in un ordine arbitrario.

Formalmente, la query può essere espressa come:

$$\mathcal{Q}(K) = \begin{cases} True & \text{se} \quad \exists \pi(s, t \mid x_1, \ldots, x_k) \\ False & \text{se} \quad \nexists \pi(s, t \mid x_1, \ldots, x_k) \end{cases}$$

Una struttura dati di questo tipo è generalmente chiamata *oracolo*.
La qualità di un oracolo si valuta in termini di:

- **occupazione di memoria**, ovvero la dimensione dell'oracolo;
- **tempo di risposta** a una generica query (*query time*);
- **tempo di costruzione** (*building time*).

In questa tesi verrà inoltre considerato un problema correlato, quello di progettazione di uno *spanner*.
In particolare, si desidera trovare un sottografo H di G tale che, per ogni sottoinsieme $K = \{x_1, \ldots, x_k\} \subseteq V$ con $|K| = k$, esista in H un cammino temporale da s a t che attraversi tutti i vertici $x \in K$ se e solo se questo è presente anche in G.
Un tale sottografo viene comunemente chiamato *spanner*.

La qualità di uno spanner si valuta in termini di:

- **dimensione**, ovvero il numero di archi che contiene;
- **tempo di costruzione** (*building time*).

Si noti che uno spanner può essere pensato come un oracolo avente la stessa dimensione e lo stesso tempo di costruzione, ma con un *query time* banale, corrispondente al tempo necessario per esplorare esplicitamente H.

3.2 k-waypoint routing con finestra temporale

Oltre alla versione appena definita, analizziamo anche una variante del problema che introduce un ulteriore vincolo temporale.
Siano $G = (V, E, \lambda)$ e due nodi $s, t \in V$ definiti come sopra.
L'obiettivo è ora progettare una struttura dati $\mathcal{O}$ tale che, dati:

- un insieme $K \subseteq V$ di nodi con $|K| = k$;
- un intervallo di tempo $[\alpha, \beta]$ con $\alpha, \beta \in \mathbb{N}$;

sia in grado di rispondere in modo efficiente alla query $\mathcal{Q}(K, [\alpha, \beta])$.
In altre parole, dato un insieme K e un intervallo temporale $[\alpha, \beta]$, vogliamo determinare se esiste un **cammino temporale valido** π in G tale che:

- ha origine nella sorgente s;
- termina nella destinazione t;
- attraversa ciascun nodo $x_i \in K$ con $1 \leq i \leq k$, in un ordine arbitrario;
- ogni arco temporale $\tau \in \pi$ viene attraversato con tempo $t \in [\alpha, \beta]$.

Formalmente, la query può essere scritta come:

$$\mathcal{Q}(K, [\alpha, \beta]) = \begin{cases} True & \text{se} \quad \exists \pi(s, t \mid x_1, \ldots, x_k) \quad \forall \tau_i \in \pi \ \text{ t.c. } t_i \in [\alpha, \beta] \\ False & \text{se} \quad \nexists \pi(s, t \mid x_1, \ldots, x_k) \quad \forall \tau_i \in \pi \ \text{ t.c. } t_i \in [\alpha, \beta] \end{cases}$$

Diversamente dal problema precedente, verrà mostrato solo il problema utilizzando un'oracolo $\mathcal{O}$. Per quanto riguarda lo spanner, questo verrà utilizzato solo per estendere un risultato dal problema precedente a questo.

3.3 Modalità d'approccio

Per affrontare in modo efficace i due problemi definiti nelle sezioni precedenti, questa tesi adotta un **approccio incrementale** basato sulla cardinalità dell'insieme K dei nodi intermedi. Tale scelta è motivata dal fatto che la complessità del problema cresce rapidamente con l'aumentare di $|K|$. Analizzare progressivamente casi con cardinalità crescente consente quindi di studiare al meglio il problema, offrendo delle soluzioni il più ottimizzate possibile per il caso preso in considerazione.

Caso con $|K| = 1$ Come punto di partenza, verrà considerata la versione più semplice del problema, ovvero quella in cui l'insieme dei waypoint contiene un solo nodo. In questo scenario, la query si riduce a verificare l'esistenza di un cammino temporale valido da s a t che attraversi un singolo nodo intermedio x.

Caso con $|K| \geq 2$ Una volta compreso e risolto il caso con $|K| = 1$, verrà analizzata la versione più generalizzata del problema, in cui $|K| \geq 2$. In questa situazione, il cammino deve attraversare un insieme di nodi intermedi in ordine arbitrario, aumentando notevolmente la complessità del problema.

4 Soluzione del problema del k-waypoint routing

In questo capitolo vengono analizzate alcune varianti e strategie di risoluzione per il problema del *k-waypoint routing* su grafo temporale.

4.1 Uno spanner per il caso $k = 1$

In questa sottosezione viene presentata una procedura per la costruzione di uno spanner valido nel caso $k = 1$.
L'algoritmo proposto consente di ottenere una struttura con tempo di costruzione $O(n + M)$ e spazio occupato $O(n)$.

4.1.1 Ridefinizione del problema

Sia $G = (V, E, \lambda)$ un grafo temporale indiretto e siano $s, t \in V$ rispettivamente il nodo sorgente e il nodo destinazione.
L'obiettivo di questa variante è costruire uno *spanner temporale* valido $H = (V, E_H, \lambda)$ con $E_H \subseteq E$ in cui vale la relazione:

$$\forall x \in V \quad \exists \pi(s, t \mid x) \in H \iff \exists \pi(s, t \mid x) \in G$$

dove $\pi(s, t \mid x)$ denota un cammino temporale valido che parte da s, termina in t e attraversa il nodo x.

4.1.2 Approccio naïve

Un primo approccio, di tipo *naïve*, consiste nel mantenere in memoria l'intero grafo G.

È evidente quanto questo approccio non sia efficiente: lo spanner H coincide con G e non vi è alcuna riduzione della dimensione del grafo. La complessità spaziale risulta quindi $O(n+m)$ con l'unico vantaggio di avere un tempo di costruzione costante $O(1)$, poiché H coincide con il grafo di input.

4.1.3 Creazione dello spanner

La costruzione dello spanner $H = (V_H, E_H, \lambda_H)$ avviene in tre fasi principali:

Fase 1 Si calcola l'*earliest arrival tree* $T_{ea}[s] = (V, E_{ea}, \lambda)$ mediante l'algoritmo precedentemente descritto, assumendo come sorgente il nodo s.

Fase 2 Come nella Fase 1, si calcola il *latest departure tree* $T_{ld}[t] = (V, E_{ld}, \lambda)$, utilizzando il nodo t come destinazione.

Fase 3 Infine, si costruisce lo spanner combinando i due alberi in modo da avere $H = (V_H, E_H, \lambda_H)$ con:

$$V_H = V, \quad E_H = E_{ea} \cup E_{ld}, \quad \lambda_H = \lambda$$

Questo è lo pseudocodice dettagliato di come viene creato lo spanner H:

Algorithm 5: *spanner creation algorithm*

Input: Un grafo temporale $G = (V, E, \lambda)$, lo stream temporale di G $Stream(G)$, una sorgente s e una destinazione t
Output: Uno spanner temporale $H = (V_H, E_H, \lambda_H)$

```
1  T_ea ← earliestArrivalTreeAlgorithm(Stream(G), s);
2  T_ld ← latestDepartureTreeAlgorithm(Stream(G), t);
3  E_ea ← E(T_ea) ;                          // insieme degli archi di T_ea
4  E_ld ← E(T_ld) ;                          // insieme degli archi di T_ld
5  V_H ← V;
6  E_H ← ∅;
7  λ_H ← ∅;
8  foreach arco e = (u, v) in E_ea do
9      E_H ← E_H ∪ {e};
10     λ_H(e) ← λ(e) ;              // preserva le informazioni temporali
11 foreach arco e = (u, v) in E_ld do
12     if e ∉ E_H then
13         E_H ← E_H ∪ {e};
14         λ_H(e) ← λ(e) ;          // preserva le informazioni temporali
15 return H = (V_H, E_H, λ_H);
```

4.1.4 Correttezza dell'algoritmo

Viene definito ora un lemma molto importante all'interno di questa tesi.

