

# Round-Based Public Transit Routing

Un algoritmo di ottimizzazione per il trasporto multimodale

Gianluca Covini

Università di Pavia

January 9, 2026

# Indice

- 1 Introduzione
- 2 Problema
- 3 Soluzioni esistenti
- 4 RAPTOR
- 5 Miglioramenti
- 6 Estensioni
- 7 Risultati e conclusioni

# Introduzione

Il problema presentato è quello del **miglior percorso nelle reti di trasporto pubblico**, andando a considerare in particolare due criteri di ottimizzazione: tempo di arrivo e numero di trasferimenti.

L'obiettivo della presentazione è formalizzare il problema, introdurre gli algoritmi esistenti, spesso varianti dell'algoritmo di Dijkstra e di presentare **RAPTOR**, Round-bAsed Public Transit Optimized Router, un algoritmo più efficiente delle soluzioni esistenti e non basato sull'algoritmo di Dijkstra.

# Dati

## Timetable

I dati del problema sono riassunti nel vettore

$$(\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$$

- $\Pi$ , periodo operativo (come secondi di una giornata);
- $\mathcal{S}$ , insieme di fermate;
- $\mathcal{T}$ , insieme di viaggi;
- $\mathcal{R}$ , insieme di itinerari;
- $\mathcal{F}$ , insieme di tracciati pedonali.

Il vettore prende il nome di **timetable**.

# Dati

- Un viaggio  $t \in \mathcal{T}$  è una sequenza di fermate di uno specifico veicolo lungo una linea;
- Un itinerario  $r \in \mathcal{R}$  è un insieme di viaggi che condivide la stessa sequenza di fermate;
- Un tracciato pedonale  $f \in \mathcal{F}$  è un percorso a piedi tra due fermate  $p_1$  e  $p_2$ , cui associo un tempo di percorrenza  $l(p_1, p_2)$ .

# Dati

- Un viaggio  $t \in \mathcal{T}$  è una sequenza di fermate di uno specifico veicolo lungo una linea;
- Un itinerario  $r \in \mathcal{R}$  è un insieme di viaggi che condivide la stessa sequenza di fermate;
- Un tracciato pedonale  $f \in \mathcal{F}$  è un percorso a piedi tra due fermate  $p_1$  e  $p_2$ , cui associo un tempo di percorrenza  $l(p_1, p_2)$ .

## Tempo di arrivo e di partenza

Dato un viaggio  $t \in \mathcal{T}$ , formato da  $p \in \mathcal{S}$  fermate associo un tempo di arrivo  $\tau_{arr}(t, p)$  e un tempo di ripartenza  $\tau_{dep}(t, p)$ . Ovviamente varrà che:

$$\tau_{arr}(t, p) \leq \tau_{dep}(t, p)$$

# Obiettivo

L'**obiettivo** è produrre un insieme di **percorsi**  $\mathcal{J}$ .

# Obiettivo

L'**obiettivo** è produrre un insieme di **percorsi**  $\mathcal{J}$ .

## Percorso

Un **percorso**  $j \in \mathcal{J}$  è una sequenza di viaggi  $t \in \mathcal{T}$  e tracciati  $f \in \mathcal{F}$  che va da una fermata iniziale  $p_s$  a una fermata finale  $p_t$ . Notiamo che un percorso che ha  $k$  viaggi avrà esattamente  $k - 1$  trasferimenti.

A un percorso sono associati dei **criteri di ottimizzazione**.

# Obiettivo

L'**obiettivo** è produrre un insieme di **percorsi**  $\mathcal{J}$ .

## Percorso

Un **percorso**  $j \in \mathcal{J}$  è una sequenza di viaggi  $t \in \mathcal{T}$  e tracciati  $f \in \mathcal{F}$  che va da una fermata iniziale  $p_s$  a una fermata finale  $p_t$ . Notiamo che un percorso che ha  $k$  viaggi avrà esattamente  $k - 1$  trasferimenti.

A un percorso sono associati dei **criteri di ottimizzazione**.

## Dominanza

Dati due percorsi  $j_1$  e  $j_2$  in  $\mathcal{J}$ . Diciamo che  $j_1$  **domina**  $j_2$  ( $j_1 \preceq j_2$ ) se  $j_1$  non è peggiore di  $j_2$  in nessun criterio.

# Obiettivo

## Fronte di Pareto

Un **fronte di Pareto** è un insieme di percorsi a due a due non dominati.

# Obiettivo

## Fronte di Pareto

Un **fronte di Pareto** è un insieme di percorsi a due a due non dominati.

