

# INTRODUZIONE ALLA RICERCA OPERATIVA

## PARTE II TEORIA DEI GRAFI

Lecture Notes, v. 9.0  
19 novembre 2023

Daniele Vigo<sup>1</sup>  
Marco Antonio Boschetti<sup>2</sup>

Alma Mater Università di Bologna

<sup>1</sup> Dip. di Ingegneria dell’Energia Elettrica e  
dell’Informazione “Guglielmo Marconi” e CIRI-ICT

<sup>2</sup> Dip. di Matematica



# Indice

<b>1 Teoria dei Grafi</b>	<b>1</b>
1.1 Introduzione . . . . .	1
1.1.1 Applicazioni . . . . .	3
1.1.2 Taglio di un grafo . . . . .	4
1.1.3 Cammini, circuiti e cicli . . . . .	5
1.1.4 Grafi parziali, sottografi e componenti . . . . .	8
1.1.5 Alberi . . . . .	9
1.1.6 Rappresentazione dei grafi . . . . .	10
1.2 Cammini minimi . . . . .	14
1.2.1 Formulazione matematica . . . . .	15
1.2.2 Assunzioni . . . . .	15
1.2.3 Distance label . . . . .	16
1.2.4 Condizioni di ottimalità . . . . .	17
1.2.5 Algoritmi Label Correcting . . . . .	18
1.2.6 Algoritmo di Dijkstra . . . . .	19
1.2.7 Algoritmo di Floyd-Warshall . . . . .	25
1.3 Alberi di copertura . . . . .	28
1.3.1 Applicazioni . . . . .	28
1.3.2 Formulazione matematica . . . . .	30
1.3.3 Condizioni di ottimalità . . . . .	31
1.3.4 Algoritmo di Prim-Dijkstra . . . . .	33
1.3.5 Algoritmo di Kruskal . . . . .	39
1.4 Flusso massimo . . . . .	43
1.4.1 Formulazione matematica . . . . .	43
1.4.2 Assunzioni e definizioni . . . . .	44
1.4.3 Condizioni di ottimalità . . . . .	45
1.4.4 Algoritmo di Ford-Fulkerson . . . . .	47
1.5 Flusso di costo minimo . . . . .	52
1.5.1 Assunzioni e definizioni . . . . .	54
1.5.2 Condizioni di ottimalità . . . . .	55
1.5.3 Cycle Cancelling Algorithm . . . . .	55
1.5.4 Successive Shortest Path Algorithm . . . . .	56
1.5.5 Primal-Dual Algorithm . . . . .	58
1.5.6 Out-of-Kilter Algorithm . . . . .	62

1.5.7	Algoritmi polinomiali . . . . .	63
1.5.8	Algoritmi Fortemente Polinomiali . . . . .	64
1.6	Network Simplex Method . . . . .	65
1.6.1	Struttura Spanning Tree . . . . .	66
1.6.2	Implementazione della Struttura Spenning Tree . . . . .	67
1.6.3	Spanning Tree Fortemente Ammissibili . . . . .	73

# Capitolo 1

## Teoria dei Grafi

### 1.1 Introduzione

Un *grafo non orientato*, rappresentato come  $G = (V, E)$ , è definito dall'insieme dei *vertici* (o *nodi*) e dall'insieme dei *lati* che congiungono coppie non ordinate di vertici. In particolare:

- $V = \{1, 2, \dots, n\}$ : insieme dei vertici (o nodi);
- $E = \{e_1, e_2, \dots, e_m\}$ : insieme dei lati, che corrispondono a coppie *non ordinate* di vertici di  $V$  che sono *collegati*, i.e., un lato  $e_k = \{i, j\}$  collega i vertici  $i$  e  $j$ .

Due vertici sono *adiacenti* se esiste il lato che li collega, mentre due lati sono *consecutivi* se hanno un vertice in comune. Risulterà molto utile denotare con  $E(S)$  l'insieme dei lati con entrambi gli estremi nel sottoinsieme di vertici  $S \subseteq V$  e  $\Gamma(i)$  insieme dei vertici collegati a  $i$ .

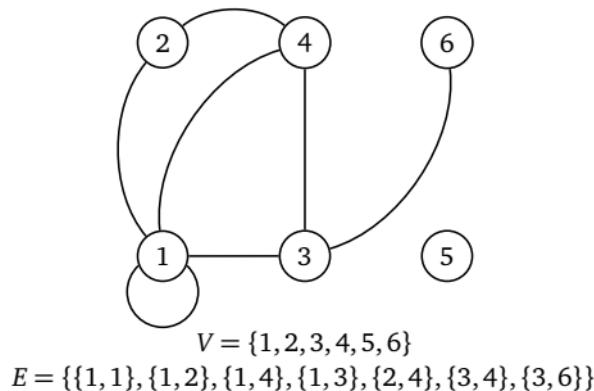


Figura 1.1: Esempio di grafo non orientato.

In figura 1.1 è riportato un grafo non orientato, in cui, per esempio, i vertici 3 e 4 sono collegati con il lato  $e = \{3, 4\}$ , e quindi sono adiacenti, mentre il vertice 5 non è collegato con nessun altro vertice. In quest'ultimo caso si dice che il vertice 5 *non è connesso*. In figura 1.1 si può osservare che, per esempio, i lati  $e = \{3, 4\}$  e  $e' = \{3, 6\}$  sono consecutivi. Inoltre, il grafo ha un *loop* (anche detto *autoanello* o *cappio*) in corrispondenza del vertice 1, in quanto esiste il lato  $\{1, 1\}$ . Se si considera il sottoinsieme di vertici  $S = \{1, 2, 4\}$  allora  $E(S) = \{\{1, 1\}, \{1, 2\}, \{1, 4\}, \{2, 4\}\}$ . Inoltre, per esempio,  $\Gamma(2) = \{1, 4\}$ ,  $\Gamma(4) = \{1, 2, 3\}$ ,  $\Gamma(5) = \emptyset$ .

Un grafo è detto *multiplo* se ha più di un lato per la stessa coppia di vertici. Un grafo non orientato è detto *semplice* se non comprende loop e archi multipli. Generalmente considereremo solo grafi semplici. Un grafo è *completo* se per ogni coppia di vertici esiste un lato. In Figura 1.2 sono proposti degli esempi di grafo multiplo, semplice e completo.

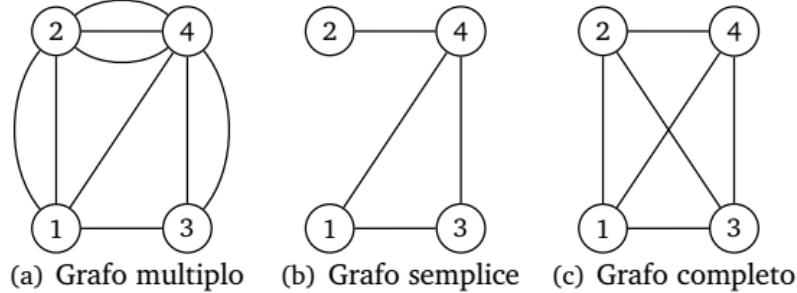


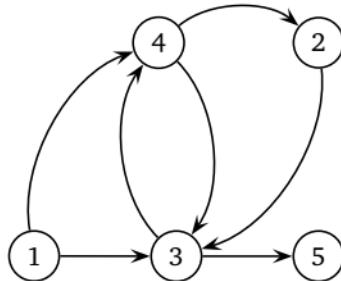
Figura 1.2: Esempi di grafo multiplo, semplice e completo.

Un *grafo orientato* (o *grafo diretto*) si differenzia da un grafo non orientato per la sostituzione dell'insieme dei lati con l'insieme degli *archi*, che sono coppie *ordinate* di vertici. Quindi, per un grafo orientato  $G = (V, A)$  avremo che:

- $V = \{1, 2, \dots, n\}$ : insieme dei vertici (o nodi);
- $A = \{a_1, a_2, \dots, a_m\}$ : insieme degli archi, che corrispondono a coppie *ordinate* di vertici di  $V$ , i.e., l'arco  $a_k = (i, j)$  indica che il vertice  $i$  è collegato al vertice  $j$ , ma il vertice  $j$  non è collegato al vertice  $i$  a meno che non esista un altro arco  $a_h = (j, i)$ .

Dato l'arco  $(i, j)$  il vertice  $i$  è detto *vertice iniziale* (o *testa*) e  $j$  è detto *vertice terminale* (o *coda*). Inoltre, il vertice  $j$  è anche detto *successore* di  $i$  mentre  $i$  è detto *predecessore* di  $j$ .

Per i grafi orientati sarà utile denotare con  $A(S)$  l'insieme degli archi con entrambi gli estremi (vertice iniziale e finale) nel sottoinsieme di vertici  $S \subseteq V$  e con  $\Gamma^+(i)$  e  $\Gamma^-(i)$  gli insiemi dei successori e dei predecessori di  $i$ , rispettivamente.



$$V = \{1, 2, 3, 4, 5\} \quad A = \{(1, 4), (1, 3), (3, 4), (4, 3), (4, 2), (2, 3), (3, 5)\}$$

Figura 1.3: Esempio di grafo orientato.

In figura 1.3 è riportato un esempio di grafo in cui, per esempio, l'arco  $(1, 4)$  *esce* dal vertice 1 ed *entra* nel vertice 4. Quindi il vertice 4 può essere raggiunto dal vertice 1, ma non può accadere il contrario, ossia dal vertice 4 non si può tornare al vertice 1. Se consideriamo il sottoinsieme di vertici  $S = \{1, 3, 4\}$ , allora  $A(S) = \{(1, 4), (1, 3), (3, 4), (4, 3)\}$ . Inoltre, per esempio,  $\Gamma^+(1) = \{3, 4\}$ ,  $\Gamma^+(4) = \{2, 3\}$ ,  $\Gamma^+(5) = \emptyset$ , mentre  $\Gamma^-(1) = \emptyset$ ,  $\Gamma^-(4) = \{1, 3\}$ ,  $\Gamma^-(5) = \{3\}$ .

Un grafo  $G$  non orientato (orientato) è pesato sui lati (archi) se esiste una funzione  $c : E \rightarrow \mathbb{R}$  ( $c : A \rightarrow \mathbb{R}$ ) che associa un *peso* ad ogni lato (arco). Il peso può rappresentare un costo, una distanza, un tempo di percorrenza, etc. Invece, il grafo  $G$  è pesato sui vertici se esiste una funzione  $w : V \rightarrow \mathbb{R}$  che associa un *peso* ad ogni vertice. In figura 1.4 è riportato un esempio di grafo non orientato pesato.

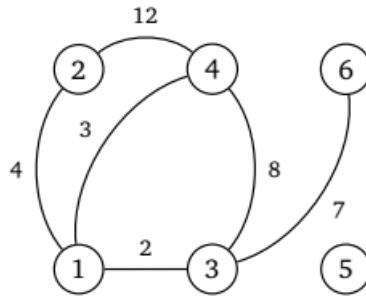


Figura 1.4: Esempio di grafo non orientato pesato.

### 1.1.1 Applicazioni

Le applicazioni della teoria dei grafi possono spaziare dalla teoria alla pratica.

Tra gli argomenti più noti nell'ambito della teoria dei grafi possiamo citare ad esempio:

- Cammini Euleriani: originato dal problema posto da Eulero, per determinare un percorso che, partendo da una qualsiasi delle quattro zone della città di Könisberg, attraversasse tutti i sette ponti una ed una sola volta ritornando al punto di partenza.
- Colorazione dei grafi: dove un esempio di applicazione e la colorazione delle mappe per garantire di non usare lo stesso colore per nazioni confinanti.
- Problema della clique (cricca): per esempio per calcolare la clique (i.e., sottografo completo) di cardinalità massima.

Per quanto riguarda le applicazioni nel mondo reale, in figura 1.5 sono riportati alcuni esempi proposti dal *MIT Center for Transportation & Logistics* in cui viene fornita anche qualche informazione su come il problema può essere modellato con i grafi. Come si può osservare, le applicazioni possono spaziare dalla progettazione e ottimizzazione di reti di comunicazione fino alle reti di trasporto.

Applications	Physical Analog of Nodes	Physical Analog of Arcs	Flow
Communication systems	Telephone exchanges, computers, transmission facilities, satellites	Cables, fiber optic links, microwave relay links	Voice messages, data, video transmissions
Hydraulic systems	Pumping stations, reservoirs, lakes	Pipelines	Water, gas, oil, hydraulic fluids
Integrated computer circuits	Gates, registers, processors	Wires	Electrical current
Mechanical systems	Joints	Rods, beams, springs	Heat, energy
Transportation systems	Intersections, airports, rail yards	Highways, railbeds, airline routes	Passengers, freight, vehicles, operators

Figura 1.5: Esempi di alcune applicazioni (tabella proposta dal *MIT Center for Transportation & Logistics*).

### 1.1.2 Taglio di un grafo

Dato un sottoinsieme  $S$  di vertici, si dice *taglio* l'insieme dei lati (o archi) che congiungono i vertici in  $S$  con quelli in  $V \setminus S$ .

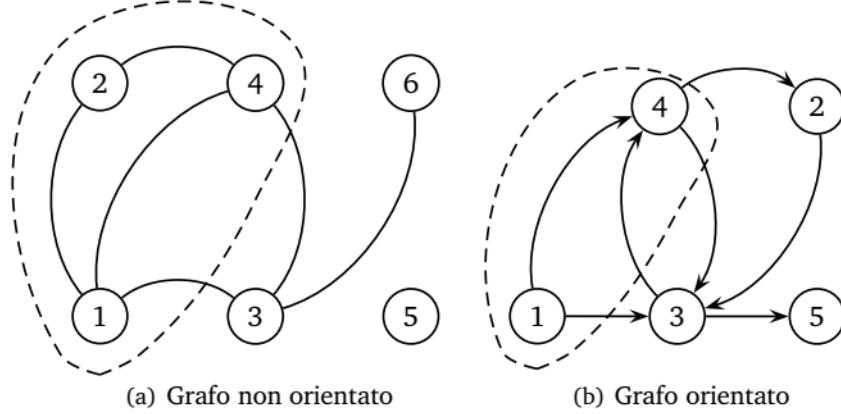


Figura 1.6: Esempio di taglio in un grafo non orientato e in uno orientato.

Nella figura 1.6 sono riportatati due esempi di taglio in un grafo non orientato e in un grafo orientato.

Per i grafi non orientati si può rappresentare il taglio come:

$$\delta_G(S) = \{\{i, j\} \in E : i \in S, j \in V \setminus S \text{ oppure } j \in S, i \in V \setminus S\}$$

e nell'esempio di figura 1.6.a sarebbe  $\delta_G(\{1, 2, 4\}) = \{(1, 3), (3, 4)\}$ .

Per i grafi orientati invece si deve distinguere tra archi uscenti ed entranti in  $S \subset V$ :

- $\delta_G^+(S) = \{(i, j) \in A : i \in S, j \notin S\};$
- $\delta_G^-(S) = \{(i, j) \in A : j \in S, i \notin S\}.$

Si noti che  $\delta_G^+(S) \equiv \delta_G^-(V \setminus S)$ . Per l'esempio di figura 1.6.b si ha  $\delta_G^+(\{1, 4\}) = \{(1, 3), (4, 2), (4, 3)\}$  e  $\delta_G^-(\{1, 4\}) = \{(3, 4)\}$ .

### 1.1.3 Cammini, circuiti e cicli

Un cammino è una sequenza di vertici  $v_1, v_2, \dots, v_k \in V$  tale che per ogni coppia di vertici consecutivi  $(v_i, v_{i+1})$  esiste il corrispondente lato (grafo non orientato) o arco (grafo orientato). Quindi, un cammino  $P$  si può rappresentare sia come una sequenza di vertici:

$$P = (v_1, v_2, v_3, \dots, v_k)$$

Però, è possibile rappresentare un cammino anche come una sequenza archi (o lati):

$$P = ((v_1, v_2), (v_2, v_3), (v_3, v_4), \dots, (v_{k-1}, v_k))$$

In figura 1.7 è proposto un grafo orientato e sono specificati alcuni esempi di cammini presenti in esso. In figura 1.8 è stato evidenziato il cammino  $P_1$ .

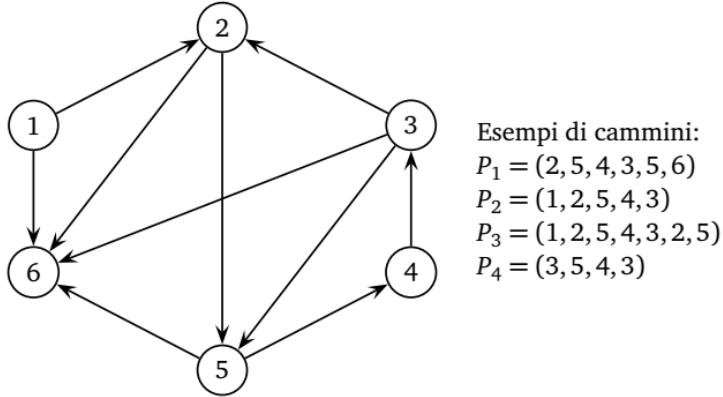
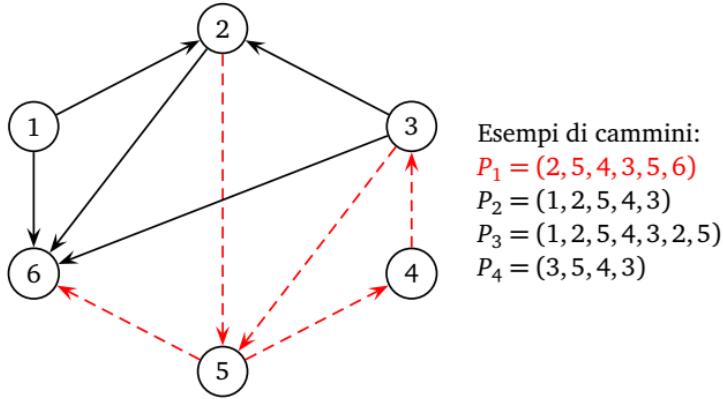


Figura 1.7: Esempio di cammini in un grafo orientato.

Figura 1.8: Esempio di cammini in un grafo orientato, in cui è evidenziato  $P_1$ .

Dato un cammino  $P = (v_1, v_2, v_3, \dots, v_k)$  il suo *costo*  $c(P)$  è dato da :

$$c(P) = \sum_{i=1}^{k-1} c_{v_i v_{i+1}}$$

dove  $c_{ij}$  è il costo dell'arco  $(i, j)$ . Il costo di un cammino, a seconda del contesto e dell'applicazione, è anche detto *lunghezza*, *peso*, etc.

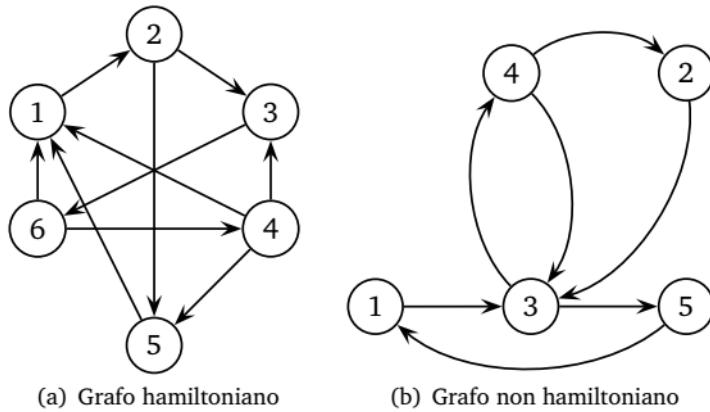
Tra i diversi cammini in un grafo possiamo identificare i seguenti casi particolari di notevole interesse:

- *Cammino semplice*: non usa più di una volta lo stesso arco/lato.
- *Cammino elementare*: non passa più di una volta per lo stesso vertice.
- *Cammino hamiltoniano*: usa una ed una sola volta tutti i vertici del grafo; quindi deve visitare tutti vertici del grafo.

- *Cammino euleriano*: usa una ed una sola volta tutti gli archi/lati del grafo.
- *Circuito*: è un cammino in un grafo orientato in cui il vertice iniziale coincide con il vertice terminale.
- *Circuito elementare*: è un circuito che, a parte il primo e l'ultimo vertice (che coincidono), non passa più di una volta per lo stesso vertice.
- *Circuito hamiltoniano*: è un circuito elementare che passa attraverso ogni vertice del grafo. Oppure, equivalentemente, è un cammino hamiltoniano chiuso (i.e., con un arco che collega l'ultimo vertice con il primo del cammino).
- *Circuito euleriano*: è un circuito elementare che passa attraverso ogni arco del grafo. Oppure, equivalentemente è un cammino euleriano chiuso.
- *Ciclo*: controparte non orientata di un circuito.

Nel grafo in figura 1.7  $P_1$ ,  $P_2$  e  $P_4$  sono cammini semplici, mentre  $P_3$  non lo è. Invece, mentre  $P_2$  è un cammino elementare,  $P_1$ ,  $P_3$  e  $P_4$  non lo sono. Nel grafo in figura 1.7 non ci sono cammini hamiltoniani ed euleriani e lo si può determinare piuttosto facilmente osservando che il vertice 1 ha solo archi uscenti e il vertice 6 ha solo archi entranti. Considerando sempre il grafo in figura 1.7 il cammino  $P = (2, 5, 4, 3, 2)$  è un circuito, che è anche elementare.

I grafi che possiedono almeno un circuito/ciclo hamiltoniano sono detti *grafi hamiltoniani*. Invece, i grafi che possiedono almeno un circuito/ciclo euleriano sono detti *grafi euleriani*. Non tutti i grafi possiedono un circuito/ciclo hamiltoniano o euleriano. Nella figura 1.9 è proposto l'esempio di due grafi, dove uno è hamiltoniano mentre l'altro non lo è.



Circuito elementare (a)  $C_1 = (1, 2, 3, 6, 1)$ , (b)  $C_2 = (3, 4, 2, 3)$ .

Circuito hamiltoniano (a)  $C_3 = (1, 2, 3, 6, 4, 5, 1)$ , (b) non ne possiede.

Figura 1.9: Esempio di grafo hamiltoniano e non hamiltoniano.

Un grafo che non contiene circuiti (cicli) è detto *grafo aciclico*. In figura 1.10 è riportato un esempio.

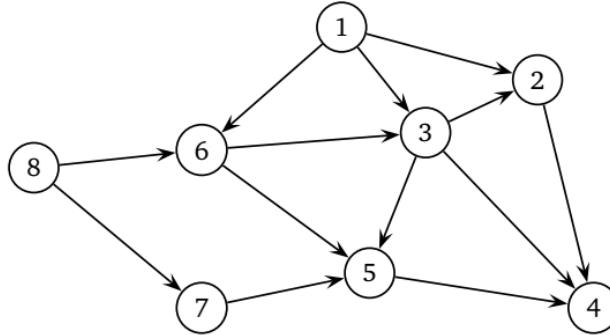


Figura 1.10: Esempio di grafo aciclico.

#### 1.1.4 Grafi parziali, sottografi e componenti

Dato un grafo  $G = (V, A)$ , un suo *grafo parziale* è il grafo  $G' = (V, A')$  dove  $A' \subset A$ . Invece, un *sottografo* di  $G = (V, A)$  è il grafo  $G' = (V', A')$  dove  $V' \subseteq V$  e  $A' \subseteq A$ .

In figura 1.11 è proposto un grafo orientato e sono proposti un suo grafo parziale e suo sottografo.

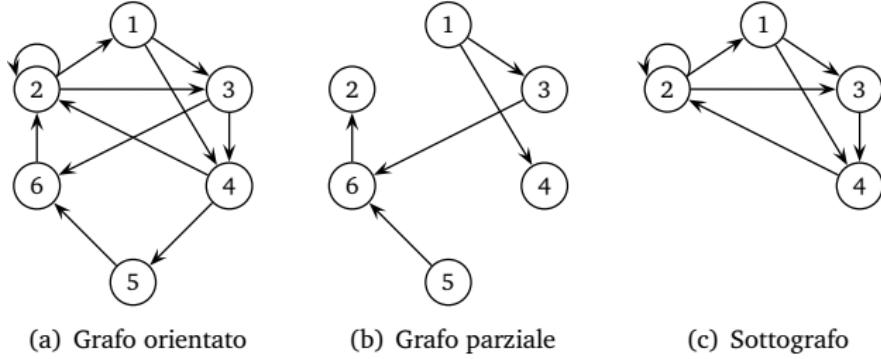


Figura 1.11: Esempio di grafo parziale e sottografo.

Un grafo è detto *connesso* se il grafo non orientato relativo al grafo orientato ha almeno un cammino che congiunge ogni coppia di vertici. Se, invece, tale cammino non esiste allora il grafo viene detto *disconnesso* (o semplicemente *non connesso*). In figura 1.12 sono forniti un esempio di grafo connesso e uno non connesso.

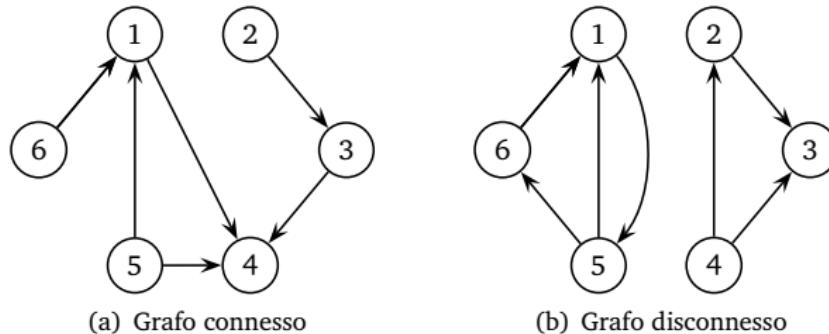


Figura 1.12: Esempio di grafo connesso e di grafo disconnesso.

Un grafo è detto *fortemente connesso* se nel grafo esiste almeno un cammino orientato che congiunge ogni coppia di vertici.

### 1.1.5 Alberi

Un grafo  $G_a$  non orientato di  $n$  vertici è un *albero* se rispetta le seguenti condizioni, che sono equivalenti:

- $G_a$  è connesso e aciclico;
  - $G_a$  è aciclico e si crea un ciclo semplice se si aggiunge un lato al grafo  $G_a$ ;
  - $G_a$  è connesso, ma diventa non connesso non appena si elimina un solo lato di  $G_a$ ;
  - $G_a$  è connesso e ha  $n - 1$  lati;
  - $G_a$  non ha cicli semplici e ha  $n - 1$  lati.

In letteratura esistono anche altre condizioni equivalenti. Ognuna di queste definizioni equivalenti può essere utile a "identificare" e "utilizzare" gli alberi.

Dato un grafo  $G$ , possono essere definiti dei sottografi di  $G$  che sono alberi. Tra questi, si definisce *albero completo* di  $G$  (detto anche *spanning tree*) un grafo parziale di  $G$  (i.e., "copre" tutti i vertici) che è un albero (si considerino gli esempi riportati in figura 1.13). Ogni grafo connesso ha almeno uno spanning tree.