Lemma 1. *Sia $G = (V, E, \lambda)$ un grafo temporale e $x \in V$.*
Siano $ea[x]$ l'earliest arrival time dal nodo sorgente v_0 al nodo x.
Sia $ld[x]$ il latest departure time dal nodo x verso la destinazione v_k.
Allora vale la seguente equivalenza:

$$ea[x] \leq ld[x] \quad \iff \quad \exists\, \pi(v_0, v_k \mid x) \text{ valido.}$$

Dimostrazione. ($\Rightarrow$)
Per definizione di $ea[x]$, esiste un cammino temporale valido $\pi_{ea}(v_0, x)$ tale che

$$\lambda(v_{i-1}, x) = ea[x].$$

Analogamente, per definizione di $ld[x]$, esiste un cammino valido $\pi_{ld}(x, v_k)$ tale che

$$\lambda(x, v_i) = ld[x].$$

Poiché per ipotesi $ea[x] \leq ld[x]$, la concatenazione $\pi = \langle v_0, \ldots, v_{i-1}, x, v_i, \ldots, v_k \rangle$ è un cammino temporale valido, in quanto rispetta la non decrescenza dei timestamp.
($\Leftarrow$) Per definizione $\pi(v_0, v_k \mid x) = \langle v_0, \ldots, v_{i-1}, x, v_i, \ldots, v_k \rangle$.
Essendo π un cammino valido, allora vale la relazione:

$$\lambda(v_{i-1}, x) \leq \lambda(x, v_i)$$

Ma noi sappiamo per definizione che $ea[x] = \lambda(v_{i-1}, x)$ per i motivi mostrati sopra, stessa cosa vale per $ld[x] = \lambda(x, v_i)$.
Quindi vale anche la relazione $\lambda(v_{i-1}, x) \leq \lambda(x, v_i)$ ovvero:

$$ea[x] \leq ld[x].$$

□

Dimostriamo ora la correttezza dello spanner costruito dall'algoritmo.

Teorema 1. *Sia $H = (V_H, E_H, \lambda_H)$ lo spanner prodotto dall'algoritmo di costruzione. Allora:*

$$\exists\, \pi(s, t \mid x) \text{ in } G \iff \exists\, \pi(s, t \mid x) \text{ in } H \quad \forall x \in V \setminus \{s, t\}.$$

Dimostrazione. Per dimostrare l'equivalenza, procediamo come segue:

$$A : \exists\, \pi(s, t \mid x) \text{ in } G \quad \text{e} \quad B : \exists\, \pi(s, t \mid x) \text{ in } H.$$

Dimostreremo separatamente:

$$A \Rightarrow B \quad \text{e} \quad \neg A \Rightarrow \neg B.$$

($A \Rightarrow B$) Supponiamo che in G esista un cammino temporale valido $\pi(s, t \mid x)$.
Dal Lemma 1 sappiamo che ciò implica:

$$ea_G[x] \leq ld_G[x].$$

Durante la costruzione dello spanner, l'algoritmo genera l'*earliest arrival tree* T_{ea} e il *latest departure tree* T_{ld}, che contengono rispettivamente tutti i cammini che realizzano gli *earliest arrival times* e i *latest departure times* per ciascun nodo.
Poiché H è definito come l'unione di tali alberi,

$$E_H = E_{ea} \cup E_{ld}$$

ne consegue che in H esiste un cammino da s a x con tempo di arrivo $ea_G[x]$ e un cammino da x a t con tempo di partenza $ld_G[x]$.
Essendo $ea_G[x] \leq ld_G[x]$, la loro concatenazione produce un cammino temporale valido in H.
Segue quindi che $A \Rightarrow B$.

($\neg A \Rightarrow \neg B$) Supponiamo ora che in G non esista alcun cammino temporale valido $\pi(s, t \mid x)$.
Per definizione, ciò implica che non esistono né un *earliest arrival path* da s a x, né un *latest departure path* da x a t tali da soddisfare $ea[x] \leq ld[x]$.
Poiché lo spanner H è costruito esclusivamente a partire dagli archi di G presenti in T_{ea} e T_{ld}, non può introdurre nuovi archi non esistenti in G.
Ne consegue che nessun nuovo cammino $\pi(s, t \mid x)$ può essere creato in H, e quindi B non è vero.
Segue dunque che $\neg A \Rightarrow \neg B$. □

4.1.5 Analisi dell'algoritmo per la creazione dello spanner

Analizzando lo pseudocodice riportato in precedenza, è possibile stimare la complessità di ciascun blocco dell'algoritmo come segue:

- le chiamate alle funzioni `earliest Arrival Tree Algorithm` e `latest Departure Tree Algorithm` (righe 1-2) presentano una complessità temporale pari a $O(n+M)$, come dimostrato in precedenza;
- le operazioni alle righe 3 e 4, che estraggono gli insiemi di archi E_{ea} ed E_{ld}, hanno complessità $O(n)$, poiché ciascun albero contiene esattamente $n-1$ archi;
- l'inizializzazione dell'insieme dei nodi V_H (riga 5) richiede anch'essa tempo $O(n)$;
- i due cicli `for` (righe 9–15) scorrono gli insiemi E_{ea} ed E_{ld}, ciascuno di cardinalità $O(n)$, per un costo complessivo $O(n)$ grazie al costo costante $O(1)$ delle operazioni al loro interno.

Si conclude quindi che la complessità temporale totale dell'algoritmo è:

$$O(n + M)$$

mentre la complessità spaziale risulta $O(n)$, in quanto le strutture dati aggiuntive (insiemi di archi e funzione λ_H) occupano spazio proporzionale al numero di nodi.

4.2 Un'oracolo per il caso $k = 1$

4.2.1 Definizione dell'oracolo

Si definisce un'oracolo $\mathcal{O}$ composto da due vettori:

- un vettore EA tale che $EA[x] = ea[x] \quad \forall x \in V$
- un vettore LD tale che $LD[x] = ld[x] \quad \forall x \in V$

4.2.2 Creazione dell'oracolo

La costruzione dell'oracolo $\mathcal{O}$ avviene in tre fasi principali.

Fase 1 Posta una sorgente $s \in V$, viene costruito un vettore EA che associa a ogni nodo $x \in V$ il proprio *earliest arrival time* $ea[x]$, calcolato mediante l'algoritmo `earliestArrivalTimeAlgorithm` presentato in precedenza.

Fase 2 Posta una destinazione $t \in V$, viene costruito un vettore LD che associa a ogni nodo $x \in V$ il proprio *latest departure time* $ld[x]$, calcolato mediante l'algoritmo `latestDepartureTimeAlgorithm`.

Fase 3 Infine, viene creato l'oracolo $\mathcal{O}$ unendo le informazioni contenute in EA e LD. Il risultato finale è una struttura dati che, per ogni nodo $x \in V$, memorizza la coppia $(ea[x], ld[x])$.

Algorithm 6: *oracle creation algorithm*

Input: Un grafo temporale $G = (V, E, \lambda)$, lo stream temporale di G $Stream(G)$, una sorgente s e una destinazione t
Output: Un vettore $\mathcal{O}$ con chiave x e valore $(ea[x], ld[x]), \forall x \in V$

```
1 EA ← earliestArrivalTimeAlgorithm(Stream(G), s);
2 LD ← latestDepartureTimeAlgorithm(Stream(G), t);
3 Inizializza il vettore O;
4 foreach nodo x ∈ V do
5 |   ea ← EA[x];
6 |   ld ← LD[x];
7 |   O[x] ← (ea, ld)
8 return O;
```

4.2.3 Correttezza dell'algoritmo

La correttezza dell'algoritmo di costruzione dell'oracolo discende direttamente dalla correttezza degli algoritmi per il calcolo degli *earliest arrival time* e dei *latest departure time*.
Infatti, l'unica operazione aggiuntiva eseguita consiste nella combinazione dei due vettori EA e LD, senza alterare la validità dei valori contenuti.