## Etichetta

Definiamo **etichetta(label)** un viaggio intermedio.

# Problemi

Possiamo considerare vari problemi, tra cui:

- *Earliest Arrival Problem*: date  $p_s$ ,  $p_t$  e  $\tau \in \Pi$ , cerca un percorso  $j \in \mathcal{J}$  che parta da  $p_s$  non prima di  $\tau$  e arrivi in  $p_t$  il prima possibile.

# Problemi

Possiamo considerare vari problemi, tra cui:

- *Earliest Arrival Problem*: date  $p_s$ ,  $p_t$  e  $\tau \in \Pi$ , cerca un percorso  $j \in \mathcal{J}$  che parta da  $p_s$  non prima di  $\tau$  e arrivi in  $p_t$  il prima possibile.
- *Multi-Criteria Problem*: è una generalizzazione che ottimizza anche altri criteri e cerca un fronte di Pareto.

# Problemi

Possiamo considerare vari problemi, tra cui:

- *Earliest Arrival Problem*: date  $p_s$ ,  $p_t$  e  $\tau \in \Pi$ , cerca un percorso  $j \in \mathcal{J}$  che parta da  $p_s$  non prima di  $\tau$  e arrivi in  $p_t$  il prima possibile.
- *Multi-Criteria Problem*: è una generalizzazione che ottimizza anche altri criteri e cerca un fronte di Pareto.
- *Range Problem*: date  $p_s$ ,  $p_t$  e  $\tau \in \Pi$ , cerca un percorso  $j \in \mathcal{J}$  che parta da  $p_s$  non più tardi di  $\tau$  e arrivi in  $p_t$  il prima possibile.

# Struttura di grafo

Problemi di questo tipo vengono formalizzati con strutture a grafi.  
Modellizziamo la rete nel seguente modo:

- Creiamo un **nodo-fermata** per ogni fermata  $p \in \mathcal{S}$ . Inoltre a ogni fermata  $p$  e itinerario  $r \in \mathcal{R}$  passante per essa associamo un **nodo-itinerario**  $r_p$ .

# Struttura di grafo

Problemi di questo tipo vengono formalizzati con strutture a grafi. Modellizziamo la rete nel seguente modo:

- Creiamo un **nodo-fermata** per ogni fermata  $p \in \mathcal{S}$ . Inoltre a ogni fermata  $p$  e itinerario  $r \in \mathcal{R}$  passante per essa associamo un **nodo-itinerario**  $r_p$ .
- All'interno di ogni fermata aggiungiamo degli **archi** (non orientati) tra il nodo-fermata e i nodi-itinerario corrispondenti, per permettere i trasferimenti. Il loro peso (costante) è dato dal tempo di trasferimento tra i viaggi che toccano  $p$ .

# Struttura di grafo

Problemi di questo tipo vengono formalizzati con strutture a grafi.  
Modellizziamo la rete nel seguente modo:

- Modellizziamo i viaggi come archi dipendenti dal tempo tra due nodi-itinerari: se un viaggio  $t \in \mathcal{T}$  porta da  $p_1$  a  $p_2$  lungo  $r$ , allora lo modellizziamo come un arco tra  $r_{p_1}$  e  $r_{p_2}$ . Il loro peso sarà funzione del tempo di viaggio  $\tau_{arr}(t, p_2) - \tau_{dep}(t, p_1)$ .

# Struttura di grafo

Problemi di questo tipo vengono formalizzati con strutture a grafi.  
Modellizziamo la rete nel seguente modo:

- Modellizziamo i viaggi come archi dipendenti dal tempo tra due nodi-itinerari: se un viaggio  $t \in \mathcal{T}$  porta da  $p_1$  a  $p_2$  lungo  $r$ , allora lo modellizziamo come un arco tra  $r_{p_1}$  e  $r_{p_2}$ . Il loro peso sarà funzione del tempo di viaggio  $\tau_{arr}(t, p_2) - \tau_{dep}(t, p_1)$ .
- I trasferimenti pedonali li modellizziamo come archi tra i nodi-fermata corrispondenti con peso  $I(p_1, p_2)$ .

# Algoritmi

Vista l'analogia con i network flow problems, possiamo costruire degli algoritmi risolutivi simili.

In particolare l'Earliest Arrival Problem si può risolvere con una variante dell'algoritmo di Dijkstra, che prende il nome di **Time-Dijkstra** (TD).

