



UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Ingegneria

Corso di Laurea in
INGEGNERIA INFORMATICA

**Sviluppo di algoritmi per la
determinazione di ipercammini
minimi**

-

**Development of algorithms for
shortest hyperpaths search**

Tesi di Laurea di
Marco Gualtieri
Luglio 2012

Relatore:
Prof. Fabio Schoen

Correlatore:
Ing. Mirko Maischberger

Anno Accademico 2011/2012

Ringraziamenti

Desidero innanzitutto ringraziare il Prof. Fabio Schoen per il tempo dedicato alla mia tesi e per la sincera disponibilità che mi ha sempre dimostrato, permettendomi di giungere alla fine di questo percorso di studi con piena soddisfazione.

Inoltre, ringrazio l'Ing. Mirko Maischberger per il prezioso lavoro di affiancamento e di supporto nella stesura di questo lavoro, e per i numerosi dubbi risolti e consigli dispensati.

La mia sincera gratitudine va ai compagni di corso Maurizio e Tony, con cui ho condiviso gran parte di questo percorso.

Infine, un ringraziamento sentito ai miei familiari ed amici per essermi stati vicino nei momenti più importanti.

Indice

Introduzione	v
1 Ipergraifi e modello della rete	1
1.1 Teoria degli ipergraifi e definizioni	1
1.1.1 Ipergraifi orientati	3
1.1.2 Definizioni	5
1.2 Rete di trasporto	6
1.2.1 Insieme attrattivo	8
1.2.2 Cammini e ipercammini	10
2 Calcolo del costo di un ipercammino	12
2.1 Notazione	12
2.2 Etichettatura degli archi	13
2.2.1 Costo	13
2.2.2 Probabilità	16
2.3 Etichettatura dei cammini	17
2.3.1 Probabilità	17
2.3.2 Costo	17
2.4 Costo degli ipercammini	18
2.4.1 Costo atteso di un sotto-ipercammino	18

2.4.2	Calcolo iterato del costo di un ipercammino	20
2.4.3	Esempio	21
3	Determinazione degli ipercammini minimi	24
3.1	Definizione del problema	24
3.2	Insieme attrattivo ottimale	26
3.2.1	Proprietà	27
3.2.2	La procedura <i>attractive-set</i>	30
3.2.3	Esempio di determinazione dell'insieme attrattivo ottimale	31
3.3	L'algoritmo <i>Shortest Hyperpath Tree</i> (SHT)	33
3.4	L'algoritmo <i>Priority - SHT</i>	36
3.5	Esempio di determinazione dell'iperalbero di costo minimo .	38
3.5.1	Osservazioni	44
4	Implementazione degli algoritmi	45
4.1	Input / Output dei dati	47
4.1.1	Formato di input dell'ipergrafo	47
4.1.2	Parsing del file di input	50
4.1.3	Formato di output dello <i>Shortest HyperPath</i>	51
4.1.4	Generazione del file di output	55
4.2	L'algoritmo sht	58
4.2.1	Implementazione della coda Q	59
4.2.2	Aggiornamento del costo dei nodi	61
4.3	L'algoritmo priority_sht	63
4.3.1	Implementazione della coda Q	63
4.3.2	Aggiornamento del costo dei nodi	65
4.4	Esempio di applicazione	67

5 Un esempio reale	70
5.1 Etichettamento degli archi	73
5.2 Esempi	74
5.2.1 Esempio 1	74
5.2.2 Esempio 2	76
A Uso delle librerie boost	79
A.1 <i>Boost Graph Library</i>	79
A.2 Modellazione dell'ipergrafo	84
A.2.1 Accesso a vertici e archi	86
Conclusioni	88
Bibliografia	90

Introduzione

Il presente lavoro di tesi è incentrato sugli **ipergrafi** orientati, particolari strutture che estendono i grafi orientati (generalizzando il concetto di arco a quello di **iperarco**).

Rispetto a grafi e cammini tradizionali, ipergrafi e **ipercammini** sono caratterizzati da un maggiore potere espressivo che li rende adatti a problemi più complessi.

Essi rappresentano quindi un valido strumento modellistico e algoritmico in vari campi di ricerca: reti di trasporto o di comunicazione, problemi di pianificazione, intelligenza artificiale, logica proposizionale...

Ad esempio, nel contesto della pianificazione industriale, le linee di assemblaggio possono essere modellate tramite ipergrafi orientati aciclici, in cui ciascun iperarco rappresenta un'operazione che lega due fasi produttive e ciascun ipercammino corrisponde ad un particolare piano di assemblaggio.

Questa tesi si concentra sulla modellazione del trasporto pubblico locale tramite ipergrafi.

Trattandosi di un problema non ancora ben approfondito e documentato nella letteratura scientifica, una prima fase del lavoro si è basata sullo studio teorico del problema e su una sua formalizzazione.

Successivamente sono stati definiti e implementati due algoritmi per la determinazione di ipercammini di costo minimo su tali ipergrafi.

Organizzazione del lavoro

Il lavoro presentato si compone di cinque capitoli e un'appendice.

Capitolo 1 : fornisce un'introduzione generale alla teoria degli ipergrafo e introduce le definizioni basilari riguardo agli ipergrafo orientati.

Viene quindi descritto il modello di ipergrafo orientato utilizzato per rappresentare la rete di trasporto, presentando i vari tipi di nodi e archi e illustrando la struttura in cui sono connessi.

Insieme a questo, viene fornita anche una prima descrizione dell'*insieme attrattivo* che caratterizza il problema del trasporto pubblico.

I concetti di cammino e ipercammino percorribili sulla rete analizzata vengono presentati nella parte finale del capitolo.

Capitolo 2 : l'attenzione è rivolta al problema della valutazione degli ipercammini sull'ipergrafo.

Vengono introdotti dei criteri di costo per i vari tipi di arco presenti, e definiti i costi di cammini e ipercammini.

Nell'ultima parte è illustrato un metodo iterativo di calcolo del costo di un ipercammino, corredata da un esempio numerico.

Capitolo 3 : viene definito e analizzato il problema centrale della tesi: la determinazione di ipercammini minimi.

Dopo aver mostrato il criterio di determinazione dell'*insieme attrattivo ottimale* (di cui è fornito un esempio numerico), vengono presentate due versioni dell'algoritmo atto a individuare gli ipercammini di costo minimo.

Nella parte finale è presente un esempio completo che illustra le iterazioni dell'algoritmo, soffermandosi sui passaggi più rilevanti.

Capitolo 4 : mostra i dettagli relativi alla fase implementativa nel linguaggio C++, ponendo l'attenzione sulle scelte più importanti.

Inizialmente vengono definiti i formati di input/output utilizzati per importare un ipergrafo ed esportare il relativo ipercammino minimo.

Successivamente vengono approfonditi alcuni dettagli riguardanti le strutture dati utilizzate e la loro gestione, presentando anche alcune porzioni di codice.

La parte finale riporta e descrive i risultati prodotti dall'applicazione degli algoritmi allo stesso esempio svolto manualmente nel capitolo 3.

Capitolo 5 : l'algoritmo sviluppato viene applicato ad un caso reale: le linee dei bus elettrici di Firenze.

Vengono presentati e discussi alcuni risultati ottenuti.

Appendice A : contiene una rapida introduzione alle librerie Boost utilizzate nel corso del lavoro, rivolgendo l'attenzione all'uso che ne è stato fatto per modellare l'ipergrafo in esame e alle tecniche adottate per manipolarlo.

La lettura di questa appendice è fortemente consigliata per una buona comprensione del capitolo 4.

Capitolo 1

Ipergrafi e modello della rete

Come anticipato nell'introduzione, gli **ipergrafi** sono una particolare struttura topologica che rappresenta un'estensione dei grafi tradizionali.

Questo capitolo ne illustrerà le caratteristiche principali e introdurrà le definizioni necessarie alla comprensione dei capitoli successivi.

1.1 Teoria degli ipergrafi e definizioni

Ricordo che un grafo non orientato è costituito da una coppia

$$G = (V, E)$$

dove V è l'insieme dei vertici (o nodi) e E l'insieme degli archi:

$$V = \{v_1, v_2, \dots, v_n\}$$

$$E = \{e_1, e_2, \dots, e_m\}$$

Ciascun arco consiste di una coppia di nodi:

$$e_i = (v_i, v_j) , \quad v_i, v_j \in V$$

Un ipergrafo non orientato è analogamente una coppia di insiemi

$$H = (V, E)$$

in cui ogni elemento di E prende il nome di **iperarco**: ciascuno di essi è costituito da un sottoinsieme dei nodi in V :

$$E \subseteq \{e_i \mid e_i \subseteq V\}$$

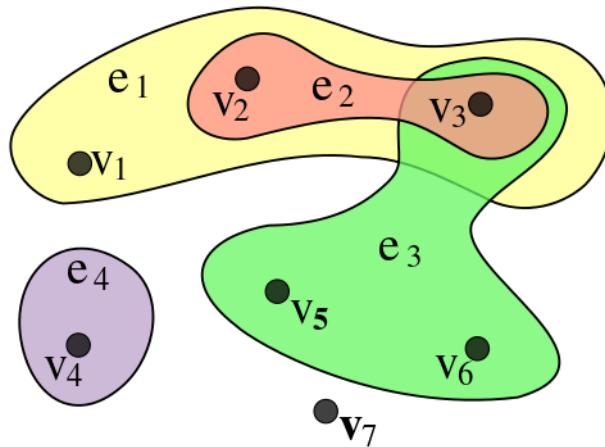


Figura 1.1: Esempio di ipergrafo non orientato

L'ipergrafo mostrato in fig. 1.1 è costituito dai nodi

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$

e dagli iperarchi

$$E = \{e_1, e_2, e_3, e_4\}$$

I suoi iperarchi sono definiti come

$$e_1 = \{v_1, v_2, v_3\} \quad e_2 = \{v_2, v_3\} \quad e_3 = \{v_3, v_5, v_6\} \quad e_4 = \{v_4\}$$

Si nota che e_4 , avente $|e_4| = 1$, è un arco di *loop* (rappresentabile graficamente come una linea che si richiude nel nodo di origine); mentre e_2 (per cui vale

$|e_2| = 2$) è un normale arco (raffigurabile con una linea che connette i due nodi).

Gli iperarchi con cardinalità > 2 sono rappresentabili con una superficie che racchiude i nodi.

Nel caso in cui $|e_i| = 2 \quad \forall e_i \in E$ l'ipergrafo si riduce ad un grafo tradizionale.

1.1.1 Ipergrafi orientati

Poichè questo lavoro è incentrato sulla ricerca di cammini e percorsi, il campo d'interesse è ristretto agli ipergrafi *diretti*.

Proseguendo il confronto con i grafi, ricordo che in un grafo orientato ciascun arco equivale ad una coppia *ordinata* di nodi: l'arco ha verso uscente dal primo nodo (nodo *coda* o *tail*) ed entrante nel secondo (nodo *testa* o *head*). Per quanto riguarda gli ipergrafi, stabilire un ordinamento tra i nodi di un iperarco equivale a partizionare l'iperarco nei due sottoinsiemi di nodi **head** e **tail**:

$$e_i = (T(e_i), h(e_i)), \quad T(e_i) \subset V, \quad h(e_i) \subset V/T(e_i) \quad (1.1)$$

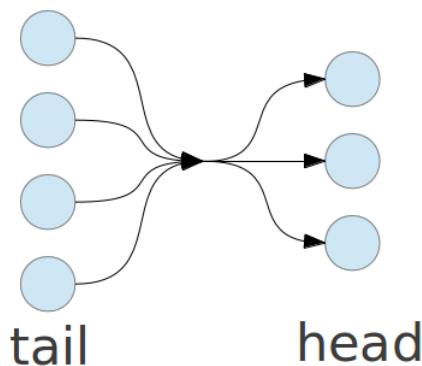


Figura 1.2: Esempio di iperarco orientato

In particolare, gli ipergrafi utilizzati in questo lavoro sono gli ipergrafi diretti aventi

$$|T(e_i)| = 1 \quad \forall e_i \in E$$

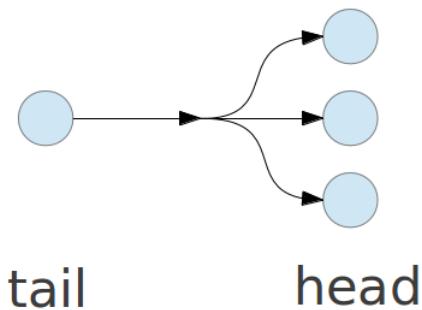


Figura 1.3: Iperarco orientato e con $|T(e)| = 1$

Gli iperarchi aventi un unico nodo di testa rappresentano uno strumento di modellazione idoneo per il trasporto pubblico: da un'unica fermata un passeggero può salire su più veicoli distinti.

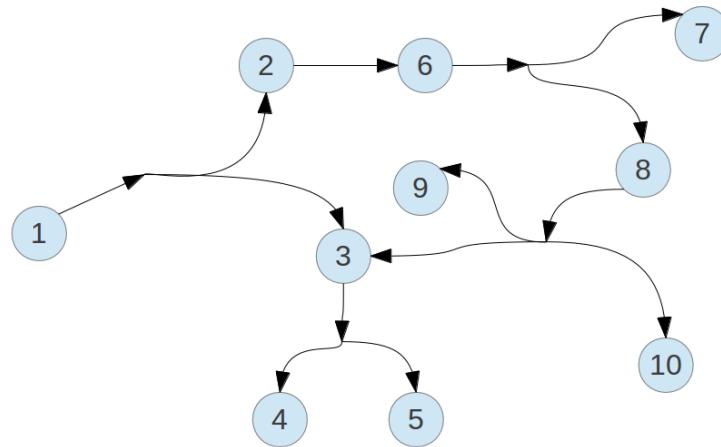


Figura 1.4: Ipergrafo orientato con $|T(e_i)| = 1 \quad \forall e_i \in E$

1.1.2 Definizioni

Nella trattazione successiva verranno utilizzati i concetti di *stella uscente* e *stella entrante*, le cui definizioni sono di seguito formulate:

Def. Dato un ipergrafo $H = (V, E)$, per ciascun nodo $v \in V$ la *stella uscente* (forward star) **FS** e la *stella entrante* (backward star) **BS** si definiscono come:

$$FS(v) = \{e_i \in E | v = T(e_i)\}$$

$$BS(v) = \{e_i \in E | v \in h(e_i)\}$$

In altre parole, la stella entrante è composta dagli archi che insistono sul nodo, mentre la stella uscente è l'insieme degli archi percorribili a partire dal nodo.

Più avanti, in particolare nella definizione di ipercammino, risulterà necessaria anche la definizione di sotto-ipergrafo:

Def. Un ipergrafo $\bar{H} = (\bar{V}, \bar{E})$ è un *sotto-ipergrafo* di $H = (V, E)$ se $\bar{V} \subseteq V$, $\bar{E} \subseteq E$ e ogni nodo presente negli iperarchi di \bar{E} è presente in \bar{V} .

1.2 Rete di trasporto

La struttura su cui opereranno gli algoritmi sviluppati modella una rete urbana, composta da una parte pedonale e da una corrispondente alle linee del trasporto pubblico locale.

La rete pedonale è rappresentata da un grafo i cui archi rappresentano i possibili tragitti percorribili; trattandosi di spostamenti da compiere a piedi, ciascun arco è percorribile in entrambi i sensi. Poichè complessivamente l'ipergrafo è orientato, questa situazione è resa dalla presenza di una coppia di archi (una per ogni senso) tra ogni coppia di *nodi pedonali*.

Ciò che rende tale struttura un ipergrafo propriamente detto è la presenza di nodi fittizi, detti *nodi fermata*, cui sono associati eventi aleatori: l'arrivo dei veicoli del trasporto pubblico.

Da ogni nodo fermata ha origine un iperarco, avente tanti rami quante sono le linee di trasporto pubblico che servono la fermata. I nodi di testa degli iperarchi vengono pertanto denominati *nodi linea* o *nodi a bordo*.

Degli opportuni *nodi di discesa* permettono di connettere ciascun nodo a bordo al grafo pedonale.

Riepilogando, nell'ipergrafo modellante la rete urbana sono presenti quattro tipi di nodi:

- **nodi pedonali:** sono i normali nodi del grafo pedonale.
- **nodi fermata:** modellano le fermate del trasporto pubblico mediante un iperarco uscente corrispondente alle linee che servono la fermata.
- **nodi linea o a bordo:** modellano le tappe di ciascuna linea.
- **nodi di discesa:** sono nodi pedonali cui un passeggero accede in seguito alla discesa da un veicolo.

Per lo studio in oggetto, è utile partizionare l'insieme V dei nodi:

$$V = N \cup F$$

dove N contiene i nodi pedonali, a bordo e di discesa, e F i nodi fermata.

E' opportuno definire una classificazione anche per gli archi (e iperarchi) dell'ipergrafo:

- **archi pedonali:** connettono i nodi del grafo pedonale.
- **archi a bordo:** connettono i nodi di linea, corrispondono al tragitto percorso da un veicolo tra una fermata e la successiva.
- **archi di discesa:** collegano un nodo di bordo con il nodo discesa associato.
- **iperarchi di salita:** come anticipato, modellano l'arrivo di un insieme di mezzi ad una fermata. Collegano un nodo fermata con un insieme di nodi linea.

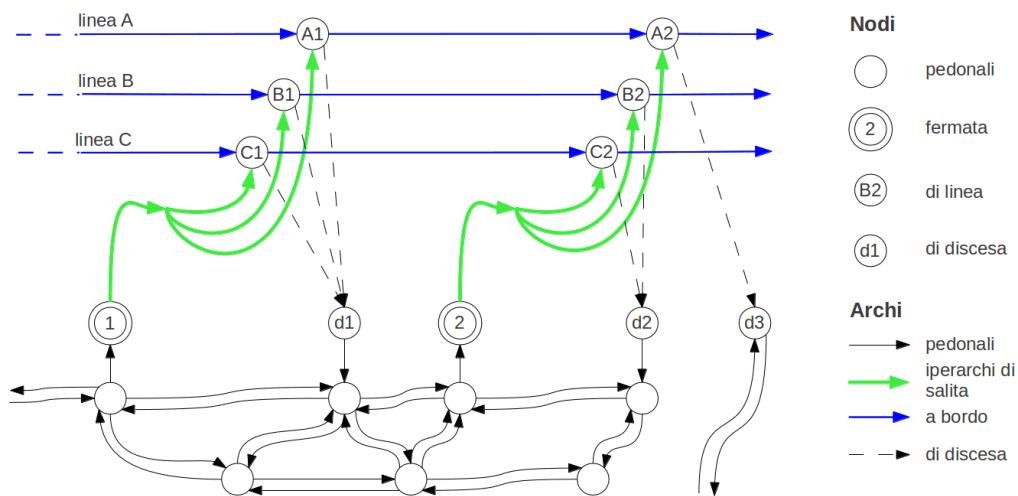


Figura 1.5: Struttura della rete di trasporto

1.2.1 Insieme attrattivo

Abbiamo già osservato che da ciascun nodo fermata $f \in F$ ha origine un iperarco di salita avente $|FS(f)|$ rami: ciascuno di essi corrisponde ad una linea di trasporto pubblico che serve il nodo f .