4.2.4 Analisi dell'algoritmo

La complessità temporale dell'algoritmo deriva unicamente dai due algoritmi per il calcolo degli *earliest arrival time* e dei *latest departure time*, con costo $O(n + M)$.
Le operazioni successive di costruzione e fusione dei dizionari hanno complessità $O(n)$, quindi trascurabili rispetto al termine dominante.
Ne consegue che la complessità complessiva dell'algoritmo è:

$$O(n + M).$$

La complessità spaziale risulta anch'essa $O(n)$, poiché è necessario memorizzare le informazioni relative a ciascun nodo del grafo.

4.2.5 Esecuzione della query

L'esecuzione della query $\mathcal{Q}(K)$, con $K = \{x\}$ e $x \in V$, avviene in due fasi distinte e sfrutta i valori precomputati contenuti all'interno dell'oracolo $\mathcal{O}$.

Fase 1 Vengono estratti dall'oracolo $\mathcal{O}$ i due valori associati al nodo x:

- $ea[x]$: l'*earliest arrival time*;
- $ld[x]$: il *latest departure time*.

Fase 2 Si confrontano i due valori estratti nella fase precedente. Se vale la relazione:

$$ea[x] \leq ld[x],$$

allora, per il Lemma 1, esiste un cammino temporale valido $\pi(s, t \mid x)$ nel grafo G, e la query $\mathcal{Q}(K)$ restituisce `True`; in caso contrario, restituisce `False`.

Algorithm 7: *oracle query algorithm*

Input: Un vettore $\mathcal{O}$ relativo ad un grafo temporale $G = (V, E, \lambda)$ e un nodo $x \in V$

Output: Un valore booleano indicante l'esistenza di un cammino $\pi(s, t \mid x)$

```
1 (ea, ld) ← O[x];
2 if ea ≤ ld then
3     return True;
4 return False;
```

4.2.6 Correttezza dell'algoritmo

La correttezza dell'algoritmo di query è garantita dal **Lemma 1**, secondo il quale:

$$ea[x] \leq ld[x] \iff \exists\, \pi(s, t \mid x) \text{ temporalmente valido.}$$

L'algoritmo, infatti, si limita a verificare questa condizione utilizzando i valori precomputati dall'oracolo. Pertanto, se il lemma è valido, anche l'output della query è corretto.

4.2.7 Analisi dell'algoritmo

L'algoritmo effettua un'unica operazione di accesso al vettore $\mathcal{O}$, seguita da un confronto tra due valori scalari. Entrambe le operazioni richiedono tempo costante, per cui la complessità temporale complessiva è O(1).
Analogamente, lo spazio aggiuntivo richiesto per l'esecuzione della query è trascurabile ($O(1)$), poiché non vengono allocate strutture dati ausiliarie.

4.3 Un lowerbound per il caso $k \geq 2$

4.3.1 Idea dietro al lowerbound

Ispirandosi alla costruzione del lower bound proposta in [2], si intende costruire un grafo temporale $G = (V, E, \lambda)$ con $|E| = \Theta(n^2)$, tale che l'eliminazione di un singolo arco comporti la disconnessione temporale di almeno una coppia di nodi coinvolti in una possibile query.
In altre parole, lo scopo è mostrare che, per mantenere la connettività temporale richiesta tra specifiche coppie di vertici, ogni arco del grafo risulta necessario.

4.3.2 Creazione dello spanner

Nei paragrafi seguenti vengono descritte in dettaglio le fasi di costruzione del grafo $G = (V, E, \lambda)$, su cui verrà poi applicata una procedura ipotetica di sparsificazione per ottenere lo spanner $H = (V, E', \lambda)$.

Fase 1 Sia $n \geq 2$ un numero pari. Si consideri l'insieme di nodi

$$\mathcal{A} = \{a_1, a_2, \ldots, a_n\},$$

che costituisce una *clique* di n nodi.
Come descritto in [2], l'insieme $\mathcal{A}$ viene suddiviso in $n/2$ cammini hamiltoniani disgiunti di lunghezza $n-1$, indicati con P_i per $i = 1, \ldots, n/2$.
A ciascun arco $e \in P_i$ è associato un timestamp $\lambda(e) = i$.
Si indicano infine con a_{2i-1} e a_{2i}, rispettivamente, i nodi iniziale e finale del cammino P_i (Figura 1).

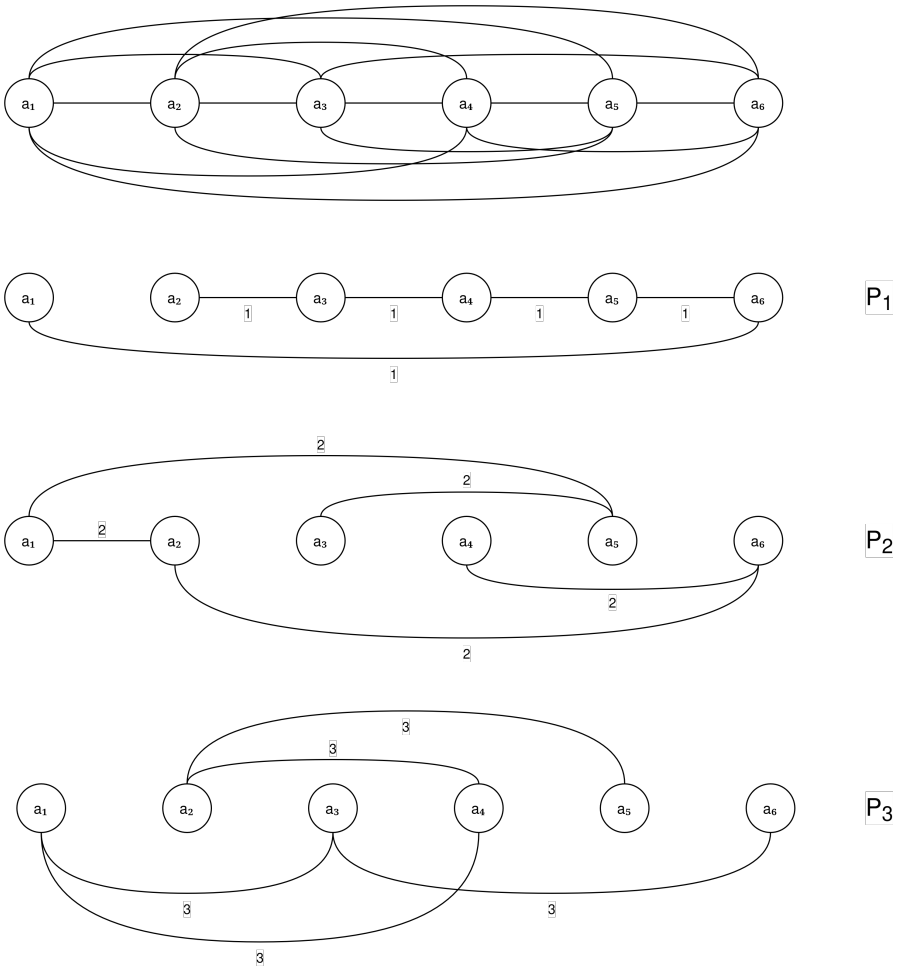

Figura 1: Clique $\mathcal{A}$ con $n = 6$ seguita dai cammini P_1, P_2, P_3

D'ora in avanti, i cammini P_i verranno rappresentati con la notazione illustrata in Figura 2.