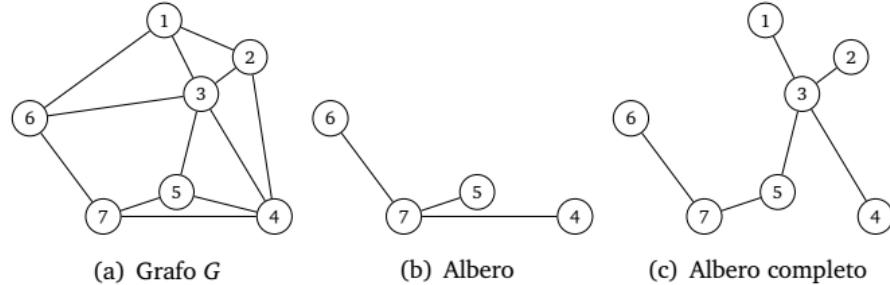


Figura 1.13: Esempio di alberi e alberi completi di un grafo  $G$ .

Un sottografo connesso di un albero è a sua volta un albero, detto *sottoalbero*. Invece, un grafo che non contiene cicli è una *foresta*; quindi, equivalentemente, si può affermare che una foresta è un insieme di alberi.

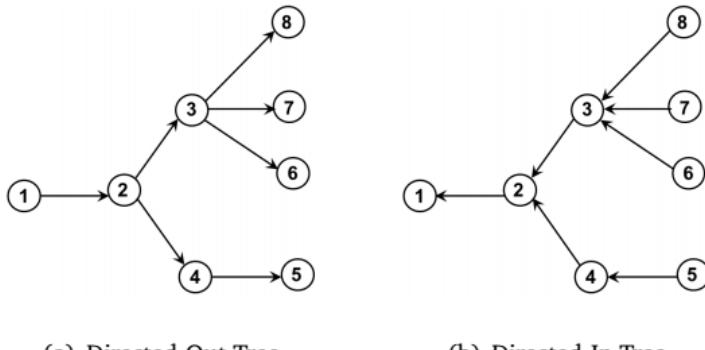


Figura 1.14: Esempio di directed-out-tree e directed-in-tree

Dato un albero si può indicare come *radice* (*root*) uno dei suoi nodi. In questo caso l'albero è denominato *albero radicato* (*rooted tree*). Se si denota con  $s$  la radice di un albero, si possono definire due tipologie di alberi (vedi figura 1.14):

- *Directed-out-tree*: è un albero in cui l'unico cammino dal nodo  $s$  a tutti gli altri nodi è diretto.
  - *Directed-in-tree*: è un albero in cui l'unico cammino da un qualsiasi altro nodo al nodo  $s$  è diretto.

### 1.1.6 Rappresentazione dei grafi

Il ruolo delle strutture dati è cruciale nello sviluppo di algoritmi efficienti. Il modo in cui sono salvati i dati del grafo (*rete*) nella memoria del calcolatore determina le performance degli algoritmi che operano su tali dati. Alcune delle operazioni che devono essere svolte dagli algoritmi sono le seguenti:

- Accedere alle informazioni dei vertici;
- Accedere alle informazioni degli archi;
- Determinare tutti gli archi che partono da un vertice  $i$ ;
- Determinare tutti gli archi che arrivano a un vertice  $i$ ;
- Determinare tutti gli archi che incidono su un vertice  $i$ .

In letteratura sono presentate numerose proposte. Alcune permettono un efficiente accesso ai dati, ma sono dispendiose dal punto di vista dell'occupazione di memoria, altre forniscono efficaci compromessi.

La scelta della struttura dati più opportuna dipende principalmente dall'algoritmo che si deve implementare e dalle “risorse” a disposizione.

La matrice di adiacenza  $Q$  di un grafo non orientato semplice  $G = (V, E)$  è la matrice simmetrica  $|V| \times |V|$  con elementi:

$$q_{ij} = \begin{cases} 1 & \text{se } \{i, j\} \in E; \\ 0 & \text{altrimenti.} \end{cases}$$

Invece, la matrice di adiacenza  $Q$  di un grafo orientato semplice  $G = (V, A)$  è la matrice  $|V| \times |V|$ , non necessariamente simmetrica, con elementi:

$$q_{ij} = \begin{cases} 1 & \text{se } (i, j) \in A; \\ 0 & \text{altrimenti.} \end{cases}$$

Nelle figure 1.15 e 1.16 sono riportati due esempi di matrice di adiacenza per un grafo non orientato e uno orientato, rispettivamente. Si noti che per il grafo orientato ogni riga e colonna  $i$  definiscono l'insieme dei successori  $\Gamma^+(i)$  e dei predecessori  $\Gamma^-(i)$ . Per un grafo orientato la matrice è simmetrica e le righe e colonne definiscono entrambe  $\Gamma(i)$ .

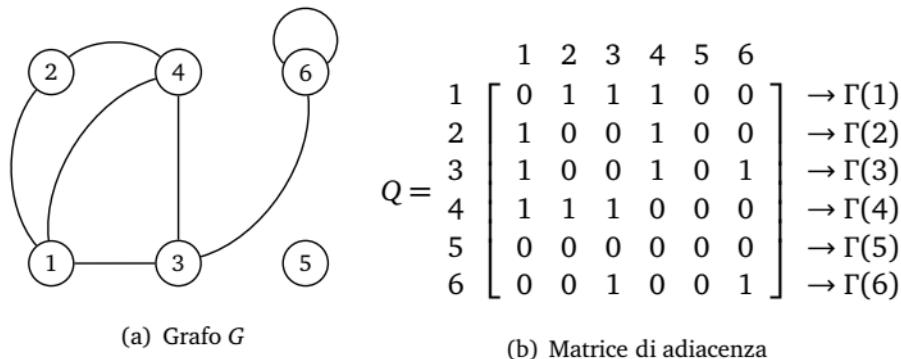


Figura 1.15: Esempio di matrice di adiacenza per un grafo non orientato.

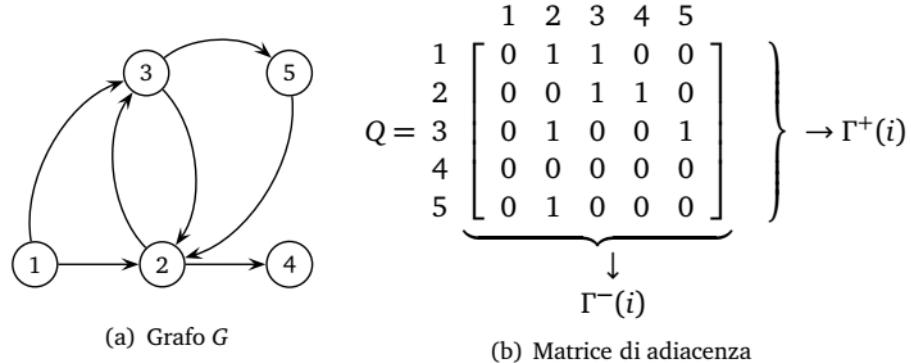


Figura 1.16: Esempio di matrice di adiacenza per un grafo orientato.

La *matrice di incidenza* vertici-lati  $D$  di un grafo non orientato  $G = (V, E)$  è la matrice  $|V| \times |E|$  con elementi:

$$d_{ik} = \begin{cases} 1 & \text{se il } k\text{-esimo lato è incidente nel vertice } i \text{ (i.e., } e_k = \{i, j\}\text{);} \\ 0 & \text{altrimenti.} \end{cases}$$

Invece, la matrice di incidenza vertici-archi  $D$  di un grafo orientato  $G = (V, A)$  è la matrice  $|V| \times |A|$  con elementi:

$$d_{ik} = \begin{cases} 1 & \text{se il } k\text{-esimo arco esce dal vertice } i \text{ (i.e., } a_k = (i, j)\text{);} \\ -1 & \text{se il } k\text{-esimo arco entra nel vertice } i \text{ (i.e., } a_k = (j, i)\text{);} \\ 0 & \text{se } i \text{ non è vertice terminale di } a_k. \end{cases}$$

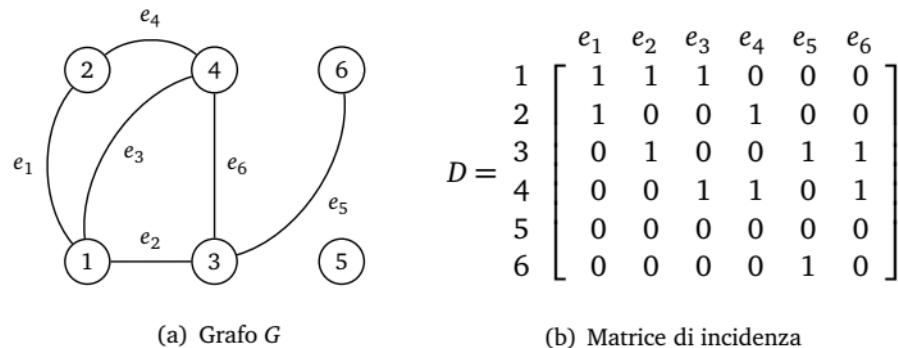


Figura 1.17: Esempio di matrice di incidenza per un grafo non orientato.

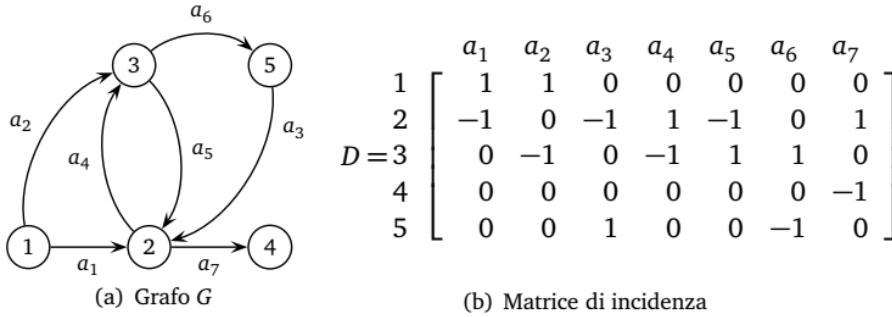


Figura 1.18: Esempio di matrice di incidenza per un grafo orientato.

Nelle figure 1.18 e 1.19 sono riportati due esempi di matrice di incidenza per un grafo non orientato e uno orientato, rispettivamente. Per un grafo non orientato la matrice ha sempre due "1" in ogni colonna (lato) e per ogni riga (vertice) un numero di "1" pari al numero di lati incidenti nel corrispondente vertice (denominato *grado del vertice* o "vertex degree"). Invece, per un grafo orientato la matrice ha sempre un "1" e un "-1" in ogni colonna (arco) e per ogni riga (vertice) un numero di "1" pari al numero di archi uscenti dal vertice (*grado di uscita* o *outdegree*) e un numero di "-1" pari al numero di archi entranti nel vertice (*grado di entrata* o *indegree*).

Le *liste di adiacenza* conservano per ogni vertice  $i$  la lista  $A(i)$  degli archi che partono da esso. Per cui, è necessario un vettore  $n$ -dimensionale ( $n = |V|$ ) *first*, dove  $first(i)$  memorizza il puntatore al primo elemento della lista  $A(i)$ . In figura 1.19 è riportato un esempio di liste di adiacenza per un grafo orientato in cui per ogni arco sono "salvate" una coppia di informazioni (per esempio, un costo e una capacità associate all'arco).

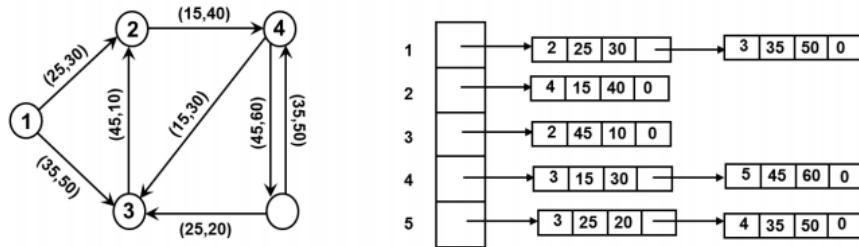


Figura 1.19: Esempio di liste di adiacenza per un grafo orientato.

Impiegando le liste di adiacenza si risparmia tempo calcolo e spazio di memoria. Però richiedono una "gestione" più complessa.

La rappresentazione *forward star* richiede di salvare le informazioni degli archi in un vettore  $m$ -dimensionale ( $m = |A|$ ). Gli archi devono essere ordinati per indice del vertice *iniziale* (*tail node*) crescente. Inoltre è necessario un vettore

$n$ -dimensionale ( $n = |V|$ ) di puntatori *point*, dove  $\text{point}(i)$  memorizza l'indice in cui sono salvate le informazioni del primo arco che *parte* dal vertice  $i$  nel corrispondente vettore. Gli archi che partono dal vertice  $i$  sono posizionati da  $\text{point}(i)$  fino a  $\text{point}(i + 1) - 1$ .

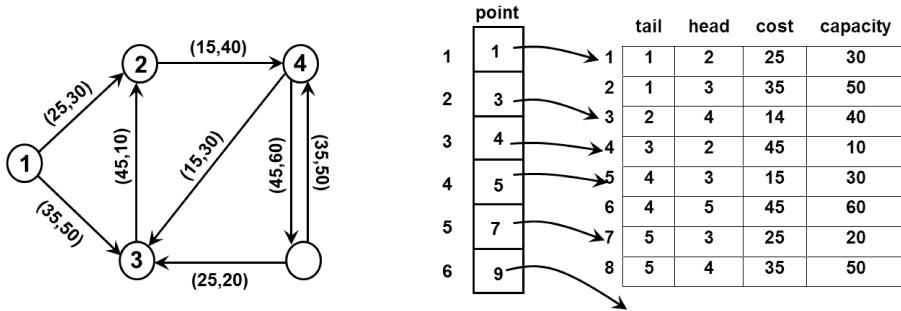


Figura 1.20: Esempio di forward star per un grafo orientato.

In figura 1.20 è riportato un esempio di forward star per lo stesso grafo orientato usato nell'esempio di figura 1.19.

La rappresentazione forward star consente di accedere in modo efficiente agli archi che partono da un determinato vertice  $i$ .

Nel caso sia necessario accedere efficacemente agli archi che arrivano a un determinato vertice  $i$  la forward star non permette un'equivalente performance. Quindi, nel caso l'algoritmo necessiti di accedere agli archi che arrivano a un determinato vertice  $i$  è necessario utilizzare la *backward star*. La rappresentazione backward star è analoga alla forward star tranne che:

- gli archi sono ordinati per indice del *vertice finale* (*head node*) crescente;
- il vettore  $\text{point}(i)$  memorizza l'indice in cui sono salvate le informazioni del primo arco che *arriva* al vertice  $i$  nel corrispondente vettore. Gli archi che arrivano al vertice  $i$  sono posizionati da  $\text{point}(i)$  fino a  $\text{point}(i + 1) - 1$ .

## 1.2 Cammini minimi

Consideriamo un grafo orientato  $G = (V, A)$  con  $n$  vertici e  $m$  archi in cui  $c_{ij}$  è il costo associato ad ogni arco  $(i, j) \in A$ . Il costo di un cammino da  $s \in V$  a  $t \in V$  è pari alla somma dei costi degli archi che lo compongono.

Il cammino di costo minimo (*cammino minimo*) da  $s$  a  $t$  è quello che, fra tutti i cammini da  $s$  a  $t$ , ha il costo più piccolo.

Nel caso in cui  $c_{ij} \geq 0$ , per ogni  $(i, j) \in A$ , il cammino minimo è elementare. Mentre, se alcuni dei costi  $c_{ij}$  sono negativi allora il grafo  $G$  può contenere circuiti di costo negativo. In questo caso il circuito di costo negativo può essere usato un numero infinito di volte per ridurre il costo.

Nel caso si voglia calcolare il *cammino minimo elementare* in un grafo in cui alcuni dei costi  $c_{ij}$  sono negativi è necessario imporre esplicitamente la

restrizione che il cammino passi attraverso ciascun vertice al massimo una sola volta. Purtroppo in presenza di cicli di costo negativo il problema è NP-Hard.

Esistono tuttavia casi particolari in cui non esistono sicuramente cicli di costo negativo e che possono essere risolti in tempo polinomiale, fra i quali: grafi aciclici; grafi con costi positivi.

### 1.2.1 Formulazione matematica

Definiamo per ogni arco  $(i, j) \in A$  la variabile decisionale  $x_{ij} \in \{0, 1\}$  che deve valere 1 se e solo se  $(i, j)$  viene scelto nel cammino, altrimenti vale 0.

Il problema del cammino minimo elementare da  $s$  a  $t$  ( $s \neq t$ ) può essere formulato come segue:

$$z_{SP} = \min \sum_{(i,j) \in A} c_{ij} x_{ij} \quad (1.1)$$

$$\text{s.t. } \sum_{j \in \Gamma^+(i)} x_{ij} - \sum_{j \in \Gamma^-(i)} x_{ji} = b_i, \quad i \in V \quad (1.2)$$

$$x_{ij} \in \{0, 1\}, \quad \forall (i, j) \in A \quad (1.3)$$

dove  $b_i = 1$ , se  $i = s$ ,  $b_i = -1$ , se  $i = t$ , e  $b_i = 0$ , se  $i \in V \setminus \{s, t\}$ . La funzione obiettivo (1.1) minimizza il costo degli archi in soluzione. Le equazioni (1.2) sono i cosiddetti vincoli di conservazione del flusso che prevedono l'immissione di una unità di flusso nel vertice  $s$ , l'assorbimento di una unità di flusso nel vertice  $t$ , mentre garantiscono che negli altri vertici il flusso entrante sia uguale al flusso uscente. I vincoli (1.3) impongono che le variabili decisionali siano binarie. In realtà potremo rilassare i vincoli di interezza (1.3) sostituendoli con i vincoli  $0 \leq x_{ij} \leq 1$ , perché la soluzione ottima sarà comunque intera (si veda la sezione sul problema dei cammini di costo minimo).

Nel caso possano esserci cicli di costo negativo è necessario aggiungere i seguenti vincoli:

$$\sum_{(i,j) \in A(S)} x_{ij} \leq |S| - 1, \quad \forall S \subseteq V, \quad S \neq \emptyset \quad (1.4)$$

dove  $A(S)$ ,  $S \subseteq V$ , è l'insieme degli archi con entrambi gli estremi in  $S$ , i.e.,  $A(S) = \{(i, j) \in A : i \in S, j \in S\}$ . Questi vincoli, che sono  $2^n - 1$ , impediscono il formarsi di *subtour* (sottocicli) e per questa ragione sono noti come vincoli di *subtour elimination*.

### 1.2.2 Assunzioni

In questa sezione descriviamo alcune assunzioni che saranno utili nel proseguo della sezione.

Innanzitutto si assume che tutti i costi degli archi  $c_{ij}$  siano interi e denotiamo con  $C$  il costo più alto, i.e.,  $C = \max\{c_{ij} : (i, j) \in A\}$ . Si noti che in linea di

principio tutti i costi “*razionali*” possono essere convertiti in interi, mentre i costi “*irrazionali*” (e.g.,  $\sqrt{2}, \pi, \dots$ ) non possono essere gestiti come interi.

Il grafo contiene un cammino diretto dal nodo  $s$  a ogni altro nodo. Per soddisfare questa assunzione si possono aggiungere degli archi artificiali con un costo “*sufficientemente*” grande.

Per alcuni algoritmi si assume che non esistano cicli di costo negativo, perché nel caso vi siano dei cicli di costo negativo, la soluzione ottima del problema sarebbe illimitata. Chiaramente questi algoritmi non possono essere utilizzati per grafi in cui vi sono cicli di costo negativo, perché non garantirebbero la soluzione e/o il corretto funzionamento.

Quando non è esplicitamente definito, si assume che il grafo sia orientato. Per soddisfare questa assunzione si può sostituire ogni arco non orientato (lato)  $\{i, j\}$  di costo  $c_{ij}$  con due archi diretti  $(i, j)$  e  $(j, i)$  entrambi di costo  $c_{ij}$ .

### 1.2.3 Distance label

Diversi algoritmi per calcolare i cammini minimi impiegano il vettore delle *distance label*. Per ogni vertice è definita una label  $d(i)$ , che rappresenta il costo di un qualche cammino diretto dal vertice sorgente  $s$  al nodo  $i$ . Le distance label sono un *upper bound* al costo del cammino minimo dal vertice sorgente  $s$  al vertice  $i$ .

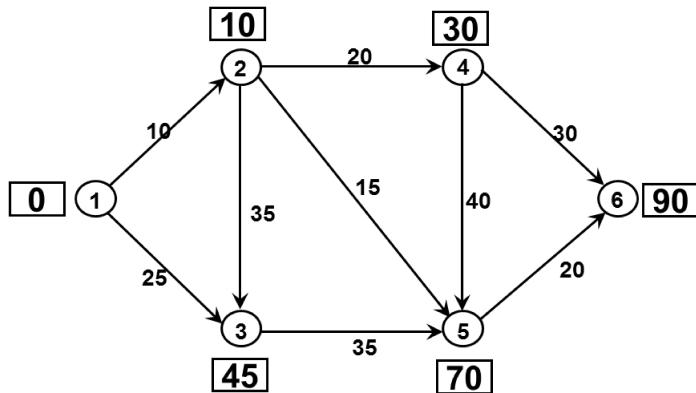


Figura 1.21: Esempio di grafo e corrispondenti distance label.

In Figura 1.21 è riportato un esempio dove il vertice di partenza del cammino è  $s = 1$  e la sua label è impostata a  $d(1) = 0$ . Le rimanenti label  $d(i)$  sono sicuramente degli upper bound al costo dei cammini necessari per raggiungere il vertice  $i$  dal vertice  $s$ . Infatti, per esempio, la distance label  $d(6) = 70$  corrisponde al costo del cammino  $P = (1, 2, 4, 5)$ .

### 1.2.4 Condizioni di ottimalità

Per definire quando le distance label  $d(i)$  rappresentano il costo del cammino di costo minimo sono necessarie delle condizioni di ottimalità.

**Lemma 1.1.** *Se le distance label  $d(i)$  rappresentano il costo del cammino di costo minimo dal vertice  $s$  al vertice  $i$ , allora devono soddisfare la seguente condizione:*

$$d(j) \leq d(i) + c_{ij} \quad \text{per ogni arco } (i, j) \in A. \quad (1.5)$$

**Dim.: 1.1.** *Se per qualche arco  $(i, j) \in A$  dovesse accadere che  $d(j) > d(i) + c_{ij}$ , allora la distance label  $d(j)$  non rappresenterebbe il costo del cammino di costo minimo dal vertice  $s$  al vertice  $j$  perché esisterebbe un cammino meno costoso che arriva dal vertice  $i$  con l'arco  $(i, j)$  (vedi figura 1.22).*

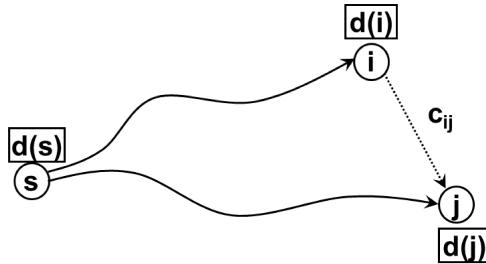


Figura 1.22: Condizioni di ottimalità: Teorema 1.1.

Il seguente teorema stabilisce le condizioni di ottimalità che sono alla base dei metodi *label setting* and *label correcting* che verranno introdotti in seguito.

**Teorema 1.1.** *Le distance label  $d(i)$  rappresentano il costo del cammino minimo se e solo se:*

$$\bar{c}_{ij} = c_{ij} + d(i) - d(j) \geq 0 \quad \text{per ogni arco } (i, j) \in A \quad (1.6)$$

**Dim.: 1.2.** *Per il Lemma 1.1 se le distance label  $d(i)$  rappresentano il costo del cammino minimo, allora  $\bar{c}_{ij} = c_{ij} + d(i) - d(j) \geq 0$  per ogni arco  $(i, j) \in A$ .*

*Ora si vuole dimostrare che se  $\bar{c}_{ij} = c_{ij} + d(i) - d(j) \geq 0$  per ogni arco  $(i, j) \in A$ , allora le distance label  $d(i)$  rappresentano il costo del cammino minimo. Dato un qualsiasi cammino  $P$  diretto dal nodo  $s$  al nodo  $k$ .*

$$\sum_{(i,j) \in P} \bar{c}_{ij} = \sum_{(i,j) \in P} (c_{ij} + d(i) - d(j))$$

*Si noti che dati due archi consecutivi  $(i, j)$  e  $(j, k)$  del path  $P$ ,  $-d(j)$  si semplifica con  $d(j)$ . Quindi, se si semplificano le distance label per tutti gli archi inclusi nel path  $P$ , si ha:*

$$\sum_{(i,j) \in P} \bar{c}_{ij} = \left( \sum_{(i,j) \in P} c_{ij} \right) + d(s) - d(k)$$

Siccome  $d(s) = 0$  e  $c_{ij} \geq 0$  per ogni arco  $(i, j) \in P$ , allora abbiamo:

$$d(k) \leq \sum_{(i,j) \in P} c_{ij} \quad (1.7)$$

Per cui  $d(k)$  è senz'altro un lower bound al costo di ogni cammino dal nodo  $s$  al nodo  $k$ . Dato che  $d(k)$  è anche la lunghezza di un qualche cammino da  $s$  a  $k$ , allora deve essere il costo del cammino minimo.

### 1.2.5 Algoritmi Label Correcting

Sulla base del Teorema 1.1, si può definire un algoritmo che determina i cammini di costo minimo correggendo le etichette fino a quando non soddisfano le condizioni di ottimalità.

Ipotizzando che il grafo orientato  $G(V, A)$  sia connesso e non abbia cicli di costo negativo (per esempio, ha costi  $c_{ij} \geq 0$  per ogni  $(i, j) \in A$ ), un algoritmo label correcting generico, che determina i cammini di costo minimo dal vertice  $s$  a tutti gli altri vertici, è riassunto nell'Algoritmo 1.

---

#### Algorithm 1 Label Correcting (Generico)

---

**Input:** Grafo orientato connesso senza cicli di costo negativo;  
**Output:** Cammini minimi da  $s$  a  $V \setminus \{s\}$  definiti da  $\text{pred}[j]$ ,  $\forall j \in V \setminus \{s\}$ ;

```

// Inizializzazione
for j = 1 to n do
    d[j] = ∞;
    pred[j] = -1;
end for
d[s] = 0;
//Ripete finché c'è una condizione violata
while (∃(i, j) ∈ A : d[j] > d[i] + cij) do
    d[j] = d[i] + cij;
    pred[j] = i;
end while

```

---

Se nel grafo  $G(V, A)$  esistono cicli di costo negativo, l'algoritmo label correcting generico qui proposto non termina mai. Se invece il grafo non ha cicli di costo negativo l'algoritmo termina sempre. In particolare, ad ogni iterazione l'algoritmo deve considerare tutti gli archi con una complessità pari a  $O(m)$ . Il numero di iterazioni è  $O(2nC)$  perché, sebbene all'inizio dell'algoritmo  $d(j) = \infty$  per ogni  $j \in V \setminus \{s\}$ , ogni distance label finita  $d(i)$  è limitata superiormente dal valore  $nC$  e limitata inferiormente dal valore  $-nC$ , inoltre ad ogni iterazione una distance label diminuisce di almeno un'unità mentre nessuna distance label aumenta. La complessità computazionale complessiva è pari a  $O(2nmC)$ , quindi è pseudo-polinomiale.

La complessità può essere diminuita a  $O(nm)$ . Per diminuire la complessità costruiamo un nuovo algoritmo label correcting basato sul seguente teorema.