Nel modello considerato, si assume che la possibilità di scelta di ogni passeggero si limiti alla determinazione, a ciascun nodo fermata $f \in F$, di un sottoinsieme di linee considerate *attrattive* per raggiungere la destinazione desiderata.

Una volta selezionato tale insieme $L(f) \subseteq FS(f)$, il passeggero è disposto a salire sul primo mezzo *attrattivo* che arriva alla fermata (naturalmente la scelta dell'insieme attrattivo è strettamente legata alla destinazione prescelta, in altre parole all'ipercammino considerato dall'utente).

Consideriamo un caso pratico: un utente è in attesa ad una fermata servita dalle linee $k1$ e $k2$. Supponiamo che il tempo di arrivo a destinazione utilizzando la linea $k1$ sia minore di quello che si avrebbe utilizzando $k2$.

La linea $k2$ è *attrattiva* se non conviene aspettare l'arrivo di $k1$ nel caso in cui sia $k2$ la prima ad arrivare alla fermata. Questo significa che il maggior tempo di viaggio legato alla linea $k2$ è compensato dalla riduzione del tempo di attesa alla fermata.

Viceversa, la linea $k2$ *non* è attrattiva se, nel caso in cui sia la prima a servire la fermata, sia conveniente per il passeggero non salire e aspettare la linea $k1$. Questo equivale a dire che la differenza tra i tempi di arrivo di $k1$ e $k2$ è tale da rendere preferibile aumentare il tempo di attesa alla fermata.

Ulteriori definizioni e proprietà relative agli insiemi attrattivi, insieme ad un metodo per la loro determinazione, sono fornite nel terzo capitolo.

Nel corso del lavoro gli esempi illustrati riguarderanno l'ipergrafo H di figura 1.6, modellante una porzione della rete di traffico:

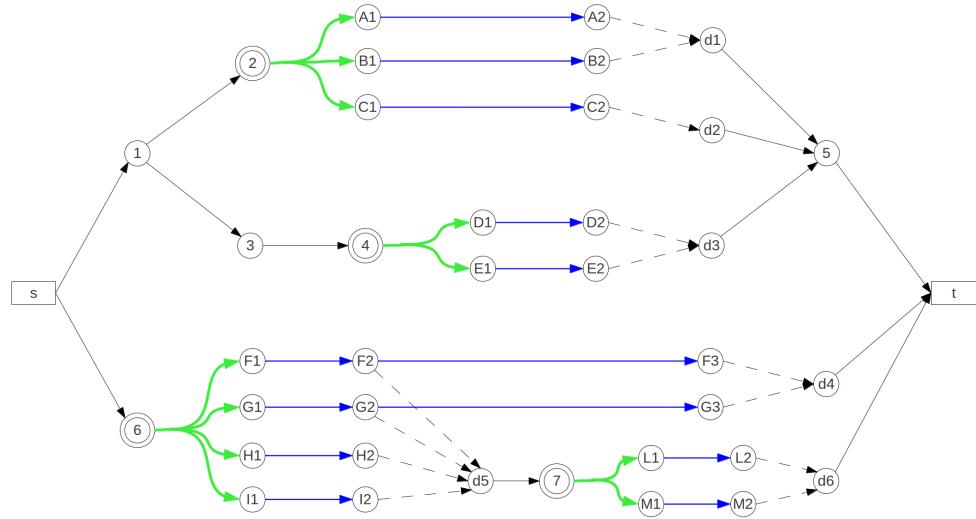


Figura 1.6: Ipergrafo H

L'ipergrafo H presenta una serie di percorsi che collegano i nodi s e t , rispettivamente origine e destinazione dei passeggeri.

Sono presenti quattro nodi $\in F$: essi consentono la salita, rispettivamente, sulle linee $[A, B, C]$, $[D, E]$, $[F, G, H, I]$, $[L, M]$.

Ciascuna linea, nella porzione di rete mostrata, compie una sola tappa tra salita e discesa finale.

Fanno eccezione le linee F e G : dopo una prima tappa, è possibile proseguire per una seconda o scendere attraverso il nodo $d5$.

1.2.2 Cammini e ipercammini

Un *cammino* q di lunghezza k tra i nodi s e t è definibile semplicemente come una sequenza di nodi (indifferentemente $\in F$ o $\in N$) e di archi (o rami di iperarchi):

$$q = \{s, e_0, v_1, e_1, v_2 \dots v_{k-1}, e_{k-1}, t\}$$

tale che

$$s = T(e_0), \quad t \in h(e_{k-1}), \quad v_j \in h(e_{j-1}) \cap T(e_j) \quad \forall j = 1, \dots, k-1$$

La figura seguente evidenzia (in rosso) un possibile cammino sull'ipergrafo H :

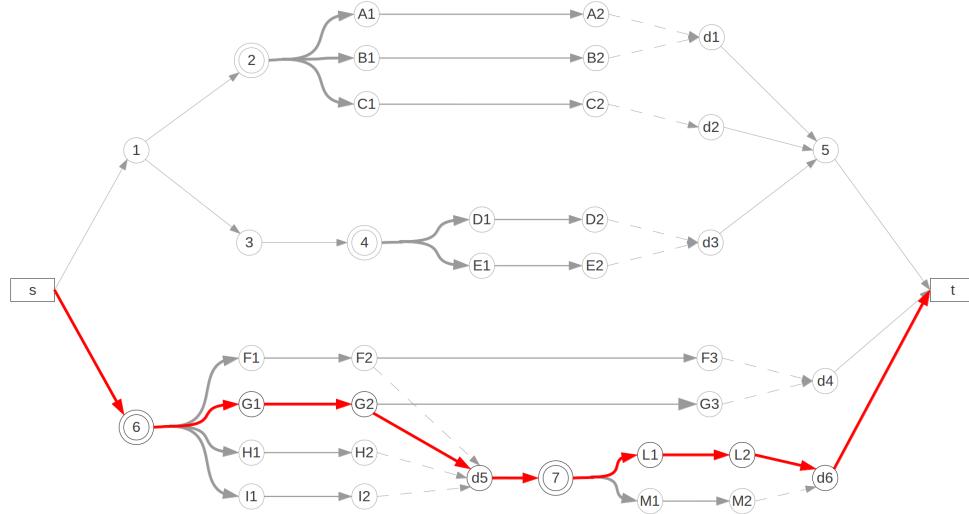


Figura 1.7: Cammino q di lunghezza 9 su H

Come precedentemente introdotto, ai nodi fermata la scelta sul prossimo arco (per la precisione *ramo dell'iperarco*) da percorrere non è deterministica ma basata su un evento aleatorio (l'arrivo del primo mezzo attrattivo); pertanto

i cammini semplici (con un'unica alternativa disponibile per ogni nodo) non sono adatti a schematizzare il percorso di un utente.

L'**ipercammino** è costituito da un insieme di possibili cammini tra la coppia origine/destinazione scelta: ciascuno di questi cammini è caratterizzato dalla probabilità di essere interamente percorso.

Un ipercammino è in pratica un *sotto-ipergrafo* p che rappresenta l'insieme dei possibili cammini da s a t , caratterizzato dalla mancanza di archi entranti in s e uscenti da t , e dall'essere fortemente aciclico (non esistono cicli orientati).

Un possibile ipercammino p che collega s a t è evidenziato nella figura 1.8.

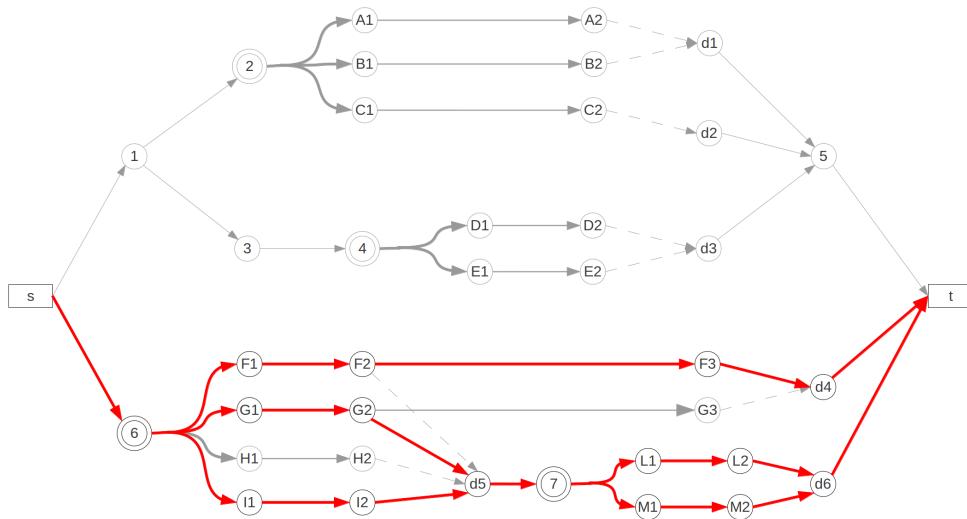


Figura 1.8: Ipercammino p su H

Capitolo 2

Calcolo del costo di un ipercammino

2.1 Notazione

Prima di proseguire nella trattazione, è opportuno riepilogare e fissare la notazione già introdotta nel primo capitolo:

p	ipercammino dall'origine s alla destinazione t
$FS(u), u \in F$	nel caso di nodi fermata, la stella uscente corrisponde all'insieme delle linee che servono la fermata
$L_p(u), u \in f$	insieme attrattivo percepito alla fermata u dagli utenti associati all'ipercammino p
φ_l	frequenza oraria (intesa come numero di passaggi in un'ora) della linea l

2.2 Etichettatura degli archi

2.2.1 Costo

Poichè nella rete di trasporto sono presenti vari tipi di arco, è necessario determinare un'unità di costo omogenea per scegliere tra le alternative possibili tenendo conto dei vari fattori considerabili (attesa alle fermate, costo del biglietto, spostamenti a piedi...).

Tale costo *generalizzato*, di seguito denotato semplicemente come *costo*, assume un diverso significato dipendentemente dal tipo di arco:

- per gli archi *pedonali* il costo dipende linearmente dalla lunghezza del tratto. Per rendere questi archi confrontabili con quelli a bordo, è stato deciso di associargli il tempo effettivo di percorrenza a piedi.
- il costo degli archi *a bordo* dipende principalmente dalla durata temporale del viaggio, ma è possibile introdurre una dipendenza anche dallo stato di congestione del mezzo (o quantità di flusso).
- gli archi *di discesa* vengono etichettati con un valore che riassume il costo del biglietto e un tasso di scomodità dovuto all'uso dei mezzi pubblici.
- i rami di ciascun iperarco *di salita* hanno un costo legato al tempo di attesa alla fermata; il calcolo di tale valore verrà approfondito nel prossimo paragrafo.

Tempo di attesa

E' necessario innanzitutto assumere delle ipotesi semplificative:

- Ad ogni nodo fermata, l'insieme attrattivo per giungere ad una certa destinazione è lo stesso per tutti i passeggeri in attesa.
- L'arrivo dei passeggeri alle fermate è indipendente dall'arrivo dei veicoli.
- Ad ogni fermata, l'arrivo dei veicoli di una linea è indipendente dall'arrivo dei veicoli di altre linee.

Introduciamo la nozione di frequenza combinata associata ad una fermata:

Def. Siano $L_p(f)$ l'insieme attrattivo del nodo fermata $f \in F$ (secondo l'ipercammino p), e φ_l la frequenza oraria di ciascuna linea attrattiva $l \in L_p(f)$.

La **frequenza combinata** dell'intero insieme attrattivo è definita come:

$$\phi_p(f) = \sum_{l \in L_p(f)} \varphi_l \quad (2.1)$$

A questo punto, prima di poter definire il tempo medio di attesa ad una fermata, è necessario fare alcune considerazioni circa la **regolarità del servizio**.

Definiamo l'*intertempo* come il tempo medio tra il passaggio di due veicoli ad una certa fermata.

Nel caso di cadenzamento perfettamente rispettato (i veicoli di una linea arrivano equispaziati temporalmente), l'attesa media corrisponde alla metà dell'*intertempo*.

Se invece l'irregolarità del servizio è massima (la congestione del traffico causa

l'arrivo di più veicoli accodati), l'attesa media equivale all'intero *intertempo*. Si introduce il parametro θ , che è allo stesso tempo il parametro di conversione in unità di *costo generalizzato* e il coefficiente di regolarità del servizio; esso può assumere valori compresi nell'intervallo [30, 60] (assumendo che i tempi di percorrenza associati agli archi siano espressi in minuti).

Con tale notazione è possibile dare la definizione seguente:

Def. Il **tempo medio di attesa** associato ad una fermata è esprimibile come:

$$w_p(f) = \frac{\theta}{\phi_p(f)} \quad (2.2)$$

Esso rappresenta il tempo medio di attesa prima dell'arrivo del primo veicolo alla fermata, pertanto ogni arco di salita corrispondente ad una linea attrattiva verrà etichettato con tale valore.

2.2.2 Probabilità

Gli archi *di viaggio* (pedonali, a bordo e di discesa) inclusi in un cammino hanno probabilità 1 di essere percorsi: per ogni nodo in N la scelta sull'arco da percorrere è deterministica.

La questione diventa rilevante nell'analisi di nodi fermata e iperarchi di salita.

Ciascuna fermata f in p avrà un proprio insieme attrattivo $L_p(f)$; la probabilità che un veicolo della linea l sia il primo ad arrivare alla fermata è data da:

$$\pi_p(f, l) = \frac{\varphi_l}{\phi_p(f)}, \quad \forall l \in L_p(f) \quad (2.3)$$

Vale ovviamente che:

$$\pi_p(f, l) > 0, \quad \sum_{l \in L_p(f)} \pi_p(f, l) = 1, \quad \forall l \in L_p(f)$$

Il valore $\pi_p(f, l)$ rappresenta la probabilità di percorrere l'arco di salita (f, l) condizionata dall'essere alla fermata f .

Secondo tale interpretazione, l'estensione della probabilità condizionata anche agli archi di viaggio è semplice:

$$\pi_p(u, v) = \begin{cases} 0, & (u, v) \notin p \\ 1, & (u, v) \in p \end{cases} \quad (u, v) \text{ arco di viaggio} \quad (2.4)$$

Quindi la probabilità di percorrere un qualsiasi arco del grafo, condizionata al fatto di trovarsi in corrispondenza di un suo nodo di coda, è:

$$\pi_p(u, v) = \begin{cases} 1, & (u, v) \in p \text{ e } u \in N \\ \frac{\varphi_v}{\phi_p(u)}, & (u, v) \in p \text{ e } u \in F \\ 0, & (u, v) \notin p \end{cases} \quad (2.5)$$

2.3 Etichettatura dei cammini

2.3.1 Probabilità

Dalle probabilità condizionate, appena introdotte per gli archi, è possibile ottenere le probabilità di percorrere i singoli cammini di un ipercammino p . Un cammino q dell'ipercammino p è costituito da una sequenza ordinata di archi che connette s a t , tale che per ogni fermata f in q solo un arco di salita tra quelli appartenenti a $L_p(f)$ appare in q .

La probabilità di percorrere il cammino q tra tutti quelli che costituiscono p è data da:

$$\Omega_p(q) = \prod_{(f,l) \in q} \pi_p(f, l) = \prod_{(u,v) \in q} \pi_p(u, v) , \quad \forall q \in p \quad (2.6)$$

(l'equivalenza è implicata dal fatto che $\pi_p(u, v) = 1$ se $(u, v) \in q$).

Vale che:

$$\Omega_p(q) > 0 , \quad \sum_{q \in p} \Omega_p(q) = 1 , \quad \forall q \in p$$

2.3.2 Costo

Il costo di un cammino è definibile semplicemente come la somma dei costi (generalizzati) degli archi che lo compongono:

$$c(q) = \sum_{(u,v) \in q} c_{uv} \quad (2.7)$$

2.4 Costo degli ipercammini

A questo punto disponiamo di tutte le definizioni necessarie per definire e calcolare il costo di un ipercammino p .

Esso dipende banalmente dal costo dei cammini da cui è costituito e dalle probabilità di percorrerli:

$$c(p) = \sum_{q \in p} \Omega_p(q) c(q) \quad (2.8)$$

La definizione di costo di ipercammino p data in (2.7) richiede la conoscenza esplicita di tutti i cammini di p , con relativi costi e probabilità di realizzazione.

La sezione 2.4.2 illustrerà una procedura di calcolo che non presuppone la conoscenza completa dei cammini.

Prima è necessario definire il costo atteso di un *sotto-ipercammino*.

2.4.1 Costo atteso di un sotto-ipercammino

Dato l'ipercammino p da s a t , indichiamo con $c_u(p)$ il costo del *sotto-ipercammino* di p da u a t .

Consideriamo il nodo u di p e supponiamo di conoscere il costo $c_v(p)$ per ogni nodo v tale che $(u, v) \in FS_p(u)$.

Il costo del sotto-ipercammino di p da u a t in funzione dei $c_v(p)$ è dato da:

$$c_u(p) = \sum_{(u,v) \in FS(u)} \pi_p(u, v) (c_{uv} + c_v(p)) \quad (2.9)$$

Analizziamo la (2.9) distinguendo due casi:

- $u \in N$

La scelta sull'arco successivo da percorrere tra tutti quelli in $FS(u)$ è univoca, pertanto esiste un unico arco $(u, v) \in FS(u)$ avente $\pi_p(u, v) = 1$. Per gli altri vale $\pi_p(u, v) = 0$ (vedi (2.4)).

La (3.1) si riduce al caso semplice di aggiornamento del costo in un cammino:

$$c_u(p) = c_{uv} + c_v(p)$$

Il costo del sotto-ipercammino da u a t è banalmente pari al costo del sotto-ipercammino da v a t addizionato al costo dell'arco di viaggio che connette u a v .

- $u \in F$

Gli archi in $FS(u)$ aventi $\pi_p(u, v) \neq 0$ sono solo quelli appartenenti all'insieme attrattivo $L_p(u)$, quindi la (2.9) può essere scritta solo in funzione di tali archi:

$$c_u(p) = \sum_{(u,v) \in L_p(u)} \pi_p(u, v) (c_{uv} + c_v(p))$$

sostituendo $\pi_p(u, v)$ con la definizione data in (2.3) e c_{uv} con la definizione di tempo di attesa in (2.2)

$$\begin{aligned} c_u(p) &= \sum_{(u,v) \in L_p(u)} \frac{\varphi_v}{\phi_p(u)} \left(\frac{\theta}{\phi_p(u)} + c_v(p) \right) = \\ &= \frac{1}{\phi_p(u)} \sum_{(u,v) \in L_p(u)} \left(\frac{\varphi_v \theta}{\phi_p(u)} + \varphi_v c_v(p) \right) = \\ &= \frac{1}{\phi_p(u)} \left(\theta \frac{\sum_{(u,v) \in L_p(u)} \varphi_v}{\phi_p(u)} + \sum_{(u,v) \in L_p(u)} \varphi_v c_v(p) \right) = \frac{\theta + \sum_{(u,v) \in L_p(u)} \varphi_v c_v(p)}{\phi_p(u)} \end{aligned}$$

Esso rappresenta la speranza matematica del costo (*costo atteso*) rispetto alle probabilità dell'arrivo del primo veicolo alla fermata.