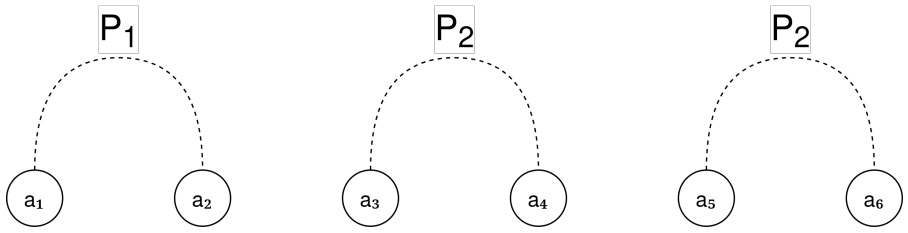

Figura 2: notazioni di P

Fase 2 Sia $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$ un insieme di n nodi.
Per ogni coppia di nodi h_{2i-1}, h_{2i}, si definiscono gli archi con rispettivi nodi a_{2i-1}, a_{2i} di $\mathcal{A}$ come segue:

$$\lambda(h_{2i-1}, a_{2i-1}) = \lambda(h_{2i}, a_{2i}) = i, \quad \forall i \in [n/2].$$

In questo modo, ciascuna coppia h_{2i-1}, h_{2i} è connessa al grafo principale tramite lo stesso timestamp i, riflettendo la struttura temporale dei cammini P_i. (Figura 3)

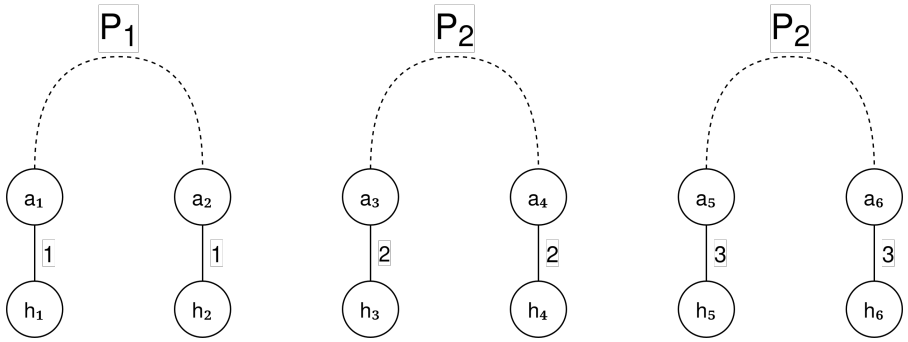

Figura 3: collegamento dei nodi in $\mathcal{H}$ con i nodi in $\mathcal{A}$

Fase 3 Sia $\Delta = \{\alpha_1, \ldots, \alpha_n\}$ un insieme di n nodi.
Per ogni $1 \leq j \leq n$, si collega ciascun nodo α_j al corrispondente nodo $h_j \in \mathcal{H}$ mediante un arco:

$$(\alpha_j, h_j) \in E, \quad \lambda(\alpha_j, h_j) = \varepsilon,$$

dove $\varepsilon > 0$ rappresenta un timestamp minimo positivo(Figura 4).

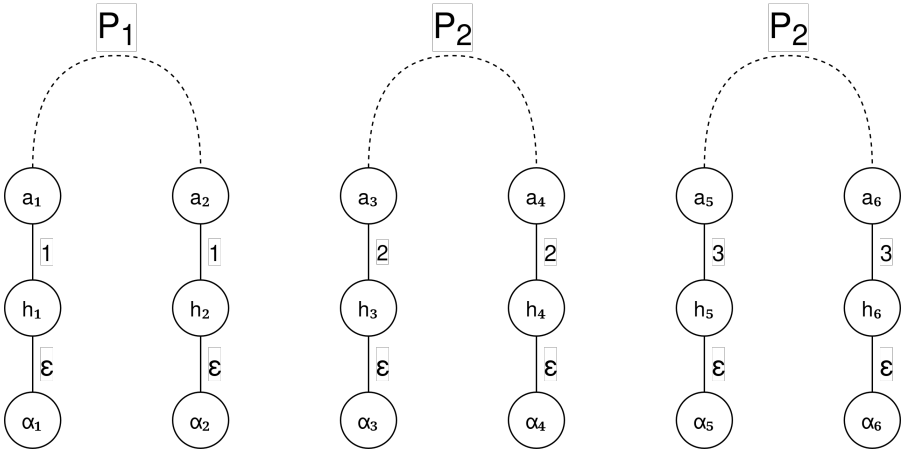

Figura 4: collegamento dei nodi in $\mathcal{H}$ con i nodi in Λ

Fase 4 Sia s il nodo sorgente del grafo.
Questo viene connesso a ciascun nodo $\alpha_j \in \Delta$, per ogni $1 \leq j \leq n$, tramite archi temporalmente precedenti a tutti gli altri:

$$(s, \alpha_j) \in E, \quad \lambda(s, \alpha_j) = 0, \quad \forall 1 \leq j \leq n.$$

Tali connessioni garantiscono che ogni cammino temporalmente valido possa iniziare da s e raggiungere, in un istante successivo, i nodi $\alpha \in \Delta$ (Figura 5).

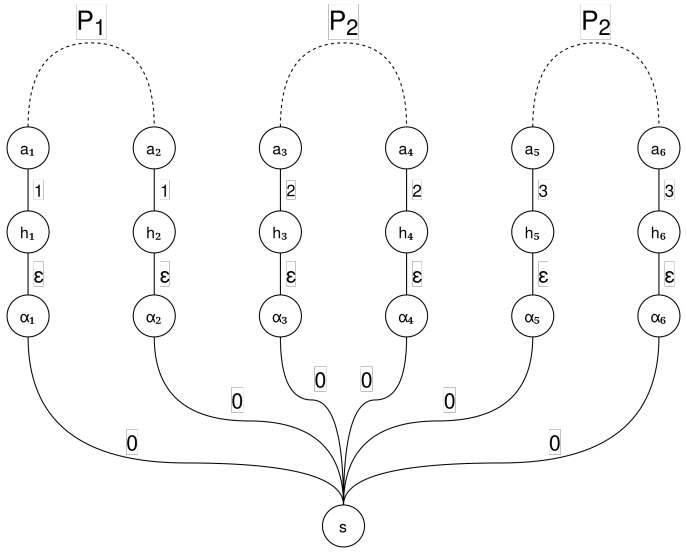

Figura 5: collegamento dei nodi in Δ con il nodo s

Fase 5 Si introduca ora un nuovo insieme $\Omega = \{\omega_1, \ldots, \omega_n\}$ di n nodi.
Ogni nodo ω_j viene connesso al rispettivo nodo $h_j \in \mathcal{H}$, con $1 \leq j \leq n$ tramite un arco avente timestamp maggiore rispetto a tutti quelli finora utilizzati:

$$(\omega_j, h_j) \in E, \quad \lambda(\omega_j, h_j) = n.$$

Questa configurazione assicura che i cammini temporali validi non possano tornare indietro una volta toccati i nodi in Ω (Figura 6).

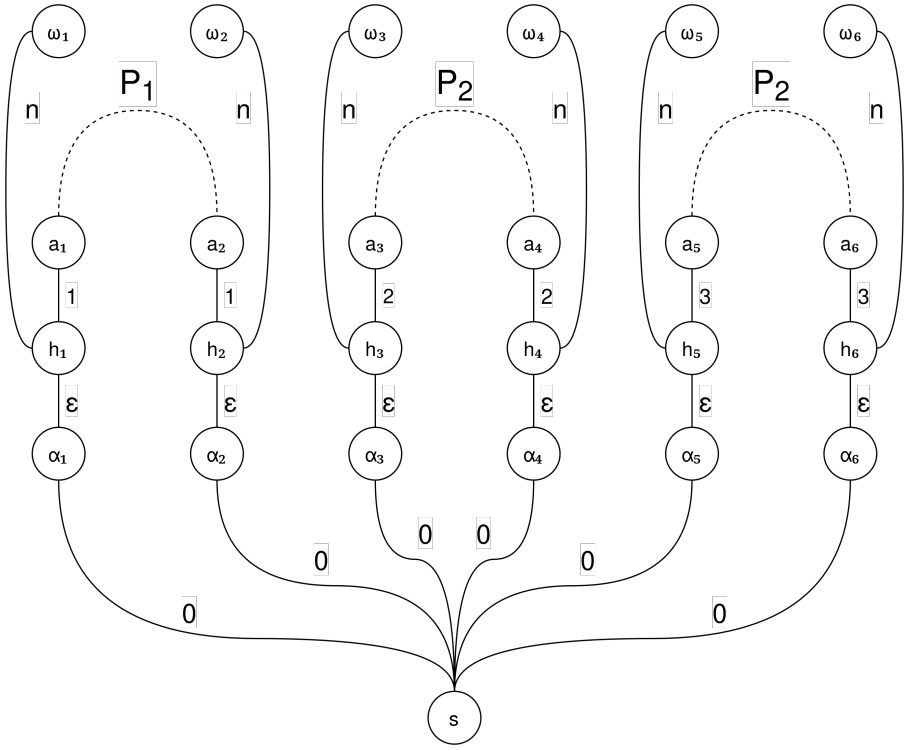

Figura 6: collegamento dei nodi in $\mathcal{H}$ con i nodi in Ω

Fase 6 Infine, si definisca t come nodo di destinazione.
Ogni nodo $\omega_j \in \Omega$, con $1 \leq j \leq n$,viene connesso a t mediante un arco con timestamp più alto rispetto ad ogni altro all'interno del grafo:

$$(\omega_j, t) \in E, \quad \lambda(\omega_j, t) = n + 1, \quad \forall 1 \leq j \leq n.$$

In questo modo, il nodo t rappresenta il punto terminale naturale di ogni cammino temporalmente valido, completando la struttura del grafo (Figura 7).