## Time-Dijkstra

L'algoritmo definisce una struttura di nodi finiti  $\mathcal{FN}^c$ . L'algoritmo rimuove uno alla volta i nodi da  $\mathcal{FN}^c$  per tempo di arrivo crescente, valutando ogni arco  $e = (u, v)$  al tempo d'arrivo in  $u$  e aggiornando di volta in volta l'etichetta. L'algoritmo si ferma quando viene analizzato il nodo d'arrivo  $p_t$ .

# Algoritmi

Il Multi-Criteria Problem si può risolvere con un algoritmo *label-correcting* che prende il nome di **multi-label-correcting-algorithm** (MLC).

## MLC

Ogni etichetta possiede vari criteri di ottimizzazione. Ogni nodo  $u$ , ora, contiene un insieme  $B_u$  ("bag") che contiene delle etichette non dominate. Procede, poi, come un algoritmo *label-correcting*. A ogni passo, prende l'etichetta minima  $L_u$  e processa il nodo  $u$ . Dopodiché per ogni arco  $(u, v)$  crea una etichetta  $L_v$ : se non è dominata da nessuna etichetta in  $B_v$  la inserisce in  $B_v$ .

# Algoritmi

Quando oltre al tempo d'arrivo l'unico altro criterio di ottimizzazione è il numero di trasferimenti, si usa una variante dell'algoritmo Time-Dijkstra, che prende il nome di **Layered Dijkstra** (LD).

## Layered Dijkstra

Sia  $K$  un limite superiore al numero di trasferimenti. Si costruiscono  $K$  copie del grafo con gli archi relativi ai trasferimenti che vanno da un grafo al successivo. Si applica, poi, l'algoritmo TD. Un percorso che termina sul  $k$ -esimo grafo avrà esattamente  $k$  trasferimenti.

# Algoritmi

Per il *Range Problem*, infine, si può usare l'algoritmo **Self-Pruning Connection-Setting** (SPCS).

## SPCS

Agisce come TD ma con la differenza che vengono eliminate tutte le etichette  $L$  di un vertice  $v$  se questo possiede già un'altra etichetta  $L'$  per cui  $\tau(L') \geq \tau(L)$ .

# RAPTOR

**RAPTOR** è un algoritmo che risolve i problemi per due criteri di ottimizzazione: il tempo di arrivo e il numero di trasferimenti.

# RAPTOR

**RAPTOR** è un algoritmo che risolve i problemi per due criteri di ottimizzazione: il tempo di arrivo e il numero di trasferimenti.

- **Dati:** prendiamo una timetable  $(\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$ , una fermata di partenza e una fermata di arrivo  $p_s \in \mathcal{S}$  e  $p_t \in \mathcal{S}$  e un tempo di partenza  $\tau \in \Pi$ .
- **Obiettivo:** per ogni  $k$  vogliamo trovare un percorso non dominato con tempo di arrivo minimo in  $p_t$ , avente al massimo  $k$  viaggi.

# Algoritmo

L'algoritmo è iterativo: al passo  $k$  si calcola il modo più veloce per arrivare all'arrivo con al più  $k - 1$  trasferimenti (cioè al più  $k$  viaggi).

- Poniamo  $K$  un limite superiore del numero di iterazioni.
- L'algoritmo associa a ogni fermata un vettore  $(\tau_0(p), \dots, \tau_K(p))$ , dove  $\tau_i(p)$  rappresenta il tempo d'arrivo minore in  $p$  con al più  $i$  viaggi.

# Algoritmo

Prima dell'inizio dell'iterazione inizializziamo i valori nel seguente modo:

## Condizioni iniziali

- Inizializziamo tutti i  $\tau_i(p)$  a  $\infty$ ;
- poniamo  $\tau_0(p_s) = \tau$ .

# Algoritmo

Prima dell'inizio dell'iterazione inizializziamo i valori nel seguente modo:

## Condizioni iniziali

- Inizializziamo tutti i  $\tau_i(p)$  a  $\infty$ ;
- poniamo  $\tau_0(p_s) = \tau$ .

All'inizio del passo  $k$  avremo che  $\tau_0(p), \dots, \tau_{k-1}(p)$  sono corrette per ogni  $p \in \mathcal{S}$ , mentre le restanti saranno ancora pari a  $\infty$ . L'obiettivo del passo  $k$  è calcolare  $\tau_k(p)$  per ogni fermata  $p$ .

# Passo $k$

Il passo  $k$  si articola in tre step.