Quindi il costo (atteso) di viaggio da ogni nodo u alla destinazione t secondo l'ipercammino p , già introdotto in (2.9), è esprimibile come:

$$c_u(p) = \begin{cases} c_{uv} + c_v(p), & u \in N \\ \theta + \sum_{(u,v) \in L_p(u)} \varphi_v c_v(p) \\ \frac{\phi_p(u)}{\phi_p(u)}, & u \in F \end{cases} \quad (2.10)$$

2.4.2 Calcolo iterato del costo di un ipercammino

L'obiettivo è determinare il costo di un ipercammino senza passare attraverso l'enumerazione completa dei cammini: questo è reso possibile dall'aciclicità degli ipercammini.

Tale proprietà consente di definire una procedura iterativa che esplora l'ipercammino *a ritroso*, calcolando ad ogni i -esimo passo il costo atteso di un sotto-ipercammino di lunghezza i .

La procedura prevede di etichettare il nodo terminale con $c_t(p) = 0$ e di visitare i nodi dell'ipercammino all'indietro, aggiornandone il costo secondo la (2.10).

Una volta raggiunto il nodo origine s , il costo complessivo dell'ipercammino è determinato:

$$c(p) = c_s(p) \quad (2.11)$$

2.4.3 Esempio

Riprendiamo l'ipercammino p preso a esempio nel secondo capitolo, caratterizzato dai costi sugli archi mostrati in fig. 2.1:

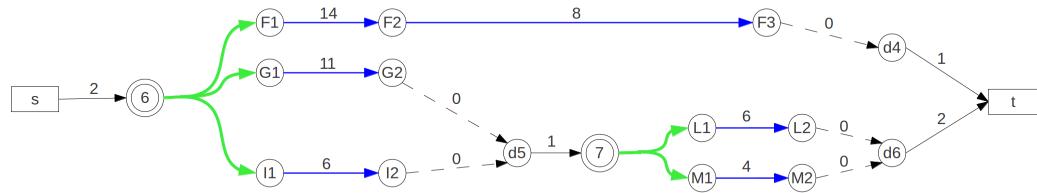


Figura 2.1: Ipercammino p su H

Le frequenze orarie delle linee alla fermata 6 sono

$$\{\varphi_F, \varphi_G, \varphi_I\} = \{5, 2, 3\}$$

e quelle al nodo fermata 7:

$$\{\varphi_L, \varphi_M\} = \{3, 3\}$$

Avvalendoci della (2.10):

$$c_u(p) = \begin{cases} c_{uv} + c_v(p), & u \in N \\ \theta + \sum_{(u,v) \in L_p(u)} \varphi_v c_v(p) \\ \frac{ }{\phi_p(u)}, & u \in F \end{cases}$$

per la determinazione del costo atteso di ciascun nodo, illustreremo le iterazioni principali (i costi attesi sono riportati in grassetto sopra ciascun nodo).

Dopo aver inizializzato $c_t = 0$ e aver applicato i primi passaggi banali, la situazione è quella mostrata in fig. 2.2.

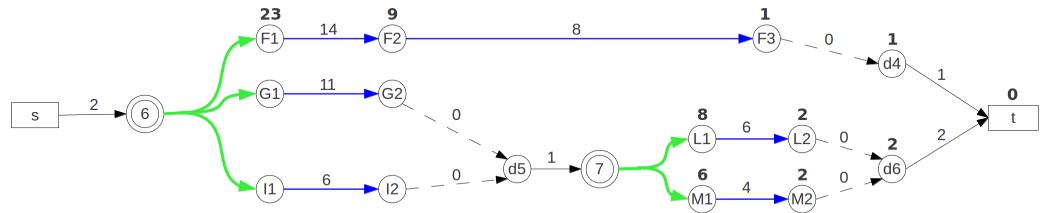


Figura 2.2: Iterazione 1

A questo punto, il costo atteso del nodo fermata 7 è dato da

$$c_7 = \frac{\theta + c_{L1} \cdot \varphi_L + c_{M1} \cdot \varphi_M}{\phi_7} = \frac{30 + 8 \cdot 3 + 6 \cdot 3}{6} = 12$$

da cui i costi di fig. 2.3:

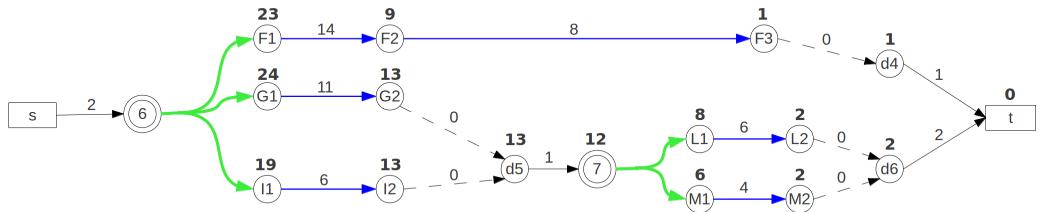


Figura 2.3: Iterazione 2

Allo stesso modo è possibile calcolare il costo atteso del nodo fermata 6 come

$$c_6 = \frac{\theta + c_{F1} \cdot \varphi_F + c_{G1} \cdot \varphi_G + c_{I1} \cdot \varphi_I}{\phi_6} = 25$$

Come mostrato in figura 2.4, l'ipercammino p ha un costo pari a

$$c = c_s = 27$$

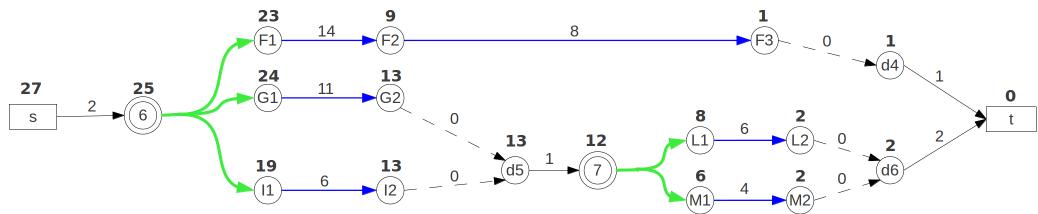


Figura 2.4: Iterazione 3

Ora che si dispone degli strumenti necessari al calcolo del costo degli ipercammini, è possibile affrontare il problema centrale di questo lavoro: la determinazione, tra tutti i possibili ipercammini che connettono origine e destinazione in un certo ipergrafo, degli ipercammini avente costo minimo.

Il capitolo terzo formalizzerà questo problema e presenterà gli schemi algoritmici studiati.

Capitolo 3

Determinazione degli ipercammini minimi

3.1 Definizione del problema

E' opportuno ricordare che il costo *atteso* di ciascun nodo u in un ipercammino p , corrispondente al costo atteso del sotto-ipercammino di p da u a t , è calcolabile come:

$$c_u(p) = \begin{cases} c_{uv} + c_v(p), & u \in N \\ \theta + \sum_{(u,v) \in L_p(u)} \varphi_v c_v(p) \\ \frac{\theta + \sum_{(u,v) \in L_p(u)} \varphi_v c_v(p)}{\phi_p(u)}, & u \in F \end{cases} \quad (3.1)$$

dove $L_p(u) \subseteq FS(u)$ è l'insieme delle linee percepite come attrattive dagli utenti associati all'ipercammino p che sono in attesa alla fermata u .

Nel corso di questo capitolo, dando per scontato di considerare l'ipercammino p , lo sottointenderemo per una maggiore chiarezza nella notazione (ad esempio, $c_u(p)$ verrà sostituito con c_u , $L_p(u)$ con $L(u)$...)

Gli ipercammini minimi sono caratterizzati dalle proprietà seguenti (analoghe a quelle valide per i cammini minimi):

Proprietà 1. *Per ogni nodo fermata $u \in F$ e ogni arco $(u, v) \in L_p(u)$, se p_u è ipercammino minimo da u a t , allora ogni sotto-ipercammino p_v di p_u da v a t è ipercammino minimo per v .*

Proprietà 2. *Per ogni generico nodo u è possibile costruire un ipercammino minimo p_u in modo che ciascun sotto-ipercammino p_v di p_u sia ipercammino minimo per il nodo v .*

L'ottimalità di un ipercammino è garantita se e solo se valgono le **condizioni generalizzate di Bellman**:

$$c_u \leq \begin{cases} c_{uv} + c_v, & \text{se } u \notin F \\ \frac{\theta + \sum_{(u,v) \in L(u)} c_v \varphi_v}{\sum_{(u,v) \in L(u)} \varphi_v}, & \text{se } u \in F \end{cases} \quad (3.2)$$

Possono anche essere rappresentate in forma di **equazioni**:

$$c_u = \begin{cases} c'_u, & \text{se } u \in N \\ c''_u, & \text{se } u \in F \end{cases} \quad (3.3)$$

dove

$$c'_u = \min \{c_{uv} + c_v : (u, v) \in FS(u)\}, \quad \forall u \in N \quad (3.4)$$

$$c''_u = \min \left\{ \frac{\theta + \sum_{(u,v) \in L(u)} c_v \varphi_v}{\sum_{(u,v) \in L(u)} \varphi_v} : L(u) \subseteq FS(u) \right\}, \quad \forall u \in F \quad (3.5)$$

Per i nodi in N le condizioni e le equazioni generalizzate coincidono con quelle valide per i grafi tradizionali.

Per i nodi fermata in F , invece, indicano che il costo (atteso) deve essere non superiore al costo relativo ad ogni possibile insieme attrattivo (condizione generalizzata), e deve esistere un insieme attrattivo (ottimo) per cui è verificata l'equazione generalizzata.

La sezione 3.2 approfondirà il concetto di insieme attrattivo ottimo e illustrerà un metodo per determinarlo.

3.2 Insieme attrattivo ottimale

Ricordo ancora una volta che il costo dell'ipercammino p da un nodo fermata $u \in f$ alla destinazione t è dato da:

$$c_u = \frac{\theta + \sum_{(u,v) \in L(u)} c_v \varphi_v}{\sum_{(u,v) \in L(u)} \varphi_v}, \text{ dove } L(u) \subseteq FS(u) \quad (3.6)$$

(sottointendendo che costi e insiemi attrattivi sono relativi all'ipercammino p).

Esso rappresenta la speranza matematica del costo necessario ad arrivare da u a t , considerando che ogni utente salga sul primo mezzo disponibile tra quelli appartenenti all'insieme $L(u)$.

Pertanto è necessario determinare quali linee in $FS(u)$ debbano far parte dell'insieme attrattivo $L(u)$ in modo da minimizzare tale costo atteso.

3.2.1 Proprietà

Supponiamo che gli l archi di salita uscenti da u siano ordinati secondo l'ordine non decrescente dei costi:

$$FS(u) = \{(u, v_1), (u, v_2), \dots (u, v_l)\}$$

dove

$$c_{v_1} \leq c_{v_2} \leq \dots \leq c_{v_l}$$

Definiamo $L^r = \{v_1, v_2, \dots, v_r\}$, con $r < l$, come un possibile insieme attrattivo formato dalle prime r linee; indicando la sua frequenza cumulata con

$$\phi_u^r = \sum_{s=1}^r \varphi_{v_s}$$

Il costo atteso relativo al nodo u considerando come insieme attrattivo L^r è calcolabile come:

$$c_u^r = \frac{\theta + \sum_{(u,v) \in L^r} c_v \varphi_v}{\sum_{s=1}^r \varphi_{v_s}} = \frac{\theta + \sum_{s=1}^r c_{v_s} \varphi_{v_s}}{\phi_u^r} \quad (3.7)$$

Valutiamo ora il costo atteso considerando l'insieme $L^{r+1} = L^r \cup \{v_{r+1}\}$:

$$c_u^{r+1} = \frac{\theta + \sum_{s=1}^{r+1} c_{v_s} \varphi_{v_s}}{\phi_u^{r+1}} = \frac{\phi_u^r}{\phi_u^{r+1}} \cdot \frac{\theta + c_{v_{r+1}} \varphi_{v_{r+1}} + \sum_{s=1}^r c_{v_s} \varphi_{v_s}}{\phi_u^r} \quad (3.8)$$

ed esprimiamolo in funzione di c_u^r :

$$c_u^{r+1} = \frac{\phi_u^r}{\phi_u^{r+1}} \cdot \left(\frac{c_{v_{r+1}} \varphi_{v_{r+1}}}{\phi_u^r} + c_u^r \right) \quad (3.9)$$

da cui:

$$\phi_u^{r+1} c_u^{r+1} = c_{v_{r+1}} \varphi_{v_{r+1}} + \phi_u^r c_u^r \quad (3.10)$$

la (3.10) esplicita la variazione del costo atteso che l'inserimento della linea v_{r+1} nell'insieme attrattivo comporta.

E' possibile determinare la relazione tra c_u^r e c_u^{r+1} facendo varie ipotesi su $c_{v_{r+1}}$. Ad esempio:

$$\begin{aligned} c_{v_{r+1}} &< c_u^r \Rightarrow \\ \phi_u^{r+1} c_u^{r+1} &= c_{v_{r+1}} \varphi_{v_{r+1}} + \phi_u^r c_u^r < (\varphi_{v_{r+1}} + \phi_u^r) c_u^r = \phi_u^{r+1} c_u^r \\ \Rightarrow c_u^{r+1} &< c_u^r \end{aligned} \quad (3.11)$$

Ragionando analogamente per i casi $c_{v_{r+1}} = c_u^r$ e $c_{v_{r+1}} > c_u^r$ si ottiene lo schema seguente:

$$\begin{cases} c_{v_{r+1}} < c_u^r \Rightarrow c_u^{r+1} < c_u^r \\ c_{v_{r+1}} = c_u^r \Rightarrow c_u^{r+1} = c_u^r \\ c_{v_{r+1}} > c_u^r \Rightarrow c_u^{r+1} > c_u^r \end{cases} \quad (3.12)$$

In pratica, l'inclusione di una nuova linea nell'insieme attrattivo comporta una diminuzione del costo atteso solo se il costo netto di tale linea è inferiore al costo lordo attuale.

Pertanto è possibile definire una procedura incrementale per la definizione dell'insieme attrattivo ottimo: dopo aver ordinato in modo non decrescente le linee uscenti dal nodo $u \in F$ e aver aggiunto la prima di esse all'insieme attrattivo, è sufficiente aggiungerle incrementalmente all'insieme fino a che

il costo netto della prossima linea non supera (o eguaglia) il costo atteso corrente del nodo fermata.

La (3.8) può essere rivista per calcolare il costo atteso relativo all'insieme attrattivo L^{r+1} a partire dal costo (già calcolato) relativo all'insieme L^r , evitando calcoli superflui e ridondanti:

$$\begin{aligned} c_u^{r+1} &= \frac{\theta + \sum_{s=1}^{r+1} c_{v_s} \varphi_{v_s}}{\phi_u^{r+1}} = \frac{\theta + \sum_{s=1}^r c_{v_s} \varphi_{v_s} + c_{v_{r+1}} \varphi_{v_{r+1}}}{\phi_u^r + \varphi_{v_{r+1}}} = \\ &= \frac{c_u^r \phi_u^r + c_{v_{r+1}} \varphi_{v_{r+1}}}{\phi_u^r + \varphi_{v_{r+1}}} = \frac{c_u^r (\phi_u^{r+1} - \varphi_{v_{r+1}}) + c_{v_{r+1}} \varphi_{v_{r+1}}}{\phi_u^r + \varphi_{v_{r+1}}} = \\ &= c_u^r + \frac{-c_u^r \varphi_{v_{r+1}} + c_{v_{r+1}} \varphi_{v_{r+1}}}{\phi_u^r + \varphi_{v_{r+1}}} \end{aligned}$$

quindi:

$$c_u^{r+1} = c_u^r - \frac{(c_u^r - c_{v_{r+1}}) \varphi_{v_{r+1}}}{\phi_u^r + \varphi_{v_{r+1}}} \quad (3.13)$$

Una formalizzazione di questa procedura è fornita dalla sezione 3.2.2.

3.2.2 La procedura *attractive-set*

Il seguente pseudocodice implementa l'algoritmo visto nella sottosezione precedente per la determinazione dell'insieme attrattivo ottimale associato a un certo nodo fermata $u \in F$.

In input al metodo sono forniti, oltre al nodo $u \in F$ selezionato, le variabili $\text{suc}[u]$, ϕ_u e c_u . Al termine della procedura, esse conterranno rispettivamente le linee appartenenti all'insieme attrattivo, la loro frequenza combinata, e il costo atteso del nodo ad esse associato.

```

function ATTRACTIVE-SET( $u$ ,  $\text{suc}[u]$ ,  $\phi_u$ ,  $c_u$ )
    sorted-FS = sort( $FS(u)$ )
     $\text{suc}[u] = v_0$ 
     $\phi_u = \varphi_{v_0}$ 
     $c_u = \frac{\theta}{\phi_u} + c_{v_0}$ 
     $k = 1$ , stop-criterion = false

    while  $k < |\text{sorted-FS}|$  and stop-criterion == false do
        if  $c_{v_k} < c_u$  then
             $\text{suc}[u] = \text{suc}[u] \cup v_k$ 
             $\phi_u = \phi_u + \varphi_{v_k}$ 
             $c_u = c_u - \frac{(c_u - c_{v_k})\varphi_{v_k}}{\phi_u}$ 
             $k = k + 1$ 
        else
            stop-criterion = true
        end if
    end while
end function

```

3.2.3 Esempio di determinazione dell'insieme attrattivo ottimale

Vediamo un esempio pratico di calcolo dell'insieme attrattivo ottimo per il nodo fermata 6 dell'ipergrafo finora analizzato:

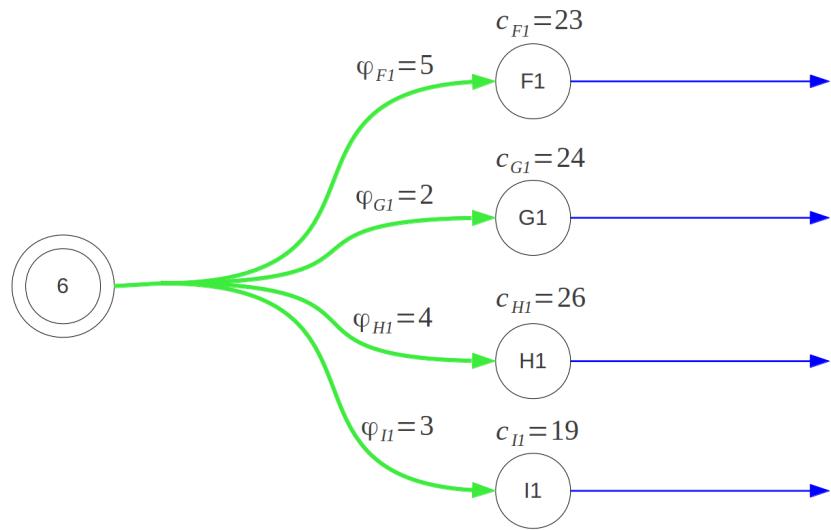


Figura 3.1: Iperarco uscente dal nodo $6 \in F$

Ciascun nodo linea è etichettato con il costo atteso per giungere a destinazione, mentre ad ogni ramo dell'iperarco di salita è associata la frequenza oraria (di passaggi alla fermata) della linea corrispondente (fig. 3.1).