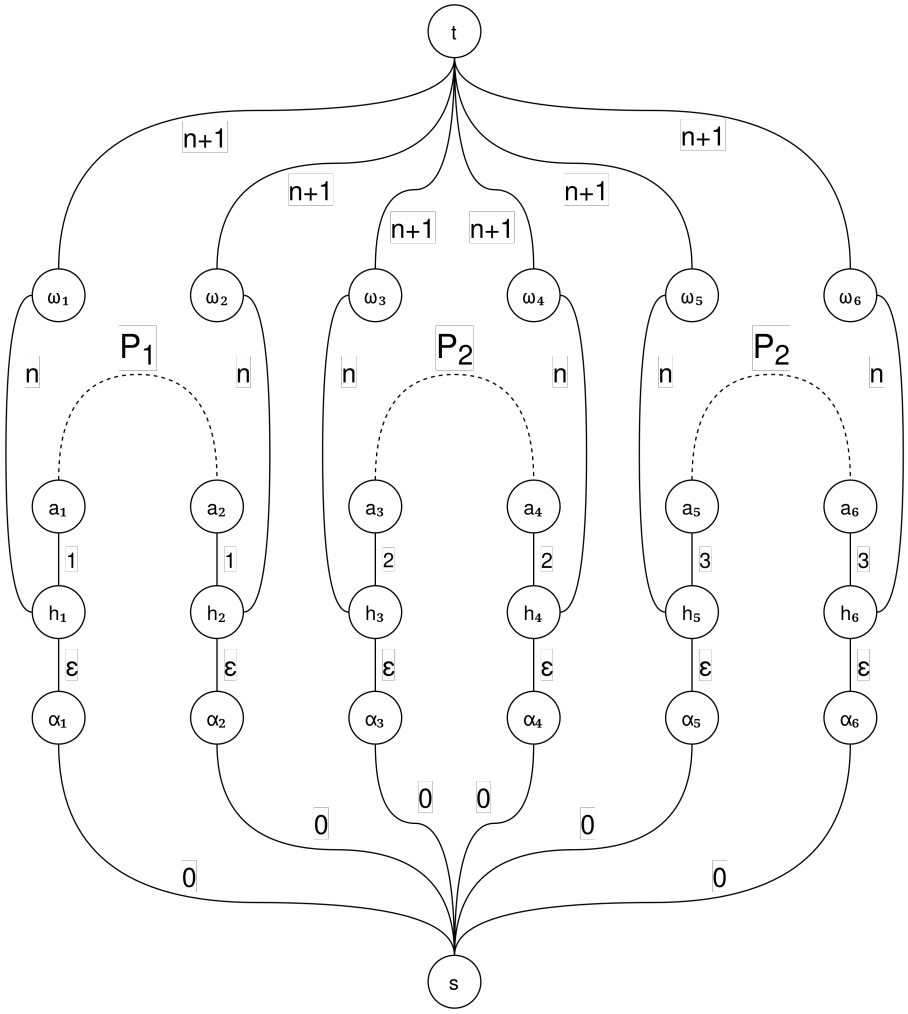

Figura 7: collegamento dei nodi in Ω con il nodo t

Lemma 2. *Se viene eliminato un arco $e \in P_i$, allora non esiste alcun cammino temporale valido fra h_{2i-1} e h_{2i} per ogni $i \in [n/2]$.*

Dimostrazione. Poiché ciascun cammino P_i rappresenta l'unico percorso temporalmente valido che collega la coppia di nodi (h_{2i-1}, h_{2i}), l'eliminazione di un arco $e \in P_i$ interrompe tale connessione.

Non esistendo archi alternativi con timestamp compatibili, nessun altro cammino $\pi(h_{2i-1}, h_{2i})$ può soddisfare la condizione di non decrescenza dei timestamps, e dunque non esiste più un percorso temporalmente valido tra i due nodi. □

Conclusione. A seguito della costruzione del grafo $G = (V, E, \lambda)$ descritta nelle fasi precedenti, si osserva che il numero di archi risulta essere $|E| = \Theta(n^2)$, principalmente a causa della densità della sottostruttura indotta dai nodi dell'insieme $\mathcal{A}$.

Teorema 2. *Ogni spanner $H = (V, E', \lambda)$ che risolve il problema del k-waypoint routing con $k \geq 2$ deve contenere un numero di archi $|E'| = \Omega(n^2)$.*

Dimostrazione. Si consideri il grafo temporale $G = (V, E, \lambda)$ costruito come descritto in precedenza.
Si supponga, per assurdo, che esista uno spanner $H = (V, E', \lambda)$ tale che $|E'| = o(n^2)$, ovvero che non contenga almeno un arco $e \in P_i$ per qualche $i \in [n/2]$.

Per il Lemma 2, l'eliminazione di tale arco implica che in H non esista alcun cammino temporale valido tra h_{2i-1} e h_{2i}, mentre tale cammino esiste in G.
Ne segue che H non preserva tutte le connessioni temporalmente valide di G, contraddicendo la definizione stessa di *spanner*, che richiede la conservazione dei cammini temporali tra tutte le coppie di nodi interessate.

Si conclude pertanto che ogni spanner H valido per il problema del k-*waypoint routing* deve necessariamente contenere $\Omega(n^2)$ archi. □

5 k-waypoint routing con finestra temporale

In questo capitolo viene analizzata una variante del problema di *k-waypoint routing* che tiene conto della presenza di una finestra temporale.
Inizialmente, l'attenzione sarà rivolta al caso particolare $k = 1$, per il quale verrà introdotta una struttura dati di tipo oracolo.
Sia $G = (V, E, \lambda)$ un grafo temporale e siano $s, t \in V$ rispettivamente la sorgente e la destinazione.
L'obiettivo è progettare un oracolo $\mathcal{O}$ che, dato un insieme $K = \{x\} \subseteq V$ e un intervallo temporale $[\alpha, \beta]$, consenta di rispondere in modo efficiente alla seguente query:

$$\mathcal{Q}(K, [\alpha, \beta]) = \begin{cases} \text{True} & \exists \pi(s, t \mid x_1, \ldots, x_k) \text{ t.c. } \forall\, \tau_i \in \pi,\ t_i \in [\alpha, \beta], \\ \text{False} & \nexists \pi(s, t \mid x_1, \ldots, x_k) \text{ t.c. } \forall\, \tau_i \in \pi,\ t_i \in [\alpha, \beta]. \end{cases}$$

L'oracolo proposto presenta le seguenti caratteristiche in termini di complessità:

- **tempo di costruzione:** $O(n + M)$;
- **tempo di query:** $O(\log M)$;
- **spazio occupato:** $O(n + M)$.

5.1 Differenze rispetto al problema precedente

L'introduzione di una finestra temporale modifica significativamente la natura del problema.
Nel problema precedente per la creazione dell'oracolo era sufficiente calcolare:

- l'*earliest arrival time* dalla sorgente s verso tutti i nodi $v \in V$;
- il *latest departure time* da ciascun nodo $v \in V$ verso la destinazione t.