## Primo step

- Per ogni  $p \in \mathcal{S}$  poniamo  $\tau_k(p) = \tau_{k-1}(p)$ , in modo da porre una limitazione superiore per  $\tau_k(p)$ .

# Passo $k$

Il passo  $k$  si articola in tre step.

## Secondo step

- Si analizza ogni itinerario  $r$  esattamente una volta.

Definiamo  $\mathcal{T}(r) = (t_0, \dots, t_{|\mathcal{T}(r)|-1})$ , i viaggi lungo l'itinerario  $r$  dal primo all'ultimo.

Sia, inoltre,  $et(r, p_i)$  il viaggio più veloce, se esiste, in  $r$  tale per cui  $\tau_{dep}(t, p_i) \geq \tau_{k-1}(p_i)$ .

Per analizzare l'itinerario, lo percorriamo finché non troviamo una fermata  $p_i$  per cui  $et(r, p_i)$  è definita. Nel caso chiamiamo *viaggio corrente* il viaggio corrispondente.

Continuiamo il procedimento percorrendo tutte le fermate dell'itinerario  $r$ .

Per ogni fermata per cui è definita possiamo aggiornare  $\tau_k(p)$  usando  $et(r, p)$ .

## Passo $k$

Notiamo che al secondo passo possiamo dover aggiornare il viaggio corrente per  $k$ : a ogni fermata  $p_i$  lungo  $r$  è possibile che ci sia un viaggio più veloce per  $p_i$  perché si è trovato al passo precedente un modo più veloce per arrivare a  $p_i$ . Si verifica, quindi, che  $\tau_{k-1}(p_i) < \tau_{arr}(t, p_i)$  e si aggiorna il viaggio corrente ricalcolando  $et(r, p_i)$ .

# Passo k

Il passo  $k$  si articola in tre step:

## Terzo step

Per ogni tracciato pedonale  $(p_i, p_j) \in \mathcal{F}$  si pone  
 $\tau_k(p_j) = \min \tau_k(p_j), \tau_k(p_i) + l(p_i, p_j)$

# Passo $k$

Il passo  $k$  si articola in tre step:

## Terzo step

Per ogni tracciato pedonale  $(p_i, p_j) \in \mathcal{F}$  si pone  
 $\tau_k(p_j) = \min \tau_k(p_j), \tau_k(p_i) + I(p_i, p_j)$

## Criteri d'arresto

L'algoritmo si ferma dopo il passo  $k$  se non è stato aggiornato nessun valore  $\tau_k(p)$  negli ultimi due step dell'iterazione.

# Complessità computazionale

## Complessità computazionale

L'algoritmo impiega  $\mathcal{O}(K(\sum_{r \in \mathcal{R}} |r| + |\mathcal{T}| + |\mathcal{F}|))$  operazioni per produrre un fronte di Pareto di percorsi, dove  $K$  è il numero di iterazioni.

Infatti, noi percorriamo ogni itinerario al più una volta: per ciascuno di essi osserviamo  $\sum_{r \in \mathcal{R}} |r|$  fermate. Per trovare  $et(r, \cdot)$  guardiamo ogni viaggio di ogni itinerario al più una volta. E infine, nel terzo step, studiamo ogni tracciato pedonale al più una volta.

# Marking

Una prima tecnica di miglioramento prevede di analizzare al passo  $k$  solamente quegli itinerari che contengono almeno una fermata raggiunta con  $k - 1$  viaggi. Quindi, ha senso considerare solamente gli itinerari che hanno avuto un aggiornamento al passo  $k - 1$ .

Lo si implementa segnando al passo  $k - 1$  le fermate  $p$  per cui c'è stato un miglioramento di  $\tau_{k-1}(p)$  e analizzando, poi, al passo  $k$  solo quegli itinerari che contengono una di queste fermate

# Local pruning

Per ogni fermata  $p_i$  fissiamo un valore  $\tau * (p_i)$  che rappresenta il più breve tempo di arrivo a  $p_i$  noto. Al passo  $k$  consideriamo una fermata solo quando il tempo di arrivo con  $k$  viaggi è minore di  $\tau * (p_i)$ .

Notiamo che il local pruning ci permette di evitare il primo step di ogni iterazione.

# Target pruning

Infine, dato che siamo interessati solo ai viaggi che giungono al target stop  $p_t$ , non ha senso segnare tutte le fermate i cui tempi d'arrivo sono maggiori di  $\tau * (p_t)$

# Transfer preferences e dominanza stretta

L'algoritmo MLC può essere esteso al concetto di dominanza stretta.