La sequenza dei nodi ordinati per costo non decrescente è $\{I1, F1, G1, H1\}$. Analizziamo ogni iterazione:

- $L^1 = \{I1\}$, $\phi_6^1 = 3$

$$c_6^1 = \frac{\theta}{\phi_6^1} + c_{I1} = 29$$

La linea successiva da valutare è $F1$, e poichè $c_6^1 > c_{F1} = 23$ la procedura prosegue inserendo $F1$ tra le linee attrattive.

- $L^2 = \{I1, F1\}$, $\phi_6^2 = 8$

$$c_6^2 = c_6^1 - \frac{(c_6^1 - c_{F1}) \varphi_{F1}}{\phi_6^2} = 25, 25$$

Anche in questo caso $c_6^2 > c_{G1} = 24$, quindi inserisco $G1$ in L .

- $L^3 = \{I1, F1, G1\}$, $\phi_6^3 = 10$

$$c_6^3 = c_6^2 - \frac{(c_6^2 - c_{G1}) \varphi_{G1}}{\phi_6^3} = 25$$

Stavolta $c_6^3 < c_{H1} = 26$, quindi la procedura termina.

Quindi l'insieme attrattivo per il nodo $6 \in F$, considerando la destinazione secondo cui sono calcolati i costi attesi dei nodi linea, è costituito dalle linee

$$L = \{I1, F1, G1\}$$

La frequenza combinata associata a L è $\phi_6 = 10$, e il tempo di attesa alla fermata è pari a

$$w_6 = \frac{\theta}{\phi_6} = 3$$

Infine, il costo atteso per giungere dalla fermata 6 alla destinazione è dato da

$$c_6 = 25$$

3.3 L'algoritmo *Shortest Hyperpath Tree* (SHT)

Scopo dell'algoritmo è determinare l'*iperalbero di costo minimo* di radice t , cioè l'albero di ipercammini minimi aventi destinazione t .

Una volta fatto questo, determinare l'ipercammino minimo che connette s a t è banale.

L'idea è iniziare l'esplorazione dal terminale; ogni volta che si esamina un nodo v viene esplorata la sua stella entrante $BS(v)$ e, per ogni arco (u, v) , si controlla la *condizione* di Bellman adeguata (sulla base di $u \in N$ o $u \in F$): se essa non è rispettata, il costo del nodo u viene aggiornato in base all'*equazione* di Bellman adeguata e il nodo viene inserito nell'insieme dei nodi da espandere.

L'algoritmo termina quando non ci sono più nodi da esplorare.

Tale procedura restituisce, per ciascun nodo, il corrispondente costo atteso per giungere alla destinazione; se il nodo è in F verrà restituita anche la sua frequenza combinata e il tempo di attesa associato.

Inoltre viene prodotto il vettore dei successori suc , definito come:

$$suc[u] = \begin{cases} v = \operatorname{argmin}\{c_{v_i} | v_i \in FS(u)\}, & \text{se } u \in N \\ L(u), & \text{se } u \in F \end{cases}$$

Esso in pratica è una lista di liste, in cui la lista associata a ciascun nodo u contiene un unico successore (quello di costo minimo) se $u \in N$, o più successori (corrispondenti alle linee attrattive) se $u \in F$.

La procedura inizializza i costi attesi a ∞ , i successori a *nil* e le frequenze combinate dei nodi fermata a 0.

Viene utilizzata una coda Q per memorizzare i nodi da esplorare: il primo nodo ad essere inserito è il terminale t , dopo averne posto il costo a 0.

Ad ogni passo si estraе un nodo v dalla coda, e ciascun nodo $u \in BS(v)$ viene valutato:

- se $u \in N$ e $c_u > c_v + c_{uv}$, viene posto $c_u = c_v + c_{uv}$ e $suc[u] = v$.
- se $u \in F$ e $c_u > c_v$ (il costo netto della linea v è minore del costo atteso corrente della fermata), il suo costo viene aggiornato (insieme a $suc[u] = L(u)$ e ϕ_u) tramite la procedura *attractive-set* già vista nella sezione 3.2.2.

In entrambi i casi, in seguito all'aggiornamento di c_u , il nodo u viene inserito nella coda Q se non già presente (l'unica eccezione è rappresentata dal nodo sorgente s , il cui inserimento nella coda comporterebbe solo iterazioni non utili allo scopo). Di seguito è riportato lo pseudocodice dell'algoritmo presentato: Per quanto appena visto, la coda Q è una semplice lista in cui i nodi vengono estratti e inseriti senza una politica definita. Questo comporta il dover ricalcolare l'insieme attrattivo e il costo atteso di ciascuna fermata ogni volta che $c_u > c_v$ (l'inserimento in $L(u)$ di una linea dal costo più basso potrebbe provocare l'esclusione di una precedentemente considerata attrattiva).

Questo inconveniente può essere evitato implementando Q in forma di coda *di priorità*, in cui il primo nodo a essere estratto è quello con il costo atteso minore tra tutti quelli presenti. Analizziamo nella prossima sezione tale scenario e l'algoritmo che si ottiene.

```

function SHT( $s, t, \theta$ )
  for each  $u \in V$  do
     $c_u = \infty$ 
     $\text{suc}[u] = \text{nil}$ 
    if  $u \in F$  then  $\phi_u = 0$  end if
  end for
   $c_t = 0$ 
  insert( $t, Q$ )
  while  $Q \neq \emptyset$  do
    extract( $v, Q$ )
    for each  $(u, v) \in BS(v)$  do
      if  $u \in F$  and  $c_u > c_v$  then
        do attractive-set( $u, c_u, \phi_u, \text{suc}[u]$ )
        if  $u \notin Q$  and  $u \neq s$  then insert( $u, Q$ )
      else if  $u \in N$  and  $c_u > c_v + c_{uv}$  then
         $c_u = c_v + c_{uv}$ 
         $\text{suc}[u] = v$ 
        if  $u \notin Q$  and  $u \neq s$  then insert( $u, Q$ )
      end if
    end for
  end while
end function

```

3.4 L'algoritmo *Priority* - SHT

Ad ogni iterazione dell'algoritmo, il nodo estratto da Q è v tale che

$$v : c_v = \min\{c_{v_i} | v_i \in Q\}$$

Questa tecnica di selezione gode della seguente proprietà, garantita dal fatto che i costi assumono solo valori non negativi:

Proprietà di monotonia. sia v_1, v_2, \dots, v_h la sequenza ordinata dei nodi estratti da Q : ogni nodo verrà estratto una e una sola volta ($h = |V|$) e i costi associati sono tali che

$$c_{v_k} \leq c_{v_{k+1}} , k = 1, \dots, |V| - 1$$

Oltre a garantire una riduzione del numero di estrazioni dalla coda (e di conseguenza una riduzione del numero di operazioni), questa proprietà permette di aggiornare il costo dei nodi fermata senza dover eseguire ogni volta la procedura *attractive-set*: la coda di priorità garantisce che le linee in $FS(v), v \in F$, siano esplorate in ordine di costo non decrescente (la stessa condizione richiesta da *attractive-set*). In questo modo è possibile aggiornare di volta in volta il costo atteso corrente (come già visto nella procedura) fino a quando la linea successiva ha un costo che supera o eguaglia il costo atteso corrente.

Tale procedura (da ora in avanti *priority-sht*) è definita dallo pseudocodice seguente:

```

function PRIORITY-SHT( $s, t, \theta$ )
  for each  $u \in V$  do
     $c_u = \infty$ 
     $\text{suc}[u] = \text{nil}$ 
    if  $u \in F$  then  $\phi_u = 0$  end if
  end for

   $c_t = 0$ 
   $\text{insert}(t, Q)$ 

  while  $Q \neq \emptyset$  do
     $v = \text{extract}(v, Q)$ 
    for each  $(u, v) \in BS(v)$  do
      if  $u \in F$  and  $c_u > c_v$  then
        if  $\phi_u = 0$  then
           $\phi_u = \varphi_v$ 
           $c_u = \frac{\theta}{\varphi_v} + c_v$ 
        else
           $\phi_u = \phi_u + \varphi_v$ 
           $c_u = c_u - \frac{(c_u - c_v)\varphi_v}{\phi_u}$ 
        end if
         $\text{suc}[u] = \text{suc}[u] \cup v$ 
        if  $u \notin Q$  and  $u \neq s$  then  $\text{insert}(u, Q)$ 
      else if  $u \in N$  and  $c_u > c_v + c_{uv}$  then
         $c_u = c_v + c_{uv}$ 
         $\text{suc}[u] = v$ 
        if  $u \notin Q$  and  $u \neq s$  then  $\text{insert}(u, Q)$ 
      end if
    end for
  end while
end function

```

3.5 Esempio di determinazione dell'iperalbero di costo minimo

Vedremo un esempio di applicazione dell'algoritmo con priorità sull'ipergrafo in esame, caratterizzato dai costi in fig. 3.2:

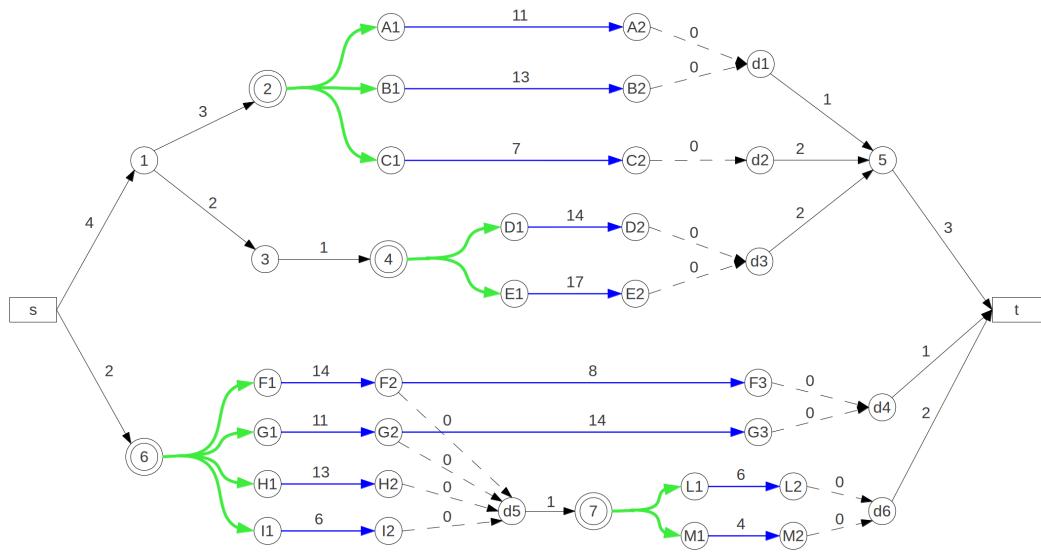


Figura 3.2: Ipergrafo H

e dalle frequenze orarie

$$\{\varphi_A, \varphi_B, \varphi_C, \varphi_D, \varphi_E, \varphi_F, \varphi_G, \varphi_H, \varphi_I, \varphi_L, \varphi_M\} = \{6, 2, 6, 4, 5, 5, 2, 4, 3, 3, 3\}$$

L'algoritmo prevede di inizializzare i costi attesi a ∞ , i successori a *nil* e le frequenze combinate dei nodi fermata a 0, quindi di assegnare costo atteso pari a 0 al nodo t e inserirlo in coda.

Pertanto al passo 0 l'iperalbero sarà come quello mostrato in fig. 3.3:

Le prime n iterazioni riguardano solo nodi in N ; per ciascuna viene indicato il nodo estratto da Q e i suoi predecessori inseriti in seguito ad un aggiornamento del costo (riportato tra parentesi):



Figura 3.3: Iperalbero dopo l'inizializzazione dell'algoritmo

1. $Q \rightarrow t$, $Q \leftarrow 5(3)$, $d4(1)$, $d6(2)$
2. $Q \rightarrow d4$, $Q \leftarrow F3(1)$, $G3(1)$
3. $Q \rightarrow G3$, $Q \leftarrow G2(15)$
4. $Q \rightarrow F3$, $Q \leftarrow F2(9)$
5. $Q \rightarrow d6$, $Q \leftarrow L2(2)$, $M2(2)$
6. $Q \rightarrow M2$, $Q \leftarrow M1(6)$
7. $Q \rightarrow L2$, $Q \leftarrow L1(8)$
8. $Q \rightarrow 5$, $Q \leftarrow d1(4)$, $d2(5)$, $d3(5)$
9. $Q \rightarrow d1$, $Q \leftarrow A2(4)$, $B2(4)$
10. $Q \rightarrow B2$, $Q \leftarrow B1(17)$

11. $Q \rightarrow A2$, $Q \leftarrow A1(15)$
12. $Q \rightarrow d3$, $Q \leftarrow D2(5)$, $E2(5)$
13. $Q \rightarrow E2$, $Q \leftarrow E1(22)$
14. $Q \rightarrow D2$, $Q \leftarrow D1(19)$
15. $Q \rightarrow d2$, $Q \leftarrow C2(5)$
16. $Q \rightarrow C2$, $Q \leftarrow C1(12)$

La coda Q a questo punto contiene i nodi seguenti, in ordine di costo:

$$Q = \{M1, L1, F2, C1, G2, A1, B1, D1, E1\}$$

e l'iperalbero finora determinato è quello riportato in figura 3.4

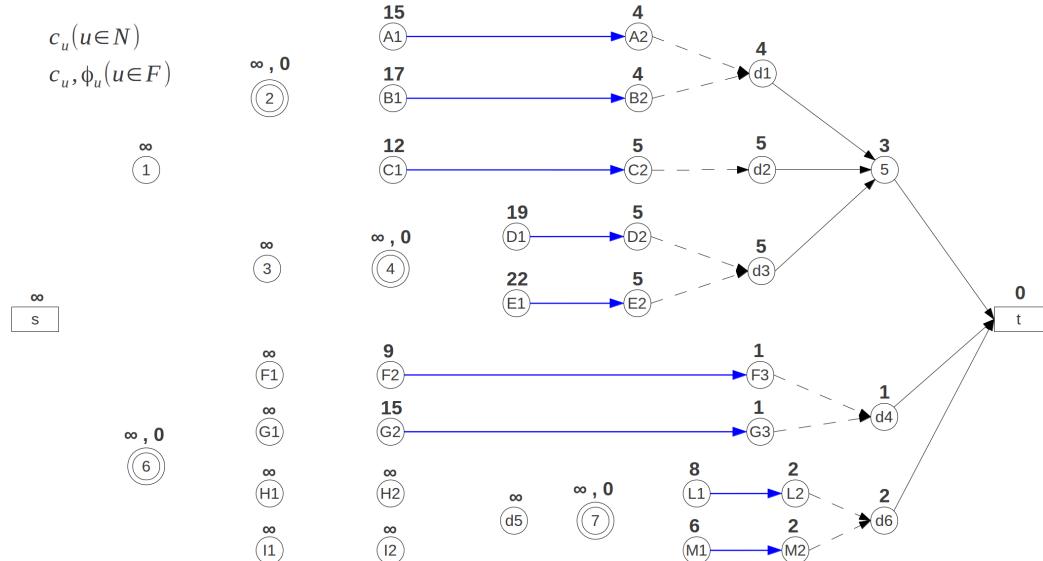


Figura 3.4: Iperalbero dopo 16 iterazioni

L'iterazione 17 prevede l'estrazione del nodo $M1$, il cui unico nodo nella stella

entrante è $7 \in F$: poichè esso viene visitato per la prima volta (è verificata la condizione $\phi_7 = 0$) la sua frequenza combinata e costo vengono aggiornati con la regola seguente:

$$\phi_7 = \varphi_M = 3 \quad , \quad c_7 = \frac{\theta}{\varphi_M} + c_{M1} = 16$$

e il nodo 7 viene inserito in Q (dopo aver posto $suc[7] = M1$).

L'iterazione successiva (18) riguarda il nodo $L1$, avente anch'esso il nodo fermata 7 come suo predecessore. Poichè $c_{L1} < c_7$ e $\phi_7 \neq 0$, vengono modificati frequenza combinata, costo atteso e insieme attrattivo:

$$\begin{aligned} \phi_7 &= \phi_7 + \varphi_L = 6 \\ c_7 &= c_7 - \frac{(c_7 - c_{L1})\varphi_L}{\phi_7} = 12 \\ suc[7] &= suc[7] \cup L1 = \{L1, M1\} \end{aligned}$$

Poichè stavolta il nodo 7 è già presente nella coda, non viene nuovamente inserito (ma l'ordine in cui sarà estratto rispecchierà la modifica del costo).

Le iterazioni successive (19, 20, 21), corrispondenti ai nodi $F2, 7, C1$ procedono in modo analogo alle iterazioni esaminate finora. E' opportuno prestare attenzione all'iterazione 22, in cui viene estratto il nodo $d5$: nella sua stella entrante compaiono, oltre ai nodi non ancora visitati $H2$ e $I2$, anche i nodi già etichettati $F2$ e $G2$.

Per il primo vale $c_{F2} = 9 < c_{d5} + c_{F2,d5} = 13$, pertanto non si procede ad una modifica del costo né ad un nuovo inserimento in coda (sottolineo che il nodo $F2$ è già stato estratto una volta, nell'iterazione 19).

Nel caso di $G2$, invece, vale che $c_{G2} = 15 < c_{d5} + c_{G2,d5} = 13$, e viene posto $c_{G2} = 13$ e $suc[G2] = d5$ (il nodo è già presente in coda e non viene reinserito ma semplicemente considerato con il costo aggiornato).