**Teorema 1.2.** Se ad ogni iterazione si esaminano tutti gli archi uno alla volta, verificando le condizioni di ottimalità e aggiornando le *distance label* quando necessario, allora dopo  $k$  iterazioni saranno determinati tutti i cammini minimi contenenti al più  $k$  archi.

**Dim.: 1.3.** Si dimostra per induzione rispetto a  $k$  (vedi Figura 1.23).

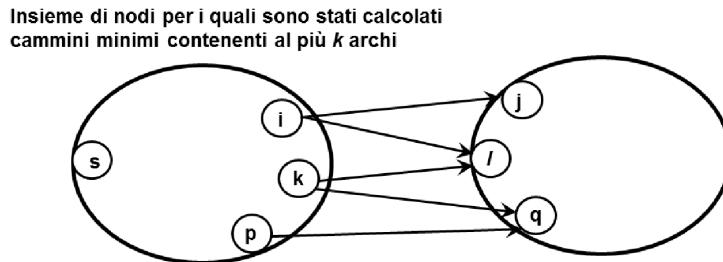


Figura 1.23: Dimostrazione per induzione per il Teorema 1.2.

Il Teorema 1.2 garantisce che al termine dell'iterazione  $k = n - 1$ , dopo che sono state aggiornate le label dopo aver considerato tutti gli archi, tutti i cammini di costo minimo di lunghezza al più  $n - 1$  sono stati generati. Siccome, in assenza assenza di cicli di costo negativo, i cammini minimi non possono avere più di  $n - 1$  archi, non è necessario svolgere altre iterazioni. Se al termine dell'iterazione  $k = n - 1$  ci sono ancora delle label che per qualche arco violano delle condizioni di ottimalità, significa che esiste un ciclo di costo negativo, che può essere identificato con  $\text{pred}[j]$ .

Nel caso in cui nel ciclo principale non vengano trovate condizioni di ottimalità violate, allora nessuna distance label  $d(i)$  verrà aggiornata. Però, se nessuna distance label verrà modificata allora anche nell'iterazione successiva nessuna condizione di ottimalità sarà violata. Quindi, nell'algoritmo si può permettere un'uscita anticipata quando nessuna distance label potrà essere ulteriormente modificata.

L'algoritmo che ne deriva è noto come Algoritmo di Bellman-Ford ed è riassunto nell'Algoritmo 2.

### 1.2.6 Algoritmo di Dijkstra

L'algoritmo di Dijkstra calcola i cammini minimi dal vertice  $s \in V$  a ogni altro vertice  $t \in V \setminus \{s\}$  solo se i costi degli archi sono *non negativi* (i.e.,  $c_{ij} \geq 0$ ,  $\forall (i, j) \in A$ ). L'algoritmo di Dijkstra si basa sul Teorema 1.3.

**Teorema 1.3.** Dato un sottoinsieme  $S \subseteq V$  che include  $s$  (i.e.,  $s \in S$ ), sia  $L_i$  il costo del cammino di costo minimo da  $s$  al vertice  $i$ , per ogni vertice  $i \in S$ . Se  $(v, h) = \text{argmin}\{L_i + c_{ij} : (i, j) \in \delta^+(S)\}$ , allora  $L_v + c_{vh}$  rappresenta il costo del cammino minimo da  $s$  ad  $h$ .

---

**Algorithm 2 Bellman-Ford**

---

**Input:** Grafo orientato;

**Output:** Cammini minimi da  $s$  a  $V \setminus \{s\}$  definiti da  $pred[j]$ ,  $\forall j \in V \setminus \{s\}$ ;

```

// Inizializzazione
for j = 1 to n do
    d[j] = ∞;
    pred[j] = -1;
end for
d[s] = 0;
// Controlla gli archi per  $n - 1$  iterazioni
for k = 1 to  $n - 1$  do
    update=False;
    for (i, j) ∈ A do
        if ( $d[j] > d[i] + c_{ij}$ ) then
            d[j] = d[i] +  $c_{ij}$ ;
            pred[j] = i;
            update=True;
        end if
    end for
    if (update==False) then
        Esci dal ciclo;
    end if
end for
// Controlla se ci sono cicli di costo negativo
for (i, j) ∈ A do
    if ( $d[j] > d[i] + c_{ij}$ ) then
        Il grafo contiene cicli di costo negativo;
    end if
end for

```

---

**Dim.: 1.4.**  $L_v + c_{vh}$  rappresenta il costo di un cammino da  $s$  ad  $h$ . Si consideri un altro cammino  $P$  che termina in  $h$ . Sia  $(i, j) \in P \cap \delta^+(S)$  e si partizionino  $P$  in  $P_1 \cup \{(i, j)\} \cup P_2$ , dove  $P_1$  e  $P_2$  sono due cammini da  $s$  ad  $i$  e da  $j$  ad  $h$ , rispettivamente (vedi Figura 1.24). Si ha:

$$C(P) = \underbrace{c(P_1)}_{\geq L_i} + c_{ij} + \underbrace{C(P_2)}_{\geq 0} \geq L_i + c_{ij} \geq L_v + c_{vh}.$$

Per cui,  $L_v + c_{vh}$  rappresenta il costo del cammino di costo minimo da  $s$  ad  $h$ .

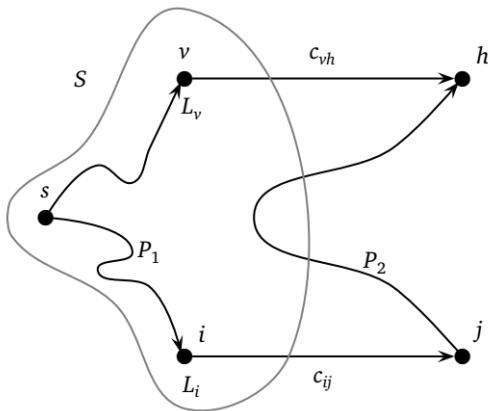


Figura 1.24: Dimostrazione del Teorema 1.3.

Il Teorema 1.3 suggerisce un algoritmo iterativo, noto come Algoritmo di Dijkstra, per la determinazione dei cammini minimi da  $s \in V$  ad ogni  $t \in V$ . Nell'algoritmo di Dijkstra l'insieme  $S$  può essere interpretato come l'insieme dei vertici *permanent*i le cui label rappresentano i costi del cammino di costo minimo. Mentre, il vertice  $h$  dato da  $(v, h) = \operatorname{argmin}\{L_i + c_{ij} : (i, j) \in \delta^+(S)\}$  rappresenta il nuovo vertice che entra nell'insieme  $S$  dei vertici permanenti. Il vertice  $h$  oltre a entrare nell'insieme  $S$  viene anche *espanso* per eventualmente aggiornare le label dei vertici successori. Una prima versione base dell'Algoritmo di Dijkstra è proposto nell'Algoritmo 3.

L'algoritmo 3 calcola i cammini minimi dal vertice  $s$  a tutti gli altri vertici del grafo. Si noti che se si è interessati a calcolare i cammini minimi dal vertice  $s$  a un sottoinsieme di vertici  $T \subseteq V$  (eventualmente contenente un solo vertice  $t$ , i.e.,  $T = \{t\}$ ), possiamo fermare l'Algoritmo di Dijkstra non appena le etichette di tutti i vertici di  $T$  sono in  $S$ , quindi permanenti.

La complessità computazionale dell'Algoritmo 3 è pari a  $O(nm)$ . Ma è possibile ottenere una complessità  $O(n^2)$  se ad ogni iterazione si sfruttano opportunamente le informazioni già acquisite nelle iterazioni precedenti. In particolare, l'algoritmo di Dijkstra potrebbe utilizzare le seguenti strutture dati definite per ogni  $j \in V$ :

---

**Algorithm 3 Dijkstra (1<sup>a</sup> versione)**

---

**Input:** Grafo orientato con costi  $\{c_{ij}\}$  non-negativi;  
**Output:** Cammini minimi da  $s$  a  $V \setminus \{s\}$  definiti da  $pred[j]$ ,  $\forall j \in V \setminus \{s\}$ ;

```

 $S = \{s\};$ 
 $L[s] = 0;$ 
 $pred[s] = s;$ 
while ( $|S| \neq n$ ) do
    if ( $\delta^+(S) \neq \emptyset$ ) then
         $(v, h) = argmin\{L[i] + c_{ih} : (i, h) \in \delta^+(S)\};$ 
         $L[h] = L[v] + c_{vh};$ 
         $pred[h] = v;$ 
         $S = S \cup \{h\};$ 
    else
        Grafo  $G$  disconnesso;
        STOP;
    end if
end while
```

---

- $flag[j] = \begin{cases} 1 & \text{se } j \in S \\ 0 & \text{altrimenti} \end{cases}$
- $L[j] = \begin{cases} \text{costo del cammino da } s \text{ a } j, & \text{se } j \in S; \\ min\{L[i] + c_{ij} : i \in S\}, & \text{se } j \notin S; \end{cases}$
- $pred[j] = \begin{cases} \text{predecessore di } j \text{ nel cammino da } s \text{ a } j, & \text{se } j \in S \\ argmin\{L[i] + c_{ij} : i \in S\}, & \text{se } j \notin S \end{cases}$

La versione revisionata dell’Algoritmo di Dijkstra di complessità  $O(n^2)$  è proposta nell’Algoritmo 4.

La complessità dell’Algoritmo di Dijkstra può essere ulteriormente ridotta utilizzando delle opportune strutture dati per mantenere *ordinate* le label dei vertici non ancora inseriti nell’insieme dei permanenti  $S$  con l’obiettivo di rendere più efficiente la selezione del nodo. Tra le diverse opzioni vi è l’impiego di una *heap* e se si utilizza una Fibonacci Heap, si ottiene una complessità  $O(m+n \log n)$ .

Un’altra opzione molto interessante è l’utilizzo dell’approccio di Dial che permette di avere una complessità  $O(m + nC)$ , che nonostante sia pseudopolinomiale raramente raggiunge il caso peggiore. Inoltre, per grafi in cui  $C$  è piccolo anche il caso peggiore risulta competitivo.

In Figura 1.25 è riportato un esempio di applicazione dell’algoritmo di Dijkstra, in cui per ogni vertice è riportata la coppia  $[pred[j], L[j]]$ , per ogni  $j \in V$ . Il primo passo dell’algoritmo consiste nell’inizializzazione delle label  $L[j]$  e del predecessore  $pred[j]$  per ogni vertice  $j \in V$ , come mostrato in Figura 1.25a. L’insieme  $S$  dei vertici permanenti è inizializzato come  $S = \{s\} = \{1\}$ . Dopodiché, in Figura 1.25b l’algoritmo esegue la prima iterazione includendo

---

**Algorithm 4 Dijkstra (versione  $O(n^2)$ )**

---

**Input:** Grafo orientato connesso con costi  $\{c_{ij}\}$  non-negativi;  
**Output:** Cammini minimi da  $s$  a  $V \setminus \{s\}$  definiti da  $pred[j]$ ,  $\forall j \in V \setminus \{s\}$ ;

```

// Inizializzazione
for j = 1 to n do
    flag[j] = 0;
    pred[j] = s;
    L[j] = csj;
end for
flag[s] = 1;
L[s] = 0;
for k = 1 to n - 1 do
    // Individua  $h = argmin\{L[j] : j \notin S\}$ 
    min = +∞;
    for j = 1 to n do
        if (flag[j] = 0) and ( $L[j] < min$ ) then
            min = L[j];
            h = j;
        end if
    end for
    // Aggiorna  $S = S \cup \{h\}$ 
    flag[h] = 1;
    // Aggiorna  $L[j]$  e  $pred[j]$  per ogni  $j \notin S$ 
    for j = 1 to n do
        if (flag[j] = 0) and ( $L[h] + c_{hj} < L[j]$ ) then
            L[j] = L[h] + chj;
            pred[j] = h;
        end if
    end for
end for

```

---

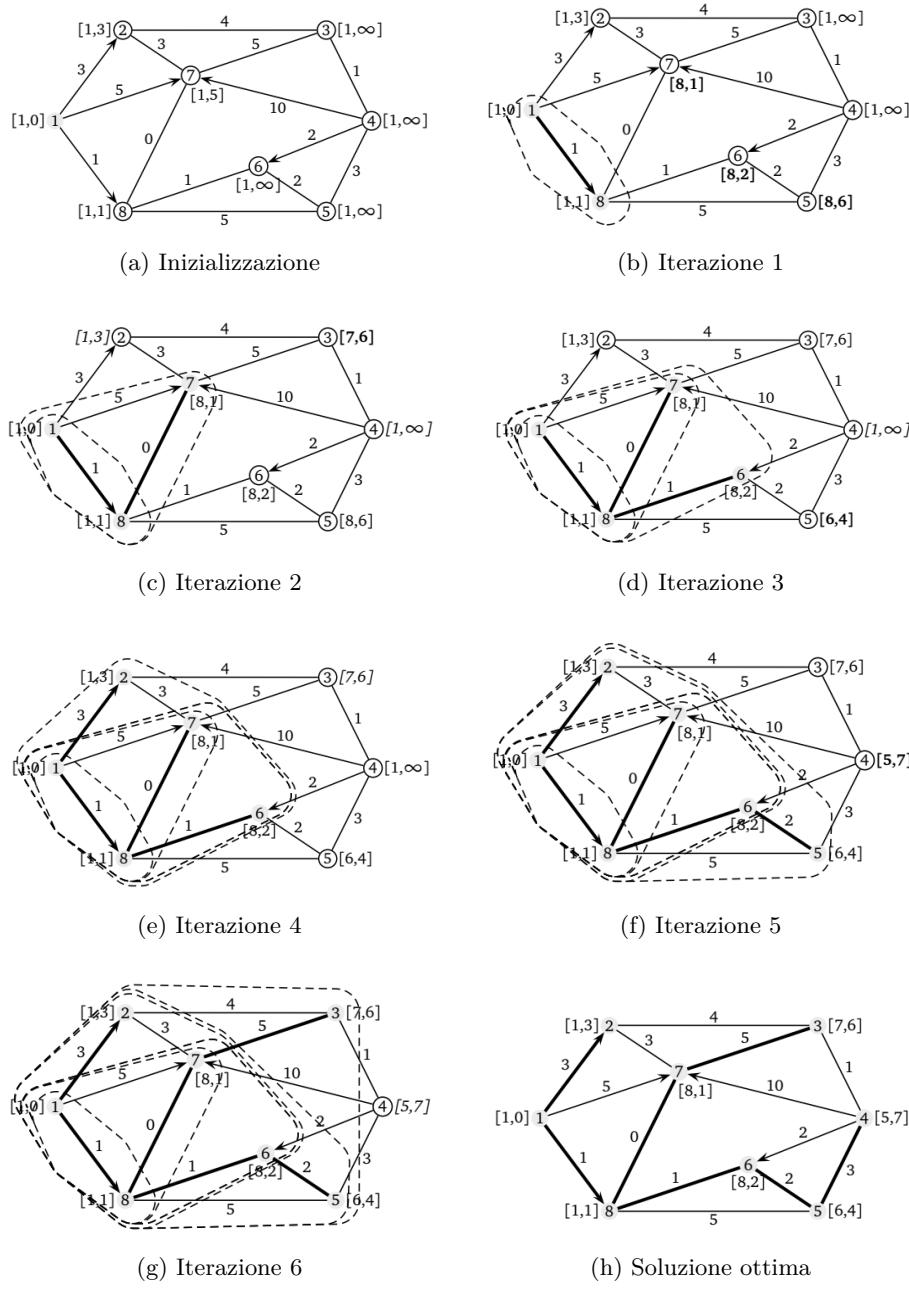


Figura 1.25: Un esempio di esecuzione dell’Algoritmo di Dijkstra

$S$	$L[j]$								$pred[j]$							
	2	3	4	5	6	7	8	2	3	4	5	6	7	8		
{1}	3	$\infty$	$\infty$	$\infty$	$\infty$	5	1	1	-	-	-	-	1	1		
{1,8}	3	$\infty$	$\infty$	6	2	1	1	1	-	-	8	8	8	1		
{1,8,7}	3	6	$\infty$	6	2	1	1	1	7	-	8	8	8	1		
{1,8,7,6}	3	6	$\infty$	4	2	1	1	1	7	-	6	8	8	1		
{1,8,7,6,2}	3	6	$\infty$	4	2	1	1	1	7	-	6	8	8	1		
{1,8,7,6,2,5}	3	6	7	4	2	1	1	1	7	5	6	8	8	1		
{1,8,7,6,2,5,3}	3	6	7	4	2	1	1	1	7	5	6	8	8	1		
{1,8,7,6,2,5,3,4}	3	6	7	4	2	1	1	1	7	5	6	8	8	1		

Figura 1.26: Tabella strutture dati usate dall’Algoritmo di Dijkstra

il vertice 8 nell’insieme dei vertici permanenti  $S$ , perché è il vertice in  $V \setminus S$  con la label  $L[j]$  più piccola. La prima iterazione termina con l’espansione del vertice 8, ossia si aggiorna la label  $L[j]$  e  $pred[j]$  per tutti i vertici  $j$  successori del vertice 8. Nella seconda iterazione, in Figura 1.25c entra in  $S$  il vertice 7 che poi viene espanso. Poi nelle iterazioni successive, nelle Figure 1.25d-1.25g, entrano in  $S$  nell’ordine i vertici 6, 2, 5, 3 e 4 che di volta in volta vengono espansi. In Figura 1.25h è riportato il risultato finale in cui i cammini di costo minimo sono evidenziati. In Figura 1.26 è riportata la tabella che illustra come varia il contenuto delle strutture dati usate dall’algoritmo di Dijkstra man mano che procede con le iterazioni.

### 1.2.7 Algoritmo di Floyd-Warshall

I cammini di costo minimo fra tutte le coppie di vertici di un grafo con costi non-negativi possono essere calcolati eseguendo  $n$  volte l’Algoritmo di Dijkstra utilizzando ad ogni esecuzione come vertice iniziale  $s$  uno degli  $n$  vertici del grafo. Ad ogni esecuzione dell’Algoritmo di Dijkstra si calcola la riga  $s$  della matrice di ordine  $n$  contenente il costo del cammino di costo minimo da  $s$  a tutti gli altri vertici. La complessità dell’algoritmo risultante è  $O(n^3)$  nel caso si usi l’Algoritmo 4.

Un metodo alternativo è quello utilizzato dall’algoritmo di Floyd-Warshall che ha sempre complessità pari a  $O(n^3)$ , ma si applica a qualsiasi grafo, compresi quelli con costi negativi, ed è in grado di riconoscere circuiti di costo negativo.

L’algoritmo di Floyd-Warshall si applica a un grafo orientato definito dalla matrice quadrata di ordine  $n$  dei costi  $[c_{ij}]$ . La sua implementazione richiede due matrici quadrate di ordine  $n$ : la matrice  $U$  per memorizzare i costi dei cammini minimi e la matrice  $Pred$  per salvare i vertici predecessori nei cammini necessari per ricostruire i cammini di costo minimo.

L’idea alla base dell’Algoritmo di Floyd-Warshall, descritto nell’Algoritmo 5, è molto semplice. La matrice  $U$  e  $Pred$  sono inizializzati con i cammini di un solo arco tra ogni coppia di vertici  $i$  e  $j$ , che hanno costo  $c_{ij}$  e predecessore  $pred[i, j] = i$ . Si noti che se l’arco  $(i, j)$  non dovesse esistere potremo impostare il suo costo

---

**Algorithm 5 Floyd-Warshall**

---

**Input:** Grafo orientato definito dalla matrice dei costi  $[c_{ij}]$ ;

**Output:** Matrici  $[u_{ij}]$  e  $[pred[i, j]]$ ;

// Inizializzazione

for  $i = 1$  to  $n$  do

    for  $j = 1$  to  $n$  do

$u_{ij} = c_{ij}$ ;

$pred[i, j] = i$ ;

    end for

end for

// Operazione triangolare su  $h$

for  $k = 1$  to  $n$  do

    for  $i = 1$  to  $n$  do

        for  $j = 1$  to  $n$  do

            if  $(u_{ik} + u_{kj} < u_{ij})$  then

$u_{ij} = u_{ik} + u_{kj}$ ;

$pred[i, j] = pred[k, j]$ ;

            end if

        end for

    end for

end for

for  $i = 1$  to  $n$  do

    if  $(u_{ii} < 0)$  then

        STOP, circuiti negativi;

    end if

end for

end for

---

a infinito, i.e.,  $c_{ij} = \infty$  (nella pratica un valore “sufficientemente grande”), inoltre,  $c_{ii} = 0$  per ogni  $i \in V$ . Dopodiché, il meccanismo di funzionamento dell’Algoritmo di Floyd-Warshall si basa sul Teorema 1.4.

**Teorema 1.4.** *Per ogni coppia di vertici  $i$  e  $j$  del grafo  $G(V, A)$ ,  $u_{ii}$  sia il costo di un qualche cammino da  $i$  a  $j$ .*

*I costi  $[u_{ij}]$  rappresentano i cammini di costo minimo tra tutte le coppie di vertici del grafo  $G$  se e solo se soddisfano la seguente condizione di ottimalità:*

$$u_{ij} \leq u_{ik} + u_{kj}, \quad \text{per tutti i vertici } i, j \text{ e } k.$$

Si noti che ciascun valore  $u_{ij}$  calcolato all’iterazione  $k$ -esima dell’Algoritmo di Floyd-Warshall rappresenta il costo del cammino di costo minimo dal vertice  $i$  al vertice  $j$  usando come vertici *interni* al cammino i vertici dell’insieme  $\{1, 2, \dots, k\}$ . Al termine dell’algoritmo, per ogni  $i, j \in V$ ,  $u_{ij}$  rappresenta il costo del cammino di costo minimo da  $i$  a  $j$  mentre  $\text{pred}[i, j]$  rappresenta il predecessore di  $j$  nel cammino di costo minimo da  $i$  a  $j$ .

Se  $u_{ii} < 0$  allora esiste un circuito di costo negativo, che può essere ricostruito a partire da  $\text{pred}[i, i]$ .

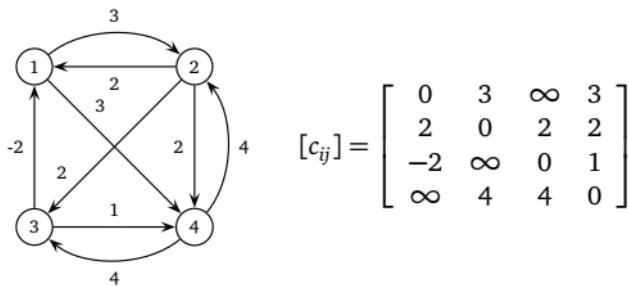


Figura 1.27: Grafo usato nell’esempio dell’Algoritmo di Floyd-Warshall.

In Figura 1.27 è riportato un grafo orientato e la corrispondente matrice dei costi  $[c_{ij}]$ . L’algoritmo di Floyd-Warshall inizializza le matrici  $U$  e  $\text{Pred}$  come illustrato in Figura 1.28. Dopodiché, l’algoritmo svolge le quattro iterazioni, illustrate in Figura 1.29, che consentiranno di ottenere le matrici  $U$  e  $\text{Pred}$  che forniranno i cammini di costo minimo per tutte le coppie di vertici  $i$  e  $j$  del grafo.

$u_{ij}$				$\text{pred}[i,j]$			
0	3	$\infty$	3				
2	0	2	2				
-2	$\infty$	0	1				
$\infty$	4	4	0	1	2	3	4
				1	2	3	4
				2	2	2	2
				3	3	3	3
				4	4	4	4

Figura 1.28: Inizializzazione delle matrici  $U$  e  $\text{Pred}$  con l’Algoritmo di Floyd-Warshall.

A ogni iterazione  $k$ , per ogni coppia di vertici  $i$  e  $j$  deve essere verificato se  $u_{ij} > u_{ik} + u_{kj}$  e nel qual caso è necessario aggiornare il costo del cammino da  $i$  a  $j$  (i.e.,  $u_{ij} = u_{ik} + u_{kj}$ ) e il predecessore (i.e.,  $\text{pred}[i, j] = \text{pred}[k, j]$ ) per consentire poi la ricostruzione del cammino.

Data la soluzione ottenuta al termine dell'ultima iterazione, se si vuole conoscere il cammino di costo minimo da 4 a 1, sappiamo che ha costo 2 perché  $u_{41} = 2$ , mentre  $\text{pred}[4, 1] = 3$  indica che il cammino termina con l'arco  $(3, 1)$  e  $\text{pred}[4, 3] = 4$  che il cammino inizia con l'arco  $(4, 3)$ . In generale, data la coppia  $i$  e  $j$  di cui vogliamo conoscere il cammino di costo minimo, l'ultimo arco è  $(g, j)$  dove  $g = \text{pred}[i, j]$ , il penultimo arco è  $(h, g)$  dove  $h = \text{pred}[i, g]$  e così via fino a quando  $\text{pred}$  restituisce  $i$ .

### 1.3 Alberi di copertura

Sia  $G = (V, E)$  un grafo non orientato连通的 con  $n$  vertici e  $m$  lati e sia  $c_e$  o  $c_{ij}$  il costo associato ad ogni lato  $e = \{i, j\} \in E$ .

Dato un albero completo di  $G$  definito dai vertici di  $V$  e dal sottoinsieme di lati  $T \subseteq E$ , il costo  $c(T)$  dell'albero  $G_T(V, T)$  è dato dalla somma dei costi degli lati che lo compongono:

$$c(T) = \sum_{e \in T} c_e$$

Spesso l'albero completo è anche chiamato *albero di copertura*, *albero di supporto*, oppure *albero coprente*.

Il *Problema dell'Albero di Copertura di Costo Minimo (Shortest Spanning Tree, SST)* consiste nell'individuare, fra i diversi alberi completi di  $G$ , un albero  $G_T = (V, T)$  di costo  $c(T)$  minimo.

#### 1.3.1 Applicazioni

Un esempio di applicazione è il disegno di una rete di comunicazione in cui, dato un grafo non orientato连通的  $G$  che rappresenta una rete dei possibili collegamenti fra  $n$  città, si vuole determinare la rete di costo minimo per connettere le  $n$  città. La rete deve consentire che ogni città sia collegata direttamente con almeno un'altra città e deve esistere un cammino che collega ogni coppia di città.