Tuttavia, in presenza di una finestra temporale $[\alpha, \beta]$, questo approccio non è più applicabile. Infatti, anche se l'*earliest arrival time* di un nodo $x \in V$ rientra nell'intervallo $[\alpha, \beta]$, non è garantito che il relativo cammino temporale $\pi(s, x)$ sia interamente contenuto nella finestra, poiché potrebbe attraversare archi temporali con timestamp $t < \alpha$. Un'osservazione analoga vale per il *latest departure time*, che potrebbe coinvolgere archi con $t > \beta$.
In altre parole, la finestra temporale limita non solo il tempo di arrivo o partenza, ma l'intero insieme degli archi che compongono il cammino, rendendo necessaria una strategia di analisi più complessa.

5.2 Un'oracolo per il caso $k = 1$

5.2.1 Descrizione generale dell'approccio

Sia $\Theta = \{\theta_1, \theta_2, \ldots, \theta_{\Delta_s}\}$ l'insieme disgiunto ordinato in modo crescente dei timestamp associati agli archi incidenti con la sorgente s, con $\Delta_s = |\Theta|$.
Analogamente, indichiamo con $\Phi = \{\phi_1, \phi_2, \ldots, \phi_{\Delta_t}\}$ l'insieme disgiunto dei timestamp associati agli archi incidenti con la destinazione t ordinato in modo decrescente, con $\Delta_t = |\Phi|$.
L'intuizione alla base dell'approccio consiste in:

- calcolare tutti gli *earliest arrival time* partendo da s con tempo $t' \geq \theta_i$, $\forall \theta_i \in \Theta$, verso tutti i nodi del grafo.
- calcolare tutti i *latest departure time* arrivando a t con tempo $t' \leq \phi_i$, $\forall \phi_i \in \Phi$, partendo da tutti i nodi del grafo.

In altri termini, invece di considerare un unico *earliest arrival time* e un unico *latest departure time* per ciascun nodo, si intende analizzare come tali valori varino al modificarsi dei vincoli sui tempi di partenza minimi e di arrivo massimi.

5.2.2 Definizione delle strutture dati

Si definisce una matrice $EA \in \mathbb{N}^{n \times \Delta_s}$, dove:

- ogni colonna corrisponde a un nodo $v_i \in V$;
- ogni riga j è associata a un tempo di partenza $\theta_j \in \Theta$, con $1 \leq j \leq \Delta_s$.

L'elemento $EA[j, v_i]$ rappresenta quindi il tempo minimo di arrivo al nodo v_i partendo dalla sorgente s in un istante $t' \geq \theta_j$:

$$EA[j, v_i] = ea_{\theta_j}[v_i].$$

In modo del tutto analogo, si definisce una seconda matrice $LD \in \mathbb{N}^{n \times \Delta_t}$, utilizzata per memorizzare i *latest departure time*. In questo caso:

- le colonne rappresentano i nodi $v_i \in V$;
- le righe j, con $1 \leq j \leq \Delta_t$, corrispondono ai diversi tempi di arrivo massimi $\phi_j \in \Phi$.

Ciascun elemento $LD[j, v_i]$ indica il tempo massimo di partenza da v_i che consente di raggiungere la destinazione t entro un istante $t' \leq \phi_j$:

$$LD[j, v_i] = ld_{\phi_j}[v_i].$$

In entrambe le strutture dati, qualora un nodo v_i non sia raggiungibile dalla sorgente (o non possa raggiungere la destinazione, nel caso di LD), il valore corrispondente viene impostato a $+\infty$. Questo permette di rappresentare in modo coerente l'assenza di un cammino temporale valido all'interno delle matrici.

5.2.3 Ottimizzazione della struttura dati

L'analisi verrà eseguita solo riguardo la struttura dati EA in quanto la struttura LD è definita in modo speculare, per cui questa verrà data per analoga.
È possibile notare come ogni colonna della matrice EA sia non decrescente. Questa proprietà è formalizzata nel seguente lemma:

Lemma 3. *Sia $EA \in \mathbb{N}^{n \times \Delta_s}$ definita come sopra e sia $x \in V$.*
Se esiste un indice i tale che $EA[i, x]$ è definito (cioè $EA[i, x] \neq +\infty$), allora vale la seguente proprietà di monotonicità:

$$\forall j \leq i \quad EA[j, x] \leq EA[i, x].$$

In altri termini, se il nodo x è raggiungibile partendo dalla sorgente s in un istante $t' \geq \theta_i$, allora risulta raggiungibile anche per ogni tempo di partenza $\theta_j \leq \theta_i$.

Dimostrazione. Procediamo per assurdo. Supponiamo che esista un indice $j \leq i$ tale che:

$$EA[j, x] > EA[i, x].$$

Per definizione di *earliest arrival time*, si ha:

$$EA[j, x] = ea_{\theta_j}[x] = \min_{\pi \in P(s,x)} \{\text{end}(\pi(s, x)) \mid \forall \tau \in \pi,\, t_\tau \geq \theta_j\},$$

e analogamente:

$$EA[i, x] = ea_{\theta_i}[x] = \min_{\pi \in P(s,x)} \{\text{end}(\pi(s, x)) \mid \forall \tau \in \pi,\, t_\tau \geq \theta_i\}.$$

Poiché $\theta_j \leq \theta_i$, l'insieme dei cammini ammissibili per $ea_{\theta_i}[x]$ è un sottoinsieme di quello relativo a $ea_{\theta_j}[x]$, cioè:

$$\{\pi(s, x) \mid t_\tau \geq \theta_i\} \subseteq \{\pi(s, x) \mid t_\tau \geq \theta_j\}.$$

Di conseguenza, per definizione di minimo,

$$ea_{\theta_j}[x] \leq ea_{\theta_i}[x],$$

che contraddice l'ipotesi $EA[j, x] > EA[i, x]$. Pertanto, per ogni $j \leq i$ deve valere $EA[j, x] \leq EA[i, x]$. □

Grazie a questo lemma è possibile effettuare la seguente osservazione:

Osservazione. Sia $v \in V$ e sia θ_i un tempo di partenza dalla sorgente s.
Se non esiste un *earliest arrival path* da s a v che parta in θ_i, allora l'*earliest arrival time* di v assume il valore corrispondente al primo cammino che, partendo a tempo $\theta_j > \theta_i$, raggiunge v.

Alla luce di tale osservazione, è possibile ridefinire la struttura EA per eliminare la ridondanza dei valori.
In particolare, invece di rappresentare EA come una matrice $n \times \Delta_s$, si propone di utilizzare un vettore in cui a ciascun nodo $v \in V$ è associata una pila contenente le sole coppie $(\theta_i, ea_{\theta_i}[v])$ tali che:

$$\forall j < i \quad ea_{\theta_j} \neq ea_{\theta_i}.$$

Verà dimostrato che una struttura così definita presenta una complessità spaziale pari a $O(n + M)$.
L'idea chiave è che ciascun arco temporale $\tau \in Stream(G)$ può contribuire, al più, alla creazione di una singola nuova entrata nella struttura.

Lemma 4. *Sia $G = (V, E, \lambda)$ un grafo temporale definito come sopra;*
sia $s \in V$ un nodo sorgente;
sia EA la struttura definita come vettore di pile, come precedentemente descritto.

$$\forall v \in V, \quad |EA[v]| \leq t_{deg}(v)$$

dove $t_{deg}(v)$ indica il temporal degree, grado temporale di v, ovvero il numero di archi temporali a lui incidenti.
In altre parole, si sta indicando che la pila relativa a v ha dimensione al più $t_{deg}(v)$.

Dimostrazione. Ogni elemento nella pila di v è associato ad un diverso tempo di arrivo in v.
Visto che il tempo di arrivo coincide con l'etichetta temporale dell'ultimo arco temporale del cammino che entra in v, allora il numero di elementi della pila deve essere al più $t_deg(v)$.

□

Come conseguenza del Lemma, si ha che:

$$\sum_{v \in V} |EA[v]| \leq \sum_{v \in V} t_{\deg}(v) = 2M = O(M).$$

5.2.4 Procedura per la creazione della struttura dati

Viene ora presentata una procedura per la costruzione della struttura dati EA in tempo $O(n + M)$.
Per semplicità di descrizione, si assume che gli archi del grafo siano *diretti*; nel caso di un grafo non orientato, è sufficiente considerare ciascun arco (u, v, t) come doppio, ossia rappresentato sia come (u, v, t) che come (v, u, t).