La situazione è presentata in fig. 3.5:

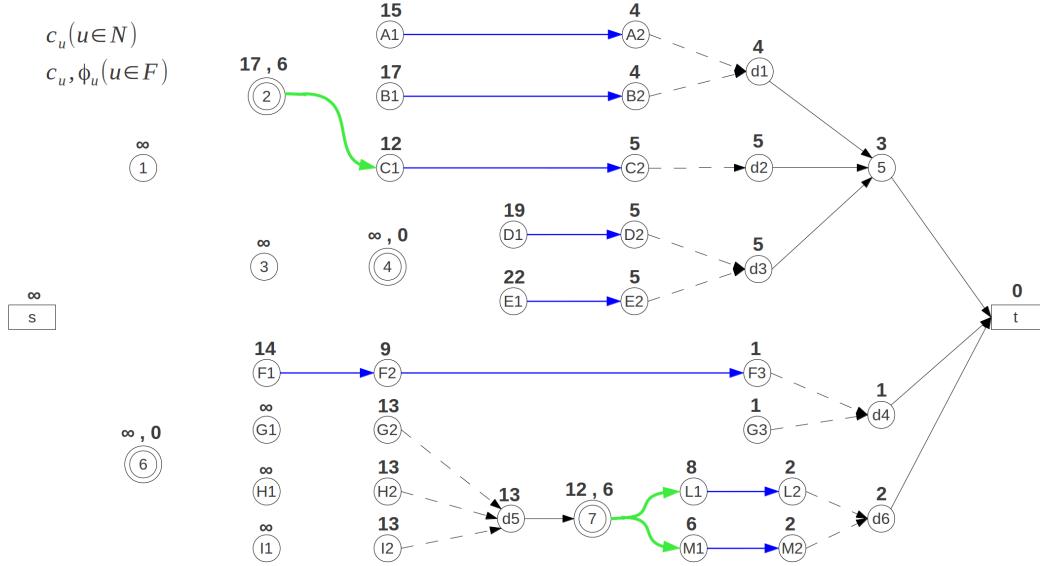


Figura 3.5: Iperalbero dopo 22 iterazioni

La coda contiene ora i nodi (in ordini non crescente di costo)

$$Q = \{G2, H2, I2, F1, A1, B1, 2, D1, E1\}$$

Ora che sono stati forniti esempi dei diversi scenari possibili, vediamo direttamente l'iperalbero costruito alla fine dell'algoritmo:

Il costo con cui ciascun nodo v_i è etichettato equivale al costo dell'ipercammino minimo da v_i a t ; è cioè il costo atteso per giungere dal nodo v_i alla destinazione t .

Partendo dal nodo s e espandendo ricorsivamente i successori di ciascun nodo è possibile ricavare l'ipercammino di costo minimo da s a t . Nell'esempio analizzato, tale ipercammino (da ora denotato con p^*) ha costo complessivo 23 ed è mostrato in figura 3.7. Sui nodi sono riportati i costi attesi prodotti dall'algoritmo (per i nodi fermata anche la frequenza combinata), mentre ciascun arco è etichettato con il costo e la probabilità di essere percorso (condizionata dal fatto di essere in corrispondenza di uno dei suoi nodi coda).

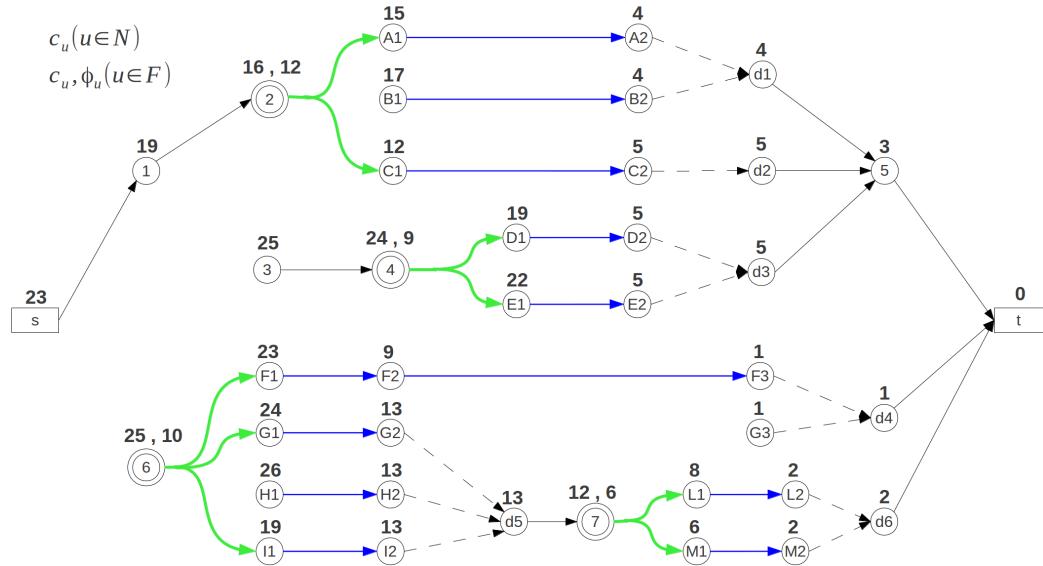


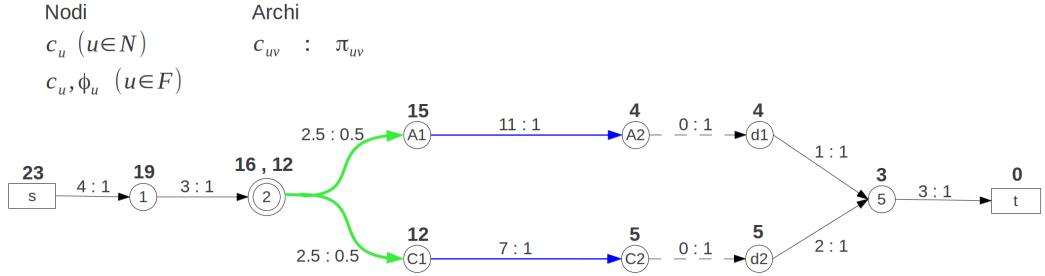
Figura 3.6: Iperalbero dopo la fine dell'algoritmo

E' opportuno ricordare che ciascun ramo di un iperarco di salita ha costo pari al tempo di attesa associato alla fermata u :

$$w_{p*}(u) = \frac{\theta}{\phi_p * (f)}$$

e che le probabilità degli archi si ottengono con:

$$\pi_p(u, v) = \begin{cases} 1, & (u, v) \in p * \text{ e } u \in N \\ \frac{\varphi_v}{\phi_{p*}(u)}, & (u, v) \in p * \text{ e } u \in F \\ 0, & (u, v) \notin p * \end{cases}$$

Figura 3.7: Ipercammino minimo p^* da s a t

3.5.1 Osservazioni

Si osservi la sequenza dei nodi estratti dall'algoritmo a ciascuna iterazione:

$[t, d4, G3, F3, d6, M2, L2, 5, d1, B2, A2, d3, E2, D2, d2, C2, M1, L1, F2, 7, C1,$

$d5, I2, H2, G2, A1, 2, B1, 1, I1, D1, E1, F1, 4, G1, 6, 3, H1]$

Il numero di iterazioni è pari a $|V| - 1 = 38$: come garantito dalla proprietà di monotonia vista nella sezione precedente, ciascun nodo è stato estratto dalla coda Q un'unica volta (ad eccezione, naturalmente, del nodo sorgente s).

Capitolo 4

Implementazione degli algoritmi

Per l'implementazione degli algoritmi è stato utilizzato il linguaggio **C++**, sfruttando gli strumenti offerti dalle librerie **Boost** per la gestione dei grafi. Nell'Appendice A vengono introdotte le caratteristiche principali di queste librerie insieme ad alcune considerazioni sull'implementazione del modello di ipergrafo analizzato nei capitoli precedenti, pertanto la lettura è fortemente consigliata per la comprensione di questo capitolo.

Dando per scontati i concetti presenti, verranno ora riportate esclusivamente le dichiarazioni basilari.

L'ipergrafo è modellato dalla classe `hypergraph`:

```
#include <boost/graph/adjacency_list.hpp>
typedef boost::adjacency_list<boost::vecS, boost::vecS,
                           boost::bidirectionalS, vertex_info, edge_info> hypergraph;
```

la quale memorizza la lista di adiacenza del grafo in esame.

Le *bundled properties* di vertici e archi sono contenute nelle **struct**:

```
struct vertex_info {
    bool stop_vertex;
    string name, lat, lon;
};

struct edge_info {
    double weight;
    string name;
};
```

Il valore booleano **stop_vertex** indica se un certo nodo u appartiene all'insieme N dei nodi *normali* o all'insieme F dei nodi *fermata*.

Le caratteristiche associate agli archi sono memorizzate in **weight**: per gli archi *di viaggio* tale peso corrisponde al concetto di *costo generalizzato* c_{uv} , mentre per gli archi *di salita* equivale alla frequenza oraria φ_v associata alla linea.

Le altre variabili non vengono direttamente utilizzate nell'algoritmo, hanno la sola funzione di rendere i risultati più facilmente presentabili.

L'oggetto **h** di tipo **hypergraph** su cui opereranno gli algoritmi è memorizzato come attributo (privato) della classe **Hypergraph**, definita nel file *header hypergraph.hpp*.

Gli algoritmi presentati nel capitolo 3, insieme ad altre procedure necessarie al loro funzionamento, sono implementati sotto forma di metodi della classe **Hypergraph**.

```

class Hypergraph {

    private:

        hypergraph h;

        bool load_hypergraph(const char* filename) { ... }

        void export_shp (...) { ... };

        vector<edge_descriptor> get_bstar(vertex_descriptor v) { ... }

        .....

    public:

        void sht(vertex_descriptor s, vertex_descriptor t, ...) { ... }

        void priority_sht(vertex_descriptor s, ...) { ... }

        .....

}

```

Il primo metodo presente, `load_hypergraph`, si occupa della corretta istanziazione del grafo a partire da un file di input.

La sezione 4.1 approfondirà i dettagli relativi alle fasi di input/output.

4.1 Input / Output dei dati

4.1.1 Formato di input dell'ipergrafo

La struttura dell'ipergrafo da importare è rappresentata dall'insieme dei vertici V e da quello degli archi E (intesi come archi semplici o rami degli iperarchi di salita).

Per ogni arco è necessario che sia specificato il nodo di testa e quello di coda.

Il funzionamento degli algoritmi richiede, oltre alla struttura basilare dell'ipergrafo, anche informazioni aggiuntive quali gli `weight` relativi agli archi e gli indicatori `stop_vertex` che consentono il partizionamento $V = N \cup F$. Infine, la fase di input vede coinvolte anche le variabili *superflue* come `name`, `lat` e `lon`.

Il formato del file di testo contenente tali dati è definito come segue:

```

n
id      stop_vertex      lat      lon      name

m
i      j      weight      name

```

I valori `n` e `m` rappresentano rispettivamente il numero di vertici e di archi presenti:

$$n = |V| , \quad m = |E|$$

e sono necessari alla corretta lettura del file.

Il campo `id` che identifica ciascun nodo assume i valori in $[0, .., n - 1]$.

Il peso `weight` associato a un arco ha il significato già introdotto precedentemente.

I campi `i, j` di ciascun arco equivalgono rispettivamente all'`id` del nodo di coda e del nodo di testa relativi a tale arco.

I dati rimanenti, intuitivamente, corrispondono alle informazioni *superflue* già menzionate.

In fig 4.1 è riportata una porzione di un file di input. Si osservi che i vari campi sono separati da un unico spazio (fa eccezione il `name`, che può contenere anche spazi al suo interno); e che l'unica linea bianca presente è quella che delimita la lista dei vertici da quella degli archi. I campi `lat` e `lon`

8					
0	0	x	x	s	
1	1	x	x	1	
2	0	x	x	A 1	
3	0	x	x	A 2	
4	0	x	x	B 1	
5	0	x	x	B 2	
6	0	x	x	d	
7	0	x	x	t	
8					
0	1	2	pedonale_1		
1	2	3	salita_A		
1	4	4	salita_B		
2	3	8	linea_A		
4	5	10	linea_B		
3	6	0	discesa_A		
5	6	0	discesa_B		
6	7	1	pedonale_2		

Figura 4.1: Esempio di file di input

sono stati riempiti con un carattere segnaposto qualunque (in questo esempio `x`).

L'ipergrafo corrispondente al file di input mostrato in 4.1 è presentato in fig. 4.2:

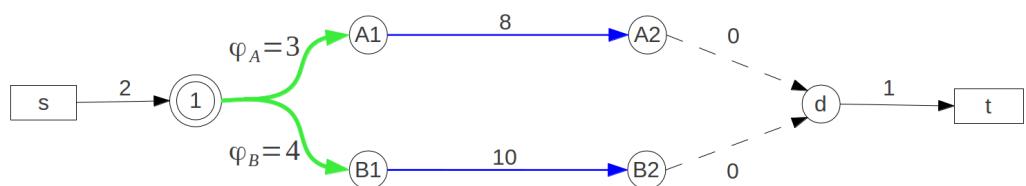


Figura 4.2: Ipergrafo corrispondente all'input di fig 4.1

4.1.2 Parsing del file di input

Il file viene letto utilizzando un `ifstream`:

```
#include <fstream>

bool load_hypergraph(const char* filename) {
    ifstream ifs(filename);
    if (!ifs) return false;
    .....
}
```

Si procede quindi al caricamento dei nodi, dopo aver istanziato l'oggetto `h` della dimensione appropriata. Gli `id` dei nodi vengono impiegati come `vertex_descriptor` per accedere e settare le *bundled properties*:

```
ifs >> n;
h = hypergraph(n);
.....
for(int line=0; line<n; line++) {
    ifs >> id >> stop_vertex >> lat >> lon;
    getline(ifs, vertex_name);
    h[id].stop_vertex = stop_vertex;
    h[id].lat = lat;
    h[id].lon = lon;
    h[id].name = vertex_name;
}
.....
```

A questo punto è possibile importare l'elenco degli archi usando i valori `i`, `j` del file, che come abbiamo visto corrispondono ai descrittori dei nodi:

```

ifs >> m;
for (int line=0;line<m;line++) {
    ifs >> i >> j >> weight;
    getline(ifs , edge_name);
    edge_descriptor e = boost::add_edge(i,j,h).first;
    h[e].weight = weight;
    h[e].name = edge_name;
}
return true;
}

```

Prima di passare alla parte riguardante l'implementazione degli algoritmi verrà approfondito, nella prossima sezione, il problema dell'output dell'algoritmo: in questo modo le variabili coinvolte saranno descritte prima di vedere in che modo vengono elaborate.

4.1.3 Formato di output dello *Shortest HyperPath*

Nelle sezioni 3.3 e 3.4 è stato mostrato l'*iperalbero di costo minimo* prodotto dagli algoritmi *sht* e *priority-sht*.

Da questo iperalbero viene estratto, e quindi memorizzato, l'*ipercammino ottimo* che connette *s* a *t*.

Rimandando alle prossime sezioni la trattazione riguardante l'implementazione degli algoritmi, vedremo ora la questione dell'output dell'ipercammino ottimo da essi determinato.

Riprendiamo ancora una volta come esempio l'ipergrafo *H* finora esaminato: la sezione 3.5 mostrava un esempio di applicazione dell'algoritmo *priority-sht*; al termine della procedura l'*iperalbero ottimo* di *H* (avente radice *t*) era quello mostrato in fig. 4.3 .

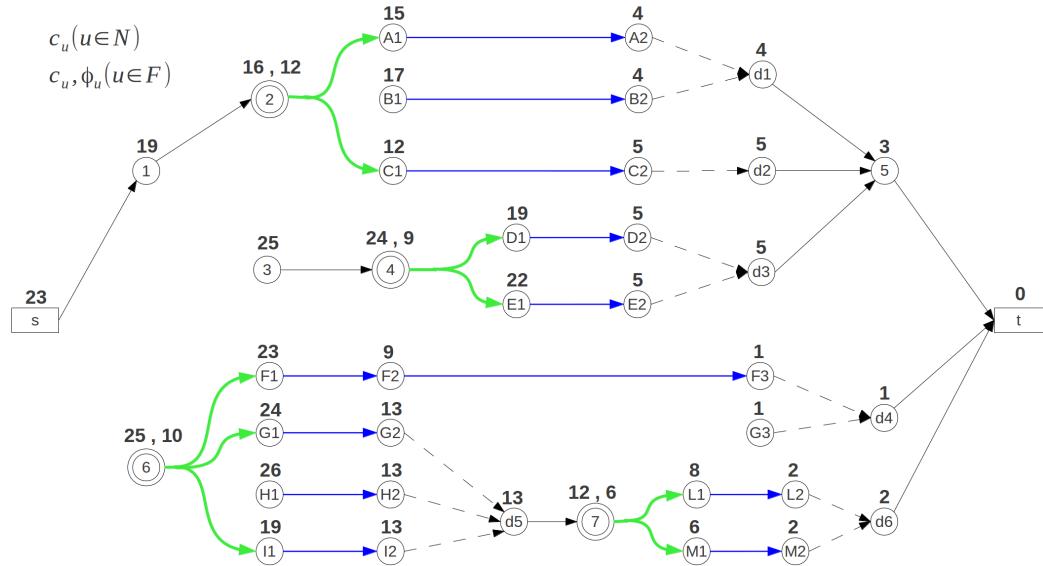


Figura 4.3: Iperalbero ottimo di \$H\$ di radice \$t\$

In pratica, l'algoritmo produce in uscita:

- un vettore dei costi (attesi) necessari per giungere da un certo nodo alla destinazione \$t\$:

$$c[u], \forall u \in V$$

indicato anche come `expected_cost`

- un vettore contenente le frequenze combinate dei nodi fermata:

$$\text{comb_freq}[u], \forall u \in F$$

- il vettore dei successori di ciascun nodo, che ricordo essere definito come

$$\text{suc}[u] = \begin{cases} v = \operatorname{argmin}\{c[v_i] | v_i \in FS(u)\}, & \text{se } u \in N \\ L(u), & \text{se } u \in F \end{cases}$$

indicato per esteso come `successors`

Per determinare l'ipercammino minimo è sufficiente espandere ricorsivamente i **successors** di ciascun nodo a partire dall'origine s .

Con riferimento alla fig. 4.3, l'ipercammino minimo p^* associato è quello mostrato in fig. 4.4.

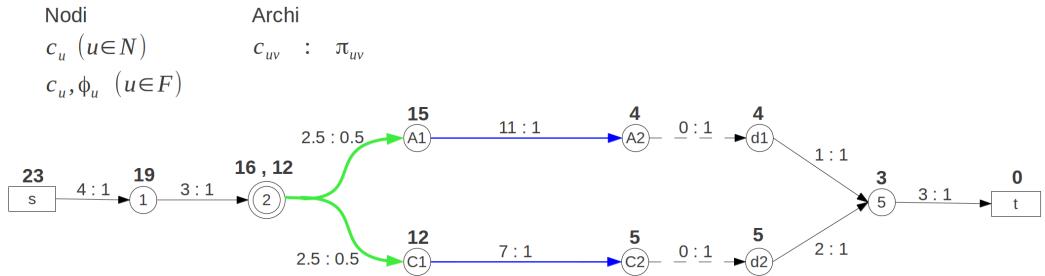


Figura 4.4: Ipercammino minimo p^* da s a t

Le grandezze che lo caratterizzano, oltre a quelle appena elencate, sono i costi e le probabilità associati agli archi.