$u_{ij}$	$\Rightarrow$	$u_{ij}$
0 3 $\infty$ 3		0 3 $\infty$ 3
2 0 2 2		2 0 2 2
-2 $\infty$ 0 1		-2 1 0 1
$\infty$ 4 4 0		$\infty$ 4 4 0

$pred[i,j]$	$\Rightarrow$	$pred[i,j]$
1 1 1 1		1 1 1 1
2 2 2 2		2 2 2 2
3 3 3 3		3 1 3 3
4 4 4 4		4 4 4 4

(a) Iterazione  $k = 1$ 

$u_{ij}$	$\Rightarrow$	$u_{ij}$
0 3 $\infty$ 3		0 3 5 3
2 0 2 2		2 0 2 2
-2 1 0 1		-2 1 0 1
$\infty$ 4 4 0		6 4 4 0

$pred[i,j]$	$\Rightarrow$	$pred[i,j]$
1 1 1 1		1 1 2 1
2 2 2 2		2 2 2 2
3 1 3 3		3 1 3 3
4 4 4 4		2 4 4 4

(b) Iterazione  $k = 2$ 

$u_{ij}$	$\Rightarrow$	$u_{ij}$
0 3 5 3		0 3 5 3
2 0 2 2		0 0 2 2
-2 1 0 1		-2 1 0 1
6 4 4 0		2 4 4 0

$pred[i,j]$	$\Rightarrow$	$pred[i,j]$
1 1 2 1		1 1 2 1
2 2 2 2		3 2 2 2
3 1 3 3		3 1 3 3
2 4 4 4		3 4 4 4

(c) Iterazione  $k = 3$ 

$u_{ij}$	$\Rightarrow$	$u_{ij}$
0 3 5 3		0 3 5 3
0 0 2 2		0 0 2 2
-2 1 0 1		-2 1 0 1
2 4 4 0		2 4 4 0

$pred[i,j]$	$\Rightarrow$	$pred[i,j]$
1 1 2 1		1 1 2 1
3 2 2 2		3 1 3 3
3 1 3 3		3 4 4 4
3 4 4 4		3 4 4 4

(d) Iterazione  $k = 4$ 

Figura 1.29: Un esempio di esecuzione dell’Algoritmo di Floyd-Warshall

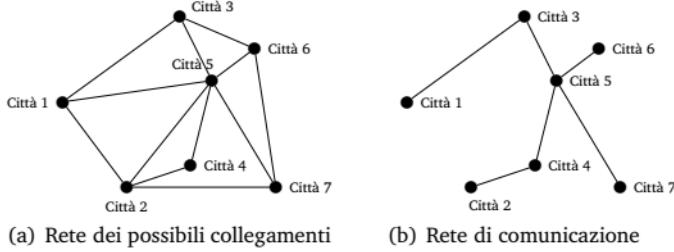


Figura 1.30: Disegno di una rete di comunicazione

In Figura 1.30 è riportato un esempio in cui dato il grafo dei possibili collegamenti tra le città (Figura 1.30a) viene mostrato un possibile albero di copertura che consente di collegare tutte le città (Figura 1.30b).

### 1.3.2 Formulazione matematica

Dato un grafo non orientato  $G = (V, E)$  con  $n$  vertici ed  $m$  lati, un suo albero completo  $G' = (V, A')$  è definito da una delle seguenti definizioni equivalenti:

- $G'$  è *connesso* e *non ha cicli*;
- $G'$  ha  $n - 1$  *lati* e *non ha cicli*;
- $G'$  è *connesso* e contiene esattamente  $n - 1$  *lati*.

Queste definizioni sono utili per definire i vincoli da includere nel modello matematico del problema. Inoltre, per formulare matematicamente il problema si deve definire per ogni lato  $e \in E$  la variabile decisionale:

$$x_e = \begin{cases} 1 & \text{se il lato } e \text{ viene scelto nell'albero minimo;} \\ 0 & \text{altrimenti.} \end{cases}$$

Una formulazione matematica per il problema è la seguente:

$$(SST) \quad z_{SST} = \min \sum_{e \in E} c_e x_e \quad (1.8)$$

$$s.t. \quad \sum_{e \in E} x_e = n - 1, \quad (1.9)$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1, \quad \forall S \subseteq V, |S| \geq 3 \quad (1.10)$$

$$x_e \in \{0, 1\}, \quad \forall e \in E \quad (1.11)$$

La funzione obiettivo (1.8) minimizza il costo dell'albero di copertura. Il vincolo (1.9) impone che siano scelti esattamente  $n - 1$  lati. Mentre, i vincoli (1.10) garantiscono l'assenza di cicli nel grafo parziale  $G_T = (V, T)$  definito dalla soluzione, i.e.,  $T = \{e \in E : x_e = 1\}$ . Il numero dei vincoli (1.10) è  $O(2^n)$ .

### 1.3.3 Condizioni di ottimalità

Dato un albero  $G_T = (V, T)$  ed un lato  $e \notin T$ , se aggiungiamo il lato  $e$  all'albero otteniamo un *ciclo*. Si denota con  $C(T, e)$  l'insieme dei lati del ciclo contenuto in  $T \cup \{e\}$  (nell'esempio  $C(T, e) = (e, f, g, h)$ ). In Figura 1.31 è riportato un esempio in cui si è aggiunto il lato  $e$  nell'albero ottenendo il ciclo contenente i lati  $\{e, f, g, h\}$ .

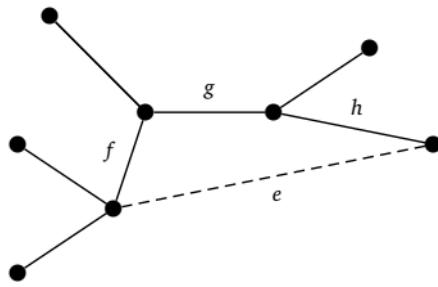


Figura 1.31: Ciclo generato dall'inserimento del lato  $e$

Dato un albero  $G_T = (V, T)$  ed un suo lato  $g \in T$ , se eliminiamo il lato  $g$  dall'albero otteniamo due alberi, i cui vertici appartengono all'insieme  $S$  e all'insieme  $V \setminus S$ . Gli insiemi  $S$  e  $V \setminus S$  determinano un *taglio*  $\delta(S) = \{\{i, j\} \in E : i \in S, j \in V \setminus S \text{ oppure } j \in S, i \in V \setminus S\}$ . In Figura 1.32 è riportato un esempio in cui è stato eliminato il lato  $g$  dall'albero ottenendo due alberi che determinano i due insiemi  $S$  e  $V \setminus S$  che definiscono un taglio.

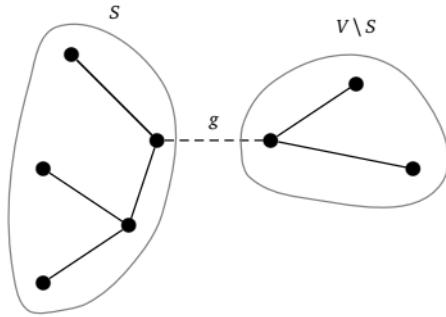


Figura 1.32: Taglio generato dall'eliminazione del lato  $g$

Qui di seguito presentiamo due condizioni di ottimalità equivalenti note come *Cut Optimality Conditions* e *Path Optimality Conditions*.

**Teorema 1.5 (Cut Optimality Conditions).** *L'albero di copertura  $G_T = (V, T)$  è di costo minimo se e solo se soddisfa le seguenti condizioni di ottimalità:*

per ogni lato appartenente all'albero  $e \in T$ ,  $c_e \leq c_g$  per ogni lato  $g$  appartenente al taglio  $S$  e  $V \setminus S$  generato dall'eliminazione del lato  $e$ .

**Dim.: 1.5.** (a) Se l'albero di copertura è di costo minimo, allora deve soddisfare le condizioni di ottimalità.

Se per un lato  $e \in T$  accadesse che  $c_e > c_g$  per un lato  $g$  appartenente al taglio  $S$  e  $V \setminus S$  generato dall'eliminazione del lato  $e$ , allora basterebbe sostituire il lato  $e$  con  $g$  in  $T$  per ottenere un albero di copertura di costo più basso contraddicendo l'ipotesi iniziale.

(b) Se l'albero di copertura rispetta le condizioni di ottimalità, allora è un albero di copertura di costo minimo.

Supponiamo che l'albero di copertura  $T$  non sia quello di costo minimo e che quello ottimo sia  $T^*$ . Quindi, esiste un lato  $e \in T$  che non è nell'albero di copertura di costo minimo, i.e.,  $e \notin T^*$ .

Eliminare il lato  $e$  da  $T$  crea un taglio corrispondente ai due alberi così ottenuti, mentre aggiungere il lato  $e$  all'albero ottimo  $T^*$  crea un ciclo (vedi Figura 1.33).

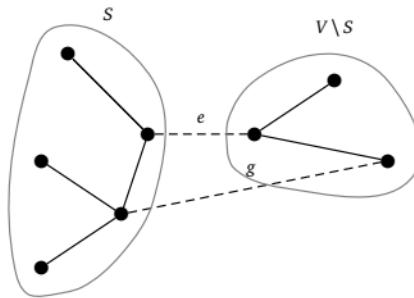


Figura 1.33: Esempio usato nelle dimostrazioni dei Teoremi 1.5 e 1.6

L'albero  $T$  soddisfa le condizioni di ottimalità, quindi  $c_e \leq c_g$  per tutti i lati del taglio compreso il lato  $g$  che assieme al lato  $e$  crea un ciclo nell'albero ottimo  $T^*$  (vedi Figura 1.33). Però, se  $T^*$  è l'albero ottimo  $c_e \geq c_g$ , altrimenti potremo trovare un albero con costo più basso scambiando i lati. Per cui,  $c_e = c_g$ . Se ripetiamo, questo processo per tutti i lati di  $T$  che non sono in  $T^*$  otterremo lo stesso risultato. Quindi, l'albero di copertura  $T$  è di costo minimo.

Il Teorema 1.5 sarà alla base dell'algoritmo di Prim-Dijkstra presentato nella Sezione 1.3.4, mentre il successivo Teorema 1.6 sarà alla base dell'algoritmo di Kruskal descritto nella Sezione 1.3.5.

**Teorema 1.6 (Path Optimality Conditions).** L'albero di copertura  $G_T = (V, T)$  è di costo minimo se e solo se soddisfa le seguenti condizioni di ottimalità: per ogni lato  $g$  non appartenente all'albero (i.e.,  $g \notin T$ ),  $c_g \geq c_e$  per ogni lato  $e$  contenuto nel path in  $T$  che collega gli estremi del lato  $g$ .

**Dim.: 1.6.** (a) Se l'albero di copertura è di costo minimo, allora deve soddisfare le condizioni di ottimalità.

Se  $e$  è un lato del path in  $T$  che collega gli estremi del lato  $g \notin T$  e  $c_e > c_g$ , potremo sostituire il lato  $e$  con  $g$  ottenendo un albero di copertura con un costo più basso contraddicendo l'ipotesi iniziale.

(b) Se l'albero di copertura rispetta le condizioni di ottimalità, allora è un albero di copertura di costo minimo.

Dimostriamo che un albero  $T$  che soddisfa le path optimality conditions soddisfa anche le cut optimality conditions, così, per il teorema precedente,  $T$  è l'albero di copertura di costo minimo.

Sia  $e$  un qualsiasi lato appartenente all'albero  $T$ , e sia  $S$  e  $V \setminus S$  il taglio generato dalla sua eliminazione. Per ogni lato  $g$  del taglio, nell'albero  $T$  ci sarà un unico path  $P$  che collega gli estremi di  $g$  (vedi Figura 1.33). Siccome  $e$  era l'unico lato del taglio che apparteneva a  $T$  (per costruzione), deve appartenere al path  $P$ .

Per ipotesi le condizioni di ottimalità sono valide, quindi  $c_g \geq c_e$  per ogni lato  $e$  contenuto nel path in  $T$  che collega gli estremi del lato  $g$ . Siccome questa condizione deve essere valida per tutti i lati non appartenenti a  $T$  del taglio ottenuto eliminando un qualsiasi lato di  $T$ , allora deve soddisfare le cut optimality conditions. Quindi l'albero  $T$  è quello di costo minimo.

La dimostrazione del Teorema 1.6 dimostra anche l'equivalenza tra le cut optimality conditions e le path optimality conditions.

### 1.3.4 Algoritmo di Prim-Dijkstra

Le cut optimality conditions suggeriscono un algoritmo iterativo per la determinazione dell'albero di copertura di costo minimo di un grafo non orientato  $G(V, E)$ .

L'algoritmo descritto in questa sezione è stato proposto per la prima volta dal matematico ceco Vojtěch Jarník nel 1930. Poi è stato riscoperto indipendentemente da Robert Prim nel 1957 e da Edsger Dijkstra nel 1959. Per queste ragioni è spesso chiamato Algoritmo di Jarník, di Prim, di Prim–Jarník, Prim–Dijkstra oppure DJP.

L'Algoritmo 6 definisce un insieme iniziale  $S$  contenente un solo vertice del grafo (e.g.,  $S = \{1\}$ ), mentre l'albero iniziale  $T$  è vuoto (i.e.,  $T = \emptyset$ ). Ad ogni iterazione dell'algoritmo si espande la componente连通的  $S$  determinando il lato  $\{i, j\}$ , con  $i \in S$  e  $j \in V \setminus S$ , di costo minimo. L'algoritmo termina quando l'albero  $T$  ha  $n - 1$  archi e, quindi, copre tutti i vertici  $V$ . Avendo costruito l'albero nel rispetto delle cut optimality conditions,  $T$  è l'albero di copertura di costo minimo.

L'algoritmo richiede  $O(m)$  confronti ad ogni iterazione, quindi la sua complessità è pari perciò a  $O(nm)$ .

In Figura 1.34 è riportato un esempio di esecuzione della prima versione proposta dell'algoritmo di Prim-Dijkstra. L'algoritmo in Figura 1.34a inizializza

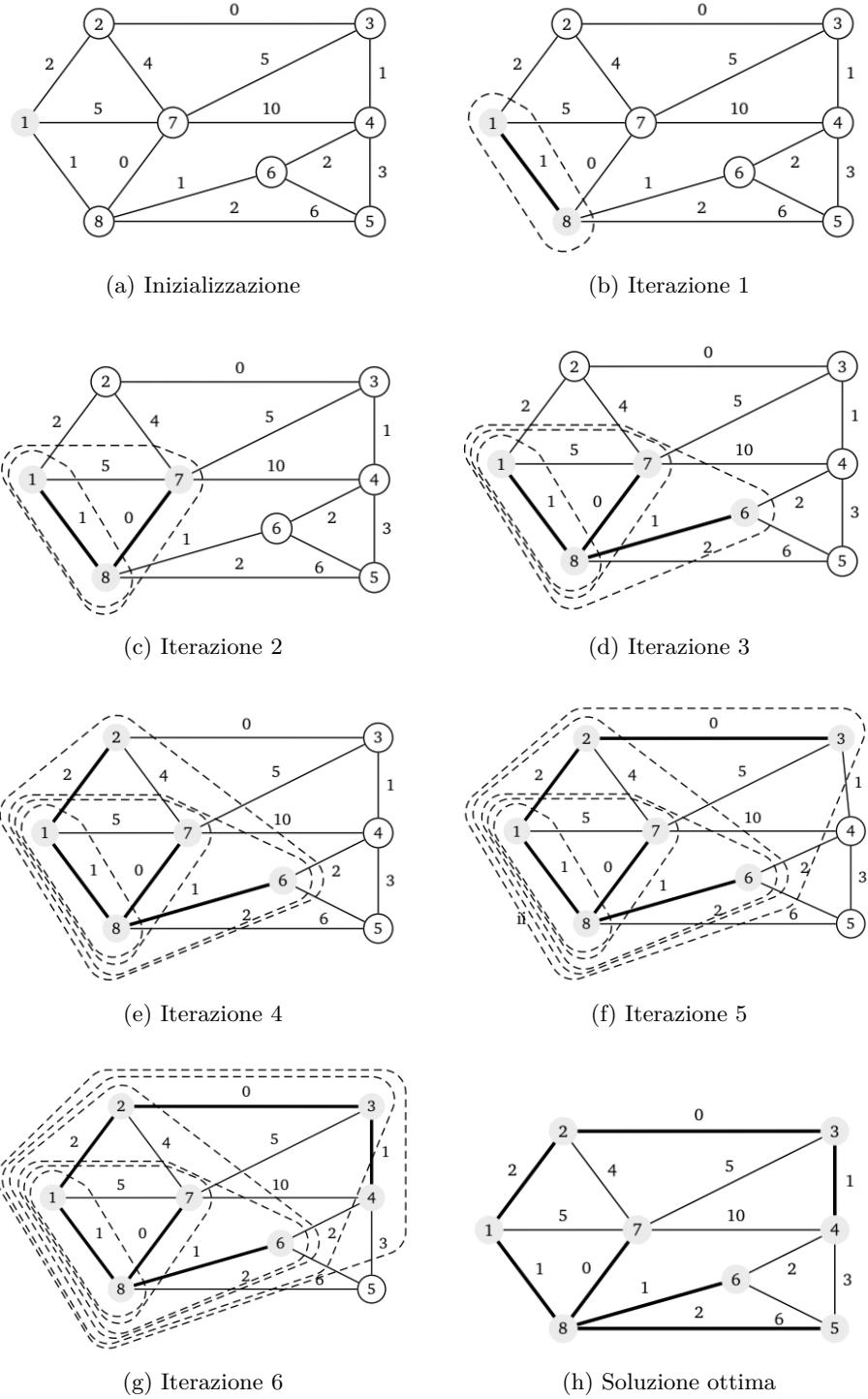


Figura 1.34: Un esempio di esecuzione dell’Algoritmo di Prim-Dijkstra

---

**Algorithm 6 Prim-Dijkstra (1<sup>a</sup> versione)**

---

**Input:** Grafo non orientato  $G(V, E)$  pesato con costi  $c_e$ ,  $\forall e \in E$ ;  
**Output:** Albero di copertura di costo minimo  $T^*$ ;

```

// Inizializzazione
 $T^* = \emptyset$ ;
 $S = \{1\}$ ;
// Costruisci l'albero di copertura
while  $|T^*| \neq n - 1$  do
    individua il lato  $\{i, j\} \in \delta(S)$  di costo minimo, con  $i \in S$  e  $j \notin S$ ;
     $T^* = T^* \cup \{\{i, j\}\}$ ;
     $S = S \cup \{j\}$ ;
end while
```

---

l'albero  $T^* = \emptyset$  e l'insieme  $S = \{1\}$ . Alla prima iterazione in Figura 1.34b seleziona il lato del taglio  $\delta(S) = \{\{1, 2\}, \{1, 7\}, \{1, 8\}\}$  di costo minimo che risulta essere  $\{1, 8\}$ , per cui aggiorna  $T^* = \{\{1, 8\}\}$  e  $S = \{1, 8\}$ . Alla seconda iterazione in Figura 1.34c il taglio è  $\delta(S) = \{\{1, 2\}, \{1, 7\}, \{8, 7\}, \{8, 6\}, \{8, 5\}\}$  e in questo caso il lato di costo minimo è  $\{8, 7\}$ , per cui aggiorna  $T^* = \{\{1, 8\}, \{8, 7\}\}$  e  $S = \{1, 8, 7\}$ . Alla terza iterazione in Figura 1.34d il taglio è  $\delta(S) = \{\{1, 2\}, \{2, 7\}, \{7, 4\}, \{8, 6\}, \{8, 5\}\}$  e il lato di costo minimo è  $\{8, 6\}$ , per cui aggiorna  $T^* = \{\{1, 8\}, \{8, 7\}, \{8, 6\}\}$  e  $S = \{1, 8, 7, 6\}$ . L'algoritmo procede eseguendo le rimanenti iterazioni mostrate nelle Figure 1.34e-1.34g fino a giungere alla soluzione ottima riportata in Figura 1.34h.

Il grafo dell'esempio in Figura 1.34 ha due diversi alberi di copertura di costo minimo. L'algoritmo di Prim-Dijkstra ne trova solo uno dei due e la soluzione finale dipenderà dall'ordine con cui verranno scelti i lati di pari costo.

Allo scopo di migliorare l'algoritmo, se si considera l'esecuzione dell'iterazione 4 in Figura 1.34e, per evitare di esaminare tutti i lati  $\{i, j\} \in E$  tali che  $i \in S$  e  $j \in V \setminus S$  (i.e.,  $(i, j) \in \delta(S)$ ) si potrebbe supporre di conoscere per ogni vertice  $j \in V \setminus S$  le seguenti *etichette*:

- $pred[j]$ : indica il vertice di  $S$  più vicino a  $j$ , ossia il vertice collegato con il lato  $\{pred[j], j\}$  del taglio  $\delta(S)$  con il costo minimo, i.e.,  $c_{\{pred[j], j\}} = \min_{i \in S} \{c_{\{i, j\}}\}$ ;
- $L[j]$ : corrisponde al costo del lato  $\{pred[j], j\}$ , i.e.,  $L[j] = c_{\{pred[j], j\}}$ .

In questo caso per l'iterazione 4 si avrà quanto riportato in Figura 1.35. Il lato di costo minimo  $\{i^*, j^*\} \in E$ , con  $i^* \in S$  e  $j^* \in V \setminus S$ , può essere calcolato determinando  $j^* \in V \setminus S$  tale che:

$$L[j^*] = \min_{j \in V \setminus S} \{L[j]\} \quad (1.12)$$

e ponendo quindi  $i^* = pred[j^*]$ . Questo metodo per calcolare, all'iterazione 4, il lato  $(i^*, j^*)$  richiede  $|V - S| \leq n$  confronti, quindi la complessità è  $O(n)$ . Le

etichette  $[pred[j], L[j]]$ ,  $\forall j \in V \setminus S$ , devono essere opportunamente aggiornate ad ogni iterazione dell'algoritmo; ciò può essere fatto con complessità  $O(n)$ . La seconda versione dell'algoritmo che ne risulta ha perciò una complessità pari a  $O(n^2)$ .

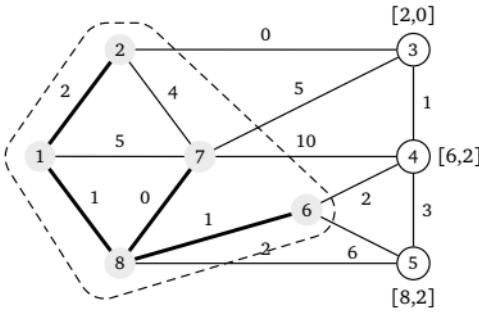


Figura 1.35: Miglioramento dell'algoritmo di Prim usando delle etichette

L'algoritmo 7 per calcolare l'albero di costo minimo  $T$  utilizza le seguenti strutture dati:

- $flag[i] = \begin{cases} 1, & \text{se } i \in S \\ 0, & \text{se } i \in V \setminus S \end{cases}$
- $L[j] = \min\{c_{ij} : i \in S\}$ , per ogni  $j \notin S$ ;
- $pred[j] = \begin{cases} \operatorname{argmin}\{c_{ij} : i \in S\}, & \text{se } j \notin S \\ \text{predecessore di } j \text{ in } T, & \text{se } j \in S \end{cases}$

In Figura 1.36 è mostrata l'esecuzione della seconda versione dell'algoritmo di Prim-Dijkstra sullo stesso esempio utilizzato in Figura 1.34. L'algoritmo in Figura 1.36a inizializza l'albero  $T^* = \emptyset$ , l'insieme  $S = \{1\}$  e le etichette  $[pred[j], L[j]]$  per ogni vertice  $j \in V$ .

Alla prima iterazione in Figura 1.36b l'algoritmo identifica il vertice  $j \notin S$  con l'etichetta  $L[j]$  minima, che in questo caso è il vertice  $j = 8$ . Quindi, il lato da inserire nell'albero di copertura di costo minimo è  $\{pred[8], 8\}$ , ossia il lato  $\{1, 8\}$ , per cui  $T^* = \{\{1, 8\}\}$  e  $S = \{1, 8\}$ . Alla seconda iterazione in Figura 1.36c il vertice con l'etichetta  $L[j]$  minima è  $j = 7$ . Il lato  $\{pred[7], 7\} = \{8, 7\}$  è aggiunto all'albero di copertura, i.e.,  $T^* = \{\{1, 8\}, \{8, 7\}\}$  e  $S = \{1, 8, 7\}$ . Alla terza iterazione in Figura 1.36d il vertice con l'etichetta  $L[j]$  minima è  $j = 6$ . Il lato  $\{pred[6], 6\} = \{8, 6\}$  è aggiunto all'albero di copertura, i.e.,  $T^* = \{\{1, 8\}, \{8, 7\}, \{8, 6\}\}$  e  $S = \{1, 8, 7, 6\}$ . L'algoritmo procede eseguendo le rimanenti iterazioni mostrate nelle Figure 1.36e-1.36g fino a giungere alla soluzione ottima riportata in Figura 1.36h.

---

**Algorithm 7 Prim-Dijkstra (2<sup>a</sup> versione)**

---

**Input:** Grafo non orientato  $G(V, E)$  pesato con costi  $c_e$ ,  $\forall e \in E$ ;  
**Output:** Albero di copertura di costo minimo  $T^*$  definito da  $pred[j]$ ,  $j \in V$ ;

```

// Inizializza le strutture dati partendo da  $S = \{1\}$ 
flag[1] = 1;
pred[1] = 1;
for  $j = 2$  to  $n$  do
    flag[j] = 0;
     $L[j] = c_{1j}$ ;
    pred[j] = 1;
end for
// Seleziona gli  $n - 1$  lati dell'albero
for  $k = 1$  to  $n - 1$  do
    // Sceglie il lato minimo in  $\delta(S)$ 
    min =  $+\infty$ ;
    for  $j = 2$  to  $n$  do
        if ( $flag[j] = 0$ ) and ( $L[j] < min$ ) then
            min =  $L[j]$ ;  $h = j$ ;
        end if
    end for
    // Include il vertice  $h$  in  $S$  e aggiorna  $L[j]$  e  $pred[j]$  per ogni  $j \notin S$ 
    flag[h] = 1;
    for  $j = 2$  to  $n$  do
        if ( $flag[j] = 0$ ) and ( $c_{hj} < L[j]$ ) then
             $L[j] = c_{hj}$ ; pred[j] = h;
        end if
    end for
end for

```

---

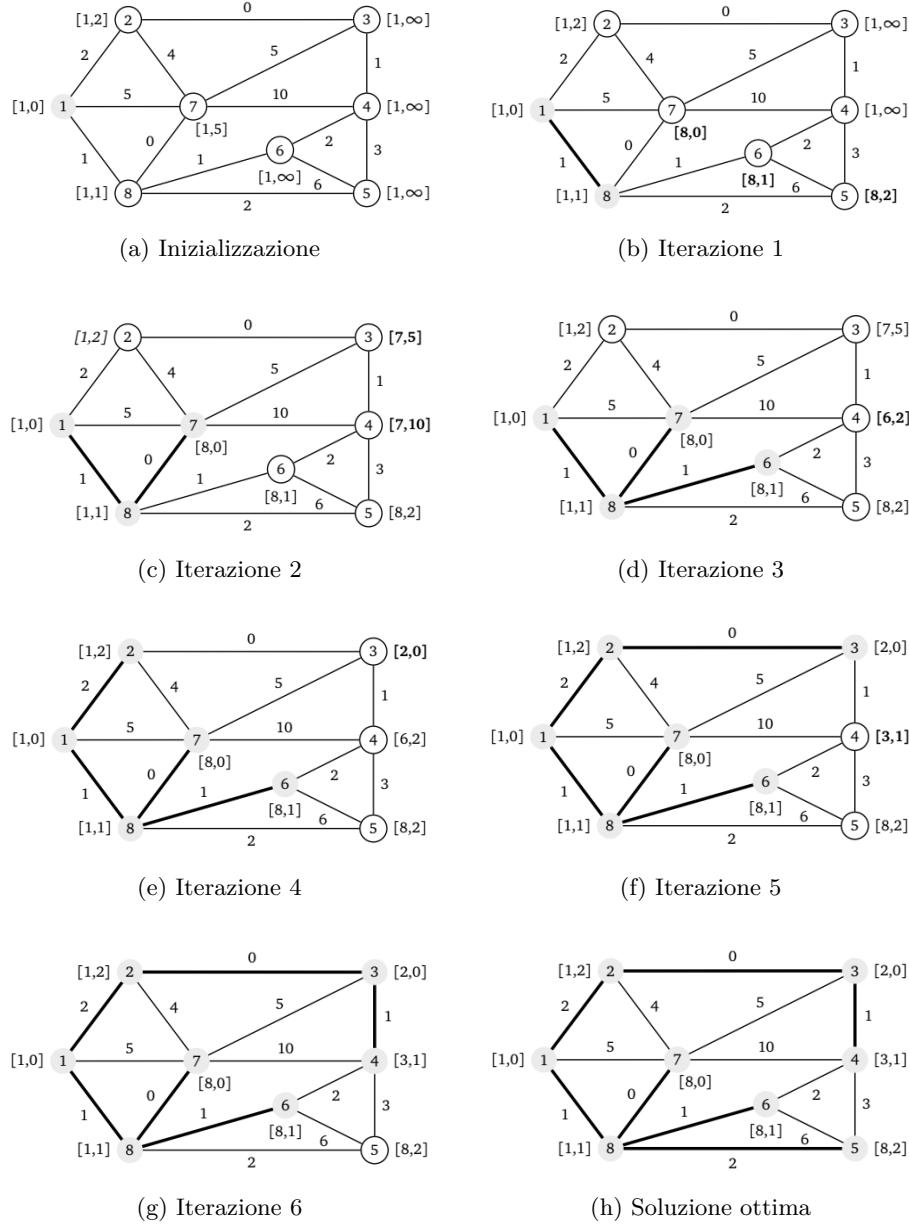


Figura 1.36: Un esempio di esecuzione della seconda versione dell’Algoritmo di Prim-Dijkstra

### 1.3.5 Algoritmo di Kruskal

Le *path optimality conditions* suggeriscono un nuovo algoritmo iterativo per la determinazione dell'albero di copertura  $T$  di costo minimo di un grafo non orientato  $G(V, E)$ .