## Dominanza stretta

Un viaggio  $j_1$  **domina strettamente** un altro viaggio  $j_2$  se è strettamente migliori in almeno un criterio.

La motivazione è quella di introdurre dei criteri di preferenza per le location in cui avvengono i trasferimenti.

# Transfer preferences e dominanza stretta

L'algoritmo MLC può essere esteso al concetto di dominanza stretta.

## Dominanza stretta

Un viaggio  $j_1$  **domina strettamente** un altro viaggio  $j_2$  se è strettamente migliori in almeno un criterio.

La motivazione è quella di introdurre dei criteri di preferenza per le location in cui avvengono i trasferimenti.

È possibile introdurre questa cosa in RAPTOR: quando si prende un nuovo viaggio è sufficiente tenere traccia della fermata preferibile in cui prendere il viaggio.

# McRAPTOR

È possibile estendere RAPTOR per gestire ulteriori criteri di ottimizzazione, introducendo **McRAPTOR**.

McRAPTOR è in grado di gestire, per esempio, l'introduzione di tariffe a zona.

# rRAPTOR

Per risolvere il range problem è possibile costruire l'estensione  
**rRAPTOR**.

# Risultati

Un'analisi sul trasporto pubblico di Londra rivela che RAPTOR performance meglio di LD e MLC che risolvono lo stesso problema, essendo 9 volte più veloce di MLC e 6 volte più veloce di LD.

Risulta che RAPTOR è più veloce, di un fattore 2, anche di TD che risolve, però, un problema più semplice, dal momento che non ottimizza il numero di trasferimenti.

Algorithm	Tr	# Rnd.	# Relax. p. Route	# Visits p. Stop	# Comp. p. Stop	# Jn.	Time [ms]
RAPTOR	●	8.4	3.0	11.1	22.2	1.9	7.3
TD	○	—	—	7.4	7.4	0.9	14.2
LD [7]	●	—	—	17.3	39.5	1.9	44.5
MLC [15]	●	—	—	12.8	28.7	1.9	67.2

# Risultati

Algorithm	R	Tr	Fz	# Rnd.	# Relax. p. Route	# Visits p. Stop	# Comp. p. Stop	# Jn.	Time [ms]
rRAPTOR	●	●	○	138.5	36.6	124.7	346.4	16.3	87.0
McRAPTOR	●	●	○	9.5	3.8	15.1	2062.7	16.3	259.8
McRAPTOR	○	●	●	10.8	4.5	17.9	396.4	9.0	107.4
MLC [15]	○	●	●	—	—	48.1	930.3	9.0	399.5
SPCS [11]	●	○	○	—	—	76.2	76.2	7.8	183.6

Algorithm	R	Tr	Fz	1 core		3 cores		6 cores		12 cores	
				# Comp. p. Stop	Time [ms]						
RAPTOR	○	●	○	21.5	7.7	21.7	5.0	21.8	4.1	21.8	3.7
rRAPTOR	●	●	○	346.4	92.3	357.7	39.5	374.0	26.8	404.6	21.6
McRAPTOR	●	●	○	2098.6	280.2	2101.2	113.1	2098.4	66.1	2098.2	50.1
McRAPTOR	○	●	●	410.0	118.6	410.6	49.4	408.4	29.9	408.3	26.1
SPCS [11]	●	○	○	76.2	183.6	79.9	69.1	85.2	44.9	95.5	38.9

# Risultati

Algorithm	R	Tr	Los Angeles		New York		Chicago	
			# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]
RAPTOR	○	●	24.0	3.4	14.6	3.1	14.4	1.8
RAPTOR-6	○	●	25.0	2.0	14.0	2.0	14.6	1.2
rRAPTOR	●	●	128.0	16.2	159.5	24.3	143.7	14.6
rRAPTOR-6	●	●	141.8	7.1	173.9	9.0	154.3	5.5
LD [7]	○	●	37.5	24.9	25.4	21.8	22.6	13.0
MLC [15]	○	●	21.7	38.1	16.8	32.2	9.8	17.6
SPCS [11]	●	○	28.4	37.9	29.6	53.7	29.3	36.3
SPCS-6 [11]	●	○	33.0	14.8	34.5	15.0	32.6	8.8

# Conclusioni e bibliografia

RAPTOR si rivela un algoritmo più efficiente delle soluzioni esistenti: a differenza di queste non opera su un grafo e non utilizza code di priorità.

## Bibliografia:

- 
- Delling, Pajor, Wernkeck,
- Round-Based Public Transit Routing*
- , 2015.