I costi corrispondono agli **weight** dati in input per quanto riguarda gli archi *di viaggio*, mentre nel caso degli archi *di salita* equivalgono al *tempo di attesa* associato alla fermata:

$$w_{p^*}(u) = \frac{\theta}{\phi_{p^*}(f)}$$

Le probabilità si determinano come

$$\pi_p(u, v) = \begin{cases} 1, & (u, v) \in p^* \text{ e } u \in N \\ \frac{\varphi_v}{\phi_{p^*}(u)}, & (u, v) \in p^* \text{ e } u \in F \\ 0, & (u, v) \notin p^* \end{cases}$$

Riepilogando, le informazioni d'interesse che caratterizzano l'**shp** sono:

- **expected_cost**, **comb_freq** per i vertici (anche i campi **id**, **stop_vertex**, **lat**, **lon** e **name** verranno mantenuti in uscita)

- **cost** e **prob** per gli archi (oltre a **i**, **j**, **name**)

Il formato del file di output è quindi definibile come:

n

id	stop_vertex	lat	lon	expected_cost	comb_freq	name
-----------	--------------------	------------	------------	----------------------	------------------	-------------

m

i	j	cost	prob	name
----------	----------	-------------	-------------	-------------

Ad esempio, il file associato all'ipercammino p^* di fig. 4.4 è illustrato in fig. 4.5.

```

1 1
0 0 x x 23 0 s
2 0 x x 19 0 1
3 1 x x 16 12 2
1 3 0 x x 12 0 C1
1 4 0 x x 5 0 C2
3 4 0 x x 5 0 d2
6 0 x x 3 0 5
1 0 x x 0 0 t
9 0 x x 15 0 A1
1 0 0 x x 4 0 A2
3 3 0 x x 4 0 d1

1 1
0 2 4 1 pedonale_1
2 3 3 1 pedonale_2
3 9 2.5 0.5 salita_A
9 10 11 1 linea_A
10 33 0 1 discesa_A
3 13 2.5 0.5 salita_B
1 3 14 7 1 linea_B
1 4 34 0 1 discesa_B
3 3 6 1 1 pedonale_3
3 4 6 2 1 pedonale_4
6 1 3 1 pedonale_5

```

Figura 4.5: File di output relativo a p^*

4.1.4 Generazione del file di output

Tale metodo ha la funzione di generare il file di output presentato in fig. 4.5 a partire dalle variabili, già elencate nella sezione precedente, che riceve come parametri:

```
void export_shp(vertex_descriptor s, vertex_descriptor t,
    double expected_cost [],
    map<vertex_descriptor, double> comb_freq,
    vector<vertex_descriptor> successors [])
```

Il file di output viene creato e scritto da un **ofstream**:

```
#include <fstream>
.....
ofstream ofs ("shortest_hyper_path_output");
```

Per isolare lo *shortest hyperpath* a partire dallo *shortest hypertree* è sufficiente espandere ricorsivamente la lista dei successori di ciascun nodo a partire dalla sorgente **s**.

Questa funzione è implementata nel metodo **print_successors**:

```
void print_successors(ofstream& ofs, int& shp_v_counter,
    vertex_descriptor v, double expected_cost [],
    map<vertex_descriptor, double> comb_freq,
    vector<vertex_descriptor> successors [], bool in_shp []) {
    if (!in_shp[v]) {
        in_shp[v] = 1;
        shp_v_counter++;
        ofs << v << " " << h[v].stop_vertex << " " << h[v].lat << " "
            << h[v].lon << " " << expected_cost[v] << " " << comb_freq[v]
```

```

    <<"_<<h[v].name<<endl;
for (int i=0; i<successors[v].size(); i++) {
    vertex_descriptor suc = successors[v][i];
    print_successors(ofs, shp_v_counter, suc, expected_cost,
                     comb_freq, successors, in_shp);
}
}
}

```

che viene chiamato all'interno di `export_shp` nel modo seguente:

```

bool in_shp[n];
for (int i=0; i<n; i++)
    in_shp[i] = 0;
int shp_v_counter = 0;
print_successors(ofs, shp_v_counter, s, expected_cost, comb_freq,
                  successors, in_shp);

```

L'array di tipo booleano `in_shp` serve a tener traccia dei nodi già esplorati, allo scopo di evitare inserimenti doppi (un singolo nodo può essere nella lista dei successori di più nodi).

La variabile intera `shp_v_counter` serve semplicemente a ricavare il numero `n` dei vertici appartenenti all'ipercammino minimo; tale valore sarà stampato come prima riga nel file di output.

Passando agli archi, la loro selezione è più immediata. E' sufficiente scorre tutti gli archi dell'ipergrafo `h` e considerare solo quelli aventi sia il nodo di testa (`target`) che quello di coda (`source`) nell'ipercammino minimo (si sfruttano i valori booleani `in_shp` già visti).

Per stampare il valore di `m` corretto, viene usata come contatore la variabile

intera `shp_e_counter`.

Per l’associazione di costo e probabilità a ciascun arco, è necessario distinguere due casi: se il nodo di coda è un nodo in N , allora il costo corrisponde al `weight` già associato a tale arco, e la probabilità vale 1.

Se invece il nodo coda è in F , allora l’arco considerato è un ramo di un *iperrarco di salita*: pertanto il suo costo equivale al tempo di attesa alla fermata ($\frac{\theta}{\phi_u}$), e la sua probabilità è pari a $\frac{\varphi_v}{\phi_u}$.

```

int shp_e_counter = 0;
edge_iterator e_i, e_end;
for(boost::tie(e_i,e_end)=edges(h); e_i!=e_end; ++e_i) {
    vertex_descriptor i, j;
    i = source(*e_i,h);
    j = target(*e_i,h);
    if(in_shp[i] && in_shp[j]) {
        string edge_name = h[*e_i].name;
        double edge_cost;
        double edge_prob;
        if(h[i].stop_vertex) {
            edge_cost = theta / comb_freq[i];
            edge_prob = h[*e_i].weight / comb_freq[i];
        }
        else {
            edge_cost = h[*e_i].weight;
            edge_prob = 1;
        }
        ofs<<i<<"_<<j<<"_<<edge_cost<<"_<<edge_prob<<"_<<edge_name<<endl;
        shp_e_counter++;
    }
}

```

4.2 L'algoritmo sht

Come già anticipato, l'algoritmo è implementato in un metodo della classe `Hypergraph`, che ha accesso all'attributo privato `hypergraph h`.

Tale metodo riceve come input il file contenente l'ipergrafo, e due valori corrispondenti ai `vertex_descriptor` di sorgente e destinazione:

```
void sht(vertex_descriptor s, vertex_descriptor t,
          const char* filename) {
    if (!load_hypergraph(filename)) return;
    .....
```

Una volta caricato l'ipergrafo, si procede all'inizializzazione delle variabili utilizzate (già elencate in 4.1.3):

```
unsigned int n = num_vertices(h);
double expected_cost[n];
for (int k=0; k<n; ++k)
    expected_cost[k] = std :: numeric_limits<double>::infinity ();
map<vertex_descriptor, double> comb_freq;
for (int k=0; k<n; ++k) {
    if(h[k].stop_vertex)
        combined_freq[k] = 0;
}
vector<vertex_descriptor> successors[n];
.....
```

Gli `expected_cost` vengono inizializzati a ∞ , le `comb_freq` dei nodi fermata a 0 e i `successors` a *nil*.

4.2.1 Implementazione della coda Q

La coda Q deve mantenere l'insieme dei nodi da esplorare.

Le operazioni che si eseguono su di essa sono:

- inserimento di nuovi nodi da esplorare
- controllo che tali nuovi nodi non siano già presenti in Q (richiede l'iterazione, nel caso pessimo, di tutti gli elementi in coda)
- estrazione (con rimozione) del prossimo nodo da espandere

La **STL** (*Standard Template Library*) del C++ fornisce una serie di *containers* utilizzabili per questo scopo:

vector : sono implementati come array dinamici. Gli elementi sono allocati in posizioni contigue della memoria, in modo da essere accessibili non solo tramite iteratori ma anche specificando un offset a partire dalla testa del vector. Rispetto agli array sono caratterizzati dalla proprietà di *resize* automatico.

In termini di prestazioni, i punti di forza dei vector comprendono inserimento e rimozione di elementi in coda. Per le operazioni che riguardano inserimento o rimozione di elementi in posizioni diverse dalla fine, sono meno performanti rispetto a *deques* e *lists*.

list : a differenza di *vector* e *queue* non consentono l'accesso diretto agli elementi a partire dalla loro posizione, ma richiedono l'iterazione da una certa posizione (in genere l'inizio) fino alla posizione desiderata. Dall'altro lato, sono migliori per operazioni di inserimento, estrazione e spostamento di elementi in una qualunque posizione per cui si disponga di un iteratore.

deque : sono delle *double-ended queue*; in pratica forniscono le stesse funzionalità di un *vector* ma offrono migliori performance per quanto riguarda inserimento/estrazione dalla testa del container (non solo dalla fine).

E' stato scelto di utilizzare una **deque**, inserendo i nuovi nodi in coda (dopo aver controllato che non siano già presenti) e estraendoli dalla testa. Questo equivale a gestire la coda **Q** con una politica **FIFO** (first in - first out).

```
deque<vertex_descriptor> Q;
expected_cost[ t ] = 0;
Q.push_back( t );
while( !Q.empty() ) {
    vertex_descriptor v = Q.front();
    Q.pop_front();
    .....
```

Il primo nodo inserito (in fondo alla coda) è la destinazione **t**, dopo averne posto il costo atteso a 0.

Ad ogni iterazione, il descrittore del nodo estratto dalla coda viene memorizzato in **v** e i suoi nodi predecessori (quelli associati alla sua *backward-star*) vengono analizzati.

```
vector<edge_descriptor> bstar = get_bstar(v);
for( int k=0; k<bstar.size(); ++k) {
    vertex_descriptor u = source( bstar[k], h );
    .....
```

L'algoritmo procede come mostrato in sezione 3.3, cioè aggiornando il costo di tali nodi se necessario e inserendoli in coda se non presenti.

4.2.2 Aggiornamento del costo dei nodi

E' necessario distinguere i casi $u \in F$ e $u \in N$. Il nodo u deve essere rietichettato con un diverso costo atteso se non sono rispettate le condizioni di Bellman definite in 3.2.

Nel caso di nodi *fermata*, come dimostrato nella sezione 3.2, l'insieme attrattivo deve essere espanso fino a quando il costo della prossima linea non supera o eguaglia il costo atteso attuale della fermata.

In questo caso, poichè le linee non sono esplorate in ordine di costo, è necessario ricalcolare da zero l'insieme attrattivo e le altre variabili da esso dipendenti: l'inserimento di una nuova linea dal costo minore del costo atteso attuale potrebbe comportare l'esclusione di una dal costo maggiore precedentemente inserita nell'insieme attrattivo).

```
if(h[u].stop_vertex && expected_cost[u]>expected_cost[v]) {
    expected_cost[u] = attractive_set(u, expected_cost,
                                       combined_freq[u], successors[u]);
    if(not_in_queue(u, Q) && u!=s)
        Q.push_back(u);
}
```

Il metodo **attractive_set** implementa la procedura definita in 3.2.2: esso restituisce il costo atteso del nodo u e assegna i valori a **comb_freq[u]** e **successors[u]** che riceve come parametri.

```
double attractive_set(vertex_descriptor u, double expected_cost[],
                      double& comb_freq, vector<vertex_descriptor>& successors) {
    vector<edge_descriptor> fstar = get_sorted_fstar(u, expected_cost);
    successors.clear();
    edge_descriptor first_edge = sorted_fstar[0];
```

```

successors.push_back(target(first_edge,h));
comb_freq = h[first_edge].weight;
double expected_cost_u = (theta/comb_freq)
    + expected_cost[target(first_edge,h)];
int index = 1;
bool stop_criterion = false;
while(index<sorted_fstar.size() && stop_criterion==false) {
    edge_descriptor next_edge = sorted_fstar[index];
    vertex_descriptor next_vertex = target(next_edge,h);
    if(expected_cost[next_vertex] < expected_cost_u) {
        successors.push_back(next_vertex);
        comb_freq += h[next_edge].weight;
        expected_cost_u -= ((expected_cost_u -
            expected_cost[next_vertex])*h[next_edge].weight) /
            comb_freq;
        index++;
    }
    else stop_criterion = true;
}
return expected_cost_u;
}

```

Per i vertici in N le operazioni sono più semplici:

```

else if (!h[u].stop_vertex &&
    expected_cost[u] > expected_cost[v]+h[bstar[k]].weight) {
    expected_cost[u] = expected_cost[v] + h[bstar[k]].weight;
    if(successors[u].empty())
        successors[u].push_back(v);
    else
        successors[u][0] = v;
    if(not_in_queue(u, Q) && u!=s)
        Q.push_back(u);
}

```

Il metodo `not_in_queue(u, Q)`, utilizzato in entrambi i casi sopra esposti, si limita a scorrere la coda `Q` e a restituire `false-true` a seconda che il nodo `u` sia già presente in `Q` o meno.

Al termine dell'algoritmo, viene chiamato il metodo `export_shp`:

```
export_shp(s, t, expected_cost, comb_freq, successors);
```

il quale riceve come parametri tutte le variabili necessarie a determinare e memorizzare (secondo le modalità esposte nella sez. 4.1.4) lo *shortest hyperpath* a partire dallo *shortest hypertree* appena ricavato.

4.3 L'algoritmo priority_sht

Valgono le stesse considerazioni già viste nella prima parte della sez. 4.2, riguardante parametri passati, input della rete e inizializzazione delle variabili. La prima differenza rispetto all'algoritmo SHT si presenta, appunto, nell'implementazione della coda `Q`: come mostrato in 3.4, l'algoritmo priority-sht prevede che il nodo estratto a ciascuna iterazione sia quello con il costo atteso minore.

4.3.1 Implementazione della coda `Q`

Le **STL** forniscono un particolare *container adaptor*, chiamato `priority_queue`, che consente di inserire elementi come in un qualsiasi *container* e di estrarre ogni volta quello con *priorità* maggiore, dove tale priorità è valutata da un operatore di confronto definito dall'utente per lo specifico tipo contenuto nella `priority_queue`.

Tuttavia tale struttura non consente un accesso diretto agli elementi, pertanto non è possibile verificare se un certo elemento è già presente.

Per questa ragione, anche per questo algoritmo la coda Q è stata implementata come una `deque`, da cui ogni volta si estraе l'elemento di testa. Per garantire che tale elemento sia quello *a priorità massima* (in questo caso, il vertice con il costo atteso minimo), è necessario che gli elementi vengano inseriti in ordine (non decrescente) di costo.

Questa operazione è svolta dal metodo `priority_insertion`:

```
void priority_insertion(vertex_descriptor u,
    deque<vertex_descriptor>& Q, double expected_cost[])
{
    int pos = 0;
    while(pos < Q.size() && expected_cost[u] > expected_cost[Q[pos]])
        pos++;
    Q.insert(Q.begin() + pos, u);
    pos++;
    while(pos < Q.size()) {
        if(u == Q[pos]) Q.erase(Q.begin() + pos);
        pos++;
    }
}
```

La coda Q viene scorsa fino a trovare la posizione appropriata per il nuovo nodo da inserire.

Si osservi che il nodo potrebbe essere già presente in lista esclusivamente con un costo maggiore di quello con cui viene inserito al passo corrente (grazie alla proprietà di monotonia), pertanto dopo aver effettuato l'inserimento la coda viene esplorata fino in fondo per cercare, ed eventualmente rimuovere, tale nodo duplicato.

4.3.2 Aggiornamento del costo dei nodi

Per i nodi in N le operazioni sono banali ed analoghe a quelle eseguite dall'algoritmo senza priorità.

Per i nodi in F , grazie alla *proprietà di monotonia*, non è necessario fare ricorso al metodo **attractive_set**: come visto in sez. 3.4, il costo atteso delle fermate può essere aggiornato a ciascuna iterazione.

Pertanto ciascun nodo u appartenente alla *backward-star* del nodo estratto verrà analizzato dal codice seguente:

```

if(h[u].stop_vertex && expected_cost[u] > expected_cost[v]) {
    if(comb_freq[u]==0) {
        comb_freq[u] = h[bstar[k]].weight;
        expected_cost[u] = (theta/comb_freq[u])+expected_cost[v];
    }
    else {
        comb_freq[u] += h[bstar[k]].weight;
        expected_cost[u] -= (expected_cost[u]-expected_cost[v])
            *h[bstar[k]].weight/comb_freq[u];
    }
    successors[u].push_back(v);
    if(u!=s)
        priority_insertion(u,Q,expected_cost);
}
else if (!h[u].stop_vertex &&
    expected_cost[u]>expected_cost[v]+h[bstar[k]].weight) {
    expected_cost[u] = expected_cost[v] + h[bstar[k]].weight;
    if(successors[u].empty())
        successors[u].push_back(v);
    else
        successors[u][0] = v;
}

```

```
if ( u!=s )
    priority_insertion (u,Q,expected_cost );
}
```

Anche in questo caso, al termine della procedura viene invocato il metodo `export_shp` per l'output dei risultati.

4.4 Esempio di applicazione

In questa sezione l'algoritmo `priority_sht` verrà applicato all'ipergrafo h finora utilizzato.

I risultati verranno comparati con quelli ricavati manualmente nella sezione 3.5 per lo stesso ipergrafo.

Innanzitutto, è stato creato manualmente il file di input che modella h :

```
39
0 0 x x s
1 0 x x t
2 0 x x 1
3 1 x x 2
.....
```

```
49
0 2 4 pedonale
2 3 3 pedonale
3 9 6 salita
.....
```

Nel `main` dell'applicazione verrà istanziato un oggetto della classe `Hypergraph` su cui richiamare il metodo presentato in 4.3. Tale metodo, come già osservato, riceve il nome del file di input e i descrittori associati ai nodi `s` e `t`:

```
#include "hypergraph.hpp"
```

```
int main() {
    Hypergraph H;
```

```

H.priority_sht(0,1,"example_hypergraph_h");
return 0;
}

```

Come è naturale aspettarsi, il file di output generato (relativo all'ipercammino p^* già osservato) è uguale a quello mostrato in fig. 4.5.