Il principio di funzionamento dell'algoritmo di Kruskal, descritto nell'Algoritmo 8, è piuttosto semplice. I lati sono inseriti nell'albero  $T$  per costi crescenti, quindi devono essere ordinati. Ciascun lato, considerato in ordine di costo crescente, è inserito in  $T$  se non forma un ciclo con i lati già in  $T$ , altrimenti viene scartato. Se un lato  $e = \{i, j\}$  non viene inserito perché forma un ciclo, sicuramente non costerà di meno di quelli già inseriti nel path da  $i$  a  $j$ , quindi rispetta le *path optimality conditions*.

---

#### Algorithm 8 Kruskal (1<sup>a</sup> versione)

---

**Input:** Grafo non orientato  $G(V, E)$  pesato con costi  $c_e, \forall e \in E$ ;  
**Output:** Albero di copertura di costo minimo  $T^*$ ;

```

// Inizializza strutture dati e ordina i lati
 $T^* = \emptyset$ ;
Ordina i lati per costi crescenti ( $E = \{e_1, \dots, e_m\}: c(e_i) \leq c(e_{i+1})$ );
// Costruisci l'albero di copertura di costo minimo
 $h = 1$ ;
while ( $|T^*| < n - 1$ ) and ( $h < m$ ) do
    Se il lato  $e_h$  non chiude un ciclo in  $T^*$ , allora  $T^* = T^* \cup \{e_h\}$ ;
     $h = h + 1$ ;
end while
```

---

Il costo computazionale maggiore è dovuto all'ordinamento degli  $m$  lati, quindi la complessità dell'algoritmo è  $O(m \log m)$ .

L'Algoritmo 8 rappresenta un'implementazione di base dell'algoritmo di Kruskal che può essere migliorata. Per definire una modalità efficiente per decidere se aggiungere a  $T^*$  il lato  $e_k = (\alpha, \beta)$ , si hanno i seguenti casi:

- (a)  $\alpha$  e  $\beta$  fanno parte di due sottoalberi distinti (vedi Figura 1.37a);
- (b)  $\alpha$  e  $\beta$  appartengono al medesimo sottoalbero, quindi l'introduzione di  $e_k$  chiude un ciclo (vedi Figura 1.37b).

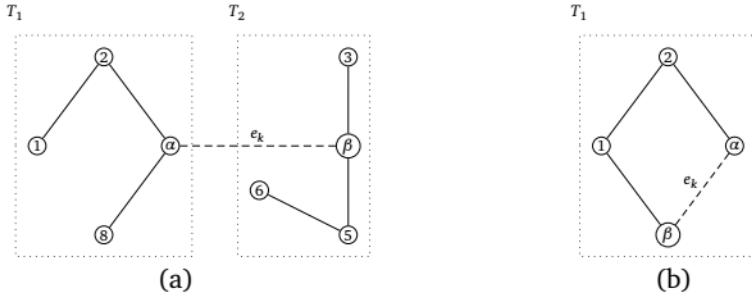


Figura 1.37: Casi da considerare per decidere se aggiungere un lato all’albero di copertura

Per verificare se per il lato  $e_k = (\alpha, \beta)$  si ha il caso (a) o (b) si associa ad ogni vertice  $i \in V$  una etichetta  $comp[i]$  che rappresenta l’indice associato al sottoalbero a cui  $i$  appartiene. Nell’esempio riportato in Figura 1.37a si ha:

- $comp[1] = comp[2] = comp[8] = comp[\alpha] = 1$
- $comp[3] = comp[5] = comp[6] = comp[\beta] = 2$

Mentre, nell’esempio in Figura 1.37b si ha:

- $comp[1] = comp[2] = comp[\alpha] = comp[\beta] = 1$

Perciò, quando si deve stabilire se il lato  $e_k = (\alpha, \beta)$  può essere inserito nell’albero di supporto di costo minimo, sarà sufficiente svolgere il seguente test:

- Se  $comp[\alpha] \neq comp[\beta]$  allora si ha il caso (a), ovvero  $\alpha$  e  $\beta$  fanno parte di due sottoalberi distinti.
- Se  $comp[\alpha] = comp[\beta]$  allora si ha il caso (b), ovvero  $\alpha$  e  $\beta$  fanno parte del medesimo sottoalbero.

Nel caso (a) si provvederà a inserire il lato  $e_k = (\alpha, \beta)$ ) nell’albero  $T_1$  e a unire l’albero  $T_1$  a  $T_2$ . Per unire gli alberi  $T_1$  e  $T_2$  si dovrà aggiornare l’etichetta  $comp[i]$  dei vertici in  $T_2$  con quella dei vertici in  $T_1$ .

L’Algoritmo 9 riassume l’algoritmo di Kruskal che utilizza la struttura dati  $comp[j]$ .

In Figura 1.38 è riportato un esempio dell’esecuzione dell’algoritmo di Kruskal. L’algoritmo inizializza  $comp[j] = j$ , in quanto ogni vertice rappresenta una singola componente连通 (vedi Figura 1.38a). Una volta ordinati i lati per costo crescente, l’algoritmo prova a inserire a turno i lati. Nella prima iterazione, riportata in Figura 1.38b, l’algoritmo seleziona il lato  $\{2, 3\}$  che ha il costo minore. Siccome i vertici 2 e 3 hanno  $comp[2] \neq comp[3]$ , l’algoritmo lo inserisce nell’albero di copertura di costo minimo e aggiorna le etichette  $comp[3] = 2$ . Dopodiché, l’algoritmo ripete l’operazione per i successivi lati. Si noti che all’iterazione 5, quando si inserisce il lato  $\{6, 8\}$  è necessario aggiornare tutte le etichette del sottoalbero contenente i lati  $\{\{1, 2\}, \{1, 8\}, \{6, 8\}, \{7, 8\}\}$ . In Figura 1.38h è riportata la soluzione ottima.

---

**Algorithm 9 Kruskal (2<sup>a</sup> versione)**

---

**Input:** Grafo non orientato  $G(V, E)$  pesato con costi  $c_e$ ,  $\forall e \in E$ ;  
**Output:** Albero di copertura di costo minimo  $T^*$ ;

```

// Inizializza le strutture dati e ordina i lati
Ordina i lati per costi crescenti ( $E = \{e_1, \dots, e_m\} : c(e_i) \leq c(e_{i+1})$ );
k = 0; h = 0;
// Ogni vertice rappresenta una componente distinta
for i = 1 to n do
    comp[i] = i;
end for
// Costruisci l'albero di copertura di costo minimo
while (k < n - 1) and (h < m) do
    Seleziona il lato  $e_h = \{i, j\}$ ;
    C1 = comp[i]; C2 = comp[j];
    if (C1 ≠ C2) then
        Inserisci  $e_h$  nell'albero  $T^*$ ; k=k+1;
        // Unisci le componenti C1 e C2
        for q = 1 to n do
            if (comp[q] = C2) then
                comp[q] = C1;
            end if
        end for
    end if
    h = h + 1;
end while
if (k ≠ n - 1) then
    Il grafo  $G$  è disconnesso;
end if

```

---

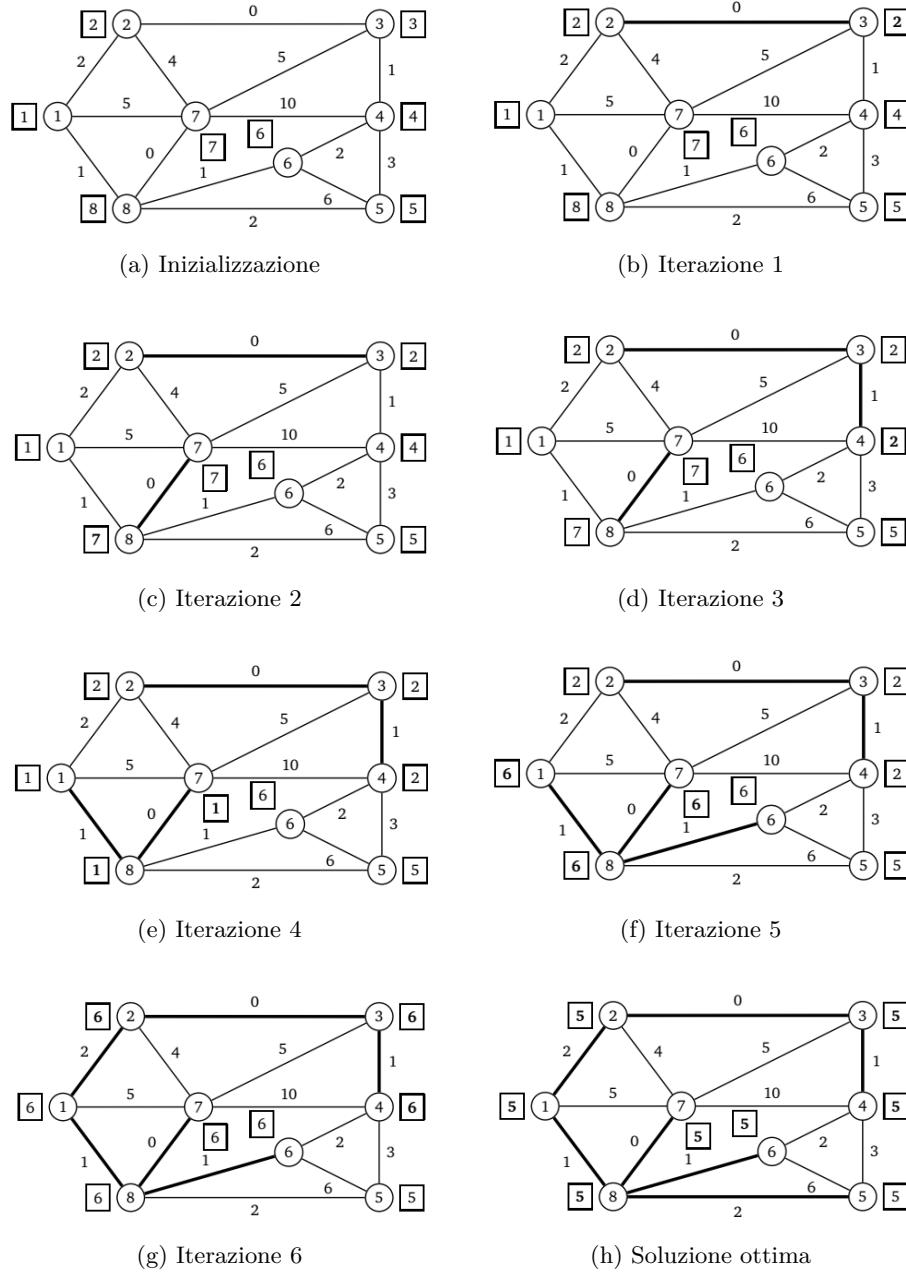


Figura 1.38: Un esempio di esecuzione della seconda versione dell'Algoritmo di Kruskal

## 1.4 Flusso massimo

Dato un grafo direzionato  $G = (N, A)$ , in cui ad ogni arco  $(i, j) \in A$  è associata una capacità  $u_{ij} \geq 0$ , si vuole determinare il *flusso massimo* che può essere inviato dal vertice *origine*  $s \in N$  al vertice *destinazione*  $t \in N$ .

In Figura 1.39 è proposto un esempio di un'istanza del problema del flusso massimo in cui per ogni arco  $(i, j) \in A$  è riportata la capacità  $u_{ij}$ , mentre l'origine è il vertice  $s = 1$  e la destinazione è  $t = 9$ .

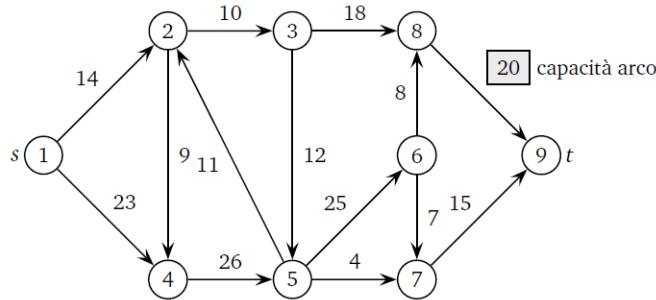


Figura 1.39: Esempio di istanza per il Problema del Flusso Massimo

### 1.4.1 Formulazione matematica

Per formulare il problema del flusso massimo sono necessarie le seguenti variabili decisionali:

- $x_{ij}$ : quantità di flusso assegnata a ciascun arco  $(i, j) \in A$ ;
- $v$ : flusso introdotto all'origine  $s$  e ricevuto alla destinazione  $t$ .

Il problema del flusso massimo può essere formulato come segue:

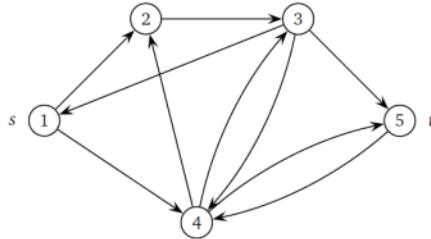
$$(MF) \quad z_{MF} = \max v \tag{1.13}$$

$$\text{s.t. } \sum_{j \in \Gamma_i} x_{ij} - \sum_{j \in \Gamma_i^{-1}} x_{ji} = \begin{cases} v, & i = s \\ -v, & i = t \\ 0, & i \neq N \setminus \{s, t\} \end{cases} \tag{1.14}$$

$$0 \leq x_{ij} \leq u_{ij}, \quad (i, j) \in A \tag{1.15}$$

dove  $\Gamma_i = \{j : (i, j) \in A\}$  e  $\Gamma_i^{-1} = \{j : (j, i) \in A\}$ . La funzione obiettivo (1.13) massimizza flusso introdotto all'origine  $s$  e ricevuto alla destinazione  $t$ . I vincoli (1.14) garantiscono la conservazione del flusso ad ogni vertice  $i \in N$ , tenendo conto che al vertice di origine  $s$  deve entrare il flusso  $v$  che poi uscirà al vertice destinazione  $t$ . I vincoli (1.15) assicurano che il flusso assegnato a ciascun arco  $(i, j) \in A$  sia non negativo e non superi la capacità  $u_{ij}$ .

Nella Figura 1.40 è riportato un esempio di un grafo con i corrispondenti vincoli di conservazione del flusso.



Vertice	$x_{12}$	$x_{14}$	$x_{23}$	$x_{31}$	$x_{34}$	$x_{35}$	$x_{42}$	$x_{43}$	$x_{45}$	$x_{54}$	$v$
$s = 1$	$x_{12} + x_{14}$		$-x_{31}$								$= v$
2	$-x_{12}$	$+x_{23}$					$-x_{42}$				$= 0$
3		$-x_{23} + x_{31} + x_{34} + x_{35}$					$-x_{43}$				$= 0$
4		$-x_{14}$		$-x_{34}$			$+x_{42} + x_{43} + x_{45} - x_{54}$				$= 0$
$t = 5$					$-x_{35}$			$-x_{45} + x_{54}$			$= -v$

Figura 1.40: Esempio di vincoli di conservazione del flusso

### 1.4.2 Assunzioni e definizioni

Introduciamo delle assunzioni e definizioni utili nel proseguo di questa sezione quando saranno descritte le condizioni di ottimalità e gli algoritmi di soluzione.

Si assume che il grafo  $G(N, A)$  sia orientato e che abbia  $n$  vertici e con  $m$  archi. Inoltre, si assume che tutte le capacità  $u_{ij}$  sono intere non negative (i.e.,  $u_{ij} \geq 0$ ) e che il grafo non contiene un cammino orientato dal vertice  $s$  al vertice  $t$  composto da soli archi di capacità infinita, per cui la soluzione ottima è limitata. Si denota con  $U$  la capacità massima degli archi, i.e.,  $U = \max\{u_{ij} : (i, j) \in A\}$ .

Per semplicità, ma senza perdere di generalità, si ipotizza che se in  $A$  è contenuto l'arco  $(i, j)$  allora è contenuto anche l'arco  $(j, i)$ . Si noti che questa assunzione non limita la possibilità di utilizzo dei risultati che seguiranno, in quanto uno dei due archi potrebbe avere capacità nulla.

Il *grafo residuo*  $G(x)$  corrispondente al flusso  $x$  nel grafo  $G$  è ottenuto sostituendo ogni arco  $(i, j) \in A$  con un arco “diretto”  $(i, j)$  e un arco “inverso”  $(j, i)$  con una *capacità residua* pari a  $r_{ij} = u_{ij} - x_{ij}$  e  $r_{ji} = x_{ij}$ .

Dato un flusso  $x$  nel grafo  $G$ , un *cammino aumentante* è un cammino da  $s$  a  $t$  nel grafo residuo  $G(x)$ . In alternativa, e in modo equivalente, si può definire il cammino aumentante come il cammino non orientato da  $s$  a  $t$  nel grafo  $G$ , in cui si può aumentare il flusso di almeno una unità negli archi percorsi nella direzione originaria e diminuire il flusso di almeno una unità negli archi percorsi nella direzione contraria rispetto a quella originaria.

Si definisce *Taglio  $s-t$*  una partizione dei vertici  $N$  del grafo  $G$  in due sottoinsiemi  $S$  e  $\bar{S} = N \setminus S$  tali che  $s \in S$  e  $t \in \bar{S}$ . La capacità di un taglio  $s-t$  è data

da:

$$u(S, \bar{S}) = \sum_{i \in S} \sum_{j \in \bar{S}} u_{ij} = \sum_{(i,j) \in \Gamma(S, \bar{S})} u_{ij}$$

dove  $\Gamma(S, \bar{S}) = \{(i, j) \in A : i \in S, j \in \bar{S}\}$ . Si noti che il flusso che attraversa gli archi del taglio appartenenti all'insieme  $\Gamma(\bar{S}, S) = \{(i, j) \in A : i \in \bar{S}, j \in S\}$  riduce il flusso da  $s$  a  $t$ .

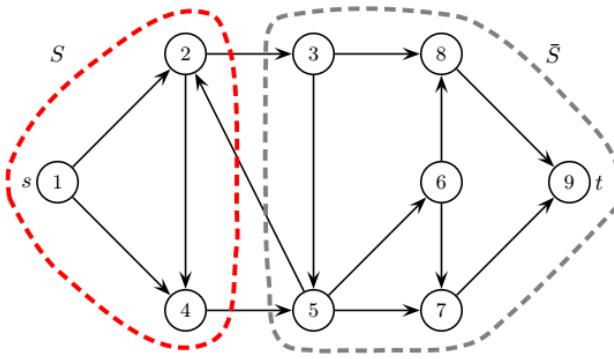


Figura 1.41: Esempio di taglio  $s-t$

In Figura 1.41 è mostrato un esempio di taglio  $s-t$  in cui  $S = \{1, 2, 4\}$  e  $\bar{S} = V \setminus S = \{3, 5, 6, 7, 8, 9\}$ , per cui si ha  $\Gamma(S, \bar{S}) = \{(2, 3), (4, 5)\}$  e  $\Gamma(\bar{S}, S) = \{(5, 2)\}$ . Si noti che il taglio è un taglio  $s-t$  perché  $s \in S$  e  $t \in \bar{S}$ .

### 1.4.3 Condizioni di ottimalità

In questa sezione vengono introdotti dei teoremi utili alla definizione di condizioni di ottimalità che sono alla base degli algoritmi di soluzione descritti in seguito, in particolare l'Algoritmo di Ford Fulkerson descritto in Sezione 1.4.4.

**Teorema 1.7.** *Dato un flusso ammissibile  $x$  pari a  $v$ , per ogni taglio  $s-t$   $(S, \bar{S})$  il flusso che attraversa il taglio è dato da:*

$$v = \sum_{(i,j) \in \Gamma(S, \bar{S})} x_{ij} - \sum_{(i,j) \in \Gamma(\bar{S}, S)} x_{ij}$$

**Dim.: 1.7.** *Sommendo i vincoli di conservazione del flusso per tutti i nodi di  $S$*

si ottiene quanto segue:

$$\begin{aligned}
 v &= \sum_{h \in S} \left[ \sum_{j \in \Gamma_h} x_{hj} - \sum_{j \in \Gamma_h^{-1}} x_{jh} \right] = \sum_{h \in S} \sum_{j \in \Gamma_h} x_{hj} - \sum_{h \in S} \sum_{j \in \Gamma_h^{-1}} x_{jh} \\
 &= \left[ \sum_{(i,j) \in A(S)} x_{ij} + \sum_{(i,j) \in \Gamma(S, \bar{S})} x_{ij} \right] - \left[ \sum_{(i,j) \in A(S)} x_{ij} + \sum_{(i,j) \in \Gamma(\bar{S}, S)} x_{ij} \right] \\
 &= \sum_{(i,j) \in \Gamma(S, \bar{S})} x_{ij} - \sum_{(i,j) \in \Gamma(\bar{S}, S)} x_{ij}
 \end{aligned}$$

dove  $A(S) = \{(i, j) \in A : i \in S, j \in S\}$ .

**Teorema 1.8.** Il valore  $v$  di ogni flusso ammissibile  $x$  in  $G$  è minore o uguale alla capacità  $u(S, \bar{S})$  di un qualunque taglio  $s-t$ .

**Dim.: 1.8.** Per il Teorema precedente il valore  $v$  del flusso  $x$  è dato da:

$$v = \sum_{(i,j) \in \Gamma(S, \bar{S})} x_{ij} - \sum_{(i,j) \in \Gamma(\bar{S}, S)} x_{ij}$$

Un limite superiore al valore di  $v$  si ottiene sostituendo  $x_{ij}$  con  $u_{ij}$ , per ogni  $(i, j) \in \Gamma(S, \bar{S})$ , e  $x_{ij}$  con 0, per ogni  $(i, j) \in \Gamma(\bar{S}, S)$ . Per cui:

$$v \leq \sum_{(i,j) \in \Gamma(S, \bar{S})} u_{ij} = u(S, \bar{S})$$

In particolare, anche il valore del flusso massimo è minore o uguale alla capacità del taglio  $s-t$  di capacità minima.

**Teorema 1.9** (Augmenting Path Theorem). Sia  $x$  un flusso ammissibile in  $G$  da  $s$  a  $t$  e sia  $G(x)$  il corrispondente grafo residuo. Il flusso  $x$  è massimo se e solo se non esiste in  $G(x)$  un cammino da  $s$  a  $t$  (i.e., non esiste un cammino aumentante da  $s$  a  $t$  in  $G$ ).

**Dim.: 1.9.** (a) Se il flusso  $x$  è massimo, allora non esiste un cammino  $P$  in  $G(x)$ .

Se per assurdo esistesse un cammino aumentante  $P$  in  $G(x)$  tale che  $\delta = \min\{r_{ij} : \forall (i, j) \in P\} > 0$ , allora sarebbe possibile aggiornare il flusso per ogni arco  $(i, j) \in P$ :

- $x_{ij} = x_{ij} + \delta$ , se  $(i, j)$  è un arco diretto;
- $x_{ji} = x_{ji} - \delta$ , se  $(i, j)$  è un arco inverso;

Il nuovo flusso aggiornato  $x$  è un flusso ammissibile di valore  $v + \delta$ , per cui il flusso  $x$  prima dell'aggiornamento non era ottimo.

(b) Se non esiste un cammino  $P$  in  $G(x)$ , allora il flusso  $x$  è massimo.

Se non esiste un cammino  $P$  in  $G(x)$ , allora esiste un taglio  $(S, \bar{S})$ , dove  $\bar{S} = V \setminus S$ , nel grafo residuo  $G(x)$  tale che  $\Gamma(S, \bar{S}) = \emptyset$ . Nella rete originale si ha allora che:

- ogni arco  $(i, j) \in \Gamma(S, \bar{S})$  è saturo (i.e.,  $x_{ij} = u_{ij}$ );
- ogni arco  $(i, j) \in \Gamma(\bar{S}, S)$  è scarico (i.e.,  $x_{ij} = 0$ ).

Quindi, il valore del flusso per il taglio  $(S, \bar{S})$  è uguale a:

$$v = \sum_{(i,j) \in \Gamma(S, \bar{S})} x_{ij} - \sum_{(i,j) \in \Gamma(\bar{S}, S)} x_{ij} = \sum_{(i,j) \in \Gamma(S, \bar{S})} u_{ij} = u(S, \bar{S})$$

e dal teorema precedente si ha che il flusso  $x$  è ottimo e  $(S, \bar{S})$  è il taglio  $s-t$  di capacità minima.

Un ulteriore importante teorema che ne deriva è il seguente:

**Teorema 1.10** (Max-Flow Min-Cut Theorem). *Il flusso massimo da  $s$  a  $t$  in  $G$  è uguale alla capacità del taglio  $s-t$  di capacità minima.*

Inoltre, una importante proprietà del problema del flusso massimo è definita dal seguente teorema.

**Teorema 1.11** (Integrality Theorem). *Se tutti gli archi hanno una capacità  $u_{ij}$  intera, allora il problema del flusso massimo ha una soluzione intera.*

#### 1.4.4 Algoritmo di Ford-Fulkerson

Un algoritmo per risolvere il problema del flusso massimo è l'Algoritmo di Ford-Fulkerson che è basato sul *Augmenting Path Theorem*.

##### Algoritmo di Ford-Fulkerson (versione 1)

Step 1. [Inizializza]

Poni  $x_{ij} = 0$ , per ogni  $(i, j) \in A$ , e  $v = 0$ .

Step 2. [Determina Cammino Aumentante]

Costruisci il grafo residuo  $G(x)$ .

Trova un cammino da  $s$  a  $t$  in  $G(x)$ .

Se non esiste un cammino da  $s$  a  $t$ , allora il flusso  $x$  è massimo, quindi STOP.

Step 3. [Aumenta il Flusso  $x$ ]

Sia  $P$  il cammino da  $s$  a  $t$  trovato e ponì  $\delta = \min\{r_{ij} : (i, j) \in P\}$ .