In input, l'algoritmo riceve lo *stream temporale* del grafo $G = (V, E, \lambda)$ e il nodo sorgente s.
Si inizializza il vettore EA, definito come in precedenza, ponendo per ogni nodo $v \in V \setminus \{s\}$ la pila iniziale

$$EA[v] \leftarrow [(0, +\infty)],$$

mentre per il nodo sorgente si imposta

$$EA[s] \leftarrow [(0, 0)].$$

Successivamente, si scorre lo *stream* degli archi temporali diretti $\tau = (u, v, t) \in Stream(G)$ in ordine crescente di timestamp t.
Nel caso in cui l'arco sia uscente da s, si estrae dalla cima della pila $EA[s]$ il tempo di partenza più recente $\theta_i \in \Theta$. Se $\theta_i < t$, viene aggiunta alla pila la coppia (t, t), poiché per definizione di Θ risulta $t = \theta_{i+1}$.

Per ciascun arco $\tau = (u, v, t)$, si estraggono quindi le coppie $(\theta_i, ea_{\theta_i}[u])$ e $(\theta_j, ea_{\theta_j}[v])$ dalle cime delle rispettive pile $EA[u]$ e $EA[v]$.
Si confrontano quindi i due tempi di partenza:

- se $\theta_i > \theta_j$ e, inoltre, $ea_{\theta_i}[u] \leq t$, allora l'arco τ può contribuire alla creazione di un nuovo *earliest arrival path* da s a v; in tal caso si aggiunge alla pila $EA[v]$ la coppia (θ_i, t);

- se $\theta_i = \theta_j$ e, inoltre, $ea_{\theta_i}[u] \leq t$, allora l'arco τ può migliorare il valore corrente di $ea_{\theta_i}[v]$. In particolare, se risulta $t < ea_{\theta_i}[v]$, la coppia $(\theta_i, ea_{\theta_i}[v])$ viene rimossa dalla pila $EA[v]$ e sostituita con (θ_i, t).

Algorithm 8: *EA optimized creation algorithm*

Input: Un grafo temporale $G = (V, E, \lambda)$, lo stream temporale di G $Stream(G)$

Output: Una vettore EA di n entrature composte da pile

```
1  Viene calcolato Θ come l'insieme definito come sopra;
2  Δ_s ← |Θ|;
3  Inizializza il EA nel seguente modo;
4      EA[s].push((0,0)) ;            // inizializzazione della pila di s
5      EA[v].push((0,+∞)) ;           // inizializzazione della pila di v
6  foreach arco τ = (u,v,t) ∈ Stream(G) do
7      θ_i, ea_u ← EA[u].top() ;          // tupla in cima alla pila di u
8      θ_j, ea_v ← EA[v].top() ;          // tupla in cima alla pila di v
9      if u = s  ∧  ea_u < t then
10         EA[s].push((t,t));
11     if θ_i > θ_j then
12         EA[v].push((θ_i,t));
13     else if θ_i = θ_j then
14         if ea_u ≤ t ≤ ea_v then
15             EA[v].pop() ;          // rimozione della tupla in cima
16             EA[v].push((θ_i,t));
17 return EA;
```

5.2.5 Correttezza della procedura

Si dimostra ora che la procedura costruisce correttamente la struttura EA, ossia che per ogni nodo $v \in V$ e per ogni tempo di partenza $\theta_i \in \Theta$, la coppia $(\theta_i, ea_{\theta_i}[v])$ memorizzata in $EA[v]$ rappresenta il corretto *earliest arrival time* da s a v.

Idea della dimostrazione. La procedura elabora gli archi temporali in ordine crescente di timestamp. Ad ogni passo, le pile $EA[v]$ contengono le coppie $(\theta_i, ea_{\theta_i}[v])$ già calcolate in modo corretto, ordinate per tempo di partenza e tali che i valori di arrivo siano non decrescenti. Si mostra per induzione sugli archi esaminati dello Stream che tale proprietà si mantiene dopo l'elaborazione di ciascun arco.

Passo base. Per $k = 0$ (nessun arco elaborato) si ha $EA[s] = [(0, 0)]$ e $EA[v] = [(0, +\infty)]$ per ogni $v \neq s$, il che è coerente con la definizione: nessun nodo è raggiungibile da s, salvo s stesso.

Passo induttivo. Si assuma che, dopo aver elaborato i primi k archi, tutte le pile $EA^k[v]$ siano corrette. Consideriamo ora l'arco (u, v, t), con t successivo nel flusso temporale. Siano $(\theta_i, ea_{\theta_i}[u])$ e $(\theta_j, ea_{\theta_j}[v])$ le coppie in cima alle rispettive pile.

- **Caso $\theta_i > \theta_j$.** Se $ea_{\theta_i}[u] \leq t$, esiste un nuovo *earliest arrival path* da s a v che parte in θ_i e arriva in t; si aggiunge quindi la coppia (θ_i, t) a $EA[v]$.
  L'ordine temporale dello stream garantisce che non esistono arrivi più precoci per la stessa partenza, quindi la correttezza è preservata.

- **Caso $\theta_i = \theta_j$.** Se $ea_{\theta_i}[u] \leq t$ e $t < ea_{\theta_i}[v]$, l'arco migliora l'arrivo minimo per la stessa partenza θ_i; la coppia corrispondente in $EA[v]$ viene aggiornata a (θ_i, t), mantenendo la monotonicità.

- **Caso $\theta_i < \theta_j$.** In questo caso esiste già un arrivo noto $ea_{\theta_j}[v]$ con partenza più recente.
  Per il Lemma 3, $ea_{\theta_i}[v] \geq ea_{\theta_j}[v]$; quindi aggiungere (θ_i, t) non produrrebbe un valore utile e la struttura resta invariata.

In ciascun caso, la proprietà di correttezza e monotonicità di EA è mantenuta. Per induzione, essa vale per ogni passo dell'elaborazione e quindi per l'intero stream.

Conclusione. Al termine della procedura, per ogni $v \in V$ e $\theta_i \in \Theta$, la pila $EA[v]$ contiene tutte e sole le coppie $(\theta_i, ea_{\theta_i}[v])$ che rappresentano correttamente gli *earliest arrival times* raggiungibili da s.
La procedura è dunque corretta.

5.2.6 Analisi della complessità

Analizzando lo pseudocodice riportato in precedenza, è possibile stimare la complessità di ciascun blocco dell'algoritmo come segue:

- il calcolo dell'insieme Θ (riga 1) richiede $\Delta_s = O(M)$ operazioni;

- l'inizializzazione del vettore EA (riga 3) ha costo $O(n)$, poiché viene creata un'entrata per ogni nodo del grafo;

- il ciclo principale (riga 6) scorre tutti gli archi dello stream, per un totale di $M = |Stream(G)|$ elementi, ed essendo tutte le operazioni al suo interno costanti, ha un costo totale di $O(M)$

Ne consegue che la complessità temporale totale dell'algoritmo è:

$$O(n + 2M) = O(n + M).$$

Lo spazio complessivo richiesto è, come mostrato all'interno del Lemma 4, anch'esso dell'ordine di:

$$O(n + M).$$

5.2.7 Creazione della struttura dati LD

Questa struttura è del tutto analoga alla struttura dati EA, con la sola differenza che lo stream viene attraversato in ordine decrescente e la logica dei controlli risulta invertita rispetto all'algoritmo precedente.