Per una maggior chiarezza sull'esecuzione dell'algoritmo, sono state inserite delle stampe nei punti rilevanti. Il seguente listato mostra la sequenza di inserimenti/estrazioni dalla lista Q a ciascuna iterazione:

1. $Q \rightarrow t$, $Q \leftarrow 5(3) \ d4(1) \ d6(2)$
2. $Q \rightarrow d4$, $Q \leftarrow F3(1) \ G3(1)$
3. $Q \rightarrow G3$, $Q \leftarrow G2(15)$
4. $Q \rightarrow F3$, $Q \leftarrow F2(9)$
5. $Q \rightarrow d6$, $Q \leftarrow L2(2) \ M2(2)$
6. $Q \rightarrow M2$, $Q \leftarrow M1(6)$
7. $Q \rightarrow L2$, $Q \leftarrow L1(8)$
8. $Q \rightarrow 5$, $Q \leftarrow d1(4) \ d2(5) \ d3(5)$
9. $Q \rightarrow d1$, $Q \leftarrow A2(4) \ B2(4)$
10. $Q \rightarrow B2$, $Q \leftarrow B1(17)$
11. $Q \rightarrow A2$, $Q \leftarrow A1(15)$
12. $Q \rightarrow d3$, $Q \leftarrow D2(5) \ E2(5)$
13. $Q \rightarrow E2$, $Q \leftarrow E1(22)$
14. $Q \rightarrow D2$, $Q \leftarrow D1(19)$
15. $Q \rightarrow d2$, $Q \leftarrow C2(5)$
16. $Q \rightarrow C2$, $Q \leftarrow C1(12)$
17. $Q \rightarrow M1$, $Q \leftarrow 7(16)$

18. $Q \rightarrow L1 , Q \leftarrow 7(12)$
19. $Q \rightarrow F2 , Q \leftarrow F1(23)$
20. $Q \rightarrow 7 , Q \leftarrow d5(13)$
21. $Q \rightarrow C1 , Q \leftarrow 2(17)$
22. $Q \rightarrow d5 , Q \leftarrow G2(13) H2(13) I2(13)$
23. $Q \rightarrow I2 , Q \leftarrow I1(19)$
24. $Q \rightarrow H2 , Q \leftarrow H1(26)$
25. $Q \rightarrow G2 , Q \leftarrow G1(24)$
26. $Q \rightarrow A1 , Q \leftarrow 2(16)$
27. $Q \rightarrow 2 , Q \leftarrow 1(19)$
28. $Q \rightarrow B1 , Q \leftarrow$
29. $Q \rightarrow 1 , Q \leftarrow$
30. $Q \rightarrow I1 , Q \leftarrow 6(29)$
31. $Q \rightarrow D1 , Q \leftarrow 4(26.5)$
32. $Q \rightarrow E1 , Q \leftarrow 4(24)$
33. $Q \rightarrow F1 , Q \leftarrow 6(25.25)$
34. $Q \rightarrow 4 , Q \leftarrow 3(25)$
35. $Q \rightarrow G1 , Q \leftarrow 6(25)$
36. $Q \rightarrow 6 , Q \leftarrow$
37. $Q \rightarrow 3 , Q \leftarrow$
38. $Q \rightarrow H1 , Q \leftarrow$

Si nota che la procedura termina in 38 passi, e ciascun nodo è estratto una e una sola volta da Q (ad eccezione di s).

La sequenza delle estrazioni combacia con quella presentata nella sezione 3.5.1.

Capitolo 5

Un esempio reale

Gli algoritmi presentati nel capitolo 4 sono stati applicati ad un ipergrafo modellante un esempio reale di trasporto pubblico: le linee dei bus elettrici che attraversano il centro storico di Firenze.

La zona di interesse è quella mostrata in fig. 5.1:



Figura 5.1: Mappa del centro storico di Firenze

Questa mappa è stata prelevata da openstreetmap.org in un formato di tipo **xml**; tramite alcuni appositi **script** è stato generato l'ipergrafo di fig. 5.2 (i *nodi fermata* sono riportati in rosso).



Figura 5.2: Ipergrafo modellante il centro storico di Firenze. In rosso i nodi fermata.

Le linee di trasporto considerate sono quattro: C1, C2, C3 e D. Per ciascuna di esse, l'itinerario è stato considerato come un unico percorso chiuso che collega le fermate, senza tener conto della distinzione andata/ritorno basata sulla *direzione* del mezzo.

Le frequenze orarie relative a ciascuna linea sono state assunte come indipendenti dall'ora del giorno. I valori sono quelli riportati da ataf.net:

$$\{\varphi_{C1}, \varphi_{C2}, \varphi_{C3}, \varphi_D\} = \{4, 8, 6, 5\}$$

E' stata ipotizzata inoltre una situazione di perfetta regolarità del servizio; poichè i tempi associati agli archi sono espressi in secondi (come vedremo

nella prossima sezione), questa condizione è stata modellata assegnando a θ il valore $\frac{3600}{2} = 1800$.

Le fig. 5.3 e 5.4 mostrano gli *archi a bordo* (in blu) relativi a ciascuna linea.

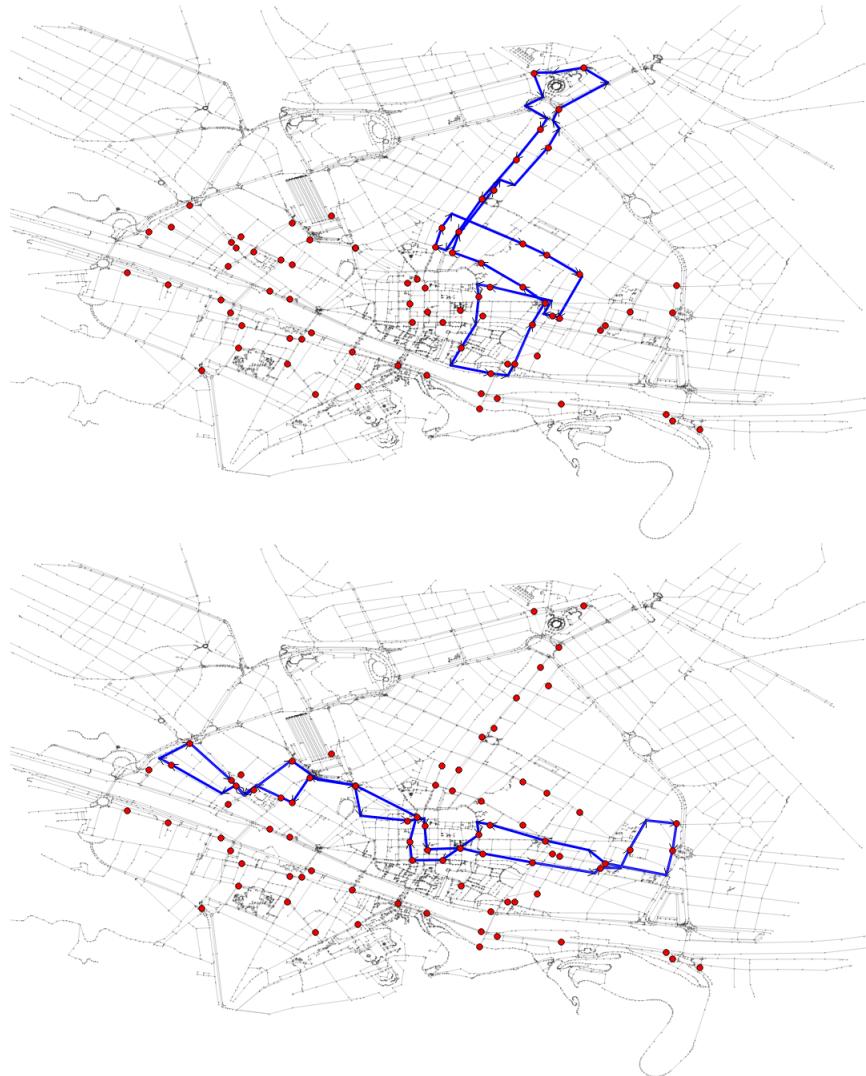


Figura 5.3: Linee C1 e C2

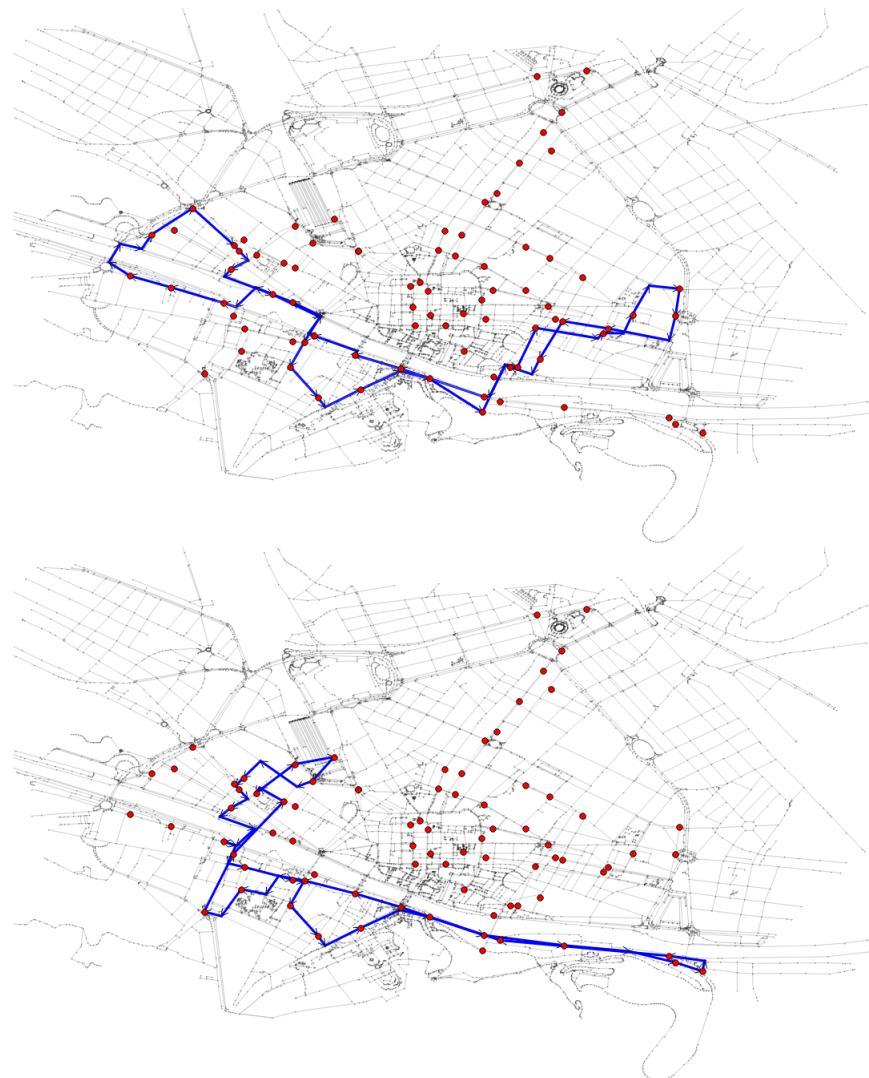


Figura 5.4: Linee C3 e D

5.1 Etichettamento degli archi

Per gli archi *di salita*, il *weight* associato corrisponde alla frequenza oraria relativa alla linea.

Gli archi *normali*, come illustrato nella sezione 2.2.1, vengono etichettati con un *costo generalizzato* dipendente dalle loro caratteristiche rilevanti.

Per gli archi *di discesa* il *weight* (che in questo caso dovrebbe simboleggiare il

costo del biglietto o la scomodità di viaggiare su un mezzo pubblico) è stato posto a 0.

Gli archi *pedonali* e *a bordo* sono stati etichettati con il relativo tempo di percorrenza, ricavato a partire dalla lunghezza (in metri) e considerando le velocità medie seguenti:

$$\begin{cases} 1.2 \frac{m}{s} \left(\sim 4.3 \frac{km}{h}\right), & \text{per gli archi pedonali} \\ 7 \frac{m}{s} \left(\sim 25 \frac{km}{h}\right), & \text{per gli archi a bordo} \end{cases}$$

5.2 Esempi

L'algoritmo utilizzato sull'ipergrafo di Firenze è `priority_sht` (sez. 4.3), e i risultati ottenuti saranno esportati nel formato definito nella sezione 4.1.3.

A partire dai risultati vengono generati, tramite `script` appositi, degli `shapefile` che consentono la presentazione grafica degli ipercammini determinati.

5.2.1 Esempio 1

Consideriamo come *origine* un nodo pedonale corrispondente a Piazza Beccaria e come *destinazione* uno associato a Porta al Prato.

Il risultato è quello mostrato in fig. 5.5: l'ipercammino minimo determinato coinvolge le linee C2 e C3, raffigurate rispettivamente in rosso e in azzurro. Le linee in nero corrispondono ad archi pedonali.

Una volta raggiunta a piedi la fermata **Beccaria**, il cui insieme attrattivo è costituito appunto dalle linee C2 e C3, l'utente associato a tale ipercammino resta in attesa del primo veicolo disponibile per un tempo medio pari a

$$w(Beccaria) = \frac{\theta}{\varphi_{C2} + \varphi_{C3}} \sim 2 \text{ minuti e } 10 \text{ secondi}$$

Indipendentemente dalla linea utilizzata, l'utente scenderà alla fermata **Leopolda** da cui raggiungerà a piedi la destinazione.

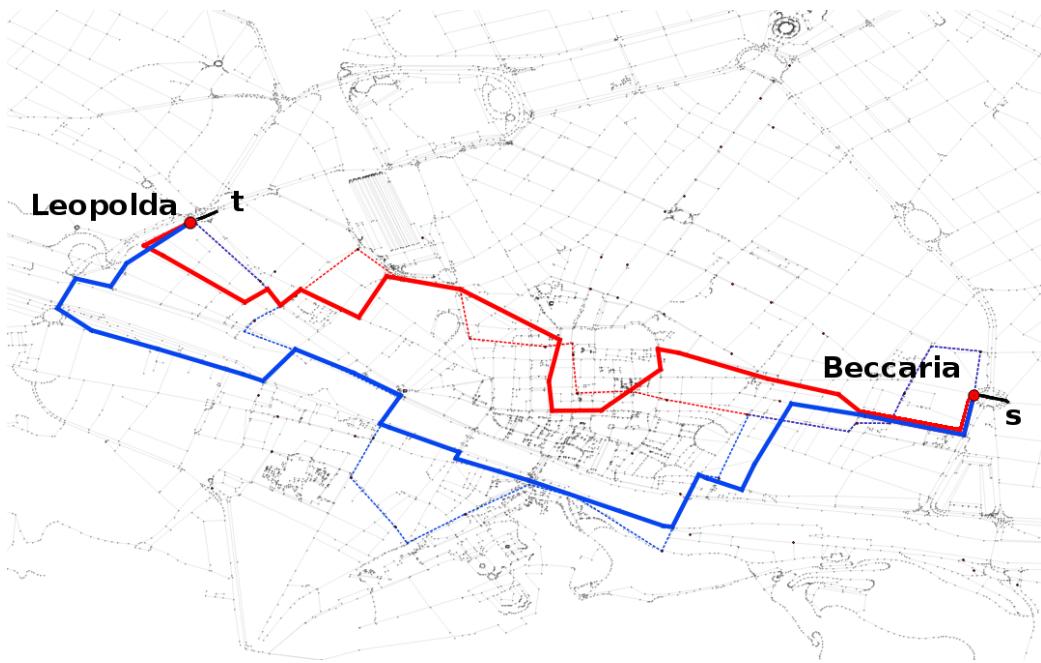


Figura 5.5: Ipercammino minimo da *Piazza Beccaria* a *Porta al Prato*

Il costo atteso di tale ipercammino è pari a

$$c = 897 \text{ secondi} \sim 15 \text{ minuti}$$

Una schematizzazione dell'ipercammino determinato è presentata in fig. 5.6, seguendo le convenzioni grafiche introdotte nel capitolo 1.

In essa sono riportate anche le probabilità condizionate degli iperarchi.

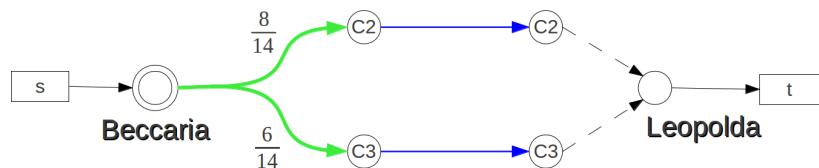


Figura 5.6: Schematizzazione dell'ipercammino minimo di fig. 5.5

Denotiamo i due cammini di cui è costituito come:

$$\begin{cases} a : \text{Beccaria - C2 - Leopolda} \\ b : \text{Beccaria - C3 - Leopolda} \end{cases}$$

Le loro probabilità di realizzazione, ricavabili con l'espressione 2.6 (definita in sezione 2.3.1), sono pari a:

$$\begin{cases} \Omega(a) = \frac{8}{14} \sim 0.57\% \\ \Omega(b) = \frac{6}{14} \sim 0.43\% \end{cases}$$

5.2.2 Esempio 2

In questo esempio l'obiettivo è trovare l'ipercammino minimo che collega Piazza Pitti a Piazza della Libertà.

Le linee interessate sono C1, C3 e D, rappresentate rispettivamente in verde, blu e rosso nella fig. 5.7.

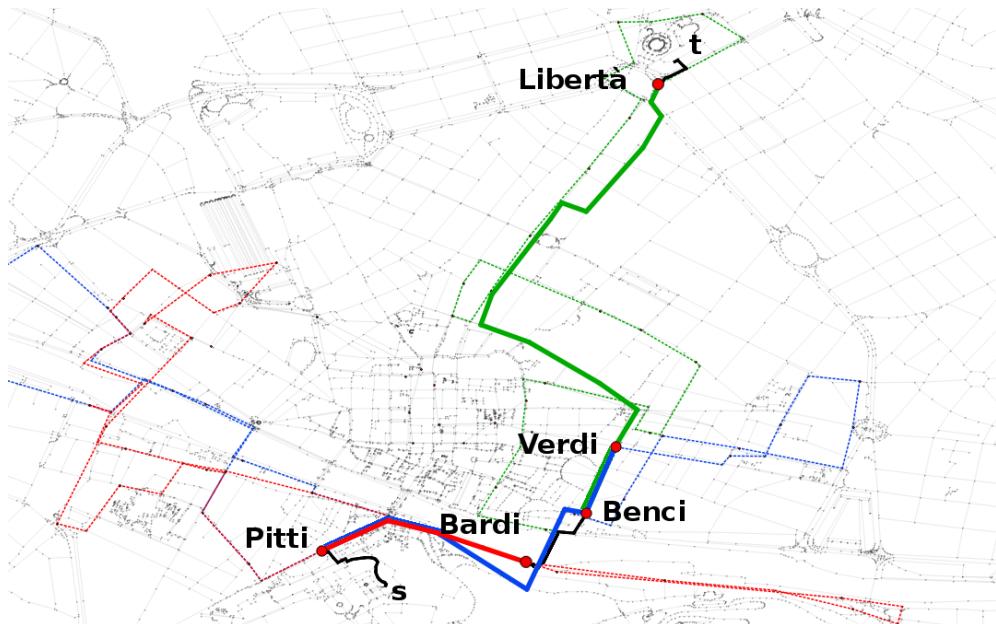


Figura 5.7: Ipercammino minimo da *Piazza Pitti* a *Piazza della Libertà*

Il primo nodo fermata incontrato è **Pitti**: l'insieme attrattivo ad esso associato è costituito dalle linee C3 e D, quindi il tempo di attesa medio alla fermata è dato da

$$w(Pitti) = \frac{\theta}{\varphi_{C3} + \varphi_D} \sim 2 \text{ minuti e } 45 \text{ secondi}$$

Se il primo mezzo transitato alla fermata è della linea C1, il passeggero prosegue il tragitto a bordo fino alla fermata **Verdi**; nel caso in cui invece l'utente sia salito su un veicolo della linea D, egli dovrà scendere alla fermata immediatamente successiva (**Bardi**) e proseguire a piedi fino alla fermata **Benci**. In corrispondenza di tale fermata l'utente salirà sul primo mezzo disponibile tra quelli della linea C1 e C3, dopo un'attesa media di

$$w(Benci) = \frac{\theta}{\varphi_{C1} + \varphi_{C3}} = 3 \text{ minuti}$$

Se il primo mezzo disponibile è della linea C1 l'utente proseguirà la marcia a bordo fino alla fermata **Libertà**, in corrispondenza del nodo **t** desiderato.