Per ogni  $(i, j) \in P$ :

- (i) Se  $(i, j)$  corrisponde all'arco  $(i, j) \in A$ :  $x_{ij} = x_{ij} + \delta$ ;
- (ii) Se  $(i, j)$  corrisponde all'arco  $(j, i) \in A$ :  $x_{ji} = x_{ji} - \delta$ ;

Aggiorna il flusso  $v = v + \delta$ .

Ritorna allo Step 2.

L'algoritmo di Ford-Fulkerson può essere riformulato senza usare il grafo residuo  $G(x)$  esplicitamente.

Per consentire il calcolo del cammino aumentante nel grafo residuo senza doverlo generare si possono usare delle etichette  $[pred[j], \delta_j]$  per ciascun vertice  $j \in N$ .  $pred[j]$  è il “predecessore” del vertice  $j$  nel cammino aumentante:

- Se  $pred[j] = i > 0$ , allora nell'arco  $(i, j) \in A$  il flusso potrebbe essere aumentato di  $\delta_j$ .
- Se  $pred[j] = -i < 0$ , allora nell'arco  $(j, i) \in A$  il flusso potrebbe essere diminuito di  $\delta_j$ .

Ciascun  $\delta_j$  indica il massimo aumento consentito nel cammino da  $s$  fino a  $j$ . Mentre, l'effettivo aumento del flusso nel cammino da  $s$  a  $t$  è determinato da  $\delta_t$ .

### Algoritmo di Ford-Fulkerson (versione 2)

Step 1. *[Inizializza]*

Poni  $x_{ij} = 0$ , per ogni  $(i, j) \in A$ , e  $v = 0$ .

Step 2. *[Determina Cammino Aumentante]*

Dichiara il vertice  $s$  non espanso con etichetta  $[+s, \infty]$ , mentre dichiara tutti gli altri vertici non etichettati (e non espansi).

- (i) Se tutti i nodi etichettati sono già espansi, allora il flusso  $v$  è massimo (i.e., la soluzione  $x$  è ottima), quindi STOP.
- (ii) Se esiste un vertice  $i$  etichettato ma non ancora espanso, provvedi a espanderlo:
  - Per ogni vertice  $j \in \Gamma_i$  non etichettato per cui  $x_{ij} < u_{ij}$  assegna l'etichetta  $[+i, \delta_j]$ , dove  $\delta_j = \min\{u_{ij} - x_{ij}, \delta_i\}$ ;
  - Per ogni vertice  $j \in \Gamma_i^{-1}$  non etichettato per cui  $x_{ji} > 0$  assegna l'etichetta  $[-i, \delta_j]$ , dove  $\delta_j = \min\{x_{ji}, \delta_i\}$ .
- (iii) Se  $t$  non risulta etichettato torna a passo (i).

Step 3. *[Aumenta il Flusso x]*

Sia  $P$  il cammino aumentante dal vertice  $s$  al vertice  $t$  (ricostruito usando le etichette).

Per ogni  $(i, j) \in P$ :

- (i) Se l'arco  $(i, j)$  è percorso nel suo verso originario:  $x_{ij} = x_{ij} + \delta_t$ ;
- (ii) Se l'arco  $(i, j)$  è percorso nel verso contrario:  $x_{ij} = x_{ij} - \delta_t$ .

Aggiorna il flusso  $v = v + \delta_t$ .

Annulla tutte etichette e ritorna allo Step 2.

Quando l’Algoritmo di Ford-Fulkerson termina, l’insieme dei vertici etichettati  $S$  e quello dei vertici non etichettati  $\bar{S}$  costituiscono un *taglio di capacità minima*.

L’algoritmo di Ford-Fulkerson ha complessità  $O(nmU)$ , perché ad ogni iterazione il flusso aumenta di  $\delta \geq 1$  e il valore del flusso è limitato superiormente da  $nU$ , mentre il calcolo di un cammino aumentante ha complessità  $O(m)$ .

Nel 1972 Edmonds e Karp hanno proposto alcune varianti dell’algoritmo di Ford-Fulkerson:

- calcolando il cammino aumentante di capacità massima la complessità è pari a  $O(nm \log U)$ ;
- calcolando il cammino aumentante di cardinalità minima la complessità è pari a  $O(n^2m)$ .

Esistono altri algoritmi con “minore” complessità, per esempio:

- FIFO preflow-push con complessità  $O(n^3)$ ;
- Highest-label preflow push con complessità  $O(n^2\sqrt{m})$ ;
- Excess scaling con complessità  $O(nm + n^2 \log U)$ .

In Figura 1.42 è riportato un esempio di esecuzione della prima versione dell’Algoritmo di Ford-Fulkerson. Questa versione usando esplicitamente il grafo residuo inizializzando la soluzione e il flusso iniziale a  $x = 0$  e  $v = 0$ . Nella prima iterazione (Figura 1.42a) il flusso può essere aumentato di  $\delta = \min\{r_{13}, r_{34}\} = 4$  unità sul cammino aumentante  $P = (1, 3, 4)$  e  $v = v + \delta = 0 + 4 = 4$ . Dopo aver aggiornato il grafo residuo  $G(x)$ , nella seconda iterazione (Figura 1.42b) si cerca un nuovo cammino aumentante. Il flusso può essere aumentato di  $\delta = \min\{r_{12}, r_{23}, r_{34}\} = 1$  unità sul cammino aumentante  $P = (1, 2, 3, 4)$  e  $v = v + \delta = 4 + 1 = 5$ . Nella terza iterazione si aggiorna nuovamente il grafo residuo  $G(x)$  (Figura 1.42c) e si scopre che il flusso può essere aumentato di  $\delta = \min\{r_{12}, r_{24}\} = 1$  unità sul cammino aumentante  $P = (1, 2, 4)$  e  $v = c + \delta = 5 + 1 = 6$ . Nell’ultima iterazione (Figura 1.42d) si aggiorna un’ultima volta il grafo residuo  $G(x)$  e si scopre che il grafo residuo non contiene alcun cammino aumentante. Quindi, il flusso massimo è  $v = 6$  e il taglio di capacità minima è definito da  $S = \{1\}$  e  $\bar{S} = \{2, 3, 4\}$ ; inoltre, si può notare che  $v = \sum_{i \in S} \sum_{j \in \bar{S}} u_{ij} = 6$ .

In Figura 1.43 è riportata l’esecuzione della prima iterazione della seconda versione dell’Algoritmo di Ford-Fulkerson che non usa esplicitamente il grafo residuo e che è stato applicato all’esempio in Figura 1.39. La prima iterazione (Figura 1.43a) parte inizializzando la soluzione e il flusso fissando  $x = 0$  e  $v = 0$  e definendo la sola etichetta del vertice  $s = 1$  impostando il valore  $[1, +\infty]$ . Nel secondo passo della prima iterazione (Figura 1.43b) si procede a espandere il vertice 1 definendo le etichette dei vertici 2 e 4. Nel terzo passo (Figura 1.43c) tra i vertici etichettati e non espansi (i.e., 2 e 4), viene espanso il vertice 2 considerando i vertici adiacenti 3 e 4, però quest’ultimo è già stato etichettato. Dopodiché, tra i nodi etichettati e non espansi (i.e., 3 e 4), viene espanso il nodo

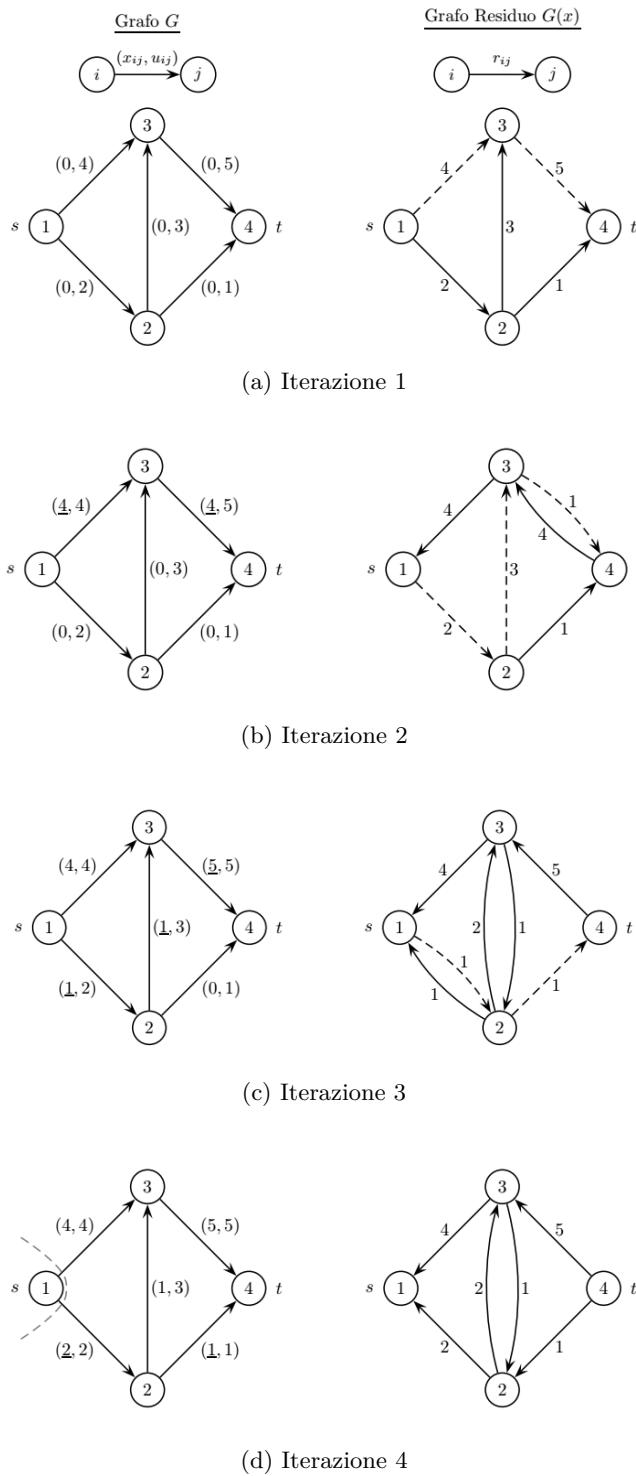


Figura 1.42: Un esempio di esecuzione della prima versione dell’Algoritmo di Ford-Fulkerson

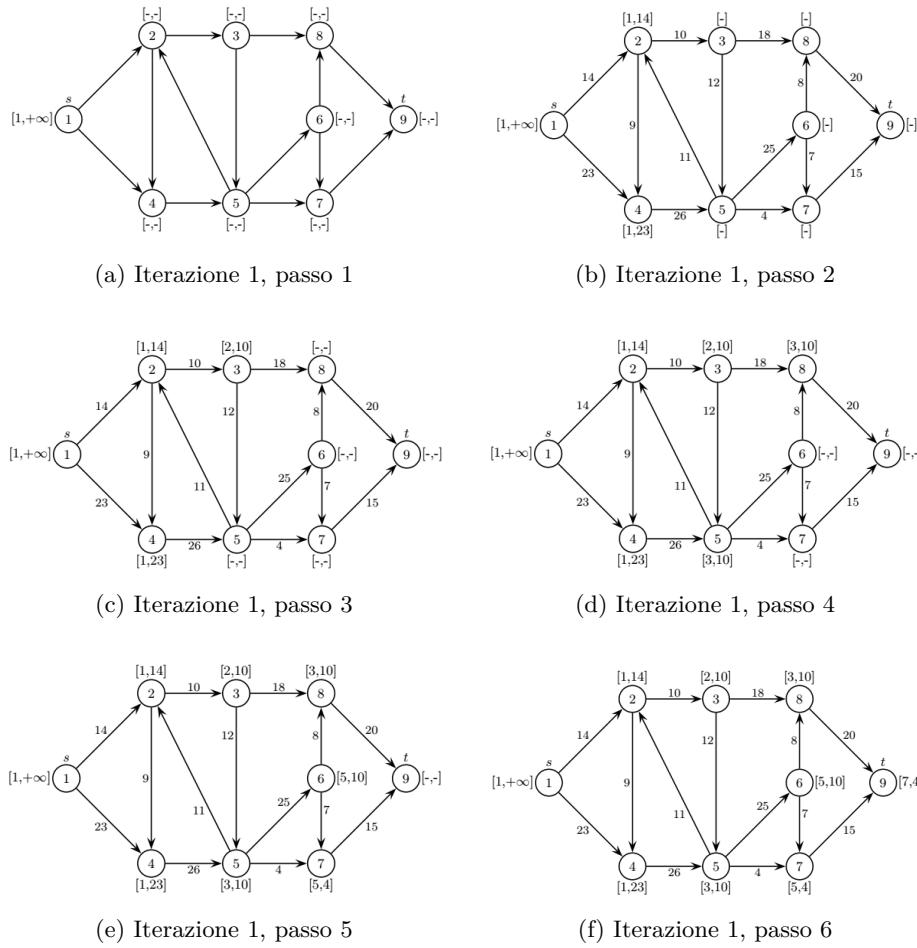


Figura 1.43: Un esempio di esecuzione della prima iterazione della seconda versione dell’Algoritmo di Ford-Fulkerson

3 considerando i nodi adiacenti 5 e 8 (Figura 1.43d). Poi, tra i nodi etichettati e non espansi (i.e., 4, 5 e 8), viene espanso il nodo 4, ma il nodo adiacente 5 è già stato etichettato. Quindi, si espande il nodo 5 che ha i nodi adiacenti 2 (già etichettato), 6 e 7 (Figura 1.43e). Infine, tra i nodi etichettati e non espansi (i.e., 6, 7 e 8), viene espanso il nodo 6, ma i nodi adiacenti 7 e 8 sono già etichettati. Quindi, si espande il nodo 7 che ha il nodo adiacenti  $t = 9$  (Figura 1.43f). Siccome il nodo  $t = 9$  è stato etichettato con  $\delta_t = 4$ , possiamo aumentare il flusso di 4 unità lungo il cammino aumentante. Il cammino aumentante  $P = (1, 2, 3, 5, 7, 9)$  è stato ricostruito utilizzando le etichette a partire dal nodo  $t = 9$ . Il flusso viene aggiornato:  $v = v + \delta_t = 0 + 4 = 4$  e il flusso degli archi (i.e., la soluzione  $x$ ) si modifica come segue:

- arco (7,9):  $x_{79} = x_{79} + \delta_t = 0 + 4 = 4$ ;
- arco (5,7):  $x_{57} = x_{57} + \delta_t = 0 + 4 = 4$ ;
- arco (3,5):  $x_{35} = x_{35} + \delta_t = 0 + 4 = 4$ ;
- arco (2,3):  $x_{23} = x_{23} + \delta_t = 0 + 4 = 4$ ;
- arco (1,2):  $x_{12} = x_{12} + \delta_t = 0 + 4 = 4$ .

Nella Figura 1.44 sono riportate tutte le iterazioni della seconda versione dell'Algoritmo di Ford-Fulkerson. Nella seconda iterazione (Figura 1.43b), al termine dell'etichettamento, si può aumentare il flusso di  $\delta_t = 6$  unità lungo il cammino aumentante  $P = (1, 2, 3, 5, 6, 7, 9)$  e il nuovo flusso è  $v = v + \delta_t = 4 + 6 = 10$ . Nella terza iterazione (Figura 1.43c), si può aumentare il flusso di  $\delta_t = 1$  unità lungo il cammino aumentante  $P = (1, 4, 5, 6, 7, 9)$  e il nuovo flusso è  $v = v + \delta_t = 10 + 1 = 11$ . Nella quarta iterazione (Figura 1.43d), il flusso è aumentato di  $\delta_t = 10$  unità lungo il cammino aumentante  $P = (1, 4, 5, 3, 8, 9)$  e il nuovo flusso è  $v = v + \delta_t = 11 + 10 = 21$ . Poi, nella quinta iterazione (Figura 1.43e), il flusso è aumentato di  $\delta_t = 8$  unità lungo il cammino aumentante  $P = (1, 4, 5, 6, 8, 9)$  e il nuovo flusso è  $v = v + \delta_t = 21 + 8 = 29$ . Infine, nell'ultima iterazione (Figura 1.43e), si scopre che non esiste un cammino aumentante che raggiunge il nodo  $t = 9$ , per cui la soluzione corrente è ottima e il flusso massimo è pari a 29. Il taglio di capacità minima è determinato dall'insieme  $S = \{1, 2, 4, 5, 6\}$  dei nodi etichettati e l'insieme  $\bar{S} = \{3, 7, 8, 9\}$  dei nodi non etichettati.

## 1.5 Flusso di costo minimo

Sia dato un grafo orientato  $G = (N, A)$ , in cui ad ogni arco  $(i, j) \in A$  è associato un costo  $c_{ij}$  e una capacità  $u_{ij}$ . Ad ogni vertice  $i \in N$  è associata una disponibilità  $b_i > 0$  oppure una richiesta  $b_i < 0$  oppure  $b_i = 0$ .

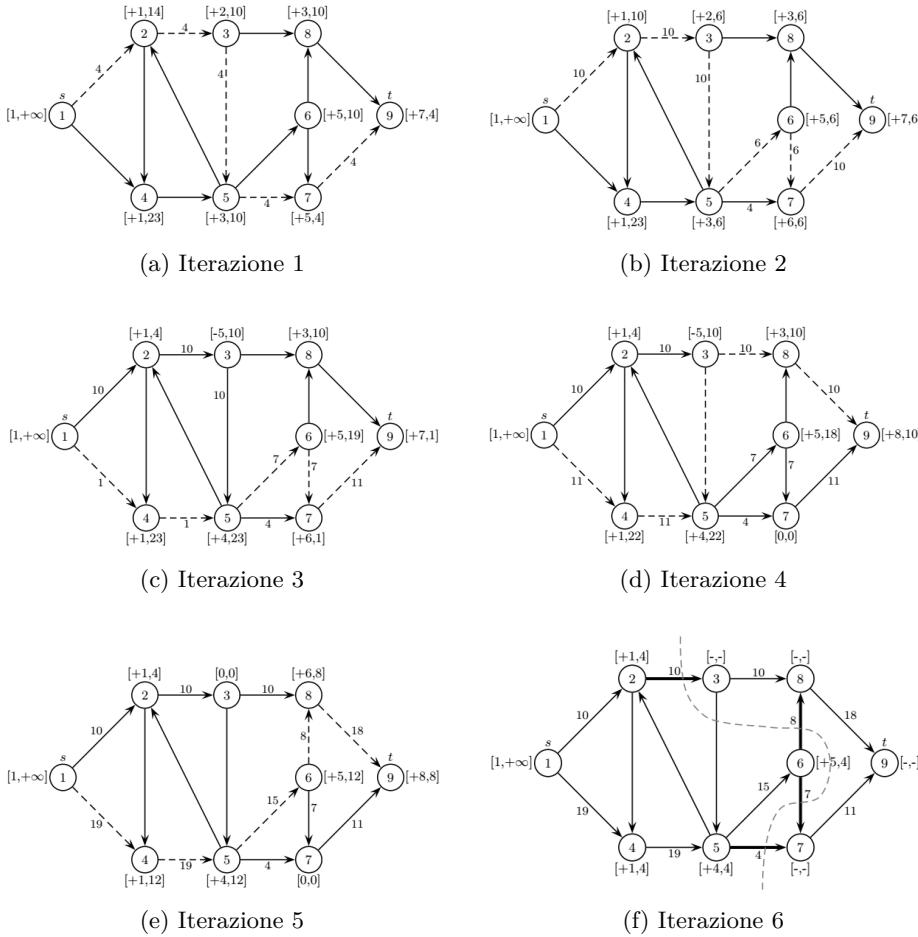


Figura 1.44: Un esempio di esecuzione di tutte le iterazioni della seconda versione dell’Algoritmo di Ford-Fulkerson

Il problema del flusso di costo minimo può essere formulato come segue:

$$(MCF) \quad z_{MCF} = \min \sum_{(i,j) \in A} c_{ij} x_{ij} \quad (1.16)$$

$$\text{s.t. } \sum_{j \in \Gamma_i} x_{ij} - \sum_{j \in \Gamma_i^{-1}} x_{ji} = b_i, \quad i \in N \quad (1.17)$$

$$0 \leq x_{ij} \leq u_{ij}, \quad (i, j) \in A \quad (1.18)$$

dove  $\Gamma_i = \{j : (i, j) \in A\}$  e  $\Gamma_i^{-1} = \{j : (j, i) \in A\}$ .

Al vincolo di conservazione del flusso corrispondente al nodo  $i \in N$  è associata una variabile duale  $\pi_i$  e al vincolo di capacità relativo all'arco  $(i, j) \in A$  è associata la variabile duale  $\alpha_{ij}$ .

Il duale della formulazione  $MCF$  del problema del flusso di costo minimo è il seguente:

$$(D) \quad z_D = \max \sum_{i \in N} b_i \pi_i - \sum_{(i,j) \in A} u_{ij} \alpha_{ij} \quad (1.19)$$

$$\text{s.t. } \pi_i - \pi_j - \alpha_{ij} \leq c_{ij}, \quad (i, j) \in A \quad (1.20)$$

$$\pi_i \text{ qualsiasi,} \quad i \in N \quad (1.21)$$

$$\alpha_{ij} \geq 0. \quad (i, j) \in A \quad (1.22)$$

### 1.5.1 Assunzioni e definizioni

In questa sezione si riassumono tutte le assunzioni che permettono di semplificare la presentazione e, in alcuni casi, garantire la convergenza alla soluzione ottima (e quindi ammissibile) del problema del flusso di costo minimo.

Si assume che il grafo  $G(N, A)$  sia orientato con  $n$  vertici ed  $m$  archi. Tutti i parametri del problema (i.e., costi, richieste, disponibilità e capacità) sono valori interi. Inoltre, tutti i costi  $c_{ij}$  e le capacità  $u_{ij}$  sono valori non negativi. Definiamo  $U = \max\{u_{ij} : (i, j) \in A\}$  e  $C = \max\{c_{ij} : (i, j) \in A\}$ .

Le richieste e le disponibilità dei diversi vertici soddisfano la condizione  $\sum_{i \in N} b_i = 0$  e che il problema di flusso di costo minimo ha una soluzione ammissibile. Si assume anche che il grafo contiene un cammino diretto di capacità infinita per ogni coppia di vertici. Con l'uso di eventuali variabili artificiali si può garantire l'esistenza di una soluzione "ammissibile" (che potrebbe non esserlo per il problema originario).

Per il problema del flusso di costo minimo, il grafo residuo  $G(x)$  corrispondente al flusso  $x$  è ottenuto sostituendo ogni arco  $(i, j) \in A$  con due archi  $(i, j)$  e  $(j, i)$  tali che:

- il loro costo è pari a  $c_{ij} = c_{ij}$  e  $c_{ji} = -c_{ij}$ .
- la loro capacità residua è pari a  $r_{ij} = u_{ij} - x_{ij}$  e  $r_{ji} = x_{ij}$ .

Si denota con  $\pi_i \in \mathbb{R}$  il *potenziale* associato ad ogni vertice  $i \in N$  e si definisce il *costo ridotto*  $c_{ij}^\pi$  come segue:

$$c_{ij}^\pi = c_{ij} - \pi_i + \pi_j$$

Si noti che il potenziale  $\pi_i$  corrisponde anche alla variabile duale associata al vincolo di conservazione del flusso relativo al vertice  $i \in N$ . I potenziali hanno le seguenti proprietà:

- Per ogni cammino diretto  $P$  dal vertice  $k$  al vertice  $l$  si ha:

$$\sum_{(i,j) \in P} c_{ij}^\pi = \sum_{(i,j) \in P} c_{ij} - \pi_k + \pi_l$$

- Per ogni ciclo diretto  $W$  si ha:

$$\sum_{(i,j) \in W} c_{ij}^\pi = \sum_{(i,j) \in W} c_{ij}$$

### 1.5.2 Condizioni di ottimalità

Se si considera una soluzione ammissibile  $x^*$  del problema di flusso di costo minimo, si può stabilire se è una soluzione ottima applicando una delle seguenti condizioni di ottimalità equivalenti:

- **Assenza Cicli di Costo Negativo**

La soluzione  $x^*$  è ottima se e solo se il grafo residuo  $G(x^*)$  non contiene cicli di costo negativo.

- **Costi Ridotti Positivi**

La soluzione  $x^*$  è ottima se e solo se è possibile trovare dei *potenziali*  $\pi$  tali che per ogni arco  $(i, j)$  di  $G(x^*)$  soddisfano la seguente condizione:

$$c_{ij}^\pi = c_{ij} - \pi_i + \pi_j \geq 0$$

- **Relazione degli Scarti Complementari**

La soluzione  $x^*$  è ottima se e solo se è possibile trovare dei *potenziali*  $\pi$  tali che per ogni arco  $(i, j) \in A$  il *costo ridotto*  $c_{ij}^\pi$  soddisfa le seguenti condizioni degli scarti complementari:

$$\begin{aligned} \text{If } c_{ij}^\pi > 0, & \text{ then } x_{ij}^* = 0 \\ \text{If } c_{ij}^\pi = 0, & \text{ then } 0 \leq x_{ij}^* \leq u_{ij} \\ \text{If } c_{ij}^\pi < 0, & \text{ then } x_{ij}^* = u_{ij} \end{aligned}$$

### 1.5.3 Cycle Cancelling Algorithm

L'algoritmo si basa sulla seguente osservazione: “se nel grafo  $G(x)$  si identifica un ciclo  $W$  di costo negativo, allora è possibile aumentare il flusso in  $W$  diminuendo il costo complessivo della soluzione”.

```

algorithm cycle cancelling
begin
    Stabilisci un flusso ammissibile  $x$  nel grafo  $G$ ;
    while  $G(x)$  contiene un ciclo di costo negativo do
        begin
            Applica un algoritmo per identificare un ciclo di costo negativo  $W$ ;
            Calcola  $\delta = \min\{r_{ij} : (i, j) \in W\}$ ;
            Aumenta di  $\delta$  unita' il flusso nel ciclo  $W$ ;
            Aggiorna  $G(x)$ ;
        end
    end

```

L'algoritmo mantiene ad ogni passo l'ammissibilità della soluzione e cerca di ottenere il soddisfacimento delle condizioni di ottimalità.

Un flusso ammissibile può essere calcolato risolvendo un problema di flusso massimo sul grafo  $G'$  costruito come segue:

- Si aggiunge a  $G$  due nuovi vertici  $s$  e  $t$ ;
- Per ogni vertice  $i \in N$  con  $b_i > 0$  si aggiunge un *arco sorgente*  $(s, i)$  con capacità  $u_{si} = b_i$ ;
- Per ogni vertice  $i \in N$  con  $b_i < 0$  si aggiunge un *arco destinazione*  $(i, t)$  con capacità  $u_{it} = -b_i$ .

Se il flusso massimo da  $s$  a  $t$  calcolato satura tutti gli archi sorgente e destinazione, allora il flusso trovato è ammissibile per il grafo  $G$ , altrimenti per  $G$  non può esistere un flusso ammissibile.

Per identificare un ciclo  $W$  di costo negativo in  $G(x)$  si può usare un algoritmo di tipo “label-correcting” per il calcolo del cammino di costo minimo come, per esempio, l’algoritmo di Bellman-Ford.