Algorithm 9: *Algoritmo ottimizzato per la creazione di LD*

Input: Un grafo temporale $G = (V, E, \lambda)$ e lo stream temporale $Stream(G)$ in ordine inverso

Output: Un vettore LD di n elementi, ciascuno composto da una pila

```
1  Calcola Φ come l'insieme definito in precedenza;
2  Δ_t ← |Φ|;
3  Inizializza LD nel seguente modo:;
4      foreach v ∈ V do
5          LD[t].push((+∞, −∞)) ;    // inizializzazione della pila di t
6          LD[v].push((0, −∞)) ;     // inizializzazione della pila di v
7  foreach arco τ = (u, v, t) ∈ Stream(G) do
8      φ_i, ld_u ← LD[u].top() ;          // tupla in cima alla pila di u
9      φ_j, ld_v ← LD[v].top() ;          // tupla in cima alla pila di v
10     if u = t  ∧  ld_u > t then
11         LD[t].push((t, t));
12     if φ_i < φ_j then
13         LD[v].push((φ_i, t));
14     else if φ_i = φ_j then
15         if ld_v ≤ t ≤ ld_u then
16             LD[v].pop() ;              // rimozione della tupla in cima
17             LD[v].push((φ_i, t));
```

5.2.8 Esecuzione della query

L'esecuzione della query $\mathcal{Q}(K, [\alpha, \beta])$, con $K = \{x\} \subseteq V$ e $\alpha, \beta \in \mathbb{N}$, avviene in tre fasi distinte e sfrutta i valori precomputati contenuti all'interno dell'oracolo $\mathcal{O}$, costituito dall'unione delle strutture dati EA e LD.

Fase 1 – Ricerca del tempo di partenza In questa prima fase si ricerca, all'interno della colonna di x nella struttura EA, il primo tempo di partenza θ_α tale che $\theta_\alpha \geq \alpha$. Questo viene realizzato tramite una *binary search*, poiché — come garantito dal Lemma 3 — la struttura EA è ordinata verticalmente in senso crescente rispetto ai tempi di partenza. Una volta individuato θ_α, si estrae il corrispondente valore $ea_{\theta_\alpha}[x]$, che rappresenta il tempo minimo di arrivo al nodo x partendo da s in un tempo $\geq \alpha$.

Fase 2 – Ricerca del tempo di arrivo Analogamente alla fase precedente, si effettua una *binary search* sulla colonna di x nella struttura LD per individuare il primo tempo di partenza ϕ_β tale che $\phi_\beta \leq \beta$. La struttura LD, ordinata verticalmente in senso decrescente (sempre per il Lemma 3), consente di determinare in modo efficiente il valore $ld_{\phi_\beta}[x]$, ossia il tempo più recente in cui è possibile lasciare x per raggiungere t entro l'istante β.

Fase 3 – Verifica di consistenza Infine, si verifica la seguente condizione:

$$ea_{\theta_\alpha}[x] \leq ld_{\phi_\beta}[x].$$

Se tale relazione risulta vera, significa che esiste un cammino $\pi(s, t \mid x)$ che attraversa il nodo x utilizzando soltanto archi con timestamp appartenenti all'intervallo $[\alpha, \beta]$. In tal caso, la query restituisce `True`; in caso contrario, restituisce `False`.

Algorithm 10: *window query algorithm*

Input: Le strutture dati EA e LD, un'intervallo $[\alpha, \beta]$ e un nodo x
Output: Un valore booleano indicante l'esistenza di un cammino $\pi(s, t \mid x)$ attraversando solo archi con $t \in [\alpha, \beta]$

```
1 ea_x ← BinarySearch su EA[x] per trovare la prima tupla con θ_α ≥ α;
2 ld_x ← BinarySearch su LD[x] per trovare la prima tupla con φ_β ≤ β;
3 if ea_x ≤ ld_x then
4     return True;
5 else
6     return False;
```

5.3 Analisi dell'algoritmo per la risoluzione della query

L'algoritmo di esecuzione della query, illustrato nella sezione precedente, presenta una complessità computazionale determinata principalmente dalle due operazioni di ricerca binaria all'interno delle strutture dati EA e LD.
In particolare:

- la ricerca binaria effettuata sulla colonna di x in EA per individuare il valore $ea_{\theta_\alpha}[x]$ ha un costo pari a $O(\log \Delta_s) = O(\log M)$, dove Δ_s rappresenta il numero di tempi di partenza distinti da s;
- la ricerca binaria eseguita su LD per il calcolo di $ld_{\phi_\beta}[x]$ ha costo $O(\log \Delta_t) = O(\log M)$, con Δ_t indicante il numero di tempi di arrivo distinti in t;
- il controllo finale della condizione $ea_{\theta_\alpha}[x] \leq ld_{\phi_\beta}[x]$ richiede tempo costante $O(1)$.

Combinando questi contributi, la complessità complessiva della procedura di query risulta pari a:

$$O(\log M).$$

5.4 Conclusioni

Ricapitolando, la procedura sviluppata per la risoluzione del problema di *k-waypoint routing* con finestra temporale, nel caso particolare $k = 1$ e mediante oracolo, presenta le seguenti caratteristiche di complessità:

- **Tempo di costruzione dell'oracolo**: $O(n+M)$, dovuto alla generazione delle strutture dati EA e LD a partire dallo stream temporale;
- **Tempo di risposta alla query**: $O(\log M)$, come dimostrato nell'analisi precedente;
- **Complessità spaziale**: $O(n + M)$, corrispondente allo spazio necessario per mantenere le due strutture EA e LD, che nel caso peggiore possono contenere un valore per ciascun arco dello stream come dimostrato nel Lemma 4.

Tali risultati dimostrano l'efficienza dell'approccio proposto, che consente di ridurre in modo significativo il tempo di esecuzione delle query grazie all'utilizzo dell'oracolo precomputato, mantenendo al contempo una complessità spaziale lineare rispetto alla dimensione del grafo temporale.

5.5 Applicazione del lower bound per $k \geq 2$ e spanner

È facile osservare come il lower bound dimostrato per il problema del *k-waypoint routing* resti valido anche nel caso del problema con finestra temporale.
Infatti, ponendo la finestra temporale $[\alpha, \beta]$ tale che:

$$\alpha = \min\{t \in \lambda(e) \mid e \in E\}, \quad \beta = \max\{t \in \lambda(e) \mid e \in E\},$$

si ottiene una finestra che copre l'intero intervallo temporale in cui il grafo è definito. In questo modo, ogni cammino temporalmente valido presente nella costruzione del lower bound originale rimane invariato, e nessun arco o percorso viene escluso dalla finestra.

Di conseguenza, la stessa argomentazione che porta alla necessità di $\Omega(n^2)$ archi per mantenere la connettività temporale continua ad applicarsi anche in presenza di vincoli temporali, purché la finestra considerata sia quella massimale $[\alpha, \beta]$.

Riferimenti bibliografici

[1] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *Proc. VLDB Endow.*, vol. 7, p. 721–732, May 2014.

[2] K. Axiotis and D. Fotakis, "On the size and the approximability of minimum temporally connected subgraphs," in *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy* (I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, and D. Sangiorgi, eds.), vol. 55 of *LIPIcs*, pp. 149:1–149:14, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

Ringraziamenti

Innanzitutto ringrazio Dora, per esserci stata sempre, per avermi tirato su di morale anche quando il suo era più a terra del mio.
La ringrazio per aver sempre ascoltato tutte le mie inutili lamentele e paranoie, per essermi stata affianco ogni singolo giorno da sei anni ad oggi, senza i quali non sarei la persona che sono. Ti amo, dal profondo del mio cuore.

Ringrazio poi Ionut, che, dopo tutte le lezioni, gli esami superati e le risate, ho imparato a volergli bene come fosse un fratello.

Ringrazio Franco, che, nonostante fosse sempre impegnato, mi è sempre stato ad ascoltare e mi ha corretto la tesi come se fosse sua.

Vorrei poi ringraziare Davide, Christian, Pietro, Ivan, Alex, Dimu, Mirco, Mario, Giorgia, Francesca, Alessio, Domenico, Vittorio, Giacomino e Adriano.
È grazie a voi se sono riuscito a sopportare tutti quei pomeriggi di studio, facendo diventare, fra una pausa caffè e l'altra, quelle quattro mura fredde che sono il laboratorio un luogo che, in futuro, potrò ricordare come se fosse casa.

Ringrazio Matteo, per essere stato una presenza fondamentale nella mia vita da quando ho memoria.

Ringrazio Leonardo, Giorgia e Jacopo, grazie ai quali ho passato tante serate in compagnia.

Ringrazio Morgan, Gabriele e Francesco, che, anche se non parliamo più spesso come un tempo, le nostre chiacchierate sono e sempre saranno piacevoli.

Ringrazio infine la mia famiglia, per avermi supportato e sopportato durante tutti questi lunghi anni.