Se invece il primo veicolo ad arrivare è della linea C3, il passeggero proseguirà fino alla fermata successiva (**Verdi**).

Una volta in prossimità di tale nodo (indipendentemente da quale cammino sia stato percorso per giungervi), l'utente resta in attesa di un mezzo della linea C1 per un tempo medio di

$$w(Verdi) = \frac{\theta}{\varphi_{C1}} = 7 \text{ minuti e } 30 \text{ secondi}$$

Il costo atteso dell'ipercammino ammonta a

$$c = 1468 \text{ secondi} \sim 24 \text{ minuti e } 30 \text{ secondi}$$

Analogamente a quanto fatto per l'esempio 1, in fig. 5.8 è riportata una schematizzazione dell'ipercammino minimo, su cui sono indicate anche le probabilità condizionate relative agli iperarchi di salita.

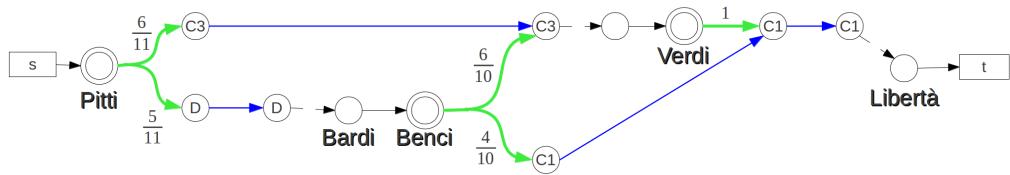


Figura 5.8: Schematizzazione dell'ipercammino minimo di fig. 5.7

In questo caso i cammini che compongono l'ipercammino minimo sono tre; denotiamoli come segue:

$$\begin{cases} a : \text{Pitti} - \text{C3} - \text{Verdi} - \text{C1} - \text{Libertà} \\ b : \text{Pitti} - \text{D} - \text{Bardi} - \text{Benci} - \text{C3} - \text{Verdi} - \text{C1} - \text{Libertà} \\ c : \text{Pitti} - \text{D} - \text{Bardi} - \text{Benci} - \text{C1} - \text{Libertà} \end{cases}$$

Le probabilità di realizzare tali cammini sono:

$$\begin{cases} \Omega(a) = \frac{6}{11} \cdot 1 \sim 0.55\% \\ \Omega(b) = \frac{5}{11} \cdot \frac{6}{10} \cdot 1 \sim 0.27\% \\ \Omega(c) = \frac{5}{11} \cdot \frac{4}{10} \sim 0.18\% \end{cases}$$

(naturalmente $\Omega(a) + \Omega(b) + \Omega(c) = 1$).

Appendice A

Uso delle librerie boost

Le librerie **Boost** sono un insieme di librerie che estendono le funzionalità del C++. Le ultime versioni contano oltre 80 librerie individuali (molte delle quali rilasciate con una licenza di tipo open-source), fornendo un'ampia gamma di campi di applicazione.

Sono basate sulla programmazione generica per consentire (tramite lo strumento dei *template* del C++) flessibilità e riutilizzo del codice, garantendo allo stesso tempo la massima efficienza possibile.

In particolare, questo lavoro fa uso della libreria **BGL** (Boost Graph Library).

A.1 *Boost Graph Library*

Le **BGL** dispongono di una serie di strumenti utili a modellare in maniera efficiente i grafi, e una serie di algoritmi ad essi applicabili.

Un aspetto rilevante di questa libreria è la *genericità*, caratterizzata da tre aspetti principali:

- **Interoperabilità di algoritmi e strutture dati**

Per permettere l'accesso agli elementi del grafo sono definite delle interfacce indipendenti dal tipo di struttura dati utilizzata per rappresentare il grafo stesso.

Nello specifico, sono presenti tre diversi *pattern* per l'attraversamento di un grafo: uno basato sull'accesso a tutti i nodi, uno che permette l'attraversamento di tutti gli archi, e un terzo che permette di esplorare la struttura di adiacenza del grafo (esplorare, a partire da un vertice, i nodi adiacenti).

Grazie a questa interfaccia generica, gli algoritmi possono essere scritti e utilizzati astraendo dalla particolare struttura dati scelta per implementare il grafo.

Le **BGL** forniscono alcuni tra i principali algoritmi *generici* tra quelli relativi alla teoria dei grafi: ricerca in ampiezza / profondità, determinazione di cammini minimi (Dijkstra, Bellman-Ford), determinazione di componenti connesse...

- **Estensione attraverso *Visitors***

Un *visitor* equivale a un function-object (o *funtore*) con metodi multipli; associato ad un algoritmo, permette di estenderlo grazie all'invocazione di metodi appositi per il grafo utilizzato.

- **Multi-parametrizzazione di nodi e archi**

In molti casi, compreso quello del lavoro in oggetto, è necessario associare parametri multipli a nodi e archi. Le **BGL** permettono questa multi-parametrizzazione con due diverse tecniche, che approfondiremo più avanti: le *bundle properties* e le *property map*.

Le **BGL** dispongono essenzialmente di due classi per la modellazione di grafi: **adjacency_list** e **adjacency_matrix**.

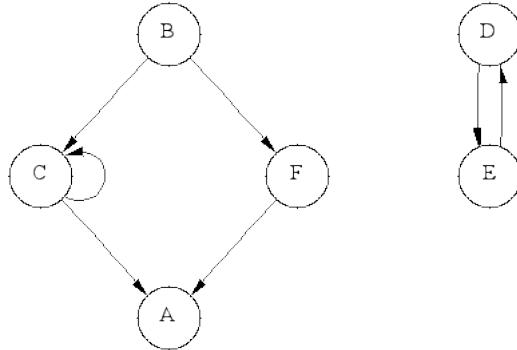


Figura A.1: Grafo diretto G

Riepiloghiamone brevemente le caratteristiche salienti:

Matrice di adiacenza : dato un grafo orientato, la sua matrice di adiacenza è definita come una matrice binaria quadrata di dimensione $|V| \times |V|$, il cui elemento (i, j) vale 1 se e solo se è presente un arco diretto da i a j . Gli altri elementi valgono 0.

La memoria richiesta per la memorizzazione di tale matrice è $O(v^2)$, e l'attraversamento di tutti gli archi uscenti da ciascun nodo ha una complessità computazionale pari a $O(V^2)$.

Lista di adiacenza : è una lista di coppie (n, L) , dove L è la lista di nodi adiacenti al nodo n -esimo.

Richiede una quantità di memoria pari a $O(|V| + |E|)$ e permette l'attraversamento degli archi uscenti da ciascun nodo in un tempo $O(|V| + |E|)$.

Il confronto evidenzia come la matrice di adiacenza sia preferibile solo nei casi di grafi *densi*, in cui $|E| \simeq V^2$.

Le fig. A.2 e A.3 mostrano rispettivamente matrice e lista di adiacenza associate al grafo di fig. A.1.

	A	B	C	D	E	F
A	0	0	0	0	0	0
B	0	0	1	0	0	1
C	1	0	1	0	0	0
D	0	0	0	0	1	0
E	0	0	0	1	0	0
F	1	0	0	0	0	0

Figura A.2: Matrice di adiacenza

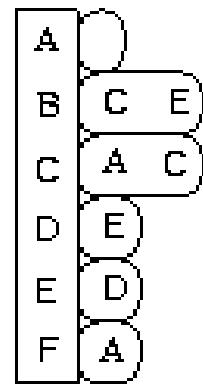


Figura A.3: Lista di adiacenza

Vista la conformazione della rete di trasporto considerata in questo lavoro, modellata da ipergrafi tendenzialmente sparsi, la scelta è ricaduta sulla lista di adiacenza.

Nelle librerie **Boost** la classe **adjacency_list** considerata ha la struttura seguente:

```
adjacency_list<OutEdgeList, VertexList, Directed,
VertexProperties, EdgeProperties>
```

Intuitivamente, il grafo è rappresentato da un insieme di vertici cui è abbinato un insieme di archi uscenti. Per accedere agli elementi di questi insieme si usano degli appositi indici detti *descrittori*: **vertex_descriptor** e **edge_descriptor**.

Vediamo in dettaglio i template che compaiono nella dichiarazione:

- *OutEdgeList* e *VertexList* indicano le strutture dati usate per mantenere l’insieme di archi e vertici. La scelta (che ricade principalmente su *vector* o *list*) influenza sui tempi di inserimento / rimozione / accesso agli elementi.

- *Directed* specifica l’orientamento del grafo. Può assumere i valori *undirected*, *directed* o *bidirectional* (quest’ultimo, rispetto a *directed*, aggiunge la possibilità di utilizzare iteratori sugli archi entranti).
- *VertexProperties* e *EdgeProperties* servono a specificare le *Bundle Properties* associate a vertici e archi.

Nell’insieme totale degli elementi del grafo è possibile distinguere cinque sottoinsiemi: *vertices*, *adjacent-vertices*, *edges*, *in-edges* e *out-edges*.

Per accedere a tali sottoinsiemi esistono degli opportuni *iteratori*:

- `vertex_iterator` serve a muoversi tra i vertici del grafo (siano essi tutti i *vertices* del grafo o solo gli *adjacent-vertices* di un vertice specifico), restituendo tramite dereferenziazione un `vertex_descriptor`.
- `edge_iterator` itera tra gli archi del grafo restituendo il rispettivo `edge_descriptor`.
- `in_edge_iterator` e `out_edge_iterator` restituiscono rispettivamente gli archi entranti/uscenti di determinato nodo.

Abbiamo già accennato alla possibilità di associare le proprietà ad archi e vertici in due modalità distinte; ora che sono stati introdotti alcuni importanti concetti, approfondiamo questi due approcci:

- le *Bundle Properties* sono memorizzate direttamente come attributi della classe o struttura C++ utilizzata per rappresentare i vertici (o archi).
- le *Property Map* consentono di associare proprietà *esterne* al grafo memorizzandole in vettori separati.

A.2 Modellazione dell'ipergrafo

L'ipergrafo sui cui opereranno gli algoritmi implementati è un oggetto del tipo `hypergraph` così definito:

```
#include <boost/graph/adjacency_list.hpp>
typedef boost::adjacency_list<boost::vecS, boost::vecS,
                           boost::bidirectionalS, vertex_info, edge_info> hypergraph;

hypergraph h;
```

Si osservi che ai template *OutEdgeList* e *VertexList* è stato assegnato il valore `boost::vecS`, il che equivale a scegliere dei `std::vector` per mantenere gli insiemi di vertici e nodi. Poichè gli algoritmi presentati non richiedono la rimozione di elementi del grafo o l'inserimento in posizioni intermedie, questa struttura è preferibile rispetto ad una `std::list` perchè garantisce una maggior velocità nell'inserimento in coda di nuovi elementi e nell'accesso ad un elemento in posizione random.

Il valore `boost::bidirectionalS`, come precedentemente introdotto, indica che il tipo di grafo definito è orientato ed è possibile iterare anche gli archi entranti in ciascun nodo.

Infine, le strutture C++ utilizzate per rappresentare le proprietà interne (*Bundle Properties*) degli elementi del grafo sono `vertex_info` e `edge_info` così definite:

```
struct vertex_info {
    bool stop_vertex;
    string name, lat, lon;
};
```

```
struct edge_info {
    double weight;
    string name;
};
```

Considerando i vertici, l'unico attributo di reale interesse è **stop_vertex**: se pari a 0 indica che il vertice in oggetto appartiene all'insieme dei nodi normali N (*pedonale, di linea o di discesa*), mentre un valore pari a 1 distingue i nodi *fermata* in F (da cui escono solo *iperarchi di salita*).

I campi **name**, **lat** e **lon** non vengono manipolati dall'algoritmo, ma utilizzati esclusivamente per consentire una migliore interpretazione visuale dei risultati (le coordinate **lat** e **lon** sono memorizzate come stringhe per evitare problemi di arrotondamento nelle fasi di input/output).

Per quanto riguarda gli archi, **weight** viene usato come parametro unico per i valori associati ai vari tipi di archi: nel caso di archi *pedonali, a bordo* o *di discesa* corrisponde al *costo generalizzato*; nel caso di *iperarchi di salita*, invece, tale attributo non esprime il costo associato (che in questo caso sarebbe equivalente al *tempo di attesa*) ma la *frequenza oraria* che caratterizza la linea corrispondente.

Per il campo **name** valgono le stesse considerazioni riguardanti i vertici.

A.2.1 Accesso a vertici e archi

Verranno utilizzati i seguenti *descrittori* e *iteratori*:

```
typedef hypergraph::vertex_descriptor vertex_descriptor;
typedef hypergraph::edge_descriptor edge_descriptor;

typedef hypergraph::vertex_iterator vertex_iterator;
typedef hypergraph::edge_iterator edge_iterator;
typedef hypergraph::out_edge_iterator out_edge_iterator;
typedef hypergraph::in_edge_iterator in_edge_iterator;
```

Vediamo, per illustrare un caso d'uso degli iteratori, il codice del metodo `get_bstar`.

Tale metodo riceve come parametro un `vertex_descriptor` e restituisce uno `std::vector` contenente gli `edge_descriptor` degli archi appartenenti alla stella entrante nel nodo.

```
vector<edge_descriptor> get_bstar(vertex_descriptor v) {
    vector<edge_descriptor> bstar;
    in_edge_iterator ie_i, ie_end;
    for(boost::tie(ie_i, ie_end)=in_edges(v,h); ie_i!=ie_end; ++ie_i)
        bstar.push_back(*ie_i);
    return bstar;
}
```

Se a questo punto fosse necessario stampare i nomi dei nodi della stella entrante, l'accesso potrebbe essere fatto come segue:

```
for (int k=0; k<bstar.size(); ++k) {
    vertex_descriptor u = source(bstar[k], h);
    cout << h[u].name << endl;
}
```

Intuitivamente, il metodo

```
source(edge_descriptor e, hypergraph h)
```

restituisce il **vertex_descriptor** corrispondente al nodo di testa (*source*) dell'arco (o iperarco) **e** nell'ipergrafo **h**.

Analogamente, per ottenere il descrittore del nodo di coda si può utilizzare il metodo

```
target(edge_descriptor e, hypergraph h).
```

A questo punto, per poter accedere ai dati contenuti nelle bundle properties è sufficiente utilizzare l'operatore '[.]' come se l'ipergrafo **h** fosse un normale vettore.

In questo modo si ottiene un riferimento alla struttura associata al descrittore, e l'accesso ai dati della struttura avviene come di consueto con l'operatore '[.]'.

Conclusioni

Lo scopo di questo lavoro era quello di approfondire e formalizzare il problema della modellazione, tramite ipergrafi orientati, delle reti del trasporto pubblico locale.

In particolare, l'attenzione è stata rivolta al problema della determinazione di ipercammini minimi su tali ipergrafi.

Tale problema è risolvibile sia riducendo l'ipergrafo ad un grafo tradizionale equivalente, sia enumerando tutti gli ipercammini possibili.

Si è invece scelto di sviluppare una procedura *Dijkstra-like* che esplori iterativamente i nodi dell'ipergrafo (partendo dalla destinazione) per generare un iperalbero di costo minimo.

Tale algoritmo (*Shortest HyperPath*) è stato sviluppato in due versioni: la versione *con priorità*, che prevede di esplorare i nodi in ordine non decrescente di costo, offre le migliori prestazioni computazionali.

La fase implementativa ha portato alla realizzazione di uno strumento direttamente utilizzabile su ipergrafi modellanti reti di trasporto, come mostrato nel capitolo 5.

I risultati ottenuti e gli algoritmi messi a disposizione possono essere utilizzati come punto di partenza per algoritmi più complessi e sviluppi futuri.

Ad esempio, una questione interessante legata agli ipergrafi è il problema del-

l’assegnazione del flusso. Nel modello del trasporto pubblico locale, l’obiettivo è ripartire i flussi di passeggeri in modo da avere uno stato di *equilibrio*.

Uno degli algoritmi più efficienti in questo campo (metodo di *Frank & Wolfe*) richiede, come passo base, la determinazione degli ipercammini minimi relativi ad una destinazione.

Sarà quindi possibile approfondire la trattazione riguardante queste particolari strutture topologiche senza dover ripartire dall’implementazione dei metodi di basso livello.

Bibliografia

- [1] PALLOTTINO, S., NGUYEN, S., GALLO, G., AND LONGO, G. *Directed hypergraphs and applications*. Dipartimento di Informatica, Università di Pisa, 1990.
- [2] PALLOTTINO, S., AND SCHETTINO, A. *Scienze delle decisioni per i trasporti. Il trasporto collettivo urbano*. Franco Angeli, 1999.
- [3] PALLOTTINO, S., AND NGUYEN, S. *Equilibrium traffic assignment for large scale transit networks*. European Journal of Operational Research 37 (1988), 176-186.
- [4] PALLOTTINO, S., AND NGUYEN, S. *Implicit enumeration of hyperpaths in a logit model for transit networks*. Transportation Science Journal 32 (1998), 54-64.
- [5] FLORIAN, M., AND MARCOTTE, P. *Transit equilibrium assignment: a model and solution algorithms*. Transportation Science Journal 28 (1994), 193-196.
- [6] BUGGIANI, D. *Algoritmi innovativi per problemi di equilibrio di traffico su ipergrafi* Tesi specialistica, Dipartimento di sistemi e informatica, Università degli studi di Firenze, 2010.

- [7] AHUJA, R., MAGNANTI, T., AND ORLIN, J. *Network flows*, Prentice Hall, 1993.
- [8] STROUSTRUP, B. *The C++ Programming Language: Special Edition*, 3 ed. Addison-Wesley Professional, February 2000.
- [9] *Boost Graph Library 1.46* (boost.org)
- [10] *OpenStreetMap Project* (openstreetmap.org)
- [11] *OSMLib Ruby Library* (osmlib.rubyforge.org)
- [12] *rubyshapelib Ruby Library* (sourceforge.net/projects/ruby-shapelib)
- [13] ataf.net