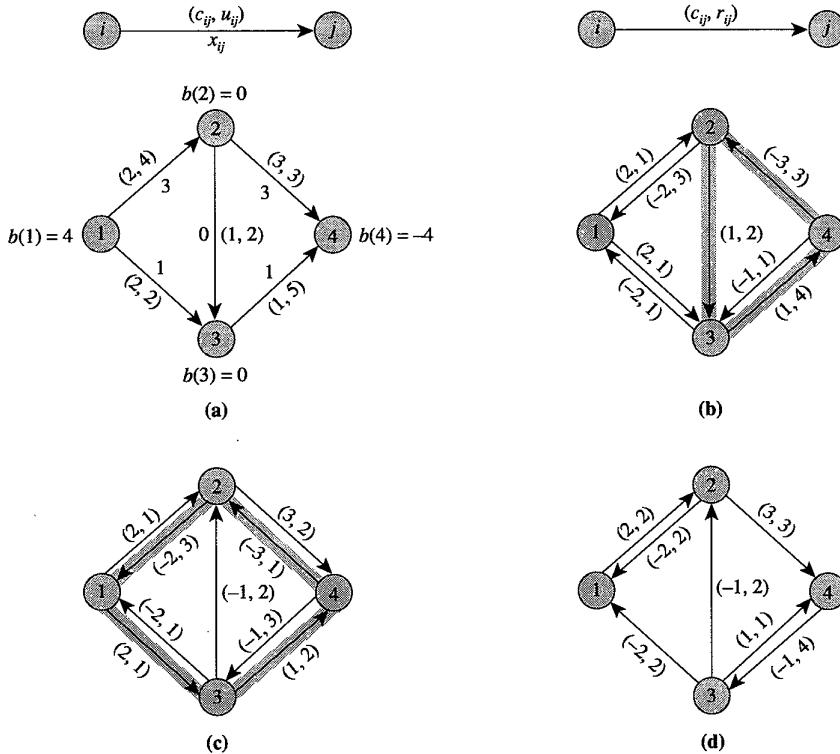
Nel caso peggiore il Cycle Cancelling Algorithm esegue  $O(mCU)$  iterazioni. Ad ogni iterazione l’algoritmo risolve il problema del cammino di costo minimo con costi qualsiasi per identificare un ciclo di costo negativo. Nel caso in cui sia utilizzato Bellman-Ford, la ricerca di un ciclo di costo negativo ha complessità pari a  $O(nm)$ . Quindi, nel caso sia impiegato Bellman-Ford, la complessità computazionale del Cycle Cancelling Algorithm è pari a  $O(nm^2CU)$ .

In Figura 1.45 è riportato un esempio di esecuzione del Cycle Cancelling Algorithm.

#### 1.5.4 Successive Shortest Path Algorithm

L’algoritmo ad ogni passo mantiene soddisfatte le condizioni di non negatività dei costi ridotti e i vincoli di capacità, mentre cerca di soddisfare i vincoli di conservazione del flusso:

$$\sum_{j \in \Gamma_i} x_{ij} - \sum_{j \in \Gamma_i^{-1}} x_{ji} = b_i$$



**Figure 9.8** Illustrating the cycle canceling algorithm: (a) network example with a feasible flow  $x$ ; (b) residual network  $G(x)$ ; (c) residual network after augmenting 2 units along the cycle 4–2–3–4; (d) residual network after augmenting 1 unit along the cycle 4–2–1–3–4.

Figura 1.45: Esempio di esecuzione del Cycle Cancelling Algorithm (Fonte: Orlin, Ahuja, Magnanti, “Network Flows: Theory, Algorithms, and Applications”, Pearson)

Il flusso  $x$  sarà ammissibile solo al termine dell’algoritmo e quindi nelle fasi intermedie si dirà che  $x$  è un *pseudoflusso*. Inoltre, per ogni vertice  $i$  definiamo:

$$e_i = b_i - \sum_{j \in \Gamma_i} x_{ij} + \sum_{j \in \Gamma_i^{-1}} x_{ji}$$

**Lemma 1.2.** Sia  $x$  un pseudoflusso che soddisfa le condizioni di non negatività dei costi ridotti per un qualche  $\pi$ , i.e.  $c_{ij}^\pi = c_{ij} - \pi_i + \pi_j \geq 0$ ,  $\forall (i, j)$  in  $G(x)$ . Sia  $d = (d_1, \dots, d_n)$  la distanza minima in  $G(x)$  da un vertice  $s$  a tutti gli altri vertici rispetto ai costi ridotti  $c_{ij}^\pi$ . Le seguenti proprietà sono valide:

- Il pseudoflusso  $x$  soddisfa le condizioni di non negatività dei costi ridotti anche per i potenziali  $\pi' = \pi - d$ ;

- I costi ridotti  $c_{ij}^{\pi'}$  sono nulli per tutti gli archi  $(i, j)$  nel cammino minimo da  $s$  a ogni altro nodo.  $\square$

**Lemma 1.3.** Sia  $x$  un pseudoflusso che soddisfa le condizioni di non negatività dei costi ridotti per un qualche potenziale  $\pi$ .

Se  $x'$  è il pseudoflusso ottenuto da  $x$  aumentando il flusso lungo il cammino di costo minimo da  $s$  a un altro vertice  $k$ , allora anche  $x'$  soddisfa le condizioni di non negatività dei costi ridotti per un qualche  $\pi'$ , e.g.  $\pi' = \pi - d$ .  $\square$

Il Lemma suggerisce la seguente strategia: “Ad ogni iterazione l’algoritmo deve selezionare un vertice  $s$  con un *eccesso* di flusso, i.e.,  $e_s > 0$ , e un vertice  $t$  con un *deficit* di flusso, i.e.,  $e_t < 0$ , e aumentare il flusso lungo il cammino di costo minimo da  $s$  a  $t$  nel grafo  $G(x)$ ”.

Si noti che nel calcolo del cammino di costo minimo si impiegano i costi ridotti  $c_{ij}^{\pi}$  che sono positivi. Pertanto, può essere utilizzato l’Algoritmo di Dijkstra.

```

algorithm successive shortest path
begin
   $x = 0$ ,  $\pi = 0$  e  $e_i = b_i$  per ogni  $i \in N$ ;
   $E = \{i \in N : e_i > 0\}$  e  $D = \{i \in N : e_i < 0\}$ ;
  while  $E \neq \emptyset$  do
    begin
      Seleziona un nodo  $k \in E$  e un nodo  $l \in D$ ;
      Calcola in  $G(x)$  le distanze minime  $d$  da  $k$ 
        a tutti gli altri nodi rispetto ai costi ridotti  $c_{ij}^{\pi}$ ;
      Sia  $P$  il cammino di costo minimo da  $k$  a  $l$ ;
      Aggiorna  $\pi = \pi - d$ ;
      Calcola  $\delta = \min\{e_k, -e_l, \min\{r_{ij} : (i, j) \in P\}\}$ ;
      Aumenta di  $\delta$  unita’ il flusso nel cammino  $P$ ;
      Aggiorna  $G(x)$ ;
    end
  end

```

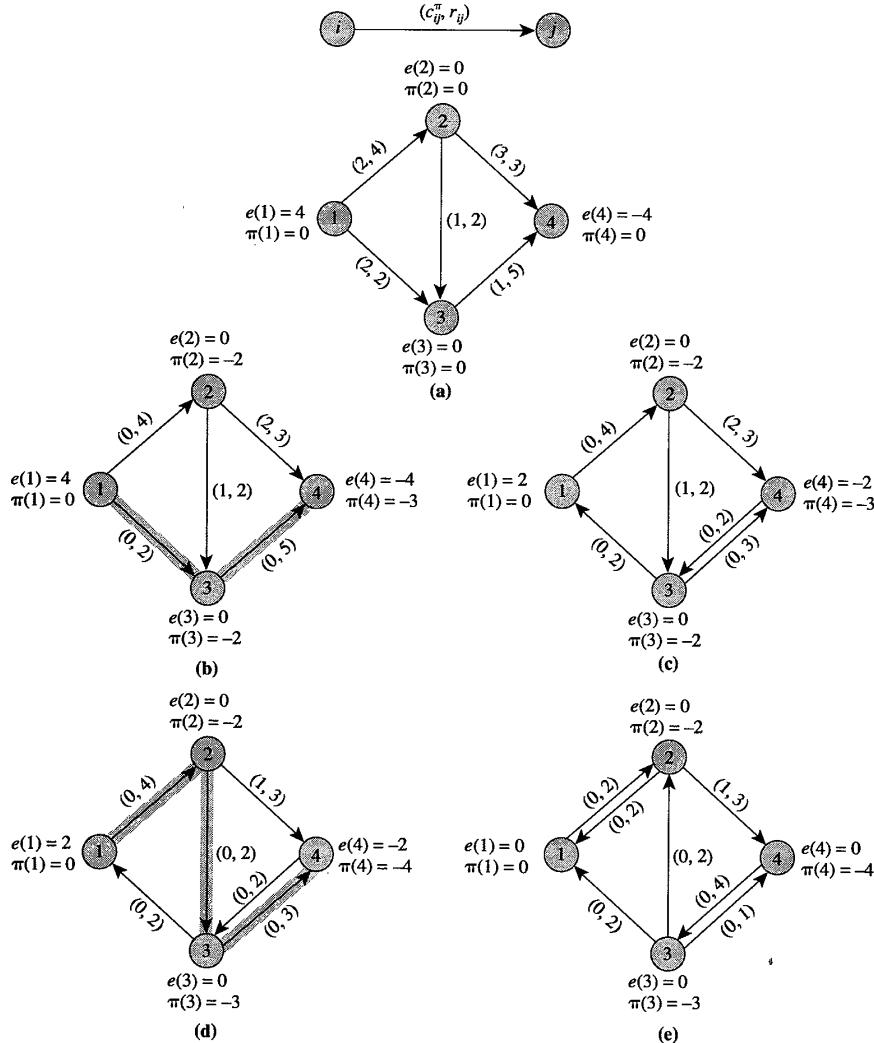
Nel caso peggiore il Successive Shortest Path Algorithm esegue  $O(nU)$  iterazioni. Ad ogni iterazione l’algoritmo risolve il problema del cammino di costo minimo con costi positivi o uguali a zero (i.e., non-negativi).

Nel caso sia utilizzato l’Algoritmo di Dijkstra con Fibonacci Heap la complessità del calcolo dei cammini di costo minimo è pari a  $O(m + n \log n)$ ; quindi la complessità computazionale complessiva del Successive Shortest Path Algorithm è pari a  $O(nU(m + n \log n))$ .

In Figura 1.46 è riportato un esempio di esecuzione del Successive Shortest Path Algorithm.

### 1.5.5 Primal-Dual Algorithm

L’algoritmo primale-duale per la soluzione del problema del flusso di costo minimo è simile al Successive Shortest Path Algorithm, in quanto anch’esso mantiene



**Figure 9.10** Illustrating the successive shortest path algorithm: (a) initial residual network for  $x = 0$  and  $\pi = 0$ ; (b) network after updating the potentials  $\pi$ ; (c) network after augmenting 2 units along the path 1–3–4; (d) network after updating the potentials  $\pi$ ; (e) network after augmenting 2 units along the path 1–2–3–4.

Figura 1.46: Esempio di esecuzione del Successive Shortest Path Algorithm (Fonte: Orlin, Ahuja, Magnanti, “Network Flows: Theory, Algorithms, and Applications”, Pearson)

un pseudoflusso che soddisfa le condizioni di non negatività dei costi ridotti e i vincoli di capacità, mentre i vincoli di conservazione del flusso sono soddisfatti solo al raggiungimento della soluzione ottima.

L'algoritmo primale-duale prevede di trasformare il problema originale in modo tale da avere un solo vertice con un eccesso di flusso e un solo vertice con un deficit di flusso. Quindi il grafo viene modificato come segue:

- Si aggiungono a  $G$  due nuovi vertici  $s$  e  $t$ ;
- Per ogni vertice  $i \in N$  con  $b_i > 0$  si aggiunge un *arco sorgente*  $(s, i)$  con capacità  $u_{si} = b_i$ ;
- Per ogni vertice  $i \in N$  con  $b_i < 0$  si aggiunge un *arco destinazione*  $(i, t)$  con capacità  $u_{it} = -b_i$ ;
- Si pone  $b_s = \sum_{i \in N^+} b_i$ , dove  $N^+ = \{i \in N : b_i > 0\}$ ,  $b_t = -b_s$  e  $b_i = 0$  per ogni  $i \in N$ .

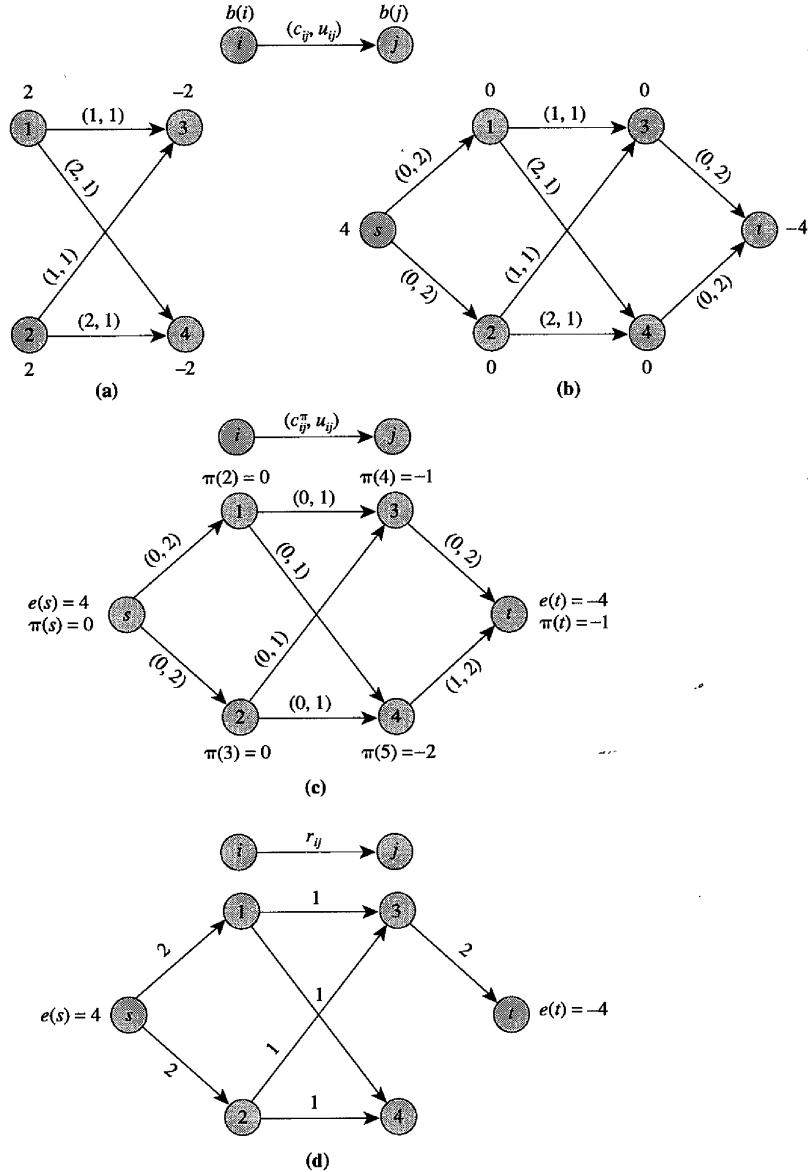
Dati i potenziali  $\pi$ , il *grafo ammissibile*  $G^a(x)$  è un sottografo di  $G(x)$  che contiene solo gli archi  $(i, j)$  di  $G(x)$  di costo ridotto nullo, i.e.  $c_{ij}^\pi = 0$ . La capacità residua  $r_{ij}$  è la stessa del corrispondente arco in  $G(x)$ .

```
algorithm primal-dual
begin
   $x = 0$  e  $\pi = 0$ ;
   $e_s = b_s$  e  $e_t = b_t$ ;
  while  $e_s > 0$  do
    begin
      Calcola in  $G(x)$  le distanze minime  $d$  da  $s$ 
      a tutti gli altri nodi rispetto ai costi ridotti  $c_{ij}^\pi$ ;
      Aggiorna  $\pi = \pi - d$ ;
      Definisci il grafo ammissibile  $G^a(x)$ ;
      Calcola in  $G^a(x)$  il flusso massimo dal  $s$  a  $t$ ;
      Aggiorna  $e_s$ ,  $e_t$  e  $G(x)$ ;
    end
  end
```

Nel caso peggiore l'algoritmo primale duale esegue un numero di iterazioni pari a  $O(\min\{nU, nC\})$ . Ad ogni iterazione l'algoritmo risolve un problema del cammino di costo minimo con costi non negativi e un problema del flusso massimo.

Nel caso sia utilizzato l'Algoritmo di Dijkstra con Fibonacci Heap la complessità del calcolo dei cammini di costo minimo è pari a  $O(m + n \log n)$ . Nel caso sia utilizzato l'Algoritmo Highest-Label Preflow-Push la complessità del calcolo del flusso massimo è pari a  $O(n^2 \sqrt{m})$ . Quindi, in questo caso la complessità computazionale dell'algoritmo primale duale è pari a  $O((\min\{nU, nC\})((m + n \log n) + (n^2 \sqrt{m})))$ .

In Figura 1.47 è riportato un esempio di esecuzione del Successive Primal-Dual Algorithm.



**Figure 9.12** Illustrating the primal-dual algorithm: (a) example network; (b) transformed network; (c) residual network after updating the node potentials; (d) admissible network.

Figura 1.47: Esempio di esecuzione del Primal-Dual Algorithm (Fonte: Orlin, Ahuja, Magnanti, “Network Flows: Theory, Algorithms, and Applications”, Pearson)

### 1.5.6 Out-of-Kilter Algorithm

L'algoritmo Out-of-Kilter mantiene soddisfatti i vincoli di conservazione del flusso ad ogni vertice  $i \in N$ , mentre consente la violazione sia delle condizioni di ottimalità che dei vincoli di capacità degli archi  $(i, j) \in A$ .

L'algoritmo iterativamente modifica il flusso  $x$  e i potenziali  $\pi$ , in modo da *diminuire* la non ammissibilità della soluzione e convergere alla soluzione ottima.

Si ricordi che la soluzione  $x^*$  è ottima se e solo se è possibile trovare dei potenziali  $\pi$  tali che per ogni arco  $(i, j) \in A$  il corrispondente costo ridotto  $c_{ij}^\pi$  soddisfa le seguenti condizioni degli scarti complementari:

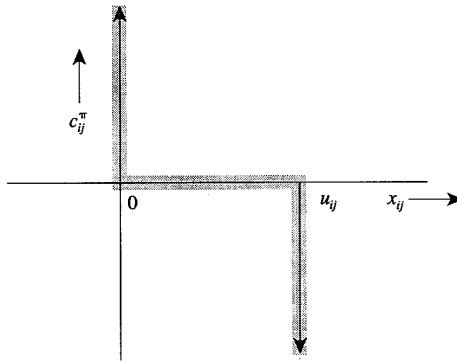
$$\begin{aligned} \text{Se } c_{ij}^\pi > 0, & \text{ allora } x_{ij}^* = 0 \\ \text{Se } c_{ij}^\pi = 0, & \text{ allora } 0 \leq x_{ij}^* \leq u_{ij} \\ \text{Se } c_{ij}^\pi < 0, & \text{ allora } x_{ij}^* = u_{ij} \end{aligned}$$

Se un arco  $(i, j) \in A$  soddisfa le condizioni di ottimalità si dirà che è *in-kilter*, mentre se non le soddisfa si dirà che è *out-of-kilter*.

Il *Kilter Number*  $k_{ij} \geq 0$  indica di quanto deve essere modificato il flusso  $x_{ij}$  per rendere *in-kilter* l'arco  $(i, j) \in A$  rispetto al costo ridotto  $c_{ij}^\pi$ :

$$\begin{aligned} \text{Se } c_{ij}^\pi > 0, & \text{ allora } k_{ij} = |x_{ij}| \\ \text{Se } c_{ij}^\pi = 0 \text{ e } x_{ij} > u_{ij}, & \text{ allora } k_{ij} = x_{ij} - u_{ij} \\ \text{Se } c_{ij}^\pi = 0 \text{ e } x_{ij} < 0, & \text{ allora } k_{ij} = -x_{ij} \\ \text{Se } c_{ij}^\pi < 0, & \text{ allora } k_{ij} = |u_{ij} - x_{ij}| \end{aligned}$$

Il kilter number  $k_{ij}$  indica il livello di non ammissibilità dell'arco  $(i, j) \in A$ . L'algoritmo Out-Of-Kilter ad ogni iterazione individua un arco  $(i, j) \in A$  out-of-kilter e riduce di una quantità positiva il corrispondente kilter number  $k_{ij}$  senza dover aumentare il kilter number degli altri nodi.



**Figure 9.13** Kilter diagram for arc  $(i, j)$ .

Figura 1.48: Kilter number (Fonte: Orlin, Ahuja, Magnanti, "Network Flows: Theory, Algorithms, and Applications", Pearson)

Se si assume di avere un flusso iniziale ammissibile, allora la definizione del kilter number per ogni arco  $(i, j) \in A$  si riduce a:

$$k_{ij} = \begin{cases} 0, & \text{se } c_{ij}^\pi \geq 0 \\ r_{ij}, & \text{se } c_{ij}^\pi < 0 \end{cases}$$

L'Algoritmo Out-Of-Kilter può essere definito come segue:

```

algorithm out-of-kilter
begin
    Sia  $\pi = 0$  e  $x$  un flusso ammissibile;
    Calcola  $G(x)$  e il kilter number  $k_{ij}$  per ogni arco  $(i, j)$ ;
    while  $G(x)$  contiene un arco  $(p, q)$  out-of-kilter do
        begin
            Definisci il costo di ogni arco  $(i, j)$  in  $G(x)$  come  $c'_{ij} = \max\{0, c_{ij}^\pi\}$ ;
            Calcola in  $G(x) - \{(q, p)\}$  le distanze minime  $d$ 
            da  $q$  a tutti gli altri nodi rispetto ai costi  $c'_{ij}$ ;
            Sia  $P$  il cammino di costo minimo da  $q$  a  $p$ ;
            Aggiorna  $\pi' = \pi - d$ ;
            if  $c_{pq}^{\pi'} < 0$  then
                 $W = P \cup \{(p, q)\}$ ;
                Calcola  $\delta = \min\{r_{ij} : (i, j) \in W\}$ ;
                Aumenta di  $\delta$  unita' il flusso nel ciclo  $W$ ;
                Aggiorna  $x$ ,  $G(x)$ , i costi ridotti  $c_{ij}^\pi$  e i kilter number  $k_{ij}$ ;
            end if
        end
    end
```

Nel caso peggiore l'algoritmo Out-Of-Kilter esegue un numero di iterazioni pari a  $O(mU)$ . Ad ogni iterazione l'algoritmo risolve il problema del cammino di costo minimo con costi non negativi.

Nel caso sia utilizzato l'Algoritmo di Dijkstra con Fibonacci Heap la complessità del calcolo dei cammini di costo minimo è pari a  $O(m + n \log n)$ ; quindi, la complessità computazionale dell'algoritmo Out-Of-Kilter è pari a  $O(mU(m + n \log n))$ .

### 1.5.7 Algoritmi polinomiali

Per ottenere degli algoritmi polinomiali per risolvere il problema del flusso di costo minimo possono essere utilizzate delle tecniche di *scaling*.

#### Capacity Scaling Algorithm

Il Capacity Scaling Algorithm è una variante del Successive Shortest Path Algorithm in cui ad ogni iterazione viene garantito che la capacità residua del cammino aumentante sia “sufficientemente grande”.

Viene definito un parametro  $\Delta$  e ad ogni iterazione:

- si selezionano i vertici  $s$  e  $t$  tali che  $e_s \geq \Delta$  e  $e_t \leq -\Delta$ ;
- si sostituisce il grafo residuo  $G(x)$  con il suo sottografo  $G(x, \Delta)$  contenente solo quegli archi di capacità residua pari almeno a  $\Delta$ .

Quando non è più possibile determinare una coppia di vertici  $s$  e  $t$  tali che  $e_s \geq \Delta$  e  $e_t \leq -\Delta$ , allora si diminuisce il parametro  $\Delta$ , i.e.,  $\Delta = \Delta/2$ . Se  $\Delta = 1$  allora il flusso  $x$  corrente è ottimo.

Se si inizializza  $\Delta = 2^{\lfloor \log U \rfloor}$  allora il Capacity Scaling Algorithm nel caso peggiore eseguirà  $O(m \log U)$  iterazioni. Quindi, nel caso sia impiegato l'Algoritmo di Dijkstra con Fibonacci Heap, la complessità computazionale complessiva del Capacity Scaling Algorithm è pari a  $O((m \log U)(m + n \log n))$ .

### 1.5.8 Algoritmi Fortemente Polinomiali

Algoritmi fortemente polinomiali per risolvere il problema del flusso di costo minimo possono essere ottenuti modificando il Cycle Cancelling Algorithm e il Capacity Scaling Algorithm.

#### Minimum Mean Cycle Cancelling Algorithm

Il Minimum Mean Cycle Cancelling Algorithm è un caso particolare del Cycle Cancelling Algorithm in cui a ogni iterazione il flusso è aumentato nel ciclo  $W$  di costo negativo del grafo  $G(x)$  che ha *costo medio* minimo. Il costo medio di un ciclo  $W$  è dato da  $(\sum_{(i,j) \in W} c_{ij})/|W|$ .

Per calcolare il ciclo  $W$  di costo medio minimo può essere impiegata una procedura di programmazione dinamica di complessità  $O(nm)$ .

La complessità computazionale del Minimum Mean Cycle Cancelling Algorithm è pari a  $O(n^2 m^3 \log n)$ .

#### Enhanced Capacity Scaling Algorithm

L'Enhanced Capacity Scaling Algorithm è una variante del Capacity Scaling Algorithm ed è basato sul seguente lemma.

**Lemma 1.4.** *Quando il Capacity Scaling Algorithm aggiorna il parametro  $\Delta$ , i.e.,  $\Delta = \Delta/2$ , se nell'arco  $(i, j)$  il flusso è tale che  $x_{ij} \geq 8n\Delta$  allora:*

- *in qualsiasi soluzione ottima  $x_{ij} > 0$ .*
- *il costo ridotto  $c_{ij}^\pi$  è sempre nullo comunque siano scelti i potenziali  $\pi$  ottimi.*  $\square$

Quando l'algoritmo identifica un arco  $(i, j)$  tale che  $x_{ij} \geq 8n\Delta$ , allora i potenziali dei vertici  $i$  e  $j$  possono essere fissati. Quindi, l'arco  $(i, j)$  viene “contratto” e i vertici  $i$  e  $j$  sono sostituiti da un nuovo vertice. L'operazione di contrazione da origine a un nuovo problema di flusso di costo minimo con un vertice in meno. L'algoritmo non effettua le operazioni di contrazione

esplicitamente ed è in grado di procedere senza dover ricalcolare i flussi e i potenziali per il nuovo problema ridotto.

La complessità computazionale dell'Enhanced Capacity Scaling Algorithm è pari a  $O((m \log n)(m + n \log n))$ .

## 1.6 Network Simplex Method

Il problema del flusso di costo minimo essendo un problema di programmazione lineare può essere risolto con il metodo del simplex.

Sfruttando delle caratteristiche strutturali del problema del flusso di costo minimo può essere implementato un metodo del simplex “*ad hoc*” più efficiente del metodo standard.

La formulazione del problema del flusso di costo minimo può essere scritta in forma matriciale come segue:

$$\begin{aligned} \text{Min } & cx \\ \text{s.t. } & Ax = b \\ & 0 \leq x \leq u \end{aligned}$$

Si noti che ogni colonna di  $A$  corrisponde a una variabile  $x_{ij}$  e ogni sua riga corrisponde a un vincolo di conservazione del flusso. Il rango della matrice  $A$  è pari a  $n - 1$ , quindi un vincolo è ridondante e  $m - n + 1$  variabili sono libere. Le colonne della matrice  $A$  possono essere partizionate in tre insiemi tali che  $A = (B, L, U)$ , dove:

- $B$ : variabili base, i.e.  $0 \leq x_{ij} \leq u_{ij}$ ;
- $L$ : variabili non base tali che  $x_{ij} = 0$ ;
- $U$ : variabili non base tali che  $x_{ij} = u_{ij}$ ;

Quindi, si possono partizionare allo stesso modo i vettori della soluzione  $x = (x_B, x_L, x_U)$  e delle capacità  $u = (u_B, u_L, u_U)$ .

La matrice  $B$  che rappresenta la base contiene  $n - 1$  colonne linearmente indipendenti.

Due delle operazioni critiche, che ad ogni iterazione il metodo del simplex deve svolgere, sono la determinazione della soluzione base ammissibile  $x_B$  e delle variabili duali  $\pi$ , risolvendo i seguenti sistemi lineari:

$$Bx_B = b - Uu_U$$

e

$$\pi B = c_B$$

Si mostrerà che la soluzione di questi sistemi lineari nel caso del problema del flusso di costo minimo è molto semplice.

**Teorema 1.12.** *Le righe e le colonne della matrice  $B$  possono essere sempre riordinate in modo da ottenere una matrice triangolare inferiore.  $\square$*

Il Teorema 1.12 implica che i sistemi lineari in cui la matrice dei coefficienti è  $B$  possono essere risolti semplicemente con delle sostituzioni forward o backward. Inoltre, si può dimostrare che la matrice  $A$  è “*totalmente unimodulare*”, quindi, se il problema di flusso di costo minimo ha capacità, domande e richieste intere, allora il flusso ottimo  $x$  sarà intero.

### 1.6.1 Struttura Spanning Tree

Il Network Simplex Method si basa sul seguente Teorema 1.13 che stabilisce la relazione tra *spanning tree* (albero di copertura) del grafo  $G$  e base  $B$ .

**Teorema 1.13.** *Ogni spanning tree (albero di copertura) definito sul grafo  $G$  definisce una base  $B$  del corrispondente problema di flusso di costo minimo. Viceversa, ogni base  $B$  relativa a un problema di flusso di costo minimo definisce uno spanning tree sul grafo  $G$ .  $\square$*

Data una soluzione base ammissibile  $x$ , si definisce una “*Struttura Spanning Tree ammissibile*” ( $T, L, U$ ) come segue:

- $T$ : archi nello spanning tree;
- $L$ : archi il cui flusso deve essere nullo;
- $U$ : archi in cui il flusso deve essere pari alla capacità.

Un arco  $(i, j)$  è “*libero*” se  $0 < x_{ij} < u_{ij}$ , mentre si dirà che è “*vincolato*” se  $x_{ij} = 0$  oppure  $x_{ij} = u_{ij}$ .

Se tutti gli archi nello spanning tree sono liberi, allora diremo che lo spanning tree è *non degenero*, altrimenti diremo che è *degenero*.

**Teorema 1.14.** *Una struttura spanning tree ( $T, L, U$ ) è una struttura spanning tree ottima del problema di flusso di costo minimo se è ammissibile e per un qualche  $\pi$  i costi ridotti  $c_{ij}^\pi$  soddisfano le seguenti condizioni:*

- Se  $c_{ij}^\pi = 0$ , allora  $(i, j) \in T$ ;
- Se  $c_{ij}^\pi \geq 0$ , allora  $(i, j) \in L$ ;
- Se  $c_{ij}^\pi \leq 0$ , allora  $(i, j) \in U$ .  $\square$

Il metodo del simplex verifica a ogni iterazione se esiste un arco  $(i, j)$  che viola le condizioni di ottimalità. Se questo arco esiste, allora inserisce  $(i, j)$  nello spanning tree  $T$ . Quando l'arco  $(i, j)$  viene inserito in  $T$  genera un ciclo  $W$  e l'insieme degli archi  $T$  non è più uno spanning tree. Quindi, per determinare la nuova base il metodo del simplex seleziona un arco dal ciclo  $W$  e lo toglie da  $T$ .

Per il momento si suppone che la struttura spanning tree non sia degenero, poi si vedrà come garantirsi la convergenza anche in caso di degenerazione.

L'algoritmo del simplex su reti può, quindi, essere riassunto come segue.

```

algorithm network-simplex
begin
    Determina una struttura  $(T, L, U)$  ammissibile;
    Calcola il flusso  $x$  corrispondente a  $(T, L, U)$ ;
    Calcola i potenziali  $\pi$  corrispondenti a  $(T, L, U)$ ;
    while esiste un arco non ammissibile do
        begin
            Seleziona l'arco  $(k, l)$  non ammissibile;
            Aggiungi l'arco  $(k, l)$  a  $T$  e determina
                l'arco  $(p, q)$  uscente;
            Aggiorna la struttura  $(T, L, U)$ ;
            Aggiorna i potenziali  $\pi$  e il flusso  $x$ ;
        end
    end

```

### 1.6.2 Implementazione della Struttura Spennning Tree

Per implementare la struttura spanning tree si definisce in modo arbitrario un *nodo radice* e ad ogni nodo  $i$  possiamo associare le seguenti tre informazioni:

- $pred(i)$ : indica il nodo padre, ossia il nodo che succede a  $i$  nel cammino da  $i$  al nodo radice.
- $depth(i)$ : indica il numero di nodi presenti nel cammino da  $i$  al nodo radice.
- $thread(i)$ : indica il nodo che in una ricerca di tipo depth-first segue il nodo  $i$ .

L'utilizzo dei vettori  $pred(i)$ ,  $depth(i)$  e  $thread(i)$  consente di effettuare in modo efficiente le operazioni richieste dal simplesso. Le operazioni richieste dal simplesso sono le seguenti:

- Calcolo di un flusso  $x$ ;
- Calcolo dei potenziali  $\pi$ ;
- Inserimento di un nuovo arco in  $T$ , identificazione di un ciclo  $W$  e determinazione dell'arco uscente;
- Aggiornamento del flusso  $x$ ;
- Aggiornamento dei potenziali  $\pi$ .

In Figura 1.49 è riportato un esempio delle strutture dati utilizzate per rappresentare uno spanning tree.

Ora si vuole mostrare come possono essere eseguite le operazioni di base utilizzando la struttura spanning tree.

L'aggiornamento dei potenziali  $\pi$  può essere svolta da una procedura come la seguente.

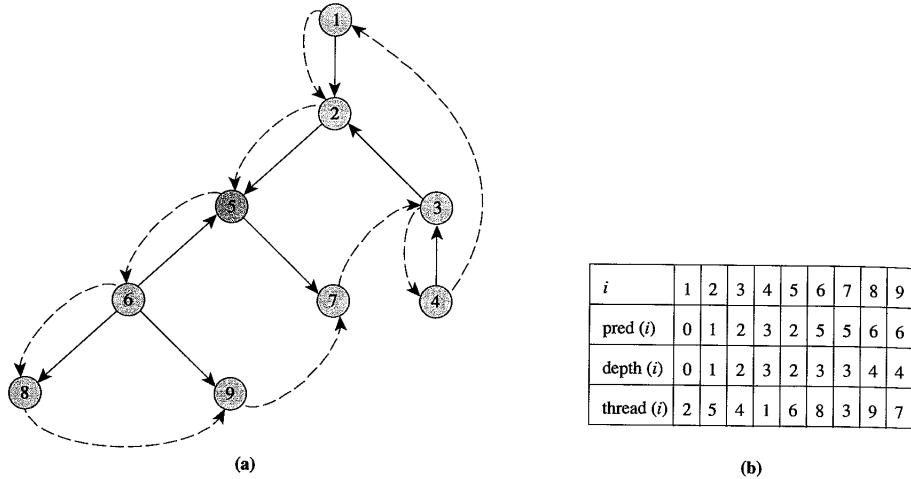


Figure 11.3 Example of a tree indices: (a) rooted tree; (b) corresponding tree indices.

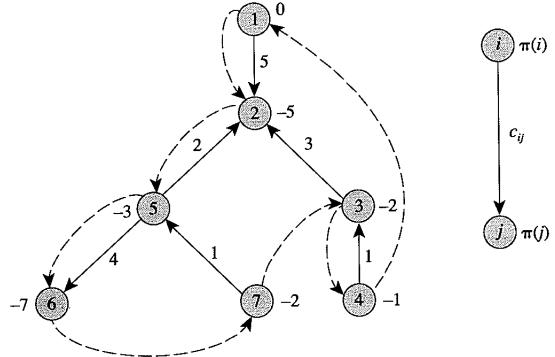
Figura 1.49: Strutture dati per rappresentare uno spanning tree (Fonte: Orlin, Ahuja, Magnanti, “Network Flows: Theory, Algorithms, and Applications”, Pearson)

```

procedure compute-potentials
begin
     $\pi_1 = 0;$ 
     $j = \text{thread}(1);$ 
    while  $j \neq 1$  do
        begin
             $i = \text{pred}(j);$ 
            if  $(i, j) \in A$  then  $\pi_j = \pi_i - c_{ij};$ 
            if  $(j, i) \in A$  then  $\pi_j = \pi_i + c_{ji};$ 
             $j = \text{thread}(j);$ 
        end
    end

```

In Figura 1.50 è riportato un esempio del calcolo dei potenziali applicando la procedura *compute-potentials*. L’albero di copertura viene visitato partendo dalla radice e valutando i nodi dell’albero seguendo l’ordine depth-first definito dal vettore  $\text{thread}(i)$ . Per ogni arco, i potenziali vengono calcolati in modo tale che  $c_{ij} - \pi_i + \pi_j = 0$ . Nell’esempio in Figura 1.50, il potenziale del nodo radice viene fissato al valore nullo (i.e.,  $\pi_1 = 0$ ), dopodiché, si considera l’arco  $(1, 2)$  e si calcola  $\pi_2 = \pi_1 - c_{12} = 0 - 5 = -5$ . Poi, si considera l’arco  $(5, 2)$  e si calcola  $\pi_5 = \pi_2 + c_{52} = -5 + 2 = -3$ . L’aggiornamento procede per i rimanenti archi dell’albero di copertura riuscendo a definire tutti i potenziali.



**Figure 11.5** Computing node potentials for a spanning tree.

Figura 1.50: Esempio di calcolo dei potenziali (Fonte: Orlin, Ahuja, Magnanti, “Network Flows: Theory, Algorithms, and Applications”, Pearson)

I calcolo del flusso  $x$  corrispondente alla base corrente può essere svolto da una procedura come la seguente.

```

procedure compute-flows
begin
  for each  $i \in N$  do  $b'_i = b_i$ ;
  for each  $(i, j) \in L$  do  $x_{ij} = 0$ ;
  for each  $(i, j) \in U$  do
    begin
       $x_{ij} = u_{ij}$ ;
       $b'_i = b'_i - u_{ij}$ ;
       $b'_j = b'_j + u_{ij}$ ;
    end
     $T' = T$ ;
    while  $T' \neq \emptyset$  do
      begin
        Selezione una foglia  $j$  nel sottoalbero  $T'$ ;
         $i = pred(j)$ ;
        if  $(i, j) \in T'$  then
           $x_{ij} = -b'_j$ ;
        else
           $x_{ji} = b'_j$ ;
           $b'_i = b'_i + b'_j$ ;
          Eliminare l'arco selezionato da  $T'$ ;
      end
    end
end

```

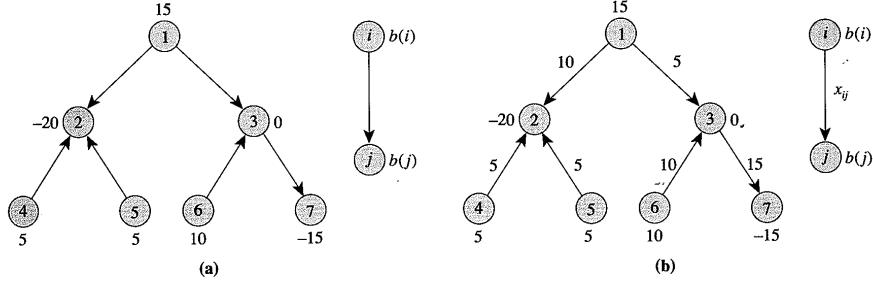


Figure 11.6 Computing flows for a spanning tree.

Figura 1.51: Esempio di calcolo del flusso (Fonte: Orlin, Ahuja, Magnanti, "Network Flows: Theory, Algorithms, and Applications", Pearson)

In Figura 1.51 è riportato un esempio del calcolo dei flussi applicando la procedura *compute-flows*. In questo caso, dopo aver aggiornato i  $b'_i$  tenendo conto dei flussi degli archi non in base, l'albero di copertura viene visitato partendo dalle foglie e valutando i nodi dell'albero seguendo un ordine bottom-up fino al nodo radice. Nell'esempio in Figura 1.51 l'aggiornamento parte dal nodo foglia  $j = 4$ , per cui  $x_{42} = b'_4 = 5$ , si aggiorna  $b'_2 = b'_2 + b'_4 = -20 + 5 = -15$  e si elimina da  $T'$  l'arco  $(4, 2)$ , così la foglia 4 non verrà più considerata. Poi si considera il nodo foglia  $j = 5$ , per cui  $x_{52} = b'_5 = 5$ , si aggiorna  $b'_2 = b'_2 + b'_5 = -15 + 5 = -10$  e si elimina da  $T'$  l'arco  $(5, 2)$ , così la foglia 5 non verrà più considerata e il nodo 2 è diventato una foglia. Il processo di aggiornamento continua fino a quando il flusso di tutti gli archi dell'albero di copertura sono stati calcolati.

Una volta selezionato l'arco  $(i, j)$  non in base che viola le condizioni di ottimalità (i.e.,  $c_{ij}^\pi > 0$ , se  $(i, j) \in U$ , oppure  $c_{ij}^\pi < 0$ , se  $(i, j) \in L$ ) si deve identificare il ciclo. L'aggiunta dell'arco  $(i, j)$  all'albero corrispondente alla base corrente genererà un ciclo, che deve essere identificato.

Una procedura che può identificare rapidamente il ciclo  $W$  generato da un arco  $(k, l)$  usando le strutture dati  $\text{pred}(i)$  e  $\text{depth}(i)$  è la seguente:

```

procedure identify-cycle( $k, l, W$ )
begin
     $i = k, j = l, W = \{(k, l)\};$ 
    while  $i \neq j$  do
        begin
            if  $\text{depth}(i) > \text{depth}(j)$  then
                Aggiungi in  $W$  l'arco di estremi  $i$  e  $\text{pred}(i)$ ;
                 $i = \text{pred}(i);$ 
            else if  $\text{depth}(j) > \text{depth}(i)$  then
                Aggiungi in  $W$  l'arco di estremi  $j$  e  $\text{pred}(j)$ ;
                 $j = \text{pred}(j);$ 
            else

```

```

    Aggiungi in  $W$  l'arco di estremi  $i$  e  $\text{pred}(i)$ ;
     $i = \text{pred}(i)$ ;
    Aggiungi in  $W$  l'arco di estremi  $j$  e  $\text{pred}(j)$ ;
     $j = \text{pred}(j)$ ;
end
end

```

Nella procedura *identify-cycle* potrà anche essere identificato l'arco  $(p, q)$  uscente e calcolato il quantitativo  $\delta$  con cui aumentare il flusso nel ciclo  $W$ .

Il ciclo  $W$  è detto “*pivot cycle*” e, dato l'arco entrante  $(k, l)$ , è orientato nella direzione:

- dell'arco  $(k, l)$  se  $(k, l) \in L$ ;
- opposta a quella dell'arco  $(k, l)$  se  $(k, l) \in U$ .

Si denota con  $\overline{W} \subseteq W$  l'insieme degli archi *forward* (quelli con lo stesso orientamento del ciclo  $W$ ) e con  $\underline{W} \subseteq W$  l'insieme degli archi *backward* (quelli con orientamento opposto rispetto a quello del ciclo  $W$ ).

Il quantitativo  $\delta$  con cui aumentare il flusso nel ciclo  $W$  è dato da:

$$\delta = \min\{\delta_{ij} : (i, j) \in W\}$$

dove

$$\delta_{ij} = \begin{cases} u_{ij} - x_{ij}, & \text{if } (i, j) \in \overline{W} \\ x_{ij}, & \text{if } (i, j) \in \underline{W} \end{cases}$$

Gli archi  $(i, j) \in W$  tali che  $\delta_{ij} = \delta$  sono detti “*blocking arc*” e l'arco uscente  $(p, q) \in W$  è scelto tra i blocking arc.

In Figura 1.52 è mostrato come viene identificato il ciclo e come viene aggiornato l'albero di copertura e, quindi, la corrispondente base. Dato il grafo in Figura 1.52a è riportato il grafo originario con un flusso ammissibile e la base corrispondente è rappresentata dall'albero di copertura riportato in Figura 1.52b a cui si aggiunge l'arco  $(3, 5)$  che viola le condizioni di ottimalità. Siccome l'arco  $(3, 5)$  ha un flusso  $x_{35}$  pari alla capacità (i.e.,  $x_{35} = u_{35} = 3$ ) per cui il ciclo ha un orientamento contrario all'arco  $(3, 5)$ . Il blocking arc che viene eliminato dall'albero di copertura è l'arco  $(2, 5)$  che ha un flusso residuo di 1 unità. Aumentando il flusso, l'arco  $(2, 5)$  raggiunge la capacità ed esce dalla base, mentre i rimanenti archi del ciclo vedono il flusso aumentato di 1 unità se hanno lo stesso orientamento del ciclo (i.e., in questo caso il solo arco  $(1, 2)$ ) e diminuito di 1 unità se hanno orientamento contrario rispetto al ciclo (i.e., l'arco  $(1, 3)$  quello appena entrato  $(3, 5)$ ). In Figura 1.52c è riportato il nuovo albero di copertura a cui si vuole aggiungere il nuovo arco  $(4, 6)$  che viola le condizioni di ottimalità. Anche in questo caso il ciclo ha un orientamento opposto a quello dell'arco entrante e il blocking arc scelto è  $(3, 5)$  e il flusso viene aumentato di 1 unità in tutti gli archi orientati come il ciclo (i.e.,  $(1, 3)$ ,  $(3, 5)$  e  $(5, 6)$ ) e diminuito di 1 unità per quelli con orientamento opposto rispetto al ciclo (i.e.,  $(1, 2)$ ,  $(2, 4)$  e  $(4, 6)$ ).

Ogni volta che l'albero di copertura e, quindi, la corrispondente base sono modificati è necessario aggiornare i potenziali utilizzando la seguente procedura:

```
procedure update-potentials( $k,l,p,q$ )
begin
```

Eliminando  $(p, q)$  da  $T$  si ottiene  $T_1$ , che contiene la radice, e  $T_2$ ;

**if**  $q \in T_2$  **then**

$y = q$ ;

**else**

$y = p$ ;

**if**  $k \in T_1$  **then**

$\gamma = -c_{kl}^\pi$ ;

**else**

$\gamma = c_{kl}^\pi$ ;

$\pi_y = \pi_y + \gamma$ ;

$z = \text{thred}(y)$ ;

**while**  $\text{depth}(z) > \text{depth}(y)$  **do**

**begin**

$\pi_z = \pi_z + \gamma$ ;

$z = \text{thred}(z)$ ;

**end**

**end**

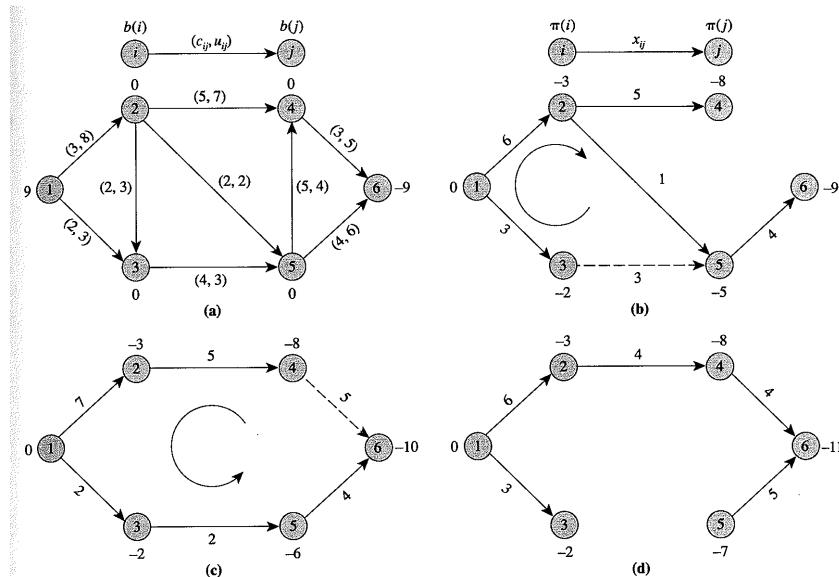


Figure 11.11 Numerical example for the network simplex algorithm.

Figura 1.52: Esempio di determinazione del ciclo e aggiornamento dell'albero di copertura (Fonte: Orlin, Ahuja, Magnanti, "Network Flows: Theory, Algorithms, and Applications", Pearson)

La procedura *update-potentials* deve aggiornare i potenziali per garantirsi che  $c_{ij}^\pi = c_{ij} - \pi_i + \pi_j = 0$  per tutti gli archi dell'albero di copertura. Quando dall'albero di copertura  $T$  viene eliminato il blocking arc  $(p, q)$  si ottengono due sotto-alberi  $T_1$ , che contiene la radice, e  $T_2$ . Quando viene aggiunto l'arco entrante  $(k, l)$  per ripristinare l'albero di copertura i potenziali di  $T_1$  non hanno bisogno di essere aggiornati, ma solo quelli di  $T_2$  partendo dal vertice estremo dell'arco  $(p, q)$  che sta in  $T_2$ .

### 1.6.3 Spanning Tree Fortemente Ammissibili

Finora si è ipotizzato che la struttura spanning tree fosse non degenere (i.e., contiene archi con flusso uguale a 0 oppure pari alla capacità dell'arco), ma studi computazionali hanno dimostrato che mediamente il 90% delle operazioni di pivot sono degeneri.

La presenza di pivot degeneri potrebbe impedire la convergenza alla soluzione ottima (*degenerazione ciclante*). Per evitare la degenerazione ciclante è sufficiente che la struttura spanning tree sia “*fortemente ammissibile*”. Una struttura spanning tree è fortemente ammissibile se ogni arco  $(i, j) \in T$  è diretto:

- verso la radice se  $x_{ij} = 0$ ;
- in direzione opposta rispetto la radice se  $x_{ij} = u_{ij}$ .

Equivalentemente, diremo che una struttura spanning tree è fortemente ammissibile se possiamo sempre inviare una quantità positiva di flusso da un qualsiasi nodo alla radice attraverso i soli archi appartenenti allo spanning tree senza violare i vincoli di capacità.

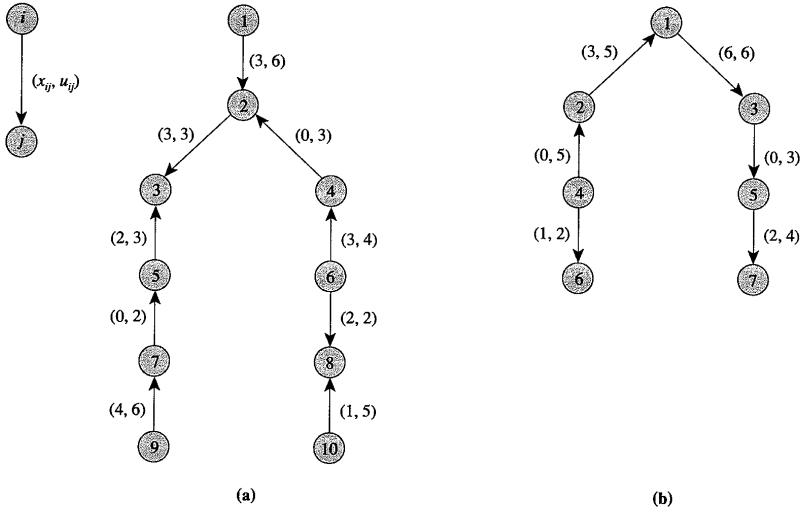
In Figura 1.53 è riportato un esempio di un albero di copertura fortemente ammissibile e uno non fortemente ammissibile.

Per garantirsi che un albero di copertura fortemente ammissibile lo rimanga anche dopo l'aggiornamento è necessario stabilire una *regola di selezione dell'arco uscente*.

Dato l'arco entrante  $(k, l)$  e il corrispondente pivot cycle  $W$ , definiamo “*apice*”  $w$  il vertice in  $W$  che è predecessore sia di  $k$  che di  $l$ . La regola di selezione dell'arco è la seguente: l'arco uscente è l'ultimo blocking arc incontrato attraversando il pivot cycle  $W$  partendo dall'apice  $w$  e seguendo il suo orientamento.

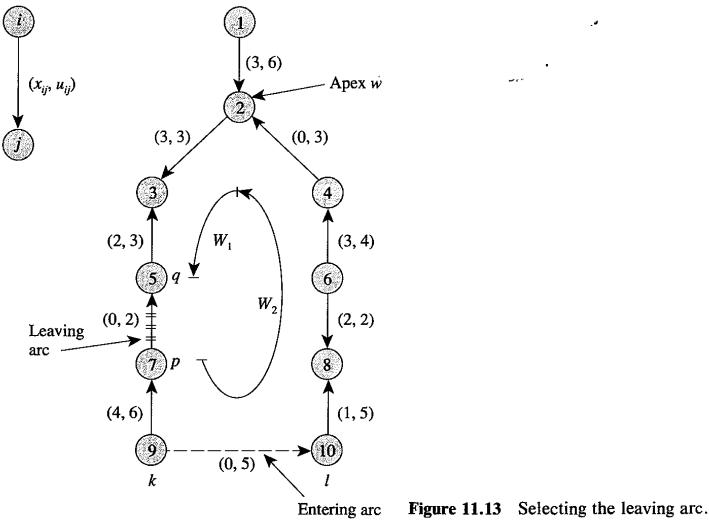
Se lo spanning tree iniziale è fortemente ammissibile, applicando questa semplice regola si ha la garanzia di mantenere lo spanning tree fortemente ammissibile e di evitare la degenerazione ciclante.

In Figura 1.54 è riportato un esempio di applicazione della regola di selezione dell'arco uscente. Nell'esempio i blocking arc sono  $(2, 3)$  e  $(7, 5)$ , ma solo l'arco  $(7, 5)$  rispetta la regola che permette di ottenere un nuovo albero di copertura fortemente ammissibile.



**Figure 11.12** Feasible spanning trees: (a) strongly feasible; (b) nonstrongly feasible.

Figura 1.53: Esempio di alberi di copertura fortemente ammissibili e non fortemente ammissibili (Fonte: Orlin, Ahuja, Magnanti, "Network Flows: Theory, Algorithms, and Applications", Pearson)



**Figure 11.13** Selecting the leaving arc.